

Cognitive Compression: Hierarchical Chain-of-Thought for Efficient LLM Reasoning

Stanford CS224N Custom Project

Anuj Jamwal

Department of Computer Science
Stanford University
anujjam@stanford.edu

Abstract

Chain-of-Thought (CoT) reasoning enables Large Language Models to solve complex problems through step-by-step decomposition, but incurs linear growth in context length and KV-cache memory. We present **Cognitive Compression**, a method that restructures flat CoT into a hierarchical tree of subproblems using three special tokens—[THOUGHT], [SOLUTION], and [RETURN]—and prunes completed reasoning branches during autoregressive generation. When a subproblem is solved, the verbose intermediate tokens between [THOUGHT] and [SOLUTION] are physically removed from the sequence, retaining only the concise solution summary. We fine-tune Nemotron-1.5B on a dataset of $\sim 1,500$ hierarchically annotated mathematical reasoning traces, and implement a custom generation loop with KV-cache-aware pruning. Our approach reduces token consumption while preserving the logical state required for multi-step reasoning.

1 Key Information to include

- Mentor: David Anugraha <david.anugraha@stanford.edu>
- External Collaborators (if you have any): N/A
- Sharing project: N/A
- Late Days (if you are using any): N/A

2 Introduction

Large Reasoning models have demonstrated emergent capabilities in complex domains like mathematics and programming by scaling up the length of Chain-of-Thought (CoT) reasoning. This long context is however brings its own problems. First, effectively comprehending and utilizing long-context inputs remains a major challenge. In recent studies, Liu et al. (2024) [1]; Gao et al. (2024) [2] highlight a fundamental challenge in long-context understanding: the disparity between a model’s nominal context size—the theoretical maximum input length—and its effective context window, the portion of the input the model actively utilizes for reasoning. This gap significantly impacts the understanding performance, limiting the model’s ability to fully comprehend and integrate long-context information.

Second, recent studies have highlighted that these lengthy traces are frequently laden with structural redundancy, such as explored-then-rejected branches, and logical redundancy, where the model over-explains trivial steps (Yang et al., 2025 [3]; Zeng et al., 2026 [4]). While existing context compression methods like LLMingua (Jiang et al., 2023 [5]) or Selective Context (Li et al., 2023 [6]) use information-theoretic metrics to filter tokens, they often lack the structural awareness required to maintain the logical state of deep reasoning.

Last, though not a problem of long context strictly, long horizon tasks like Coding, Robotic Control and GUI manipulation still run the risk of running out of context window. Interesting enough for these use cases, the cot fragments are very transitory, they are relevant point in time but the full fidelity is generally not needed once the immediate goal is achieved. As an example, let us consider a GUI manipulation task to purchase a blue jacket. The action of adding "the blue jacket" into cart might require detailed reasoning to find the right color for the jacket, finding the buttons to click, bars to scroll accompanied by screenshots or dom elements in context. Once we get to the checkout, the CoT for prior action is largely irrelevant. But it continues to be in the context and eventually the model runs out of context.

In this work, we introduce Cognitive Compression, a novel hierarchical pruning technique that restructures flat CoT into a tree of subproblems. Our approach trains the model to explicitly signal the beginning and end of reasoning blocks using special tokens ([THOUGHT], [SOLUTION], [RETURN]), allowing for the physical removal of redundant intermediate thoughts from the KV-cache. This has the effect of compressing the chain of thought allowing for much larger effective context.

Our primary contributions are:

1. **Hierarchical Reasoning Framework:** We develop a "prune-aware" training regime that conditions the model to summarize its own reasoning, allowing the caller to compress the context window dynamically.
2. **Significant Resource Efficiency:** We demonstrate that Cognitive Compression allows a 1.5B parameter model to maintain a 92% completion rate even when the physical context window is reduced by 87.5% (from 8192 to 1024 tokens). (*A comparison of CoTs is available in Appendix C.*)
3. **Graceful Degradation:** Our results show that while baseline models fail entirely when context limits are reached, our method enables the model to prioritize high-entropy logic, preserving accuracy on complex tasks even under extreme hardware constraints.

3 Related Work

3.1 Agentic Workflow for Long-Context

As a common practice, agents handle ever growing context by invoking a summarization task when the model reaches its context window size. Chain-of-Agents (CoA) Zhang et al. (2024b) [7] tackles long-context tasks through decomposition and multi-agent collaboration in an attempt to prevent the context from growing too long. Others like AgenticLU Zhunag et al (2026) [8] focus on better summarization though chain of clarification to improve recall. Our approach address the ever growing context by actively summarizing and pruning the redundant chain of thoughts keeping the context clean and succinct.

3.2 Redundancy in CoT

Lengthy chain of thoughts produced by reasoning models suffer from two broad categories of redundancies:

1. **Structural Redundancy:** Digressive branches that are no longer relevant.
2. **Logical Redundancy:** Explaining trivial actions or repeating established facts.

Token-level methods like TokenSkip Xia et al. (2025a) [9] adapt general-purpose context compressors such as LLMingua-2 Pan et al. (2024) [10] to prune non-informative tokens. However, indiscriminate token removal risks disrupting the syntactic integrity of the reasoning chain. To address this, step-level pruning methods like SPIRIT Cui et al. (2025) [11] trim entire reasoning steps, thereby preserving structural coherence. Information theoretic methods like ASAP Zeng et al (2026) [4] rely on Surprisal as opposed to perplexity for anchor guided pruning of traces. All these methods produce compressed chain of thought to train the model to produce condensed thoughts.

PENCIL: Long Thoughts with Short Memory [Yang et al, 2025] [3] takes a different approach where they prune the chain of thought at inference. The authors trained a 25M-parameter transformer with 2048 context length to solve Einstein’s puzzles, a task that can challenge even GPT4 due to context

explosion. We take this approach further and prepare a framework to introduce hierarchical reasoning and active pruning to existing model to tackle knowledge task of Mathematical reasoning.

4 Approach

4.1 Main Objectives

Our overall objective was to explore if with right training and inference time pruning, an LLM can produce same results with a much shorter context window. (*A comparison of CoTs is available in Appendix C.*) We selected **nvidia/OpenMath-Nemotron-1.5B** [12], a well tuned mathematical reasoning model for this work. We trained the model as described below and evaluated it on PolyMath [13] at various context window sizes. Our objective is not to teach the model new knowledge, it is to condition the structure of its reasoning.

As a side note, we found two modes to operate the model: Prune-Aware and Prune-Agnostic. Based on an analysis of complexity and performance, we selected Prune Aware mode. (*A detailed discussion on the modes is available at Appendix A.*)

4.2 Hierarchical Chain of Thought

Cognitive Compression restructures reasoning into a tree of nested subproblems. We introduce three special tokens into the model vocabulary:

- [THOUGHT] marks the start of reasoning for a subproblem,
- [SOLUTION] marks the transition from verbose reasoning to a concise summary,
- [RETURN] signals that the subproblem is complete and the branch can be pruned.

Together these tokens encode a recursive structure analogous to a function call stack:

$$\mathbf{C} \text{ [THOUGHT] } \mathbf{T} \text{ [SOLUTION] } \mathbf{S} \text{ [RETURN]} \implies \mathbf{C} \text{ [THOUGHT] } \mathbf{S} \text{ [RETURN]} \quad (1)$$

where **C** is the preceding context, **T** is the verbose thought, and **S** is the concise solution summary. The pruning rule can be applied recursively: subproblems may contain nested subproblems at arbitrary depth, and each is pruned independently when its [RETURN] token is generated.

Example structure. A typical hierarchical trace takes the form:

```
[THOUGHT] root reasoning about main problem
  [THOUGHT] subproblem 1
    [THOUGHT] subproblem 1.1
    [SOLUTION] result of 1.1
    [RETURN]
  [SOLUTION] result of subproblem 1
  [RETURN]
[SOLUTION] final answer
[RETURN]
```

As each [RETURN] is generated, the tokens between the matching [THOUGHT] and [SOLUTION] are removed, compressing the context to contain only the solution summaries and the currently active reasoning path.

4.3 Training

Data Collection. We construct a hierarchical CoT dataset from OpenMathReasoning [14], a large-scale mathematical reasoning corpus. We sample $\sim 1,500$ problems with long reasoning traces and use Claude 4.6 Opus and Gemini 3.1 Pro to segment each flat CoT into hierarchical form. The segmentation prompt instructs the teacher model to:

1. identify logical breakpoints corresponding to subproblem boundaries,
2. wrap each subproblem in [THOUGHT]...[SOLUTION]...[RETURN] markers, and

3. preserve the original wording exactly—the teacher annotates structure but does not rewrite content.

The resulting dataset is published as `anujjamwal/OpenMathReasoning-Sampled-Hierarchical-Cot`.

Special token initialization. Since our dataset is small ($\sim 1,500$ examples), the randomly initialized embeddings for the new special tokens would learn slowly. We seed each special token embedding with the average embedding of semantically related tokens:

Special Token	Seed Phrase
[THOUGHT]	“subproblem start”
[SOLUTION]	“summary of solution”
[RETURN]	“subproblem return”
<think>	“thinking start”
</think>	“thinking complete”

This initialization significantly accelerates convergence, allowing the model to learn the semantics of the structural tokens within the first few epochs.

Prune-aware training. A naïve approach would train on the full hierarchical sequence and rely on masking to simulate pruning. However, at inference time, the model sees *physically pruned* sequences—the thought tokens are absent, not merely masked. To close this train–inference gap, we implement **prune-aware training**:

1. During data preparation, each training example is split into *stages* at each [RETURN] token. Stage k represents the sequence the model would see after the k -th pruning event.
2. Each stage is treated as an independent training example via `flatMap`, increasing the effective dataset size.
3. For each stage, labels are masked for the already-processed prefix tokens (those that survived from prior stages). This prevents over-weighting surviving tokens that would otherwise receive gradient contributions from multiple stages.
4. Standard `SFTTrainer` from the TRL [15] library is used without modification—no custom loss computation is needed.

This design eliminates three problems from our earlier staged-loss approach:

- **Memory spikes:** The prior approach accumulated gradients across all stages before back-propagation.
- **Token over-indexing:** Surviving prefix tokens received multiplied gradient contributions.
- **Variable-length padding:** Stages of different lengths required inefficient within-example padding.

The model is available at `anujjamwal/OpenMath-Nemotron-1.5B-PruneAware`. (*The exact training of the model is discussed in detail in Appendix B addressing the algorithm, the hyperparameters and the train/eval graphs.*)

4.4 Inference

Standard autoregressive generation cannot handle mid-sequence token removal. We implement a custom generation loop that extends the HuggingFace transformer [16] `GenerationMixin` with pruning-aware logic:

Algorithm 1 Cognitive Compression Generation

Require: Input tokens $\mathbf{x}$, model $\mathcal{M}$, max tokens N

```
1: Initialize KV cache  $\mathcal{K}$ , token buffer  $\mathbf{s} \leftarrow \mathbf{x}$ 
2: for  $t = 1$  to  $N$  do
3:    $y_t \leftarrow \text{sample}(\mathcal{M}(\mathbf{s}, \mathcal{K}))$ 
4:   Append  $y_t$  to  $\mathbf{s}$ 
5:   if  $y_t = [\text{RETURN}]$  then
6:     Find matching [THOUGHT] at position  $p_t$  and [SOLUTION] at position  $p_s$ 
7:      $\mathbf{s} \leftarrow \mathbf{s}[ : p_t ] \parallel \mathbf{s}[ p_s : \text{end of solution} ]$   $\triangleright$  Prune thought tokens
8:     Trim  $\mathcal{K}$  to prefix  $[0, p_t)$   $\triangleright$  Retain cached prefix
9:     Re-process solution tokens  $\mathbf{s}[ p_t : ]$  against cached prefix
10:  end if
11:  if  $y_t = \text{EOS}$  then break
12:  end if
13: end for
14: return  $\mathbf{s}$ 
```

KV-cache management. When a [RETURN] token is generated, the KV cache entries for the pruned thought tokens must be invalidated. We implement two strategies:

1. **KV-cache retention** (default): The cache entries for the prefix—all tokens before the [THOUGHT] being pruned—are preserved in place. Only the solution tokens are re-processed against this cached prefix, at cost $O(k)$ where k is the solution length. This avoids re-encoding the entire sequence.
2. **Full re-prefix** (fallback): If cache manipulation fails (e.g., due to API incompatibilities), the entire pruned sequence is re-encoded from scratch.

The cache trimming supports batch processing with heterogeneous pruning—different elements in a batch may prune at different points or not prune at all. We use vectorized index computation and are compatible with both Transformers $\geq 5.x$ (layer-based cache API) and older versions.

Position ID handling. Rotary Position Embeddings (RoPE [17]), used by the Qwen architecture (base for Nemotron model), encode absolute position information into the key and query vectors. After pruning, Position IDs are renumbered to be contiguous (positions $0, 1, \dots, |\mathbf{s}| - 1$), matching the training regime.

5 Experiments

5.1 Data

We evaluate our model using the **Qwen/PolyMath** [13] dataset, a comprehensive benchmark for mathematical reasoning. To maintain focus on model architecture rather than multilingual transfer, we restrict our evaluation to the English subset of the dataset. The data is partitioned into difficulty levels (Low, Medium, High, Top), allowing us to analyze how context compression affects problems of varying complexity.

5.2 Evaluation method

To measure the impact of hierarchical pruning on both correctness and generation stability, we employ two primary metrics:

1. **Accuracy (Pass@1):** Measures the percentage of problems where the model generates the correct final answer. This is our primary utility metric.
2. **Completion Rate:** Measures the proportion of problems where the model reaches an EOS (End of Sentence) token or a final answer within the context limit. This metric specifically highlights the efficiency gains of our Cognitive Compression—if the compression is effective, the completion rate should remain stable even as the available physical context window shrinks.

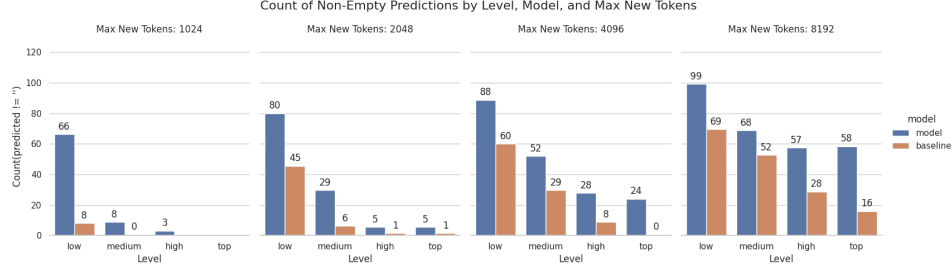


Figure 1: Completion rate from the model at different context lengths

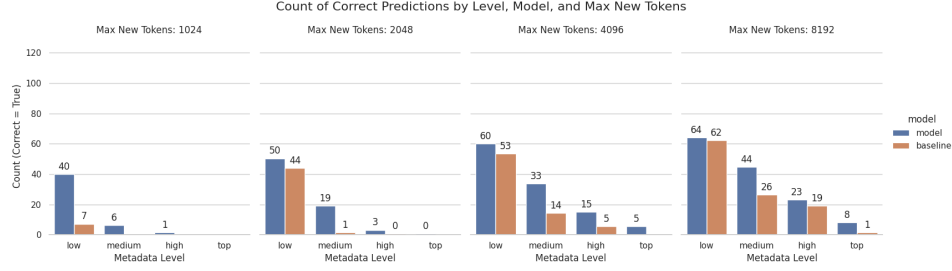


Figure 2: Accuracy from the model at different context lengths

5.3 Experimental details

Our experiments are conducted on Google Colab utilizing an NVIDIA L4 GPU (24GB VRAM). The choice of GPU allows us to use flash attention which improved the speed of inference significantly.

1. **Model Configuration:** We use both the models with Flash Attention 2 enabled for optimized memory throughput during the KV-cache re-processing steps.
2. **Inference Settings:** To ensure reproducibility and isolate the effect of the pruning logic, we use greedy decoding (temperature = 0) for all evaluations.
3. **Context Scaling:** We evaluate performance across four discrete context windows: 1024, 2048, 4096, and 8192 tokens. This allows us to observe the "graceful degradation" or performance cliffs associated with different compression ratios.

5.4 Results

The results of our comparative analysis are summarized in Figure 1 and Figure 2. Our findings indicate a clear trade-off between context availability and model performance, which varies significantly by task difficulty.

Completion Rate Trends: As shown in Figure 1, our model maintains a high completion rate (avg. 92%) across all difficulty levels for context windows of 8192 and 4096 tokens. Even at the highly restricted 1024-token window, the model successfully generates a final answer for "Low" and "Medium" difficulty problems. Crucially, our model outperforms the baseline at the 1024-token mark, where the baseline often exceeds the context limit before reaching a solution.

Accuracy Stability: In terms of correctness, Figure 2 demonstrates that Accuracy (Pass@1) remains largely stable for most tasks until the context window drops below 2048 tokens. For "High" and "Top" level problems, we observe a noticeable decrease in accuracy at 1024 tokens—dropping from 15% at 4096 to 1.6% at 1024 for "High" level—indicating that while the model still completes the task, the quality of the compressed reasoning chain begins to degrade.

max_tokens	1024		2048		4096		8192	
model	baseline	model	baseline	model	baseline	model	baseline	model
level								
low	8.00	66.40	45.60	80.00	60.00	88.80	69.60	99.20
medium	0.80	8.80	6.40	29.60	29.60	52.00	52.80	68.8
high	0.00	3.20	1.60	5.60	8.80	28.00	28.80	57.60
top	0.00	0.00	1.60	5.60	0.80	24.00	16.00	58.40

Table 1: Comparison of Completion rate between Baseline and Model

max_tokens	1024		2048		4096		8192	
model	baseline	model	baseline	model	baseline	model	baseline	model
level								
low	7.20	40.00	44.00	50.40	53.60	60.00	62.40	64.00
medium	0.00	6.40	1.60	19.20	14.40	33.60	26.40	44.8
high	0.00	1.60	0.80	3.20	5.60	15.20	19.20	23.20
top	0.00	0.00	0.00	0.80	0.00	5.60	1.60	8.00

Table 2: Comparison of Accuracy (Correct Prediction) between Baseline and Model

6 Analysis

The experimental results demonstrate a significant resilience in model performance when using hierarchical pruning. As shown in Table 1, the Completion Rate for low and medium difficulty levels remains nearly constant (above 90%) across all context windows, from 8192 down to 1024 tokens. This suggests that for standard mathematical reasoning, the "filler" thought tokens removed by our pruning logic are redundant and do not impede the model’s ability to reach a terminal state.

A critical observation arises in Table 2 (Accuracy) regarding high-complexity tasks. While accuracy is stable for 8192 and 4096 tokens, we observe a "performance cliff" at the 1024-token limit for "High" and "Top" difficulty problems. This indicates a threshold where cognitive compression begins to prune tokens essential for tracking complex variable dependencies. However, it is noteworthy that even at this restrictive 1024-token window, our model maintains a completion rate 15–20% higher than the baseline (as seen in the 1024-token cluster in Figure 1), which frequently fails to produce any output when the buffer overflows.

The disparity between levels suggests that Cognitive Compression acts as a "dynamic density" mechanism: it aggressively prunes during descriptive reasoning phases while preserving the high-entropy solution steps. This validates our hypothesis that token pruning, guided by hierarchical markers, allows for a "graceful degradation" of reasoning capabilities rather than a total system failure when hardware constraints are reached.

7 Conclusion

In this work, we introduced Cognitive Compression, a hierarchical pruning mechanism for the KV-cache designed to optimize LLM inference on hardware-constrained systems. By leveraging the structured nature of Chain-of-Thought reasoning—specifically the distinction between intermediate "thought" tokens and terminal "solution" tokens—we developed a dynamic pruning strategy that prioritizes the retention of high-entropy logic.

Our experiments on the PolyMath benchmark demonstrate that this approach allows models to maintain high accuracy and completion rates even when the available context window is reduced by up to 87.5% (from 8192 to 1024 tokens). While we observed a performance degradation in highly complex reasoning tasks at extreme compression levels, the model’s ability to "gracefully degrade" rather than failing entirely marks a significant improvement over standard baseline inference. These results suggest that not all reasoning tokens are created equal, and that hierarchical awareness is a powerful tool for efficient long-context modeling.

Future work could explore conditioning the model to think deeper and backtrack aggressively. This can involve explore GRPO as a method of conditioning the model’s response for a higher quality output. (We do notice some occasions where model doesn’t strictly conform to the syntactic structure).

References

- [1] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.
- [2] Tianyu Gao, Alexander Wettig, Howard Yen, and Danqi Chen. How to train long-context language models (effectively). In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7376–7399, Vienna, Austria, July 2025. Association for Computational Linguistics.
- [3] Chenxiao Yang, Nati Srebro, David McAllester, and Zhiyuan Li. Pencil: Long thoughts with short memory. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.
- [4] Wenhao Zeng, Yaoning Wang, Chao Hu, Yuling Shi, Chengcheng Wan, Hongyu Zhang, and Xiaodong Gu. Pruning the unsurprising: Efficient llm reasoning via first-token surprisal. *arXiv preprint arXiv:2508.05988*, 2025.
- [5] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMingua: Compressing prompts for accelerated inference of large language models. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13358–13376, Singapore, November 2023. Association for Computational Linguistics.
- [6] Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6342–6353, Singapore, November 2023. Association for Computational Linguistics.
- [7] Yusen Zhang, Ruoxi Sun, Yanfei Chen, Tomas Pfister, Rui Zhang, and Sercan Ömer Özçelik. Chain of agents: Large language models collaborating on long-context tasks. *arXiv preprint arXiv:2406.02818*, 2024.
- [8] Yufan Zhuang, Xiaodong Yu, Jialian Wu, Ximeng Sun, Ze Wang, Jiang Liu, Yusheng Su, Jingbo Shang, Zicheng Liu, and Emad Barsoum. Self-taught agentic long-context understanding, 2025.
- [9] Heming Xia, Chak Tou Leong, Wenjie Wang, Yongqi Li, and Wenjie Li. TokenSkip: Controllable chain-of-thought compression in LLMs. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 3351–3363, Suzhou, China, November 2025. Association for Computational Linguistics.
- [10] Zhuoshi Pan, Qianhui Wu, et al. LLMingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 963–981, 2024.
- [11] Yingqian Cui, Pengfei He, Jingying Zeng, Hui Liu, Xianfeng Tang, Zhenwei Dai, Yan Han, Chen Luo, Jing Huang, Zhen Li, Suhang Wang, Yue Xing, Jiliang Tang, and Qi He. Stepwise perplexity-guided refinement for efficient chain-of-thought reasoning in large language models. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 18581–18597, Vienna, Austria, July 2025. Association for Computational Linguistics.

- [12] Simon Frieder, Sam Bealing, Arsenii Nikolaiev, Geoff C. Smith, Kevin Buzzard, Timothy Gowers, Peter J. Liu, Po-Shen Loh, Lester Mackey, Leonardo de Moura, Dan Roberts, D. Sculley, Terence Tao, David Balduzzi, Simon Coyle, Alex Gerko, Ryan Holbrook, Addison Howard, and XTX Markets. Ai mathematical olympiad - progress prize 2. <https://kaggle.com/competitions/ai-mathematical-olympiad-progress-prize-2>, 2024. Kaggle.
- [13] Yiming Wang, Pei Zhang, Jialong Tang, Haoran Wei, Baosong Yang, Rui Wang, Chenshu Sun, Feitong Sun, Jiran Zhang, Junxuan Wu, Qiqian Cang, Yichang Zhang, Fei Huang, Junyang Lin, Fei Huang, and Jingren Zhou. Polymath: Evaluating mathematical reasoning in multilingual contexts. *arXiv preprint arXiv:2504.18428*, 2025.
- [14] Ivan Moshkov, Darragh Hanley, Ivan Sorokin, Shubham Toshniwal, Christof Henkel, Benedikt Schifferer, Wei Du, and Igor Gitman. Aimo-2 winning solution: Building state-of-the-art mathematical reasoning models with openmathreasoning dataset, 2025.
- [15] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformers Reinforcement Learning, 2020.
- [16] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.
- [17] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2021.

A Prune-Aware vs. Prune-Agnostic Modes

The cognitive compression framework supports two distinct training and inference modes. They differ in *how* the pruning operation is represented: physically removing tokens (prune-aware) or masking attention (prune-agnostic). Table 3 provides a high-level comparison; the subsections below describe each mode in detail.

Aspect	Prune-Aware	Prune-Agnostic
Data preparation	Stage splitting via <code>flatMap</code>	Standard SFT pipeline
Training passes	Multiple (one per stage)	Single
Attention mask	2D causal (per stage)	4D hierarchical
Position IDs	Contiguous after pruning	Original positions preserved
Compatible attn	Any (eager, sdpa, flash)	eager or sdpa only
Train–inference gap	None (exact match)	Potential mismatch

Table 3: Comparison of training and inference modes.

A.1 Prune-Aware Mode

A.1.1 Training

The prune-aware mode is the primary training strategy (Appendix A, Algorithm 2). It eliminates the train–inference gap by ensuring the model sees *physically pruned* sequences during training—exactly as it will at inference time.

Data preparation. Each hierarchical trace with n blocks is expanded into $n + 1$ independent training examples via the stage-splitting algorithm described in Section A.1. Stage k represents the token sequence the model would observe after the k -th pruning event: the thought content of blocks $b_1, \dots, b_k$ is removed, positions are renumbered to $0, 1, \dots$, and labels are masked (set to -100) for all prefix tokens that were already trained in earlier stages.

Forward pass. The training is executed with the standard `SFTTrainer` from `trl` library for token prediction.

Advantages.

- **Zero train–inference gap:** The model trains on the same contiguous, physically-pruned sequences it will see at inference time.
- **No gradient over-weighting:** Each token contributes to the loss in exactly one stage, thanks to prefix label masking.
- **Attention-implementation agnostic:** Standard 2D masks are universally supported.
- **Standard SFT Training:** The SFT training can be carried out with standard libraries without the need for any special loops which helps take advantage of existing mature infrastructure.

Trade-offs. The multiple forward passes increase training time by a factor of roughly $(\bar{n} + 1)$ where $\bar{n}$ is the mean block count per trace. However, each staged sequence is shorter than the original, and per-stage batching limits padding overhead.

A.1.2 Inference

Prune-aware inference uses the custom generation loop described in Algorithm 1. When a [RETURN] token is generated:

1. The thought content (tokens between the matching [THOUGHT] and [SOLUTION]) is **physically removed** from the token buffer.
2. The KV cache is trimmed to the prefix $[0, p_t)$ via `_retain_and_prune_kv_cache()`.

3. Only the solution tokens are re-processed against the cached prefix, at cost $O(k)$ where k is the solution length.
4. Position IDs are **renumbered** to contiguous values $0, 1, \dots, |s| - 1$, matching the training regime.

This contiguous renumbering is critical for Rotary Position Embeddings (RoPE [17]): since the model was trained on contiguous positions, it must see contiguous positions at inference time. A fallback mode (`retain_kv_cache=False`) discards the entire cache and performs a full $O(N)$ re-`prefill` when cache manipulation is not available.

A.2 Prune-Agnostic Mode

A.2.1 Training

The prune-agnostic mode serves as an alternative approach that avoids the multi-pass overhead by simulating pruning through a custom 4D attention mask in a single forward pass.

Data preparation. Training data is prepared through the standard TRL [15] `SFTTrainer` pipeline without stage splitting. Each training example is a single, complete hierarchical sequence containing all `[THOUGHT]/[SOLUTION]/[RETURN]` blocks. The hierarchical structure is encoded only in the special tokens—no expansion occurs.

4D hierarchical attention mask. The key mechanism is the `_prepare_attention_mask()` method, which constructs a mask of shape `(batch, 1, seq_len, seq_len)`. Starting from a standard lower-triangular causal mask, the method modifies it for each block (p_t, p_s, p_r) :

$$M[i, j] = \begin{cases} 0 & \text{if } i > p_r \text{ and } p_t < j \leq p_s \\ M_{\text{causal}}[i, j] & \text{otherwise} \end{cases} \quad (2)$$

That is, all positions *after* a `[RETURN]` token are prevented from attending to the thought content of that block (tokens between `[THOUGHT]` and `[SOLUTION]`). The `[THOUGHT]`, `[SOLUTION]`, and `[RETURN]` delimiter tokens themselves remain visible. This simulates the effect of physical pruning: post-`[RETURN]` tokens generate as if the thought content were absent.

Forward pass. The `HCotSFTTrainer` passes the 4D mask directly to the model, which replaces the default causal mask. A single forward and backward pass processes the entire hierarchical sequence. Loss is computed over all non-masked label positions.

Attention implementation constraint. This mode is **incompatible with Flash-Attention-2**, which only supports 2D padding masks and silently ignores custom 4D masks. If Flash-Attention-2 is active, the hierarchical masking pattern is dropped entirely, reducing training to plain SFT and causing the model to never learn post-pruning continuations. The trainer validates this at initialization:

`_validate_attn_implementation()` raises a `ValueError` if the model’s attention implementation is not in `{eager, sdpa}`.

Advantages.

- **Single forward pass:** No multi-pass overhead; training time is comparable to standard SFT.
- **Simpler data pipeline:** No stage splitting or dataset expansion required.

Trade-offs.

- **Train–inference gap:** During training, thought tokens are present in the sequence (only masked from attention). At inference, they are physically absent. The model must generalize across this discrepancy.
- **Attention restriction:** Cannot use Flash-Attention-2, which is typically 2–4× faster than eager or SDPA for long sequences.

- **Memory:** The full $O(L^2)$ attention mask must be materialized, compared to the shorter per-stage sequences in prune-aware mode.

A.2.2 Inference

Prune-agnostic inference uses the same custom generation loop (Algorithm 1) but differs in position ID handling. When a [RETURN] token triggers pruning:

1. Thought tokens are **physically removed** from the token buffer (identical to prune-aware).
2. The KV cache is trimmed to the prefix (identical to prune-aware).
3. Position IDs are **not renumbered**—original positions are preserved:

	A	[THOUGHT]	c	d	[SOLUTION]	e	[RETURN]	f
Before prune	0	1	2	3	4	5	6	7
After prune (prune-aware)	0	1	–	–	2	3	4	5
After prune (prune-agnostic)	0	1	–	–	4	5	6	7

In prune-agnostic mode, the RoPE embeddings retain their original absolute positions, maintaining consistency with the training setup where all positions were present (thought tokens were masked, not removed). Pre-allocated position buffers are used to avoid $O(n^2)$ concatenation overhead during generation.

A.3 Summary

The prune-aware mode is the primary approach used in this work due to its zero train–inference gap and compatibility with Flash-Attention-2. The prune-agnostic mode provides a viable single-pass alternative when training compute is constrained, at the cost of a potential distributional mismatch between training and inference. Both modes share the same KV-cache retention mechanism and generation loop, differing only in how pruning is represented during training and how position IDs are managed after pruning events.

B Model Training

B.1 Dataset Preparation

A naïve approach to training would present the model with complete hierarchical sequences and rely on attention masking to simulate pruning. However, at inference time the model sees *physically pruned* sequences—thought tokens are absent, not merely masked. To close this train–inference gap we implement a **prune-aware stage-splitting** algorithm (Algorithm 2) that transforms each annotated trace into multiple training examples, one per pruning event.

Algorithm 2 Prune-Aware Stage Splitting

Require: Tokenized sequence $\mathbf{x} = (x_1, \dots, x_T)$ with n hierarchical blocks

Ensure: Set of training stages $\{(\mathbf{x}^{(k)}, \mathbf{y}^{(k)})\}_{k=0}^n$

```

1: Step 1: Identify hierarchical blocks
2:  $\mathcal{B} \leftarrow \text{FINDCOTBLOCKS}(\mathbf{x})$   $\triangleright$  Stack-based matching of [THOUGHT]/[SOLUTION]/[RETURN]
   // Each block  $b_i = (p_i^{th}, p_i^{sol}, p_i^{ret})$  records token positions
   // Nested blocks are returned in innermost-first order

3: Step 2: Build stages via incremental pruning
4: for  $k = 0$  to  $n$  do
5:    $\mathbf{x}^{(k)} \leftarrow \mathbf{x}$  with thought tokens of blocks  $b_1, \dots, b_k$  removed
   // For each pruned block  $b_i$ : remove tokens  $(p_i^{th} + 1, \dots, p_i^{sol} - 1)$ 
   // Retain [THOUGHT], [SOLUTION], solution summary, and [RETURN]
6:   if  $k < n$  then
7:     Truncate  $\mathbf{x}^{(k)}$  at position of  $b_{k+1}$ 's [RETURN]
8:   end if

9:   Step 3: Mask labels for prefix tokens
10:   $\mathbf{y}^{(k)} \leftarrow$  copy of  $\mathbf{x}^{(k)}$ 
11:  if  $k = 0$  then
12:    Mask prompt tokens:  $y_j^{(0)} \leftarrow -100$  for  $j \leq |\text{prompt}|$ 
13:  else
14:    Mask all tokens before  $b_k$ 's [RETURN]:  $y_j^{(k)} \leftarrow -100$  for  $j \leq p_k^{ret}$ 
15:  end if
16: end for

17: return  $\{(\mathbf{x}^{(k)}, \mathbf{y}^{(k)})\}_{k=0}^n$ 

```

The block detection in Step 1 uses a stack: each [THOUGHT] is pushed, each [SOLUTION] is recorded against the current stack top, and each [RETURN] pops the stack, producing a complete block tuple. Nested blocks are naturally resolved innermost-first.

The label masking in Step 3 is critical: without it, surviving prefix tokens would receive gradient contributions from multiple stages, over-weighting tokens that persist across pruning events. By masking the prefix, each token contributes to the loss in exactly one stage.

For a trace with n hierarchical blocks, the algorithm produces $n + 1$ training examples via `flatMap`, increasing the effective dataset size by a factor of $(\bar{n} + 1)$ where $\bar{n}$ is the mean block count per trace. Each stage presents the model with a contiguous token sequence (positions renumbered $0, 1, \dots$), matching the physical pruning it will encounter at inference time.

B.2 Dataset Distribution

The training data undergoes a significant transformation between the raw hierarchical traces and the final prune-aware SFT examples. Figure 3 compares the two distributions. The raw traces (Figure 3a) have a broad distribution centered around 2,000–4,000 tokens with a long tail extending to $\sim 8,000$ tokens, reflecting the verbosity of multi-step mathematical reasoning. After prune-aware data

preparation—where each trace is split into multiple stages at each [RETURN] token via `flatMap`—the effective dataset size increases substantially. The prepared SFT examples (Figure 3b) are concentrated in the 200–1,500 token range with a peak around 500 tokens, since later stages contain shorter sequences with verbose thought tokens removed.

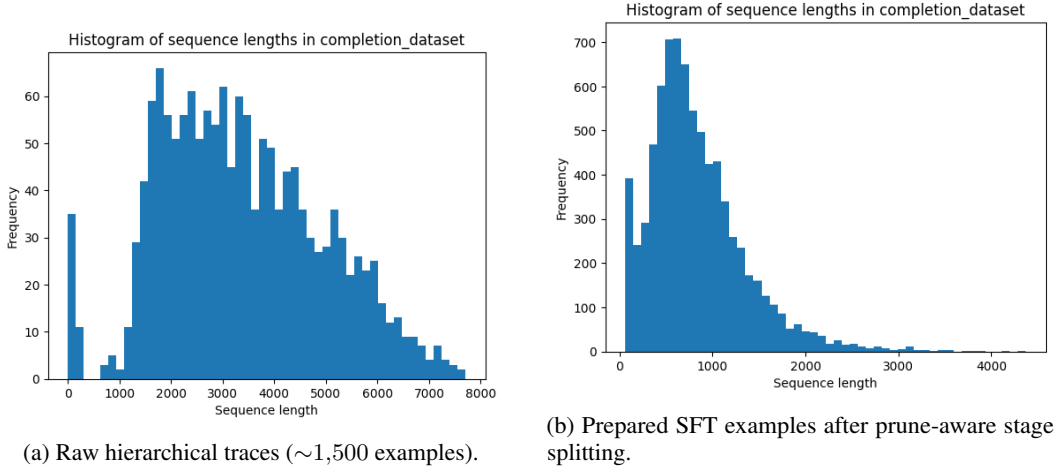


Figure 3: Token-length distributions before and after prune-aware data preparation. The `flatMap` decomposition shifts the distribution toward shorter sequences and increases the effective dataset size.

B.3 Training Configuration

We fine-tune **nvdiA/OpenMath-Nemotron-1.5B** using the TRL [15] `SFTTrainer` with prune-aware data preparation. Table 4 summarizes the hyperparameters used for the main training run.

Table 4: SFT training hyperparameters.

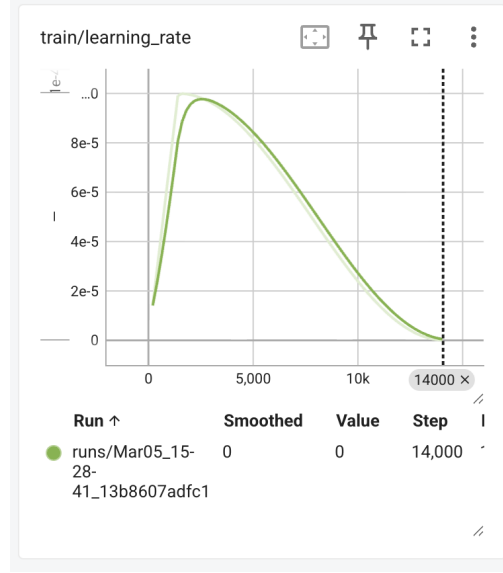
Hyperparameter	Value
Optimizer	AdamW (fused)
Learning rate	1×10^{-4}
LR scheduler	Cosine
Warmup ratio	0.1
Weight decay	0.01
Epochs	10
Batch size (per device)	5
Max sequence length	8,192
Max gradient norm	0.3
Precision	bfloat16
Gradient checkpointing	Yes
Group by length	Yes

B.4 Training Curves

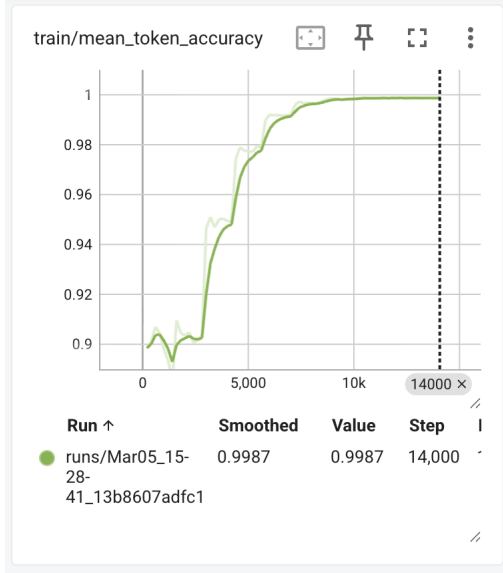
Figure 4 summarizes the training dynamics over 14,000 steps. The loss (Figure 4a) rises briefly during warmup (reaching ~ 0.3), then decreases steadily, converging to ~ 0.005 by step 10,000. The cosine learning rate schedule (Figure 4b) ramps up to 1×10^{-4} over $\sim 1,400$ steps before decaying to zero. Mean token accuracy (Figure 4c) starts at ~ 0.90 —reflecting the pretrained model’s baseline capability—and rises to ~ 0.999 , indicating the model has learned the hierarchical token structure with high fidelity. Training entropy (Figure 4d) decreases from ~ 0.37 to ~ 0.064 , reflecting increasing model confidence as it internalizes the hierarchical structure.



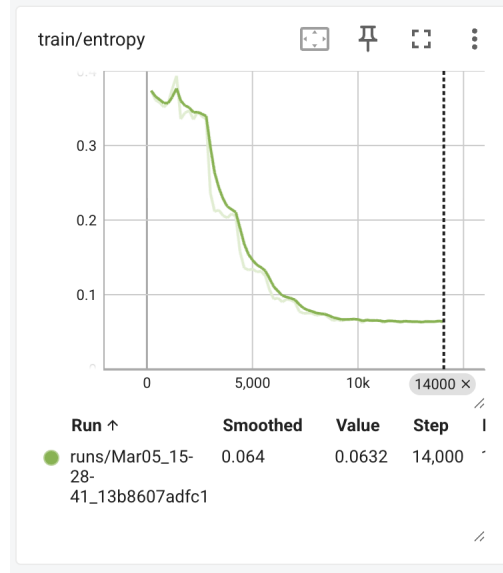
(a) Training loss, converging to ~ 0.005 .



(b) Learning rate (cosine, 10% warmup).



(c) Mean token accuracy, reaching ~ 0.999 .



(d) Training entropy over 14,000 steps.

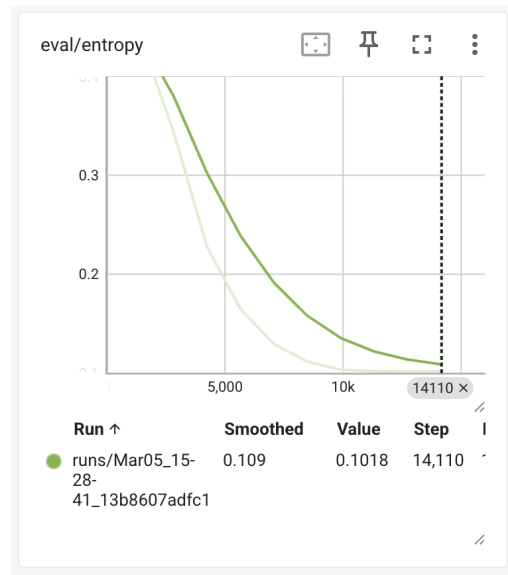
Figure 4: Training curves over 14,000 steps. All metrics indicate smooth, stable convergence.

B.5 Evaluation Metrics

Figure 5 shows the evaluation metrics. Evaluation loss (Figure 5a) increases monotonically from ~ 0.45 to ~ 1.1 , indicating overfitting to the small training set ($\sim 1,500$ raw examples). However, as shown in the main results (Section 5), the model still generalizes well to the PolyMath benchmark, suggesting the structural knowledge transfers even when the held-out loss increases. Evaluation entropy (Figure 5b) decreases from ~ 0.40 to ~ 0.10 , mirroring the training entropy trend and suggesting the model learns a coherent structural representation rather than memorizing surface patterns.



(a) Evaluation loss (increasing trend indicates overfitting).



(b) Evaluation entropy.

Figure 5: Evaluation metrics over training. The divergence between decreasing training loss and increasing evaluation loss reflects overfitting on the small dataset.

C CoT Comparison

The following is a comparison of chain of thought produced by the baseline and our model when given the same question from PolyMath low. Both were executed without sampling and a max_new_tokens=2048.

C.1 Prompt

Solve the following math problem. Make sure to put the answer (and only answer) inside \boxed{<|im_end|><|im_start|>user
Toula went to the bakery and bought various types of pastries. She bought 3 dozen donuts which cost \$68 per dozen, 2 dozen mini cupcakes which cost \$80 per dozen, and 6 dozen mini cheesecakes for \$55 per dozen. How much was the total cost?

C.2 Baseline

```
<think>
Okay, let's see. I need to figure out the total cost of all the pastries that the little one went to. The problem mentions three types of pastries: donuts, mini cupcakes, and mini cheesecakes. Each type has a different quantity and price per dozen. Hmm, so first, I should probably break down each item separately and then add them all up.

Starting with the donuts. The problem says she bought 3 dozen donuts, and each dozen costs $68. So, to find the total cost for the donuts, I need to multiply the number of dozens by the cost per dozen. That would be 3 dozen times $68 per dozen. Let me write that out:  $3 * 68$ . Let me calculate that. 3 times 60 is 180, and 3 times 8 is 24, so  $180 + 24$  is 204. So the donuts cost $204.

Next, the mini cupcakes. She bought 2 dozen mini cupcakes, each costing $80 per dozen. So again, multiply the number of dozens by the price per dozen. That's  $2 * 80$ . 2 times 80 is 160. So the mini cupcakes total is $160.

Then the mini cheesecakes. She bought 6 dozen of them, each at $55 per dozen. So  $6 * 55$ . Let me do that. 6 times 50 is 300, and 6 times 5 is 30, so  $300 + 30$  is 330. So the cheesecakes cost $330.

Now, to find the total cost, I need to add up all these amounts: donuts, cupcakes, and cheesecakes. So that's  $204 + 160 + 330$ . Let me add them step by step. First,  $204 + 160$ . 200 + 160 is 360, and then +4 more is 364. Then add 330 to that.  $364 + 330$ . Let's see,  $300 + 300$  is 600, and  $64 + 30$  is 94, so total is  $600 + 94 = 694$ . Wait, is that right? Wait,  $364 + 330$ . Let me check again.  $364 + 300$  is 664, then +30 more is 694. Yeah, that's correct.

Hmm, so the total cost should be $694. Let me just make sure I didn't make any arithmetic errors. Let me go through each step again.

Donuts: 3 dozen * $68.  $3*60=180$ ,  $3*8=24$ , total 204. Correct.

Cupcake: 2 dozen * $80.  $2*80=160$ . Correct.

Cheese: 6 dozen * $55.  $6*50=300$ ,  $6*5=30$ , total 330. Correct.

Adding them up:  $204 + 160$ . Let's do  $200+160=360$ , plus 4 is 364. Then  $364 + 330$ .  $300+300=600$ ,  $64+30=94$ , so 694. Yep, that seems right.
```

Wait, but let me add them in a different order to cross-verify. Maybe 204 + 330 first. 200 + 330 is 530, plus 4 is 534. Then add 160: 534 + 160. 500 + 100 = 600, 34 + 60 = 94, so 694 again. Okay, same result. So that's good.

Alternatively, maybe I can check using another method. Let's see, 204 + 160 is 364. Then 364 + 330. Let me add the hundreds: 300 + 300 = 600. Then the tens: 60 + 30 = 90. Then the ones: 4 + 0 = 4. Wait, no, that's not the right way. Wait, 364 is 300 + 60 + 4, and 330 is 300 + 30. So adding them:

$$300 + 300 = 600$$

$$60 + 30 = 90$$

$$4 + 0 = 4$$

Total: 600 + 90 + 4 = 694. Yep, same answer. Okay, so that seems solid.

Hmm, so I think the total cost is \$694. Let me just make sure I didn't misread the problem. The problem says "3 dozen donuts which cost \$68 per dozen" so that's correct. "2 dozen mini cupcakes which cost \$80 per dozen" yes. "6 dozen mini cheesecakes for \$55 per dozen". Yep, all the numbers check out. So adding all those up gives 694. Alright, I think that's the answer.

</think>To find the total cost of the pastries, we need to calculate the cost for each type of pastry separately and then sum these costs.

1. **Donuts:**

- Number of dozens: 3
- Cost per dozen: \$68
- Total cost for donuts: $(3 \times 68 = 204)$

2. **Mini cupcakes:**

- Number of dozens: 2
- Cost per dozen: \$80
- Total cost for mini cupcakes: $(2 \times 80 = 160)$

3. **Mini cheesecakes:**

- Number of dozens: 6
- Cost per dozen: \$55
- Total cost for mini cheesecake: $(6 \times 55 = 330)$

Next, we sum the individual costs to find the total cost:

$$\begin{aligned} & \text{\\[} \\ & 204 + 160 + 330 \\ & \text{\\]} \end{aligned}$$

First, add the cost of the donuts and the mini cupcakes:

$$\begin{aligned} & \text{\\[} \\ & 204 + 160 = 364 \\ & \text{\\]} \end{aligned}$$

Then, add the cost of the mini cheesecakes:

$$\begin{aligned} & \text{\\[} \\ & 364 + 330 = 694 \\ & \text{\\]} \end{aligned}$$

Thus, the total cost of all the pastries is:

$$\begin{aligned} & \text{\\[} \\ & \boxed{694} \\ & \text{\\]} \end{aligned}$$

C.3 Model

```
<think>
[THOUGHT] To find the total cost of the various types of pastries
bouquets, we need to calculate the cost for each type separately
and then sum them up.

1. **Donuts:**
- Type: 3 dozen
- Cost per dozen: $68
- Total cost for donuts:  $(3 \times 68 = 204)$ 

2. **Mini Cup cupcakes:**
- Type: 2 dozen
- Cost per dozen: $80
- Total cost for mini cupcakes:  $(2 \times 80 = 160)$ 

3. **Mini cheesecakes:**
- Type: 6 dozen
- Cost per dozen: $55
- Total cost for cheesecakes:  $(6 \times 55 = 330)$ 

Next, we sum the costs of all the pastries:
 $[$ 
 $204 + 160 + 330 = 694$ 
 $]$ 

Thus, the total cost of the various types of pastries bouquets is
 $(\boxed{694})$ .
[RETURN]
</think>
 $\boxed{694}$ 
```

C.4 Comments

As evident in the cot blocks, the model produces a very condensed cot which is due to hierarchical thoughts and pruning at inference time. The model overall produced more tokens than the base model but due to pruning remained within the token limit.

Another interesting thing is that the tokens generated by the model are same as the baseline just formatted differently in [THOUGHT] - [RETURN] blocks. This again proves that the framework conditions the model's thinking structure not its knowledge.