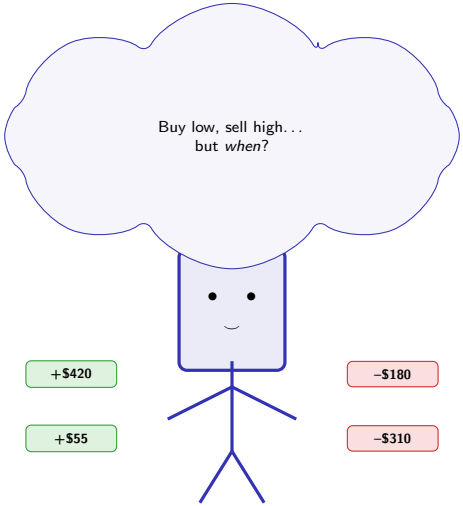
Reinforcement Learning Complete

From Bandits to RLHF — Every Concept, Every Formula

Methods & Algorithms

MSc Data Science – Spring 2026

Can a Machine Learn to Trade by Trial and Error?



RL agents learn from outcomes, not from labeled examples — the core distinction from supervised learning.

Learning Objectives

After this lecture you will be able to:

1. **Formalize** sequential decisions as MDPs with states, actions, and rewards
2. **Derive** Bellman equations and temporal-difference update rules
3. **Compare** SARSA, Q-learning, DQN, and PPO on convergence and sample efficiency
4. **Design** reward functions for financial trading agents
5. **Evaluate** when RL is appropriate versus supervised learning or optimization

---

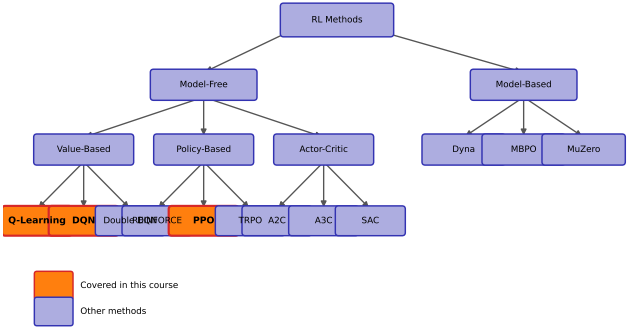
All objectives target Bloom's taxonomy levels 4–5 (analyze, evaluate, create).

Roadmap

1. Foundations
2. Value Functions
3. Model-Free Prediction
4. Model-Free Control
5. Deep RL
6. Policy Gradients
7. Modern RL
8. RL in Finance

We progress from simple bandits to state-of-the-art RLHF, one concept per slide.

The RL Landscape



RL Method Taxonomy

This lecture follows the progression from simple bandits (top-left) to modern policy methods (bottom-right).

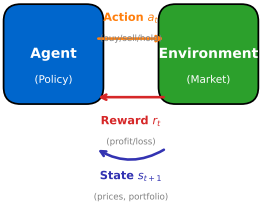
Each node in this taxonomy corresponds to one or more slides in this lecture.

What Is Reinforcement Learning?

- **Agent** interacts with **environment** over time
- Not supervised: no labeled training data
- Not unsupervised: agent has a goal (maximize reward)
- Learning by **trial and error** — actions produce consequences

Key elements: **state** s, **action** a, **reward** r, **policy** π

Reinforcement Learning: Agent-Environment Interaction



At each time step t:

Agent observes state, takes action, receives reward

https://github.com/Digital-AI-Finance/methods-algorithms/tree/master/slides/L06_Embeddings_RL/03_rl_loop

The agent-environment loop is the universal RL interface: observe, act, receive reward, repeat.

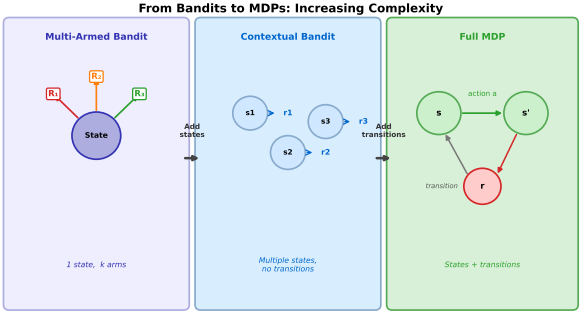
The Simplest RL Problem — Multi-Armed Bandits

- K slot machines, each with unknown payoff distribution
- Goal: maximize cumulative reward over T pulls
- Action value:

$$Q(a) = \mathbb{E}[R \mid A = a]$$

Sample-average update:

$$Q_{n+1}(a) = Q_n(a) + \frac{1}{n}\left[R_n - Q_n(a)\right]$$



Bandits are MDPs with a single state — the simplest testbed for exploration strategies.

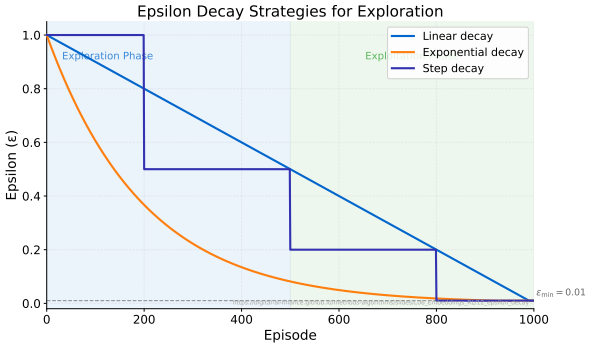
Exploration vs Exploitation

ε-greedy:

- Random action with probability ε
- Greedy action with probability $1 - \varepsilon$
- Decay: $\varepsilon_t = \varepsilon_0 \cdot e^{-\beta t}$

Upper Confidence Bound:

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$



ε-greedy is simple but wastes exploration budget; UCB targets under-explored arms.

Markov Decision Process (MDP)

An MDP is a tuple:

$$\mathcal{M} = (\mathcal{S},\ \mathcal{A},\ P,\ R,\ \gamma)$$

- $\mathcal{S}$ — **state space** (all possible situations)
- $\mathcal{A}$ — **action space** (all possible decisions)
- $P(s' \mid s, a)$ — **transition function** (dynamics of the environment)
- $R(s, a, s')$ — **reward function** (immediate feedback)
- $\gamma \in [0, 1)$ — **discount factor** (how much we value future rewards)

Markov property:

$$P(S_{t+1} \mid S_t, A_t) = P(S_{t+1} \mid S_1, A_1, \ldots, S_t, A_t)$$

The Markov property says the future depends only on the current state, not the full history.

Policies and Trajectories

Deterministic policy:

$$\pi(s) = a$$

Stochastic policy:

$$\pi(a \mid s) = P(A_t = a \mid S_t = s)$$

Trajectory:

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \ldots)$$

The goal of RL: find π^* that maximizes the expected **return** over trajectories.

A policy is the agent's strategy; a trajectory is one possible rollout of that strategy.

Definition

The **return** G_t is the discounted sum of future rewards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$

Worked example: If rewards are $+1, +1, +1, \ldots$ and $\gamma = 0.9$:

$$G_0 = 1 + 0.9 + 0.81 + \cdots = \frac{1}{1 - 0.9} = 10$$

Why discount?

- Uncertainty about the future — nearby rewards are more certain
- Mathematical convergence — ensures $G_t < \infty$ for bounded rewards

The discount factor γ encodes how far into the future the agent plans.

State-Value Function

Definition

The **state-value function** under policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t \mid S_t = s] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right]$$

Interpretation: "How good is it to *be in* state s under policy π?"

A state near a large reward has high value; a state near a penalty has low value.

V^{π} answers "what is the long-run expected payoff starting from here, following π?"

Action-Value Function

Definition

The **action-value function** under policy π:

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

Interpretation: "How good is it to *take action a* in state s under policy π?"

Relationship to V^π:

$$V^\pi(s) = \sum_a \pi(a \mid s)\, Q^\pi(s, a)$$

Q is action-specific; V is the policy-weighted average over all actions.

Bellman Expectation Equation

The value of a state decomposes recursively:

$$V^{\pi}(s) = \sum_{a} \pi(a \mid s) \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V^{\pi}(s') \right]$$

In words: the value of state s equals the expected immediate reward plus the discounted value of the next state.

This **recursive decomposition** is the foundation of all RL algorithms — every method exploits Bellman structure in some form.

Bellman (1957) showed that optimal sequential decisions reduce to solving this recursion.

Bellman Optimality Equation

For the **optimal** value functions:

$$V^*(s) = \max_a \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V^*(s') \right]$$

$$Q^*(s, a) = \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right]$$

The **optimal policy** acts greedily with respect to Q^*:

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

If we could solve Bellman optimality exactly, RL would reduce to a lookup table. In practice, we approximate.

Two alternating phases:

1. Policy Evaluation — solve for V^π via iterative Bellman updates:

$$V_{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a) \left[R + \gamma V_k(s') \right]$$

2. Policy Improvement — act greedily w.r.t. current values:

$$\pi'(s) = \arg\max_a Q^\pi(s, a)$$

Repeat until π converges. Requires full model knowledge: $P(s'|s, a)$ and R.

Policy iteration guarantees monotonic improvement and converges in finitely many steps.

Dynamic Programming — Value Iteration

Combines evaluation and improvement in a single update:

$$V_{k+1}(s) = \max_a \sum_{s'} P(s' \mid s, a) \left[R + \gamma V_k(s') \right]$$

- Converges to V^* as $k \to \infty$
- Extract optimal policy from final values: $\pi^*(s) = \arg\max_a Q^*(s, a)$
- Faster per iteration than policy iteration, but may need more iterations

Limitation: Both DP methods require the full model — not available in most real problems.

DP methods are the theoretical foundation; model-free methods (next section) remove the model requirement.

Update Rule (after complete episode)

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right]$$

- Uses the *actual return* G_t from the episode
- **Pro:** Unbiased estimate (no approximation of future values)
- **Con:** High variance; must wait until episode ends
- Works for **episodic tasks** only (games, episodes with terminal states)

MC averages complete trajectories — simple and unbiased, but slow to converge.

Temporal Difference Learning — TD(0)

Update Rule (after every step)

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]$$

The term $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is the **TD error**.

- **Pro:** Updates every step (online learning), works for continuing tasks
- **Con:** Biased — **bootstraps** from the current estimate $V(S_{t+1})$
- Combines ideas from MC (sampling) and DP (bootstrapping)

TD learning is the central idea that distinguishes RL from other learning paradigms.

MC vs TD — When Does Each Win?

Monte Carlo

- Unbiased
- High variance
- Episodic tasks only
- No bootstrapping
- Converges to minimum MSE

Temporal Difference

- Biased (bootstraps)
- Low variance
- Works online / continuing
- Bootstraps from estimates
- Converges to MLE of Markov model

Key insight: TD exploits the **Markov property**; MC does not.

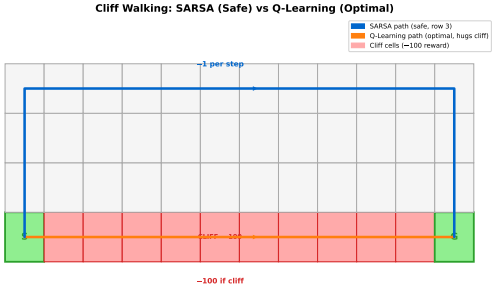
If the environment is truly Markov, TD is more data-efficient. If not, MC may be safer.

SARSA — On-Policy TD Control

Update Rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big]$$

- **On-policy**: uses action A_{t+1} actually taken under π
- Name: **S**tate-**A**ction-**R**eward-**S**tate-**A**ction
- Learns a safe policy (avoids cliffs)



SARSA learns the value of the policy it follows, including its exploration mistakes.

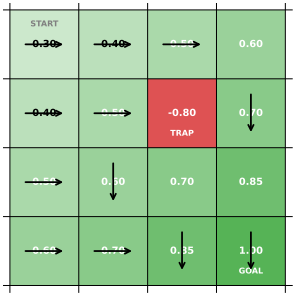
Q-Learning — Off-Policy TD Control

Update Rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\right]$$

- **Off-policy**: maximizes over all actions regardless of π
- Key difference from SARSA: $\max_{a'}$ vs $Q(S_{t+1}, A_{t+1})$
- Learns optimal policy even with exploratory behavior

Q-Learning: Grid World with Learned Q-Values



https://github.com/Digital-AI-Finance/methods-algorithms/tree/master/slides/L06_Embeddings_RL/04_q_learning_grid

Q-learning (Watkins, 1989) was the first provably convergent off-policy control algorithm.

Q-Learning Worked Example

Setup: 3×3 grid, $\alpha = 0.1$, $\gamma = 0.9$, all Q initialized to 0.

Step 1: State $(1,1)$, action = right, reward = 0, next state $(1,2)$:

$$Q((1,1),\text{right}) \leftarrow 0 + 0.1\big[0 + 0.9 \cdot \max_{a'} Q((1,2),a') - 0\big] = 0$$

Step 2: State $(1,2)$, action = right, reward = 0, next state $(1,3)$:

$$Q((1,2),\text{right}) \leftarrow 0 + 0.1\big[0 + 0.9 \cdot 0 - 0\big] = 0$$

Step 3: State $(1,3)$, action = down, reward = $+1$ (goal!), terminal:

$$Q((1,3),\text{down}) \leftarrow 0 + 0.1\big[1 + 0 - 0\big] = 0.1$$

Values propagate backward from the goal through repeated episodes.

This "backward flow" of values is how the agent discovers paths to reward.

Convergence Conditions

Q-learning converges to Q^* if:

1. All (s, a) pairs visited infinitely often
2. Learning rate satisfies **Robbins-Monro**:

$$\sum_{t=0}^{\infty} \alpha_t = \infty \quad \text{and} \quad \sum_{t=0}^{\infty} \alpha_t^2 < \infty$$

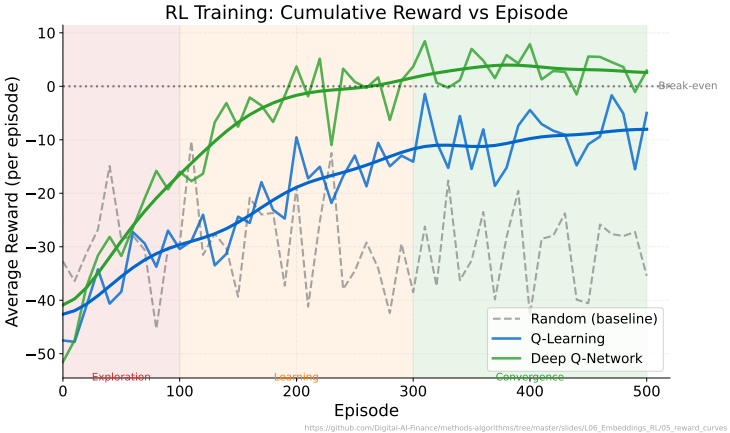
In practice:

- $\alpha_t = \frac{1}{t^{0.8}}$ (polynomial decay)
- Constant α with sufficient exploration also works empirically
- Convergence is asymptotic — finite-sample guarantees are weaker

Reference: Watkins & Dayan (1992)

These conditions ensure the agent explores enough while still settling on stable estimates.

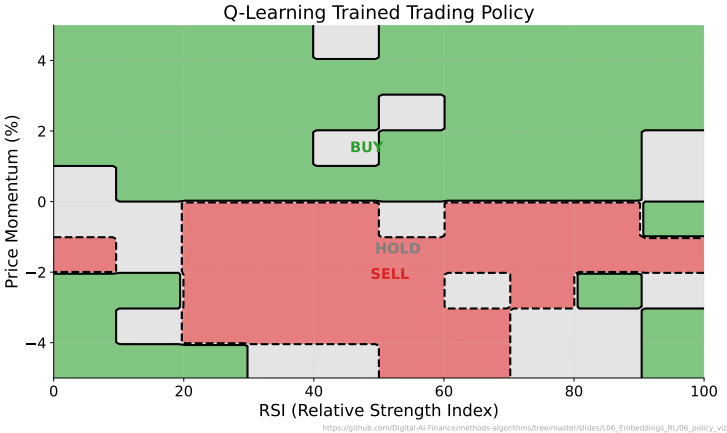
Watching the Agent Learn



Cumulative reward increases as the agent discovers better policies through repeated interaction.

Learning curves are the primary diagnostic: look for monotonic improvement and eventual plateau.

The Learned Policy



Arrows show the optimal action in each state after convergence.

A converged policy assigns a clear best action to every state — no more exploration needed.

Why Go Deep?

Tabular Q-learning

- Works for small, discrete state spaces
- Example: 4×4 grid $= 16$ states
- Stores one Q-value per (s, a)

The scaling problem

- Atari: $\sim 10^{70}$ pixel configurations
- Robotics: continuous joint angles
- Finance: real-valued prices, volumes

Solution: Approximate $Q(s, a) \approx Q(s, a; \theta)$ with a **neural network**.

Function approximation replaces the table with a parameterized model that generalizes across states.

Deep Q-Network (DQN)

Loss function:

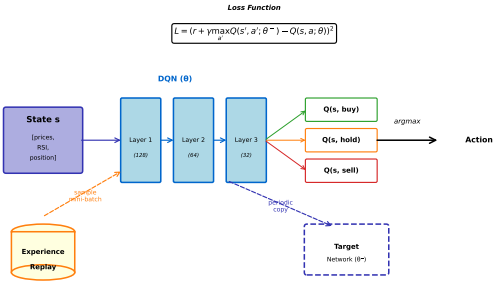
$$L(\theta) = \mathbb{E}\Big[\big(y - Q(s, a; \theta)\big)^2\Big]$$

where the **target** is:

$$y = R + \gamma \max_{a'} Q(s', a'; \theta^-)$$

Two key stability tricks:

- **Experience replay**
- **Target network** θ^-



Deep Q-Network Architecture for Trading

Reference: Mnih et al. (2015) — human-level Atari

DQN was the breakthrough that launched modern deep RL — the first agent to match human play.

DQN Stability: Replay + Target Network

Experience Replay

- Store transitions (s, a, r, s') in buffer $\mathcal{D}$
- Sample random mini-batches for training
- Breaks correlation between consecutive samples
- Reuses past experience (data-efficient)

Target Network

- Separate network θ^- for computing targets
- Updated slowly via soft copy:

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$$

- Prevents "moving target" problem
- Typical $\tau = 0.005$

Without these tricks, DQN diverges — the combination of function approximation, bootstrapping, and off-policy data is inherently unstable.

The Policy Gradient Theorem

Instead of learning Q, parameterize the **policy** directly: $\pi_\theta(a \mid s)$.

Objective: maximize expected return:

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

Policy Gradient Theorem:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a \mid s)\, Q^{\pi_\theta}(s, a)]$$

The gradient points in the direction that increases the probability of **high-reward actions**.

The log-derivative trick converts an intractable expectation into a sample-based gradient estimate.

REINFORCE Algorithm

Monte Carlo policy gradient:

1. Sample trajectory τ under π_θ
2. Compute return G_t for each time step
3. Update parameters:

$$\theta \leftarrow \theta + \alpha \, \nabla_\theta \log \pi_\theta(A_t \mid S_t) \, G_t$$

- Uses actual returns G_t — **unbiased**
- High variance — each trajectory is noisy
- Requires complete episodes (like MC prediction)

REINFORCE (Williams, 1992) is the simplest policy gradient method — elegant but impractical alone.

The Variance Problem and Baselines

Problem: REINFORCE has high variance $\rightarrow$ slow, unstable training.

Solution: Subtract a **baseline** $b(s)$ that does not depend on a:

$$\nabla_\theta J = \mathbb{E}\big[\nabla_\theta \log \pi_\theta(a \mid s)\,\big(Q^\pi(s,a) - b(s)\big)\big]$$

Natural choice: $b(s) = V^\pi(s)$, giving the **advantage**:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

- Positive advantage $\rightarrow$ action better than average $\rightarrow$ increase probability
- Negative advantage $\rightarrow$ action worse than average $\rightarrow$ decrease probability

---

Baselines reduce variance without introducing bias — a free improvement to any policy gradient.

Actor-Critic

Two networks working together:

- **Actor** $\pi_\theta(a|s)$ — the policy
- **Critic** $V_\phi(s)$ — the value function

Actor update:

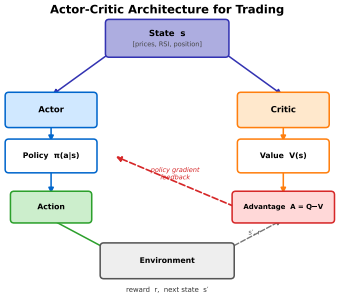
$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(a|s)\, \hat{A}_t$$

Critic update:

$$\phi \leftarrow \phi - \alpha_\phi \nabla_\phi \left(V_\phi(S_t) - G_t\right)^2$$

Advantage estimate:

$$\hat{A}_t = R_{t+1} + \gamma V_\phi(S_{t+1}) - V_\phi(S_t)$$



Actor-Critic combines policy gradients (actor) with value-based bootstrapping (critic) for lower variance.

PPO — Proximal Policy Optimization

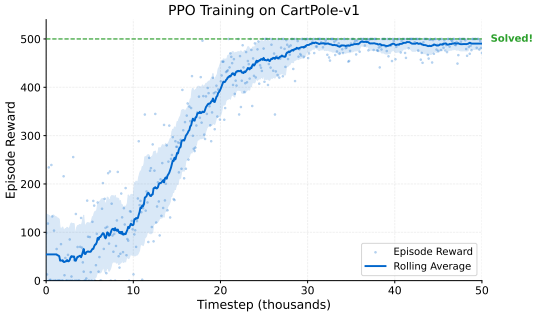
Clipped surrogate objective:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\Big(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t, 1-\varepsilon_c, 1+\varepsilon_c)\hat{A}_t\Big)\Big]$$

where the probability ratio is:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

- $\varepsilon_c \approx 0.2$ (clipping range)
- Prevents destructively large updates
- Distinct from ε (exploration)



Reference: Schulman et al. (2017)

PPO is the workhorse of modern RL — used in robotics, games, and LLM alignment (RLHF).

RLHF — RL from Human Feedback

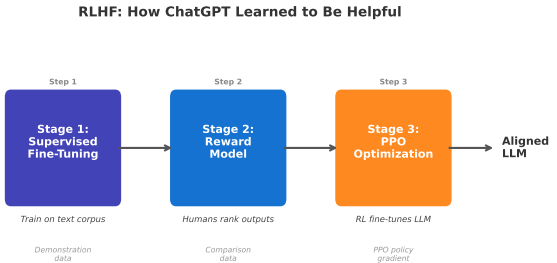
Three stages:

1. Supervised fine-tuning on demonstrations
2. Train reward model $r_\phi(x, y)$ from human preference pairs
3. Optimize policy via PPO against r_ϕ

Bradley-Terry preference model:

$$P(y_1 \succ y_2) = \sigma(r_\phi(x, y_1) - r_\phi(x, y_2))$$

Used by ChatGPT, Claude, Gemini.



RLHF turns human preferences into a reward signal — the bridge between RL and language models.

Offline RL — Learning Without Exploration

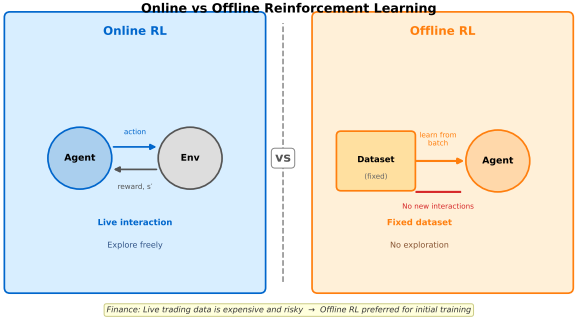
Problem: Online exploration is dangerous or impossible (medicine, finance).

Solution: Learn from a fixed dataset $\mathcal{D} = \{(s, a, r, s')\}$ collected by another policy.

Challenge: **Distribution shift** — agent queries $Q(s, a)$ for (s, a) not in $\mathcal{D}$.

CQL penalty:

$$\alpha \cdot \mathbb{E}_s \left[\log \sum_a \exp Q(s, a) \right]$$



Offline RL is critical for finance: we cannot "explore" with real money to gather training data.

Algorithmic Trading as an MDP

MDP mapping:

- **State:** portfolio weights, prices, indicators, cash
- **Actions:** buy, sell, hold (continuous or discrete)
- **Reward:** period return, risk-adjusted return, or Sharpe ratio
- **Transition:** market dynamics (stochastic, non-stationary)

Challenges unique to finance:

- Transaction costs and slippage
- Regime changes (bull $\rightarrow$ bear)
- Non-stationarity of returns
- Low signal-to-noise ratio

The MDP formulation makes financial trading a natural RL problem — but the devil is in the reward design.

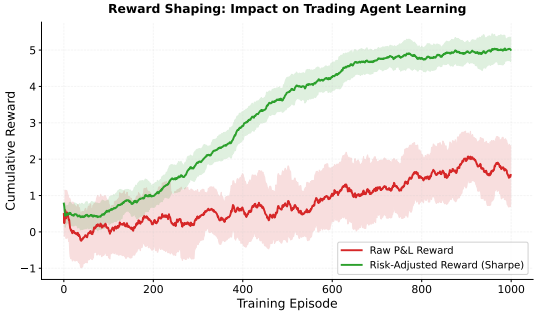
Composite reward function:

$$r_t = R_{p,t} - \lambda \cdot \mathrm{DD}_t - c \cdot |w_t - w_{t-1}|$$

Components:

- R_p: portfolio return
- DD: drawdown penalty (λ controls risk aversion)
- $c \cdot |\Delta w|$: transaction cost penalty

"The reward function is the most important design choice in financial RL."

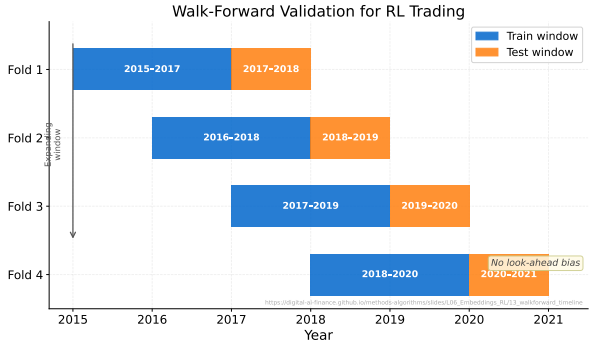


A poorly designed reward leads to degenerate policies — agents that churn or never trade.

Walk-Forward Backtesting

- Standard train/test split leaks future information in time series
- **Walk-forward**: rolling train window → test on next period → advance
- Critical for honest RL evaluation in finance

Prevents **look-ahead bias** — the most common pitfall in financial backtests.



Any RL trading result without walk-forward validation should be treated with extreme skepticism.

When to Use RL — Decision Framework

Reinforcement Learning

- Sequential decisions
- Delayed rewards
- Exploration is valuable
- Environment is interactive

Supervised Learning

- Labels available
- i.i.d. data
- Single-step prediction
- Static dataset

Optimization

- Model is known
- Constraints explicit
- No learning needed
- Closed-form or solver

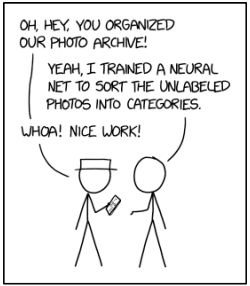
RL is powerful but expensive — use supervised learning or optimization when they suffice.

Key Takeaways

1. **MDP:** RL formalizes sequential decisions as $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$
2. **Bellman:** Value functions decompose recursively; all algorithms exploit this
3. **TD vs MC:** TD bootstraps (biased, low variance); MC waits (unbiased, high variance)
4. **Q-Learning:** Off-policy TD control converges to Q^* under Robbins-Monro conditions
5. **DQN:** Neural networks scale RL to high-dimensional state spaces
6. **PPO:** Clipped surrogate objective enables stable policy optimization
7. **Finance:** Reward engineering is the hardest part of RL in trading

These seven ideas form a complete conceptual toolkit for understanding modern RL systems.

One Last Thought...



XKCD #2173 by Randall Munroe (CC BY-NC 2.5).

References

- Sutton & Barto (2018). *Reinforcement Learning: An Introduction*. MIT Press.
- Watkins & Dayan (1992). Q-Learning. *Machine Learning*, 8(3).
- Mnih et al. (2015). Human-level control through deep RL. *Nature*, 518.
- Schulman et al. (2017). Proximal Policy Optimization Algorithms. *arXiv:1707.06347*.
- Ouyang et al. (2022). Training language models to follow instructions (InstructGPT). *NeurIPS*.
- Levine et al. (2020). Offline Reinforcement Learning: Tutorial, Review, and Perspectives.

Sutton & Barto is the essential reference — freely available at incompleteideas.net.