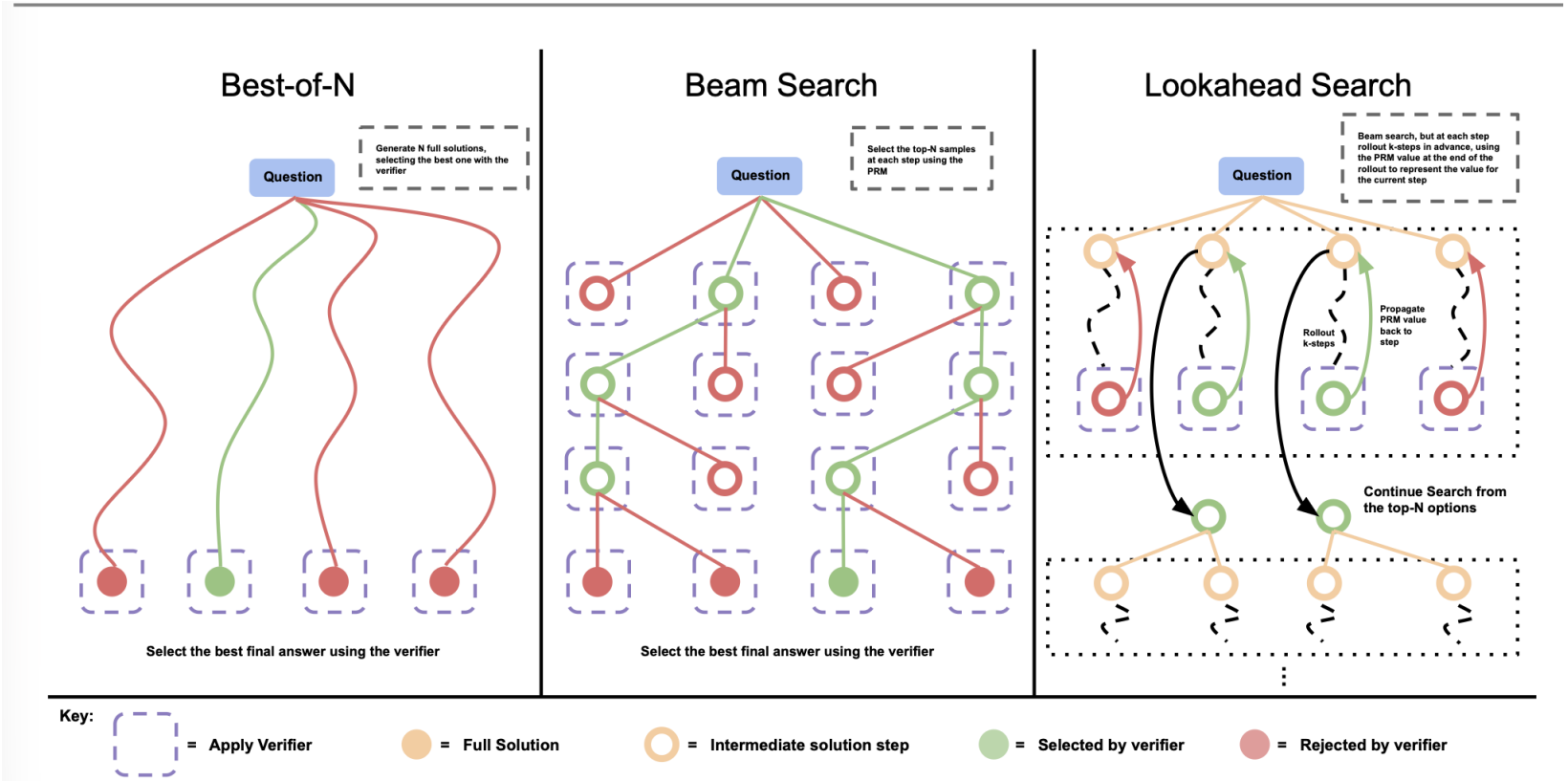
Efficient Test-Time Inference

Jun 22

Test-time Inference

Test-time inference is a phase where the training is done and weights are frozen, and people figure out some strategies to optimize its outputs.



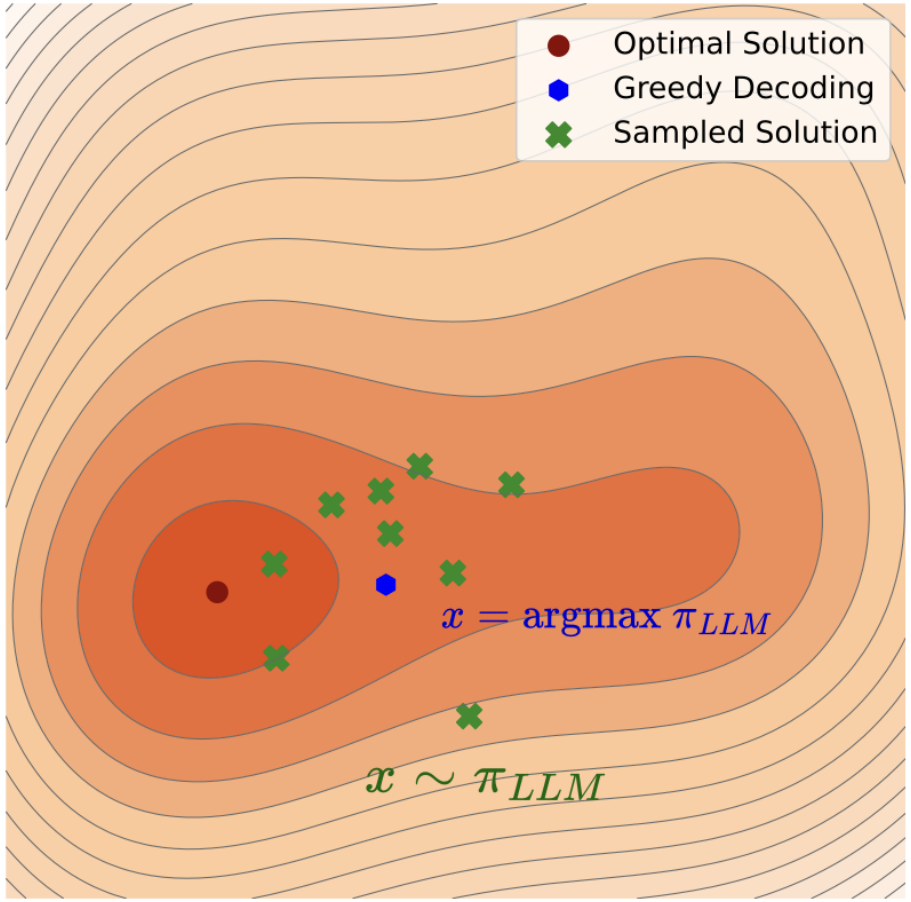
∇-REASONER: LLM REASONING VIA TEST-TIME GRADIENT DESCENT IN LATENT SPACE

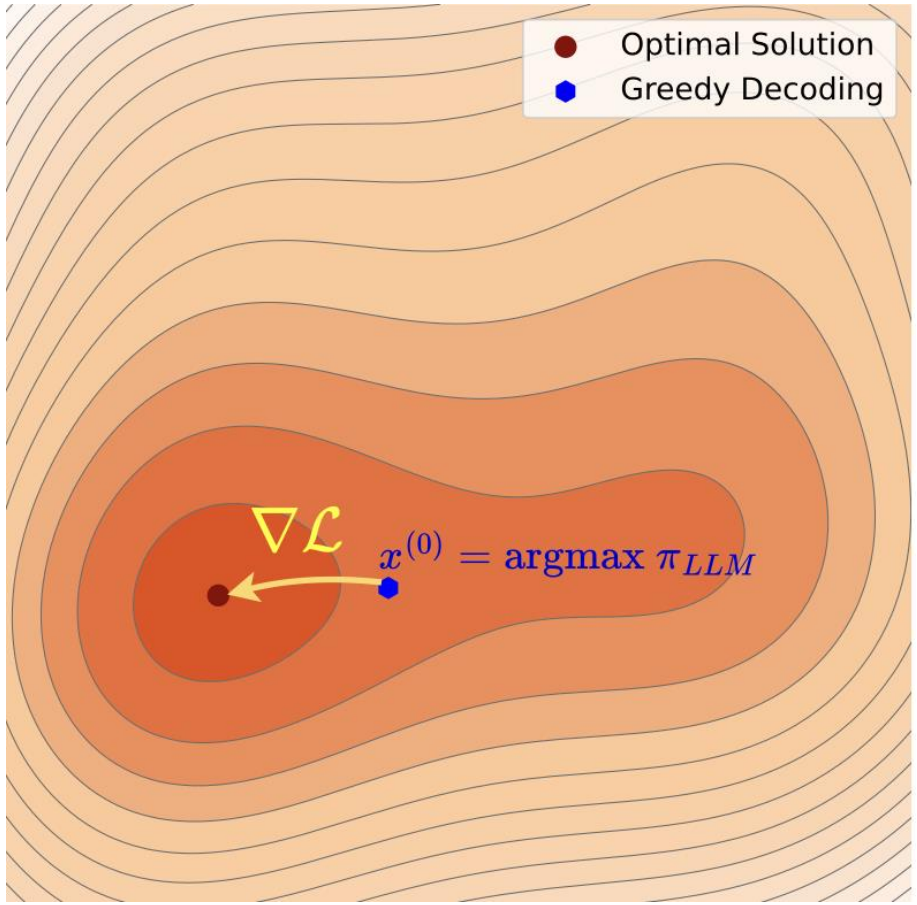
Peihao Wang[1]*, Ruisi Cai[1*], Zhen Wang[2], Hongyuan Mei[3], Qiang Liu[1], Pan Li[4], Zhangyang Wang[1]
[1]The University of Texas at Austin, [2]UC San Diego, [3]TTIC, [4]Georgia Tech

∇-REASONER

- Motivation: existing methods typically rely on inefficient discrete search (CoT, ToT, …) or trial-and-error (Best-of-N, Self-Consistency) prompting to improve the online policy.

∇-Reasoner


Optimal Solution
Greedy Decoding
Sampled Solution
$x = \text{argmax}\, \pi_{LLM}$
$x \sim \pi_{LLM}$


Optimal Solution
Greedy Decoding
$\nabla \mathcal{L}$
$x^{(0)} = \text{argmax}\, \pi_{LLM}$

∇-REASONER

1) But LLM tokens are discrete.

-- How to make it differentiable so that we can get gradient descent?

2) And this is test-time inference.

--How to get gradient descent in sample space?

∇-REASONER

Step 0: Formulate reasoning as sequential decision making problem via Bellman Equation.

Future outputs till the end

$$\pi^*_{LLM}(\cdot|\boldsymbol{y}_{\leq i-1}, \boldsymbol{x}) = \arg\max_{\boldsymbol{y}_i \in \mathcal{V}} \mathbb{E}_{\boldsymbol{y}_{\geq i+1} \sim \pi^*_{LLM}(\cdot|\boldsymbol{y}_i, \boldsymbol{y}_{\leq i-1}, \boldsymbol{x})} [r(\boldsymbol{y}_{\leq i-1}, \boldsymbol{y}_i, \boldsymbol{y}_{\geq i+1}|\boldsymbol{x})],$$

Optimal policy, given prompt **x**, and previous generated text **y**.

Choose the current word **y_i** from the vocabulary **V.**

Overall reward score for the concatenated sentence

∇-REASONER

Algorithm 1 ∇-Reasoner: Decoding with DTO

Require: Prompt $\boldsymbol{x}$, language model π_{LLM}, reward model r, stop criteria StopCriteria$(\cdot)$.

repeat
  $\boldsymbol{y}, \boldsymbol{z} \sim \pi_{LLM}(\cdot|\boldsymbol{x})$
  $\widetilde{\boldsymbol{z}} \leftarrow \text{DTO}(\boldsymbol{x}, \boldsymbol{z}, \pi_{LLM}, r)$.
  $\widetilde{\boldsymbol{y}}_1 \sim \text{softmax}(\widetilde{\boldsymbol{z}}_1/\tau)$.
  if $\widetilde{\boldsymbol{y}}_1 \neq \boldsymbol{y}_1$ **then**
    $\widetilde{\boldsymbol{y}}, \widetilde{\boldsymbol{z}} \sim \pi_{LLM}(\cdot|\widetilde{\boldsymbol{y}}_1, \boldsymbol{x})$
    if $r(\widetilde{\boldsymbol{y}}, \widetilde{\boldsymbol{y}}_1|\boldsymbol{x}) > r(\boldsymbol{y}|\boldsymbol{x})$ **then**
      $\boldsymbol{x} \leftarrow \text{concat}[\boldsymbol{x}, \widetilde{\boldsymbol{y}}_1]$
    else
      $\boldsymbol{x} \leftarrow \text{concat}[\boldsymbol{x}, \boldsymbol{y}_1]$
    end if
  else
    $\boldsymbol{x} \leftarrow \text{concat}[\boldsymbol{x}, \boldsymbol{y}_1]$
  end if
until StopCriteria$(\boldsymbol{x})$
return $\boldsymbol{x}$

Step 1 -- Rollout:
Given the prompt x, generate a response y from the LLM model.

Step 2 -- Differentiable Textual Optimization (DTO).

Step 3 -- Resample

Step 4 -- Accept / Reject

∇-REASONER

Algorithm 2 Differentiable Textual Optimization (DTO)

Require: Prefix $\boldsymbol{x}$, initial logits $\boldsymbol{z}$, language model π_{LLM}, reward model r, and the number of training steps T.

$\boldsymbol{z}^{(1)} \leftarrow \boldsymbol{z}$
for $t = 1, \cdots, T$ **do**
  for every $i = 1, \cdots, |\boldsymbol{y}|$ **do**
    $j^* \leftarrow \arg\max_{j \in [|\mathcal{V}|]} \boldsymbol{z}_{ij}^{(t)}$
    $\boldsymbol{y}_i^{(t)} \leftarrow \boldsymbol{\delta}_{j^*} + \text{softmax}(\boldsymbol{z}_i^{(t)}/\tau) - \text{StopGrad}(\text{softmax}(\boldsymbol{z}_i^{(t)}/\tau))$
  end for
  $\mathcal{L}_{nll} = -\sum_i \log \pi_{LLM}(\boldsymbol{y}^{(t)} | \boldsymbol{y}_{\leq i-1}^{(t)}, \boldsymbol{x})$
  $\mathcal{L}_{reward} = -r(\boldsymbol{y}^{(t)} | \boldsymbol{x})$.
  $\mathcal{L} = \mathcal{L}_{nll} + \lambda \mathcal{L}_{reward}$.
  $\boldsymbol{z}^{(t+1)} \leftarrow \boldsymbol{z}^{(t)} - \eta \nabla_{\boldsymbol{z}} \mathcal{L}$.
end for
return $\boldsymbol{z}^{(T)}$

Objective: $\mathcal{L}(\boldsymbol{y}) := -\lambda r(\boldsymbol{y}|\boldsymbol{x}) - \log \pi_{LLM}(\boldsymbol{y}|\boldsymbol{x})$

Reward Fluency

- Reward scores are from reward models (Skywork-V2-Qwen-4B for Qwen-based models and Skywork-V2-Llama-8B for Llama family models)

∇-REASONER

Reward Fluency

Objective: $\mathcal{L}(\boldsymbol{y}) := -\lambda r(\boldsymbol{y}|\boldsymbol{x}) - \log \pi_{LLM}(\boldsymbol{y}|\boldsymbol{x})$

$$\log \pi_{LLM}(\boldsymbol{y}|\boldsymbol{x}) = \sum_{i=1}^{|\boldsymbol{y}|} \boldsymbol{y}_i^\top \log \mathrm{Cat}\left(\pi_{LLM}(\cdot|\boldsymbol{y}_{\leq i-1}, \boldsymbol{x})\right)$$

one hot vector

Given:
Prompt x: “1+1 = ?”.
Response y: “equals two”.
Vocabulary V [one, two, three, equals, cat]

y_2:

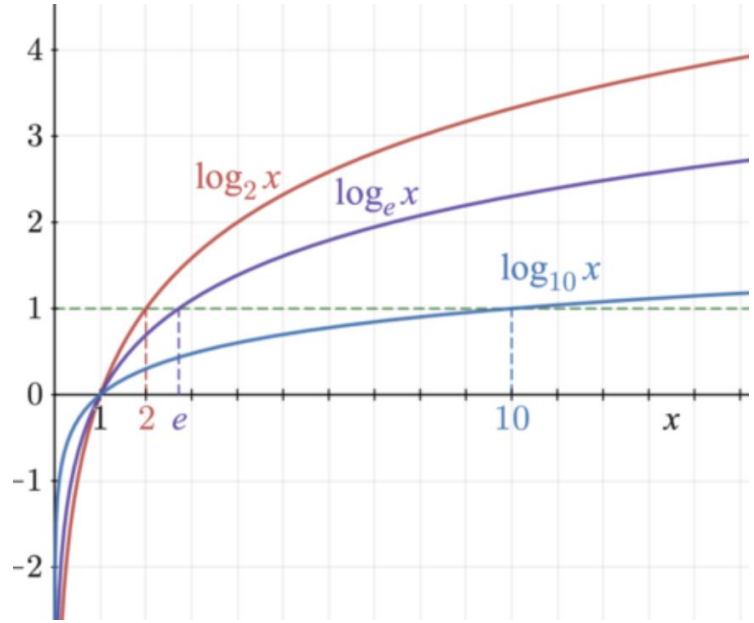
one	two	three	equals	cat
0	1	0	0	0

∇-REASONER

Reward Fluency

Objective: $\mathcal{L}(\boldsymbol{y}) := -\lambda r(\boldsymbol{y}|\boldsymbol{x}) - \log \pi_{LLM}(\boldsymbol{y}|\boldsymbol{x})$

$$\log \pi_{LLM}(\boldsymbol{y}|\boldsymbol{x}) = \sum_{i=1}^{|\boldsymbol{y}|} \boldsymbol{y}_i^\top \log \mathrm{Cat}\left(\pi_{LLM}(\cdot|\boldsymbol{y}_{\leq i-1}, \boldsymbol{x})\right)$$



Given:
Prompt x: "1+1 = ?".
Initial response y: "equals two".
Vocabulary V [one, two, three, equals, cat]

$\mathrm{Cat}(\pi_{\mathrm{LLM}}(\cdot\,|equals, \mathrm{x}))$

one	two	three	equals	cat
0.05	0.80	0.10	0.03	0.02

∇-REASONER

Now we have an objective function to take the derivative of.

$$\mathcal{L}(\boldsymbol{y}) := -\lambda r(\boldsymbol{y}|\boldsymbol{x}) - \log \pi_{LLM}(\boldsymbol{y}|\boldsymbol{x})$$

But we do not have a continuous variable yet to differentiate it with respect to, since tokens are discrete.

Therefore, we choose to use logits z, where softmax (z) -> y.

∇-Reasoner

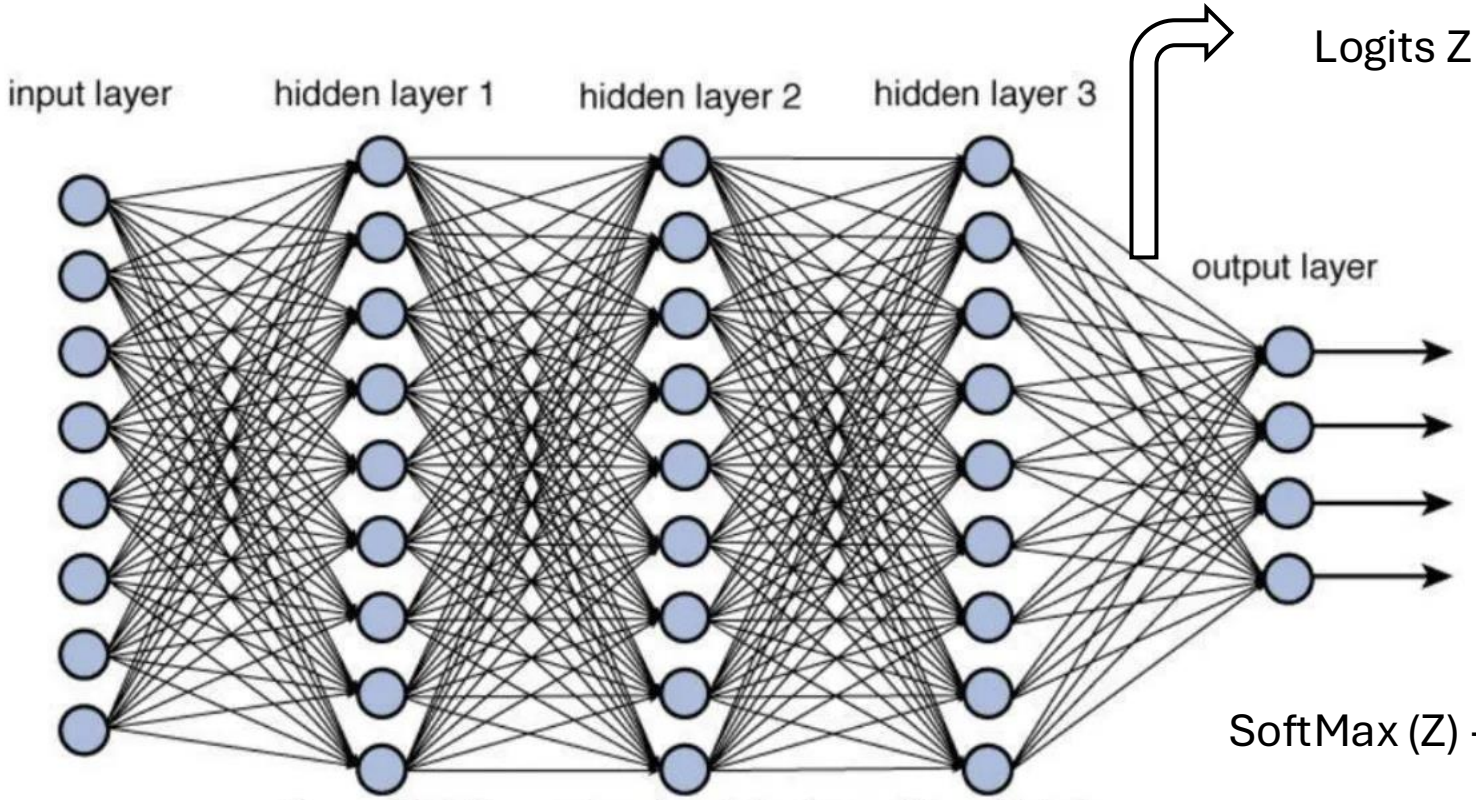


Figure 12.2 Deep network architecture with multiple layers.

Logits Z resides here!

$$y^{new} = y^{old} - \eta \cdot \frac{\partial L}{\partial y}$$

$$z^{new} = z^{old} - \eta \cdot \frac{\partial L}{\partial z}$$

SoftMax (Z) -> Y

∇-REASONER

Now we have a loss function contains Y but we want to take derivative on Z.

$$\mathcal{L}(\boldsymbol{y}) := -\lambda r(\boldsymbol{y}|\boldsymbol{x}) - \log \pi_{LLM}(\boldsymbol{y}|\boldsymbol{x})$$

How: We use Gumbel SoftMax straight-through trick to parameterize

$$\boldsymbol{y}_i^{(t)} = \boldsymbol{\delta}_{\arg\max_{i \in |\mathcal{V}|} \boldsymbol{z}_{ij}^{(t)}} + \text{softmax}(\boldsymbol{z}_i^{(t)}/\tau) - \text{StopGrad}(\text{softmax}(\boldsymbol{z}_i^{(t)}/\tau))$$

By this means, gradient descent can be equivalently performed on the space of $\boldsymbol{z}^{(t)}$, as

$$\boldsymbol{z}^{(t+1)} = \boldsymbol{z}^{(t)} - \eta \nabla_{\boldsymbol{z}} \mathcal{L}(\boldsymbol{z}^{(t)})$$

∇-REASONER

1) But LLM tokens are discrete.

-- How to make it differentiable so that we can get gradient descent?

Use logits Z space to surrogate token Y.

2) And this is test-time inference.

--How to get gradient descent in sample space?

Does not update the global weights at all. Only adjust the outputs for the specific question pairs.

∇-REASONER

To Speed Up:

- Gradient Caching:

  one-hot vector is hardly ever overturned;

- Rollout Trajectory Reusing:

- Confidence & Gradient – Guided Selection:

Table 1: Accuracy (%) on math reasoning datasets compared with baseline methods, including both test-time and training-time approaches. We skip results on AIME datasets for Llama-3.1-8B as it is incapable of generating reasonable performance. We mark the best performer in **bold** and the runner-up with underline. Our method outperforms all test-time baselines and even achieves performance on par with the training-based methods (SFT and GRPO), respectively.

Models	Methods	MATH-500	AMC	AIME24	AIME25
Qwen-2.5-7B	Greedy decoding	43.8	33.0	6.7	6.7
	SC (Xie et al., 2024) ($N = 8$)	69.8	49.4	22.5	20.0
	BoN (Stiennon et al., 2020) ($N = 8$)	70.2	50.1	22.5	13.3
	ToT (Yao et al., 2024)	57.8	42.4	6.7	10.0
	RAP (Hao et al., 2023)	68.6	50.1	18.3	14.2
	SFT (Ouyang et al., 2022)	65.8	36.4	6.3	11.7
	GRPO (Guo et al., 2025)	70.8	**52.8**	20.8	**16.7**
	∇-Reasoner ($N_{max} = 8$)	**71.0**	<u>51.5</u>	**23.3**	<u>15.0</u>
Qwen-2.5-7B-Instruct	Greedy decoding	71.2	43.0	5.3	7.5
	SC (Xie et al., 2024) ($N = 8$)	76.6	55.5	25.0	**22.5**
	BoN (Stiennon et al., 2020) ($N = 8$)	77.8	55.9	22.5	18.3
	ToT (Yao et al., 2024)	75.4	48.2	20.0	18.3
	RAP (Hao et al., 2023)	80.2	54.6	1.6	12.5
	TPO (Li et al., 2025)	77.6	55.9	6.7	11.1
	∇-Reasoner ($N_{max} = 8$)	80.4	56.8	26.6	20.0
Llama-3.1-8B-Instruct	Greedy decoding	40.6	19.3	-	-
	SC (Xie et al., 2024) ($N = 8$)	54.8	25.7	-	-
	BoN (Stiennon et al., 2020) ($N = 8$)	52.2	26.1	-	-
	ToT (Yao et al., 2024)	50.2	25.6	-	-
	RAP (Hao et al., 2023)	55.4	25.8	-	-
	SFT (Ouyang et al., 2022)	46.6	20.2	-	-
	∇-Reasoner ($N_{max} = 8$)	**55.8**	**28.9**	-	-

∇-REASONER

Table 3: Analysis of rejection rate (%) in rejection sampling. We set $N = 8$ for the BoN baseline and also set $N_{max} = 8$ for our ∇-Reasoner. The theoretical rejection rate of the baseline is 66.0%.

Model	Baseline	∇-Reasoner
Qwen-2.5	65.9	**32.8**
Qwen-2.5-Instruct	66.5	**28.9**
Llama-3-Instruct	66.9	**40.1**

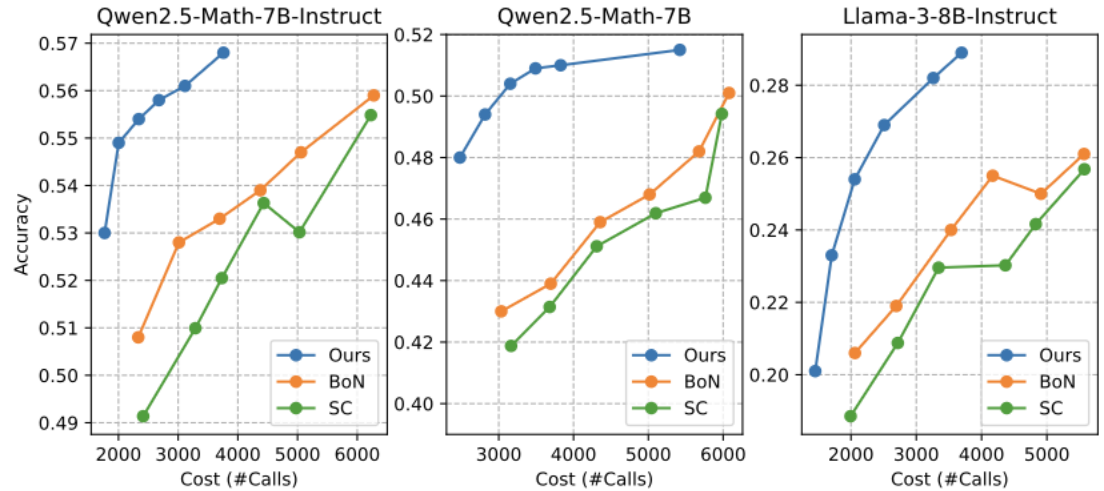


Figure 4: Test-time scaling curves comparing our method with BoN and SC. We change the number of samples N for BoN and SC and number of rollouts N_{max} for our method. The results show ∇-Reasoner achieves superior performance with reduced cost across multiple models.

∇-REASONER

Contribution:

1) Achieves 20% accuracy improvement on a challenging math reasoning benchmark while reducing 10-40% model calls compared to strong baselines.

2) Introduces a paradigm shift from zeroth-order search to first-order optimization at test time.

Rollout Roulette: A Probabilistic Inference Approach to Inference-Time Scaling of LLMs using Particle-Based Monte Carlo Methods

Isha Puri[1] **Shiv Sudalairaj**[2] **GX Xu**[2] **Kai Xu**[2] **Akash Srivastava**[2]

[1]MIT CSAIL [2]RedHat AI

NeurIPS 2025

Rollout Roulette

- Background: Reward models are imperfect for early pruning.

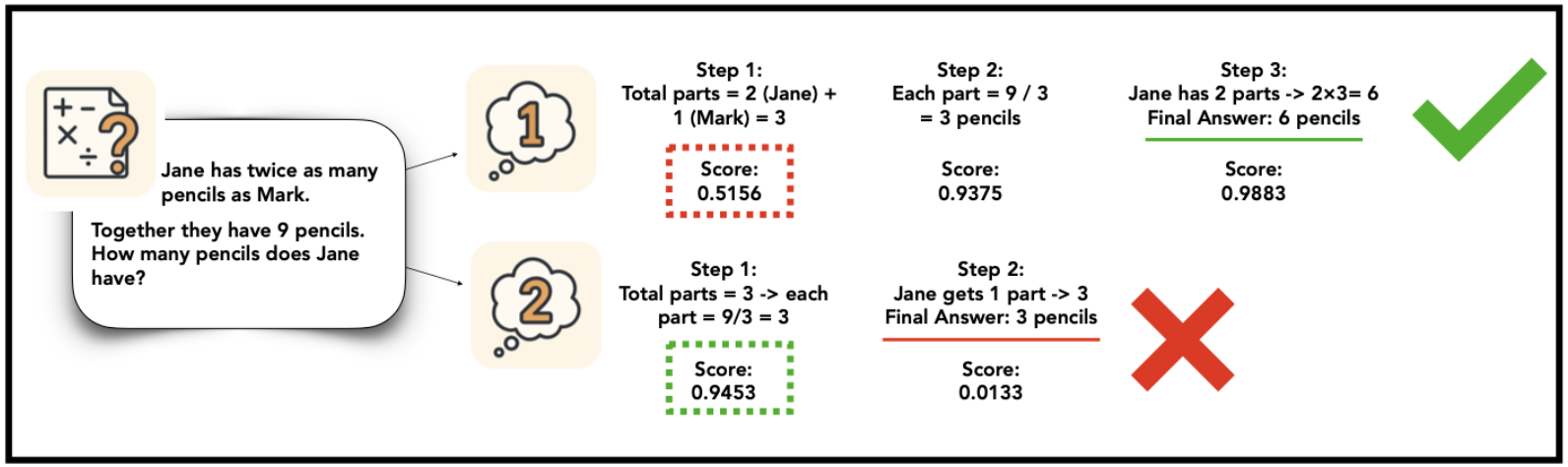
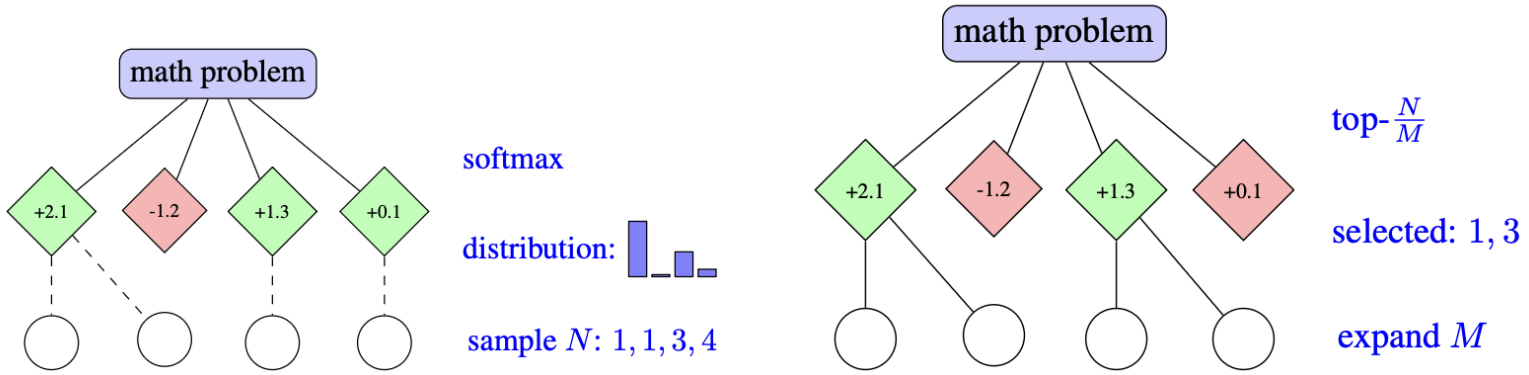


Figure 1: A true example of PRM assigning a lower score to the first step of a solution that turns out to be correct. In deterministic scaling methods, this solution would have been discarded in favor for one that had a higher initial PRM score but turned out to be incorrect.

Rollout Roulette



(a) Particle filtering uses the rewards to produce a softmax distribution and does stochastic expansion of N based sampling.

(b) Beam search treats the rewards as exact and performs deterministic expansion based on beam size N and beam width M.

Figure 8: A side-by-side comparison between particle filtering and its closest search-based counterpart, beam search. Compared with beam search in Figure 8b where the selection and expansion is deterministic (implicitly assumes the rewards are correct), particle filtering in Figure 8a trusts the rewards with uncertainty and propagates the expansion via sampling.

Rollout Roulette

- Novelty: From Search to Inference.

Search

1) Rely on noisy reward models;
2) Keep top-k paths and prune aggressively;
3) Mode;
4) Sensitive to early sub-optimal

Inference

1) Acknowledge rewards models are imperfect and can contain noises;
2) Maintain distribution for many paths;
3) Use posterior and propagate uncertainty;
3) Recover from early errors;

Traditional reward models give P(O|X) while this paper approximate P(X|O) to evaluate the Quality of reasoning, where O is reward feedback while X represents reasoning output.

Rollout Roulette

Method: models the LLM reasoning as state space models (SSM) and use particle filtering to approximate the posterior distribution in the SSM.

- A SSM models a process that we observe via indirect measurements and aim to infer the underlying state.

- They consist of: 1) a sequence of latent states $\{x_t\}_{t=1}^{T}$ -- Reasoning Steps
  2) corresponding observations $\{o_t\}_{t=1}^{T}$ -- Accepted / Not Accepted
  3) a transition model $p(x_t \mid x_{<t-1})$ that governs evolution of states -- LLM
  4) an emission model $p(o_t|x_t)$ to produce observations -- Process Reward Model

Rollout Roulette

"This formulation makes particle filtering a natural and theoretically grounded choice for inference"

The goal is to infer the posterior distribution over latent trajectories that yield fully accepted sequences $p_M(x_{1:T} \mid c, o_{1:T} = \mathbf{1})$

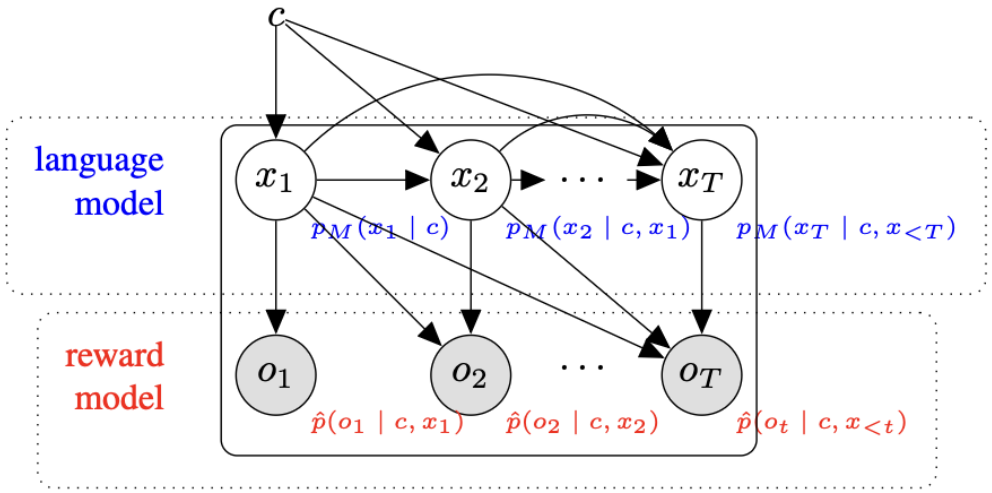


Figure 3: State-space model for inference-time scaling. c is a prompt, $x_1, \ldots, x_T$ are LLM outputs, and $o_1, \ldots, o_T$ are "observed" acceptances from a reward model. We estimate the latent states conditioned on $o_t = 1$ for all t.

Rollout Roulette

Step 1: Initialize N particles and assign scores.

In this step, let LLM generates the very first step repeatedly, and use a reward model to score them.

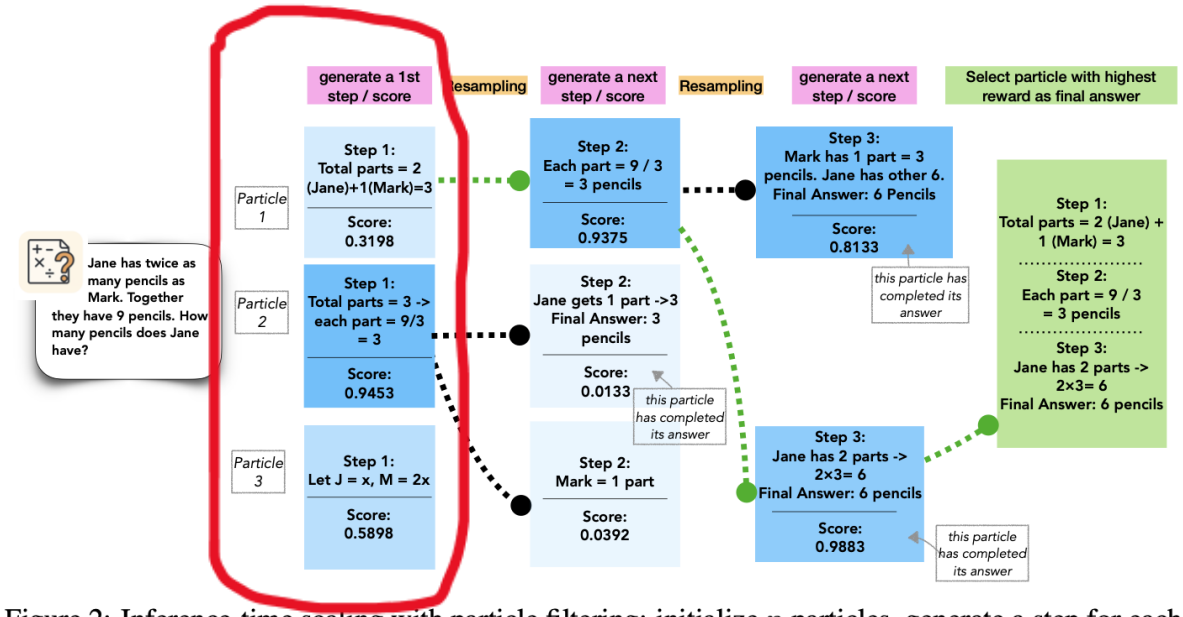


Figure 2: Inference-time scaling with particle filtering: initialize n particles, generate a step for each, score with the PRM, resample via softmax-weighted scores, and repeat until full solutions are formed.

Algorithm 1 Particle Filtering for Inference-Time Scaling

Input: the number of particles N, a reward model $\hat{r}$, a LLM p_M and the prompt c

Initialize N particles $\{x_1^{(i)} \sim p_M(\cdot \mid c)\}_{i=1}^N$

$t \leftarrow 1$

while not all particles stop **do**

  Update rewards $\mathbf{w} = [\hat{r}(x_{1:t}^{(1)}), \ldots, \hat{r}(x_{1:t}^{(N)})]$

  Compute softmax distribution $\theta = \text{softmax}(\mathbf{w})$

  Sample indices $\{j_t^{(i)}\}_{i=1}^N \sim \mathbb{P}_t(j = i) = \theta_i$

  Update the set of particles as $\{x_{1:t}^{(j_t^{(i)})}\}_{i=1}^N$

  Transition $\{x_{t+1}^{(i)} \sim p_M(\cdot \mid c, x_{1:t}^{(i)})\}_{i=1}^N$

  $t \leftarrow t + 1$

end while

Return: the set of weighted particles in the end

Rollout Roulette

Step 2: Resampling.

a. Convert the reward scores to a probability distribution [0.2393,0.4473,0.3134]

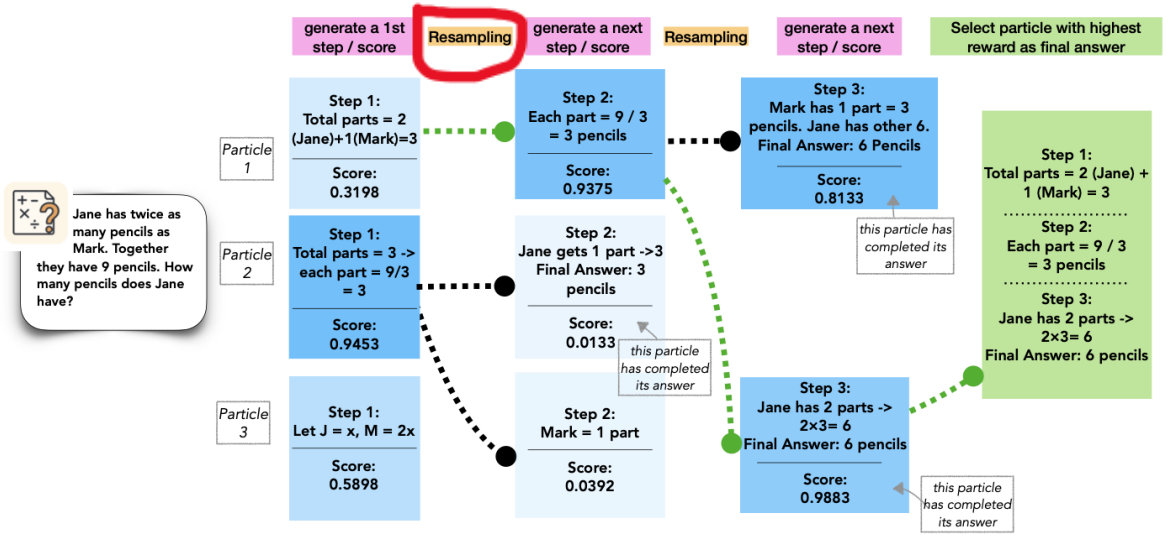


Figure 2: Inference-time scaling with particle filtering: initialize n particles, generate a step for each, score with the PRM, resample via softmax-weighted scores, and repeat until full solutions are formed.

Algorithm 1 Particle Filtering for Inference-Time Scaling

Input: the number of particles N, a reward model $\hat{r}$, a LLM p_M and the prompt c

Initialize N particles $\{x_1^{(i)} \sim p_M(\cdot \mid c)\}_{i=1}^N$

$t \leftarrow 1$

while not all particles stop **do**

  Update rewards $\mathbf{w} = [\hat{r}(x_{1:t}^{(1)}), \ldots, \hat{r}(x_{1:t}^{(N)})]$

  Compute softmax distribution $\theta = \text{softmax}(\mathbf{w})$

  Sample indices $\{j_t^{(i)}\}_{i=1}^N \sim \mathbb{P}_t(j = i) = \theta_i$

  Update the set of particles as $\{x_{1:t}^{(j_t^{(i)})}\}_{i=1}^N$

  Transition $\{x_{t+1}^{(i)} \sim p_M(\cdot \mid c, x_{1:t}^{(i)})\}_{i=1}^N$

  $t \leftarrow t + 1$

end while

Return: the set of weighted particles in the end

Rollout Roulette

Step 2: Resampling.

a. Convert the reward scores to a probability distribution [0.2393,0.4473,0.3134]

b. Sample N particles with replacement according to this probability distribution

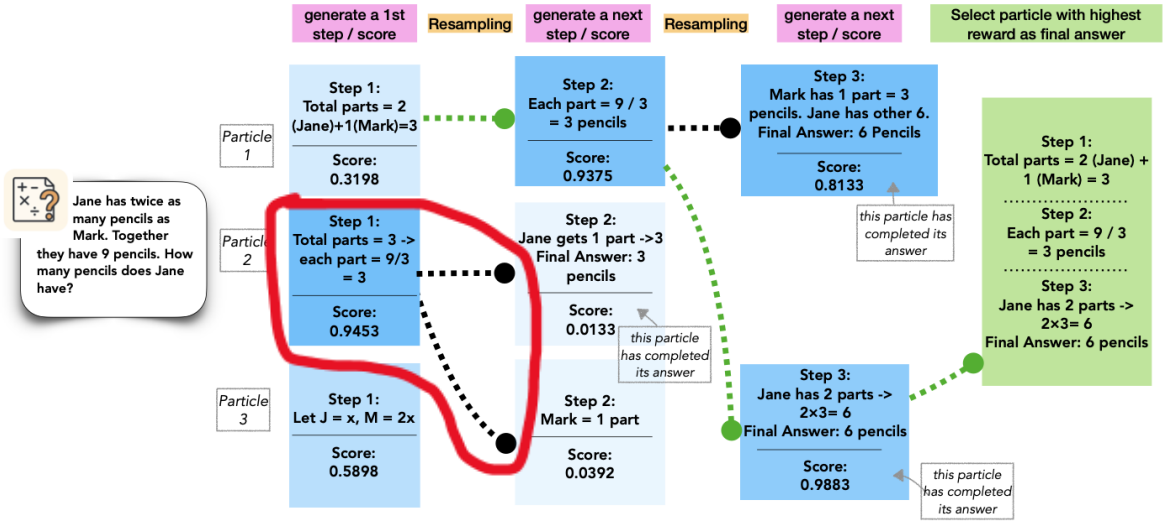


Figure 2: Inference-time scaling with particle filtering: initialize n particles, generate a step for each, score with the PRM, resample via softmax-weighted scores, and repeat until full solutions are formed.

Algorithm 1 Particle Filtering for Inference-Time Scaling

Input: the number of particles N, a reward model $\hat{r}$, a LLM p_M and the prompt c

Initialize N particles $\{x_1^{(i)} \sim p_M(\cdot \mid c)\}_{i=1}^N$

$t \leftarrow 1$

while not all particles stop **do**

  Update rewards $\mathbf{w} = [\hat{r}(x_{1:t}^{(1)}), \ldots, \hat{r}(x_{1:t}^{(N)})]$

  Compute softmax distribution $\theta = \text{softmax}(\mathbf{w})$

  Sample indices $\{j_t^{(i)}\}_{i=1}^N \sim \mathbb{P}_t(j = i) = \theta_i$

  Update the set of particles as $\{x_{1:t}^{(j_t^{(i)})}\}_{i=1}^N$

  Transition $\{x_{t+1}^{(i)} \sim p_M(\cdot \mid c, x_{1:t}^{(i)})\}_{i=1}^N$

  $t \leftarrow t + 1$

end while

Return: the set of weighted particles in the end

Rollout Roulette

Step 3: Transition -- form the next generation from the selected particles

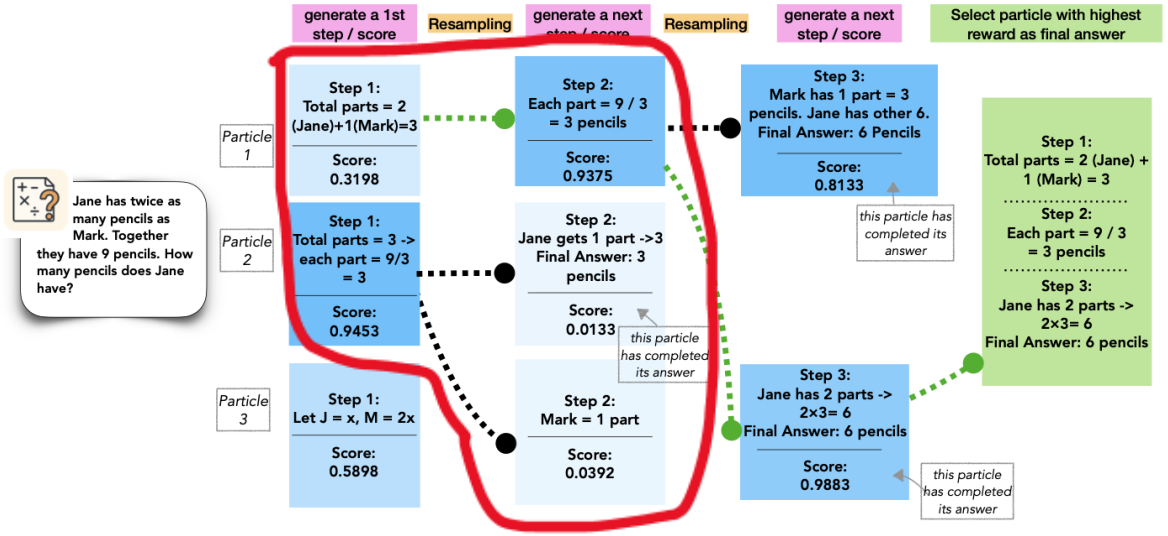


Figure 2: Inference-time scaling with particle filtering: initialize n particles, generate a step for each, score with the PRM, resample via softmax-weighted scores, and repeat until full solutions are formed.

Algorithm 1 Particle Filtering for Inference-Time Scaling

Input: the number of particles N, a reward model $\hat{r}$, a LLM p_M and the prompt c

Initialize N particles $\{x_1^{(i)} \sim p_M(\cdot \mid c)\}_{i=1}^N$

$t \leftarrow 1$

while not all particles stop **do**

  Update rewards $\mathbf{w} = [\hat{r}(x_{1:t}^{(1)}), \ldots, \hat{r}(x_{1:t}^{(N)})]$

  Compute softmax distribution $\theta = \text{softmax}(\mathbf{w})$

  Sample indices $\{j_t^{(i)}\}_{i=1}^N \sim \mathbb{P}_t(j = i) = \theta_i$

  Update the set of particles as $\{x_{1:t}^{(j_t^{(i)})}\}_{i=1}^N$

  Transition $\{x_{t+1}^{(i)} \sim p_M(\cdot \mid c, x_{1:t}^{(i)})\}_{i=1}^N$

  $t \leftarrow t + 1$

end while

Return: the set of weighted particles in the end

Rollout Roulette

Step 4: Loop until all particles complete. Finally get a set of weighted particles.

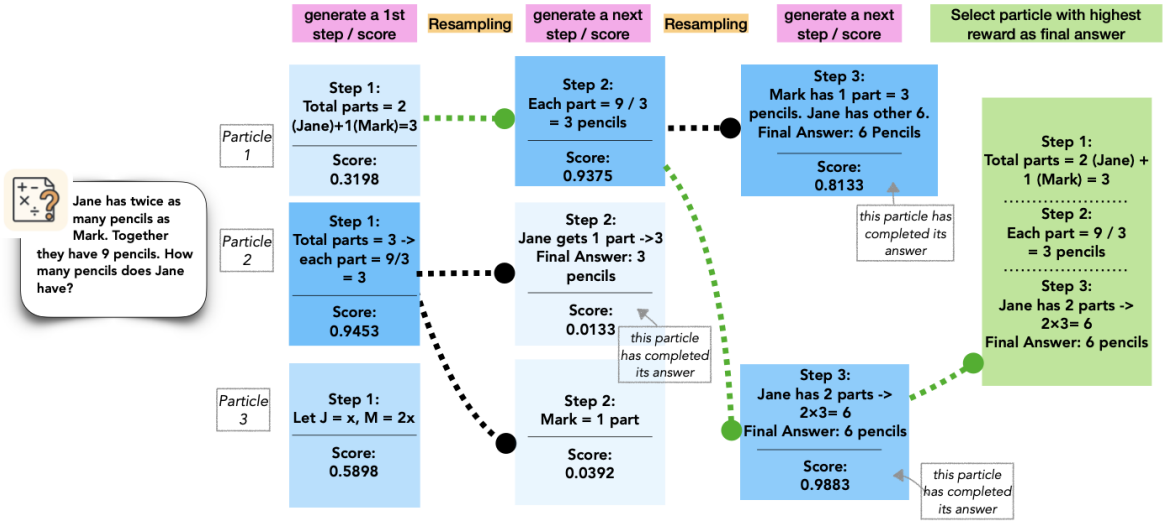


Figure 2: Inference-time scaling with particle filtering: initialize n particles, generate a step for each, score with the PRM, resample via softmax-weighted scores, and repeat until full solutions are formed.

Algorithm 1 Particle Filtering for Inference-Time Scaling

Input: the number of particles N, a reward model $\hat{r}$, a LLM p_M and the prompt c

Initialize N particles $\{x_1^{(i)} \sim p_M(\cdot \mid c)\}_{i=1}^N$

$t \leftarrow 1$

while not all particles stop **do**

  Update rewards $\mathbf{w} = [\hat{r}(x_{1:t}^{(1)}), \ldots, \hat{r}(x_{1:t}^{(N)})]$

  Compute softmax distribution $\theta = \text{softmax}(\mathbf{w})$

  Sample indices $\{j_t^{(i)}\}_{i=1}^N \sim \mathbb{P}_t(j = i) = \theta_i$

  Update the set of particles as $\{x_{1:t}^{(j_t^{(i)})}\}_{i=1}^N$

  Transition $\{x_{t+1}^{(i)} \sim p_M(\cdot \mid c, x_{1:t}^{(i)})\}_{i=1}^N$

  $t \leftarrow t + 1$

end while

Return: the set of weighted particles in the end

Rollout Roulette

Joint Distribution of states and observations:

$$p(x_{1:T}, o_{1:T}) = p(x_1)\prod_{t=2}^{T} p(x_t \mid x_{<t-1})\prod_{t=1}^{T} p(o_t \mid x_t)$$

An simple example of joint distribution:

P (X = Sunny, Y = On Time) ;
P (X = Sunny, Y = Late);
P (X = Rainy, Y = On Time);
P (X = Rainy, Y = Late);

Combinatorial Explosion makes The Joint Distribution

Rollout Roulette

To connect the process with posterior distribution….

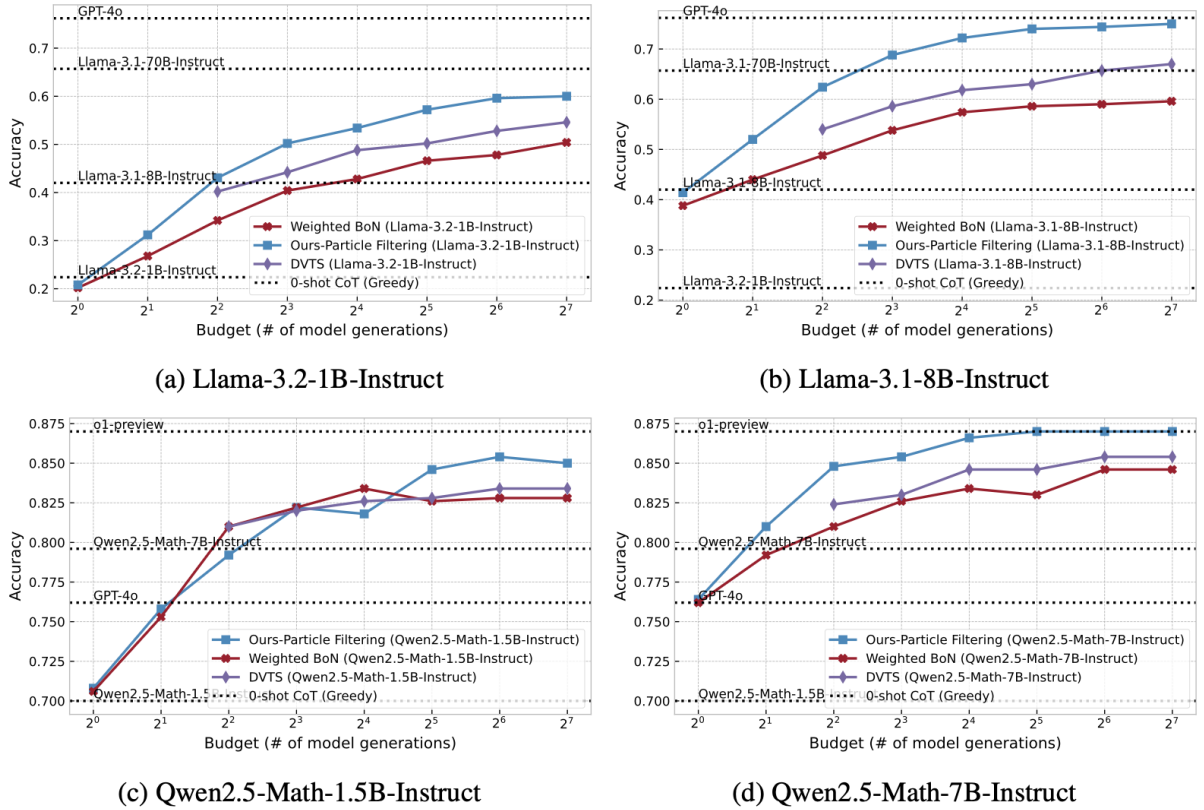
$$P(X|O) = P(X, O) / P(O)$$

$$p_M(x_{1:T}, o_{1:T} \mid c) \propto \prod_{t=1}^{T} p_M(x_t \mid c, x_{<t-1}) \prod_{t=1}^{T} p(o_t \mid c, x_{<t}),$$

Algorithm 1 Particle Filtering for Inference-Time Scaling

Input: the number of particles N, a reward model $\hat{r}$, a LLM p_M and the prompt c
Initialize N particles $\{x_1^{(i)} \sim p_M(\cdot \mid c)\}_{i=1}^{N}$
$t \leftarrow 1$
while not all particles stop **do**
  Update rewards $\mathbf{w} = [\hat{r}(x_{1:t}^{(1)}), \ldots, \hat{r}(x_{1:t}^{(N)})]$
  Compute softmax distribution $\theta = \text{softmax}(\mathbf{w})$
  Sample indices $\{j_t^{(i)}\}_{i=1}^{N} \sim \mathbb{P}_t(j = i) = \theta_i$
  Update the set of particles as $\{x_{1:t}^{(j_t^{(i)})}\}_{i=1}^{N}$
  Transition $\{x_{t+1}^{(i)} \sim p_M(\cdot \mid c, x_{1:t}^{(i)})\}_{i=1}^{N}$
  $t \leftarrow t + 1$
end while
Return: the set of weighted particles in the end

Rollout Roulette



(a) Llama-3.2-1B-Instruct

(b) Llama-3.1-8B-Instruct

(c) Qwen2.5-Math-1.5B-Instruct

(d) Qwen2.5-Math-7B-Instruct

Figure 4: Accuracy vs. Generation Budget across models using different inference-time strategies.

Efficient Reasoning:

Achieves a superior performance
Under the same budget.

Rollout Roulette

Achieves comparable performance with Famous closed LLMs (GPT-4o, Claude 3.5 Sonnet) with small size (1.5B / 7B) open source LLMs.

Model	Method	MATH500	AIME 2024
Closed Source LLMs			
GPT-4o	–	76.2	4/30
o1-preview	–	**87.0**	**12/30**
Claude 3.5 Sonnet	–	78.2	5/30
Open Source LLMs			
Llama-3.1 70B Instruct	–	65.6	5/30
Qwen-2.5 Math 72B Instruct	–	82.0	9/30
Open Source General SLMs			
Qwen-2.5 1.5B Instruct	Greedy	54.4	1/30
	Self Consistency	61.0	2/30
	BoN	67.8	1/30
	WBoN	69.2	2/30
	Beam Search	76.2	5/30
	DVTS	76.6	4/30
	Particle Filtering (Ours)	**79.3**	**6/30**
Open Source Math SLMs			
Qwen-2.5 Math 1.5B Instruct	Greedy	70.0	3/30
	Self Consistency	79.6	6/30
	BoN	81.8	4/30
	WBoN	82.6	4/30
	Beam Search	83.0	4/30
	DVTS	82.8	5/30
	Particle Filtering (Ours)	**84.6**	**7/30**
Qwen-2.5 Math 7B Instruct	Greedy	79.6	5/30
	Self Consistency	84.0	4/30
	BoN	82.6	5/30
	WBoN	83.0	5/30
	Beam Search	86.9	7/30
	DVTS	84.6	6/30
	Particle Filtering (Ours)	**87.7**	**10/30**

Table 1: Results of various LLMs on MATH500 and AIME 2024, highlighting particle filtering performance. All methods used a compute budget of 32 generations with Qwen2.5-Math-PRM-7B as the reward model. Notably, Qwen2.5-Math-7B, with just 32 particles, matches o1-preview on MATH500, demonstrating PF's effectiveness.

Dynamic Cheatsheet: Test-Time Learning with Adaptive Memory

Mirac Suzgun[π] **Mert Yuksekgonul**[π] **Federico Bianchi**[γ] **Dan Jurafsky**[π] **James Zou**[π,γ]

[π]Stanford University [γ]Together AI

ACL 2026

Dynamic Cheatsheet

Motivation:

- Current LMs typically operate without retaining insights from previous attempts;
- This study present Dynamic Cheatsheet (DC) is a lightweight framework that provides LLM an external, non-parametric, and evolving memory storage at inference time.

Dynamic Cheatsheet

- If correct / meaningful -> store the concise and transferable snippets
- If find a better one / error -> replace / delete
- Reorganize the memory and make it concise without repeats.

BL (Baseline)

1: **for** $i \in [1, \ldots, n]$ **do**
2: $\quad \tilde{y}_i = \texttt{Gen}(x_i)$ ▷ *Solution generation*
3: **end for**

DC-RS (Retrieval and Synthesis)

1: $M_0 \leftarrow \emptyset$ ▷ *Memory initialization*
2: **for** $i \in [1, \ldots, n]$ **do**
3: $\quad R_i = \texttt{Retr}(x_i, \{(x_j, \tilde{y}_j)\}_{j<i}, k)$ ▷ *Retrieval*
4: $\quad M_i = \texttt{Cur}(M_{i-1}, x_i, R_i)$ ▷ *Memory curation*
5: $\quad \tilde{y}_i = \texttt{Gen}(x_i, M_i)$ ▷ *Solution generation*
6: **end for**

DC-∅ (Empty Memory)

1: **for** $i \in [1, \ldots, n]$ **do**
2: $\quad M_i = \emptyset$ ▷ *Empty memory*
3: $\quad \tilde{y}_i = \texttt{Gen}(x_i, M_i)$ ▷ *Solution generation*
4: **end for**

DC-Cu. (Cumulative)

1: $M_0 \leftarrow \emptyset$ ▷ *Memory initialization*
2: **for** $i \in [1, \ldots, n]$ **do**
3: $\quad \tilde{y}_i = \texttt{Gen}(x_i, M_{i-1})$ ▷ *Solution generation*
4: $\quad M_i = \texttt{Cur}(M_{i-1}, x_i, \tilde{y}_i)$ ▷ *Memory curation*
5: **end for**

FH (Full History)

1: **for** $i \in [1, \ldots, n]$ **do**
2: $\quad M_i = \texttt{Concat}(\{(x_j, \tilde{y}_j)\}_{j<i})$ ▷ *Append full history*
3: $\quad \tilde{y}_i = \texttt{Gen}(x_i, M_i)$ ▷ *Solution generation*
4: **end for**

DR (Dynamic Retrieval)

1: **for** $i \in [1, \ldots, n]$ **do**
2: $\quad R_i = \texttt{Retr}(x_i, \{(x_j, \tilde{y}_j)\}_{j<i}, k)$ ▷ *Retrieval*
3: $\quad M_i = R_i$ ▷ *Memory contains only select examples*
4: $\quad \tilde{y}_i = \texttt{Gen}(x_i, M_i)$ ▷ *Solution generation*
5: **end for**

Figure 2: Dynamic Cheatsheet (DC)-based approaches. `Gen` represents the solution generator model, `Cur` the memory curator, and `Retr` the retriever. We use the same LLM for generation and curation with different prompts. Retrieval ranks historical inputs by cosine similarity, selecting the most relevant examples and solutions.

Dynamic Cheatsheet

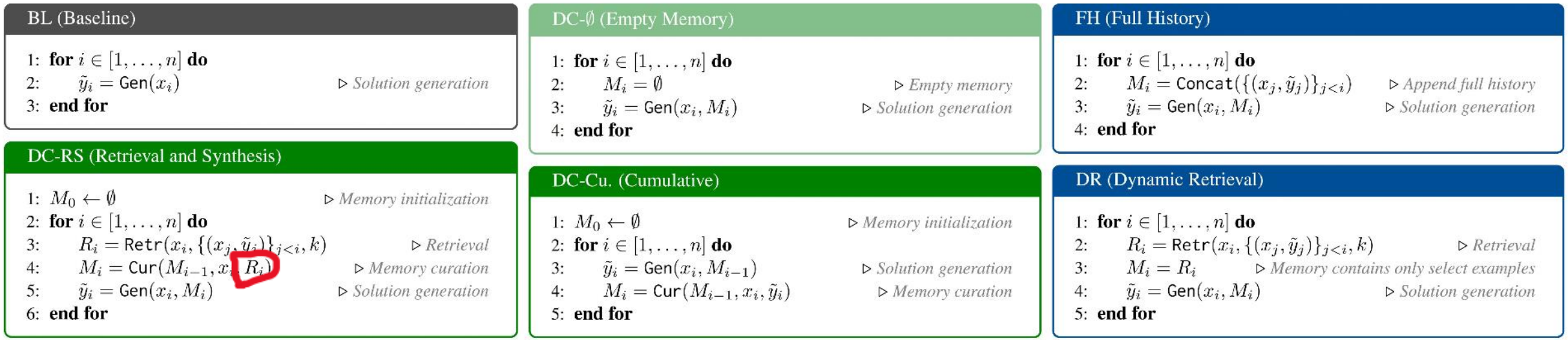


Figure 2: Dynamic Cheatsheet (DC)-based approaches. `Gen` represents the solution generator model, `Cur` the memory curator, and `Retr` the retriever. We use the same LLM for generation and curation with different prompts. Retrieval ranks historical inputs by cosine similarity, selecting the most relevant examples and solutions.

Dynamic Cheatsheet

How the memory are stored:

Dynamic Cheatsheet

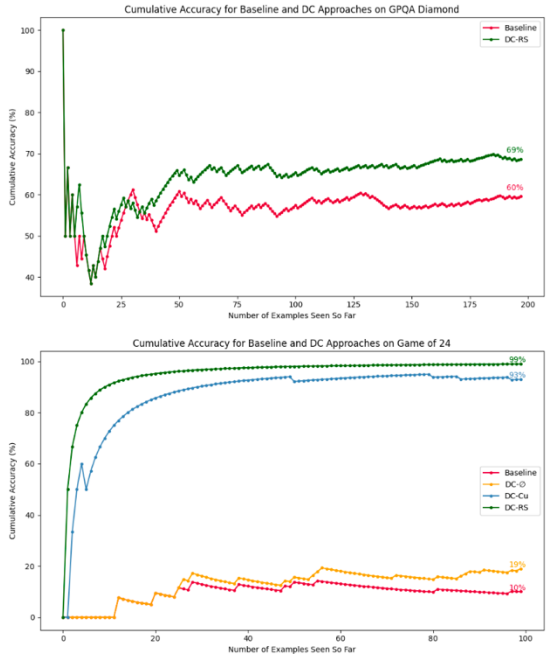


Figure 4: *Cumulative* performance progression under DC for GPQA-Diamond (*left*) and Game of 24 (*right*). In GPQA-Diamond, Claude 3.5 Sonnet steadily improves as it accumulates relevant knowledge snippets (first few points are noisy as y measures cumulative accuracy). Meanwhile, in Game of 24, GPT-4o rapidly gets near-perfect performance once it recognizes a Python-based solution.

Tasks	Claude 3.5 Sonnet					GPT-4o				
	BL	DC-∅	DR	DC-Cu.	DC-RS	BL	DC-∅	DR	DC-Cu.	DC-RS
AIME 2024	23.3	36.7	43.3	**50.0**	46.7	20.0	36.7	26.7	36.7	**40.0**
AIME 2025	6.7	23.3	23.3	**36.7**	30.0	6.7	10.0	10.0	16.7	**20.0**
AIME 2020–24	6.7	30.1	39.1	38.4	**40.6**	9.8	24.1	24.1	20.3	**24.8**
Game of 24	12.0	10.0	11.0	**14.0**	**14.0**	10.0	19.0	6.0	93.0	**99.0**
GPQA Diamond	59.6	60.1	63.6	61.1	**68.7**	**57.1**	57.1	55.1	58.1	57.1
Math Eqn. Balancer	44.8	56.4	60.4	**100**	97.8	50.0	88.0	**100**	**100**	99.2
MMLU Pro Eng.	61.2	57.2	65.2	66.8	**67.6**	**53.2**	51.6	48.8	44.0	51.2
MMLU Pro Physics	74.0	75.6	80.4	77.6	**82.0**	75.6	70.8	**75.6**	70.4	75.2

Table 1: Performance comparison of Dynamic Cheatsheet (DC) for Claude 3.5 Sonnet and GPT-4o across multiple benchmarks. **BL** (Baseline): standard inference without memory; **DC-∅** (Empty Memory): includes structured problem-solving and explicit tool-use instructions but no memory; **DR** (Dynamic Retrieval): uses retrieval but lacks memory updates; **DC-Cu** (Cumulative Memory): iteratively accumulates model solutions but lacks retrieval; and **DC-RS** (Retrieval & Synthesis): combines retrieval with memory refinement/synthesis. These results highlight accuracy gains under DC: Claude Sonnet's AIME 2024 accuracy jumps by 27% under DC-Cu, and GPT-4o's Game of 24 accuracy leaps from 10% to 99% under DC-RS.