# A Deep Dive into REINFORCE The Foundational Monte Carlo Policy Gradient Algorithm

**An Explanatory Guide by Taha Majlesi**

## 1 REINFORCE: A Foundational Policy Gradient Algorithm

The **Policy Gradient Theorem** provides us with the theoretical blueprint for optimizing a policy directly. It gives us a formula for the gradient of the policy's expected reward. The **REINFORCE** algorithm, introduced by Ronald Williams, stands as the most direct and foundational implementation of this theorem. It is a **Monte Carlo** policy gradient method, meaning it learns from complete episodes of experience. This section breaks down its mechanism, its implementation, and its fundamental strengths and weaknesses, which motivate the development of more advanced algorithms.

### 1.1 A Monte Carlo Approach to Policy Gradients

REINFORCE's identity as a **Monte Carlo (MC) method** is its most defining characteristic. In the context of Reinforcement Learning, MC methods learn from **complete trajectories**. The agent runs its policy from a start state until it reaches a terminal state (completes an "episode"), and only then does it update its parameters. The return for each step is calculated from the sequence of rewards obtained during that specific episode.

The Policy Gradient Theorem states that the gradient of the objective function $J(\theta)$ is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[ \left( \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) r(\tau) \right]$$

Since we cannot compute the true expectation (which would require averaging over all possible trajectories), we **estimate** it by sampling. The REINFORCE algorithm approximates this expectation by generating a batch of $N$ trajectories and averaging the results:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left( \sum_{t=0}^{T_i-1} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left( \sum_{t'=0}^{T_i-1} r(s_{i,t'}, a_{i,t'}) \right) \tag{1}$$

**Deconstructing the Gradient Formula (1)**

Let's break down this crucial formula piece by piece:

- $\frac{1}{N}\sum_{i=1}^{N}$: This is the Monte Carlo approximation. We average the gradient estimates from a batch of $N$ different episodes (trajectories) to get a more stable estimate of the true gradient. In the simplest case, we update after every single episode, so $N = 1$.
- $\sum_{t=0}^{T_i-1}$: This is a sum over all the time steps within a single trajectory, $i$. $T_i$ is the length of that specific trajectory.
- $\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})$: This is the **score function**. It's a vector that points in the direction in our parameter space ($\theta$) that would most increase the probability of selecting action $a_{i,t}$ when in state $s_{i,t}$. We calculate this for every action that was actually taken in the episode.
- $\left(\sum_{t'=0}^{T_i-1} r(s_{i,t'}, a_{i,t'})\right)$: This is the **total reward** for the entire trajectory, often denoted $r(\tau_i)$ or $G_i$. This term is a scalar value that acts as a weight. It tells us "how good" the entire trajectory was.

**The Intuition:** The algorithm multiplies the "direction to increase action probability" (the score) by "how good the overall outcome was" (the total reward). If a trajectory resulted in a high total reward, the update pushes the parameters $\theta$ to make all actions taken during that trajectory more likely in the future. If the total reward was low or negative, it pushes to make them less likely.

## 1.2 The REINFORCE Algorithm: Detailed Pseudocode and Explanation

The REINFORCE algorithm cycles through three core phases: **1. Generate Data**, **2. Calculate Returns**, and **3. Update Policy**. The pseudocode below details this loop.

**Algorithm 1** REINFORCE (Monte Carlo Policy Gradient)

```
1: Initialize policy parameters $\theta$ randomly.
2: Set learning rate $\alpha$.
3: for episode $i = 1, 2, \ldots, M$ do
4:                                   ▷ Phase 1: Generate an episode using the current policy
5:     Generate a trajectory $\tau_i = (s_0, a_0, r_1, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, r_T)$ by following $\pi_\theta(a|s)$.
6:                                   ▷ Phase 2: Calculate the return(s) for the episode
7:     // Common Variant: Use return-from-time-t for causality (see explanation below)
8:     Initialize an empty list for updates.
9:     for $t = 0, 1, \ldots, T-1$ do
10:        Calculate the return from time $t$: $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$. ▷ $\gamma$ is the discount factor
11:    end for
12:                                  ▷ Phase 3: Update the policy parameters
13:    Initialize total gradient update $\Delta\theta \leftarrow 0$.
14:    for $t = 0, 1, \ldots, T-1$ do
15:            ▷ Accumulate the gradient for each step, scaled by its corresponding future return
16:        $\Delta\theta \leftarrow \Delta\theta + G_t \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)$
17:    end for
18:                                  ▷ Apply the accumulated gradient using gradient ascent
19:    $\theta \leftarrow \theta + \alpha \cdot \Delta\theta$
20: end for
```

**A Crucial Refinement: Causality and Return-from-Time-t**

The initial Policy Gradient Theorem uses the **total trajectory reward** $r(\tau)$ as the weighting factor for every single action taken in that trajectory. However, this is not optimal. Think about it: an action taken at time $t$ can only influence rewards received *after* time $t$. It cannot possibly affect rewards that have already been collected $(r_1, \ldots, r_t)$.
A simple yet powerful improvement is to weight the score function $\nabla_\theta \log \pi_\theta(a_t|s_t)$ not by the total reward, but by the **return from time t onwards**, defined as:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$$

This is often called the **reward-to-go**. This adjustment respects the principle of **causality**.
**Why is this better?** This change doesn't alter the expected value of the gradient (it remains unbiased), but it can significantly **reduce the variance** of the gradient estimate. By assigning credit for an action only based on its future consequences, the learning signal becomes much more precise and less noisy, which typically leads to faster and more stable convergence. The pseudocode above uses this superior, more common variant.

## 1.3 Analysis of Strengths and Weaknesses

REINFORCE is a vital conceptual stepping stone. Understanding its properties is essential for appreciating why more complex policy gradient methods (like Actor-Critic) were developed.

### 1.3.1 Strengths

- **Simplicity and Generality:** The algorithm is easy to implement and understand. It can be applied to any problem where the policy can be differentiated, making it a ”one-size-fits-all” starting point that works for environments with discrete, continuous, or high-dimensional action spaces.
- **Unbiased Gradient Estimate:** This is a strong theoretical guarantee. Because the Monte Carlo return $G_t$ is an unbiased sample of the true expected future reward, the overall policy gradient estimate is also unbiased. This means that, on average, the updates are pointing in the correct direction of improvement. The algorithm might take noisy steps, but it does not have a systematic error steering it in the wrong direction.

### 1.3.2 Weaknesses

- **High Variance (The Achilles' Heel):** This is by far the most significant drawback of REINFORCE and all simple Monte Carlo methods. The return $G_t$ is a sum of many random variables (rewards and state transitions), which makes its value highly variable from one episode to the next.

**Analogy: The Noisy Signal**

Imagine an agent learning to play a video game. It might execute a series of 99 brilliant moves but then make one unlucky mistake at the very end, resulting in a low total score. The basic REINFORCE algorithm would receive this low total score and use it to *discourage* all 100 actions taken, including the 99 brilliant ones. Conversely, an agent could play poorly but get a lucky break at the end, leading to a high score; REINFORCE would then reinforce all the poor actions. The learning signal is "drowned out" by the random noise of the outcomes. This high variance leads to unstable learning and requires an enormous number of episodes to average out the noise.

- **Sample Inefficiency:** REINFORCE is profoundly inefficient with data for two main reasons linked to its on-policy, Monte Carlo nature:

  1. **On-Policy Learning:** The samples used for updates *must* be generated by the current policy $\pi_\theta$. As soon as we update the parameters $\theta$ to a new $\theta'$, all the trajectories we collected with the old policy are rendered invalid and must be discarded. This means we are constantly throwing data away.
  2. **Monte Carlo Updates:** The algorithm must wait until an entire episode is finished before it can perform a single update. It cannot learn incrementally during the episode, which slows down the learning process, especially for long-running tasks.

  This stands in stark contrast to off-policy, value-based methods like DQN, which can store millions of transitions in a replay buffer and reuse them efficiently, making them far more sample-efficient.

In conclusion, REINFORCE is a cornerstone algorithm that beautifully illustrates the core concepts of policy gradients. However, its practical utility is limited by high variance and poor data efficiency. It serves as the essential foundation upon which more advanced, variance-reduced, and efficient algorithms like Actor-Critic methods are built.