POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Numerical Analysis for Machine Learning project

A taste of the cherry on top

A gentle introduction to Reinforcemnt Learning and Policy Gradient methods

Author: Paolo Ginefra

Professor: Edie Miglio

Academic year: 2024/25

Premise

This Theory project is heavily based on the wonderful book "Reinforcement Learning: An Introduction"[5] by Richard Sutton and Andrew G. Barto. For a more comprehensive understanding of the Reinforcement Learning field, this book is a must read.

1. Introduction

Learning through interaction with our environment is fundamental to how we acquire knowledge. From an infant exploring the world through play to mastering complex skills like driving, we constantly learn by observing how our actions affect our surroundings. This natural learning process, where we discover what works through trial and error, forms the foundation of reinforcement learning (RL).
Reinforcement learning is the computational approach to learning what actions to take in order to maximize rewards. Unlike supervised learning, where we learn from labeled examples provided by a teacher, RL agents must discover optimal strategies through their own experience. There's no explicit instruction manual, only the feedback of rewards and penalties that guide the learning process. What makes RL particularly challenging and fascinating are two key characteristics: **trial-and-error search** and **delayed rewards**. An agent must not only discover which actions yield immediate benefits but also understand how current decisions influence future outcomes. This creates the fundamental **exploration-exploitation** dilemma: should the agent stick with known good actions (exploit) or try new ones that might be even better (explore)?
RL differs fundamentally from other machine learning paradigms. While supervised learning relies on correct examples and unsupervised learning seeks hidden patterns in data, reinforcement learning focuses on goal-directed behavior in uncertain environments. It's simultaneously a problem formulation, a class of solution methods, and a research field, making it a unique and powerful framework for artificial intelligence.
This report explores the world of reinforcement learning, with particular focus on policy gradient methods, which are at the center of modern RL techniques.

1.1. Examples of Reinforcement Learning

To understand RL's versatility, consider these diverse scenarios where intelligent agents learn through interaction:

- **Strategic Decision-Making**: A chess grandmaster combines deep planning with intuitive position evaluation, refining their

strategic intuition through countless games of experience.
- **Industrial Control**: An adaptive controller optimizes a petroleum refinery's yield-cost-quality balance in real-time, learning to deviate from preset parameters when beneficial.
- **Biological Learning**: A newborn gazelle calf transforms from helpless to running 20 mph in thirty minutes: nature's own reinforcement learning in action.
- **Autonomous Systems**: A cleaning robot decides whether to explore new rooms or return to its charging station based on battery level and past recharging experiences.
- **Everyday Intelligence**: Even making breakfast reveals complex sequential decision-making, from selecting cereals to coordinating hand movements, each action serving multiple nested goals.

These examples share crucial features: **active agents interacting with uncertain environments**, **actions that affect future states**, **delayed consequences requiring foresight**, and **explicit goals with measurable progress**. Most importantly, all agents improve their performance through experience.

2. Recipe for an RL Problem

Every reinforcement learning system consists of key ingredients that work together to enable intelligent behavior. Beyond the fundamental agent-environment interaction, we can identify four main components that form the "recipe" for any RL problem.

2.1. Policy: The Agent's Behavioral Blueprint

A **policy** defines how an agent behaves: it's a mapping from environmental states to actions. Think of it as the agent's decision-making rulebook, corresponding to stimulus-response associations in psychology. Policies can range from simple lookup tables to complex computational processes involving extensive search. Crucially, policies may be *stochastic*, specifying probabilities over actions rather than deterministic choices. The policy is the heart of any RL agent, as it alone determines behavior.

2.2. Reward Signal: The Immediate Compass

The **reward signal** defines the goal by providing immediate feedback, a single number indicating how well the agent performed at each step. The agent's sole objective is maximizing cumulative reward over time. Rewards are the primary mechanism for policy improvement: low rewards trigger policy adjustments for future similar situations. Like biological pleasure and pain, rewards provide immediate and defining feedback about environmental events.

2.3. Value Function: The Long-term Oracle

While rewards indicate immediate desirability, **value functions** capture long-term prospects. The value of a state represents the total expected future reward starting from that state. This distinction is crucial: a state might yield low immediate reward but high value if it leads to rewarding future states, or vice versa. Values enable farsighted decision-making: we choose actions leading to high-value states, not necessarily high-reward states, because this maximizes long-term reward. Estimating values from experience sequences is arguably the most important challenge in RL.

2.4. Model: The Environment Simulator (Optional)

Some RL systems include an **environment model**: a learned representation that predicts how the environment behaves. Given a state and action, the model forecasts the next state and reward. Models enable *planning*: deliberating over possible future scenarios before taking action. This creates two paradigms: **model-based methods** that learn and use environment models, and **model-free methods** that rely purely on trial-and-error learning. Modern reinforcement learning spans this entire spectrum, from reactive trial-and-error learning to sophisticated deliberative planning, providing a rich toolkit for intelligent sequential decision-making.

3. Why Can't You Just Simulate Evolution?

A natural question arises: if we want agents to learn optimal behavior, why not simply simulate biological evolution? After all, evolution has produced remarkably skilled organisms without any individual learning during their lifetimes (even though an individual might be considered a learner per se, no individual is capable of adapting its genetic makeup during its lifetime). **Evolutionary approaches** do exist for reinforcement learning problems. These methods, including genetic algorithms, genetic programming, and simulated annealing, work by evaluating multiple static policies over extended periods, then carrying forward the highest-performing policies (plus random variations) to the next generation. This process repeats until good policies emerge. However, evolutionary methods have significant limitations for RL problems. Most critically, **they ignore the rich structure of sequential decision-making**. They don't exploit the fact that policies map states to actions, nor do they track which specific states an agent visits or which actions prove effective in particular situations. This makes them much less sample-efficient than methods that learn from individual behavioral interactions. Consider the difference: traditional RL methods can immediately adjust behavior based on a single bad outcome in a specific state. Evolutionary methods, by contrast, must wait for entire policy lifetimes to complete before making any adjustments, discarding valuable information about *when* and *where* things went wrong. That said, evolutionary approaches aren't obsolete. In [2], OpenAI has shown that evolution strategies can rival standard RL techniques on modern benchmarks, particularly when traditional RL faces challenges like sparse rewards or non-differentiable objectives. While less data-efficient than RL, evolution strategies offer many benefits, including easier parallelization and robustness to noisy gradients. The key insight is that while evolution and learning share many features and can work together, evolution alone typically cannot match the sample efficiency of methods that learn from individual experiences. This is why modern RL focuses on algorithms that continuously adapt during environment interaction rather than waiting for generational turnover.

4. Mathematical Abstraction Time, the language of MDPs

Now we transition from intuitive understanding to mathematical rigor. The elegant complexity of reinforcement learning requires a formal framework that can capture sequential decision-making under uncertainty while remaining mathematically tractable. Enter **finite Markov Decision Processes (MDPs)**, the mathematical backbone of modern reinforcement learning. MDPs formalize the sequential decision-making problem by adding an *associative* dimension through the concept of **states**: the description of the current situation the agent is in (its state) is enough to identify which action to make and how to assign a reward. This captures the essence of sequential decision-making where actions influence not only immediate rewards but also future states and, through them, all subsequent rewards. MDPs sacrifice some real-world complexity to gain theoretical precision, but this compromise has proven extraordinarily fruitful for developing practical algorithms. In the following subsections, we'll build this mathematical foundation step by step, showing how this abstract framework translates into concrete algorithmic insights.

4.1. Agent-Environment Interface

MDPs provide a clean framing of goal-directed learning through interaction. The **agent** is the learner and decision-maker, while the **environment** comprises everything outside the agent. These two entities interact continually: the agent selects actions, and the environment responds with new situations and rewards. Formally, agent and environment interact at discrete time steps $t = 0, 1, 2, 3, \ldots$. At each time step t, the agent receives the environment's state $S_t \in \mathcal{S}$ and selects an action $A_t \in \mathcal{A}(s)$. One time step later, the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subseteq R$ and finds itself in a new state S_{t+1}. This creates a trajectory:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \qquad (1)$$

In finite MDPs, the sets $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ all have finite elements. The dynamics are captured by transition probabilities that depend only on the preceding state and action:

$$p(s', r|s, a) \doteq \Pr(S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a) \tag{2}$$

The dynamics function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ completely characterizes the environment, satisfying:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{3}$$

This dependence only on the immediately preceding state and action defines the **Markov property**: the future depends on the present, not the entire history. From the four-argument function p, we can derive useful quantities like state-transition probabilities:

$$\begin{aligned} p(s'|s, a) \doteq \\ &\doteq \Pr(S_t = s' | S_{t-1} = s, A_{t-1} = a) \\ &= \sum_{r \in \mathcal{R}} p(s', r|s, a) \end{aligned} \tag{4}$$

And expected rewards for state-action pairs:

$$\begin{aligned} r(s, a) &\doteq E[R_t | S_{t-1} = s, A_{t-1} = a] \\ &= \sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} r \cdot p(s', r|s, a) \end{aligned} \tag{5}$$

The MDP framework is remarkably flexible. Time steps need not be fixed intervals as they can represent arbitrary decision stages. Actions range from low-level motor controls to high-level strategic choices. States can be direct sensor readings or abstract symbolic descriptions. Importantly, the **agent-environment boundary** represents the limit of the agent's absolute control, not its knowledge. Everything the agent cannot arbitrarily change, including its motors, sensors, and reward computation, belongs to the environment. This boundary can be drawn at different levels for different purposes, enabling hierarchical decomposition of complex systems.

4.2. Goals and Rewards

In reinforcement learning, the agent's purpose is formalized through a special signal called the **reward**, passing from environment to agent. At each time step, the reward is simply a number $R_t \in R$. The agent's goal is to maximize not immediate reward, but cumulative reward over time. This leads to the **reward hypothesis**: all goals and purposes can be thought of as maximizing the expected value of the cumulative sum of a scalar reward signal. Using rewards to formalize goals is a distinctive feature of RL that, despite appearing limiting, has proven remarkably flexible. Consider these examples:

- Robot locomotion: Reward proportional to forward motion encourages walking.
- Maze escape: Reward of -1 per time step until escape encourages speed.
- Object collection: Reward of $+1$ per collected item, with negative rewards for collisions.
- Game playing: Reward of $+1$ for winning, -1 for losing, 0 for draws and intermediate positions.

In all cases, the agent learns to maximize reward. If we want specific behavior, we must design rewards so that maximizing them achieves our goals. This makes reward design critical: rewards must truly indicate what we want accomplished. Crucially, **rewards should specify what to achieve, not how to achieve it**. For chess, reward only actual wins, not subgoals like capturing pieces or controlling the center. Otherwise, the agent might achieve subgoals while failing at the true objective (perhaps capturing pieces at the cost of losing the game).

4.3. It's all about Value

The heart of reinforcement learning beats with a simple yet profound question: *how good is it to be here?* Nearly every RL algorithm revolves around estimating **value functions**, the mathematical oracles that predict the long-term desirability of states and actions based on expected future rewards.

A *policy* π formally maps states to action probabilities, where $\pi(a|s)$ represents the likelihood of selecting action a in state s. Policies define how agents behave, but their worth is measured through value functions that capture expected cumulative rewards.

We define two complementary value functions that estimate "how good" different situations are:

- **State-value function**:

  $$v_\pi(s) = E_\pi[G_t | S_t = s]$$

  The expected return starting from state s under policy π

- **Action-value function**:

$$q_\pi(s,a) = E_\pi[G_t | S_t = s, A_t = a]$$

expected return taking action a in state s, then following π

4.4. Learning Values from Experience.

These functions can be estimated from interaction with the environment. Monte Carlo methods maintain running averages of actual returns experienced from each state. As the number of visits approaches infinity, these averages converge to true values. For problems with vast state spaces, agents maintain parameterized value functions with fewer parameters than states, adjusting them to better match observed returns.

4.5. The power of Recursion

Value functions satisfy a fundamental recursive relationship: the *Bellman equation*:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] \tag{6}$$

This elegant equation expresses a consistency condition: the value of state s must equal the expected immediate reward plus the discounted value of successor states.
The Bellman equation reveals why value estimation is RL's central challenge: it connects present decisions to future consequences, enabling agents to make farsighted choices in uncertain environments.
These kind of backup relationships form the core of reinforcement learning methods, transferring value information from successor states back to current states.

4.6. The RL Problem Formulation

Solving a reinforcement learning task means finding a policy that achieves maximum reward over the long run. But what does "optimal" actually mean in mathematical terms?

4.6.1 Optimal Policies and Value Functions

Value functions create a natural ordering over policies: policy π dominates policy π' if $v_\pi(s) \geq v_{\pi'}(s)$ for all states $s \in S$. Remarkably, there always exists at least one policy that dominates all others—an *optimal policy* π^*.
All optimal policies share the same **optimal state-value function**:

$$v^*(s) = \max_\pi v_\pi(s) \tag{7}$$

and the same **optimal action-value function**:

$$q^*(s,a) = \max_\pi q_\pi(s,a) \tag{8}$$

4.6.2 The Bellman Optimality Equation

The optimal value function satisfies a special consistency condition: the *Bellman optimality equation*:

$$v^*(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v^*(s')\right] \tag{9}$$

This equation captures a fundamental insight: the value of a state under optimal behavior equals the expected return from the best possible action. Unlike regular Bellman equations that depend on specific policies, this equation defines optimality itself.

4.6.3 From Optimal Values to Optimal Actions.

Once we have v^*, finding optimal actions becomes trivial through *greedy action selection*. For any state, simply choose the action that maximizes immediate reward plus discounted future value. The beauty of v^* is that greedy choices yield globally optimal behavior because v^* already encodes all future reward considerations.
With q^*, optimal action selection becomes even simpler: just pick $\arg\max_a q^*(s,a)$. The action-value function pre-computes the results of all one-step lookaheads, eliminating the need to know environment dynamics.

4.6.4 The Computational Reality

In principle, we could solve the Bellman optimality equation directly: it's a system of $|S|$ equations in $|S|$ unknowns. In practice, this approach faces three crushing limitations: unknown environment dynamics, computational intractability

(chess has $\sim 10^{47}$ states), and the Markov assumption.
This forces us into the realm of *approximation.* Most RL methods can be understood as approximate solutions to the Bellman optimality equation, using actual experience instead of complete environmental knowledge. Rather than viewing this as a limitation, RL embraces approximation intelligently, focusing computational effort on frequently encountered states while accepting suboptimality in rare situations.

4.6.5 The Complete Formulation

We can now precisely frame the reinforcement learning problem as an optimization problem: *find a policy that maximizes the expected cumulative reward, equivalently expressed as maximizing the value function $v_\pi(s)$ for all states.*

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) \forall s \in S \tag{10}$$

This mathematical formulation transforms the intuitive goal of "learning good behavior" into the concrete objective of value function maximization—the cornerstone of modern reinforcement learning algorithms.

5. The Tale of a Branching Field

Reinforcement learning stands as one of the most expansive and rapidly evolving domains within machine learning, which in turn is a field so vast that it defies simple categorization. Like a mighty river delta, RL has branched into numerous specialized directions, that, despite sharing the same water, each addresses different aspects of learning optimal behavior through interaction. This diversity reflects both the field's youth and its immense potential, as researchers continue to discover new pathways toward intelligent decision-making.
This complexity can be understood through fundamental organizing principles that unite the field: the estimation of value functions, the backing up of values along trajectories, and the iterative improvement of policies through generalized policy iteration (GPI). Yet even within this unified framework, the space of possible methods spans multiple dimensions, creating a vast methodological landscape.
Methods vary along several key spectra: from sample updates (learning from individual experiences) to expected updates (requiring complete distributional knowledge), and from shallow bootstrapping (like one-step TD learning) to deep updates (extending to Monte Carlo's full returns). At the extremes lie dynamic programming with its exhaustive expected updates, Monte Carlo methods with their complete sample trajectories, and temporal difference learning, bridging these approaches.
The field extends across numerous other critical dimensions: the distinction between on-policy and off-policy learning, the choice between episodic and continuing tasks, and the fundamental decision between estimating state values, action values, or afterstate values. The exploration-exploitation dilemma alone has spawned entire research programs, from simple ε-greedy action selection to sophisticated approaches like Upper Confidence Bound methods and optimistic initialization strategies.
Perhaps the most transformative development has been the emergence of function approximation methods. Traditional tabular methods become computationally intractable for problems with large or continuous state spaces, necessitating approximation schemes that generalize learned knowledge across similar states. At the forefront of this revolution stands *Deep Q-Learning* (DQN), which married Q-learning's theoretical foundations with deep neural networks' representational power.
The seminal work published in Nature as "Human-level control through deep reinforcement learning"[1], demonstrated that agents could learn to play Atari games directly from pixel inputs, achieving performance comparable to expert human players. DQN works by approximating the Q-function using a convolutional neural network that takes raw pixels as input and outputs Q-values for each possible action. The key insight was using experience replay (storing and randomly sampling past experiences) and a separate target network to stabilize training, overcoming the instability that plagued earlier attempts to combine neural networks with Q-learning.
This breakthrough validated the potential of deep reinforcement learning and spawned an entire subfield. Beyond DQN, the field continues

expanding into multi-agent learning, hierarchical approaches, inverse reinforcement learning, and meta-learning, each addressing fundamental challenges like sample efficiency, generalization, and real-world deployment.
This vast methodological landscape creates both opportunities and challenges. The diversity provides powerful tools for specific problems, but choosing the right approach requires a deep understanding of both problem characteristics and algorithmic alternatives. For this project, we focus on one particular branch: policy gradient methods, which offer a direct approach to policy optimization that has proven especially powerful for continuous control and complex action spaces.

6. Policy Gradient Methods

After exploring the vast landscape of reinforcement learning methods, we now turn our attention to the star of the show: policy gradient methods. This represents a fundamental departure from the approaches we have encountered so far, marking a conceptual shift that has proven transformative in modern reinforcement learning.
Until now, we have primarily talked about action-value methods: approaches that learn the values of actions and then select actions based on their estimated action values. In these methods, policies exist only as a byproduct of value estimation; without the underlying action-value estimates, there would be no policy at all. The agent's behavior emerges implicitly from the learned values through action selection mechanisms like greedy policies.
Policy gradient methods fundamentally invert this relationship. Instead of learning values to derive policies, these methods learn a parameterized policy directly, one that can select actions without ever consulting a value function. While value functions may still play a supporting role in learning the policy parameters, they are no longer required for action selection itself. This direct approach to policy learning opens new possibilities and addresses limitations that plague value-based methods.
We represent the policy using a parameter vector $\boldsymbol{\theta} \in R^{d'}$, giving us the parameterized policy:

$$\pi(a|s, \boldsymbol{\theta}) = \Pr\{A_t = a | S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\} \quad (11)$$

This notation captures the probability that action a is selected at time t, given that the environment is in state s and the policy parameters are $\boldsymbol{\theta}$. When methods also employ a learned value function, we denote the value function's weight vector as $\mathbf{w} \in R^d$, as in $\hat{v}(s, \mathbf{w})$.
The defining characteristic of policy gradient methods lies in their optimization approach: they learn policy parameters by following the gradient of some scalar performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameters. Since we seek to maximize performance, the parameter updates approximate gradient ascent:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla \widehat{J(\boldsymbol{\theta}_t)} \quad (12)$$

where $\nabla \widehat{J(\boldsymbol{\theta}_t)} \in R^{d'}$ represents a stochastic estimate whose expectation approximates the true gradient of the performance measure.
This general schema encompasses all policy gradient methods, regardless of whether they also learn approximate value functions. When methods do learn both policy and value function approximations, they are often called *actor-critic methods*. In this theatrical metaphor, the "actor" refers to the learned policy that performs actions, while the "critic" refers to the learned value function—typically a state-value function—that evaluates the actor's performance.
The performance measure $J(\boldsymbol{\theta})$ can be defined in different ways depending on the problem setting. For episodic tasks, performance is naturally defined as the value of the start state under the parameterized policy.
This direct parameterization of policies offers several compelling advantages. Unlike value-based methods that may struggle with large or continuous action spaces, policy gradient methods can naturally handle complex action spaces by parameterizing the policy appropriately. They can learn stochastic policies directly, which may be optimal in partially observable environments or when mixed strategies are advantageous. Perhaps most importantly, they provide a principled framework for policy optimization that can incorporate domain knowledge through careful policy parameterization.
As we delve deeper into policy gradient methods, we will explore their theoretical foundations, examine specific algorithms, and understand how

they have revolutionized fields from robotics to game playing.

6.1. Advantages of Policy Gradients

Policy gradient methods offer remarkable flexibility in how policies can be parameterized. The only requirement is that $\pi(a|s, \boldsymbol{\theta})$ remains differentiable with respect to its parameters, that is, $\nabla_{\boldsymbol{\theta}} \pi(a|s, \boldsymbol{\theta})$ exists and is finite for all states, actions, and parameters. To ensure exploration, we typically require that policies never become completely deterministic.
For discrete action spaces, the most common parameterization uses *soft-max in action preferences*. We define numerical preferences $h(s, a, \boldsymbol{\theta}) \in R$ for each state-action pair, then convert these to probabilities via the soft-max distribution:

$$\pi(a|s, \boldsymbol{\theta}) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}} \tag{13}$$

These preferences can be parameterized arbitrarily—from simple linear combinations $h(s, a, \boldsymbol{\theta}) = \boldsymbol{\theta}^T \mathbf{x}(s, a)$ using feature vectors, to complex deep neural networks where $\boldsymbol{\theta}$ represents all connection weights.

6.1.1 Key Advantages

Natural approach to deterministic policies. Unlike ε-greedy action selection, which always maintains some probability of random action, soft-max parameterization can approach deterministic policies naturally. Action preferences aren't constrained to specific values, they're driven to produce optimal stochastic policies. If the optimal policy is deterministic, preferences for optimal actions are driven infinitely higher than suboptimal ones.
Arbitrary action probabilities. Policy methods enable selection of actions with any desired probabilities. In problems requiring mixed strategies, like bluffing in poker with specific probabilities, this capability is essential. Action-value methods struggle to find stochastic optimal policies, while policy methods handle them naturally.
Simpler function approximation. The complexity of policies versus action-value functions varies across problems. When policies are simpler to represent than value functions, policy-based methods typically learn faster and achieve superior performance.
Prior knowledge injection. Policy parameterization provides a natural mechanism for incorporating domain knowledge about desired policy structure: often the most compelling reason for choosing policy-based methods.
These advantages make policy gradient methods particularly powerful for complex action spaces, stochastic strategies, and problems where domain expertise can guide policy structure.

7. The policy Gradient Theorem

With continuous policy parameterization, the action probabilities change smoothly as a function of the learned parameter. Largely because of this, stronger convergence guarantees are available for policy-gradient methods than for action-value methods.
The episodic (constant horizon) and continuing (varying horizon) cases require a different performance measure $J(\theta)$. For simplicity's sake, we will stick with the episodic case for this project; however, the continuing one can be addressed with very similar steps.
For the episodic case, the metric to be maximized is the state value function of the starting state s_0.

$$J(\theta) \doteq v_{\pi_\theta}(s_0) \tag{14}$$

The fundamental challenge in policy optimization lies in performance dependence on both action selections and the distribution of states where those selections occur, both affected by policy parameters. While the effect on actions can be computed straightforwardly from the parameterization, the policy's effect on state distribution depends on unknown environment dynamics. How can we estimate performance gradients when they depend on these unknown state distribution changes?
The answer comes through the **policy gradient theorem**, which provides an analytic expression for the performance gradient that completely sidesteps state distribution derivatives.
The policy gradient theorem establishes:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q^\pi(s, a) \nabla \pi(a|s, \theta) \tag{15}$$

where $\mu(s)$ is the on-policy state distribution under π, and the gradients are column vectors of partial derivatives with respect to θ's components. The proportionality constant equals the average episode length.

7.1. Proof

Here we prove the policy gradient theorem for the episodic case.
With elementary calculus and rearranging of terms, we can prove the policy gradient theorem from first principles. To keep notation simple, we leave it implicit that π is a function of θ, and all gradients are with respect to θ.
First, note that the gradient of the state-value function can be written in terms of the action-value function as:

$$\begin{aligned}&\nabla v^\pi(s) = \\ &= \nabla \left[\sum_a \pi(a|s) q^\pi(s,a)\right] \\ &= \sum_a \left[\nabla \pi(a|s) q^\pi(s,a) + \pi(a|s)\nabla q^\pi(s,a)\right]\end{aligned}$$

Now we expand $\nabla q^\pi(s,a)$ using the Bellman equation:

$$\begin{aligned}\nabla q^\pi(s,a) &= \nabla \left[\sum_{s',r} p(s',r|s,a)[r + \gamma v^\pi(s')]\right] \\ &= \sum_{s'} p(s'|s,a)\nabla v^\pi(s') \\ &\text{(since } p \text{ and } r \text{ don't depend on } \theta) \\ &\text{(for simplicity assume } \gamma = 1)\end{aligned}$$

Substituting back:

$$\nabla v^\pi(s) = \\ \sum_a \left[\nabla \pi(a|s) q^\pi(s,a) + \pi(a|s)\sum_{s'} p(s'|s,a)\nabla v^\pi(s')\right]$$

Through repeated unrolling of this recursive relationship, we obtain:

$$\nabla v^\pi(s) = \\ \sum_{x\in\mathcal{S}}\sum_{k=0}^{\infty} \Pr(s \to x, k, \pi)\sum_a \nabla\pi(a|x) q^\pi(x,a)$$

where $\Pr(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π.

For the performance gradient, we have:

$$\begin{aligned}&\nabla J(\theta) = \nabla v^\pi(s_0) \\ &= \sum_s \sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi)\sum_a \nabla\pi(a|s) q^\pi(s,a) \\ &= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q^\pi(s,a)\end{aligned}$$

where $\eta(s) = \sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi)$ is the unnormalized state visitation frequency.
Since $\mu(s) = \frac{\eta(s)}{\sum_{s'}\eta(s')}$ is the normalized on-policy state distribution, we get:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \nabla\pi(a|s) q^\pi(s,a)$$

Q.E.D.

This remarkable result transforms an intractable optimization problem into a practical algorithm. Rather than requiring knowledge of how policies affect state distributions (typically unknown and environment-dependent) we need only the policy's gradient and action-value estimates. The theorem's beauty lies in its elimination of the most challenging component of policy optimization, providing the theoretical foundation for all modern policy gradient methods.

7.2. REINFORCE: Monte Carlo Policy Gradient

Now we derive our first concrete policy gradient algorithm. The policy gradient theorem provides an exact expression proportional to the gradient; we just need a sampling method whose expectation equals this expression.
Since the theorem's right-hand side involves states weighted by their occurrence frequency under policy π, we can rewrite it as an expectation. Starting from the policy gradient theorem:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q^\pi(s,a)\nabla\pi(a|s,\theta)$$

We transform this into a sample-based estimate by replacing the sum over actions with an expectation under π. Introducing the weighting $\pi(a|S_t,\theta)$ without changing equality:

$$\nabla J(\theta) \propto E_\pi \left[\sum_a \pi(a|S_t,\theta)\frac{q^\pi(S_t,a)\nabla\pi(a|S_t,\theta)}{\pi(a|S_t,\theta)}\right]$$

$$= E_\pi \left[q^\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right]$$

(replacing a by the sample $A_t \sim \pi$)

Since $E_\pi[G_t|S_t, A_t] = q^\pi(S_t, A_t)$, we can replace the action-value with the actual return G_t:

$$\nabla J(\theta) \propto E_\pi \left[G_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right]$$

This yields the **REINFORCE update rule**:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}$$

7.2.1 Intuitive Understanding

The update has elegant intuitive appeal: each increment is proportional to the return G_t times a direction vector $\frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$. This vector points in the parameter direction that most increases the probability of repeating action A_t in state S_t.

The algorithm increases parameters proportional to the return and *inversely* proportional to the action probability. High returns drive stronger updates in favorable directions, while the inverse probability weighting prevents frequently selected actions from dominating updates—ensuring fair credit assignment across all actions.

7.2.2 The Eligibility Vector

Using the identity $\nabla \ln x = \frac{\nabla x}{x}$, we can rewrite the fractional gradient as:

$$\frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} = \nabla \ln \pi(A_t|S_t, \theta)$$

This **eligibility vector** $\nabla \ln \pi(A_t|S_t, \theta)$ is the only place policy parameterization appears in the algorithm, making REINFORCE remarkably general.

7.2.3 Monte Carlo Nature

REINFORCE uses the complete return G_t from time t until episode termination, making it a Monte Carlo algorithm. All updates occur retrospectively after episode completion, ensuring unbiased gradient estimates but potentially high variance.

As a stochastic gradient method, REINFORCE enjoys strong theoretical convergence properties. The expected update direction matches the true performance gradient, guaranteeing improvement for sufficiently small step sizes and convergence to local optima under standard conditions. However, the Monte Carlo nature can produce high variance, leading to slow learning in practice.

7.2.4 Pseudocode

Algorithm 1 REINFORCE: Monte-Carlo Policy-Gradient Control

```
1: **Input:** a differentiable policy parameterization $\pi(a|s, \theta)$
2: **Algorithm parameter:** step size $\alpha > 0$
3: Initialize policy parameter $\theta \in R^{d'}$ (e.g., to $\mathbf{0}$)
4: **repeat**
5:    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$
6:    **for** each step of the episode $t = 0, 1, \ldots, T-1$ **do**
7:       $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
8:       $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \theta)$
9:    **end for**
10: **until** convergence or maximum episodes reached
```

7.3. The Importance of a Baseline

While REINFORCE is theoretically sound, its Monte Carlo nature creates high variance that can severely slow learning. The solution lies in variance reduction through a carefully chosen baseline.

The policy gradient theorem generalizes to include an arbitrary baseline $b(s)$:
$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a [q^\pi(s, a) - b(s)] \nabla \pi(a|s, \theta)$
The baseline can be any function that doesn't depend on actions, because the subtracted term vanishes:

$$\begin{aligned} &\sum_a b(s) \nabla \pi(a|s, \theta) \\ &\qquad = b(s) \nabla \sum_a \pi(a|s, \theta) \\ &\qquad = b(s) \nabla 1 \\ &\qquad = 0 \end{aligned}$$

This gives us the **REINFORCE with baseline** update:

$$\theta_{t+1} = \theta_t + \alpha[G_t - b(S_t)]\frac{\nabla\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

The baseline leaves the expected update unchanged but can dramatically reduce variance. The natural choice is the state-value function $\hat{v}(S_t, w)$: in high-value states we need a high baseline to distinguish good actions from great ones, while low-value states require correspondingly low baselines.

Algorithm 2 REINFORCE with Baseline

```
1: **Input:** differentiable policy parameterization $\pi(a|s,\theta)$
2: **Input:** differentiable state-value function parameterization $\hat{v}(s,w)$
3: **Algorithm parameters:** step sizes $\alpha_\theta > 0$, $\alpha_w > 0$
4: Initialize $\theta \in R^{d'}$ and $w \in R^d$ (e.g., to $\mathbf{0}$)
5: **repeat**
6:     Generate episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\theta)$
7:     **for** each step $t = 0, 1, \ldots, T-1$ **do**
8:         $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
9:         $\delta \leftarrow G - \hat{v}(S_t, w)$
10:        $w \leftarrow w + \alpha_w \gamma^t \delta \nabla \hat{v}(S_t, w)$
11:        $\theta \leftarrow \theta + \alpha_\theta \gamma^t \delta \nabla \ln \pi(A_t|S_t,\theta)$
12:    **end for**
13: **until** convergence
```

This algorithm learns two parameter vectors simultaneously: θ for the policy and w for the baseline. The quantity $\delta = G_t - \hat{v}(S_t, w)$ represents the *advantage*: how much better the actual return was compared to our expectation. Setting step sizes requires care: α_w follows standard value function guidelines, while α_θ depends on reward ranges and policy parameterization.

7.4. The REINFORCE Revolution

Ronald J. Williams' 1992 REINFORCE algorithm [6] fundamentally transformed reinforcement learning by introducing the first practical policy gradient method.
REINFORCE's core insigh, that we can differentiate through stochastic policies to find better action probabilities, laid the theoretical foundation for modern policy optimization. The algorithm's Monte Carlo approach, while suffering from high variance, established the template that would inspire generations of more sophisticated methods.
The algorithm's legacy is most evident in today's dominant RL techniques. Trust Region Policy Optimization (TRPO) [3] and Proximal Policy Optimization (PPO) [4] are direct descendants of REINFORCE, addressing its variance issues while preserving its core policy gradient philosophy. PPO's role in LLM alignment through RLHF demonstrates how Williams' foundational work continues to shape cutting-edge AI systems.
REINFORCE proved that learning policies directly from experience was not only possible but practical, establishing policy gradients as one of RL's three fundamental paradigms alongside value methods and actor-critic approaches.

8. The Cherry on Top

In his influential analogy, Turing Award laureate Yann LeCun described machine intelligence as a cake: "If intelligence is a cake, the bulk of the cake is unsupervised learning, the icing on the cake is supervised learning, and the cherry on the cake is reinforcement learning". While this metaphor positions RL as the smallest component of intelligence, it also highlights its unique and perhaps most intriguing role: the element that transforms competent systems into truly creative ones.
The "cherry" nature of RL becomes evident when we consider its capacity to generate novel solutions that transcend human intuition. Perhaps no example illustrates this better than AlphaGo's infamous Move 37 in its historic match against Lee Sedol. It was a move that no human would've ever made. This move, described by Go professionals as "beautiful" and "creative," exemplified RL's ability to discover strategies that lie beyond the boundaries of human experience and conventional wisdom.
This creative potential of RL stems from its exploration-driven learning paradigm. Unlike supervised learning, which is constrained by the patterns present in training data, the agent's interaction with its environment allows it to uncover non-obvious correlations and develop in-

novative approaches that might never occur to human experts.
However, this creative prowess comes at a significant cost: sample inefficiency. RL algorithms typically require millions of interactions with their environment to achieve superhuman performance, a stark contrast to human learning efficiency. AlphaGo, for instance, trained on millions of self-play games before achieving its breakthrough performance. This computational intensity has historically limited RL's practical applications compared to its supervised counterparts.
The landscape has evolved dramatically with the advent of large language models (LLMs). Reinforcement Learning from Human Feedback (RLHF) has emerged as a critical component in aligning these models with human preferences and values. Recent developments, particularly in models like DeepSeek-R1, demonstrate how RL can enhance reasoning capabilities and improve the quality of generated outputs. In this context, RL serves not merely as the "cherry on top" but as an essential mechanism for fine-tuning and optimizing model behavior post-training.
The integration of RL in the LLM era represents a paradigm shift. Rather than learning from scratch, RL now operates on top of pretrained foundation models, leveraging their existing knowledge while refining their decision-making processes. This approach partially addresses the sample efficiency challenge, as the agent begins with a sophisticated understanding of language and reasoning rather than starting from tabula rasa.
LeCun's analogy, while capturing the relative scale of different learning paradigms, perhaps understates RL's transformative potential. The "cherry" may be small, but it is often what makes the difference between a merely competent system and one capable of genuine innovation and creative problem-solving. As we continue to push the boundaries of artificial intelligence, RL's role as the catalyst for truly novel and unexpected solutions becomes increasingly valuable, even if it remains the most computationally demanding component of the intelligence cake.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015.

[2] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

[3] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[6] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992.