WiDS 5.0 Project Report

Monte Carlo Methods: From Geometry to Reinforcement Learning

Weeks 1-4 Summary

Atharva Bhutada
Roll No: 24B0996

January 30, 2026

Abstract

This report documents my journey through the WiDS 5.0 Monte Carlo Project, covering Weeks 1-4. Starting from Python fundamentals, I progressed through Monte Carlo simulations for numerical integration to implementing a Reinforcement Learning agent that learns optimal Blackjack strategy purely from experience. The project culminated in an AI agent that discovered the mathematically optimal "Basic Strategy" after playing hundreds of thousands of games, starting with zero knowledge of the game.

Contents

1 Introduction

1.1 Project Overview

The WiDS Monte Carlo Project is a structured 5-week curriculum designed to teach Monte Carlo methods and their applications in Reinforcement Learning. This report covers the first four weeks:

- **Week 1:** Python, NumPy, and building a Blackjack game engine
- **Week 2:** Monte Carlo estimation for π, e, and high-dimensional integration
- **Week 3:** Reinforcement Learning theory (MDPs, Bellman equations)
- **Week 4:** Implementing Monte Carlo Prediction and Control for Blackjack

1.2 Goals Achieved

- Built a complete Blackjack game engine in Python with proper OOP design
- Mastered NumPy vectorization achieving significant speedups over traditional loops
- Estimated mathematical constants using Monte Carlo simulation
- Understood the theoretical foundations of Reinforcement Learning
- Implemented an RL agent that learned optimal Blackjack strategy from scratch

2 Week 1: Python Fundamentals and Game Development

2.1 Learning Objectives

Master Python programming, NumPy operations, and object-oriented design by building a functional Blackjack game.

2.2 Key Concepts Learned

- Python basics: data structures, control flow, functions
- Object-Oriented Programming: classes, methods, encapsulation
- NumPy fundamentals: arrays, indexing, slicing, vectorization
- Clean code principles: modularity, documentation, testing

2.3 Implementation: Blackjack Game Engine

I designed the game with three main classes:

1. **Card:** Represents individual playing cards with suit and rank
2. **Deck:** Manages the 52-card deck with shuffle and deal operations
3. **BlackjackGame:** Contains game logic with a `step()` function

The `step()` function was designed for RL compatibility:

```
def step(action):
    # action: 0 = stick, 1 = hit
    # returns: (state, reward, done)
    #   state: (player_sum, dealer_card, usable_ace)
    #   reward: +1 (win), -1 (loss), 0 (draw)
    #   done: True if episode finished
```

Key Insight

Designing the game with RL in mind from Week 1 made Week 4 implementation straightforward. The `step()` function became the foundation for training the agent.

2.4 Challenges

- **Ace handling:** Implemented logic to track "usable aces" that count as 11 without busting
- **State representation:** Chose tuple format for compatibility with RL algorithms

3 Week 2: Monte Carlo Estimation and Vectorization

3.1 Learning Objectives

Use random sampling to solve mathematical problems and eliminate all for-loops through NumPy vectorization.

3.2 Core Concept: Monte Carlo Integration

Monte Carlo methods estimate quantities by random sampling. For area estimation:

$$\text{Area} \approx \frac{\text{Points Inside Shape}}{\text{Total Points}} \times \text{Bounding Box Area} \tag{1}$$

3.3 Task 1: Estimating π

Method: Generate random points in a unit square and count how many fall inside a quarter circle.

$$\pi \approx 4 \times \frac{\text{Points in quarter circle}}{\text{Total points}} \tag{2}$$

Results: As the number of samples increased, the estimation converged closer to the true value of π. With a large number of samples, the error became negligibly small, demonstrating the effectiveness of Monte Carlo methods.

3.4 Task 2: Estimating Euler's Number

Estimated e using the integral: $e - 1 = \int_0^1 e^x dx$

By sampling random points and calculating the average value of e^x, I successfully estimated Euler's number with high accuracy.

3.5 The Power of Vectorization

Comparison of loop-based vs. vectorized code:

```
# Slow (for-loop): Several seconds for large samples
for i in range(n):
    x = random.uniform(0, 1)
    if x**2 + y**2 <= 1:
        count += 1

# Fast (vectorized): Fraction of a second for same samples
x = np.random.uniform(0, 1, n)
y = np.random.uniform(0, 1, n)
count = np.sum(x**2 + y**2 <= 1)
```

Result: Vectorized code provided dramatic speedups, often 50-100 times faster than loop-based approaches.

Key Insight

Monte Carlo error decreases as $\frac{1}{\sqrt{N}}$. To halve the error, you need 4 times more samples. This $\sqrt{N}$ convergence is fundamental to all Monte Carlo methods.

3.6 Task 3: Monte Carlo vs. Riemann Sums

I compared Monte Carlo integration with traditional Riemann sum approaches for calculating integrals. While Riemann sums can be more accurate in low dimensions with fewer evaluations, Monte Carlo methods become increasingly advantageous as dimensionality increases.

3.7 High-Dimensional Integration

Explored volume estimation of high-dimensional hyperspheres. Monte Carlo remained practical while grid-based methods became computationally infeasible. In high dimensions, grid methods require exponentially many points, whereas Monte Carlo maintains constant computational efficiency regardless of dimension.

4 Week 3: Reinforcement Learning Foundations

4.1 Learning Objectives

Understand Markov Decision Processes, value functions, and Bellman equations—the theoretical foundation of RL.

4.2 The RL Framework

Reinforcement Learning involves an agent learning to make decisions through interaction:

- **Agent:** The learner (our Blackjack player)
- **Environment:** The world (the casino/game)

- **State (s):** Current situation (cards dealt)
- **Action (a):** Agent's choice (hit/stick)
- **Reward (r):** Feedback (+1 win, -1 loss, 0 draw)
- **Policy (π):** Strategy mapping states to actions

4.3 Markov Decision Processes

An MDP is defined by: (S, A, P, R, γ)

- S: Set of states
- A: Set of actions
- $P(s'|s, a)$: Transition probability
- $R(s, a)$: Reward function
- γ: Discount factor for future rewards

Markov Property: The future depends only on the present state, not the history.

$$P(s_{t+1}|s_t, s_{t-1}, \ldots, s_0) = P(s_{t+1}|s_t) \tag{3}$$

4.4 Value Functions

State-Value Function $V^\pi(s)$: Expected total reward from state s following policy π

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_t \mid S_0 = s \right] \tag{4}$$

Action-Value Function $Q^\pi(s, a)$: Expected total reward from state s, taking action a, then following π

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_t \mid S_0 = s, A_0 = a \right] \tag{5}$$

4.5 Bellman Equations

The fundamental recursive relationship in RL:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s, a)[R(s, a) + \gamma V^\pi(s')] \tag{6}$$

Interpretation: Value of current state = immediate reward + discounted value of next state
Bellman Optimality Equation:

$$V^*(s) = \max_a \sum_{s'} P(s'|s, a)[R(s, a) + \gamma V^*(s')] \tag{7}$$

Key Insight

The Bellman equation is the cornerstone of RL. It transforms the problem of finding optimal policies into solving a system of equations. All RL algorithms are essentially different ways of solving or approximating these equations.

4.6 Key Insights from Theory

- Value functions guide optimal behavior without explicit rules
- The discount factor γ balances immediate vs. future rewards
- Optimal policies exist and can be derived from value functions
- RL learns from experience, not from labeled examples

5 Week 4: Monte Carlo Prediction and Control

5.1 Learning Objectives

Implement Monte Carlo methods to evaluate policies (Prediction) and learn optimal policies (Control) for Blackjack.

5.2 Why Monte Carlo Methods?

Traditional RL methods (dynamic programming) require knowing $P(s'|s, a)$—the environment's transition probabilities. Monte Carlo methods learn from **experience** without needing this model.

Core Idea: Play many episodes, observe returns, and average them.

5.3 Monte Carlo Prediction

Goal: Estimate $V^\pi(s)$ for a fixed policy.

Simple Policy Tested: "Stick on 20-21, otherwise hit"

Algorithm:

1. Play episode following policy π
2. For each state visited, record the return G (total reward from that point)
3. Average all returns from each state: $V(s) = \text{mean}(G_1, G_2, \ldots, G_n)$

Results: The value function estimates showed that states with high player sums (near 21) had positive expected values, while low sums had negative values. This aligned with intuition—being close to 21 is advantageous in Blackjack.

5.4 Monte Carlo Control

Goal: Find the optimal policy π^* without knowing game dynamics.

The Exploration-Exploitation Dilemma:

- **Exploitation:** Use best known actions to maximize reward
- **Exploration:** Try new actions to discover better strategies

Solution: ϵ-Greedy Policy

- With probability ϵ: choose random action (explore)
- With probability $1 - \epsilon$: choose best action (exploit)

5.5 Implementation Details

State Representation:

$$s = (\text{player\_sum}, \text{dealer\_card}, \text{usable\_ace}) \tag{8}$$

Action Space:

- Action 0: Stick (stop taking cards)
- Action 1: Hit (take another card)

Q-Value Update (Incremental Mean):

$$Q(s,a) \leftarrow Q(s,a) + \alpha[G_t - Q(s,a)] \tag{9}$$

where α is the learning rate and G_t is the observed return.

Epsilon Decay:

$$\epsilon_{t+1} = \max(\epsilon_{\min}, \epsilon_t \times \text{decay\_rate}) \tag{10}$$

Start with high exploration, gradually reduce to favor exploitation.

5.6 Training Results

The agent was trained over hundreds of thousands of episodes, starting with random play and gradually improving its strategy.

Observations:

- Initial performance showed frequent losses due to random exploration
- Performance improved steadily as Q-values converged
- Eventually stabilized at near-optimal play
- Final strategy closely matched professional "Basic Strategy"

Learning Curve: The rolling average reward improved from highly negative (losing most games) to near break-even, demonstrating successful learning. Convergence occurred after exploring the state space sufficiently.

5.7 Learned Optimal Strategy

The agent discovered the "Basic Strategy" used by professional players:

Key Patterns:

- Always stick on hard 17 or higher
- Always hit on 11 or lower (can't bust)
- Hit on 12-16 when dealer shows strong cards (7-10)
- Stick on 12-16 when dealer shows weak cards (2-6, dealer likely to bust)
- Different strategy with usable ace (soft hands are more flexible)

Player Sum	Dealer Card A	2	3	4	5	6	7	8	9	10
21	S	S	S	S	S	S	S	S	S	S
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	H	H	S	S	S	S	H	H	H	H
15	H	H	S	S	S	S	H	H	H	H
14	H	H	S	S	S	S	H	H	H	H
13	H	H	S	S	S	S	H	H	H	H
12	H	H	H	S	S	S	H	H	H	H

Table 1: Learned strategy without usable ace (H=Hit, S=Stick)

Key Insight

The agent learned this strategy with **zero prior knowledge** of Blackjack rules. It discovered optimal play purely through winning and losing games. This demonstrates the power of RL: complex optimal behavior emerging from simple reward signals.

5.8 Conceptual Questions

5.8.1 Infinite Deck Assumption

Question: Why does card counting break the Markov property?

Answer: With finite decks, past cards affect future probabilities. If many high-value cards have been dealt, drawing another becomes less likely. The state representing only current cards has different implications depending on the history. To enable card counting, we'd need to augment the state with information about cards already seen:

$$s = (\text{player\_sum}, \text{dealer\_card}, \text{usable\_ace}, \text{card\_count}) \tag{11}$$

Trade-off: This significantly increases the state space, requiring many more episodes to achieve convergence as each state-action pair needs sufficient visits for accurate value estimation.

5.8.2 First-Visit vs. Every-Visit MC

Given an episode where state A appears multiple times with different subsequent rewards:

- **First-Visit MC:** Only count the first occurrence of state A in each episode
- **Every-Visit MC:** Count all occurrences of state A in the episode

Both methods converge to the true value function as the number of episodes increases. First-visit is more commonly used due to simpler theoretical analysis, while every-visit can provide more data points per episode.

5.9 Challenges and Solutions

5.9.1 Convergence Speed

Challenge: Initial learning was slow with many states rarely visited.

Solution: Balanced exploration through epsilon-greedy policy with gradual decay, ensuring sufficient early exploration while eventually settling on optimal actions.

5.9.2 Reward Variance

Challenge: Blackjack has high variance—good decisions can still lose individual games.

Solution: Used rolling averages for tracking progress and ran training for sufficient episodes to average out the inherent randomness.

5.9.3 State Coverage

Challenge: Some states occur infrequently during gameplay.

Solution: Maintained high initial exploration rate to ensure broad state space coverage before transitioning to exploitation-focused behavior.

6 Conclusion

6.1 Project Success

This project successfully took me from Python fundamentals to implementing a working RL agent in four weeks. The structured progression—from tools to simulations to theory to application—built both practical skills and theoretical understanding.

The key achievements include:

- Mastering essential Python and NumPy programming skills
- Understanding and applying Monte Carlo methods for both integration and learning
- Developing strong theoretical foundations in Reinforcement Learning
- Successfully implementing algorithms that learn from experience
- Building an agent that discovered optimal strategies independently

6.2 Applications Beyond Blackjack

The techniques learned apply broadly across many domains:

- **Robotics:** Learning control policies from sensor data
- **Finance:** Portfolio optimization and algorithmic trading
- **Game AI:** Creating intelligent agents for complex games
- **Healthcare:** Optimizing treatment plans and clinical decisions
- **Operations:** Resource allocation and scheduling problems

6.3 Future Directions

Week 5 (Optional): Importance sampling for off-policy learning
Beyond WiDS:

- Deep Q-Networks (DQN) using neural networks for large state spaces
- Policy gradient methods for continuous action spaces
- Temporal difference learning methods (SARSA, Q-learning)
- Multi-agent reinforcement learning systems
- Model-based RL approaches

6.4 Final Reflection

This journey demonstrated that complex intelligent systems are built from simple, well-understood principles. Starting with basic Python programming and progressing through mathematical foundations to practical implementation, each week built upon the previous one.

The most profound insight was watching an agent with no prior knowledge discover optimal strategies through pure experience. This exemplifies the core principle of reinforcement learning: intelligence can emerge from the simple interaction of trying actions, observing outcomes, and updating beliefs accordingly.

The project reinforced the importance of:

- Strong theoretical foundations guiding practical implementation
- Patient, systematic debugging of complex systems
- Balancing exploration and exploitation in learning
- Vectorization and efficient coding practices
- Understanding the mathematical principles underlying algorithms

"Learning happens through experience, both for humans and machines."