Barcelona School of Economics

Reinforcement Learning Notes

DSM – BSE

Alex Malo, *Spring 2025*

Contents

June 11, 2025

1 Introduction

(a) Motivation for Reinforcement Learning

Traditional ML (e.g., supervised learning) is about mapping inputs to outputs based on labeled examples.

Reinforcement Learning (RL) is different: it's about learning how to act, not just how to predict.

In RL, an agent interacts with an environment and receives rewards based on its actions. The goal is to maximize cumulative reward over time.

Core Motivation: Learning from consequences of actions rather than static labels.

(b) Key Differences Between RL and Supervised Learning

Feature	Supervised Learning	Reinforcement Learning
Data	Fixed, labeled dataset	Agent-generated through interaction
Feedback	Immediate, correct labels	Delayed, scalar rewards
Objective	Minimize prediction error	Maximize long-term reward
Behavior	Passive learning	Active decision-making
Exploration	Not needed	Essential

In supervised learning, you don't influence which data you see.

In RL, your actions change your future experiences — creating a feedback loop.

3. Fundamental Components of an RL Problem

To formally describe reinforcement learning problems, we use **Markov Decision Processes (MDPs)**.

An MDP is defined by:

- **Agent** – the learner/decision-maker.
- **Environment** – everything the agent interacts with.
- **State** (s) – the current situation the agent is in.
- **Action** (a) – what the agent can do in a state.
- **Reward** (r) – a scalar signal received after taking an action.
- **Policy** (π) – the agent's strategy: a mapping from states to actions.
- **Trajectory (Episode)** – a sequence of states, actions, and rewards.

The defining assumption of MDPs is the **Markov property**:

"The next state depends only on the current state and action, not on the sequence of states and actions that preceded it"

This allows for recursive formulations and makes planning and learning tractable in RL

(c) Exploration vs Exploitation

One of the two core challenges in RL.

Exploitation: choosing the action that has given high rewards in the past.

Exploration: trying new actions that may lead to better long-term outcomes.

If the agent only exploits, it may miss better strategies. If the agent only explores, it may fail to capitalize on what it already knows.

(d) Credit Assignment Problem

The second core challenge in RL.

How does the agent figure out which actions actually caused a good (or bad) outcome, especially when rewards are delayed?

Rewards may come many steps after the actions that caused them.

This makes it difficult to assign "credit" correctly to individual decisions.

(e) The Role of the Discount Factor (γ)

The agent's total return is:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \tag{1}$$

$\gamma \in [0, 1]$ controls how far into the future the agent "cares."

- $\gamma = 0$: only immediate reward matters (short-sighted).
- $\gamma \approx 1$: future rewards are nearly as important as immediate ones (long-term planning).
- $\gamma = 1$: possible only when total return is guaranteed to be finite (e.g., finite-horizon tasks).

2 Dynamic Programming

(a) Overview

Dynamic Programming (DP) provides a set of algorithms for solving Markov Decision Processes (MDPs) when the environment dynamics are fully known — that is, we have access to the transition probabilities $P(s' \mid s, a)$ and reward function $R(s, a, s')$.

DP methods are:

- Exact
- Rely on bootstrapping (using estimates to update estimates)
- Foundational for approximate and model-free RL

2. Basic Definitions

- **Policy (π):** A rule that tells the agent what action to take in each state. Can be deterministic ($\pi(s) = a$) or stochastic ($\pi(a \mid s)$).
- **State-Value Function ($V^\pi(s)$):** The expected return starting from state s, following policy π.
- **Action-Value Function ($Q_\pi(s, a)$):** The expected return starting from state s, taking action a, and then following policy π.

(b) The Bellman Equations

Bellman Expectation Equation for $V_\pi(s)$:

$$V^\pi(x) = r(x, \pi(x)) + \gamma \sum_y P(y \mid x, \pi(x)) V^\pi(y) \tag{2}$$

where:

- x is a **state**.
- $\pi(x)$ is the **action** your policy chooses in state x.
- $r(x, \pi(x))$ is the **immediate reward** from taking that action in that state.
- $P(y \mid x, \pi(x))$ is the **probability of ending up in state** y after taking action $\pi(x)$ in state x.
- $V^\pi(y)$ is the **value of the next state** y.

The Bellman equation says:

> "The value of the current state is the expected reward from taking an action now, plus the discounted value of the next most likely state, following your current policy π. You average over all actions your policy might take because you do not know exactly what it will do."

The Bellman equation for a fixed policy π can be written in matrix form as:

$$V^{\pi} = r^{\pi} + \gamma P^{\pi} V^{\pi} \tag{3}$$

(i) Explanation of Terms

- V^{π} is the **value function vector** — it stores the expected return starting from each state when following policy π.
- r^{π} is the **reward vector** — the expected immediate reward for each state under policy π.
- γ is the **discount factor** — a number between 0 and 1 that weights future rewards.
- P^{π} is the **state transition matrix under policy** π — it contains the probabilities of moving between states when following π. $P_{\pi}[x, y]$ gives the probability of transitioning from state x to y under policy pi. It is calculated as the sum over all possible actions, weighted by the probability of taking each action according to the policy. Mathematically:

$$P^{\pi}[x, y] = \sum_{a} P[x, y, a] \cdot \pi[x, a]$$

Why are they so powerful

- It gives us a system of equations (one per state) that we can solve numerically.
- It forms the core of algorithms like Value Iteration and Policy Iteration.
- It appears everywhere in RL — even deep RL algorithms like DQN or A3C build on it.

(c) . Policy Evaluation

Goal: Compute $V_{\pi}(s)$ for a fixed policy π.

We have two main options: power iteration or solving the Bellman Equation analytically. The analytical solution is presented in part 3. Below is the **power iteration** subroutine:

1. Start with an initial guess for V_{π} (e.g., all zeros)
2. Iteratively update using the Bellman equation for V_{π}
3. Stop when values converge (change is below a threshold)

The value we find, for any of the two solutions, is a matrix where every entry is a pixel in our environment. For every pixel, the value is the expected total reward for such state following the policy we are evaluating. A high value for a specific pixel indicates that if the agent starts in that state and follows the given policy (strategy), it is expected to achieve a high cumulative reward over time.

The sum of all pixels give the total value to the policy we are evaluating.

(d) Policy Iteration

A two-step algorithm that alternates between:

- **Policy Evaluation:** Compute V_π for the current policy
- **Policy Improvement:** Improve the policy by choosing better actions in each state based on the current value function, using the new value function:

$$\pi_V(x) = arg\max_a \left\{ r(x,a) + \gamma \sum_y P(y \mid x, a) V(x) \right\} \tag{4}$$

Guarantee: In finite MDPs, converges to the optimal policy π^* in a finite number of steps because:

- The number of possible policies is finite
- Each new policy is strictly better unless already optimal

(e) Value Iteration

Value iteration is an algorithm to compute the **optimal state-value function** $V^*(s)$. It's used when we know the environment (i.e., we have the transition matrix P and the reward vector r).

The goal is to estimate how valuable it is to be in each state if you act optimally from there on.

(i) How It Works (Plain Terms)

At each iteration:

- For every state x, you look at all possible actions a.
- For each action, you calculate the **expected value** of doing that action in state x, considering all possible next states and their values.
- You pick the action with the highest value — this gives you the best you can do in state x if you act optimally.
- You then update the value of state x as:

$$V = r + \gamma \cdot \max_a P[X, Y, a] \cdot V$$

- $r[x]$: the immediate reward for being in state x,
- γ: the discount factor,
- $P[X, Y, a] \cdot V$: matrix for the expected value of future states, following action a. We write X,Y because this is computed for all states.

- The max over a: represents choosing the best action.

You repeat this process for a number of iterations, improving the estimate each time. Eventually, the value function converges to the optimal one V^*.

A more in depth look: Imagine you're at a current state x. You have multiple possible actions a you can choose.

For each action a, you have:

- A probability distribution over next states x' (this is $P(x' \mid x, a)$).
- A reward now, plus future rewards from x' onward.

Thus, the expected value of taking action a at state x is:

$$Q(x, a) = \sum_{x'} P(x' \mid x, a) \times V(x')$$

where:

- $P(x' \mid x, a)$: probability of landing in next state x' if you take action a in state x.
- $V(x')$: value of being in state x' (future reward).

So: For each action at each state:

- You "look ahead" by one step:
  - Sum over all possible next states.
  - Weight the future value of x' by how likely you are to reach x' from x using action a.

After you compute $Q(x, a)$ for all actions a at state x, you choose the action with the maximum $Q(x, a)$.

This gives you the best thing you can do starting at state x.

(ii) Extracting the optimal policy

The value function is just a means to an end — our real goal is the **optimal policy** $\pi^*(s)$, which tells us what to do in each state.

The optimal policy is the one that chooses the action that maximizes the expected return, using the optimal value function V^*:

$$\pi^*(x) = \arg\max_a \sum_y P[x, y, a] \cdot [r[y] + \gamma V^*(y)]$$

- $P[x, y, a]$: probability of transitioning from state x to state y under action a,
- $r[y]$: reward for landing in state y,
- γ: discount factor,
- $V^*(y)$: the optimal value of state y.

This equation says: for each state x, choose the action that leads to the highest expected reward plus the discounted value of the next state.

(f) Final Insights and comparison

- **Policy Iteration:** Alternates between full evaluation and greedy improvement. It runs under fewer iterations (because policy stabilizes quickly) but each iteration is heavier (policy evaluation needs solving V).
- **Value Iteration:** Skips full evaluation and updates values directly. It has many iterations (small updates), but each iteration is light.

Both converge to the optimal value function V^* and policy π^* when transition dynamics are known.
When to use which:

- If policy evaluation is cheap (e.g., small or tabular state spaces), **Policy Iteration** can be more efficient overall.
- If evaluation is expensive (e.g., large or continuous state spaces), **Value Iteration** may be preferable due to its simplicity and lower per-iteration cost.

3 From Dynamic Programming to RL

(a) Why Move Beyond Dynamic Programming?

Dynamic Programming (DP) methods (e.g., Value Iteration, Policy Iteration) assume:

- Full knowledge of the transition probabilities $P(s' \mid s, a)$
- Full knowledge of the reward function $r(s, a, s')$
- Small, manageable state and action spaces

Advantages:

- DP methods are exact and deterministic.

Limitations in real-world problems:

- P and r are unknown
- The environment can only be sampled through experience
- The state/action spaces may be too large or continuous

(b) Conceptual Shift in Reinforcement Learning

We now move from:

"Compute expectations using known models"

to:

"Estimate expectations using sampled trajectories"

This marks the beginning of **model-free reinforcement learning**.

(c) Levels of Access to the Environment

Access Type	**Description**	**Enables**
Full access to P, r	Planning / simulation	Dynamic Programming
Generative model	Can query $P(\cdot \mid x, a)$ directly	Model-based RL
Simulated episodes	Can reset and run full trajectories	Monte Carlo
Online only	One uninterrupted trajectory	TD Learning, SARSA, Q-learning

(d) Monte Carlo Methods for Policy Evaluation

Goal: Estimate the value function:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s]$$

Core Idea:

- Run multiple episodes under policy π
- For each occurrence of a state s, record the return G_t
- Estimate $V^\pi(s) \approx$ average of returns from s

Two Variants:

- **First-visit MC:** only use the first time a state is visited in each episode
- **Every-visit MC:** use all visits to a state in all episodes

(i) Advantages of Monte Carlo

- Doesn't require knowledge of P or r
- Easy to implement: average observed returns
- Works well in environments where episodes can be simulated

(ii) Disadvantages of Monte Carlo

- Must wait for the end of the episode to update
- High variance in estimates
- Inefficient data use (no bootstrapping)

4 Basic RL Algorithms

(a) Monte Carlo Policy Evaluation

Monte Carlo (MC) methods estimate value functions by averaging returns from full episodes:

$$V^\pi(s) \approx \frac{1}{N(s)} \sum_{i=1}^{N(s)} G^{(i)}(s)$$

Limitations of Monte Carlo:

- Must wait until the end of the episode
- High variance due to long returns
- Inefficient in continuing or online environments

(b) Temporal Difference Learning: TD(0)

TD(0) bridges Monte Carlo and Dynamic Programming by using **bootstrapping**.

It updates value estimates using one-step returns. We simulate n episodes, and keep track of all states we visited, their rewards and actions. Then, for each simulation we go over all states and update the value function one-step at a time:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_t + \gamma V(s_{t+1}) - V(s_t)\right]$$

Where:

- α: learning rate
- $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$: TD error.

The TD error represents the difference between what we just saw: $r_t + \gamma V(s_{t+1})$ and *what we were expecting*: $V(s_t)$. Multiplying δ_t by the learning rate α nudges $V(s_t)$ toward making the gap smaller next time.

Intuition:

Rather than waiting for the full return G_t, TD estimates:

$$G_t \approx r_t + \gamma V(s_{t+1})$$

This is faster, lower variance, and works online.

(c) Bias-Variance Tradeoff

Method	**Target**	**Bias**	**Variance**
MC	Full return G_t	Low	High
TD(0)	One-step bootstrapped	Biased	Low

TD updates use estimates of future values $\rightarrow$ introduces bias, but results in more stable and sample-efficient learning.

(d) When is TD(0) Better?

- In non-terminating tasks (continuing problems)
- When we need online learning
- When computation and memory are limited
- When we want to update after each step, not episode

5. From Evaluation to Control

Once we estimate how good actions are, we want to improve the policy.
Instead of only learning $V(s)$, we now estimate:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[G_t \mid s_t = s, a_t = a]$$

(e) SARSA: On-Policy Control

SARSA updates action values using actual experience under the current policy:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]$$

Called SARSA because it uses: **S**tate, **A**ction, **R**eward, **S**tate, **A**ction.
Characteristics:

- On-policy: learns the value of the policy it follows (e.g., ε-greedy)
- Safer in risky or uncertain environments
- More conservative behavior when exploratory actions are dangerous

(f) Q-learning: Off-Policy Control

Q-learning updates toward the best possible action, regardless of the one taken:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right]$$

Characteristics:

- Off-policy: learns about the greedy policy while behaving otherwise
- More aggressive — may find the optimal policy faster
- Prone to overestimation (maximization bias) and instability with function approximation

(g) When to Use What?

Method	Tracks	Updates with	Pros	Cons
SARSA	current policy	actual next action	safer, stable	may be suboptimal
Q-learning	greedy policy	max over next actions	optimal (in theory)	can be risky, unstable

5 Approximate Dynamic Programming

(a) Motivation

In small, finite MDPs, we can store a table with one value per state. But what happens when:

- The state space is continuous (e.g., position in the real world)?
- There are millions or billions of states (e.g., Go, Atari, robots)?

It becomes impossible to:

- Visit every state even once,
- Store every value explicitly,
- Learn anything useful from sparse data.

So we generalize using function approximation: Instead of learning values for each state, learn a function that estimates values across similar states.

(b) The Function Approximator

We approximate the value function as:

$$V_\theta(x) = \theta^\top \phi(x)$$

Where:

- $\phi(x) \in \mathbb{R}^d$: feature vector that captures information about state x. In a grid map, this could represent the coordinates, plus other variables that encode different descriptive information.
- $\theta \in \mathbb{R}^d$: weights that we learn; how much value we give to each of the features in the feature vector.

Goal: learn weights θ so that $V_\theta(x) \approx V_\pi(x)$ — the true value of state x under policy π. This is like linear regression where the value of a state is predicted from its features.

(c) TD(0) Update with LFA

From a single transition (x_t, r_t, x_{t+1}), we compute:

- Prediction: $\theta^\top \phi(x_t)$
- TD target: $r_t + \gamma\theta^\top \phi(x_{t+1})$
- TD error: $\delta_t = r_t + \gamma\theta^\top \phi(x_{t+1}) - \theta^\top \phi(x_t)$

Then we update:

$$\theta \leftarrow \theta + \alpha \cdot \delta_t \cdot \phi(x_t)$$

This update moves θ to reduce the prediction error, just like in stochastic gradient descent.

Interpretation: You're trying to solve the projected Bellman equation:

$$V_\theta \approx \Pi T_\pi V_\theta$$

Where:

- T_π is the Bellman operator
- Π projects the target back into the space of functions spanned by ϕ

TD(0) with LFA performs stochastic approximation to this fixed point.

Pros

- Simple and online — no need to store data
- Can scale to large or continuous spaces
- Supports real-time updating

Cons

- Sensitive to learning rate α
- High variance and slow convergence
- Learning depends on good feature design

(d) Least-Squares Temporal Difference Learning (LSTD)

(i) Motivation

TD(0) updates slowly using one transition at a time. What if we could solve for θ exactly, based on a batch of data?

That's what LSTD does: Instead of approximating the solution through many updates, compute the best-fit θ that minimizes the TD error over a batch.

(ii) Derivation

You want:

$$V_\theta(x_t) \approx r_t + \gamma V_\theta(x_{t+1}) \Rightarrow \theta^\top \phi(x_t) \approx r_t + \gamma\theta^\top \phi(x_{t+1}) \Rightarrow \theta^\top(\phi(x_t) - \gamma\phi(x_{t+1})) \approx r_t$$

Stack these across all transitions and solve the least-squares problem:

$$\min_\theta \sum_{t=1}^{T} \left(\theta^\top(\phi_t - \gamma\phi_{t+1}) - r_t\right)^2$$

(iii) Algorithm

Define:

$$A = \sum_t \phi_t(\phi_t - \gamma\phi_{t+1})^\top$$

$$b = \sum_t r_t\phi_t$$

Then:

$$\theta = A^{-1}b \quad \text{or with regularization} \quad \theta = (A + \lambda I)^{-1}b$$

Intuition: Each transition says:

> "The value of this state should equal the observed reward plus the discounted value of the next state."

LSTD solves for the θ that best satisfies all these constraints — in the least-squares sense.

It's like solving a linear regression problem where:

- Inputs = $\phi_t - \gamma\phi_{t+1}$
- Targets = r_t

Pros

- No learning rate to tune
- Fast convergence — finds θ in one step
- Uses all data efficiently

Cons

- Requires batch access to data
- Needs to compute and invert $A \in \mathbb{R}^{d\times d}$
- Can be unstable when A is ill-conditioned (use regularization)

(e) Least-Squares Policy Evaluation (LSPE)

(i) Motivation

LSPE is an alternative to LSTD for evaluating a fixed policy using function approximation, especially when exact LSTD is too expensive or unstable.

(ii) Intuition

LSTD solves:

$$A_T\theta = b_T$$

This assumes:

- We can compute and invert A_T exactly,
- We want the fixed point immediately.

In contrast, LSPE says:

> Let's update θ iteratively toward a solution that minimizes the TD error over a batch of samples.

It's like projected value iteration, but in the function space.

(iii) LSPE Update Rule

We want to minimize the Bellman residual over a dataset $D = \{(x_t, r_t, x_{t+1})\}$:
Define a surrogate loss:

$$L_n(\hat{V}; \hat{V}^{k-1}) = \frac{1}{n}\sum_{t=1}^{n}\left(r_t + \gamma\hat{V}^{k-1}(x_{t+1}) - \hat{V}(x_t)\right)^2$$

This is a regression problem:

- **Target**: $r_t + \gamma\hat{V}^{k-1}(x_{t+1})$
- **Input**: x_t
- **Model**: $\hat{V}(x_t) = \theta^\top\phi(x_t)$

(iv) Algorithm (Batch Gradient View)

At iteration k, do:

1. For each $t = 1, \ldots, n$, compute:

$$y_t = r_t + \gamma\theta_{k-1}^\top\phi(x_{t+1})$$

2. Solve for θ_k using regression:

$$\theta_k = \arg\min_\theta \sum_t (y_t - \theta^\top\phi(x_t))^2$$

This is just linear regression: fit θ so that $\theta^\top\phi(x_t) \approx y_t$.

(v) How Is This Different from LSTD?

Feature	**LSTD**	**LSPE**
Solves for θ	All at once	Iteratively
Target	$r_t + \gamma\theta^\top\phi(x_{t+1})$	Uses θ_{k-1} in targets
Viewpoint	Solving projected Bellman equation	Regression to TD target
Cost	High upfront cost (inversion)	Lower per-iteration, more flexible

Pros of LSPE

- Doesn't require matrix inversion
- More numerically stable than LSTD
- Scales better when new data arrives — can update θ incrementally
- Natural link to SGD and nonlinear approximators

Cons of LSPE

- Needs several iterations to converge
- Less direct than LSTD
- Can be slower overall if sample size is small and well-conditioned

When to Use LSPE

- When LSTD is too expensive or numerically unstable
- When you want to use SGD-like updates over batches
- When you want to generalize LSPE to nonlinear approximators (e.g., with neural nets)

6 Multi-Armed Bandits (MAB)

(a) The Intuition: Slot Machines in a Casino

Imagine you walk into a casino with 10 slot machines (a.k.a. "one-armed bandits"). Each machine has a different probability of paying out, but you don't know what those probabilities are. You have a limited number of coins, say 100 pulls. What's your goal?

Maximize the total reward you get by choosing wisely which machine to play each time.

This is the essence of the Multi-Armed Bandit problem:

- Each arm (slot machine) gives a random reward drawn from a fixed but unknown distribution.
- At each time step, you choose one arm to pull.
- You observe the reward but not the probability distribution behind it.
- Your goal is to maximize the cumulative reward over time.

(b) Why It Matters

MAB problems appear everywhere:

- **Online ads:** Which ad to show to maximize clicks?
- **A/B testing:** Which version of a web page works best?
- **Clinical trials:** Which treatment is more effective?
- **Recommendation systems:** Which product to recommend?

What makes MAB hard is the **exploration vs. exploitation tradeoff**:

- **Exploration:** Try different arms to learn more about their payoffs.
- **Exploitation:** Choose the best-known arm so far to get the highest reward.

(c) Formal Setup

We define the problem as follows:

- There are K arms (indexed by $a = 1, 2, \ldots, K$).
- Each arm a is associated with an unknown reward distribution R_a with expected value μ_a.
- At each time $t = 1, 2, \ldots, T$, you select an arm $A_t \in \{1, \ldots, K\}$.
- You observe a reward $r_t \sim R_{A_t}$.

The objective is to maximize the expected cumulative reward:

$$\mathbb{E}\left[\sum_{t=1}^{T} r_t\right]$$

Since you don't know the μ_a's, you have to balance:

- Learning more about μ_a (exploration),
- Using the arm with the highest estimated μ_a (exploitation).

(d) Regret: Measuring How Much You Lose by Not Knowing

The performance of a MAB algorithm is often measured in terms of **regret**, which is the difference between the reward you could have gotten by always choosing the best arm vs. what you actually got.

$$\text{Regret}(T) = T\mu^* - \mathbb{E}\left[\sum_{t=1}^{T} r_t\right]$$

where $\mu^* = \max_a \mu_a$ is the best possible mean reward.

A good algorithm should have **sublinear regret**, i.e., $\text{Regret}(T) = o(T)$, meaning that the average regret per round goes to zero as $T \to \infty$.

(e) Algorithms for Solving MAB Problems

There are several standard approaches to the MAB problem:

1. **Epsilon-Greedy**

   With probability ε, choose a random arm (exploration). With probability $1 - \varepsilon$, choose the arm with the highest estimated mean (exploitation).

   Simple but inefficient as it explores uniformly and may pull bad arms too often.

2. **UCB (Upper Confidence Bound)**

   For each arm a, maintain an estimate $\hat{\mu}_a$ of the mean reward and count n_a of times it's been played. Add an optimism term to form the UCB:

   $$\text{UCB}_a(t) = \hat{\mu}_a + \sqrt{\frac{2\log t}{n_a}}$$

   At time t, pull the arm with the highest UCB.

   Balances exploitation (high $\hat{\mu}_a$) and exploration (large uncertainty when n_a is small).

3. **Thompson Sampling (Bayesian)**

   Maintain a probability distribution (posterior) over the expected reward of each arm. Sample one expected reward from each posterior and pick the arm with the highest sampled value.

   Naturally balances exploration and exploitation based on uncertainty.

(f) Digging Deeper: Where Do These Formulas Come From?

UCB: The Exploration Bonus

The bonus term $\sqrt{\frac{2\log t}{n_a}}$ comes from Hoeffding's inequality, which gives a high-probability bound on how far the empirical mean is from the true mean:

$$\mathbb{P}\left(\mu_a > \hat{\mu}_a + \epsilon\right) \leq e^{-2n_a\epsilon^2}$$

So we construct an upper bound that likely contains the true mean:

$$\hat{\mu}_a + \sqrt{\frac{2\log t}{n_a}}$$

As time increases, this bonus shrinks (you become more certain), but early on, it encourages trying lesser-known arms.

Thompson Sampling: A Bayesian View

For each arm, maintain a posterior distribution over μ_a. For example, in the Bernoulli reward case (reward is 0 or 1), the posterior for μ_a is a Beta distribution:

$$\mu_a \sim \text{Beta}(\alpha_a, \beta_a)$$

Update counts after each reward:

- If success (reward = 1): $\alpha_a \leftarrow \alpha_a + 1$
- If failure (reward = 0): $\beta_a \leftarrow \beta_a + 1$

Then sample $\tilde{\mu}_a \sim \text{Beta}(\alpha_a, \beta_a)$ and choose the arm with the highest $\tilde{\mu}_a$. The higher the uncertainty, the more variation in the samples, encouraging exploration.

(g) Choosing the Right Algorithm: When and Why

Different algorithms for Multi-Armed Bandits have different strengths. Choosing the right one depends on your problem's characteristics: how much data you have, whether you need fast decisions, and how complex the environment is.

Epsilon-Greedy: Simple and Effective for Stationary Problems

This strategy is easy to implement and works well when:

- The number of arms is small.
- The reward distributions are stationary (do not change over time).
- You don't need a very sophisticated exploration strategy.

It is often used as a baseline in experiments or when interpretability and simplicity are key.

UCB: Theoretical Guarantees and Deterministic Exploration

The UCB family of algorithms performs well when:

- You want strong theoretical guarantees on regret.
- You want a deterministic algorithm (no randomness in decision-making).

- You prefer confidence-bound-based decision-making (e.g., in medical trials or high-stakes applications).

UCB methods can struggle in non-stationary settings, where reward probabilities change over time.

Thompson Sampling: Probabilistic and Often State-of-the-Art
Thompson Sampling is widely used in practice because:

- It naturally adapts to uncertainty using Bayesian reasoning.
- It often outperforms others in real-world problems with limited data.
- It works well in both stationary and moderately non-stationary environments.

It is used by major companies like Google and Microsoft in A/B testing and recommendation systems.

When You Need More: Contextual and Full Reinforcement Learning
Multi-Armed Bandits assume that each decision is independent of the past state or future actions. But in many real-world tasks (games, robotics, inventory management), actions affect future states.

This is where **contextual bandits** and **reinforcement learning (RL)** come in.

- **Contextual Bandits** allow the algorithm to consider external information ("context") when making a decision — for example, user demographics when showing ads. Algorithms like LinUCB or Contextual Thompson Sampling are popular in personalized recommendations.
- **Reinforcement Learning (RL)** generalizes this even further. The agent learns a policy that maximizes expected cumulative rewards over time, accounting for changing states and delayed consequences.

Q-Learning: A Classic RL Algorithm
Q-learning is a model-free RL algorithm that estimates the value of action-state pairs, called Q-values:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Here:

- s is the current state, a is the action taken,
- r is the reward received, s' is the next state,
- α is the learning rate, and γ is the discount factor.

Q-learning works well in environments with:

- Finite, discrete state and action spaces,
- Observable state transitions,
- Markovian structure (future depends only on the current state and action).

In large or continuous environments, modern variants like **Deep Q-Networks (DQN)** are used, which combine Q-learning with neural networks to approximate value functions.

Modern RL in Industry

In practice, companies use:

- **Thompson Sampling** for scalable and adaptive online decision-making (e.g., recommender systems).
- **Contextual Bandits** in personalization (e.g., news feeds, ads).
- **Deep RL algorithms** like DQN, PPO, or A3C in gaming, robotics, and simulations.

Summary: How to Choose

- Use **Epsilon-Greedy** if simplicity is key and the number of actions is small.
- Use **UCB** if you want guarantees and controlled exploration.
- Use **Thompson Sampling** if you want strong empirical performance and probabilistic reasoning.
- Use **Contextual Bandits** if the best action depends on additional features or environment.
- Use **Full RL (e.g., Q-learning, DQN)** if your actions affect future states and rewards over time.