

CS388: Natural Language Processing

Lecture 14: Reinforcement Learning



TEXAS

The University of Texas at Austin

Administration

- Makeup midterm Tomorrow 14th, 2 to 3pm GDC 3.816
- Project proposals

Today

- What do we use RL for?
- Markov Decision Processes
- Policy gradients
- PPO and NLPO

SFT supervision signal

- SFT provides token-level supervision
 - What is the “correct” next token to produce
- What about tasks where there are many “correct” signals?
- Or tasks for which correctness can only be measured in terms of a whole response?

Example: Reasoning tasks

Example problem (GSM8K)

Natalia sold clips to 48 of her friends in April, and then she sold half as many clips in May. How many clips did Natalia sell altogether in April and May?

Final answer: 72

Checker

Pull out the model's final number and compare it to 72: correct or incorrect.

Many tasks work like this

- **Math:** compare the final number
- **Code:** run the unit tests
- **Puzzles:** check the constraints
- **Proofs:** run a proof checker

Option 1: supervised fine-tuning

Training target: one reference chain

Let's think step-by-step...

In May, Natalia sold $48 / 2 = 24$ clips.

Altogether she sold $48 + 24 = 72$ clips.

72

Loss

$$\mathcal{L}(\theta) = - \sum_t \log \pi_{\theta}(y_t \mid x, y_{<t})$$

Other correct chains the model could write

Half of 48 is 24. $24 + 48 = 72$.

Answer: 72, correct

May = 0.5×48 , so total = $1.5 \times 48 = 72$.

Answer: 72, correct

Why is this a problem?

We train the model to copy one solution token by token, but we only care that it reaches 72.

Other valid solutions get no credit.

Why else?

Generation

Prompt: Write a short story about a robot who learns to paint.

Story A: preferred

Unit 7 had painted the same gray wall for ten years. One morning a child left crayons on the floor. Unit 7 drew a sun, badly, and kept the drawing.

Story B: dispreferred

There was a robot. It wanted to paint. It got some paint and painted a picture. The picture was good. The end.

$$\mathcal{L}(\theta) = - \sum_t \log \pi_{\theta}(y_t \mid x, y_{<t})$$

Why is this a problem?

We raise the probability of story A for this prompt, but have we taught the model anything about writing stories?

We want to reward the outcome

SFT

Supervises

Every token of one reference answer

Training data

Fixed examples

What we want

Supervises

Only whether the final outcome is good

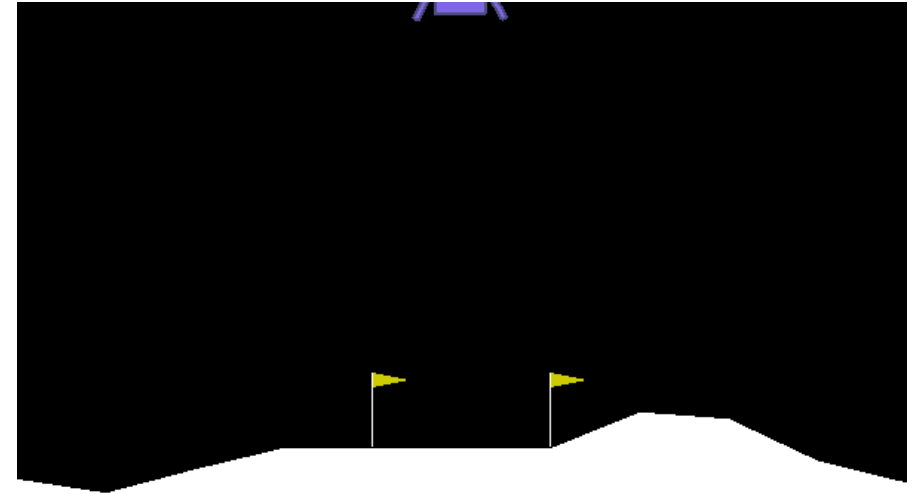
Training data

The model's own attempts, each scored by a reward

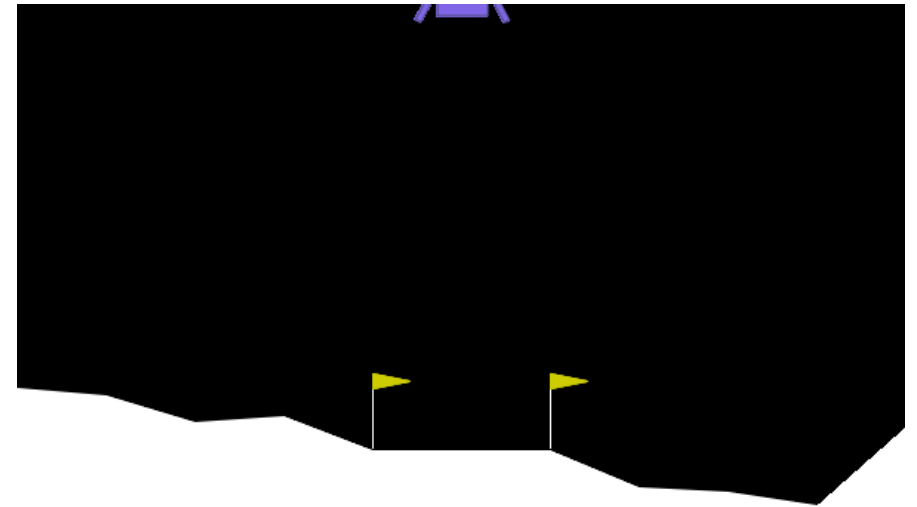
Reinforcement learning lets the model try, scores the outcome, and makes high-reward attempts more likely, without supervising how it got there.

Why RL: learning to move

- RL trains a policy on the **end outcome**
- The agent tries actions and gets a reward
- No demonstrations
- Reward: land the lander
- Complex control behavior emerge from trial and error

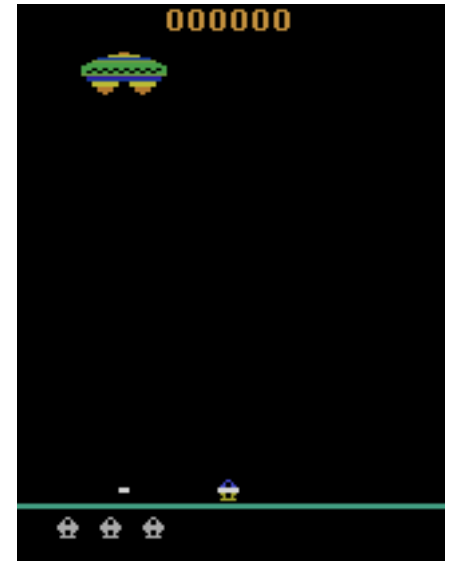
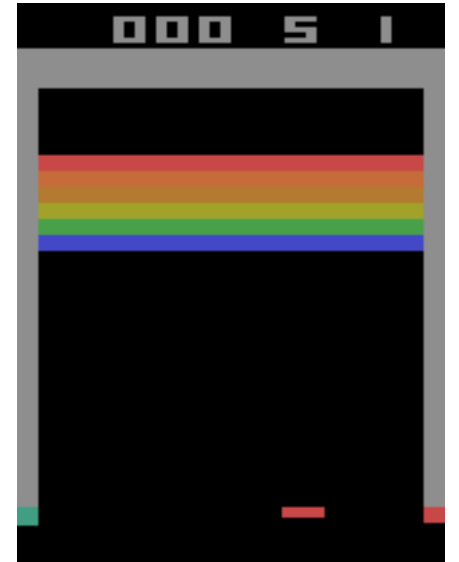


Sum Reward: 0

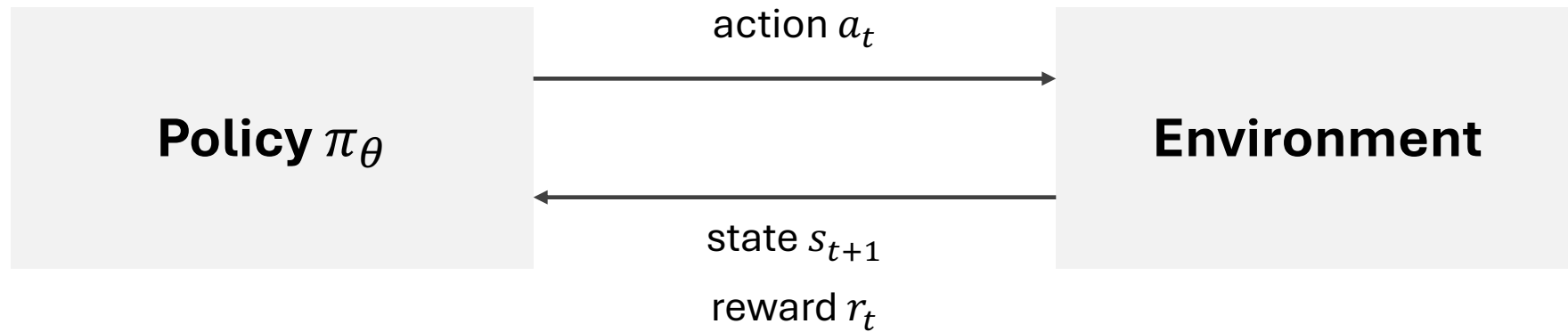


Why RL: playing Atari

- Input: raw pixels
- Reward: the game score
- One algorithm and one set of settings across 49 games
- Human-level play on many of them
- Finds strategies nobody showed it



What is RL



- The policy picks an action; the environment returns a new state and a reward
- Goal: maximize **total reward** over an episode
- For LMs: action = next token, reward = verifier or reward model

Markov Decision Processes

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$:

- States $\mathcal{S}$ and actions $\mathcal{A}$
- Transition $T(s_{t+1} \mid s_t, a_t)$ and reward $r(s_t, a_t)$
- Discount $\gamma \in [0, 1]$
- **Markov**: the next state depends only on the current state and action

A policy $a_t \sim \pi_\theta(\cdot \mid s_t)$ produces a trajectory and its return:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots), \quad R(\tau) = \sum_{t=0}^T \gamma^t r_t$$

MDP example: Atari Breakout

MDP piece	Breakout
State s_t	Last 4 screen frames (raw pixels)
Action a_t	Left, right, fire, or no-op
Transition P	The game engine: ball and paddle physics
Reward r_t	Points for each brick broken
Episode	One game, until all lives are lost
Discount γ	0.99: points now beat points later

Note >1 frame: otherwise you can't see which way the ball moves

MDP example: chess

MDP piece	Chess
State s_t	
Action a_t	
Transition P	
Reward r_t	
Episode	
Discount γ	

The opponent makes transitions random, and reward comes only at the end

Policy gradients

Notation

- RL papers: (s, a) for **states** and **actions**
- LM papers: (x, y) for **prompt** and **completion**
- Many RLHF implementations treat the whole completion as one action
- r_t : per-step reward
- $R(\tau)$: return of a trajectory

Language models as episodic MDPs

MDP concept	Language model
State s_t	Prompt + tokens so far: $(x, y_{<t})$
Action a_t	Next token y_t
Transition	Deterministic: append the token
Policy $\pi_\theta(a_t \mid s_t)$	LM next-token distribution
Episode	One prompt $\rightarrow$ one completion
Terminal reward	Reward model score or verifier output
Discount γ	Usually 1.0 (no discounting)

What are we optimizing?

- Choose θ to maximize reward **on average** under the current policy

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

- Estimate with sampled rollouts:

$$J(\theta) \approx \frac{1}{N} \sum_{i=1}^N R(\tau_i), \quad \tau_i \sim p_\theta$$

- Next: a gradient we can use to optimize this

How a policy gradient works

Make actions more likely when they lead to better outcomes.

$$\Delta\theta \propto \Psi_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

- $\nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$: **which action?** How each parameter changed its probability
- Ψ_t : **how good was it?** A scalar score for the outcome
- $\Psi_t > 0$: make a_t more likely. $\Psi_t < 0$: less likely
- Algorithms differ mainly in how they choose Ψ_t

The policy gradient equation

A preview of where we end up:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim p_{\theta}} \left[\sum_{t=0}^T \Psi_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right]$$

- **Sample** trials in the environment and **estimate** the gradient
- Ψ_t is the **learning signal**: how good was the action?
- The choice of Ψ_t sets variance, bias and compute cost

Key RL quantities

- **Value** $V(s)$: expected future return from state s

$$V(s) = \mathbb{E}[G_t \mid s_t = s], \quad G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

- **Action-value** $Q(s, a)$: expected return after taking a in s

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

- **Advantage** $A(s, a) = Q(s, a) - V(s)$: how much better than average?

What Ψ_t can be

	Ψ_t	Description	Variance	Bias
1	$R(\tau) = \sum_{t=0}^T \gamma^t r_t$	Total trajectory reward	Highest	None
2	$G_t - b(s_t)$	Baselined return	Lower	None
3	$A^\pi(s_t, a_t) = Q - V$	Advantage	Low	None

- **Variance:** how much the gradient estimate swings between batches; high means noisy, sample-hungry learning
- **Bias:** whether the estimate is off on average; biased can converge to the wrong policy
- **Total reward:** unbiased, but every action gets credit for the whole trajectory, so variance is highest
- **Baselined return:** subtracting a baseline cuts variance without changing the expected gradient
- **Advantage:** subtracts the baseline $V(s_t)$, keeping only “better or worse than expected”; low variance
- In practice V is learned, so we trade a little bias for lower variance

Deriving the policy gradient

Setup: differentiating an expectation

- We want $\nabla_{\theta} J(\theta)$, but the sampling distribution depends on θ
- Written as an integral:

$$J(\theta) = \mathbb{E}_{\tau \sim p_{\theta}}[R(\tau)] = \int_{\tau} p_{\theta}(\tau) R(\tau) d\tau$$

- Taking the gradient directly:

$$\nabla_{\theta} J(\theta) = \int_{\tau} \nabla_{\theta} p_{\theta}(\tau) R(\tau) d\tau$$

- We can sample from $p_{\theta}(\tau)$, but not from $\nabla_{\theta} p_{\theta}(\tau)$
- A few tricks let us estimate it anyway

Computing the gradient from trajectories

- Substitute the log-derivative identity (next slide):

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \int_{\tau} \nabla_{\theta} p_{\theta}(\tau) R(\tau) d\tau && \text{differentiate the objective} \\ &= \int_{\tau} p_{\theta}(\tau) R(\tau) \nabla_{\theta} \log p_{\theta}(\tau) d\tau && \text{log-derivative identity} \\ &= \mathbb{E}_{\tau \sim p_{\theta}} [R(\tau) \nabla_{\theta} \log p_{\theta}(\tau)] && \text{expectation form}\end{aligned}$$

- Now we can estimate it with **Monte Carlo sampling**

The log-derivative identity

- The only non-obvious step, from the chain rule for log :

$$\nabla_{\theta} \log p_{\theta}(\tau) = \frac{\nabla_{\theta} p_{\theta}(\tau)}{p_{\theta}(\tau)}$$

$$\Rightarrow \nabla_{\theta} p_{\theta}(\tau) = p_{\theta}(\tau) \nabla_{\theta} \log p_{\theta}(\tau)$$

- Turns the gradient of a probability into a gradient of a **log-probability**
- Log-probabilities are much easier to work with

Where did $p_\theta(\tau)$ go?

- It became the **sampling distribution** of the expectation:

$$\int p_\theta(\tau) f(\tau) d\tau = \mathbb{E}_{\tau \sim p_\theta}[f(\tau)]$$

- Monte Carlo estimates it by sampling from the policy:

$$\mathbb{E}_{\tau \sim p_\theta}[f(\tau)] \approx \frac{1}{N} \sum_{i=1}^N f(\tau_i), \quad \tau_i \sim p_\theta$$

- No explicit $p_\theta(\tau_i)$ in code: likely trajectories just **appear more often**
- Sample via rollouts
- Loss computes $R(\tau_i) \nabla_\theta \log p_\theta(\tau_i)$

Expanding the log-probability of a trajectory

$$\log p_{\theta}(\tau) = \log d_0(s_0) + \sum_{t=0}^T \log \pi_{\theta}(a_t \mid s_t) + \sum_{t=0}^T \log p(s_{t+1} \mid s_t, a_t)$$

- Initial state and environment dynamics don't depend on θ : their gradients are 0
- Only the policy terms survive:

$$\nabla_{\theta} \log p_{\theta}(\tau) = \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t)$$

- For LMs: sequence log-prob = **sum of token log-probs**

```
seq_log_probs = (token_log_probs * completion_mask).sum(dim=-1)
loss = -(seq_log_probs * advantages).mean()
```

From full return to future rewards

- An action at time t can't affect past rewards
- So instead of weighting by the full return $R(\tau)$:

$$\nabla_{\theta} J = \mathbb{E}_{\tau \sim p_{\theta}} \left[\sum_{t=0}^T R(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right]$$

- Weight by the **return-to-go** $G_t = \sum_{t'=t}^T r_{t'}$:

$$\nabla_{\theta} J = \mathbb{E}_{\tau \sim p_{\theta}} \left[\sum_{t=0}^T G_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right]$$

- Same expected gradient, less noise

Baselines keep the estimator unbiased

- Raw returns are noisy; a baseline **centers** them: better or worse than expected?
- Use $\Psi_t = G_t - b(s_t)$, where b depends only on the state
- The extra term has zero expected gradient:

$$\mathbb{E}[b(s)\nabla_{\theta}\log\pi_{\theta}(a \mid s)] = b(s) \sum_a \nabla_{\theta}\pi_{\theta}(a \mid s) = b(s)\nabla_{\theta} \underbrace{\sum_a \pi_{\theta}(a \mid s)}_{=1} = 0$$

- Any function of state works as a baseline

The policy gradient estimator

Log-derivative trick + return-to-go + baseline:

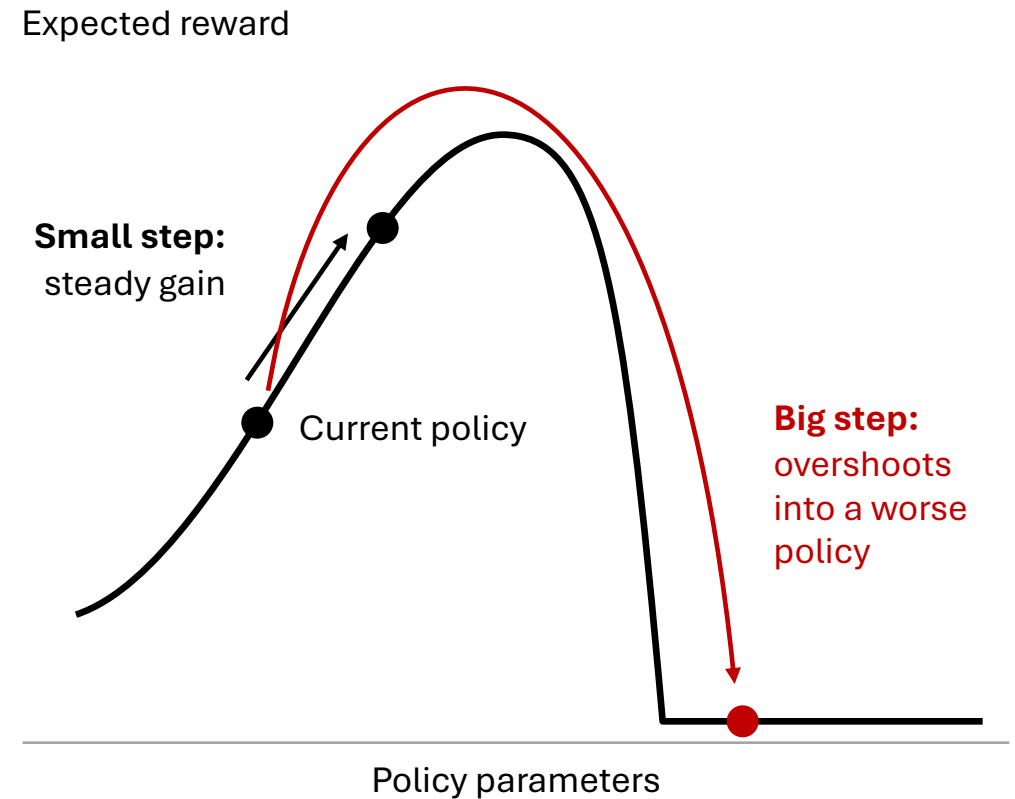
$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim p_{\theta}} \left[\sum_{t=0}^T \Psi_t \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t) \right]$$

- The **policy gradient theorem**
- Every algorithm today picks a Ψ_t and a regularizer
- For LMs: sum over generated tokens, each weighted by Ψ_t

PPO

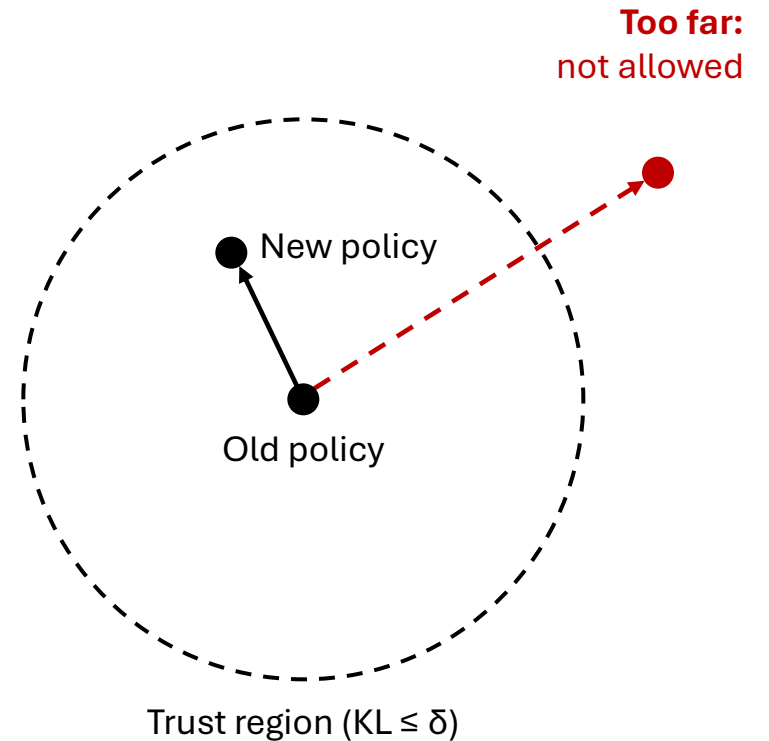
What's wrong with vanilla policy gradients

- **One step per batch:** sample rollouts, take one gradient step, throw them away
 - Why: after one step, the policy is no longer the one that generated the rollouts
- Rollouts are expensive, especially LM generation
- **Step size is hard to pick** (noisy gradient)
 - Too small: slow learning
 - Too big: policy gets worse, and the next rollouts come from that worse policy
- Want: **many steps per batch**, without moving too far



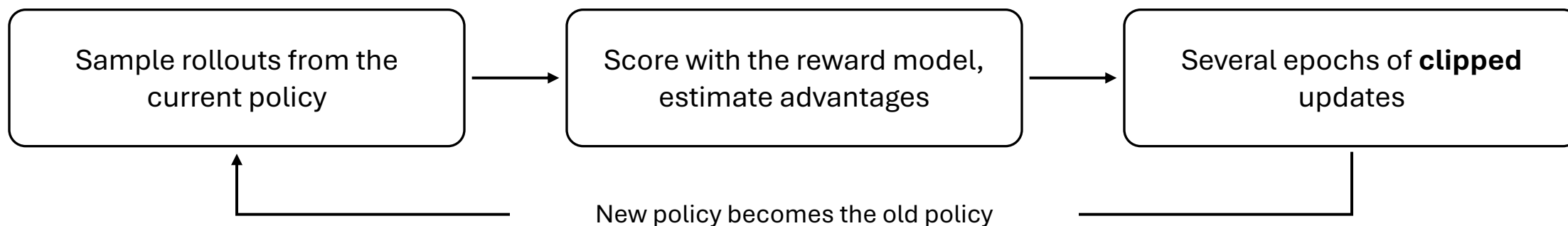
Surrogate objective and TRPO

- **Surrogate objective:** estimate how good a new policy is, using samples from the old one
- **Importance ratio:** $\pi_{\text{new}}(a | s) / \pi_{\text{old}}(a | s)$
 - How much more likely the action is now than when sampled
- Surrogate $\approx$ ratio $\times$ advantage
- Only accurate **near the old policy**
- **TRPO:** maximize the surrogate, but keep the new policy inside a KL “trust region”
 - Works, but needs expensive second-order optimization



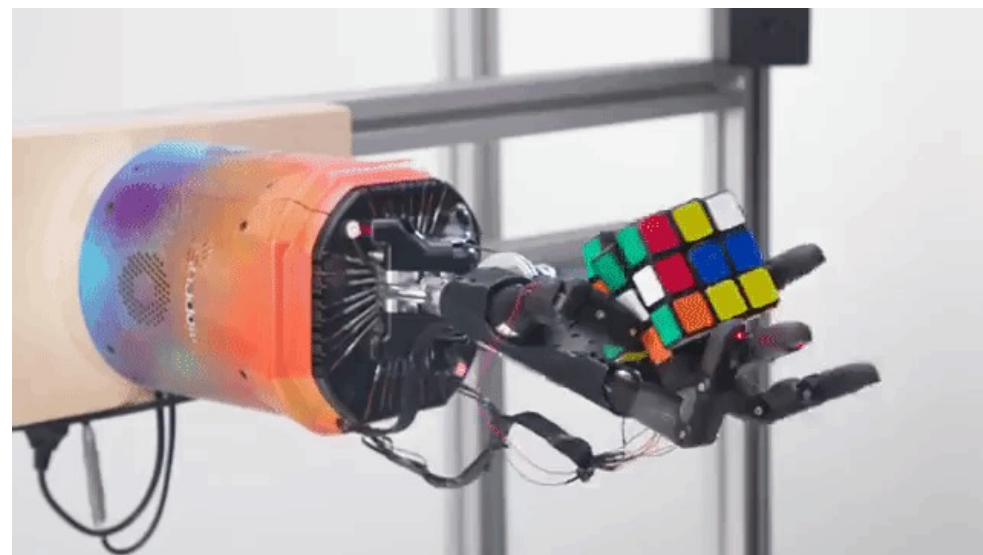
PPO: the same idea with a simple clip

- Same surrogate (ratio $\times$ advantage), but **no KL constraint**
- **Clip the ratio** to about 0.8–1.2: once an action's probability has moved $\sim 20\%$, stop pushing it
- Plain SGD/Adam, with **several epochs on each batch**



PPO is a standard RL algorithm in practice

- Simple, general-purpose, works across many domains
- **Early RLHF for LLMs**
 - InstructGPT (OpenAI, 2022)
 - Anthropic's helpful and harmless assistant (2022)
 - Llama 2-Chat (Meta, 2023)
- **Beyond language**
 - OpenAI Five: beat the Dota 2 world champions (2019)
 - Robot hand solving a Rubik's cube (OpenAI, 2019)
- **Still in use for LLMs**
 - RL with verifiable rewards, e.g. Tülu 3 (Ai2, 2024)
 - GRPO (DeepSeek) is a PPO variant without a value model



Takeaways

- RL trains on the **outcome**, not one reference answer
- Generation is an MDP: state = prefix, action = next token
- Policy gradient: raise log-probs in proportion to Ψ_t
- Baselines and advantages cut variance without bias
- PPO reuses samples and clips the ratio
- NLPO masks the vocabulary to make exploration manageable