Fundamentals of Reinforcement Learning

Luka Meladze

Abstract

This paper provides an overview of key introductory concepts in Reinforcement Learning (RL), suitable for beginners. It defines the Markov Decision Process (MDP) and its components, elaborates on the distinctions between RL and Supervised Learning (including Imitation Learning) and describes a real-world application using RL terminology. Finally, the paper provides a mathematical formulation of the RL objective function under both finite and infinite horizon settings and provides a detailed explanation of derivation of policy gradients, including methods for reducing variance and a simplified algorithmic outline for practical implementation.

1 What is a Markov Decision Process (MDP)?

A **Markov Decision Process (MDP)** is a framework used to describe sequential decision-making problems, where a decision-maker (agent) interacts with an environment over time. At each time step t, the agent observes an observation of the state of the world s_t (possibly partial o_t), takes an action a_t, and then receives a reward r_t while making a transition to the new state s_{t+1}. This cycle continues, and the agent's ultimate goal is to find a **policy** (a strategy for choosing actions) that maximises the total expected reward.

The key idea and assumption of MDP is rooted in the **Markov property**, which states that the next state depends **only** on the current state and action, not on the history of previous states or actions. That doesn't mean that the future is perfectly predictable; there might still be randomness, but knowing the past doesn't help at all to resolve that randomness.

An MDP is defined by a **5-tuple**: $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$.

- **States ($\mathcal{S}$)**: The set of all possible states the environment of an agent can be in. Each state $s \in \mathcal{S}$ is a **complete** description of the environment, while an observation o is a partial description of a state. A state s must satisfy the **Markov Property**: If the current state s_t is given, the future state s_{t+1} and future rewards are independent of past states $(s_1, \ldots, s_{t-1})$ and actions $(a_1, \ldots, a_{t-1})$. Mathematically, $P(s_{t+1}|s_t, a_t, \ldots, s_1, a_1) = P(s_{t+1}|s_t, a_t)$.
- **Actions ($\mathcal{A}$)**: The set of all possible & valid actions (moves or decisions) that the agent can make in any state is often called the action space. Some environments have discrete action spaces, while others have continuous action spaces where actions are real-valued vectors.
- **Transition Probability Operator/Function(T)**: It is a **tensor** which specifies the probability of transitioning from the current state s to the next state s' after taking action a. $T_{i,j,k} = p(s_{t+1} = i|s_t = j, a_t = k)$ also denoted as $P(s'|s, a)$. The environment's response to an action can be **deterministic** (always leads to the same next state) or **stochastic** (leads to different next states).
- **Reward Function (R)**: The reward function R is very important in reinforcement learning, which is a mapping from $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. It depends on the current state of the world and the action just taken. It quantifies the desirability of an action or state, typically denoted as $R(s)$ or $R(s, a)$. The agent's ultimate goal in RL is to maximise ***not*** *the immediate* but the *cumulative* reward over time (over a trajectory).
- **Discount Factor (γ)**: For tasks that continue indefinitely, the cumulative sum of rewards could be infinite. We need ways to ensure the objective remains finite.

Discount Factor is a value between 0 and 1 ($\gamma \in [0, 1)$ typically 0.9 to 0.99) which weights rewards received sooner more heavily than rewards received later to guarantee that the infinite sum converges to a finite value.

2 How Reinforcement Learning Differs from Supervised Learning (Imitation Learning)

Reinforcement Learning (RL) and Supervised Learning (SL), including Imitation Learning (IL), learn in fundamentally different ways and are distinguishable by their data sources, goals, inherent assumptions, and feedback mechanisms. The Supervised Learning works fine under the assumption that each data example is Independent and Identically Distributed. **Imitation Learning (Behavioral Cloning)** is a form of Supervised Learning applied to sequential decision-making. It trains a policy π_θ to mimic an expert's actions. However, IL inherits SL's limitations, particularly the distributional shift problem, meaning it cannot guarantee optimal performance outside the expert's observed trajectory space and cannot discover behaviors superior to the expert's. RL directly addresses these limitations by learning through interaction and optimizing for long-term outcomes.

- In supervised learning, including imitation learning, the agent learns from a fixed dataset (**Independent and Identically Distributed data**) of input-output pairs $\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^{N}$. The model learns from these static examples and receives direct, immediate, and explicit "ground truth" labels (correct answers) for each input.

  In contrast, reinforcement learning does not assume access to an expert or labelled actions. Instead, the agent sequentially interacts with the environment directly. It observes the state, selects actions, and receives rewards based on those actions. The agent's goal is not to imitate an expert but to maximise the total cumulative reward it receives over time.
- **Learning Objective**: In supervised learning, the goal is to generalise patterns from given data and copy the behaviour or labels. i.e. learn mapping $f_\theta : \mathcal{X} \to \mathcal{Y}$ that accurately predicts outputs for unseen inputs and minimises prediction error.
  In Reinforcement Learning, the goal is to learn a **policy** $\pi_\theta : \mathcal{S} \to \mathcal{A}$ that maximises the **total future reward** over a sequence of interactions. It's about figuring out what the agent should do to reach a goal, potentially discovering solutions better than any human expert could show (like AlphaGo's surprising "Move 37"). It maximises $J(\theta) = \mathbb{E}_{\tau \sim P_\pi(\tau)} \left[\sum_{t=1}^{T} \gamma^t R(s_t, a_t) \right]$.

3 Real-World Case: Autonomous Drone trained by RL

Let's describe a real-world case of an autonomous drone that navigate through a forest or different obstacles to reach a destination while avoiding any barriers. The AI-powered drone's goal is to learn a policy that maximises its total reward, meaning it learns to reach its destination safely while minimising crashes and being efficient.

- **Agent**: The drone's control system (the AI brain that makes decisions about flying).
- **The state** s_t at time t includes all relevant information about the environment, such as the drone's position, speed, orientation, battery level, and a camera image (the pixels showing trees, clear paths, etc.) of its surroundings.
- **Actions ($\mathcal{A}$)**: These are the commands the drone can give to its motors to change its flight. The action a_t could be continuous control commands like pitch and roll or **discrete** commands like "turn left," "turn right," or "go straight."
- **Reward Function (R)**: How the drone gets feedback on its performance to learn and improve:
  - **Positive Reward**: For moving closer to the destination, or for each second it stays flying without crashing.

  - **Negative Reward (Penalty)**:
    - A large negative reward for colliding with a tree or any other obstacle.
    - A small negative reward for using a lot of battery power.

- **Transition Probability Function ($P(s_{t+1}|s_t, a_t)$)**: This describes how the drone moves through the forest based on the current state when an action is applied, including sensor noise and wind disturbances. This is typically **stochastic** due to external factors like wind.
- **Policy (π_θ)**: This is the learned strategy or "brain" of the drone. It tells the drone what action to take in the current state (e.g., "When it sees a particular state/image of obstacles ahead, tilt hard right!").
- **Trajectory (τ)**: A sequence of states and actions along the way. $\tau = (s_1, a_1, s_2, a_2 \ldots)$.
- **Objective**: The objective is to learn policy parameters θ^* that maximises the **expected cumulative reward** over extended periods and to train the drone to learn this policy so that it can reach the destination safely and efficiently.

4 The Objective of Reinforcement Learning in Math

The fundamental objective in Reinforcement Learning is to find an **optimal policy (π^*)** that is defined by parameters (θ^*) and maximises the **expected cumulative reward** (also known as "return") over time. This expectation accounts for the stochasticity of both the environment and the agent's policy.

4.1 Finite Horizon (Total Reward)

For tasks that have a fixed, finite number of time steps T the objective is to maximise the expected sum of rewards the agent collects until the end of completing the task.

The **return** for a single trajectory $\tau = (s_1, a_1, \ldots, s_T, a_T, s_{T+1})$ is:

$$G_0 = \sum_{t=1}^{T} R(s_t, a_t)$$

The **objective function**, which we aim to maximise by optimising the policy's parameters θ, is the **expected return**:

$$J(\theta) = \mathbb{E}_{\tau \sim P_\pi(\tau)} \left[\sum_{t=1}^{T} R(s_t, a_t) \right]$$

where:

- $P_\pi(\tau)$: The probability of a specific trajectory τ occurring under policy π_θ:

$$P_\pi(\tau) = P(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) P(s_{t+1}|s_t, a_t)$$

- $R(s_t, a_t)$: The immediate reward received at time t in state s_t taking an action a_t.

An equivalent formulation for finding the optimal policy parameters θ^* in the finite horizon case is to maximise the sum of expected rewards at each time step. Here, $p_\theta(s_t, a_t)$ represents the probability of visiting state s_t and taking action a_t at time t under policy π_θ.

$$\theta^* = \arg\max_\theta \sum_{t=1}^{T} \mathbb{E}_{(s_t, a_t) \sim p_\theta(s_t, a_t)} [R(s_t, a_t)]$$

4.2 Infinite Horizon (Average Reward)

For tasks where the agent operates continuously without a natural ending point and without a discount factor, the objective is to maximise the long-term **average** reward per time step.

The **average reward objective** is defined as:

$$J_{avg}(\theta) = \lim_{T\to\infty} \frac{1}{T} \mathbb{E}_{\tau\sim P_\pi(\tau)} \left[\sum_{t=1}^{T} R(s_t, a_t) \right]$$

When the system converges to a stationary distribution $p_\theta(s, a)$, this objective simplifies to the expected immediate reward under that distribution:

$$J_{avg}(\theta) = \mathbb{E}_{(s,a)\sim p_\theta(s,a)}[R(s, a)]$$

The optimal policy parameters θ^* for the average reward case are therefore found by maximising this expected immediate reward under the stationary distribution

$$\theta^* = \arg\max_\theta \mathbb{E}_{(s,a)\sim p_\theta(s,a)}[r(\mathbf{s}, \mathbf{a})]$$

4.3 Infinite Horizon (Discounted Total Reward)

For continuous tasks without a natural ending point, we also use a **discounted sum of rewards** to ensure the total return remains finite and converges.

The **discounted return** for a single trajectory $\tau = (s_1, a_1, s_2, a_2, \ldots)$ is:

$$G_0 = \sum_{t=1}^{\infty} \gamma^t R(s_t, a_t)$$

The **objective function** to maximise is the **expected discounted return**:

$$J(\theta) = \mathbb{E}_{\tau\sim P_\pi(\tau)} \left[\sum_{t=1}^{\infty} \gamma^t R(s_t, a_t) \right]$$

where:

- γ: The **discount factor**, a value in $[0, 1)$ that exponentially weighs future rewards less than immediate ones.

The ultimate goal in Reinforcement Learning is to find the policy parameters θ^* that maximises this objective function:

$$\theta^* = \arg\max_\theta J(\theta)$$

This means we are searching for the best possible strategy that, on average, leads to the highest accumulation of rewards over the long run, considering both the agent's actions and the environment's stochastic (random) nature.

5 Policy Gradients

5.1 Deriving the Basic Policy Gradient

To achieve the main goal in Reinforcement Learning, i.e. to find the best settings (parameters θ) for a policy π_θ so that the average total reward $J(\theta) = \mathbb{E}_{\tau\sim P_\pi(\tau)} \left[\sum_{t=1}^{T} R(s_t, a_t) \right]$ is as high as possible, policy gradient methods are used that directly optimize a parameterized policy. The underlying principle of policy gradients involves iteratively adjusting policy parameters to increase the probabilities of actions that lead to higher returns while simultaneously decreasing the probabilities of actions that result in lower returns. This iterative refinement

process inherently embodies a "trial and error" (Levine, 2021) learning paradigm, where the agent continuously adapts its policy based on observed outcomes. The optimisation of $J(\theta)$ is achieved through an iterative process known as **gradient ascent**, that calculates "steepest way up" given by the **gradient** of the objective function, $\nabla_\theta J(\theta)$. Once this gradient is known, policy parameters θ are adjusted in that direction:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

131 Here, α (alpha) is a small step size, called the **learning rate**.

The challenge is to calculate $\nabla_\theta J(\theta)$ because the trajectory τ depends on the rules of the environment, such as the unknown transition probabilities $P(s_{t+1}|s_t, a_t)$ or initial state $P(s_1)$. We can only interact with the world to collect samples. Mathematically, $J(\theta)$ can be expressed as an average over all possible trajectories τ:

$$J(\theta) = E_{\tau \sim p_\theta(\tau)}[r(\tau)] = \int p_\theta(\tau) r(\tau) d\tau$$

Next, taking the gradient with respect to θ, since the gradient operation is linear, it can be moved inside the integral:

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

Now, a very useful trick called the "convenient identity" (log-derivative trick) is employed. This trick states that for any probability distribution $P(\tau)$:

$$\nabla_\theta P(\tau) = P(\tau) \nabla_\theta \log P(\tau)$$

Applying this convenient identity is useful because it essentially converts a gradient of a probability into a probability times the gradient of its logarithm. The equation becomes:

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

Now, let's determine $\nabla_\theta \log p_\theta(\tau)$. The probability of a trajectory is defined as: $p_\theta(\tau) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$ Taking the logarithm of both sides (which turns products into sums) gives the following equation:

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

A crucial simplification occurs when taking the gradient ∇_θ with respect to θ, since the terms $\log p(s_1)$ (initial state distribution) and $\log p(s_{t+1}|s_t, a_t)$ (environment dynamics & transition probabilities) vanish, as they are independent of the policy parameters θ. This is because the environment's rules do not depend on the policy parameters θ. So, only the policy term remains:

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

This means the gradient computation now only requires differentiating the known, parameterized policy $pi_\theta(a_t|s_t)$, not the unknown environment. Substituting this back into the policy gradient equation yields the fundamental **basic policy gradient formula** (the REINFORCE Algorithm). This seminal algorithm was formally introduced by (Williams, 1992), providing a foundational method for estimating the policy gradient through a sample mean derived from collected trajectories.

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

In practice, since this expectation cannot be calculated perfectly, it is estimated **(unbiased)** by running the policy π_θ in the environment N times and collecting many trajectories. Then, for each sampled trajectory the calculated terms are averaged to get an estimate of gradient. So, the **MONTE Carlo approximation formula for policy gradient (MIT-6.7920, 2024)**:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \right) \right]$$

5.2 Adding a Baseline for the Policy Gradient and Why It is Needed

The basic policy gradient, while mathematically correct, has a significant practical issue: it can be very **noisy** and have **high variance** (Afachao, 2023). This means that if a small number of trajectories are collected, the estimate of the gradient can jump around a lot, making it hard for policy parameters to learn & improve consistently. Additionally, if all the rewards are positive, even for "bad" trajectories, the basic policy gradient method will try to increase the probability of all sampled actions, simply because their reward is positive. This goes against the intuition that the goal is to increase probabilities for "better than average" actions and decrease them for "worse than average" ones.

To avoid this problem and make learning more stable and efficient, two key tricks, reward-to-go and baselines (Afachao, 2023) help reduce the **high variance** of policy gradients:

5.2.1 Exploiting Causality - "Reward-to-Go" ($\hat{Q}_t$)

An action taken at time t cannot affect rewards that were received in the past (before time t) (Schulman, 2016). This is the principle of **causality**. So, when evaluating how good an action a_t was, only the rewards that came **after** that action, from time t until the end of the trajectory, need to be considered. This sum of future rewards is called the **reward-to-go**:

$$\hat{Q}_t = \sum_{t'=t}^{T} r(s_{t'}, a_{t'})$$

Using this reward-to-go term instead of the total reward from the entire trajectory reduces noise by discarding irrelevant past rewards. The policy gradient now looks like:

$$\nabla_\theta J(\theta) \approx E_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \hat{Q}_t \right]$$

5.2.2 Baselines $b(s_t)$

To further reduce noise and make the learning more intuitive, a value called a **baseline**, $b(s_t)$, is subtracted from the reward-to-go term. This baseline typically represents the average or expected reward-to-go from a given state s_t. The use of such a baseline is formally justified by the EGLP (Expected Gradient of Log Policy) Lemma, which states that adding a baseline $b(s_t)$ that is independent of the action a_t does not change the expectation of the policy gradient, while significantly reducing its variance (OpenAI, Achiam, 2018).

By using a baseline, the actual reward received ($\hat{Q}_t$) is essentially compared to what was expected from that state ($b(s_t)$).

- If $(\hat{Q}_t - b(s_t))$ is positive, it means the action taken led to a **better-than-expected** outcome. So, the policy will be adjusted to make that action more likely.
- If $(\hat{Q}_t - b(s_t))$ is negative, it means the action led to a **worse-than-expected** outcome. So, the policy will be adjusted to make that action less likely.

This "centering" provides a more stable learning signal, leading to more reliable policy improvement. So, the formula (MIT-6.7920, 2024) with reward-to-go and a baseline is:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \left(\hat{Q}_{i,t} - b \right)$$

5.3 On-policy algorithm for the Policy Gradient

This algorithm for the policy gradient outlines the steps for how an agent learns using policy gradients with a basic baseline. It focuses on the "trial and error" learning process that policy gradients formalise, allowing an agent to learn complex behaviours by continually refining its strategy based on the outcomes it experiences.

Algorithm: Policy Gradient Method (Simplified version of REINFORCE algorithm (Achiam, 2018))

1. **Input**:
   - Initial policy parameters θ_0 (the starting settings for the "brain")
   - Learning rate α (how big of a step to take when adjusting settings)
2. **For** $k = 0, 1, 2, \ldots$ (repeat for many learning cycles) **do**:
   (a) **Collect Trajectories (Run the Policy)**:
   - Use the current policy $\pi(\theta_k)$ to interact with the environment.
   - Collect a set of trajectories $\mathcal{D}_k = \{\tau_i\}_{i=1}^{N}$ (many paths the robot takes). Each τ_i includes a sequence of states, actions, and rewards: $(s_{i,1}, a_{i,1}, r_{i,1}, s_{i,2}, a_{i,2}, r_{i,2}, \ldots, s_{i,T}, a_{i,T}, r_{i,T})$.

   (b) **Compute Rewards-to-Go ($\hat{Q}_t$)**:
   - For each time step t in every collected trajectory $\tau_i \in \mathcal{D}_k$, calculate the reward-to-go $\hat{Q}_{i,t}$:

$$\hat{Q}_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$$

   (c) **Compute Simple Baseline (b)**:
   - Calculate the average of all $\hat{Q}_{i,t}$ values across all time steps in all trajectories in the current batch $\mathcal{D}_k$. This average will be our baseline b:

$$b = \frac{1}{\sum_{\tau \in \mathcal{D}_k} |\tau|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=1}^{|\tau|} \hat{Q}_{i,t}$$

   (where $|\tau|$ is the length of trajectory τ)

   (d) **Estimate Policy Gradient ($\hat{g}_k$)**:
   - Calculate the estimated policy gradient $\hat{g}_k$ using the collected data, reward-to-go values, and the baseline:

$$\hat{g}_k = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\hat{Q}_{i,t} - b)$$

   (e) **Update Policy Parameters**:
   - Adjust the policy parameters θ by taking a step in the direction of the estimated gradient:

$$\theta_{k+1} = \theta_k + \alpha \hat{g}_k$$

References

Joshua Achiam. Vanilla policy gradient — spinning up documentation, 2018. URL https://spinningup.openai.com/en/latest/algorithms/vpg.html#pseudocode.

Kevin Afachao. Deep dive into reinforcement learning: Policy gradient algorithms. *Medium*, 2023. URL https://medium.com/@thekevin.afachao/deep-dive-into-reinforcement-learning-policy-gradient-algorithms-bbeabe2fc989. Accessed: 2025-05-23.

Sergey Levine. Policy gradients (uc berkeley cs 285 lecture notes), 2021. URL https://rail.eecs.berkeley.edu/deeprlcourse/.

MIT-6.7920. Policy gradient. https://web.mit.edu/6.7920/www/lectures/L13&14-2024fa-PolicyGradient.pdf, 2024. Lecture notes for MIT 6.7920: Reinforcement Learning, Fall 2024.

OpenAI, Achiam. Spinning up in deep rl. https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html, 2018. Accessed: 2025-05-23.

John Schulman. *Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs*. PhD thesis, University of California, Berkeley, 2016. Specifically Chapter 2, as referenced in OpenAI Spinning Up documentation.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992. doi: 10.1007/BF00992696. URL https://link.springer.com/article/10.1007/BF00992696.