Reinforcement Learning: The Big Picture

Every Concept Explained — No Formulas Required

Methods & Algorithms

MSc Data Science – Spring 2026

What If a Computer Could Learn from Its Mistakes?

XKCD #1838 by Randall Munroe (CC BY-NC 2.5). RL is the branch of ML that learns by doing.

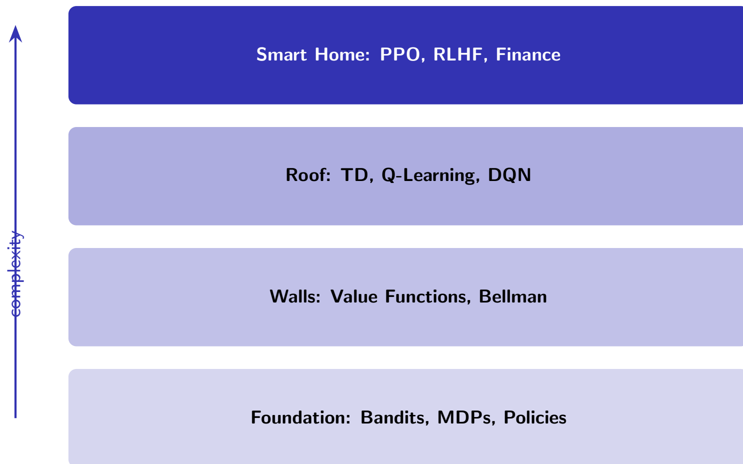
What You Will Learn

1. Understand the building blocks that every RL system is made of
2. Explain what each algorithm actually does in plain English
3. See how simple ideas combine into powerful systems like ChatGPT
4. Know when RL is the right tool — and when it's not
5. Connect every concept to a real-world analogy you can remember

---

This lecture is the conceptual companion to L06g. No formulas here — see L06g for all the math.

Roadmap — The RL Builder's Journey

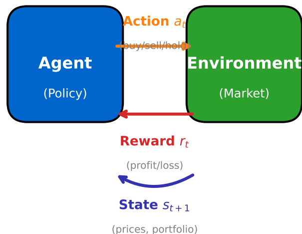


We'll build your understanding one layer at a time.

Each layer builds on the one below. By slide 42, you'll see the full picture.

What Is Reinforcement Learning in One Sentence?

Reinforcement Learning: Agent-Environment Interaction



At each time step t:

Agent observes state, takes action, receives reward

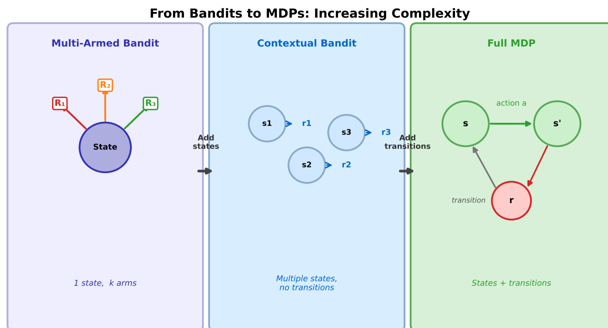
https://github.com/Digital-AI-Finance/methods-algorithms/tree/master/slides/L06_Embeddings_RL/03_rl_loop

"An agent learns to make good decisions by trying things, seeing what happens, and adjusting its strategy."

- Not supervised: no one tells it the right answer
- Not unsupervised: it has a clear goal (maximize reward)
- It learns by **trial and error**

The agent–environment loop is the central diagram of RL. Every algorithm fits inside this picture.

What Is the Simplest Possible Decision Problem?



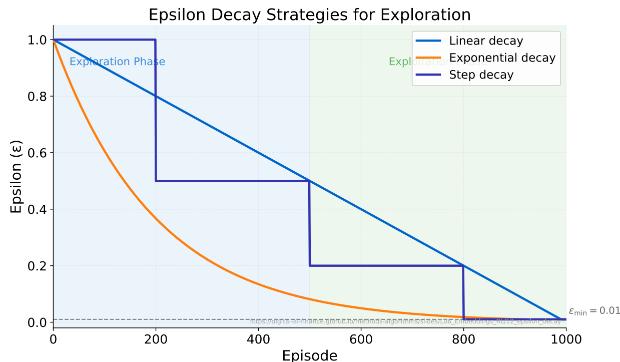
Imagine a row of slot machines, each with a different (unknown) payout. You want to find the best one as fast as possible.

This is the **multi-armed bandit** problem — the simplest form of RL.

The agent's job: figure out which actions pay best by trying them.

Bandits have no states — every pull is independent. Real RL adds states and transitions on top.

Should You Try Something New or Stick with What Works?



The **restaurant problem**: your favorite restaurant is great (exploit). But what if there's an even better one you haven't tried (explore)?

RL agents balance this by exploring randomly some fraction (ε) of the time, and exploiting their best-known action the rest.

ε-greedy is the simplest strategy. ε typically decays over time: explore more early, exploit more later.

Why Does Your Position Matter?

Slot machines don't have memory — pull any handle anytime. But most real problems do.

In chess, WHICH move is best depends on WHERE the pieces are. Your **state** changes what you should do.

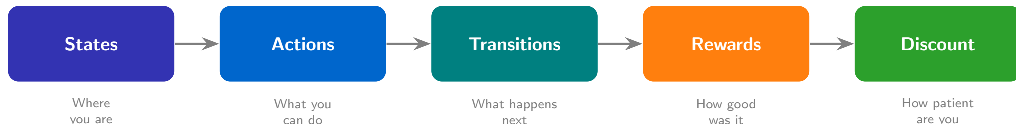
This is the jump from bandits to full RL: actions depend on states.

- A slot machine has one state — every pull is the same situation
- A chess game has billions of states — every board position is different
- A trading bot's state includes prices, holdings, and market conditions

---

The "Markov" part means: the current state contains all the information you need. The past doesn't matter beyond what's captured in the state.

What Are the Five Ingredients of Any Decision Problem?



This is the **Markov Decision Process (MDP)** — the universal recipe for any sequential decision problem.

Every RL problem, from Atari to trading, fits this template.

An MDP is fully defined by the tuple (States, Actions, Transitions, Rewards, Discount). Everything else is algorithm design.

What Is a Policy?

A **policy** is a complete strategy — a rule that tells you what to do in EVERY possible situation.

Think of it as a playbook: if you're at intersection A, turn left. If you're at intersection B, go straight. A policy covers every intersection.

- A deterministic policy: "always turn left at this corner"
- A stochastic policy: "turn left 70% of the time, right 30%"
- The goal of RL: find the BEST policy — the one that collects the most reward over time

---

Policies can be simple lookup tables or complex neural networks. The representation changes; the idea stays the same.

What Is a Trajectory, and Why Does Patience Matter?

Trajectory: The full recording of one episode — every state visited, action taken, and reward received. Like the play-by-play of a game.

The Return: The total reward you collect, but with a twist — future rewards count less. A dollar today is worth more than a dollar tomorrow.

The **discount factor** γ controls how patient you are:

- γ close to 1: very patient — cares about distant future rewards
- γ close to 0: impatient — only cares about immediate rewards
- $\gamma = 0.99$ is typical — mildly patient, but the far future fades away

---

Discounting also ensures the total return stays finite in problems that never end (continuing tasks).

How Good Is This State?

The **value** of a state is a prediction: if I'm here right now, how much total reward do I expect to collect from this point forward?

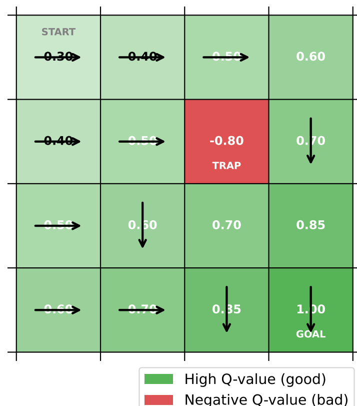
Think of it like GPS: "estimated time remaining." A state with high value means you're in a good position. Low value means trouble ahead.

- Value is always relative to a policy — different strategies assign different values to the same state
- High-value states: near the goal, many paths forward
- Low-value states: near a cliff, few options left

The state-value function $V(s)$ maps every state to a single number: expected future reward under the current policy.

How Good Is This Action from This State?

Q-Learning: Grid World with Learned Q-Values



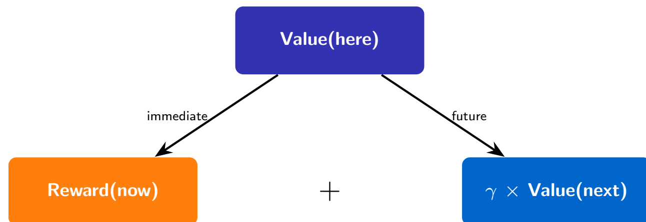
https://github.com/Digital-AI-Finance/methods-algorithms/tree/master/slides/L06_Embeddings_RL/04_q_learning_grid

The **Q-value** answers a more specific question: how good is it to take THIS particular action from THIS particular state?

Like a chess engine that rates every move from every position. "Move the knight here: +3.2. Move the bishop there: +1.7."

$Q(s, a)$ **gives one number per state-action pair. The grid shows Q-values the agent has learned through exploration.**

How Do You Break a Huge Problem into Tiny Pieces?



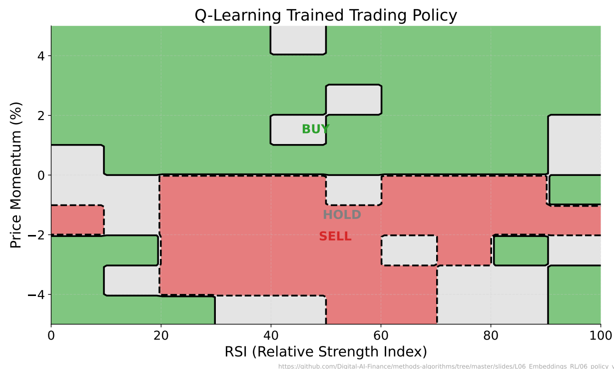
The most important idea in RL: you don't need to simulate the entire future.

The value of being HERE equals the reward you get NOW plus the (discounted) value of wherever you end up NEXT.

This recursive trick is what makes RL computationally feasible.

This is the Bellman equation. It turns an impossibly long sum into a one-step relationship. Every RL algorithm exploits this.

What Would a Perfect Agent Look Like?

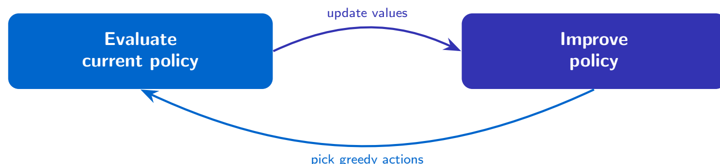


If you somehow knew the exact Q-value for every state-action pair, the best policy would be trivial: always pick the action with the highest Q-value.

The arrows show the optimal action in each state. RL's job is to DISCOVER these values — and it does so without ever seeing the full map.

Optimal policies are deterministic: for every state, there is one best action. The challenge is finding it.

Can You Plan the Perfect Strategy If You Have a Map?



If you have a complete model of the environment (every transition probability, every reward), you can COMPUTE the optimal policy directly.

- **Policy iteration**: evaluate then improve, repeat until convergence
- **Value iteration**: cut to the chase — keep asking "what's the best I can do from here?"
- Problem: in the real world, you rarely have the full map

Dynamic programming is the theoretical foundation but requires a known model. Model-free methods don't.

What If You Don't Have the Rules?

Everything above assumed you KNOW the rules — all transition probabilities, all rewards. But what if you don't?

That's most real-world problems. You don't know the traffic patterns. You don't know the stock market dynamics. You can only INTERACT with the world and learn from experience.

This is **model-free** RL — learning without a map.

- No transition model needed — just take actions and observe
- Two main approaches: Monte Carlo and Temporal Difference
- Both learn from raw experience, but in very different ways

Model-free methods are the workhorses of practical RL. Most real applications use them.

What If You Play the Whole Game Before Learning?

Approach 1: Monte Carlo. Play an entire episode (hand of poker). See how much you won. Update your estimate of how good each state was.

- Pro: you're using the REAL outcome — no approximation
- Con: you have to wait until the end — and outcomes are noisy
- One lucky hand of poker doesn't tell you much

This is Monte Carlo learning: learn from complete experiences.

MC methods are unbiased but high-variance. They require episodic tasks (ones that end).

What If You Update After Every Single Step?

Approach 2: Temporal Difference. Don't wait for the end. After EACH step, update your estimate using the reward you just got plus your ESTIMATE of what's ahead.

Like GPS recalculating after every turn — you don't wait until you arrive to know if you're on track.

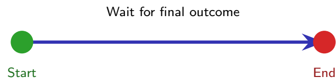
- Pro: fast, works online, doesn't need episodes to end
- Con: you're bootstrapping — using an estimate to update an estimate — which introduces bias

---

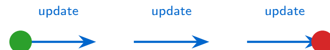
TD learning is the most distinctive idea in RL. It combines Monte Carlo sampling with dynamic programming bootstrapping.

How Do Monte Carlo and TD Compare?

Monte Carlo



Temporal Difference

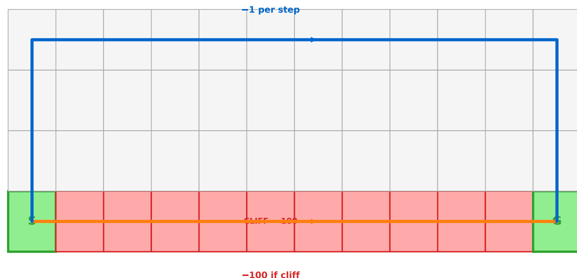


Unbiased but noisy (high variance)

Biased but stable (low variance)

TD exploits the pattern that states connect to each other. MC treats each outcome independently.

In practice, TD methods dominate because they learn faster and work in continuing (non-episodic) tasks.



SARSA updates based on the action you ACTUALLY took — including your exploratory mistakes.

Result: it learns a SAFE policy. On the cliff walk (chart), SARSA takes the long way around because it accounts for accidental cliff falls during exploration.

On-policy: learns the value of the strategy it's actually following.

SARSA stands for State-Action-Reward-State-Action — the five elements it uses for each update.

Can You Learn the Best Strategy While Exploring Randomly?

Q-learning updates based on the BEST possible action — even if you didn't take it.

Result: it learns the OPTIMAL policy, regardless of how it explores. On the cliff walk, Q-learning takes the risky edge path because that IS the shortest route.

Off-policy: learns the value of the best strategy, even while following a different one.

- SARSA: "what reward did I get following MY strategy?" (safe)
- Q-learning: "what reward COULD I get following the BEST strategy?" (optimal)
- Same data, different questions, different learned policies

Q-learning (Watkins, 1989) is one of the most important algorithms in RL. It converges to the optimal policy under mild conditions.

How Does Q-Learning Actually Work?

Imagine a 3$\times$3 grid. You start knowing nothing — all Q-values are zero.

Step 1: You're at (1,1), move right to (1,2), get reward 0.
Update: slightly increase Q((1,1), right).

Step 2: At (1,2), move right to (1,3), get reward 0.
Update: slightly increase Q((1,2), right).

Step 3: At (1,3), move down to goal, get reward $+10$.
Update: SIGNIFICANTLY increase Q((1,3), down).

Over thousands of steps, these small updates propagate backward until Q((1,1), right) reflects the distant $+10$.

This backward propagation of value is the key mechanism. Information flows from rewarding states backward to earlier states.

What Guarantees That Q-Learning Actually Works?

Two conditions in plain English:

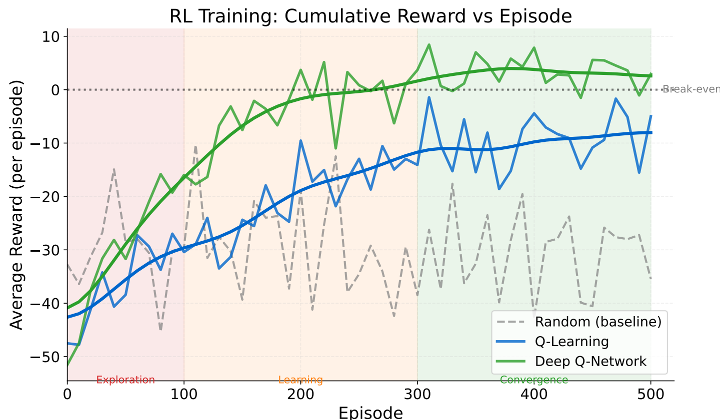
Condition 1: Visit everywhere. The agent must try every state-action pair infinitely often. If you never try action X in state Y, you'll never learn its value.

Condition 2: Slow down your learning. The learning rate must decrease over time — big updates early, tiny updates later. This lets early exploration be bold while late updates are precise.

When both conditions hold, Q-learning is *mathematically guaranteed* to find the optimal policy.

These are the Robbins-Monro conditions. In practice, fixed small learning rates work well even though the formal guarantee requires decay.

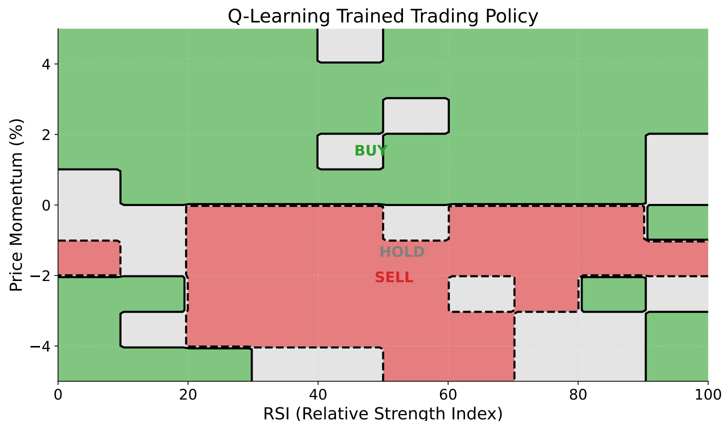
What Does Learning Look Like Over Time?



The reward goes up because the agent gets better at choosing. Early on, it explores randomly. Over time, it locks in on the best strategy.

Reward curves are the standard way to visualize RL training progress. Plateaus often indicate the agent has found a stable policy.

What Does the Final Strategy Look Like?



After thousands of episodes, the agent has learned which action is best in every state. The arrows show its final strategy — a complete policy, learned entirely from experience.

Compare this to slide 15 — the agent discovered the same optimal policy without ever seeing the transition model.

When Does the Q-Table Become Impossibly Large?

Q-learning stores a Q-value for every state-action pair in a table. A 4×4 grid has 16 states $\times$ 4 actions $=$ 64 entries. Manageable.

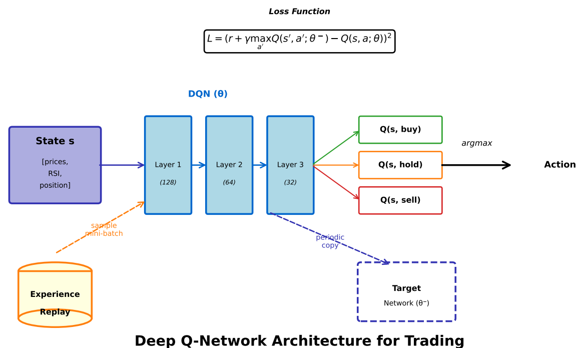
An Atari game has roughly 10^{70} possible screen configurations. That table would be larger than the number of atoms in the universe.

We need a way to GENERALIZE — predict Q-values for states we've never seen.

- Tabular methods: exact, but only work for tiny problems
- Function approximation: approximate, but scales to any size
- The key insight: similar states should have similar values

This is the "curse of dimensionality" — state spaces grow exponentially with the number of features.

Can a Neural Network Replace a Q-Table?



Deep Q-Network Architecture for Trading

The solution: replace the table with a **neural network**. Feed in a state, get out Q-values for every action.

The network learns PATTERNS — so it can estimate Q-values for new states by finding similarities to states it's seen before.

This is **Deep Q-Network (DQN)** — the breakthrough that let RL play Atari at human level (Mnih et al., 2015).

DQN was the paper that launched the deep RL revolution. It combined Q-learning with convolutional neural networks.

What Two Tricks Make DQN Actually Work?

Experience Replay

"Shuffle your homework."

Instead of learning from experiences in order (which creates bias), store past experiences in a buffer and sample randomly.

Like studying flashcards in random order instead of front-to-back.

Target Network

"Use yesterday's answer key."

A separate, slowly-updated network provides stable learning targets.

Without it, you're chasing a moving target — like trying to hit a bullseye that keeps jumping.

Both tricks stabilize training. Without them, DQN diverges. Together, they made deep RL practical for the first time.

Is There Another Way Besides Learning Values?

Everything so far learns VALUE and then derives the policy. But there's an alternative: learn the POLICY directly.

Instead of asking "how good is each action?" ask "what PROBABILITY should I assign to each action?"

This is **policy gradient** — and it's the foundation of modern methods like PPO.

- Value-based (Q-learning, DQN): learn values, pick the best action
- Policy-based (REINFORCE, PPO): learn the policy directly
- Both converge to the same optimal behavior, but through different paths

---

Policy gradients naturally handle continuous action spaces (e.g., how much torque to apply) where Q-learning struggles.

What Is the Simplest Policy Gradient?

The simplest policy gradient: play an episode. If the outcome was good, make all those actions MORE likely next time. If bad, make them LESS likely.

Like coaching: "Whatever you did in that winning game, do more of it."

This is **REINFORCE** — the simplest policy gradient algorithm.

- Pro: elegant, simple, theoretically sound
- Con: very noisy — one lucky game might reinforce bad habits
- Needs many episodes to average out the randomness

REINFORCE (Williams, 1992) was the first practical policy gradient. Modern methods build on its core idea.

How Do You Know If a Score Is Good or Bad?

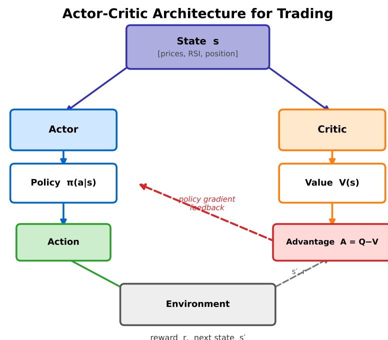
Imagine a student scores 72 on a test. Was that good or bad? You can't tell without context.

The fix: subtract the CLASS AVERAGE. If the average is 65, a 72 is above average — reinforce. If the average is 80, a 72 is below average — discourage.

- This "class average" is called the **baseline**
- The difference (score minus baseline) is called the **advantage**
- It dramatically reduces the noise in learning

The baseline doesn't change what the algorithm converges to — it only reduces variance, making learning faster and more stable.

Can Two Networks Coach Each Other?



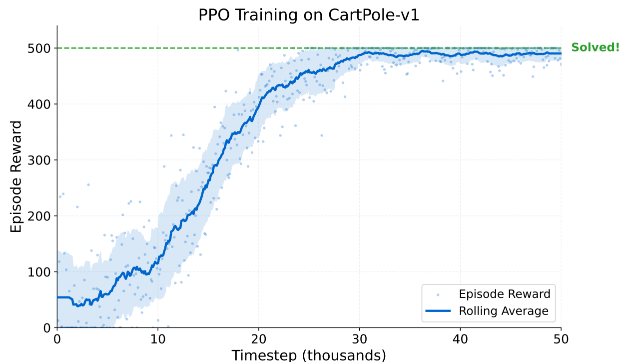
The **actor** decides what to do (the policy). The **critic** evaluates how good the decision was (the value function).

The critic tells the actor: "That was 3 points better than average — keep it up" or "That was 2 points worse — try something different."

They improve together: the critic gets better at evaluating, and the actor gets better at deciding.

Actor-critic combines the strengths of policy gradients (actor) and value methods (critic) into one architecture.

How Does PPO Prevent Catastrophic Updates?



The problem with policy gradients: one big update can destroy a good policy.

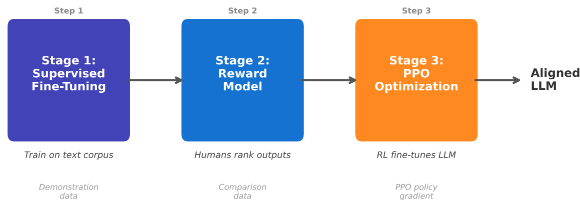
PPO prevents this by limiting how much the policy can change in a single step.

Think of it as a guardrail: if the update would change the policy too dramatically, it gets clipped back to a safe range.

PPO is the workhorse of modern RL — used in ChatGPT, robotics, and game-playing agents.

PPO (Schulman et al., 2017) achieves near-TRPO performance with much simpler implementation. It's the default choice for most RL applications.

How Does ChatGPT Know What “Helpful” Means?



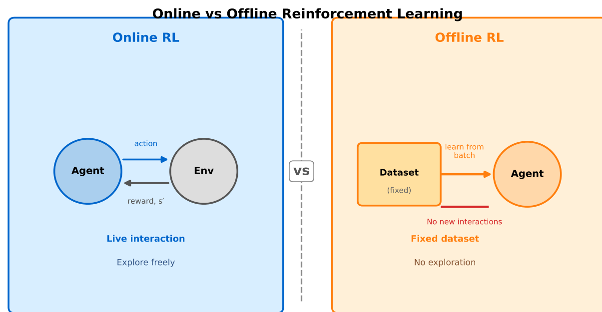
How do you reward an AI for being “helpful”? You can’t write a formula for helpfulness.

Solution: ask HUMANS. Show them two AI responses, ask which is better. Train a reward model from thousands of these comparisons. Then optimize the AI using PPO against this learned reward.

This is **RLHF** — how ChatGPT, Claude, and Gemini learned to be helpful.

RLHF was introduced by Christiano et al. (2017) and scaled by InstructGPT (Ouyang et al., 2022).

Can You Learn a Good Policy Without Any New Exploration?



Some environments are too dangerous or expensive to explore. You can't let a medical AI experiment on patients. You can't let a trading bot lose millions learning the market.

Solution: learn from RECORDINGS of past behavior — a fixed dataset of past decisions and outcomes.

Offline RL turns historical data into better policies, without any new exploration.

Offline RL is especially relevant for finance, healthcare, and any domain where exploration is costly or dangerous.

How Do You Map Trading to the RL Framework?

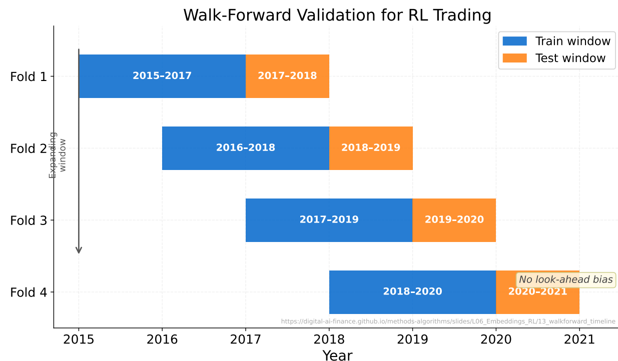
Map trading to the MDP framework:

- State = portfolio holdings, prices, indicators, cash position
- Action = buy, sell, hold (how much, which assets)
- Reward = profit minus risk penalty minus transaction costs

The hardest part: designing the reward. Maximize returns alone, and the agent takes insane risks. Add a risk penalty, and it becomes conservative. Getting the balance right is the art of **reward engineering**.

Reward shaping is arguably the most important design decision in applied RL. A badly shaped reward leads to unexpected behavior.

How Do You Honestly Test a Trading Agent?

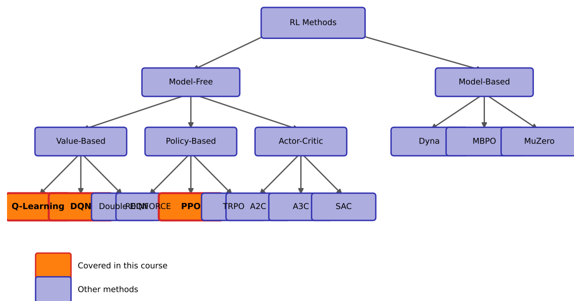


A common mistake: training and testing on the same time period. This "peeks into the future" and gives falsely good results.

The fix: **walk-forward backtesting**. Train on months 1–6, test on month 7. Then train on months 2–7, test on month 8. Roll the window forward.

This gives an honest estimate of how the agent would perform in real time.

Walk-forward testing is the gold standard for evaluating time-series models. It prevents look-ahead bias.



RL Method Taxonomy

Use RL when:

- Decisions are sequential
- Rewards are delayed
- Exploration is possible or affordable

Use supervised learning when:

- You have labeled data
- Decisions are one-shot

Use optimization when:

- You know the model
- Constraints are explicit

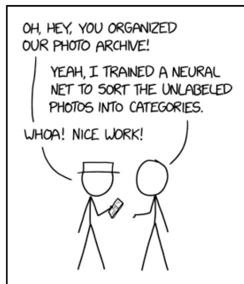
RL is the right tool when the problem has sequential structure and delayed feedback. Otherwise, simpler methods win.

Key Takeaways

1. **MDP**: Every decision problem has states, actions, transitions, rewards, and a patience factor
2. **Value**: The value of a state is a prediction of future reward — like GPS estimated time remaining
3. **Bellman**: Break big problems into “reward now + value of next” — the recursive insight
4. **TD vs MC**: Learn every step (fast, approximate) or wait for the outcome (slow, exact)
5. **Q-learning**: Rate every action from every state — converges to optimal with enough exploration
6. **Deep RL**: Neural networks replace tables when the state space is too large to count
7. **PPO + RLHF**: Small, safe policy updates + human preference data = modern AI assistants

Each takeaway maps to a section of this lecture. For formulas behind each concept, see L06g: RL Complete.

One Last Thought...



You've built every piece of the RL puzzle — no formulas needed.
For the math behind each concept, see **L06g: RL Complete**.

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

Where to Go from Here

Next Steps:

- For formulas: L06g RL Complete (43 slides)
- Textbook: Sutton & Barto (2018), free online
- Coding: L06 Jupyter notebook (hands-on Q-learning)

Key Vocabulary:

- Agent, Environment
- State, Action, Reward
- Policy, Value function
- Q-value, TD error
- Advantage

Every term on the right was introduced in this lecture with an analogy you can remember.

Sutton & Barto, "Reinforcement Learning: An Introduction," 2nd ed. (2018). Available free at incompleteideas.net/book/the-book.html.