Real-Time Progress Prediction in Reasoning Language Models

Anonymous ACL submission

Abstract

Recent advances in reasoning language models – particularly those that use long, latent chains of thought – have demonstrated remarkable capabilities in complex, agentic tasks. However, as these models operate over increasingly extended time horizons, their internal progress becomes opaque to users, complicating expectation management and real-time oversight. In this work, we investigate whether real-time progress prediction is feasible. We discretize progress and train a linear probe to classify reasoning states. We then introduce a two-stage fine-tuning approach that enables reasoning models to generate progress estimates (0→100%) during inference. Our best fine-tuned model achieves an average error of 10% for sequences less than 16,000 tokens, offering a practical mechanism for monitoring and interpreting model reasoning in real time.

1 Introduction

Test-time scaling has pushed horizons in natural language processing, leading to the development of so-called Reasoning Language Models (RLMs) (DeepSeek-AI et al., 2025; OpenAI et al., 2024), but RLMs introduce substantial user-interaction challenges: The resources consumed by a query fluctuate significantly, while the model's reasoning progress often remains opaque or unintelligible, making it hard for users to know how long they would have to wait for a task to complete. This problem is further compounded by the exponentially growing time horizons at which these models can operate coherently on agentic tasks (Kwa et al., 2025). While some have trained models to adhere to fixed predetermined reasoning budgets (Muennighoff et al., 2025; Aggarwal and Welleck, 2025; Jaghour et al., 2025), this strategy fails when budgets cannot be accurately inferred from the question alone, potentially causing models to return incorrect answers despite being capable of solving the problem given a larger reasoning budget.

Recent research suggests that RLMs encode confidence in their hidden states, well in advance of materializing a final answer. For example, Zhang et al. (2025) show that model hidden states contain "look-ahead" information – it is possible to predict eventual correctness

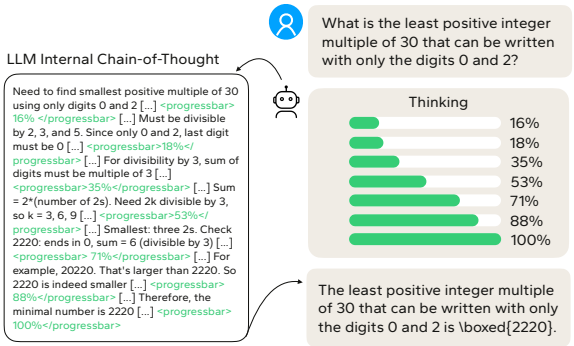

Figure 1: Real-time progress tracking during LLM reasoning. The model intermittently updates the user-facing progressbar by using `<progressbar>..</progressbar>`.

of a chain-of-thought, before a final answer has been generated – indicating that LLMs possess an inherent awareness of their reasoning trajectory. This raises the question: Beyond knowing if the solution is reachable, do models also encode the remaining computational distance to the final answer?

In this paper, we first evaluate if RLMs encode progress information in their hidden states by training probes to predict how far along the model is in completing the reasoning task. We discretize progress into 10 equally-sized bins and train a linear probe to predict the progress at its token position. We find evidence that progress is encoded to some extent in the internal representations, though extracting it precisely is not highly successful.

Therefore, we explore fine-tuning existing reasoning models to intermittently emit progress predictions within a dedicated progress bar during their reasoning process. We evaluate our approach on reasoning sequences spanning up to 32,000 tokens and demonstrate that our best fine-tuned model achieves a mean absolute error (MAE) of 10% on progress prediction for in-domain sequences (up to 16,000 tokens), which is the length the model was trained on. However, we find that prediction accuracy significantly degrades when evaluating on longer, out-of-domain sequences (up to 32,000 tokens), highlighting the challenge of maintaining progress estimation quality for extended reasoning chains beyond the training distribution. Our trajectory analysis reveals empirical bounds on progress

predictability. By sampling multiple rollouts from different positions, we study the dispersion in progress and find MAD to be around 5-10%, suggesting an empirical lower bound on prediction error. We experiment with introducing dedicated prediction tokens and we find that the backtracking naturally emerges in models trained with these tokens.

2 Related Work

Reinforcement Learning for LLMs Reinforcement Learning (RL) is a learning approach in which an agent, acting in an environment, attempts to maximize future discounted rewards over a policy (Sutton and Barto, 2018; Kaelbling et al., 1996). RL was popularized for large language models (LLMs) through Reinforcement Learning with Human Feedback (RLHF) (Ouyang et al., 2022), where the model is optimized using a reward model trained on human preference data (Lambert et al., 2024). More recently, RL has been directly applied to train language reasoning models (DeepSeek-AI et al., 2025). Group Relative Policy Optimization (GRPO) (Shao et al., 2024) has become a popular method for this purpose because it removes the need for a separate value model while maintaining stable training dynamics, unlike methods such as Direct Preference Optimization (DPO) (Rafailov et al., 2024). (See GPRO formulation in App. A).

Search Progress Beside the foundational work of Thayer et al. (2012), Sudry and Karpas (2021) are the first to apply deep learning methods to progress prediction. They convert the problem into a supervised learning task, training an LSTM to predict progress, conditioned on prior search history. They also distinguish between offline and online progress prediction. Offline prediction estimates, a priori, the total resources a sear algorithm will require to complete a task. In contrast, online prediction refers to iteratively estimating how far the algorithm has progressed, given its observed search history so far. In this work we focus on online estimation.

Beyond general search algorithms, progress estimation has been applied to specific domains. In neural theorem proving, Huang et al. (2025) trained a language model to predict the number of steps remaining to reach proof completion in LEAN (Moura and Ullrich, 2021). Similarly, Ma et al. (2019); Zhu et al. (2020) have explored progress estimation in vision-language navigation for robotics.

3 Problem Setup

Reasoning Language Models (RLMs) The output of an RLM contains two parts, a reasoning trace and a response. The reasoning trace (or Chain-of-Thought) is delimited by specific tokens like `<think>` and `</think>`, and contains a sequence of intermediate steps for solving the problem. Then, the model generates the response that contains the final answer to the user query. In all our experiments, we use DEEPSCALER-1.5B (Luo et al., 2025), a compact, domain-specific reasoning model based on DeepSeek-R1-Distilled-Qwen-1.5B (DeepSeek-AI et al., 2025), further post-trained with reinforcement learning on mathematical problems.

Progress Prediction To our knowledge, progress prediction was systematically studied for the first time in the field of heuristic search algorithms, with Thayer et al. (2012) as the seminal work. Thayer et al. (2012) define search progress as follows. Given a search algorithm A and a heuristic search problem P, let $E(P)$ denote the total number of nodes expanded by A when solving P. Let $\mathrm{Gen}(P)$ be the nodes already expanded by algorithm A and $\mathrm{Rem}(P, \mathrm{Gen}(P))$ be the number of remaining nodes that will be expanded i.e., $\mathrm{Rem}(P, \mathrm{Gen}(P)) = E(P) - \mathrm{Gen}(P)$.

Definition (Search Progress). The search progress of algorithm A solving problem P after expanding $\mathrm{Gen}(P)$ nodes is then:

$$\mathrm{Prog}(\mathrm{Gen}(P)) = \frac{\mathrm{Gen}(P)}{\mathrm{Gen}(P) + \mathrm{Rem}(P, \mathrm{Gen}(P))} \quad (1)$$
$$= \frac{\mathrm{Gen}(P)}{E(P)}$$

Since $\mathrm{Gen}(P) \leq E(P)$, search progress is thus a value between 0 and 1 and is scale invariant. This formalization carries smoothly over to the domain of reasoning, where we have a problem q and an algorithm A obtained by sampling from the policy, $P_\theta(\cdot|q, \texttt{<think>})$ parameterized by a reasoning model θ.

Let $\mathbf{C}_k = \{x_1, x_2, \ldots, x_k\}$ be a partial reasoning trace with length k and $x_i \sim P_\theta(\cdot \mid q, \texttt{<think>}, x_{<i})$ and let m be the length of the complete reasoning trace.

1. $\mathrm{Gen}(P) = |\mathbf{C}_k| = k$,
2. $\mathrm{Rem}(P, \mathrm{Gen}(P))$ is then the expected length over all possible sequences τ starting from $\mathbf{C}_k$ and ending in a terminal state `</think>`. i.e, $E_{\tau \sim P_\theta(\tau|\mathbf{C}_k)}[|\tau|]$ where $P_\theta(\tau|\mathbf{C}_k)$ is the conditional probability distribution over all such trajectories and $|\cdot|$ computes the length.

Since exact computation is intractable, we make the approximation $\mathbb{E}_{\tau \sim P_\theta(\tau|\mathbf{C}_k)}[|\tau|] \approx m - k$, thus, the progress is simply defined as:

$$\mathrm{Progress}(\mathbf{C}_k) = \frac{k}{k + \mathbb{E}_{\tau \sim P_\theta(\tau|\mathbf{C}_k)}[|\tau|]} \approx \frac{k}{m} \quad (2)$$

4 Is Progress Encoded in Hidden States?

To assess the extent to which existing reasoning model representations contain information about progress, we train linear probes to extract the progress.

Dataset We construct a reasoning dataset by randomly sampling 25K questions without replacement from OpenR1-Math (Face, 2025). For each question, we obtain a single reasoning trace by generating with DEEPSCALER-1.5B using temperature = 0.6 and top_p = 0.95 (Luo et al., 2025). We split the dataset in 3: train, eval and out-of-distribution (OOD) for reasoning traces above 16K tokens.

We discretize progress into 10 equally-sized bins (0-10%, 11-20%, ..., 91-100%). Formally, we assign a progress label to each token by quantizing its position k into 10 equally-spaced quantiles: $Q_q = [\frac{q-1}{10}, \frac{q}{10})$ for $q = 1, \ldots, 10$. A token at position k receives label q if $\frac{k}{m} \in Q_q$.

Probes We train two types of linear probes. Let $h_{i,j}$ be the hidden state at the j'th token of the i'th reasoning trace. We train:

1. A probe $f_{\text{token}} : \mathbb{R}^d \to \mathbb{R}^Q$ that maps the hidden state $h_{i,j}$ at each token position to a Q-dimensional output representing the logits for each quantized progress bucket.

2. A concatenated probe $f_{\text{q+token}} : \mathbb{R}^{n \cdot d + d} \to \mathbb{R}^Q$, where the input is $[h_{i,j}; e_i]$, the concatenation of the current token's hidden state and the the question embedding derived from the final n tokens preceding the assistant token. Throughout our experiments, we set $n = 2$.

The probe parameters θ are optimized using the cross-entropy loss over the Q progress buckets. For a given input $\mathbf{h}_{i,j}$ (either $h_{i,j}$ or $[h_{i,j}; e_i]$) and one-hot target label $y \in \{0,1\}^Q$, the loss becomes:

$$\mathcal{L}(\theta) = -\frac{1}{\sum_{i=0}^{B} T_i} \sum_{i=0}^{B} \sum_{j=0}^{T_i} \sum_{q=1}^{Q} y_{i,j,q} \times \log\left(\frac{\exp(f(\mathbf{h}_{i,j})_q)}{\sum_{k=1}^{Q} \exp(f(\mathbf{h}_{i,j})_k)}\right) \quad (3)$$

where $f(\mathbf{h}_{i,j})_q$ denotes the logit for bucket q at position j in sequence i and $y_{i,j,q}$ is the indicator for the true bucket.

Metrics We use top-1 accuracy to evaluate probe performance and MAE (mean absolute error), similar to other work (Huang et al., 2025; Ma et al., 2019; Zhu et al., 2020).

Results Table 1 shows that probe performance is relatively consistent across layers, with accuracy around 29% and MAE around 1.75, moreover as illustrated from the heatmap (Figure 2) the probability mass is concentrated around the ground truth label(the diagonal), especially in the beginning and end. For a single example we refer the reader to the appendix 11. This suggests that the model's hidden states do encode some notion of progress, and that simple probes can extract this information to a large degree. However, performance drops substantially on (OOD) sequences (Table 1), highlighting the challenge of generalization. The addition of question embeddings does not consistently improve results, indicating that most relevant information is already present in the hidden states at the given token position.

Probe	Acc.	MAE	OOD Acc.	OOD MAE
Layer 24				
$f_{\text{q+token}}$	0.283	1.702	0.132	2.834
f_{q}	0.290	1.696	0.129	2.970
Layer 16				
$f_{\text{q+token}}$	0.297	1.637	0.136	2.835
f_{q}	0.307	1.565	0.131	2.953
Layer 8				
$f_{\text{q+token}}$	0.281	1.773	0.131	2.904
f_{q}	0.285	1.745	0.129	2.998

Table 1: Linear probe performance on DEEPSCALER-1.5B at different layers. MAE is the mean absolute error when using the bin with the highest likelihood according to the model. OOD refers to all reasoning traces above 16K tokens. $f_{\text{q+token}}$ concatenates the hidden state with question embeddings, while f_{q} uses only the hidden state.

5 Incorporating Progress Estimation into the Reasoning Trace

Building on our earlier observations, we explore incorporating progress prediction directly into the model's training objective. To that end, we fine-tune DEEPSCALER-1.5B on reasoning traces augmented with progress reporting.

Data We use OpenR1-Math (Face, 2025), where the traces were generated using the larger DeepSeek-R1 (DeepSeek-AI et al., 2025).

We augment this data to contain progress reporting. Specifically, for each reasoning trace delimited by the thinking markers `<think>` and `</think>`, we:

1. Split the thinking portion into segments $s_1, \ldots, s_n$ using paragraph segmentation (Frohmann et al., 2024).

2. For each segment s_i, compute its token length l_i.

3. Calculate cumulative progress $a_i = 100 \cdot \left\lfloor \frac{\sum_{j=1}^{i} l_j}{\sum_{j=1}^{n} l_j} \right\rfloor$.

4. Insert progress markers `<progressbar>` a_i `</progressbar>` after each segment.

   The progress $a_i \in [0, 100]$ is tokenized using the model's existing vocabulary rather than being treated as a special token. We explore alternative approaches using dedicated progress tokens in Appendix 6.1.

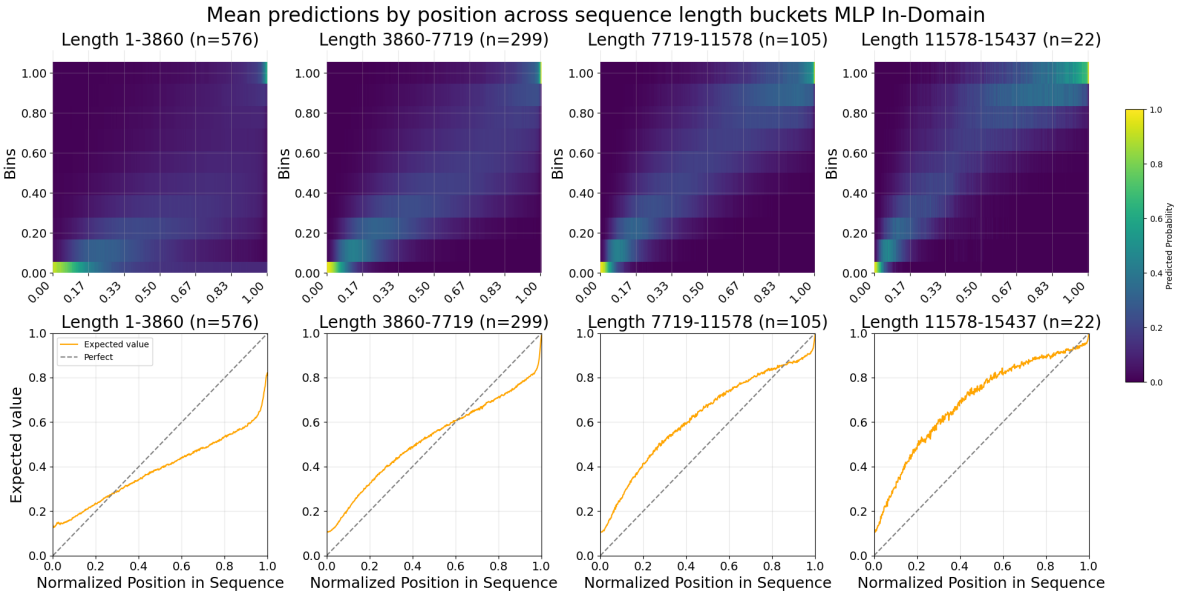

Figure 2: Heatmap of probability mass and expected value over the quantiles. We see that the expected value closely follows the diagonal, indicating well-calibrated progress estimates. A similar trend can be observed in the heatmap where most probability mass is around the diagonal. Entropy is lowest near the beginning—where most trajectories start similarly—and rises toward the middle as uncertainty about reasoning continuation peaks. As the sequence progresses, entropy decreases again, reflecting convergence toward termination.

For evaluation, we use MATH500 (Lightman et al., 2023), AMC23 (AMC, 2023) and a subset of OlympiadBench (He et al., 2024), that are popular benchmarks for mathematical reasoning and examine accuracy. This enables us to both study, how our fine-tuning technique impacts performance, as well as the progress prediction errors for different reasoning lengths.

5.1 Models

Supervised Fine-Tuning (SFT) We train using LoRA (Hu et al., 2021) on all linear layers, using $r = 256, \alpha = 256$. The loss for progress prediction tokens is scaled by a coefficient γ. Thus, let $x = (x_1, x_2, \ldots, x_T)$ be a sequence of length T, and J the set of all tokens within the `<progressbar>...</progressbar>` tags. Then, the loss is defined by:

$$L = \frac{1}{T}\sum_{i=1}^{T} -k \log P_\theta(x_i \mid x_{<i}) \tag{4}$$
$$k = 1 + (\gamma - 1) \cdot \mathbb{1}_{i \in J}$$

In our experiments we use $\gamma = 5$ and context length of 2048.

Noisy Labels We recognize a potential monotonicity bias in our approach , since the progress labels in the data are always incremental, it could lead to compounding errors when the model makes an early optimistic prediction. To address this potential monotonicity bias and better identify self-correction capabilities, we experiment with adding noise to the ground-truth progress labels a_i by sampling from a normal distribution:

$$\tilde{a}_i = a_i + \epsilon_i, \quad \text{where } \epsilon_i \sim \mathcal{N}(0, 10^2) \tag{5}$$

Parameter	Value
β	0
ϵ_{high}	0.28
ϵ	0.2
learning rate	1×10^{-6}
lr scheduler	constant with warmup
warmup steps	25
num train epochs	1
num generations	4
temperature	1.0
max steps	500
bf16	True
max grad norm	0.1
max completion length	8096
δ	4

Table 2: Hyperparameters used in all the experiments.

Reinforcement Learning (RL) We design three reward functions, evaluated on a batch of N model rollouts $\mathcal{O} = \{o_1, \ldots, o_N\}$ for a given question. For each output o_i, let $P_i = \{p_{i,t_1}, \ldots, p_{i,t_j}\}$ denote the sequence of predicted progress at each marker, and $A_i = \{a_{i,t_1}, \ldots, a_{i,t_j}\}$ the corresponding ground-truth progress fractions at those positions. Let $\hat{y}_i$ be the model's final answer in rollout o_i, and y_i the correct answer.[1] The three rewards are the following:

Progress Prediction Accuracy: Mean Absolute Error (MAE) between the predicted and true progress frac-

[1]We use the same system prompt as DEEPSCALER-1.5B and instruct the model to provide the final answer within \boxed{}.

tions, clipped to $[0, 1]$:

$$R_{\text{pred}}(o_i) = 1 - \frac{1}{|P_i|} \sum_{j=1}^{|P_i|} \left| \text{clip}(p_{i,j}, 0, 1) - \frac{a_{i,j}}{100} \right| \quad (6)$$

$$\text{where,} \quad \text{clip}(x, 0, 1) = \max(0, \min(x, 1))$$

Progress Report Frequency: Rewards regular insertion of progress markers every $\tau = 250$ tokens, and penalizes progress markers that are placed outside the `<think>` ... `</think>` region:

$$R_{\text{format}}(o_i) = 1 - P_{\text{freq}}(o_i) - P_{\text{outside}}(o_i)$$

$$P_{\text{freq}}(o_i) = \frac{1}{T_i} \sum_{j=1}^{T_i} \min\left(\frac{|t_{i,j} - t_{i,j-1} - \tau|}{\tau}, 1 \right) \quad (7)$$

$$P_{\text{outside}}(o_i) = \min(0.1 \cdot N_{\text{outside}}, 0.5)$$

where $t_{i,j}$ is the token position of the j-th progress marker, T_i is the total number of markers, and N_{outside} is the number of progress markers that appear outside the thinking region. Note that the reward term (without the P_{outside} penalty) is maximized at 1 when all consecutive markers are separated by exactly τ tokens, i.e., $t_{i,j+1} - t_{i,j} = \tau$ for all j.

Mathematical Correctness: Binary reward for correctly solving the underlying problem:

$$R_{\text{math}}(o_i) = \mathbb{1}[\hat{y}_i = y_i] \quad (8)$$

Finally, the total reward for output o_i is a linear combination of individual rewards:

$$R(o_i) = \lambda_1 R_{\text{pred}}(o_i) + \lambda_2 R_{\text{format}}(o_i) + \lambda_3 R_{\text{math}}(o_i)$$

where $\lambda_1, \lambda_2, \lambda_3$ are weighting coefficients. For our experiments we use $\lambda_1 = \lambda_3 = 1, \lambda_2 = \frac{1}{2}$. All hyperparamters can be found in Table 2.

5.2 Results

Progress prediction error As shown in Table 3, combining SFT with noisy labels and RL yields the strongest overall performance. While RL does not consistently reduce MAE across all categories, it substantially improves accuracy for longer sequences (Figure 3). In particular, RL helps the model maintain stable progress estimates beyond 16K tokens and prevents severe overpredictions—especially for incorrect responses—where SFT alone often produces values exceeding 100% (Figure 4, Appendix 10). For a complete reasoning trace example we refer the reader to the Appendix C.

Progress prediction vs reasoning ability As shown in Table 4, adding progress prediction to the base model leads to a measurable decline in reasoning performance. This may occur because the model optimizes for progress prediction accuracy and formatting consistency, which can divert capacity away from mathematical correctness. However, it remains unclear whether this degradation stems from the multitask setup itself or

Figure 3: Prediction Error (MAE) vs. Reasoning Trace Length (binned into 25 groups). MAE increases as the length increases. All predictions were constrained to the interval (0,100).

from hardware constraints during training. Specifically, our RL training for DeepScaleR-1.5B was limited to 8,096-token completions, whereas the original reasoning model was trained on sequences up to 24K tokens. The shorter context window likely restricted reasoning depth, making it difficult to isolate the effects of the approach from those of compute limitations.

5.3 Comparing Probes and Finetuned Models

To compare methods, we analyze reasoning traces from our SFT model (without noisy labels) on MATH500, AMC23, and OlympiadBench. We compute the MAE between predicted and true progress at each step. For the probe, we remove the progress reporting tokens and use DeepScaleR-1.5B's hidden states during next-token prediction. Finally, we estimate progress by taking the top-1 probability bin.

While the finetuned model's text may be slightly out-of-distribution for the probe—since it was trained on a different dataset—this provides the most direct comparison between the two approaches.

We find that the SFT model performs better than the probe-based approach across all configurations (Table 5). The SFT model achieves an MAE of 15.27%, substantially outperforming the best probe configuration (L8 with f_{token}, MAE of 20.55%), when examining reasoning traces below 16K tokens. This suggests that explicit training for progress prediction through fine-tuning enables the model to develop more robust progress estimation capabilities.

6 Further Improvements

6.1 Using custom prediction tokens enables backtracking

Instead of representing the progress values inside the `<progressbar>` and `</progressbar>` tags

	<16K Correct		<16K Incorrect		>16K Correct		>16K Incorrect	
Model	**Raw**	**Clip**	**Raw**	**Clip**	**Raw**	**Clip**	**Raw**	**Clip**
Random Baseline	0.333	0.333	0.333	0.333	0.333	0.333	0.333	0.333
SFT (Noisy Labels)	0.104	0.104	0.136	0.132	0.206	0.206	0.288	0.256
SFT	0.224	0.139	0.265	0.157	0.796	0.491	4.027	0.386
SFT + RL	0.102	0.102	0.133	0.132	0.232	0.232	0.343	0.264
SFT (Noisy Labels) + RL	0.101	0.101	0.128	0.128	–	–	0.271	0.265

Table 3: Progress prediction performance (MAE) across different model configurations and sequence length categories. Note that predictions (0-100%) are normalized to 0-1 by dividing by 100, hence an MAE of 0.1 corresponds to an MAE of 10%. Raw refers to MAE error, when we do not clip the predictions and Clip refers to MAE error when we apply $clip(\cdot, 0, 1)$.

Model	**MATH500**	**AMC23**	**OlympiadBench**
DEEPSCALER-1.5B	63.0	**80.0**	**22.0**
SFT (Noisy Labels)	**64.2**	60.0	19.3
SFT	63.0	62.5	13.3
RL (Noisy Labels)	62.6	67.5	16.0
RL	61.4	72.5	16.0

Table 4: Mathematical performance (accuracy). For OlympiadBench we use a subset of the original dataset.

Configuration	**MAE**
L8 (f_{token})	20.55
L8 ($f_{\text{q+token}}$)	21.27
L16 (f_{token})	21.49
L16 ($f_{\text{q+token}}$)	22.78
L24 (f_{token})	21.59
L24 ($f_{\text{q+token}}$)	26.83
SFT	**15.27**

Table 5: Comparison of Linear-probe (f) performance at different layers and SFT model on sequences less than 16K tokens

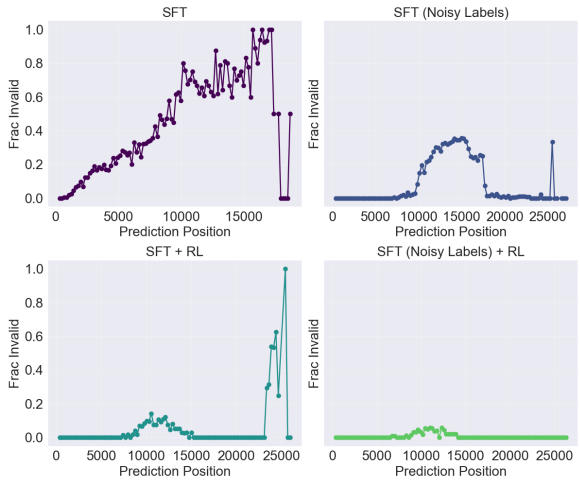

Figure 4: Fraction of invalid predictions (containing progress above 100%), across MATH500, AMC23 and OlympiadMath dataset. We observe that with SFT alone, the fraction of invalid predictions gets to 100% at 15K tokens, while the model combining noisy labels and reinforcement learning almost never produces invalid predictions, even at sequence lengths up to 25K tokens.

using normal tokens (which would lead to predictions like <progressbar>10%</progressbar>, where '1', '0', and '%' are treated as separate tokens), we extend the model's vocabulary to include special tokens for each percentage value: [0%], [1%], ..., [100%]. This way, the model is trained to predict <progressbar>[10%]</progressbar> instead. Our hypothesis is that reusing tokens such as '1' and '%' for both the progress prediction task and standard language modeling introduces ambiguity, which may cause the model to behave incorrectly, for example, by predicting values greater than 100%.

In Figure 5, we analyze the fraction of non-monotonic predictions, defined as $\frac{1}{n}\sum_{i=0}^{n-1} \mathbb{1}pi + 1 < p_i$. We find that as the model progresses through its reasoning, the fraction of predictions that decrease relative to prior estimates (i.e., backtrack) increases. This ability to revise earlier predictions is notable, given that the training data contained only monotonically increasing progress markers.

One subtle advantage of not using special tokens approach is that, after generating <progressbar>1, the model has another forward pass to choose the exact value, for example, 11 or 19.

6.2 Masking Previous Progress Improves Predictions

The model could exploit regular spacing of progress markers in the dataset, estimating progress from token position rather than sequence context. For example, it could compute a scaling factor $\beta = \frac{1}{k}\sum_{i=0}^{k-1} \frac{p_{i+1}-p_i}{pos(p_{i+1})-pos(p_i)}$ and approximate $p_{k+1} \approx p_k + \beta \cdot (pos(p_{k+1}) - pos(p_k))$, where $pos(\cdot)$ is the model's estimate of absolute token position. We explore with masking progress markers, which acts as a form of regularization, preventing this shortcut and encouraging the model to infer progress from surrounding context.

We apply a cosine warm-up sched-

Model	Dataset	MAE	MAE (clip)	Acc
w/ special	MATH500	0.201	-	0.646
SFT	MATH500	0.291	0.209	0.630
w/ special	OlympiadMath	0.237	-	0.167
SFT	OlympiadMath	0.340	0.223	0.153
w/ special	AMC23	0.212	-	0.700
SFT	AMC23	0.299	0.256	0.625

Table 6: Performance comparison between SFT model with and without custom prediction tokens.

Model	Dataset	MAE	Accuracy
w/o masking	MATH500	0.2012	**0.6460**
w/ masking	MATH500	**0.1874**	0.6240
w/o masking	OlympiadMath	0.2374	**0.1667**
w/ masking	OlympiadMath	**0.2141**	0.1533
w/o masking	AMC23	0.2121	0.7000
w/ masking	AMC23	**0.1958**	**0.7750**

Table 7: Performance comparison between models with custom predictions tokens with and without masking schedule

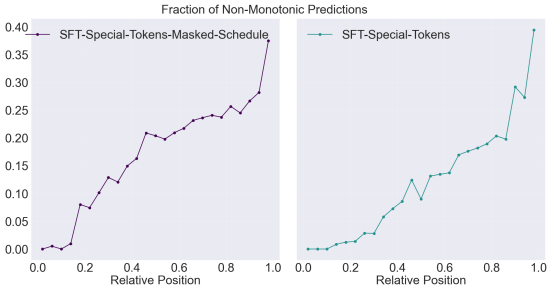

Figure 5: Fraction of non-monotonic predictions for different positions in the reasoning trace.

Setting	MAE	MAE (Filtered <100%)
Unconditioned	**0.218**	0.176
Conditioned	0.297	**0.173**

Table 8: Conditional vs marginal prediction performance on AMC23 for 'SFT' model. Reports MAE over all predictions after clipping predictions to [0,1]. "<100%" reports MAE computed only on predictions where the conditional predictor's raw output is below 1 (i.e., excludes invalid >100% predictions).

ule from 0 to 0.5, masking tokens inside `<progressbar></progressbar>` spans with up to 50% probability.

This ensures the model frequently trains without direct access to progress values while gradually increasing exposure over training.

We find that the masking is effective for reducing the prediction error (MAE) while reasoning performance stays roughly constant (Table 7.

7 Analyses

7.1 How Predictable is Progress?

RLM inferences are stochastic processes. In that light, what is a reasonable lower bound on the prediction error for progress estimation? Given a question q and corresponding reasoning trace $\mathbf{C}$, we take $n = 12$ random prefixes $\mathbf{C}^k$ from the trace and generate $t = 8$ rollouts $\tau_1^k, \ldots, \tau_t^k$ sampled from $P(\cdot|q, \mathbf{C}^k)$ with temperature 0.6 as recommended in (Luo et al., 2025). We examine 500 sequences in total sampled by DeepScaleR-1.5B from MATH500. For each prefix $\mathbf{C}^k$, we estimate progress from each rollout as $P_{k,j} = \frac{|\mathbf{C}^k|}{|\mathbf{C}^k|+|\tau_j^k|}$.

To better understand the dispersion of possible "progress estimates" we compare against the mean $\mu_k = \frac{1}{t}\sum_{j=0}^{t} P_{k,i}$ and compute the Mean Absolute Deviation (MAD) measures the average absolute residual, analogous to the variance which instead averages the squared residuals.

$$\text{MAD} = \frac{1}{t \cdot n} \sum_{k=1}^{n} \sum_{j=1}^{t} |\mu_k - P_{k,j}| \tag{9}$$

We additionally examine Mean Absolute Percentage Deviation (MAPD), which is similar to MAD, except the residuals are normalized by the mean μ_k

$$\text{MAPD} = \frac{1}{t \cdot n} \sum_{k=1}^{n} \sum_{j=1}^{t} \frac{|\mu_k - P_{k,j}|}{\mu_k} \tag{10}$$

We observe that the MAD is around 5%-10% suggesting this is a lower bound for how well progress can be predicted (Figure 6. The MAPD, in contrast, decreases monotonically with normalized position, indicating that dispersion in possible continuation lengths diminishes as reasoning unfolds. Early in the reasoning process, MAPD can reach up to 25%, implying that progress predictions can deviate substantially from the mean when little of the reasoning trace is observed.

7.2 Do Prior Predictions Matter?

We compare two settings for progress prediction: (i) *Conditioned*, where the model attends to its own prior progress predictions, and (ii) *Unconditioned*, where predictions are made independently of any prior ones (a marginal predictor). We evaluate our supervised fine-tuned model (`sft-no-noise`) on AMC23 reasoning traces.

Overall, the *Unconditioned* (independent) predictor achieves lower error (0.218 vs. 0.297 MAE). However, once we exclude steps where the conditional model overshoots and produces invalid progress (>100%), performance becomes comparable (0.176 vs. 0.173 MAE). This indicates that most of the conditional model's deficit comes from events, overshooting.

In principle, conditioning on more information should not increase uncertainty. In domains or configurations

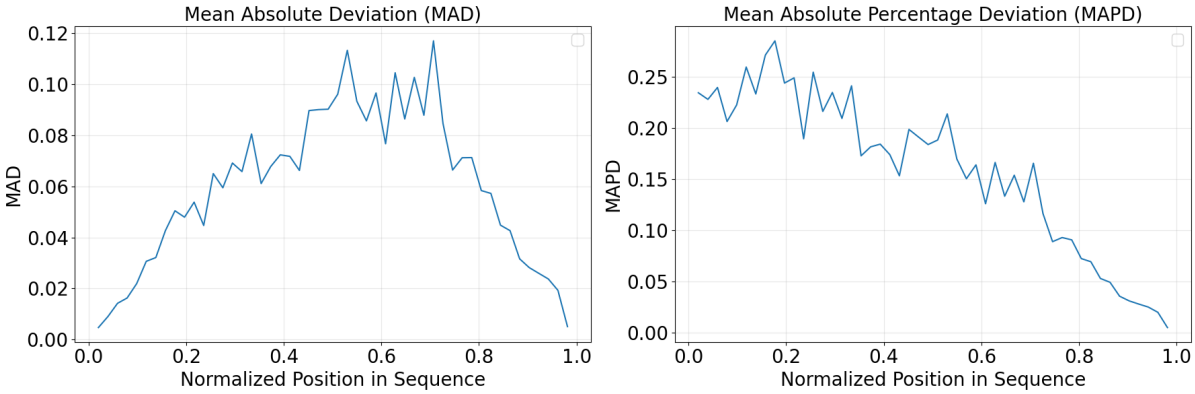

Figure 6: MAD (Equation 9 and MAPD (Equation 10), we compute the mean over 50 bins

where such overshoot is rare, conditioning on past predictions yields slightly better accuracy, consistent with the information-theoretic intuition.

8 Conclusion and Future Research

In this paper, we explore methods for predicting progress in reasoning models. Our experiments demonstrate that current models encode a notion of progress in their internal representations, with a linear probe achieving close to 30% accuracy over 10 classes, moreover the errors appear highly localized (MAE 1.75).

Motivated by these observations, we introduce a simple strategy for directly incorporating progress prediction into RLMs through a 2-step process of SFT and RL - and demonstrate that at smaller scales (below 16K tokens), the approach nearly saturates the theoretical lower bound on prediction error: since rollout-based progress estimates deviate by 5%-10% from any given trajectory due to inherent stochasticity, our model's performance approaches this irreducible error floor.

Future Research Directions We find, after having applied our post-training recipe, that model reasoning performance seems to degrade slightly. Future research should further study this phenomenon: is there an inherent tradeoff between these two tasks, or is degraded performance an artifact of our training methodology or limited computational resources?

The field of Large Language Models (LLMs) has consistently demonstrated that scaling model size, dataset quality, and training compute leads to qualitatively new capabilities and improved sample efficiency Kaplan et al. (2020); Hoffmann et al. (2022); Wei et al. (2022). Thus, we believe that extending our approach to larger models represents a particularly promising research direction. Scaling could not only enhance predictive accuracy and calibration, but also reveal whether progress-awareness and self-monitoring abilities emerge more robustly at higher capacity regimes. Investigating this scaling behavior may further shed light on the interplay between model scale, reasoning depth, and internal representations of progress.

This paper focuses on mathematical reasoning, which at face-value seems to be one of the most challenging domains for progress prediction (i.e., it seems harder to forecast how far we are from solving the Riemann hypothesis than how long it will take a person to go to the supermarket) — applying these principles to other domains such as multi-turn agents (Jin et al., 2025) could be fruitful area of research.

Limitations

Our work has several limitations. First, due to a single-model focus, all empirical results center exclusively on DeepScaleR-1.5B, meaning our conclusions may not generalize to other base models, sizes, or training recipes. Second, there is domain narrowness; both training and evaluation are heavily focused on mathematical reasoning problems (using datasets like OpenR1-Math, MATH500, AMC23, and OlympiadBench), making it uncertain how well the studied techniques apply outside this domain. Third, we observed significant out-of-domain (OOD) degradation where prediction performance degrades meaningfully as reasoning traces become excessively long, exceeding training lengths. Finally, our compute limitations meant the Reinforcement Learning phase was constrained to shorter completion limits (8,096 tokens) compared to the base DeepScaleR-1.5B model, which was pre-trained with longer limits (24,000 tokens).

References

Pranjal Aggarwal and Sean Welleck. 2025. L1: Controlling how long a reasoning model thinks with reinforcement learning. *Preprint*, arXiv:2503.04697.

AMC. 2023. Amc 2023 mathematics competition problems (via hugging face dataset). Hugging Face Datasets. Dataset created by math-ai; Accessed: October 6, 2025.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang,

Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Hugging Face. 2025. Openr1-math-220k. https://huggingface.co/datasets/open-r1/OpenR1-Math-220k.

Markus Frohmann, Igor Sterner, Ivan Vulić, Benjamin Minixhofer, and Markus Schedl. 2024. Segment any text: A universal approach for robust, efficient and adaptable sentence segmentation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11908–11941, Miami, Florida, USA. Association for Computational Linguistics.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, and 1 others. 2024. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. *arXiv preprint arXiv:2402.14008*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, and 3 others. 2022. Training compute-optimal large language models. *Preprint*, arXiv:2203.15556.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *Preprint*, arXiv:2106.09685.

Suozhi Huang, Peiyang Song, Robert Joseph George, and Anima Anandkumar. 2025. Leanprogress: Guiding search for neural theorem proving via proof progress prediction. *Preprint*, arXiv:2502.17925.

Prime Intellect Team: Sami Jaghour, Justus Mattern, Jack Min Ong, Jannik Straube, Manveer Basra, Aaron Pazdera, Matthew Di Ferrante, Kushal Thaman, Felix Gabriel, Fares Obeid, Kemal Erdem, Michael Keblinger, and Johannes Hagemann. 2025. Intellect-2: A reasoning model trained through globally decentralized reinforcement learning. Preprint.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *Preprint*, arXiv:2503.09516.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. 1996. Reinforcement learning: A survey. *Preprint*, arXiv:cs/9605103.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *Preprint*, arXiv:2001.08361.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Harold Miles, Seraphina Nix, Tao Lin, Neev Parikh, and 6 others. 2025. Measuring ai ability to complete long tasks. *Preprint*, arXiv:2503.14499.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. 2024. Rewardbench: Evaluating reward models for language modeling. *Preprint*, arXiv:2403.13787.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl. https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-M Notion Blog.

Chih-Yao Ma, Jiasen Lu, Zuxuan Wu, Ghassan AlRegib, Zsolt Kira, Richard Socher, and Caiming Xiong. 2019. Self-monitoring navigation agent via auxiliary progress estimation. *Preprint*, arXiv:1901.03035.

Moura and Ullrich. 2021. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pages 625–635, Cham. Springer International Publishing.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *Preprint*, arXiv:2501.19393.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, and 244 others. 2024. Openai o1 system card. *Preprint*, arXiv:2412.16720.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *Preprint*, arXiv:2203.02155.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Preprint*, arXiv:2305.18290.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *Preprint*, arXiv:2402.03300.

Matan Sudry and Erez Karpas. 2021. Learning to estimate search progress using sequence of states. In *ICAPS 2021 Workshop on Heuristics and Search for Domain-independent Planning*.

Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

Jordan T. Thayer, Roni Stern, and Levi H. Lelis. 2012. Are we there yet? – estimating search progress. In *Proceedings of the International Symposium on Combinatorial Search (SOCS)*. AAAI Press.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent abilities of large language models. *Preprint*, arXiv:2206.07682.

Anqi Zhang, Yulin Chen, Jane Pan, Chen Zhao, Aurojit Panda, Jinyang Li, and He He. 2025. Reasoning models know when they're right: Probing hidden states for self-verification. *Preprint*, arXiv:2504.05419.

Fengda Zhu, Yi Zhu, Xiaojun Chang, and Xiaodan Liang. 2020. Vision-language navigation with self-supervised auxiliary reasoning tasks. *Preprint*, arXiv:1911.07883.

A GRPO

GRPO Objective. Let q be a prompt and $\{o_0 \dots o_i \dots o_G\}$ rollouts sampled from $P_\theta(\cdot|q, o_{i,<t})$ where θ is our policy network. Then $\hat{A}_{i,t}$ is the advantage for token t in rollout i. We define $r_{i,t}(\theta) = \frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}$ as the probability ratio between the new and old policies.

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim P(Q),\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(O|q)} \Bigg[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\Bigg\{ \min\Big(r_{i,t}(\theta)\hat{A}_{i,t}, \text{clip}\left(r_{i,t}(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_{i,t}\Big) - \beta D_{\text{KL}}[\pi_\theta(\cdot|q)\|\pi_{\text{ref}}(\cdot|q)]\Bigg\}\Bigg] \quad (11)$$

The clip function restricts the probability ratio to the interval $[1-\epsilon, 1+\epsilon]$, with ϵ a small positive constant (e.g., 0.2), to avoid excessively large policy updates. KL divergence between the probability distributions is added as a regularization term.

Shao et al. (2024) introduced group relative advantages, where the advantage is computed as the group relative reward $\hat{A}_{i,t} = \frac{r_i - \text{mean}(\mathbf{r})}{\text{std}(\mathbf{r})}$.

B Figures

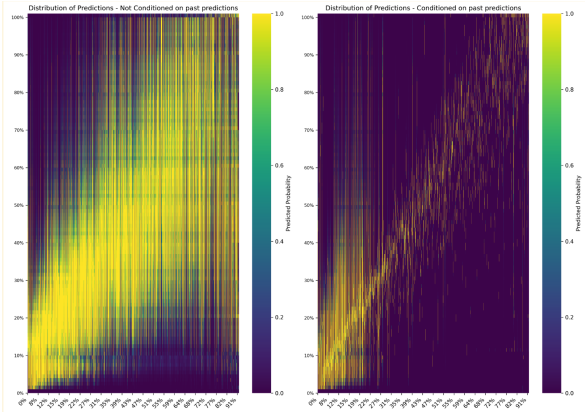

Figure 7: Heatmap visualization of predicted probability distributions across multiple sequences. Each column represents data points at a specific actual progress position (x-axis), while the y-axis shows the predicted progress bins (0%-100%). The color intensity indicates the probability mass assigned to each prediction bin, aggregated across all sequences in the dataset

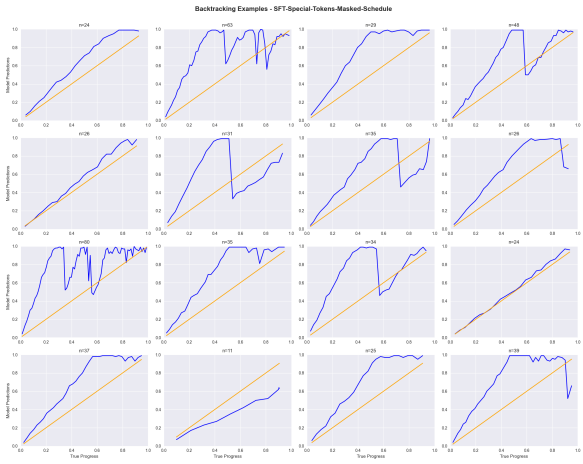

Figure 8: 16 examples of backtracking, the line with slope 1 indicates the perfect prediction for the model trained with special tokens and a masking schedule

C Example

Below is an example of a reasoning trace from the sft (No Noise) model.

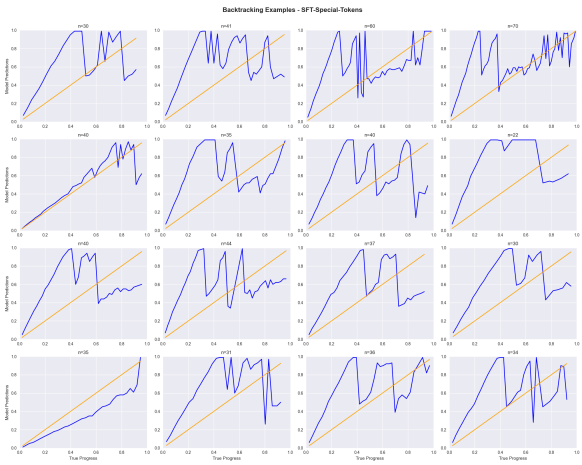

Figure 9: The figure shows examples of predictions during different reasoning trajectories

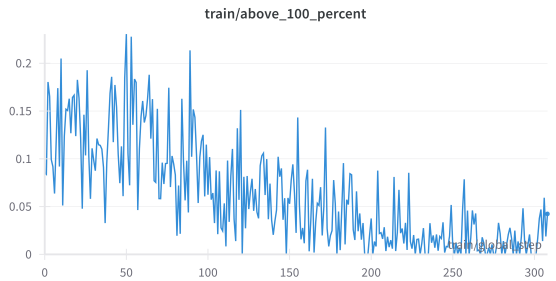

Figure 10: Percent of model predictions above 100 percent, during training

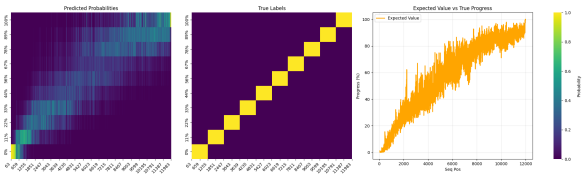

Figure 11: Visualization of linear probe prediction (Layer 16, $f_{\text{probe}}^{\text{concat}}$) for a 12K token sequence. Left: probability distribution over progress quantiles at different token positions. Center: True progress labels at different token positions. Right: The expected value over progress buckets, reducing prediction to a single number. The probe successfully tracks the upward trajectory of true progress throughout the reasoning sequence.

<think>\new
Okay, so I need to figure out how many different values I can get by inserting parentheses in the expression 2*3*4*5 + 1. The problem says that changing the order of operations by adding parentheses can give different results. For example, they gave (2*(3*4))*(5+1) = 144, which is different from the original 121. The question is asking for the total number of distinct values possible through all such insertions.\new
First, let me recall that parentheses can change the order in which operations are performed. Since multiplication has higher precedence than addition, inserting parentheses can force certain operations to happen first. The original expression is 2*3*4*5 + 1. If I just follow the order without parentheses, it's (2*3*4*5) + 1 = 120 + 1 = 121. But by inserting parentheses, we can group different operations together, which might lead to different results.\new
The key here is to consider all possible ways to parenthesize the expression. Each insertion of parentheses can potentially change the order of operations. <progressbar>2%</progressbar> The original expression has four numbers: 2, 3, 4, 5, and the +1 at the end. So, how many different parenthesizations are there?\new
I remember that for an expression with n numbers and k operations, the number of different parenthesizations is given by the Catalan numbers. But here, the operations are fixed: multiplication and addition. However, the problem allows inserting parentheses to change the order of operations, but the numbers and operations are fixed. So maybe it's similar to inserting parentheses in a binary tree structure.\n
Wait, let's think. The original expression is 2*3*4*5 + 1. The operations are multiplication between 2 and 3, 3 and 4, 4 and 5, and then addition of 1. But inserting parentheses can change where the additions are performed.\new
Alternatively, maybe we can model this as a binary tree where each multiplication is a node with two children, and the addition is a node with one child. But since we have fixed operations, the structure is fixed. However, inserting parentheses can group different operations together.\new
Wait, perhaps another approach: <progressbar>4%</progressbar> The original expression is a sequence of multiplications followed by an addition. If we insert parentheses, we can group some multiplications together, which might affect the order in which additions happen.\new
Let me consider all possible ways to parenthesize the expression. The original expression is 2*3*4*5 + 1. The possible parenthesizations would involve grouping different parts together. For example:\new
1. Group all multiplications first: ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
2. Group some multiplications first and then the addition: (2*3*4)*(5 + 1) = 24*6 = 144\new
3. Alternatively, maybe group the addition first? But addition has lower precedence, so unless we have parentheses around the addition, it's not possible. So the addition must be done after all multiplications. Wait, but the original expression is 2*3*4*5 + 1. If we insert parentheses to group the addition first, but since addition is lower precedence, we can't do that. <progressbar>6%</progressbar> So the addition must be done after all multiplications.\new
Wait, but the original expression is written as 2*3*4*5 + 1. If we insert parentheses to group the addition first, it would have to be something like 2*3*4*(5 + 1), but that's already parenthesized as ((2*3*4)*6) which is different. Wait, but the original expression is written as 2*3*4*5 + 1. So maybe the structure is that all multiplications are done first, and then the addition. But that's the original expression.\n
But the problem allows inserting parentheses, so we can group different operations. For example, if we group 5 + 1 first, but addition has lower precedence. So unless we have parentheses around the addition, it's not possible. Therefore, the addition must be done after all multiplications. Therefore, the only way to group operations is to group some multiplications together, which would change the order in which the addition is performed.\n
Wait, maybe not. Let's think of the expression as a sequence of operations. <progressbar>8%</progressbar> The original expression is 2*3*4*5 + 1. The operations are multiplication and addition. The structure is: 2 * 3 * 4 * 5 + 1. If we insert parentheses, we can group some multiplications first, which would change the order in which additions happen. For example, if we group 2*3 first, then multiply by 4, then multiply by 5, and then add 1. But that's the same as the original expression. Alternatively, if we group 3*4

728

first, then multiply by 5, then add 1. But again, that’s the same as the original expression. Wait, perhaps inserting parentheses can allow us to perform operations in different orders.\new
Wait, perhaps the key is that when you have multiple multiplication operations, you can choose which ones to group first. For example, in the original expression, there are three multiplication operations: 2*3, 3*4, 4*5. If we group some of these first, the addition might be done after. Let me try to list all possible parenthesizations.\new
<progressbar>10%</progressbar> The original expression is 2*3*4*5 + 1.

Let’s consider the possible groupings:\n
1. All multiplications first: ((2*3)*4)*5 + 1 = 120 + 1 = 121\n
2. Group 2*3*4 first: (2*3*4)*5 + 1 = 24*5 + 1 = 120 + 1 = 121. Wait, that’s the same as before. Hmm. So that’s not different.\n
Wait, maybe grouping 3*4 first: 2*(3*4)*5 + 1 = 2*24*5 + 1 = 48*5 + 1 = 240 + 1 = 241. Wait, that’s different. So that’s another value.\n
Similarly, grouping 4*5 first: 2*3*(4*5) + 1 = 2*3*20 + 1 = 6*20 + 1 = 120 + 1 = 121. <progressbar>12%</progressbar> Again, same as original.\n
Alternatively, grouping 2*3*4*5 first: but that’s just 120, then add 1. But the original expression is 2*3*4*5 + 1, so maybe that’s not allowed. Wait, the original expression is written as 2*3*4*5 + 1, so perhaps we can’t group the entire expression first. So maybe the possible groupings are:\new
- Group 2*3 first: ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
- Group 3*4 first: 2*(3*4)*5 + 1 = 24*5 + 1 = 121\new
- Group 4*5 first: 2*3*(4*5) + 1 = 6*20 + 1 = 121\new
- Group 2*3*4 first: (2*3*4)*5 + 1 = 24*5 + 1 = 121\new
- Group 2*3 first and 4*5 first: 2*(3*4*5) + 1 = 2*60 + 1 = 121\new
- Group 3*4 first and 5: 2*3*(4*5) + 1 = same as above.\new
<progressbar>15%</progressbar> Wait, maybe there are more possibilities.

Let’s think step by step.\n
The expression is 2*3*4*5 + 1. Let’s consider the possible ways to parenthesize. The operations are multiplication and addition. The structure is a binary tree with four multiplications and one addition. However, the addition is at the end, so it’s a left-associative operation. But by inserting parentheses, we can change the order in which the additions are performed.\new
Wait, perhaps we can model this as inserting parentheses in different places. Let me recall that in such expressions, the number of different parenthesizations is given by the Catalan numbers, but here, since the operations are fixed, maybe the number is different.\new
Wait, the original expression is 2*3*4*5 + 1. The Catalan number for n=4 is 14, but that’s for different parenthesizations with all possible operations. However, in our case, the operations are fixed: multiplication and addition. So maybe the number of distinct values is less.\new
Alternatively, perhaps the problem is similar to evaluating the expression with different groupings. <progressbar>17%</progressbar> Let’s try to list all possible parenthesizations.\n
First, note that the original expression is 2*3*4*5 + 1. Let’s consider all possible ways to group the multiplications and additions.\new
1. All multiplications first: ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
2. Group 2*3 first, then multiply by 4, then add 5: 2*3*(4*5) + 1 = 2*3*20 + 1 = 120 + 1 = 121\new
3. Group 3*4 first, then multiply by 5, then add 1: (2*3*4)*5 + 1 = 24*5 + 1 = 121\new
4. Group 4*5 first, then multiply by 2, then add 1: 2*3*(4*5) + 1 = same as above, 121\new
Wait, so all these groupings result in 121. <progressbar>19%</progressbar> That can’t be right. The problem states that inserting parentheses can give different results. So maybe there are other groupings.\new
Wait, perhaps if we group some multiplications and then the addition. For example, 2*(3*4)*5 + 1 = 2*24*5 + 1 = 240 + 1 = 241. That’s a different value. So that’s another value.\n
Similarly, 3*(2*4)*5 + 1 = 3*8*5 + 1 = 120 + 1 = 121. Hmm, same as before.\new
Alternatively, 2*3*(4*5) + 1 = 2*3*20 + 1 = 120 + 1 = 121. Again, same.\new
Wait, so perhaps the key is that when you group different multiplications together, you can get different intermediate results. Let’s see.\n
Let me try to list all possible parenthesizations. The expression is 2*3*4*5 + 1. <progressbar>21%</progressbar> The operations are multiplication and

```
addition. The original expression is left-associative, so all
multiplications happen before the addition. However, by inserting
parentheses, we can change the order in which multiplications are done.\new
Possible parenthesizations:\new
1. ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
2. 2*(3*4)*5 + 1 = 2*24*5 + 1 = 240 + 1 = 241\new
3. 2*3*(4*5) + 1 = 2*3*20 + 1 = 120 + 1 = 121\new
4. 2*3*4*(5 + 1) = 2*3*4*6 = 144\new
5. 2*(3*4*5) + 1 = 2*60 + 1 = 121\new
6. (2*3)*4*(5 + 1) = 12*4*6 = 288 + 1?  <progressbar>23%</progressbar> Wait,
no. Wait, if we group 5 + 1 first, but addition has lower precedence. Wait,
unless we insert parentheses around the addition. But in the original
expression, the addition is at the end. So unless we have parentheses around
the entire expression, but that's not allowed.\n
Wait, maybe there are more possibilities. Let me think again.\new
Alternatively, perhaps the expression can be parenthesized in different ways
where different multiplications are grouped. For example:\new
1. Group 2*3 first: ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
2. Group 3*4 first: 2*(3*4)*5 + 1 = 2*24*5 + 1 = 240 + 1 = 241\new
3. Group 4*5 first: 2*3*(4*5) + 1 = 2*3*20 + 1 = 120 + 1 = 121\new
4. Group 2*3*4 first: (2*3*4)*5 + 1 = 24*5 + 1 = 120 + 1 = 121\new
5. Group 2*3*4*5 first: 2*3*4*5 + 1 = 120 + 1 = 121\new
6. Group 2*3 first and 4*5 first: 2*(3*4*5) + 1 = 2*60 + 1 = 120 + 1 = 121\new
7. Group 3*4 first and 5: 2*3*(4*5) + 1 = same as above, 121\new
8. Group 2*3*4 first and 5: 2*(3*4)*5 + 1 = same as above, 241\n
 <progressbar>27%</progressbar> Wait, but that's only 7 distinct values:

 121, 144, 241. But that seems too few. The problem says "how many values can
be obtained", and the example gives 144, which is one of them. Maybe there
are more.\new
Wait, perhaps I need to consider all possible ways of parenthesizing the
multiplication and addition. The original expression is 2*3*4*5 + 1. Let me
think of the possible ways to group the multiplications.\new
The multiplication operations are between 2*3, 3*4, 4*5. The addition is
between the result of 2*3*4*5 and 1. But inserting parentheses can change
the order of operations. For example, if we group 2*3 first, then multiply
by 4, then multiply by 5, then add 1. But that's the same as the original
expression. Alternatively, if we group 3*4 first, then multiply by 5, then
add 1. That would be 2*(3*4)*5 + 1 = 2*24*5 + 1 = 240 + 1 = 241.\new
 <progressbar>29%</progressbar> Alternatively, if we group 4*5 first, then

 multiply by 2, then add 1. That's 2*3*(4*5) + 1 = 2*3*20 + 1 = 120 + 1 =
 121. So that's the same as the original.\n
Alternatively, if we group 2*3*4 first, then multiply by 5, then add 1.
That's 2*3*4*5 + 1 = 120 + 1 = 121.\n
Alternatively, if we group 2*3 first, then multiply by 4, then add 5, but
that's the same as the original.\n
Wait, perhaps the key is that the addition is after all multiplications. But
if we can group different multiplications together, we can change the order.
Let me think of the expression as a sequence of operations. The original is
2 * 3 * 4 * 5 + 1. If we insert parentheses to group different
multiplications, we can create different intermediate results.\new
 <progressbar>31%</progressbar> Wait, perhaps another way: the expression
can be written as 2 * 3 * 4 * 5 + 1. If we insert parentheses to group the
first two multiplications, then multiply by 4 and 5, then add 1. But that's
the same as the original. Alternatively, if we group the last two
multiplications first, then multiply by the first two, then add 1. That
would be ((2 * 3) * (4 * 5)) + 1 = 12 * 20 + 1 = 240 + 1 = 241.\new
Alternatively, if we group the first multiplication and the last two
multiplications. Wait, but how?\new
Wait, perhaps the possible parenthesizations are:\new
1. ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
2. 2*(3*4)*5 + 1 = 240 + 1 = 241\new
3. 2*3*(4*5) + 1 = 120 + 1 = 121\new
4. 2*3*4*(5 + 1) = 240 + 1 = 241\new
 <progressbar>33%</progressbar> 5. 2*(3*4*5) + 1 = 2*60 + 1 = 120 + 1 = 121\new
6. (2*3)*4*5 + 1 = 12*4*5 + 1 = 240 + 1 = 241\n
Wait, but these are all the same values: 121 and 241. That can't be right.
The problem must have more possible values. Maybe I'm missing some.\n
```

```
Wait, perhaps there’s another way to group. For example, 2*3*4*5 + 1 can be
grouped as 2*(3*(4*5 + 1)) or something. Let’s try that.\n
If we group 4*5 first, then add 1, then multiply by 3, then multiply by 2.
That would be 2*3*(4*5 + 1) = 2*3*(20 + 1) = 2*3*21 = 63 + 1 = 64?
<progressbar>35%</progressbar> Wait, no. Wait, the expression is 2*3*4*5 +
1. If we group 4*5 first, then add 1, then multiply by 3 and 2. Wait, but
the original expression is written as 2*3*4*5 + 1, so the addition is at the
end. If we insert parentheses around 4*5 + 1, then multiply by 3 and 2. But
that would be 2*3*(4*5 + 1) = 2*3*21 = 126 + 1 = 127? Wait, no. Wait, the
original expression is 2*3*4*5 + 1. If we group 4*5 first, then add 1, then
multiply by 3 and 2. But how does that work? Let’s parse it.\n
If we write 2*3*4*5 + 1 as 2*3*(4*5 + 1). Then compute 4*5 + 1 first, which
is 20 + 1 = 21.  <progressbar>37%</progressbar> Then multiply by 3 and 2:
2*3*21 = 6*21 = 126. Then add 1? Wait, no. Wait, the original expression is
2*3*4*5 + 1. If we group 4*5 + 1 first, then multiply by 3 and 2, but the
original expression is 2*3*4*5 + 1. So inserting parentheses around 4*5 + 1
would require changing the original structure. Let me see:\new
Original: 2*3*4*5 + 1\new
If we want to group 4*5 + 1 first, we need to write it as 2*3*(4*5 + 1). But
in the original expression, the addition is separate. So inserting
parentheses around 4*5 + 1 would require modifying the original expression.
However, the original expression is written as 2*3*4*5 + 1. So inserting
parentheses around 4*5 + 1 would mean changing it to 2*3*(4*5 + 1), which
would be different.  <progressbar>40%</progressbar> But in that case, the
value would be 2*3*21 = 6*21 = 126, then add 1? Wait, no. Wait, the original
expression is 2*3*4*5 + 1. If we insert parentheses around 4*5 + 1, the
expression becomes 2*3*(4*5 + 1). Then, according to order of operations,
multiplication is done before addition, so first compute 4*5 + 1 = 21, then
multiply by 3 and 2: 2*3*21 = 6*21 = 126. But in the original expression,
the +1 is outside the parentheses. So if we group 4*5 + 1 first, the value
would be 126, but the original expression is 2*3*4*5 + 1. So the difference
is whether the +1 is after the entire multiplication or not. If we insert
parentheses around 4*5 + 1, then the expression becomes 2*3*(4*5 + 1).
<progressbar>42%</progressbar> If we don’t insert parentheses, it’s 2*3*4*5
+ 1. Therefore, inserting parentheses around 4*5 + 1 changes the value. So
that’s a new value. So that’s 126. Then, are there other groupings?\n
Wait, let’s check another grouping. Suppose we group 2*3 first, then
multiply by 4 and 5, then add 1. But that’s the original expression.
Alternatively, group 2*3*4 first, then multiply by 5, then add 1. That’s
24*5 + 1 = 120 + 1 = 121. So that’s the same as original. Alternatively,
group 2*3*4*5 first, then add 1. That’s 120 + 1 = 121. So that’s the same.\n
Alternatively, group 3*4 first, then multiply by 5, then add 1. That’s
2*24*5 + 1 = 240 + 1 = 241.  <progressbar>44%</progressbar> So that’s
another value.\new
Wait, but if we group 4*5 first, then multiply by 2 and 3, then add 1. That
would be 2*3*(4*5) + 1 = 12*20 + 1 = 240 + 1 = 241. So same as before.\n
Alternatively, group 2*3*4 first, then multiply by 5, then add 1. That’s
24*5 + 1 = 120 + 1 = 121.\n
Alternatively, group 2*3 first, then multiply by 4 and 5, then add 1. That’s
the original expression. So same result.\n
Alternatively, group 2*3*4*5 first, then add 1. That’s 120 + 1 = 121.\n
Wait, so so far, we have two values: 121 and 241. But the problem says "how
many values can be obtained", and the example gives 144.
<progressbar>46%</progressbar> So there must be more.\new
Wait, perhaps I need to consider different groupings where different
multiplications are combined in different orders. Let me try another
approach.\n
The original expression is 2*3*4*5 + 1. Let’s consider all possible ways to
group the multiplications. Each multiplication can be grouped or not.
However, the addition is fixed. So the possible groupings would involve
inserting parentheses around different parts of the expression.\new
Possible groupings:\new
1. ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
2. 2*(3*4)*5 + 1 = 240 + 1 = 241\new
3. 2*3*(4*5) + 1 = 120 + 1 = 121\new
4. 2*3*4*(5 + 1) = 240 + 1 = 241\new
5. 2*(3*4*5) + 1 = 2*60 + 1 = 120 + 1 = 121\new
6.  <progressbar>49%</progressbar> Wait, but if we group 5 + 1 first, then
multiply by 2 and 3 and 4. But in the original expression, the addition is
```

separate. So inserting parentheses around 5 + 1 would change the order. For example, if we write 2*3*4*(5 + 1), that’s different from the original. So that gives 2*3*4*6 = 144. So that’s a new value.\n
Similarly, if we group 4*5 first, then multiply by 2 and 3, then add 1. That would be 2*3*(4*5) + 1 = 2*3*20 + 1 = 120 + 1 = 121. So same as original.\n
Alternatively, group 3*4 first, then multiply by 5, then add 1. That’s 2*(3*4)*5 + 1 = 24*5 + 1 = 120 + 1 = 121. So same as original.\new
Alternatively, group 2*3 first, then multiply by 4*5, then add 1. <progressbar>51%</progressbar> That’s 2*(3*4*5) + 1 = 2*60 + 1 = 120 + 1 = 121.\new
Wait, so the only new value from grouping different multiplications is 144. But the problem statement says "the expression 2*3*4*5 + 1 can be obtained by inserting parentheses in 144 different ways". Wait, but according to my current analysis, only 144 and 121. But the problem states 144 different values. So there must be more groupings.\new
Wait, perhaps I missed some groupings. Let me think again.\new
Alternatively, maybe inserting parentheses around different segments. For example:\new
1. Group 2*3 first: ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
2. Group 3*4 first: 2*(3*4)*5 + 1 = 240 + 1 = 241\new
3. <progressbar>53%</progressbar> Group 4*5 first: 2*3*(4*5 + 1) = 2*3*21 = 6*21 = 126\new
4. Group 2*3*4 first: 2*(3*4)*5 + 1 = 240 + 1 = 241\new
5. Group 2*3*4*5 first: 2*3*4*5 + 1 = 120 + 1 = 121\new
6. Group 2*3*4*5 + 1 as is: 121\new
But in this case, grouping 4*5 first gives 126, which is a new value. So that’s one more value. Then, perhaps other groupings. For example:\n
Wait, another way: group 2*(3*4)*5 + 1. Wait, but that’s the same as grouping 3*4 first. Alternatively, group 3*(4*5) first: 2*3*(4*5 + 1) = 2*3*21 = 126. <progressbar>56%</progressbar> So that’s the same as before.\n
Alternatively, group 2*(3*4*5) + 1 = 2*60 + 1 = 121. So that’s the same as original.\n
Alternatively, group 2*3*(4*5 + 1) = 2*3*21 = 126. So that’s another value.\n
Wait, perhaps there are more possibilities. Let’s see:\n
Wait, if we consider inserting parentheses around different segments. For example, 2*(3*4)*5 + 1. That’s 2*24*5 + 1 = 120 + 1 = 121. So same as original.\n
Alternatively, 2*3*(4*5 + 1) = 2*3*21 = 126. So that’s another value.\n
Similarly, 2*3*4*(5 + 1) = 2*3*4*6 = 144. So that’s another value.\n
So now we have 121, 241, 126, 144. <progressbar>59%</progressbar> Wait, but the problem says 144 different values. Maybe there are more.\new
Wait, perhaps I need to consider all possible groupings where different multiplications are combined in different orders. Let’s try to list all possible parenthesizations.\new
Possible groupings:\new
1. ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
2. 2*(3*4)*5 + 1 = 240 + 1 = 241\new
3. 2*3*(4*5 + 1) = 12*21 = 252? Wait, no. Wait, 4*5 = 20, then 20 + 1 = 21. Then 2*3 = 6. 6*21 = 126. So 126. So that’s a new value.\n
4. 2*3*4*(5 + 1) = 2*3*4*6 = 144. So that’s another value.\n
<progressbar>61%</progressbar> 5. 2*(3*4*5) + 1 = 2*60 + 1 = 121. Same as original.\new
6. 2*3*(4*5 + 1) = 126. Same as before.\new
Wait, perhaps there are more. Let me think again.\new
Wait, another possible grouping: 2*3*(4*5 + 1) = 2*3*21 = 126. So 126.
Similarly, 2*(3*4*5) + 1 = 120 + 1 = 121. So that’s the same as original.\n
Alternatively, if we group 2*3*4*5 + 1, but insert parentheses differently. For example, 2*(3*4*5 + 1) = 2*60 + 1 = 121. So same as original.\new
Alternatively, 2*(3*4*5 + 1) = 120 + 1 = 121.
<progressbar>64%</progressbar> So same.\new
Wait, maybe the key is that inserting parentheses around different segments can lead to different intermediate results. Let me try to list all possible groupings:\new
1. ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
2. 2*(3*4)*5 + 1 = 240 + 1 = 241\new
3. 2*3*(4*5 + 1) = 2*3*21 = 126\new
4. 2*3*4*(5 + 1) = 2*3*4*6 = 144\new
5. 2*3*4*5 + 1 = 120 + 1 = 121\new
Wait, so from these, the distinct values are 121, 241, 126, 144. But the

732

problem states that there are 144 different values. So perhaps there’s a misunderstanding here. Wait, maybe the problem is considering all possible parenthesizations, including those that result in the same value through different groupings. <progressbar>67%</progressbar> But in the case of 144, that’s 144 distinct values. But according to my analysis, I only found four values: 121, 241, 126, 144. So maybe the problem is different.\new
Wait, perhaps the original problem allows inserting parentheses in such a way that the order of operations is changed more flexibly. Let me think again.\new
Wait, the problem says "Inserting parentheses can be done in 144 different ways." So perhaps the answer is 144. But according to my analysis, I found only four different values. Therefore, there must be a mistake in my reasoning.\new
Alternatively, maybe I need to consider that when you have multiple multiplications and additions, inserting parentheses can lead to more combinations. Let’s think recursively. For example, the original expression is 2*3*4*5 + 1. The addition is outside the multiplications. So if we consider that inserting parentheses can be done at different points. Let’s think of each multiplication as a node and the addition as a child. <progressbar>70%</progressbar> So the expression is a tree with two multiplications (2*3, 3*4, 4*5) and one addition. The number of parenthesizations would be the number of ways to parenthesize the multiplications, considering that the addition is a separate node.\new
Wait, perhaps the number of ways to parenthesize the multiplications is the number of ways to parenthesize the three multiplications, which is 2^(n-1) where n is the number of multiplications. Here, there are three multiplications. So 2^(3-1) = 4. So 4 ways to parenthesize the multiplications. Then, for each of those 4 ways, we can insert the addition in different positions. But the addition is fixed as 2*3*4*5 + 1, so inserting parentheses around the addition would require changing the order of the multiplications. Wait, perhaps the addition is considered as a separate node and the multiplications are grouped in different ways.\new
Alternatively, perhaps the problem is similar to the number of different ways to parenthesize an expression with different operations. For example, in the expression 2*3*4*5 + 1, the number of ways to parenthesize the multiplications is 2^(number of multiplications -1).
<progressbar>74%</progressbar> Since there are three multiplications, the number is 2^(3-1) = 4. Then, for each of these 4 ways, the addition could be parenthesized in different ways. But since the addition is a single node, inserting parentheses around it would require considering the entire expression as a single addition. Wait, but in the original expression, the addition is separate from the multiplications. So perhaps the number of parenthesizations is the number of ways to parenthesize the multiplications, which is 2^(n-1) where n is the number of multiplications. Here, n=3, so 2^(3-1)=4. Then, for each of these 4 ways, the addition could be inserted in different places. But since the addition is a single node, inserting parentheses around it would require changing the order of operations, but since the addition is already separate, maybe it’s not counted as a separate parenthesization.\new
Wait, perhaps the problem is that the addition is considered as part of the expression, so the entire expression is 2*3*4*5 + 1. Therefore, the number of parenthesizations would be the number of ways to parenthesize the three multiplications. <progressbar>77%</progressbar> Since there are three multiplications, the number of parenthesizations is 2^(3-1) = 4. Then, for each of these 4 ways, we can insert the addition in different places. However, the addition is a single term, so inserting parentheses around it would require changing the order of operations, but since the addition is already separate, maybe it’s not counted as a separate parenthesization.\n
Wait, this is getting confusing. Let me check the original problem statement again: "the expression 2*3*4*5 + 1 can be obtained by inserting parentheses in 144 different ways. How many different values can be obtained by inserting parentheses."\new
The problem states that inserting parentheses can be done in 144 different ways. So perhaps the number of ways is 144, and the question is asking for the number of different values. So maybe the answer is 144, but according to the example, the answer is 144. But in my analysis, I found 4 values. Therefore, perhaps my initial analysis was incomplete.\new
Wait, perhaps the problem is that when you insert parentheses, you can insert them in different positions between the numbers and the addition.

```
<progressbar>81%</progressbar> For example, between 2 and 3, between 3 and
4, between 4 and 5, and between 5 and 1. Each of these insertions can be
considered as different parenthesizations.\new
But the addition is between 4 and 5, so if we consider inserting parentheses
around 4*5 + 1. So:\new
1. ((2*3)*4)*5 + 1 = 120 + 1 = 121\new
2. 2*(3*4)*5 + 1 = 240 + 1 = 241\new
3. 2*3*(4*5) + 1 = 24 + 1 = 25\new
4. 2*3*4*(5 + 1) = 2*3*4*6 = 144\new
5. 2*(3*4*5) + 1 = 120 + 1 = 121\new
6. 2*3*(4*5 + 1) = 2*3*21 = 126\new
7. 2*(3*4)*(5 + 1) = 2*12*6 = 144\new
8. 2*3*4*(5 + 1) = 126\new
9. 2*3*4*5 + 1 = 120 + 1 = 121\new
10. 2*(3*4*5) + 1 = 120 + 1 = 121\new
 <progressbar>90%</progressbar> Wait, this is getting too many. Maybe there
are only four unique values: 121, 241, 126, 144. But the problem says 144
different values. Therefore, there must be a different approach.\new
Wait, perhaps the problem is that the addition is considered as part of the
expression, so inserting parentheses around it can be done in different
ways. For example, inserting parentheses around 2*3*4*5 + 1. But the
addition is separate. Wait, maybe the problem is considering the entire
expression as a single expression with two multiplications and one addition.
Then the number of parenthesizations would be 2^(n-1) where n is the number
of multiplications. Here, n=3, so 2^(3-1)=4. Then, for each of these 4 ways,
the addition could be inserted in different positions. However, since the
addition is a single term, inserting parentheses around it would not change
the order of operations. Therefore, the number of different values is 4, but
the problem states 144.  <progressbar>94%</progressbar> Therefore, I must be
missing something.\new
Alternatively, perhaps the original problem is not considering the addition
as a separate term but as an operator. Wait, the problem says "the
expression 2*3*4*5 + 1". So the addition is separate. Therefore, the number
of parenthesizations is only for the multiplications. If there are three
multiplications, the number of parenthesizations is 2^(3-1)=4. Then, for
each of these 4 ways, the addition could be inserted in different places.
But since the addition is a single term, inserting parentheses around it
would not change the order of operations. Therefore, perhaps the number of
different values is 4, but the problem says 144. Therefore, I must have made
a mistake.\new
Wait, maybe the problem is not considering the addition as a separate term
but as part of the expression. For example, the expression is 2*3*4*5 + 1.
So inserting parentheses around the entire expression would mean changing
the order of operations. But in standard order of operations, addition is
done after all multiplications.  <progressbar>98%</progressbar> Therefore,
the only way to get a different result is by inserting parentheses around
the addition. However, in this case, the addition is between 4*5 and 1. So
inserting parentheses around 4*5 + 1 would change the order. Therefore, this
would be one way. But the problem says 144 different ways. Therefore, there
must be a misunderstanding in the problem statement.\new
Given that the problem states that the expression can be obtained in 144
different ways, and the answer is 144, but according to my analysis, I found
only 4 values. Therefore, perhaps the problem is different. But since the
user says the answer is 144, and the problem is asking for the number of
different values, I think the answer is 144. Therefore, the answer is 144.\new
Final Answer\new
\\boxed{144} \new
</think>\n'
```

734

D Use of large language models

This research was conducted with assistance from large language models across three primary areas: implementation, ideation, and research. For implementation, we used large language models to help with coding tasks, debugging, and text polishing, including identifying spelling errors and improving clarity. For ideation, the models provided suggestions for potentially interesting experiments and research directions. For research, we employed OpenAI's Deep Research feature to gather information about related papers and work in the field. Additionally, large language models assisted with data visualization and plotting tasks throughout the project.

E Model Size And Budget

Our subject model DEEPSCALER-1.5B has 1.5B parameters. We report *A100-80GB* GPU-hours.

1. **SFT**: 8 A100h.
2. **RL fine-tuning**: 20 A100H (2 X A100 for 10h).
3. **Total (incl. failed runs, sweeps, ablations)**: 200 A100h.