AI Agents for Science

Lecture 14, November 12
Finetuning and reinforcement learning

Instructor: Ian Foster
TA: Alok Kamatar

Crescat scientia; vita excolatur

CMSC 35370 -- https://agents4science.github.io
https://canvas.uchicago.edu/courses/67079

Reinforcement learning papers

Adapting agents with reinforcement learning and real-world training.

OpenPipe/ART: Agent Reinforcement Trainer

Agent Lightning: Train ANY AI Agents with Reinforcement Learning

Recall: An agent is …

- An agent is a system that:
  - **Senses** (reads inputs, environment, or tool responses)
  - **Plans** (decides what to do next)
  - **Acts** (calls a tool)
  - **Learns** (updates state)
- Its operations are governed by policy that governs how it chooses *actions* in response to *states* (its current context)
- An LLM/RM may be used in various contents

Possible roles of LLMs/RMs in agents

Stage	Role / Function	Possible LLM Involvement
Sense	Observe the environment: read user inputs, tool outputs, or API responses	LLM interprets or summarizes current context (e.g., “What did the tool return?”)
Plan	Decide *what to do next*: which subgoal or tool to use, how to proceed	LLM generates next-step plans or chain-of-thought reasoning
Act	Execute chosen step: call a function, run a tool, or generate a response	LLM issues structured commands or final answers
Learn	Improve based on results, rewards, or feedback	RL or prompt optimization adjusts LLM behavior or policy

Improving agent performance

- Goal: alter agent behavior, e.g., improves some aspect of performance
- Generally we want to do this by training, which may involve:
  - **LLM fine-tuning**: Presenting agent LLM with structured data (e.g., prompt/answer pairs) and adjusting LLM parameters to increase match to answer
  - **Reinforcement learning**: Running the agent in a real or simulated environment and adjusting aspects of the agent implementation (LLM, other policy components) to improve the reward obtained
- Training can allow the agent to:
  - Choose better next actions (more optimal tool calls, reasoning steps, responses)
  - Improve reliability and efficiency over repeated interactions
  - Learn from *experience*, not just static data

Levels at which performance may be improved

Level	Object being improved	Typical method	What changes
Model (LLM)	A single neural policy that maps *text* → *text*	Fine-tuning (supervised or RL-based)	Model parameters or adapters
Agent implement.	A *system* that uses one or more LLMs plus tools, memory, control logic	Training loop/RL algorithm (e.g., PPO, GRPO, hierarchical RL)	Which sub-actions or calls the agent chooses, and how it coordinates them
Ecosystem / runtime	The deployed environment where agents act and learn	Experience collection, reward shaping, orchestration	Datasets, trajectories, or policies for multiple agents

Recall: Fine tuning, in brief

Fine tuning refines a pre-trained model's weights on domain-specific or task-specific examples to improve accuracy, style, or reasoning

- Collect (prompt → ideal response) pairs
- Train with gradient descent on supervised or RL objectives to update model weights
- Validate and deploy new model checkpoint

Type	**Purpose**
Supervised fine-tuning (SFT)	Teach format, reasoning, tone
Instruction tuning	Align with human prompts
Domain tuning	Specialize to specific domains
LoRA / PEFT	Lightweight, adapter-based updates

Fine-tuning paradigms

Type	Description	Typical Use
Supervised fine-tuning (SFT)	Train model on labeled examples of desired input→output	Instruction tuning
Reinforcement learning (RLHF/RLAIF)	Optimize model by reward feedback	Alignment
Agent fine-tuning	Optimize entire *agent workflow* using task success signals	Adaptive agents

Fine tuning, RL, agents

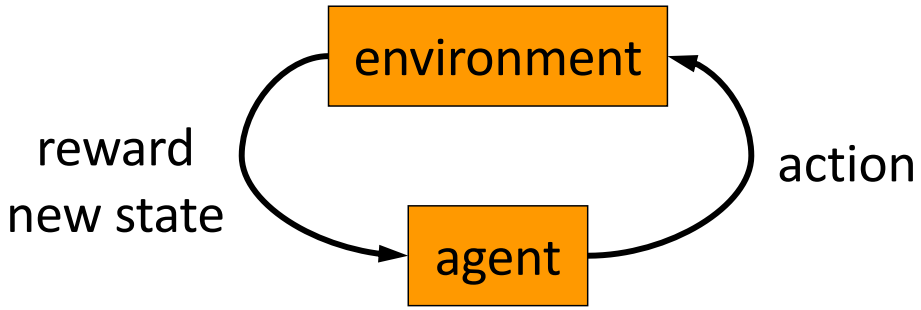
- **Fine-tuning** is a technique that changes the LLM itself.
  - Can be **supervised** (SFT, instruction tuning) or **reinforcement-based** (RLHF, GRPO, PPO)
  - Its output is an improved model checkpoint or adapter
- **Reinforcement Learning** is a *training paradigm* that can operate **inside or around** an agent
  - When the RL algorithm's gradient flows into the LLM weights → that's **RL-based fine-tuning**
  - When RL updates only the *policy logic* (e.g., planner, routing, parameter selection) → it's **agent-level training** without touching model weights.
- **Agent frameworks** (like Agent Lightning or ART) handle the *outer loop*: how experience is gathered, rewards computed, and updates applied.
  - Fine-tuning (of LLM weights) is one possible update target.
  - Prompt optimization, rule tuning, or memory shaping are others.

Reinforcement learning

Reinforcement learning (RL), which has driven recent advances in reasoning models such as DeepSeek- R1 and Kimi k1.5, offers a powerful paradigm for optimizing LLMs in agentic scenarios. **While supervised learning requires detailed step-by-step annotations—which are scarce and costly for complex interactive tasks—RL relies on outcome-based reward signals**. This eliminates the need for task-specific curated data and allows agents to learn desirable behaviors directly from environment feedback across diverse tasks. Moreover, the trial-and-error nature of RL closely mirrors how humans acquire problem-solving skills, enabling models to learn action policies grounded in deployment contexts. This capability opens up the potential for transforming LLM-generated text tokens into real-world actions, making RL a natural fit for training models in agent-based systems.

Reinforcement learning

- Problems involving an agent interacting with an environment, which provides numeric reward signals
- Goal: Learn how to take actions in order to maximize reward



Fine tuning vs. reinforcement learning

When to Fine-Tune	When to Apply RL
You have high-quality labeled data	Labels are unavailable but you can define a *reward* (success metric, correctness, user satisfaction)
You need consistent, static behavior (e.g., summarization style)	You need adaptive, goal-directed behavior (e.g., tool use, planning, dialogue).
Cost of annotation is lower than cost of rollout	Cost of environment interaction is lower than mass labeling
You want fast, repeatable training cycles.	You want continual improvement from real-world feedback

Why fine-tune LLMs?

LLMs trained on static Internet data struggle with real-world, interactive tasks, e.g.:

- Tool-using agents
- Code-executing or debugging agents
- Retrieval-augmented generation (RAG) agents
- Conversational agents in long-horizon interactions
- Scientific or experimental agents
- Game-playing or embodied agents

Why static LLMs struggle in the real world

Domain	Example Failure	Why Static Data Fails
Tool Use (SQL, APIs)	Misformats queries, can't fix execution errors	Never sees real API responses or error messages
Code Agents	Outputs code that fails runtime tests	No reward for successful execution
Retrieval-Augmented QA	Retrieves irrelevant docs; hallucinates	No supervision from retrieval success/failure
Conversational Agents	Breaks down over long dialogs; repeats mistakes	No turn-level feedback or satisfaction signal
Scientific Agents	Suggests infeasible experiments	Never observes outcomes or experiment results
Embodied / Game Agents	Knows rules but can't win	No experience-based learning from rewards

Why fine-tune agents?

LLMs trained on static Internet data struggle with real-world, interactive tasks

- Agents (tool users, planners, retrievers) generate **rich experience traces** unavailable in pretraining
- Real-world improvement loop:

  Deploy → Observe → Reward → Update

Key idea: *Environment provides the missing signal for continual learning*

Tool-using agents: APIs, databases, …

Example: A text-to-SQL assistant like “Generate SQL for this query” works fine on benchmark data, but:

- fails when it encounters an unfamiliar schema or proprietary function (LEFT JOIN inventory vs. JOIN inv_table)
- does not know **when** to retry or **how** to parse an error message from a real database
- cannot adapt to reward signals like *execution success* or *query latency*

Static-data limitation: Training on example pairs does not expose the model to *feedback loops* or *action–outcome dynamics*. It never learns that *syntax errors → penalty, correct execution → reward.*

Code-executing or debugging agents

Example: A code-writing model may output syntactically valid code but repeatedly fail runtime tests (e.g., off-by-one errors, undefined variables). As a static model it has no mechanism to:

- Re-run code, see failures, and adjust strategy
- Prefer code that passes tests over code that merely “looks right”

Reinforcement Learning (RL)-based adaptation can be a solution

- Training from *execution rewards* teaches it to explore, test, and self-correct: the idea behind **DeepSeek-R1** and **Agent Lightning’s multi-step credit assignment**.

Retrieval-augmented generation (RAG) agents

Example: A RAG system answering "What are the latest results on superconducting hydrides?" might:

- Retrieve irrelevant documents due to query mis-formulation
- Produce confident but hallucinated summaries

Static LLMs fail because:

- They're trained to *predict text*, not to *optimize retrieval relevance or factuality*
- They have no gradient signal from "did this retrieval actually help answer the question?"

Conversational agents in long-horizon interactions

- **Example:** A customer-service chatbot can generate fluent single replies, but breaks down when:
  - It needs to maintain consistent memory across 10–20 turns
  - It misinterprets user feedback (e.g., “That didn’t help”)
  - It can’t adapt its strategy after repeated failures
- **Static corpus problem:** No natural signal for *turn-level success*, *conversation satisfaction*, or *task completion*

Scientific or experimental agents

- A chemistry-design agent proposes a synthesis plan that is infeasible when run in the lab, or doesn't adjust when the experiment yields unexpected results
- Static pretraining lacks **closed-loop experience** with experimental outcomes; hence the push toward *real-world fine-tuning* from observed results

Game-playing or embodied agents

- **Example:** An LLM describing “how to play chess” doesn’t learn *to win games*
- Winning requires trial-and-error reward feedback
- This is why systems like **OpenPipe/ART** and **Agent Lightning** treat the agent’s world as an *environment* with rewards

Methods for correcting limitations of static data

Challenge	RL Solution	Example Framework
No feedback loop	Collects environment rewards → updates model	**Agent Lightning**, **ART**
No credit assignment	Decomposes multi-turn traces into transitions	**LightningRL** hierarchical policy
No adaptation	Iteratively improves via rollout → reward → update	**OpenPipe / ART** pipelines
No intermediate signals	Automatic Intermediate Rewarding (AIR)	**Agent Lightning** client runtime
Long-horizon tasks	Policy learning across multiple steps	Hierarchical RL / GRPO / PPO
Static prompts	Optimizes prompts or examples from data	**DSPy** teleprompters

Summary

Challenge	Why static data fails	What RL fixes
Action-dependent outcomes	No feedback loop in text corpora	Reward from environment outcomes
Error recovery	No notion of “try → fail → retry”	Credit assignment over sequences
Long-horizon consistency	Training truncates context	Policy learning across steps
Real-world variation	Internet text ≠ dynamic tools/APIs	Experience-driven adaptation

OpenPipe / ART: Agent Reinforcement Trainer

- **Core concepts**
  - "OpenPipe" is a middleware for reinforcement tuning of agents, providing interfaces for reward collection, logging, and evaluation
  - **ART (Agent Reinforcement Trainer)** abstracts away infrastructure, connecting LLMs, environments, and reward models
  - Supports multi-episode training, rollout–train cycles, and integration with RL frameworks like Hugging Face's TRL or Microsoft's VeRL
- **Key contribution:** makes RL-style fine-tuning *operational* for deployed agents, not just isolated models

Agent Lightning: RL for any agent

- **Core ideas:**
  - **Training–Agent Disaggregation:** separates agent execution (client) from RL training (server)
  - **Unified Data Interface:** every LLM/tool call logged as (input, output, reward) transition.
  - **LightningRL:** hierarchical RL method compatible with PPO/GRPO/REINFORCE++ (no masking, no DAG parsing).
  - **Automatic Intermediate Rewarding (AIR):** converts system telemetry (e.g., tool success) into dense rewards.
- **Example:** training a calculator-using MathQA agent or text-to-SQL workflow; each tool call becomes a transition used for policy optimization.
- **Notes:**

  → Enables fine-tuning **without code modification** of existing agents (LangChain, AutoGen, etc.)

  → Bridges research RL frameworks (like VeRL) with real production agents.

ART vs. Agent Lightning

Aspect	ART / OpenPipe	Agent Lightning
Focus	Training *loop orchestration* and logging	Algorithm + data interface + system integration
Data model	Agent episodes with rewards	Unified `(state, action, reward)` transitions
Flexibility	Works with various RL frameworks	Works with *any* agent architecture
Innovation	Standardized RL infra for agents	Decoupled training–execution + AIR

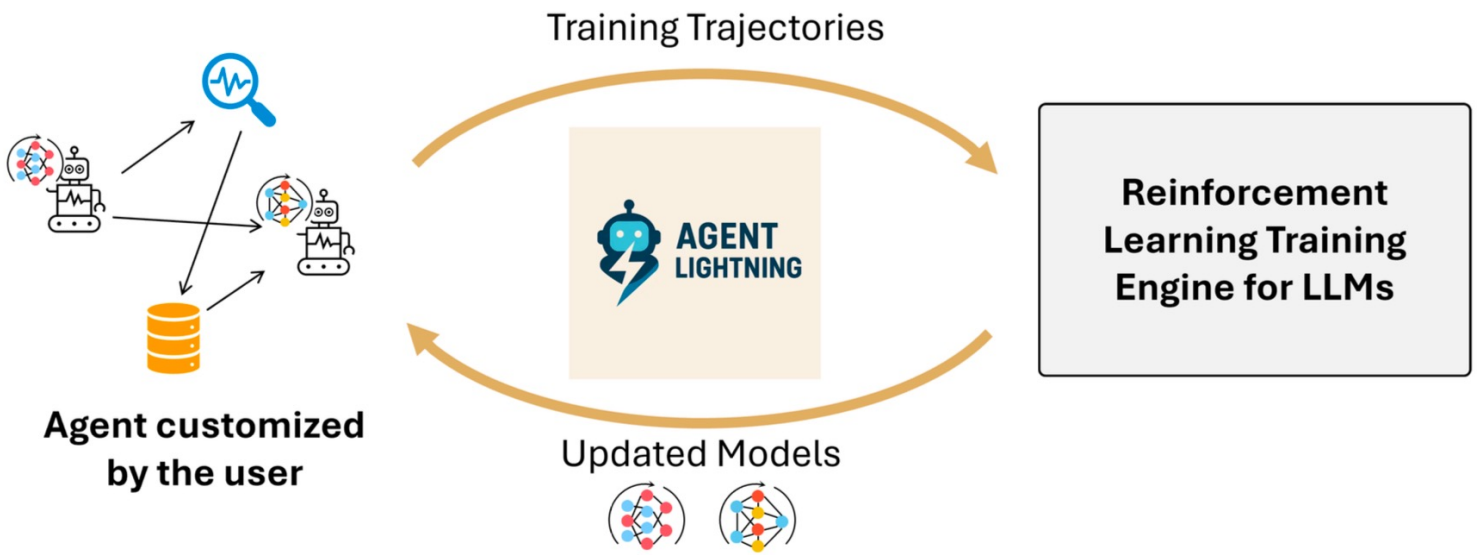


Figure 1: Overview of *Agent Lightning*, a flexible and extensible framework that enables reinforcement learning of LLMs for ANY AI agents.

For example, Math QA

1) Algebraic Manipulation

- **Problem:** If $x = 4$ and $y = 2x - 3$, compute $(x^2 - y^2)/(x - y)$.
- **Solution reasoning:**

$$y = 2(4) - 3 = 5$$
$$x^2 - y^2 = (16 - 25) = -9$$
$$x - y = -1$$

→ Result = 9

2) Geometry / Trigonometry

- **Problem:** A right triangle has sides of lengths 3 and 4. Find sine of larger acute angle
- **Solution reasoning:**

  Hypotenuse = 5

  sin(θ) = opposite/hypotenuse = 4/5 = **0.8**

3) "If $a = \frac{3}{5}$ and $b = 7$, compute $\left(a^{-2} + \sqrt{b}\right) - \frac{12}{3 \cdot 4}$. Return a single number."

Agent Lightning and Math QA

- **Agent implementation:** Single-LLM workflow that (a) plans, (b) issues calculator calls, (c) integrates results, (d) answers
- **Tool:** calculator(expr: string) -> number (stateless).
  AIR (Automatic Intermediate Rewarding) will mark a tool call “valid/invalid” to yield intermediate rewards (format OK, tool executed, syntactically valid)
- **Terminal reward:** exact-match on numeric answer (1.0 if correct, else 0.0)
- **LightningRL** converts each **LLM call** into an **action**, does simple **credit assignment** (same return to each action by default), then applies a single-turn RL loss (e.g., GRPO/PPO) token-wise on each call’s output

Ground-truth answer (for reference)

"If $a = \frac{3}{5}$ and $b = 7$, compute $\left(a^{-2} + \sqrt{b}\right) - \frac{12}{3 \cdot 4}$. Return a single number."

- Compute stepwise (what the **calculator** is for):
  - $a^{-2} = \left(\frac{3}{5}\right)^{-2} = \left(\frac{5}{3}\right)^{2} = \frac{25}{9}$
  - $\sqrt{b} = \sqrt{7}$
  - $\frac{12}{3 \cdot 4} = \frac{12}{12} = 1$
- So the target is:

$$\frac{25}{9} + \sqrt{7} - 1 = \frac{16}{9} + \sqrt{7} \approx 1.777\ldots + 2.64575\ldots \approx 4.42275$$

Agent execution timeline

- **Transition T1 – Plan & Compute (3/5)^(-2)**
  LLM output: {"expr":"(3/5)^(-2)"}
  Tool: 2.7778
  AIR reward ≈ 0.15
- **Transition T2 – Compute sqrt(7)**
  Tool: 2.6458
  AIR reward ≈ 0.15
- **Transition T3 – Compute 12/(3*4)**
  Tool: 1
  AIR reward ≈ 0.15
- **Transition T4 – Combine & Answer**
  LLM: {"answer":"(25/9)+sqrt(7)-1≈4.4228"}
  Terminal reward = 1.0 (correct)
- **Total Return R ≈ 1.45**

Automatic Intermediate Rewarding (AIR) enables the assignment of intermediate rewards to transitions based on system monitoring signals (such as tool call return statuses)

T1 computation in more detail

- Input_1 (state → observation):

  System: You are a math solver. Use the calculator tool for exact arithmetic.

  User: If a=3/5 and b=7, compute (a^-2 + sqrt(b)) - 12/(3*4).

  Respond in JSON with fields: {"plan": "...", "next_action": "compute" | "answer", "expr": "..."}

- LLM Output_1 (action a1):

  {"plan":"Compute a^-2, compute sqrt(7), compute 12/(3*4), then combine.", "next_action":"compute","expr":"(3/5)^(-2)"}

- AIR intermediate reward r_1 (format): +0.05 if JSON parseable & fields present
- AIR intermediate reward r_1 (tool-eligibility): +0.05 if expr is calculator-valid (simple static check)

Unified data trace

- **Logged transitions (simplified JSONL):**

  {"t":1, "input":"UserInput", "output":"(3/5)^(-2)", "reward":0.15}

  {"t":2, "input":"...ToolResult1...", "output":"sqrt(7)", "reward":0.15}

  {"t":3, "input":"...ToolResult2...", "output":"12/(3*4)", "reward":0.15}

  {"t":4, "input":"...ToolResult3...", "output":"Answer≈4.4228", "reward":1.0}

- **Observations**
  - Each LLM call = one action in the Markov decision process
  - No masking, no concatenation: just clean transitions for RL

LightningRL Optimization Flow

- **Credit Assignment:** Assign per-step or uniform return to each transition
- **Token-Level Optimization:** Apply single-turn RL loss (GRPO/ PPO/ REINFORCE++)
- **Batching:** Transitions grouped by task for advantage estimation
- **AIR:** Provides dense shaping rewards to accelerate learning
- **Benefits:**
  - Modular and scalable (no coupling between agent logic & RL engine)
  - Works across AutoGen, LangChain, or custom agents

Policy-gradient RL for language models

- **Policy-gradient** methods optimize a parameterized model $\pi_{theta}(a \mid s)$ (the LLM) to *increase expected reward*. The general objective is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s,a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta\,(a \mid s)\, A(s, a)\right],$$

  where **A(s,a)** ("advantage") measures how much better the sampled action's reward is than average

- In LLM fine-tuning,
  - **state s** = prompt or context
  - **action a** = the generated text
  - **reward r** = numeric score (helpfulness, correctness, etc.)
  - **advantage A** = signal telling the model which responses were better

Policy gradient methods used in Agent Lightning

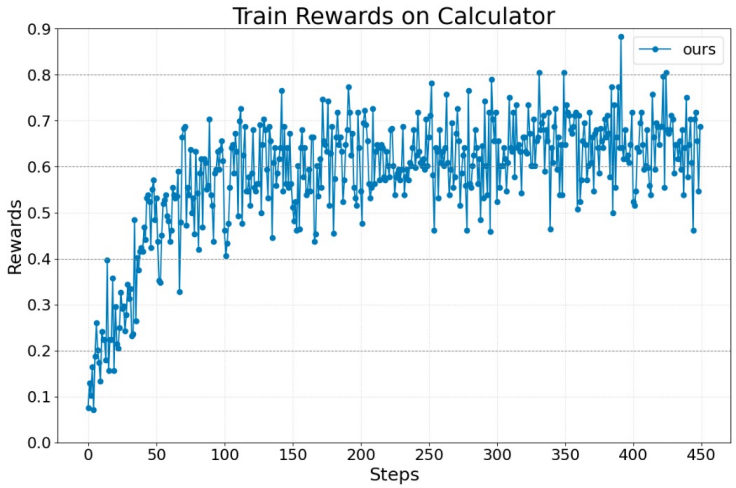
- **Proximal Policy Optimization (PPO):** take conservative updates so new policy does not drift too far from previous
- **Group Relative Policy Optimization (GRPO):** Group several model outputs for same prompt/task, normalize rewards. Default.
- **REINFORCE++:** Simplest; no critic, no grouping

Methods differ only in **how they compute A(s,a)**

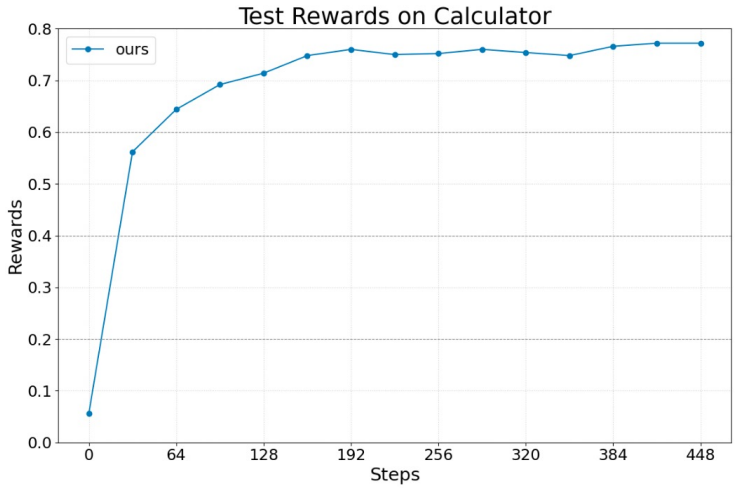
Feature	PPO	GRPO	REINFORCE++
Critic network	yes	none	none
Advantage baseline	Value function	Group mean / std	Batch mean
Stability	Very high	Moderate	Lower
Compute cost	Highest	Medium	Lowest
Use in Agent Lightning	Supported but heavy	Default choice	Optional

Training results

- **Dataset:** Calc-X + **Base Model:** Llama-3.2-3B-Instruct
- Smooth, consistent improvement in both train & test reward curves
- Improved accuracy in symbolic + numeric tasks
- Robust handling of multi-turn reasoning with tool invocations



(a) Train reward



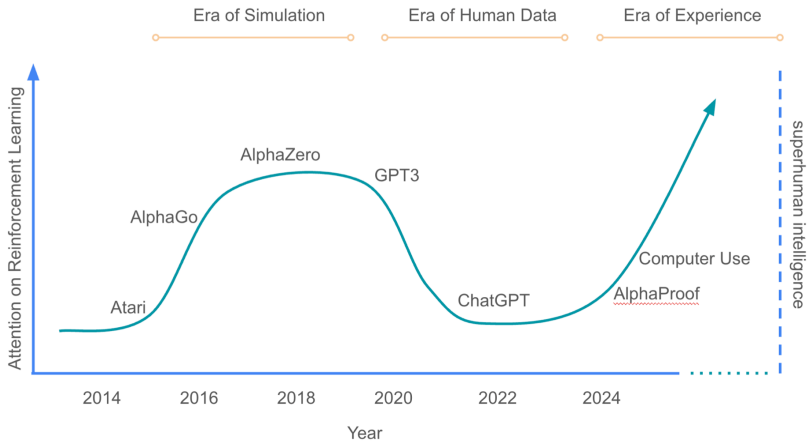
(b) Test reward

Agent Lightning summary

- **Transition-based modeling** enables fine-grained RL on complex workflows
- **Automatic Intermediate Rewarding (AIR)** mitigates sparse reward problem
- **LightningRL** reuses efficient single-turn RL across multi-step, tool-augmented agents
- **Training-Agent Disaggregation** (server ↔ client) allows zero agent code modification
- **Outcome:** Agent Lightning continuously improves tool-using math agents, achieving both reliability and scalability

Welcome to the Era of Experience

David Silver, Richard S. Sutton*



Our contention is that incredible new capabilities will arise once the full potential of experiential learning is harnessed. This era of experience will likely be characterised by agents and environments that, in addition to learning from vast quantities of experiential data, will break through the limitations of human-centric AI systems in several further dimensions:

- Agents will inhabit streams of experience, rather than short snippets of interaction.
- Their actions and observations will be richly grounded in the environment, rather than interacting via human dialogue alone.
- Their rewards will be grounded in their experience of the environment, rather than coming from human prejudgement.
- They will plan and/or reason about experience, rather than reasoning solely in human terms