ChatGPT

REINFORCE Algorithm: From Basics to Implementation

Reinforcement learning (RL) problems involve an agent interacting with an environment over a sequence of **states** ($s_1, s_2, \ldots$), taking **actions** ($a_1, a_2, \ldots$) and receiving **rewards** ($r_1, r_2, \ldots$). The goal of the agent is to learn a **policy** $\pi_\theta(a \mid s)$ parameterised by θ that maximises the expected sum of rewards. In policy-gradient methods like **REINFORCE** we do not derive a value function directly; instead we optimise the parameters of a stochastic policy using gradient ascent on the expected return.

Trajectories and returns

An entire episode of interaction is called a **trajectory** $\tau = (s_1, a_1, s_2, a_2, \ldots, s_T, a_T)$. The total return of a trajectory, for finite-horizon tasks, is often written as

$$R(\tau) = r_1 + r_2 + \ldots + r_T.$$

The agent's objective can be written as an expectation over all possible trajectories that the current policy could produce:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\, R(\tau) \,],$$

where the expectation is taken with respect to the distribution over trajectories induced by the policy. The Spinning Up notes clarify that we wish to maximise **expected return** and that the policy gradient provides a way to differentiate this objective [1].

To make this more concrete, denote by $p_\theta(\tau)$ the probability of a trajectory under the policy. It can be factored as

$$p_\theta(\tau) = p(s_1) \cdot \prod_{t=1}^{T} \pi_\theta(a_t \mid s_t) \cdot p(s_{t+1} \mid s_t, a_t),$$

where $p(s_1)$ is the initial state distribution and $p(s_{t+1} \mid s_t, a_t)$ are the environment's transition dynamics [2]. **Importantly, the dynamics and the initial-state distribution do not depend on the policy parameters** [3], which means that the only terms containing θ in $p_\theta(\tau)$ come from the policy itself.

Because the return $R(\tau)$ is a function of the state and reward sequence, it is fixed once a trajectory is given and does not explicitly depend on the policy. We therefore write the objective as the integral over all trajectories:

$$J(\theta) = \int p_\theta(\tau)\, R(\tau)\, d\tau.$$

This formulation expresses the expected return as an integral over the trajectory distribution [4]. It is the starting point for deriving the policy gradient.

Deriving the REINFORCE gradient

To optimise J(θ) we need its gradient with respect to θ. The derivation proceeds in a few algebraic steps, each of which has a clear intuitive interpretation.

1. Differentiate under the integral sign

The gradient of the objective can be expressed as

$$\nabla_\theta J(\theta) = \nabla_\theta \int p_\theta(\tau) R(\tau) d\tau = \int \nabla_\theta p_\theta(\tau) R(\tau) d\tau.$$

In words: we bring the gradient inside the integral since the return does not depend on θ. The integrand now involves the gradient of the trajectory probability.

2. Apply the log-derivative trick

Directly differentiating $p_\theta(\tau)$ is inconvenient because it is a product of many probabilities. The **log-derivative trick** (or likelihood ratio trick) rewrites this derivative as

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \cdot \nabla_\theta \log p_\theta(\tau).$$

Substituting into the integral yields

$$\nabla_\theta J(\theta) = \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) R(\tau) d\tau = E_{\tau \sim p_\theta}[\nabla_\theta \log p_\theta(\tau) \cdot R(\tau)].$$

In words: we have replaced the gradient of a probability with the probability times the gradient of its log. This is useful because we can estimate expectations of $\nabla_\theta \log p_\theta(\tau)$ with samples.

3. Remove environment terms

The log probability of a trajectory can be expanded using the factorisation of $p_\theta(\tau)$:

$$\log p_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} [\log p(s_{t+1}|s_t,a_t) + \log \pi_\theta(a_t|s_t)].$$

Because the initial state distribution and transition dynamics do **not** depend on θ, their gradients are zero [3]. Consequently

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t).$$

In words: only the policy terms contribute to the gradient; the environment's randomness does not. This step is crucial because it removes unknown dynamics from the gradient expression [4].

4. Final policy gradient expression

Plugging the simplified gradient back into the expectation gives the **REINFORCE gradient**:

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot R(\tau) \right].$$

This result shows that the gradient of expected return can be estimated by summing, over each time step, the gradient of the log policy multiplied by the **same total return** $R(\tau)$. Spinning Up's derivation arrives at the same formula [5]. It has two important properties:

- **Unbiased estimate.** When we sample trajectories under π_θ and compute the sum above, the expectation equals the true gradient. Thus we can perform stochastic gradient ascent to improve the policy.
- **High variance.** Using the full return $R(\tau)$ for every time step often leads to noisy gradient estimates, especially for long trajectories. Later sections address variance reduction techniques such as reward-to-go and baselines.

5. Sample-based estimation

In practice we do not compute the expectation exactly; instead we sample N trajectories $\{\tau_i\}$ by running the policy in the environment and compute the empirical mean

$$\hat{g} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \cdot R(\tau_i),$$

where T_i is the length of trajectory i. This quantity is an unbiased estimator of the true gradient [6].

Reward-to-go and baseline as variance reduction

The simple REINFORCE gradient uses the same total return $R(\tau)$ at every time step, giving each action credit or blame for the entire episode. Two standard modifications reduce variance while keeping the gradient unbiased.

Reward-to-go (step-wise returns)

Instead of using the full return, we can replace $R(\tau)$ by the **reward-to-go** $G_t = r_t + r_{t+1} + \dots + r_T$, the sum of rewards from the current time step onward. Because future rewards after time t do not depend on earlier actions, using G_t still gives an unbiased gradient estimator. It reduces variance by giving each action credit only for what happens after it.

Adding a baseline

The **expected grad–log–prob (EGLP) lemma** states that for any function $b(s_t)$ depending only on the state, the expectation of $\nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot b(s_t)$ under the policy is zero [7]. Therefore we can subtract $b(s_t)$ from the return without changing the expected value of the gradient:

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (G_t - b(s_t)) \right].$$

Any such b is called a **baseline** [8]. The most common choice is the **on-policy state value function** V^π(s_t), which measures the expected return from s_t under the current policy. Spinning Up notes that using this baseline reduces the variance of the gradient estimator and leads to faster, more stable learning [9]. In practice V^π(s_t) is approximated by a neural network V_φ(s_t); its parameters are updated by minimising a mean squared error between predicted values and empirical returns [10]. When we subtract V(s_t) from the return, the term G_t – V(s_t) is called the **advantage** A(s_t,a_t), indicating how much better or worse the action was compared with the average outcome in that state.

Pseudocode for REINFORCE without baseline

The following pseudocode outlines the REINFORCE algorithm in its simplest form (no reward-to-go or baseline), and maps each operation back to the mathematical concepts above. It assumes access to an environment that can be reset and stepped, and a policy network that maps states to a probability distribution over actions.

1. **Initialise policy parameters** θ (for example, neural network weights).
   Corresponds to starting with some policy π_θ we wish to improve.
2. **For each update** (iteration of gradient ascent):
3. **Collect trajectories:** run the current policy in the environment for K episodes. For each episode record (s_t,a_t,r_t, log π_θ(a_t ∣ s_t)).
   This corresponds to sampling trajectories from p_θ(τ) to approximate the expectation E_{τ~π_θ}[...].
4. **Compute returns:** for each trajectory, compute the total return R(τ) = r_1 + … + r_T.
   This implements the return R(τ) used in the gradient formula.
5. **Compute the policy loss:** for each trajectory, sum the log probabilities of actions and multiply by the return: `Loss = -∑_t log π_θ(a_t∣s_t) ·R(τ)`.
   The negative sign reflects performing gradient descent on the loss to achieve gradient ascent on J(θ); this term matches the gradient expression $\Sigma\ \nabla_\theta \log \pi_\theta(a_t \mid s_t)\ R(\tau)$.
6. **Backpropagate and update:** compute the gradient of the loss with respect to θ and take a small step in the direction of the gradient.
   This approximates updating $\theta \leftarrow \theta + \alpha\ \nabla_\theta J(\theta)$.

This version of REINFORCE is easy to implement but suffers from high variance because it assigns credit for the entire episode's return to every action.

Pseudocode for REINFORCE with reward-to-go and baseline

The variance of policy gradient estimates can be significantly reduced by using reward-to-go and a baseline. The following pseudocode incorporates these modifications:

1. **Initialise policy parameters** θ and value function parameters φ. The value function approximator V_φ(s) will be used as the baseline.
2. **For each update:**
3. **Collect K trajectories** by running π_θ. For each time step store (s_t,a_t,r_t, log π_θ(a_t ∣ s_t)).
4. **Compute reward-to-go:** for each trajectory and each time step compute G_t = r_t + r_{t+1} + … + r_T.
   This replaces the full return with step-wise returns, reducing variance.

5. **Evaluate baseline:** compute V_φ(s_t) for each recorded state.
   These are the baseline estimates b(s_t) used to subtract from the returns.
6. **Compute advantages:** A_t = G_t – V_φ(s_t). Optionally normalise advantages over the batch.
   The advantage is the return-to-go minus the baseline; it measures how much better an action was compared with the expected outcome.
7. **Policy loss:** sum `-log π_θ(a_t|s_t) · A_t` over all time steps and episodes. Add an entropy bonus term (weighted by β) if desired to encourage exploration.
8. **Update policy parameters:** backpropagate the policy loss to update θ.
9. **Value loss:** compute the mean squared error between G_t and V_φ(s_t) and update φ by gradient descent.
   This trains the value network to approximate V^π(s), the average return from state s under the current policy [10] *.*

This enhanced version implements the unbiased gradient estimator with baseline [11] and reduces variance. In practice it is referred to as the **vanilla policy gradient** or **REINFORCE with baseline** and forms the foundation for more advanced actor–critic methods.

Summary

The REINFORCE algorithm is a simple Monte-Carlo policy gradient method: it estimates the gradient of the expected return by sampling trajectories and weighting the log-probabilities of actions by the observed returns. The derivation leverages the log-derivative trick and the independence of environment dynamics [3], leading to an elegant expression for the policy gradient [4]. Reward-to-go and baselines are optional modifications that retain unbiasedness while reducing variance. Using an on-policy value function as a baseline is particularly effective [9] and is standard in modern policy gradient implementations.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] Part 3: Intro to Policy Optimization — Spinning Up documentation
https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html