CS498: Algorithmic Engineering

Lecture 29: Reasoning RL & The Frontier

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 16

Outline

Where We Left Off

Last lecture we trained a model to follow instructions with SFT.

Where We Left Off

Last lecture we trained a model to follow instructions with SFT.

SFT works by imitating training examples. The model learns to produce responses that look like the examples it was trained on.

Where We Left Off

Last lecture we trained a model to follow instructions with SFT.

SFT works by imitating training examples. The model learns to produce responses that look like the examples it was trained on.

But imitation has a ceiling: the model can never be better than the data it imitates.

Where We Left Off

Last lecture we trained a model to follow instructions with SFT.

SFT works by imitating training examples. The model learns to produce responses that look like the examples it was trained on.

But imitation has a ceiling: the model can never be better than the data it imitates.

Expert iteration (STaR) pushes past this by filtering for correct solutions. But the filter is binary: correct or not. A solution that's 90% right gets the same score as gibberish.

Where We Left Off

Last lecture we trained a model to follow instructions with SFT.

SFT works by imitating training examples. The model learns to produce responses that look like the examples it was trained on.

But imitation has a ceiling: the model can never be better than the data it imitates.

Expert iteration (STaR) pushes past this by filtering for correct solutions. But the filter is binary: correct or not. A solution that's 90% right gets the same score as gibberish.

What if we could give the model a richer signal? Not just "right or wrong," but "how good is this compared to other attempts?"

Where We Left Off

Last lecture we trained a model to follow instructions with SFT.

SFT works by imitating training examples. The model learns to produce responses that look like the examples it was trained on.

But imitation has a ceiling: the model can never be better than the data it imitates.

Expert iteration (STaR) pushes past this by filtering for correct solutions. But the filter is binary: correct or not. A solution that's 90% right gets the same score as gibberish.

What if we could give the model a richer signal? Not just "right or wrong," but "how good is this compared to other attempts?"

That's reinforcement learning.

A Motivating Example

Problem: “What is 7×8?”

A Motivating Example

Problem: “What is 7×8?”

We ask the model to generate 4 responses:

A Motivating Example

Problem: "What is 7×8?"

We ask the model to generate 4 responses:

Response	Answer	Correct?
"$7 \times 8 = 56$"	56	Yes
"Let me think... $7 \times 8 = 54$"	54	No
"56"	56	Yes
"I'm not sure, maybe 48?"	48	No

A Motivating Example

Problem: "What is 7×8?"

We ask the model to generate 4 responses:

Response	Answer	Correct?
"$7 \times 8 = 56$"	56	Yes
"Let me think... $7 \times 8 = 54$"	54	No
"56"	56	Yes
"I'm not sure, maybe 48?"	48	No

With SFT, we'd keep the two correct ones and retrain. We learn nothing from the wrong ones.

A Motivating Example

Problem: "What is 7×8?"

We ask the model to generate 4 responses:

Response	Answer	Correct?
"$7 \times 8 = 56$"	56	Yes
"Let me think... $7 \times 8 = 54$"	54	No
"56"	56	Yes
"I'm not sure, maybe 48?"	48	No

With SFT, we'd keep the two correct ones and retrain. We learn nothing from the wrong ones.

With RL, we can do better: push up the probability of correct responses, push down the probability of wrong ones. All four responses contribute to learning.

The RL Setup for Language Models

Let's define the problem precisely.

The RL Setup for Language Models

Let's define the problem precisely.

Policy π_θ: the language model. Given a prompt, it generates a response by sampling tokens.

The RL Setup for Language Models

Let's define the problem precisely.

Policy π_θ: the language model. Given a prompt, it generates a response by sampling tokens.

Reward R: a score for the complete response. For math:

$$\mathrm{R}(\text{response}) = \begin{cases} 1 & \text{if final answer is correct} \\ 0 & \text{otherwise} \end{cases}$$

The RL Setup for Language Models

Let's define the problem precisely.

Policy π_θ: the language model. Given a prompt, it generates a response by sampling tokens.

Reward R: a score for the complete response. For math:

$$\mathrm{R}(\text{response}) = \begin{cases} 1 & \text{if final answer is correct} \\ 0 & \text{otherwise} \end{cases}$$

Goal: adjust θ so that π_θ produces responses that get higher reward.

The RL Setup for Language Models

Let's define the problem precisely.

Policy π_θ: the language model. Given a prompt, it generates a response by sampling tokens.

Reward R: a score for the complete response. For math:

$$\mathrm{R}(\text{response}) = \begin{cases} 1 & \text{if final answer is correct} \\ 0 & \text{otherwise} \end{cases}$$

Goal: adjust θ so that π_θ produces responses that get higher reward.

This is a bandit problem: one action (generate a full response), one reward. No sequential decisions within the response.

The Collapse Problem

If we just maximize reward, the model finds one answer format that works and repeats it forever.

The Collapse Problem

If we just maximize reward, the model finds one answer format that works and repeats it forever.

For "What is 7×8?" it might always output "56" with no reasoning. That gets reward 1. But the model loses the ability to show work, explain, or handle novel questions.

The Collapse Problem

If we just maximize reward, the model finds one answer format that works and repeats it forever.

For "What is 7×8?" it might always output "56" with no reasoning. That gets reward 1. But the model loses the ability to show work, explain, or handle novel questions.

Fix: add a penalty for drifting too far from the SFT model:

The Collapse Problem

If we just maximize reward, the model finds one answer format that works and repeats it forever.

For "What is 7×8?" it might always output "56" with no reasoning. That gets reward 1. But the model loses the ability to show work, explain, or handle novel questions.

Fix: add a penalty for drifting too far from the SFT model:

$$\mathcal{J}(\theta) = \underbrace{\mathbb{E}_{\pi_\theta}[\mathrm{R}]}_{\text{reward}} - \underbrace{\beta \cdot \mathrm{KL}(\pi_\theta \| \pi_{\text{ref}})}_{\text{stay close to SFT}}$$

The Collapse Problem

If we just maximize reward, the model finds one answer format that works and repeats it forever.

For "What is 7×8?" it might always output "56" with no reasoning. That gets reward 1. But the model loses the ability to show work, explain, or handle novel questions.

Fix: add a penalty for drifting too far from the SFT model:

$$\mathcal{J}(\theta) = \underbrace{\mathbb{E}_{\pi_\theta}[\mathrm{R}]}_{\text{reward}} - \underbrace{\beta \cdot \mathrm{KL}(\pi_\theta \| \pi_{\text{ref}})}_{\text{stay close to SFT}}$$

π_{ref} is the SFT checkpoint (frozen). β controls the tradeoff: high β means "stay close," low β means "explore more."

The Collapse Problem

If we just maximize reward, the model finds one answer format that works and repeats it forever.

For "What is 7×8?" it might always output "56" with no reasoning. That gets reward 1. But the model loses the ability to show work, explain, or handle novel questions.

Fix: add a penalty for drifting too far from the SFT model:

$$\mathcal{J}(\theta) = \underbrace{\mathbb{E}_{\pi_\theta}[\mathrm{R}]}_{\text{reward}} - \underbrace{\beta \cdot \mathrm{KL}(\pi_\theta \| \pi_{\text{ref}})}_{\text{stay close to SFT}}$$

π_{ref} is the SFT checkpoint (frozen). β controls the tradeoff: high β means "stay close," low β means "explore more."

Maximize reward, but don't forget what you already know.

The Problem: Gradients Through Sampling

We want to maximize $\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta}[\mathrm{R}]$. We need $\nabla_\theta \mathcal{J}$.

The Problem: Gradients Through Sampling

We want to maximize $\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta}[\mathrm{R}]$. We need $\nabla_\theta \mathcal{J}$.

If the response were a continuous function of θ, we'd just backpropagate. But generation involves discrete sampling (picking token IDs), and you can't differentiate through sampling (in general).

The Problem: Gradients Through Sampling

We want to maximize $\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta}[\mathrm{R}]$. We need $\nabla_\theta \mathcal{J}$.

If the response were a continuous function of θ, we'd just backpropagate. But generation involves discrete sampling (picking token IDs), and you can't differentiate through sampling (in general).

Let's write out what $\mathbb{E}_{\pi_\theta}[\mathrm{R}]$ means:

The Problem: Gradients Through Sampling

We want to maximize $\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta}[\mathrm{R}]$. We need $\nabla_\theta \mathcal{J}$.

If the response were a continuous function of θ, we'd just backpropagate. But generation involves discrete sampling (picking token IDs), and you can't differentiate through sampling (in general).

Let's write out what $\mathbb{E}_{\pi_\theta}[\mathrm{R}]$ means:

$$\mathcal{J}(\theta) = \sum_{\mathrm{o}} \pi_\theta(\mathrm{o} \mid \mathrm{q}) \cdot \mathrm{R}(\mathrm{o})$$

where the sum is over all possible responses o to prompt q.

The Problem: Gradients Through Sampling

We want to maximize $\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta}[\mathrm{R}]$. We need $\nabla_\theta \mathcal{J}$.

If the response were a continuous function of θ, we'd just backpropagate. But generation involves discrete sampling (picking token IDs), and you can't differentiate through sampling (in general).

Let's write out what $\mathbb{E}_{\pi_\theta}[\mathrm{R}]$ means:

$$\mathcal{J}(\theta) = \sum_{\mathrm{o}} \pi_\theta(\mathrm{o} \mid \mathrm{q}) \cdot \mathrm{R}(\mathrm{o})$$

where the sum is over all possible responses o to prompt q.

This sum has $|\mathrm{vocab}|^{\mathrm{T}}$ terms (every possible sequence of T tokens). We can't enumerate it. But we can estimate the gradient.

The Log-Derivative Trick

Take the gradient of $\mathcal{J}$:

The Log-Derivative Trick

Take the gradient of $\mathcal{J}$:

$$\nabla_\theta \mathcal{J} = \sum_{\mathrm{o}} \nabla_\theta \big[\pi_\theta(\mathrm{o}) \cdot \mathrm{R}(\mathrm{o})\big] = \sum_{\mathrm{o}} \mathrm{R}(\mathrm{o}) \cdot \nabla_\theta \pi_\theta(\mathrm{o})$$

The Log-Derivative Trick

Take the gradient of $\mathcal{J}$:

$$\nabla_\theta \mathcal{J} = \sum_{\mathrm{o}} \nabla_\theta \big[\pi_\theta(\mathrm{o}) \cdot \mathrm{R}(\mathrm{o})\big] = \sum_{\mathrm{o}} \mathrm{R}(\mathrm{o}) \cdot \nabla_\theta \pi_\theta(\mathrm{o})$$

Now the trick. Multiply and divide by $\pi_\theta(\mathrm{o})$:

The Log-Derivative Trick

Take the gradient of $\mathcal{J}$:

$$\nabla_\theta \mathcal{J} = \sum_{\mathrm{o}} \nabla_\theta \big[\pi_\theta(\mathrm{o}) \cdot \mathrm{R}(\mathrm{o})\big] = \sum_{\mathrm{o}} \mathrm{R}(\mathrm{o}) \cdot \nabla_\theta \pi_\theta(\mathrm{o})$$

Now the trick. Multiply and divide by $\pi_\theta(\mathrm{o})$:

$$= \sum_{\mathrm{o}} \mathrm{R}(\mathrm{o}) \cdot \pi_\theta(\mathrm{o}) \cdot \frac{\nabla_\theta \pi_\theta(\mathrm{o})}{\pi_\theta(\mathrm{o})}$$

The Log-Derivative Trick

Take the gradient of $\mathcal{J}$:

$$\nabla_\theta \mathcal{J} = \sum_{\mathrm{o}} \nabla_\theta \big[\pi_\theta(\mathrm{o}) \cdot \mathrm{R}(\mathrm{o})\big] = \sum_{\mathrm{o}} \mathrm{R}(\mathrm{o}) \cdot \nabla_\theta \pi_\theta(\mathrm{o})$$

Now the trick. Multiply and divide by $\pi_\theta(\mathrm{o})$:

$$= \sum_{\mathrm{o}} \mathrm{R}(\mathrm{o}) \cdot \pi_\theta(\mathrm{o}) \cdot \frac{\nabla_\theta \pi_\theta(\mathrm{o})}{\pi_\theta(\mathrm{o})}$$

Recall that $\nabla_\theta \log \mathrm{f}(\theta) = \frac{\nabla_\theta \mathrm{f}(\theta)}{\mathrm{f}(\theta)}$. So:

The Log-Derivative Trick

Take the gradient of $\mathcal{J}$:

$$\nabla_\theta \mathcal{J} = \sum_{\mathrm{o}} \nabla_\theta \big[\pi_\theta(\mathrm{o}) \cdot \mathrm{R}(\mathrm{o})\big] = \sum_{\mathrm{o}} \mathrm{R}(\mathrm{o}) \cdot \nabla_\theta \pi_\theta(\mathrm{o})$$

Now the trick. Multiply and divide by $\pi_\theta(\mathrm{o})$:

$$= \sum_{\mathrm{o}} \mathrm{R}(\mathrm{o}) \cdot \pi_\theta(\mathrm{o}) \cdot \frac{\nabla_\theta \pi_\theta(\mathrm{o})}{\pi_\theta(\mathrm{o})}$$

Recall that $\nabla_\theta \log \mathrm{f}(\theta) = \frac{\nabla_\theta \mathrm{f}(\theta)}{\mathrm{f}(\theta)}$. So:

$$= \sum_{\mathrm{o}} \pi_\theta(\mathrm{o}) \cdot \mathrm{R}(\mathrm{o}) \cdot \nabla_\theta \log \pi_\theta(\mathrm{o}) = \mathbb{E}_{\pi_\theta}\big[\mathrm{R}(\mathrm{o}) \cdot \nabla_\theta \log \pi_\theta(\mathrm{o})\big]$$

The Log-Derivative Trick

Take the gradient of $\mathcal{J}$:

$$\nabla_\theta \mathcal{J} = \sum_{\mathrm{o}} \nabla_\theta \big[\pi_\theta(\mathrm{o}) \cdot \mathrm{R}(\mathrm{o})\big] = \sum_{\mathrm{o}} \mathrm{R}(\mathrm{o}) \cdot \nabla_\theta \pi_\theta(\mathrm{o})$$

Now the trick. Multiply and divide by $\pi_\theta(\mathrm{o})$:

$$= \sum_{\mathrm{o}} \mathrm{R}(\mathrm{o}) \cdot \pi_\theta(\mathrm{o}) \cdot \frac{\nabla_\theta \pi_\theta(\mathrm{o})}{\pi_\theta(\mathrm{o})}$$

Recall that $\nabla_\theta \log \mathrm{f}(\theta) = \frac{\nabla_\theta \mathrm{f}(\theta)}{\mathrm{f}(\theta)}$. So:

$$= \sum_{\mathrm{o}} \pi_\theta(\mathrm{o}) \cdot \mathrm{R}(\mathrm{o}) \cdot \nabla_\theta \log \pi_\theta(\mathrm{o}) = \mathbb{E}_{\pi_\theta}\big[\mathrm{R}(\mathrm{o}) \cdot \nabla_\theta \log \pi_\theta(\mathrm{o})\big]$$

An expectation! We can estimate it by sampling.

REINFORCE

The log-derivative trick gives us:

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{\pi_\theta}\left[\mathrm{R} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o} \mid \mathrm{q})\right]$$

REINFORCE

The log-derivative trick gives us:

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{\pi_\theta}\left[\mathrm{R} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o} \mid \mathrm{q})\right]$$

To estimate this, sample N responses and average:

REINFORCE

The log-derivative trick gives us:

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{\pi_\theta}\left[\mathrm{R} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o} \mid \mathrm{q})\right]$$

To estimate this, sample N responses and average:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{R_i} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o_i} \mid \mathrm{q})$$

REINFORCE

The log-derivative trick gives us:

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{\pi_\theta}\left[\mathrm{R} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o} \mid \mathrm{q})\right]$$

To estimate this, sample N responses and average:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{R_i} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o_i} \mid \mathrm{q})$$

In words:

REINFORCE

The log-derivative trick gives us:

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{\pi_\theta}\left[\mathrm{R} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o} \mid \mathrm{q})\right]$$

To estimate this, sample N responses and average:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{R}_\mathrm{i} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o}_\mathrm{i} \mid \mathrm{q})$$

In words:

1. Sample a response o_i from the model.

REINFORCE

The log-derivative trick gives us:

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{\pi_\theta}\left[\mathrm{R} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o} \mid \mathrm{q})\right]$$

To estimate this, sample N responses and average:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{R_i} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o_i} \mid \mathrm{q})$$

In words:

1. Sample a response $\mathrm{o_i}$ from the model.
2. Compute its reward $\mathrm{R_i}$.

REINFORCE

The log-derivative trick gives us:

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{\pi_\theta}\left[\mathrm{R} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o} \mid \mathrm{q})\right]$$

To estimate this, sample N responses and average:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{R_i} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o_i} \mid \mathrm{q})$$

In words:

1. Sample a response $\mathrm{o_i}$ from the model.
2. Compute its reward $\mathrm{R_i}$.
3. Compute $\nabla_\theta \log \pi_\theta(\mathrm{o_i} \mid \mathrm{q})$. This is just the usual cross-entropy gradient from the forward pass. PyTorch gives us this for free.

REINFORCE

The log-derivative trick gives us:

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{\pi_\theta}\left[\mathrm{R} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o} \mid \mathrm{q})\right]$$

To estimate this, sample N responses and average:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{R_i} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o_i} \mid \mathrm{q})$$

In words:

1. Sample a response $\mathrm{o_i}$ from the model.
2. Compute its reward $\mathrm{R_i}$.
3. Compute $\nabla_\theta \log \pi_\theta(\mathrm{o_i} \mid \mathrm{q})$. This is just the usual cross-entropy gradient from the forward pass. PyTorch gives us this for free.
4. Scale by $\mathrm{R_i}$ and step.

REINFORCE

The log-derivative trick gives us:

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{\pi_\theta}\left[\mathrm{R} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o} \mid \mathrm{q})\right]$$

To estimate this, sample N responses and average:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{R_i} \cdot \nabla_\theta \log \pi_\theta(\mathrm{o_i} \mid \mathrm{q})$$

In words:

1. Sample a response $\mathrm{o_i}$ from the model.
2. Compute its reward $\mathrm{R_i}$.
3. Compute $\nabla_\theta \log \pi_\theta(\mathrm{o_i} \mid \mathrm{q})$. This is just the usual cross-entropy gradient from the forward pass. PyTorch gives us this for free.
4. Scale by $\mathrm{R_i}$ and step.

This is REINFORCE (Williams, 1992).

What Is $\log \pi_\theta(o \mid q)$?

A response o is a sequence of tokens $t_1, t_2, \ldots, t_T$. By the chain rule:

What Is $\log \pi_\theta(o \mid q)$?

A response o is a sequence of tokens $t_1, t_2, \ldots, t_T$. By the chain rule:

$$\pi_\theta(o \mid q) = \prod_{j=1}^{T} \pi_\theta(t_j \mid q, t_1, \ldots, t_{j-1})$$

What Is $\log \pi_\theta(o \mid q)$?

A response o is a sequence of tokens $t_1, t_2, \ldots, t_T$. By the chain rule:

$$\pi_\theta(o \mid q) = \prod_{j=1}^{T} \pi_\theta(t_j \mid q, t_1, \ldots, t_{j-1})$$

Taking the log:

What Is $\log \pi_\theta(o \mid q)$?

A response o is a sequence of tokens $t_1, t_2, \ldots, t_T$. By the chain rule:

$$\pi_\theta(o \mid q) = \prod_{j=1}^{T} \pi_\theta(t_j \mid q, t_1, \ldots, t_{j-1})$$

Taking the log:

$$\log \pi_\theta(o \mid q) = \sum_{j=1}^{T} \log \pi_\theta(t_j \mid q, t_1, \ldots, t_{j-1})$$

What Is $\log \pi_\theta(o \mid q)$?

A response o is a sequence of tokens $t_1, t_2, \ldots, t_T$. By the chain rule:

$$\pi_\theta(o \mid q) = \prod_{j=1}^{T} \pi_\theta(t_j \mid q, t_1, \ldots, t_{j-1})$$

Taking the log:

$$\log \pi_\theta(o \mid q) = \sum_{j=1}^{T} \log \pi_\theta(t_j \mid q, t_1, \ldots, t_{j-1})$$

Each term $\log \pi_\theta(t_j \mid \ldots)$ is the log-probability that the model assigns to the token it actually generated at position j.

What Is $\log \pi_\theta(o \mid q)$?

A response o is a sequence of tokens $t_1, t_2, \ldots, t_T$. By the chain rule:

$$\pi_\theta(o \mid q) = \prod_{j=1}^{T} \pi_\theta(t_j \mid q, t_1, \ldots, t_{j-1})$$

Taking the log:

$$\log \pi_\theta(o \mid q) = \sum_{j=1}^{T} \log \pi_\theta(t_j \mid q, t_1, \ldots, t_{j-1})$$

Each term $\log \pi_\theta(t_j \mid \ldots)$ is the log-probability that the model assigns to the token it actually generated at position j.

This is exactly the same quantity as the cross-entropy loss from training (L26). We already know how to compute it. The only difference: we multiply by R instead of averaging.

REINFORCE: Back to Our Example

We sampled 4 responses to “What is 7×8?”

REINFORCE: Back to Our Example

We sampled 4 responses to "What is 7×8?"

Response	R	Gradient direction
"$7 \times 8 = 56$"	1	$+1 \cdot \nabla \log \pi$ (push up)
"Let me think... 54"	0	$0 \cdot \nabla \log \pi$ (no change)
"56"	1	$+1 \cdot \nabla \log \pi$ (push up)
"Maybe 48?"	0	$0 \cdot \nabla \log \pi$ (no change)

REINFORCE: Back to Our Example

We sampled 4 responses to "What is 7×8?"

Response	R	Gradient direction
"$7 \times 8 = 56$"	1	$+1 \cdot \nabla \log \pi$ (push up)
"Let me think... 54"	0	$0 \cdot \nabla \log \pi$ (no change)
"56"	1	$+1 \cdot \nabla \log \pi$ (push up)
"Maybe 48?"	0	$0 \cdot \nabla \log \pi$ (no change)

Half the samples contribute nothing to the gradient. We only learn from successes.

REINFORCE: Back to Our Example

We sampled 4 responses to "What is 7×8?"

Response	R	Gradient direction
"$7 \times 8 = 56$"	1	$+1 \cdot \nabla \log \pi$ (push up)
"Let me think... 54"	0	$0 \cdot \nabla \log \pi$ (no change)
"56"	1	$+1 \cdot \nabla \log \pi$ (push up)
"Maybe 48?"	0	$0 \cdot \nabla \log \pi$ (no change)

Half the samples contribute nothing to the gradient. We only learn from successes.

And the gradient is noisy: depending on which responses we happened to sample, the update could point in very different directions.

REINFORCE: Back to Our Example

We sampled 4 responses to "What is 7×8?"

Response	R	Gradient direction
"$7 \times 8 = 56$"	1	$+1 \cdot \nabla \log \pi$ (push up)
"Let me think... 54"	0	$0 \cdot \nabla \log \pi$ (no change)
"56"	1	$+1 \cdot \nabla \log \pi$ (push up)
"Maybe 48?"	0	$0 \cdot \nabla \log \pi$ (no change)

Half the samples contribute nothing to the gradient. We only learn from successes.

And the gradient is noisy: depending on which responses we happened to sample, the update could point in very different directions.

We need a way to learn from all the samples, not just the correct ones.

Why Is the Variance So High?

Consider two scenarios for the same prompt:

Why Is the Variance So High?

Consider two scenarios for the same prompt:

Scenario A: we sample 4 responses and get rewards $[1, 1, 1, 0]$.

Why Is the Variance So High?

Consider two scenarios for the same prompt:

Scenario A: we sample 4 responses and get rewards $[1, 1, 1, 0]$.

Gradient: push up three responses, ignore one. Strong signal: "keep doing what you're doing."

Why Is the Variance So High?

Consider two scenarios for the same prompt:

Scenario A: we sample 4 responses and get rewards $[1, 1, 1, 0]$.

Gradient: push up three responses, ignore one. Strong signal: "keep doing what you're doing."

Scenario B: we sample 4 responses and get rewards $[0, 0, 0, 1]$.

Why Is the Variance So High?

Consider two scenarios for the same prompt:

Scenario A: we sample 4 responses and get rewards $[1, 1, 1, 0]$.

Gradient: push up three responses, ignore one. Strong signal: "keep doing what you're doing."

Scenario B: we sample 4 responses and get rewards $[0, 0, 0, 1]$.

Gradient: push up one response, ignore three. Weak, noisy signal based on a single lucky sample.

Why Is the Variance So High?

Consider two scenarios for the same prompt:

Scenario A: we sample 4 responses and get rewards $[1, 1, 1, 0]$.

Gradient: push up three responses, ignore one. Strong signal: "keep doing what you're doing."

Scenario B: we sample 4 responses and get rewards $[0, 0, 0, 1]$.

Gradient: push up one response, ignore three. Weak, noisy signal based on a single lucky sample.

Both scenarios come from the same model. But the gradient estimate is completely different depending on which responses we happened to sample.

Why Is the Variance So High?

Consider two scenarios for the same prompt:

Scenario A: we sample 4 responses and get rewards $[1, 1, 1, 0]$.

Gradient: push up three responses, ignore one. Strong signal: "keep doing what you're doing."

Scenario B: we sample 4 responses and get rewards $[0, 0, 0, 1]$.

Gradient: push up one response, ignore three. Weak, noisy signal based on a single lucky sample.

Both scenarios come from the same model. But the gradient estimate is completely different depending on which responses we happened to sample.

With binary reward (0 or 1), the variance is especially bad because most of the probability mass contributes zero gradient.

The Baseline Trick

Instead of using raw reward R, subtract a baseline b:

The Baseline Trick

Instead of using raw reward R, subtract a baseline b:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} (\mathrm{R_i} - \mathrm{b}) \cdot \nabla_\theta \log \pi_\theta(\mathrm{o_i})$$

The Baseline Trick

Instead of using raw reward R, subtract a baseline b:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{N} \sum_{i=1}^{N} (R_i - b) \cdot \nabla_\theta \log \pi_\theta(o_i)$$

Crucially, subtracting b does not change the expected gradient (you can verify: $\mathbb{E}[b \cdot \nabla \log \pi] = b \cdot \nabla \sum_o \pi_\theta(o) = b \cdot \nabla 1 = 0$). But it reduces variance.

The Baseline Trick

Instead of using raw reward R, subtract a baseline b:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{N} \sum_{i=1}^{N} (R_i - b) \cdot \nabla_\theta \log \pi_\theta(o_i)$$

Crucially, subtracting b does not change the expected gradient (you can verify: $\mathbb{E}[b \cdot \nabla \log \pi] = b \cdot \nabla \sum_o \pi_\theta(o) = b \cdot \nabla 1 = 0$). But it reduces variance.

Now the update says:

The Baseline Trick

Instead of using raw reward R, subtract a baseline b:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{N} \sum_{i=1}^{N} (R_i - b) \cdot \nabla_\theta \log \pi_\theta(o_i)$$

Crucially, subtracting b does not change the expected gradient (you can verify: $\mathbb{E}[b \cdot \nabla \log \pi] = b \cdot \nabla \sum_o \pi_\theta(o) = b \cdot \nabla 1 = 0$). But it reduces variance.

Now the update says:

- $R_i > b$: better than average. Push up.

The Baseline Trick

Instead of using raw reward R, subtract a baseline b:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{N} \sum_{i=1}^{N} (R_i - b) \cdot \nabla_\theta \log \pi_\theta(o_i)$$

Crucially, subtracting b does not change the expected gradient (you can verify: $\mathbb{E}[b \cdot \nabla \log \pi] = b \cdot \nabla \sum_o \pi_\theta(o) = b \cdot \nabla 1 = 0$). But it reduces variance.

Now the update says:

- $R_i > b$: better than average. Push up.
- $R_i < b$: worse than average. Push down.

The Baseline Trick

Instead of using raw reward R, subtract a baseline b:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{N} \sum_{i=1}^{N} (R_i - b) \cdot \nabla_\theta \log \pi_\theta(o_i)$$

Crucially, subtracting b does not change the expected gradient (you can verify: $\mathbb{E}[b \cdot \nabla \log \pi] = b \cdot \nabla \sum_o \pi_\theta(o) = b \cdot \nabla 1 = 0$). But it reduces variance.

Now the update says:

- $R_i > b$: better than average. Push up.
- $R_i < b$: worse than average. Push down.

Every sample contributes. Wrong responses get pushed down, not ignored.

The Baseline Trick

Instead of using raw reward R, subtract a baseline b:

$$\nabla_\theta \mathcal{J} \approx \frac{1}{N} \sum_{i=1}^{N} (R_i - b) \cdot \nabla_\theta \log \pi_\theta(o_i)$$

Crucially, subtracting b does not change the expected gradient (you can verify: $\mathbb{E}[b \cdot \nabla \log \pi] = b \cdot \nabla \sum_o \pi_\theta(o) = b \cdot \nabla 1 = 0$). But it reduces variance.

Now the update says:

- $R_i > b$: better than average. Push up.
- $R_i < b$: worse than average. Push down.

Every sample contributes. Wrong responses get pushed down, not ignored.

What should b be? A natural choice: the mean reward of the group. This is GRPO.

Baseline: Back to Our Example

Same 4 responses, but now with baseline b = mean = 0.5:

Baseline: Back to Our Example

Same 4 responses, but now with baseline b = mean = 0.5:

Response	R	R − b	Gradient direction
"$7 \times 8 = 56$"	1	+0.5	push up
"Let me think... 54"	0	−0.5	push down
"56"	1	+0.5	push up
"Maybe 48?"	0	−0.5	push down

Baseline: Back to Our Example

Same 4 responses, but now with baseline b = mean = 0.5:

Response	R	R − b	Gradient direction
"$7 \times 8 = 56$"	1	+0.5	push up
"Let me think... 54"	0	−0.5	push down
"56"	1	+0.5	push up
"Maybe 48?"	0	−0.5	push down

Now all four responses contribute equally. The correct ones are reinforced, the wrong ones are suppressed.

Baseline: Back to Our Example

Same 4 responses, but now with baseline b = mean = 0.5:

Response	R	R − b	Gradient direction
"$7 \times 8 = 56$"	1	+0.5	push up
"Let me think... 54"	0	−0.5	push down
"56"	1	+0.5	push up
"Maybe 48?"	0	−0.5	push down

Now all four responses contribute equally. The correct ones are reinforced, the wrong ones are suppressed.

This is much more informative than REINFORCE, where the wrong responses were invisible (R = 0, zero gradient).

GRPO: The Idea

Group Relative Policy Optimization (Shao et al., 2024).

GRPO: The Idea

Group Relative Policy Optimization (Shao et al., 2024).

Used to train DeepSeek R1. The idea is remarkably simple.

GRPO: The Idea

Group Relative Policy Optimization (Shao et al., 2024).

Used to train DeepSeek R1. The idea is remarkably simple.

For each prompt q:

GRPO: The Idea

Group Relative Policy Optimization (Shao et al., 2024).

Used to train DeepSeek R1. The idea is remarkably simple.

For each prompt q:

1. Sample G responses from the current model: $o_1, o_2, \ldots, o_G$

GRPO: The Idea

Group Relative Policy Optimization (Shao et al., 2024).

Used to train DeepSeek R1. The idea is remarkably simple.

For each prompt q:

1. Sample G responses from the current model: $o_1, o_2, \ldots, o_G$
2. Score each one: $R_1, R_2, \ldots, R_G$

GRPO: The Idea

Group Relative Policy Optimization (Shao et al., 2024).

Used to train DeepSeek R1. The idea is remarkably simple.

For each prompt q:

1. Sample G responses from the current model: $o_1, o_2, \ldots, o_G$
2. Score each one: $R_1, R_2, \ldots, R_G$
3. Normalize rewards within this group:

$$\hat{A}_i = \frac{R_i - \text{mean}(R_1, \ldots, R_G)}{\text{std}(R_1, \ldots, R_G)}$$

GRPO: The Idea

Group Relative Policy Optimization (Shao et al., 2024).

Used to train DeepSeek R1. The idea is remarkably simple.

For each prompt q:

1. Sample G responses from the current model: $o_1, o_2, \ldots, o_G$
2. Score each one: $R_1, R_2, \ldots, R_G$
3. Normalize rewards within this group:

$$\hat{A}_i = \frac{R_i - \text{mean}(R_1, \ldots, R_G)}{\text{std}(R_1, \ldots, R_G)}$$

$\hat{A}_i$ is the advantage: how much better (or worse) response i is compared to its siblings from the same prompt.

GRPO: Walkthrough

Prompt: “Solve: $2x + 3 = 11$”

GRPO: Walkthrough

Prompt: “Solve: $2x + 3 = 11$”

We sample $G = 4$ responses:

GRPO: Walkthrough

Prompt: "Solve: $2x + 3 = 11$"

We sample $G = 4$ responses:

Response	R	$\hat{A}$	Effect
"$2x = 8$, so $x = 4$"	1	$+0.87$	push up
"$2x = 7$, so $x = 3.5$"	0	-0.87	push down
"Check: $2(4) + 3 = 11$. Yes, $x = 4$."	1	$+0.87$	push up
"I think $x = 5$"	0	-0.87	push down

GRPO: Walkthrough

Prompt: "Solve: $2x + 3 = 11$"

We sample $G = 4$ responses:

Response	R	$\hat{A}$	Effect
"$2x = 8$, so $x = 4$"	1	$+0.87$	push up
"$2x = 7$, so $x = 3.5$"	0	-0.87	push down
"Check: $2(4) + 3 = 11$. Yes, $x = 4$."	1	$+0.87$	push up
"I think $x = 5$"	0	-0.87	push down

Mean $= 0.5$, std $= 0.577$. So $\hat{A} = (R - 0.5)/0.577$.

GRPO: Walkthrough

Prompt: "Solve: $2x + 3 = 11$"

We sample $G = 4$ responses:

Response	R	$\hat{A}$	Effect
"$2x = 8$, so $x = 4$"	1	$+0.87$	push up
"$2x = 7$, so $x = 3.5$"	0	-0.87	push down
"Check: $2(4) + 3 = 11$. Yes, $x = 4$."	1	$+0.87$	push up
"I think $x = 5$"	0	-0.87	push down

Mean $= 0.5$, std $= 0.577$. So $\hat{A} = (R - 0.5)/0.577$.

Correct responses get positive advantage, wrong ones get negative. The normalization ensures the scale is consistent across prompts with different difficulty.

What Happens When a Prompt Is Easy or Hard?

Easy prompt (model gets 4/4 correct): rewards $= [1, 1, 1, 1]$.

What Happens When a Prompt Is Easy or Hard?

Easy prompt (model gets 4/4 correct): rewards $= [1, 1, 1, 1]$.

Mean $= 1$, std $= 0$. All advantages are 0. No update. The model already knows this.

What Happens When a Prompt Is Easy or Hard?

Easy prompt (model gets 4/4 correct): rewards $= [1, 1, 1, 1]$.

Mean $= 1$, std $= 0$. All advantages are 0. No update. The model already knows this.

Hard prompt (model gets 1/4 correct): rewards $= [0, 0, 0, 1]$.

What Happens When a Prompt Is Easy or Hard?

Easy prompt (model gets 4/4 correct): rewards $= [1, 1, 1, 1]$.

Mean $= 1$, std $= 0$. All advantages are 0. No update. The model already knows this.

Hard prompt (model gets 1/4 correct): rewards $= [0, 0, 0, 1]$.

The one correct response gets a large positive advantage. The model strongly reinforces whatever reasoning led to that rare success.

What Happens When a Prompt Is Easy or Hard?

Easy prompt (model gets 4/4 correct): rewards $= [1, 1, 1, 1]$.

Mean $= 1$, std $= 0$. All advantages are 0. No update. The model already knows this.

Hard prompt (model gets 1/4 correct): rewards $= [0, 0, 0, 1]$.

The one correct response gets a large positive advantage. The model strongly reinforces whatever reasoning led to that rare success.

Impossible prompt (model gets 0/4 correct): rewards $= [0, 0, 0, 0]$.

What Happens When a Prompt Is Easy or Hard?

Easy prompt (model gets 4/4 correct): rewards = $[1, 1, 1, 1]$.

Mean = 1, std = 0. All advantages are 0. No update. The model already knows this.

Hard prompt (model gets 1/4 correct): rewards = $[0, 0, 0, 1]$.

The one correct response gets a large positive advantage. The model strongly reinforces whatever reasoning led to that rare success.

Impossible prompt (model gets 0/4 correct): rewards = $[0, 0, 0, 0]$.

All advantages are 0. No update. Can't learn from total failure.

What Happens When a Prompt Is Easy or Hard?

Easy prompt (model gets 4/4 correct): rewards $= [1, 1, 1, 1]$.

Mean $= 1$, std $= 0$. All advantages are 0. No update. The model already knows this.

Hard prompt (model gets 1/4 correct): rewards $= [0, 0, 0, 1]$.

The one correct response gets a large positive advantage. The model strongly reinforces whatever reasoning led to that rare success.

Impossible prompt (model gets 0/4 correct): rewards $= [0, 0, 0, 0]$.

All advantages are 0. No update. Can't learn from total failure.

GRPO automatically focuses on prompts at the edge of the model's ability: hard enough that some responses fail, easy enough that some succeed.

GRPO: The Full Loss

The gradient uses the advantage, but we need two more ingredients:

GRPO: The Full Loss

The gradient uses the advantage, but we need two more ingredients:

Problem 1: if $\hat{A}_i$ is very large, one gradient step can change the policy drastically.

GRPO: The Full Loss

The gradient uses the advantage, but we need two more ingredients:

Problem 1: if $\hat{A}_i$ is very large, one gradient step can change the policy drastically.

Fix: clip the probability ratio $r_i = \pi_\theta(o_i)/\pi_{\text{old}}(o_i)$ to $[1-\epsilon, 1+\epsilon]$:

GRPO: The Full Loss

The gradient uses the advantage, but we need two more ingredients:

Problem 1: if $\hat{A}_i$ is very large, one gradient step can change the policy drastically.

Fix: clip the probability ratio $r_i = \pi_\theta(o_i)/\pi_{\text{old}}(o_i)$ to $[1-\epsilon, 1+\epsilon]$:

$$\mathcal{L} = -\frac{1}{G}\sum_{i=1}^{G} \min\left(r_i \hat{A}_i,\ \text{clip}(r_i, 1-\epsilon, 1+\epsilon)\,\hat{A}_i\right)$$

GRPO: The Full Loss

The gradient uses the advantage, but we need two more ingredients:

Problem 1: if $\hat{A}_i$ is very large, one gradient step can change the policy drastically.

Fix: clip the probability ratio $r_i = \pi_\theta(o_i)/\pi_{\text{old}}(o_i)$ to $[1-\epsilon, 1+\epsilon]$:

$$\mathcal{L} = -\frac{1}{G}\sum_{i=1}^{G} \min\left(r_i \hat{A}_i,\ \text{clip}(r_i, 1-\epsilon, 1+\epsilon)\,\hat{A}_i\right)$$

It prevents catastrophic updates.

GRPO: The Full Loss

The gradient uses the advantage, but we need two more ingredients:

Problem 1: if $\hat{A}_i$ is very large, one gradient step can change the policy drastically.

Fix: clip the probability ratio $r_i = \pi_\theta(o_i)/\pi_{\text{old}}(o_i)$ to $[1-\epsilon, 1+\epsilon]$:

$$\mathcal{L} = -\frac{1}{G}\sum_{i=1}^{G} \min\left(r_i \hat{A}_i,\ \text{clip}(r_i, 1-\epsilon, 1+\epsilon)\,\hat{A}_i\right)$$

It prevents catastrophic updates.

Problem 2: the policy might drift far from the SFT model.

GRPO: The Full Loss

The gradient uses the advantage, but we need two more ingredients:

Problem 1: if $\hat{A}_i$ is very large, one gradient step can change the policy drastically.

Fix: clip the probability ratio $r_i = \pi_\theta(o_i)/\pi_{\text{old}}(o_i)$ to $[1-\epsilon, 1+\epsilon]$:

$$\mathcal{L} = -\frac{1}{G}\sum_{i=1}^{G} \min\left(r_i \hat{A}_i,\ \text{clip}(r_i, 1-\epsilon, 1+\epsilon)\,\hat{A}_i\right)$$

It prevents catastrophic updates.

Problem 2: the policy might drift far from the SFT model.

Fix: add KL penalty $\beta \cdot \text{KL}(\pi_\theta \| \pi_{\text{ref}})$.

GRPO: The Full Loss

The gradient uses the advantage, but we need two more ingredients:

Problem 1: if $\hat{A}_i$ is very large, one gradient step can change the policy drastically.

Fix: clip the probability ratio $r_i = \pi_\theta(o_i)/\pi_{\text{old}}(o_i)$ to $[1-\epsilon, 1+\epsilon]$:

$$\mathcal{L} = -\frac{1}{G}\sum_{i=1}^{G} \min\left(r_i \hat{A}_i,\ \text{clip}(r_i, 1-\epsilon, 1+\epsilon)\,\hat{A}_i\right)$$

It prevents catastrophic updates.

Problem 2: the policy might drift far from the SFT model.

Fix: add KL penalty $\beta \cdot \text{KL}(\pi_\theta \| \pi_{\text{ref}})$.

Typical values: $\epsilon = 0.2$, $\beta = 0.01$, $G = 16$ to 64 responses per prompt.

Why Does This Produce Reasoning?

Nobody tells the model “show your work” or “verify your answer.”

Why Does This Produce Reasoning?

Nobody tells the model “show your work” or “verify your answer.”

But after GRPO training, something interesting happens:

Why Does This Produce Reasoning?

Nobody tells the model “show your work” or “verify your answer.”

But after GRPO training, something interesting happens:

Responses that include verification (“Check: $2(4) + 3 = 11$”) are correct more often. So they get positive advantage and get reinforced.

Why Does This Produce Reasoning?

Nobody tells the model “show your work” or “verify your answer.”

But after GRPO training, something interesting happens:

Responses that include verification (“Check: $2(4) + 3 = 11$”) are correct more often. So they get positive advantage and get reinforced.

Responses that skip steps (“I think $x = 5$”) are wrong more often. Negative advantage. Suppressed.

Why Does This Produce Reasoning?

Nobody tells the model "show your work" or "verify your answer."

But after GRPO training, something interesting happens:

Responses that include verification ("Check: $2(4) + 3 = 11$") are correct more often. So they get positive advantage and get reinforced.

Responses that skip steps ("I think x = 5") are wrong more often. Negative advantage. Suppressed.

Over many training steps, the model discovers that showing work and checking answers leads to higher reward. Chain-of-thought reasoning emerges from the reward signal.

Why Does This Produce Reasoning?

Nobody tells the model "show your work" or "verify your answer."

But after GRPO training, something interesting happens:

Responses that include verification ("Check: $2(4) + 3 = 11$") are correct more often. So they get positive advantage and get reinforced.

Responses that skip steps ("I think $x = 5$") are wrong more often. Negative advantage. Suppressed.

Over many training steps, the model discovers that showing work and checking answers leads to higher reward. Chain-of-thought reasoning emerges from the reward signal.

This is fundamentally different from SFT, where someone has to write chain-of-thought examples. With RL, the model figures it out.

Verified Rewards: Why Math and Code Are Special

For GRPO to work, you need a reward signal. Where does it come from?

Verified Rewards: Why Math and Code Are Special

For GRPO to work, you need a reward signal. Where does it come from?

For math: parse the final answer, compare to ground truth. Exact, automatic, free.

Verified Rewards: Why Math and Code Are Special

For GRPO to work, you need a reward signal. Where does it come from?

For math: parse the final answer, compare to ground truth. Exact, automatic, free.

For code: run the test suite. Pass/fail. Exact, automatic, free.

Verified Rewards: Why Math and Code Are Special

For GRPO to work, you need a reward signal. Where does it come from?

For math: parse the final answer, compare to ground truth. Exact, automatic, free.

For code: run the test suite. Pass/fail. Exact, automatic, free.

For formal proofs: Lean/Coq type checker. Exact, automatic, free.

Verified Rewards: Why Math and Code Are Special

For GRPO to work, you need a reward signal. Where does it come from?

For math: parse the final answer, compare to ground truth. Exact, automatic, free.

For code: run the test suite. Pass/fail. Exact, automatic, free.

For formal proofs: Lean/Coq type checker. Exact, automatic, free.

These are verified rewards. No learned reward model needed. The verifier is perfect.

Verified Rewards: Why Math and Code Are Special

For GRPO to work, you need a reward signal. Where does it come from?

For math: parse the final answer, compare to ground truth. Exact, automatic, free.

For code: run the test suite. Pass/fail. Exact, automatic, free.

For formal proofs: Lean/Coq type checker. Exact, automatic, free.

These are verified rewards. No learned reward model needed. The verifier is perfect.

For open-ended tasks ("write a good essay"), you need a learned reward model trained on human preferences (RLHF). That's noisier and harder. Verified rewards are why reasoning RL works so well for math and code specifically.

DeepSeek R1-Zero: Pure RL

DeepSeek (January 2025) ran an experiment: what happens if you apply GRPO to a base model without any SFT?

DeepSeek R1-Zero: Pure RL

DeepSeek (January 2025) ran an experiment: what happens if you apply GRPO to a base model without any SFT?

They called it R1-Zero. Start from the pre-trained DeepSeek-V3 (671B MoE), apply GRPO with verified math rewards, no instruction tuning at all.

DeepSeek R1-Zero: Pure RL

DeepSeek (January 2025) ran an experiment: what happens if you apply GRPO to a base model without any SFT?

They called it R1-Zero. Start from the pre-trained DeepSeek-V3 (671B MoE), apply GRPO with verified math rewards, no instruction tuning at all.

The model had never seen a chain-of-thought example. It had never been told to "think step by step."

DeepSeek R1-Zero: Pure RL

DeepSeek (January 2025) ran an experiment: what happens if you apply GRPO to a base model without any SFT?

They called it R1-Zero. Start from the pre-trained DeepSeek-V3 (671B MoE), apply GRPO with verified math rewards, no instruction tuning at all.

The model had never seen a chain-of-thought example. It had never been told to "think step by step."

What happened?

DeepSeek R1-Zero: Pure RL

DeepSeek (January 2025) ran an experiment: what happens if you apply GRPO to a base model without any SFT?

They called it R1-Zero. Start from the pre-trained DeepSeek-V3 (671B MoE), apply GRPO with verified math rewards, no instruction tuning at all.

The model had never seen a chain-of-thought example. It had never been told to "think step by step."

What happened?

Over the course of training, R1-Zero spontaneously developed:

DeepSeek R1-Zero: Pure RL

DeepSeek (January 2025) ran an experiment: what happens if you apply GRPO to a base model without any SFT?

They called it R1-Zero. Start from the pre-trained DeepSeek-V3 (671B MoE), apply GRPO with verified math rewards, no instruction tuning at all.

The model had never seen a chain-of-thought example. It had never been told to "think step by step."

What happened?

Over the course of training, R1-Zero spontaneously developed:

- Extended reasoning (thousands of tokens of "thinking" before answering)

DeepSeek R1-Zero: Pure RL

DeepSeek (January 2025) ran an experiment: what happens if you apply GRPO to a base model without any SFT?

They called it R1-Zero. Start from the pre-trained DeepSeek-V3 (671B MoE), apply GRPO with verified math rewards, no instruction tuning at all.

The model had never seen a chain-of-thought example. It had never been told to "think step by step."

What happened?

Over the course of training, R1-Zero spontaneously developed:

- Extended reasoning (thousands of tokens of "thinking" before answering)
- Self-verification ("Let me check...")

DeepSeek R1-Zero: Pure RL

DeepSeek (January 2025) ran an experiment: what happens if you apply GRPO to a base model without any SFT?

They called it R1-Zero. Start from the pre-trained DeepSeek-V3 (671B MoE), apply GRPO with verified math rewards, no instruction tuning at all.

The model had never seen a chain-of-thought example. It had never been told to "think step by step."

What happened?

Over the course of training, R1-Zero spontaneously developed:

- Extended reasoning (thousands of tokens of "thinking" before answering)
- Self-verification ("Let me check...")
- Backtracking ("Wait, that doesn't work. Try a different approach.")

DeepSeek R1-Zero: Pure RL

DeepSeek (January 2025) ran an experiment: what happens if you apply GRPO to a base model without any SFT?

They called it R1-Zero. Start from the pre-trained DeepSeek-V3 (671B MoE), apply GRPO with verified math rewards, no instruction tuning at all.

The model had never seen a chain-of-thought example. It had never been told to "think step by step."

What happened?

Over the course of training, R1-Zero spontaneously developed:

- Extended reasoning (thousands of tokens of "thinking" before answering)
- Self-verification ("Let me check...")
- Backtracking ("Wait, that doesn't work. Try a different approach.")

All from the reward signal alone.

The “Aha Moment”

From the DeepSeek R1 paper. A real model output during training:

The "Aha Moment"

From the DeepSeek R1 paper. A real model output during training:

I need to find the value of $\sqrt{2+\sqrt{2+\sqrt{2+\cdots}}}$

Let me call this x. Then $x = \sqrt{2+x}$.

$x^2 = 2 + x$

$x^2 - x - 2 = 0$

$(x-2)(x+1) = 0$

$x = 2$ or $x = -1$. Since $x > 0$, we have $x = 2$.

Wait, I should verify this. Does $\sqrt{2+2} = 2$? $\sqrt{4} = 2$. Yes!

The answer is $\boxed{2}$.

The "Aha Moment"

From the DeepSeek R1 paper. A real model output during training:

I need to find the value of $\sqrt{2+\sqrt{2+\sqrt{2+\cdots}}}$

Let me call this x. Then $x = \sqrt{2+x}$.

$x^2 = 2 + x$

$x^2 - x - 2 = 0$

$(x-2)(x+1) = 0$

$x = 2$ or $x = -1$. Since $x > 0$, we have $x = 2$.

Wait, I should verify this. Does $\sqrt{2+2} = 2$? $\sqrt{4} = 2$. Yes!

The answer is $\boxed{2}$.

"Wait, I should verify this" was never in any training example. The model learned that responses with verification are correct more often, so verification got reinforced.

R1-Zero's Problems

R1-Zero was impressive but had issues:

R1-Zero's Problems

R1-Zero was impressive but had issues:

- Readability: the reasoning was messy. It would mix languages, repeat itself, go on tangents.

R1-Zero's Problems

R1-Zero was impressive but had issues:

- Readability: the reasoning was messy. It would mix languages, repeat itself, go on tangents.
- Formatting: no consistent output structure. Hard to parse the final answer.

R1-Zero's Problems

R1-Zero was impressive but had issues:

- Readability: the reasoning was messy. It would mix languages, repeat itself, go on tangents.
- Formatting: no consistent output structure. Hard to parse the final answer.
- Non-math tasks: without SFT, the model couldn't follow basic instructions outside of math.

R1-Zero's Problems

R1-Zero was impressive but had issues:

- Readability: the reasoning was messy. It would mix languages, repeat itself, go on tangents.
- Formatting: no consistent output structure. Hard to parse the final answer.
- Non-math tasks: without SFT, the model couldn't follow basic instructions outside of math.

The fix: DeepSeek R1 (the full model) used a multi-stage pipeline:

R1-Zero's Problems

R1-Zero was impressive but had issues:

- Readability: the reasoning was messy. It would mix languages, repeat itself, go on tangents.
- Formatting: no consistent output structure. Hard to parse the final answer.
- Non-math tasks: without SFT, the model couldn't follow basic instructions outside of math.

The fix: DeepSeek R1 (the full model) used a multi-stage pipeline:

1. A small amount of "cold start" SFT data (just to teach output format)

R1-Zero's Problems

R1-Zero was impressive but had issues:

- Readability: the reasoning was messy. It would mix languages, repeat itself, go on tangents.
- Formatting: no consistent output structure. Hard to parse the final answer.
- Non-math tasks: without SFT, the model couldn't follow basic instructions outside of math.

The fix: DeepSeek R1 (the full model) used a multi-stage pipeline:

1. A small amount of "cold start" SFT data (just to teach output format)
2. GRPO on math and code with verified rewards

R1-Zero's Problems

R1-Zero was impressive but had issues:

- Readability: the reasoning was messy. It would mix languages, repeat itself, go on tangents.
- Formatting: no consistent output structure. Hard to parse the final answer.
- Non-math tasks: without SFT, the model couldn't follow basic instructions outside of math.

The fix: DeepSeek R1 (the full model) used a multi-stage pipeline:

1. A small amount of "cold start" SFT data (just to teach output format)
2. GRPO on math and code with verified rewards
3. A rejection sampling stage to collect high-quality reasoning traces

R1-Zero's Problems

R1-Zero was impressive but had issues:

- Readability: the reasoning was messy. It would mix languages, repeat itself, go on tangents.
- Formatting: no consistent output structure. Hard to parse the final answer.
- Non-math tasks: without SFT, the model couldn't follow basic instructions outside of math.

The fix: DeepSeek R1 (the full model) used a multi-stage pipeline:

1. A small amount of "cold start" SFT data (just to teach output format)
2. GRPO on math and code with verified rewards
3. A rejection sampling stage to collect high-quality reasoning traces
4. Final SFT on the collected traces (to improve readability)

R1-Zero's Problems

R1-Zero was impressive but had issues:

- Readability: the reasoning was messy. It would mix languages, repeat itself, go on tangents.
- Formatting: no consistent output structure. Hard to parse the final answer.
- Non-math tasks: without SFT, the model couldn't follow basic instructions outside of math.

The fix: DeepSeek R1 (the full model) used a multi-stage pipeline:

1. A small amount of "cold start" SFT data (just to teach output format)
2. GRPO on math and code with verified rewards
3. A rejection sampling stage to collect high-quality reasoning traces
4. Final SFT on the collected traces (to improve readability)

The RL step is the core. The SFT steps are scaffolding to make the output usable.

Distillation: Making It Small

R1 is a 671B parameter MoE model. Too expensive for most uses.

Distillation: Making It Small

R1 is a 671B parameter MoE model. Too expensive for most uses.

DeepSeek's solution: distill R1's reasoning into smaller models.

Distillation: Making It Small

R1 is a 671B parameter MoE model. Too expensive for most uses.

DeepSeek's solution: distill R1's reasoning into smaller models.

1. Use R1 to generate solutions to 800K math and code problems.

Distillation: Making It Small

R1 is a 671B parameter MoE model. Too expensive for most uses.

DeepSeek's solution: distill R1's reasoning into smaller models.

1. Use R1 to generate solutions to 800K math and code problems.
2. SFT smaller models (Qwen2.5-1.5B, 7B, 14B, 32B) on these solutions.

Distillation: Making It Small

R1 is a 671B parameter MoE model. Too expensive for most uses.

DeepSeek's solution: distill R1's reasoning into smaller models.

1. Use R1 to generate solutions to 800K math and code problems.
2. SFT smaller models (Qwen2.5-1.5B, 7B, 14B, 32B) on these solutions.

Results:

Distillation: Making It Small

R1 is a 671B parameter MoE model. Too expensive for most uses.

DeepSeek's solution: distill R1's reasoning into smaller models.

1. Use R1 to generate solutions to 800K math and code problems.
2. SFT smaller models (Qwen2.5-1.5B, 7B, 14B, 32B) on these solutions.

Results:

Model	AIME 2024	Size
DeepSeek R1	79.8%	671B MoE
R1-Distill-Qwen-32B	72.6%	32B
R1-Distill-Qwen-7B	55.5%	7B
R1-Distill-Qwen-1.5B	28.6%	1.5B

Distillation: Making It Small

R1 is a 671B parameter MoE model. Too expensive for most uses.

DeepSeek's solution: distill R1's reasoning into smaller models.

1. Use R1 to generate solutions to 800K math and code problems.
2. SFT smaller models (Qwen2.5-1.5B, 7B, 14B, 32B) on these solutions.

Results:

Model	AIME 2024	Size
DeepSeek R1	79.8%	671B MoE
R1-Distill-Qwen-32B	72.6%	32B
R1-Distill-Qwen-7B	55.5%	7B
R1-Distill-Qwen-1.5B	28.6%	1.5B

The 32B distilled model retains most of R1's reasoning ability. You can run it on Delta.

Test-Time Compute: Thinking Longer

OpenAI's o1 (September 2024) introduced a different kind of scaling.

Test-Time Compute: Thinking Longer

OpenAI's o1 (September 2024) introduced a different kind of scaling.

Traditional scaling: make the model bigger, train on more data.

Test-Time Compute: Thinking Longer

OpenAI's o1 (September 2024) introduced a different kind of scaling.

Traditional scaling: make the model bigger, train on more data.

Test-time scaling: keep the model fixed, but let it think longer at inference.

Test-Time Compute: Thinking Longer

OpenAI's o1 (September 2024) introduced a different kind of scaling.

Traditional scaling: make the model bigger, train on more data.

Test-time scaling: keep the model fixed, but let it think longer at inference.

Model	AIME 2024 (out of 30)	Thinking
GPT-4o	9	None
o1-preview	12	Some
o1	17	Extended
o3-mini (high)	22	Very extended

Test-Time Compute: Thinking Longer

OpenAI's o1 (September 2024) introduced a different kind of scaling.

Traditional scaling: make the model bigger, train on more data.

Test-time scaling: keep the model fixed, but let it think longer at inference.

Model	AIME 2024 (out of 30)	Thinking
GPT-4o	9	None
o1-preview	12	Some
o1	17	Extended
o3-mini (high)	22	Very extended

Same base architecture. The difference: how many tokens of "thinking" the model generates before committing to an answer.

Why Does Thinking Longer Help?

When you think for 100 tokens, you can do:

Why Does Thinking Longer Help?

When you think for 100 tokens, you can do:

- Read the problem and give a quick answer.

Why Does Thinking Longer Help?

When you think for 100 tokens, you can do:

- Read the problem and give a quick answer.

When you think for 5,000 tokens, you can do:

Why Does Thinking Longer Help?

When you think for 100 tokens, you can do:

- Read the problem and give a quick answer.

When you think for 5,000 tokens, you can do:

- Try approach A. Get stuck. Backtrack.

Why Does Thinking Longer Help?

When you think for 100 tokens, you can do:

- Read the problem and give a quick answer.

When you think for 5,000 tokens, you can do:

- Try approach A. Get stuck. Backtrack.
- Try approach B. Make progress. Hit an edge case.

Why Does Thinking Longer Help?

When you think for 100 tokens, you can do:

- Read the problem and give a quick answer.

When you think for 5,000 tokens, you can do:

- Try approach A. Get stuck. Backtrack.
- Try approach B. Make progress. Hit an edge case.
- Fix the edge case. Arrive at an answer.

Why Does Thinking Longer Help?

When you think for 100 tokens, you can do:

- Read the problem and give a quick answer.

When you think for 5,000 tokens, you can do:

- Try approach A. Get stuck. Backtrack.
- Try approach B. Make progress. Hit an edge case.
- Fix the edge case. Arrive at an answer.
- Verify the answer by plugging it back in.

Why Does Thinking Longer Help?

When you think for 100 tokens, you can do:

- Read the problem and give a quick answer.

When you think for 5,000 tokens, you can do:

- Try approach A. Get stuck. Backtrack.
- Try approach B. Make progress. Hit an edge case.
- Fix the edge case. Arrive at an answer.
- Verify the answer by plugging it back in.
- Double-check with a different method.

Why Does Thinking Longer Help?

When you think for 100 tokens, you can do:

- Read the problem and give a quick answer.

When you think for 5,000 tokens, you can do:

- Try approach A. Get stuck. Backtrack.
- Try approach B. Make progress. Hit an edge case.
- Fix the edge case. Arrive at an answer.
- Verify the answer by plugging it back in.
- Double-check with a different method.

More thinking tokens means more room for trial and error. The model can explore multiple solution paths and converge on the right one.

Why Does Thinking Longer Help?

When you think for 100 tokens, you can do:

- Read the problem and give a quick answer.

When you think for 5,000 tokens, you can do:

- Try approach A. Get stuck. Backtrack.
- Try approach B. Make progress. Hit an edge case.
- Fix the edge case. Arrive at an answer.
- Verify the answer by plugging it back in.
- Double-check with a different method.

More thinking tokens means more room for trial and error. The model can explore multiple solution paths and converge on the right one.

Diminishing returns: the first 1000 tokens help a lot (try one approach, verify). Going from 5000 to 10000 helps less (the model is mostly repeating itself).

Where We Are Now (2026)

Every major lab now trains reasoning models with RL:

Where We Are Now (2026)

Every major lab now trains reasoning models with RL:

Model	Reasoning RL?	Open weights?	Key idea
OpenAI o3	Yes	No	Test-time compute
Claude Opus 4	Yes	No	Extended thinking
Gemini 2.5 Pro	Yes	No	Thinking mode
DeepSeek R1	Yes	Yes	GRPO
Qwen3	Yes	Yes	Thinking mode

Where We Are Now (2026)

Every major lab now trains reasoning models with RL:

Model	Reasoning RL?	Open weights?	Key idea
OpenAI o3	Yes	No	Test-time compute
Claude Opus 4	Yes	No	Extended thinking
Gemini 2.5 Pro	Yes	No	Thinking mode
DeepSeek R1	Yes	Yes	GRPO
Qwen3	Yes	Yes	Thinking mode

DeepSeek R1 showed it's possible to do this in the open. The weights, the paper, and the distilled models are all public.

The Open Question

We know RL works for math and code, where we have perfect verifiers.

The Open Question

We know RL works for math and code, where we have perfect verifiers.

What about everything else? Writing, analysis, planning, persuasion?

The Open Question

We know RL works for math and code, where we have perfect verifiers.

What about everything else? Writing, analysis, planning, persuasion?

For those, you need a learned reward model: train a separate neural network to predict human preferences. This is RLHF (Reinforcement Learning from Human Feedback).

The Open Question

We know RL works for math and code, where we have perfect verifiers.

What about everything else? Writing, analysis, planning, persuasion?

For those, you need a learned reward model: train a separate neural network to predict human preferences. This is RLHF (Reinforcement Learning from Human Feedback).

RLHF is how ChatGPT and Claude learn to be helpful and harmless. But the reward model is imperfect, and the model can learn to exploit its flaws ("reward hacking").

The Open Question

We know RL works for math and code, where we have perfect verifiers.

What about everything else? Writing, analysis, planning, persuasion?

For those, you need a learned reward model: train a separate neural network to predict human preferences. This is RLHF (Reinforcement Learning from Human Feedback).

RLHF is how ChatGPT and Claude learn to be helpful and harmless. But the reward model is imperfect, and the model can learn to exploit its flaws ("reward hacking").

The frontier question: can we build verifiers for more domains? Can we verify whether an argument is logically valid? Whether code is secure? Whether a scientific claim follows from the data?

The Open Question

We know RL works for math and code, where we have perfect verifiers.

What about everything else? Writing, analysis, planning, persuasion?

For those, you need a learned reward model: train a separate neural network to predict human preferences. This is RLHF (Reinforcement Learning from Human Feedback).

RLHF is how ChatGPT and Claude learn to be helpful and harmless. But the reward model is imperfect, and the model can learn to exploit its flaws ("reward hacking").

The frontier question: can we build verifiers for more domains? Can we verify whether an argument is logically valid? Whether code is secure? Whether a scientific claim follows from the data?

The more domains get verified rewards, the more domains get reasoning RL.

Thank you.

Good luck on the final project.