ATTEND TO YOUR OWN THOUGHTS: CALIBRATION MATTERS FOR 1.58-BIT REASONING LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

In this paper, we present ScaleQ-1.58, a ternary post-training quantization (PTQ) framework for reasoning LLMs. Its core insight stems from an empirical finding: although modern LLMs are typically trained to have chain-of-thought reasoning capabilities, in the PTQ regime, even recently proposed differentiable ternarization still leads to performance collapse on challenging mathematics and coding tasks. We reveal that this is caused by conventional calibration schemes that ignore the model's reasoning process. Motivated by this, we present Attend to Your Own Thoughts (AYOT), a new and simple calibration method, which constitutes the core contribution of this paper. Given a pre-trained target LLM, AYOT uses its self-generated reasoning traces and final answers on a proper set of calibration samples, together with the corresponding questions, as context inputs during ternarization. ScaleQ-1.58 is built by combining AYOT with differentiable ternarization, and demonstrates several favorable scaling properties. With only 4M calibration tokens: (1) Qwen3-1.7B ternarized by ScaleQ-1.58 reaches over 90.52% of the performance of the prior best BitNet b1.58 2B4T averaged over 4 mathematics and coding tasks, and our ternary Qwen3-4B shows an absolute gain of 8.97%, while requiring 1,000,000$\times$ fewer calibration tokens for ternarization; (2) ScaleQ-1.58 generalizes well to 8 dense and MoE architectures, with performance significantly improving as model scale increases from 1.7B to 235B parameters; (3) ScaleQ-1.58 exhibits strong generalization across 13 tasks of varying difficulty levels, including challenging mathematics, coding and scientific logic reasoning, as well as basic commonsense reasoning and language generation. Intriguingly, ScaleQ-1.58 shows continuously improving performance as the number of calibration tokens increases. Besides, AYOT generalizes well beyond 1.58-bit quantization to other bit-width settings. With the default 4M calibration tokens, ScaleQ-1.58 requires only 4 to 240 hours on 8 A100-80GB GPUs to ternarize 1.7B to 235B reasoning LLMs.

1 INTRODUCTION

In recent years, the reasoning capability of large language models (LLMs) on challenging tasks such as mathematics and coding has been greatly improved by the advent of chain-of-thought (CoT) prompting (Wei et al., 2022; Jaech et al., 2024; Guo et al., 2025). Moreover, CoT enhances the benefits of scaling up model size and exploiting advanced model architectures, resulting in improved performance across a wide range of tasks (Achiam et al., 2023; Yang et al., 2025; Singh et al., 2025; Comanici et al., 2025; Hurst et al., 2024; Muennighoff et al., 2025; Team et al., 2025; Team, 2026; DeepSeek-AI, 2026). Consequently, the evolution of LLMs has accelerated rapidly. However, deploying LLMs in real-world applications, particularly on resource-constrained platforms, remains constrained due to their high memory and computational costs. Post-training quantization (PTQ) offers a practically appealing solution to this problem, as it is low-cost and easy to implement. Despite significant advances in PTQ research for LLMs, most existing methods (Shen et al., 2020; Dettmers et al., 2022; Yao et al., 2022; Frantar et al., 2023; Xiao et al., 2023; Shao et al., 2024; Lin et al., 2024b; Chee et al., 2023; Ashkboos et al., 2024; Liu et al., 2025b; Sun et al., 2025; Lin et al., 2024a; Hu et al., 2025; Wang et al., 2026a) focus on conventional low-bit settings (e.g., 8-bit/4-bit quantization) and primarily evaluate their performance on basic language generation and commonsense reasoning tasks, largely overlooking complex reasoning tasks such as mathematics and coding that are more critical in practice. A recent work (Liu et al., 2025a) benchmarks the impact of quantization on reasoning

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

LLMs, revealing that existing PTQ methods generally incur serious performance degradation on mathematics and coding tasks when reducing model precision to 3-bit.

In this paper, we study post-training quantization for reasoning LLMs through the lens of ternary (i.e., 1.58-bit) quantization from an unexplored perspective of scaling behavior. The concept of ternary quantization was originally proposed in TWN (Li et al., 2016) to train compact convolutional neural networks (ConvNets) for computer vision tasks. It approximates the high-precision weights $\mathbf{W}$ for a linear layer as $\mathbf{W} \approx \alpha\mathbf{T}$, where α is a scaling factor and $\mathbf{T}$ denotes the corresponding ternary weights. The elements of $\mathbf{T}$ are obtained by a hard ternarization function based on a weight threshold Δ:

$$T_i = \begin{cases} 1, & if\ W_i > \Delta \\ 0, & if\ |W_i| \leq \Delta \\ -1, & if\ W_i < -\Delta. \end{cases} \tag{1}$$

By constraining model weights to $\{1, 0, -1\}$, most of floating-point multiplication operations can be replaced by lower-cost integer additions and subtractions, thereby substantially reducing memory footprint and accelerating inference. Because of this hardware-friendly advantage, increasing research efforts have been devoted to developing 1.58-bit LLMs for natural language processing tasks . TernaryBERT (Zhang et al., 2020) makes the first attempt to train 1.58-bit small BERT models (up to 418M parameters) (Devlin et al., 2019) via fine-tuning for simple language understanding tasks. Due to its extremely low-bit nature, ternary quantization is highly susceptible to severe performance degradation. TernaryBERT uses knowledge distillation (KD) (Jiao et al., 2020; Hinton et al., 2015) To mitigate this issue. Some subsequent studies (Wu et al., 2025; Chen et al., 2024) extend this KD-based ternarization strategy with fine-tuning to relatively larger models (up to 8B parameters), but still focus on simple tasks. Unlike them, BitNet b1.58 series (Ma et al., 2024; Wang et al., 2025), a highly influential open-source family of ternary LLMs whose largest model has 7B parameters, are trained from scratch on a corpus of 100B tokens. Following BitNet b1.58 series, subsequent 1.58-bit LLM families, such as TriLM with its largest 3.9B model trained on 300B tokens (Kaushal et al., 2025) and Tequila with its largest 3B model trained on 10B tokens (Huang et al., 2025), are also built based on quantization-aware training (QAT). Although QAT can largely alleviate performance degradation by simulating weight ternarization during full-precision training, it requires substantial training resources, such as a vast amount of training tokens, large-scale GPU clusters, long training time and considerable power consumption. The demand for such training resources further grows with increasing model scale and architectural complexity. As a result, these existing QAT-based ternarization methods are limited to specific dense LLM architectures (e.g., Llama-like models (Touvron et al., 2023a;b)) with relatively small model scales (typically below 10B parameters). Furthermore, in performance evaluation, they neglect challenging reasoning tasks, such as mathematics and coding, similar to most conventional PTQ methods for LLMs. BitNet b1.58 2B4T (Ma et al., 2025), a 1.58-bit reasoning LLM comprising 2 billion parameters, which is trained from scratch on an extremely large corpus of 4 trillion tokens, including text, mathematics, code and conversation data. It is the first 1.58-bit LLM capable of handling complex reasoning tasks. Despite a relatively lightweight 2B model scale, its training pipeline is both costly and complex, involving two-stage large-scale pre-training, supervised fine-tuning and direct preference optimization, along with meticulous hyperparameter tuning to prevent training instability. This restricts its scalability to much larger LLMs.

Note that, according to Eq. 1, the strategies for estimating the scaling factor α, the weight threshold Δ, and gradients (since ternarization is non-differentiable) are three primitive components of ternary quantization from an optimization perspective. Following TWN (Li et al., 2016) tailored for ConvNets, all aforementioned ternarization methods for LLMs also adopt statistical strategies to determine α and Δ in a static manner, and employ straight-through estimator (STE) (Hubara et al., 2016) to approximate gradients during ternarization. Instead, by extending learning-based quantization methodologies (Rastegari et al., 2016; Zhu et al., 2017; Esser et al., 2020; Gong et al., 2019; Chen et al., 2024), the authors of (Wang et al., 2026b) recently presented CAT-Q, the first differentiable ternarization method, which learns group-wise scaling factors and weight thresholds for pre-trained LLMs via a soft ternarization function. It achieves competitive performance at comparable model scales to QAT-based methods such as BitNet b1.58 series, TriLM, and Tequila, and demonstrates favorable scaling behavior as model scale increases, extending beyond 100B parameters. However, it still focuses on basic commonsense reasoning rather than challenging reasoning tasks. In short, the lack of a scalable ternary PTQ method for efficiently building diverse and high-performing 1.58-bit reasoning LLMs remains a key gap in the community. *To unlock the potential of ternary quantization*

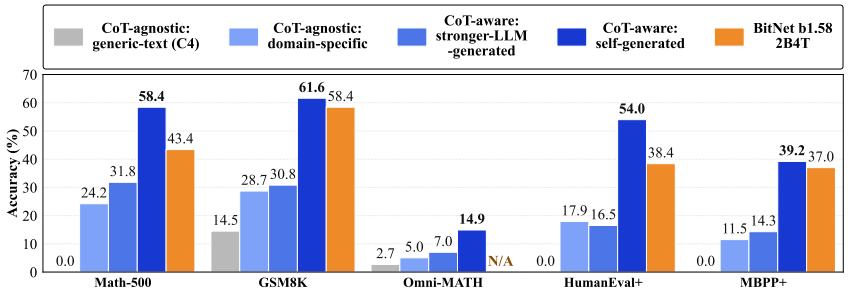

Figure 1: Performance comparison of 1.58-bit Qwen3-4B models obtained by CAT-Q with four different calibration schemes (defined in the Method section) across five challenging mathematics and coding tasks. Using our calibration method AYOT (i.e., CoT-aware: self-generated), with only 4M calibration tokens, the resulting 1.58-bit Qwen3-4B in the PTQ regime yields an absolute gain of 8.97% (averaged over Math-500, GSM8K, HumanEval+ and MBPP+) compared to BitNet b1.58 2B4T (the prior best 1.58-bit reasoning LLM trained from scratch on 4T tokens in the QAT regime), while requiring 1,000,000× fewer calibration tokens for ternarization. More accurate 1.58-bit LLMs spanning 8 different architectures and model scales are shown in Table 1.

for reasoning LLMs and enable their broad applications and accessibility, this paper, based on CAT-Q, takes the first step toward investigating: in the PTQ regime, (1) whether ternary quantization can scale across challenging reasoning tasks, diverse model architectures and larger model scales; (2) if not, what underlying factors underlie this bottleneck? To explore these two questions, we adopt CAT-Q and conduct an empirical study on five mathematics (Math-500, GSM8K and Omni-MATH) and coding (HumanEval+ and MBPP+) tasks. Surprisingly, our empirical study shows that, even for the pre-trained Qwen3-4B (Yang et al., 2025) with a twofold larger model scale, the resulting 1.58-bit model by CAT-Q is significantly worse than BitNet b1.58 2B4T, as shown in Fig. 1.

We reveal that the performance collapse is caused by the calibration scheme. For pre-trained LLMs, existing PTQ methods mostly adopt sequential layer-wise or block-wise reconstruction, which relies on a set of calibration data to estimate the output distribution of each layer or block. Most commonly, calibration data is sampled from web text corpora such as C4 (Raffel et al., 2020) and WikiText2 (Merity et al., 2017). However, calibration in this way has two limitations. First, it assumes that PTQ methods are robust to calibration variations. This tends to be valid under conventional low-bit formats, e.g., 8-bit and 4-bit quantization, where their expressive capacities are reasonably sufficient to approximate the FP16 format, but it becomes significantly less effective in the case of ultra-low-bit quantization such as ternary due to severe information loss, especially for complex reasoning tasks. Second, CoT demonstrations from pre-trained high-precision reasoning LLMs play an essential role in tackling mathematics and coding tasks, which break down a complex problem into a series of intermediate logical reasoning steps leading to a correct answer, whereas web text corpora mainly reflect generic local word-to-word relationships but not explicit logical reasoning patterns.

In light of this analysis, we present Attend to Your Own Thoughts (AYOT), a simple and effective calibration method, which constitutes the core contribution of this paper. AYOT relies on two coupled strategies: appropriate calibration data selection and the use of CoT demonstrations generated by the pre-trained target LLM itself. We apply AYOT to CAT-Q, forming a scalable ternary PTQ framework for reasoning LLMs, which we term ScaleQ-1.58. We conduct extensive experiments, showing the favourable scaling properties of ScaleQ-1.58 as well as the promising generalization ability of AYOT.

2 METHOD

In this section, we begin by briefly defining CAT-Q (Wang et al., 2026b), the basic ternarization module of ScaleQ-1.58, and then introduce AYOT, the core technical contribution of this work.

2.1 PRELIMINARY CONCEPT

Ternary quantization is difficult to optimize owing to its extreme discreteness and non-differentiable nature. In the PTQ regime, only a limited set of calibration samples is available, and neither retraining nor fine-tuning is permitted. This makes PTQ significantly more challenging than QAT, which dominates current research on 1.58-bit LLMs. To address this issue, CAT-Q (Wang et al., 2026b)

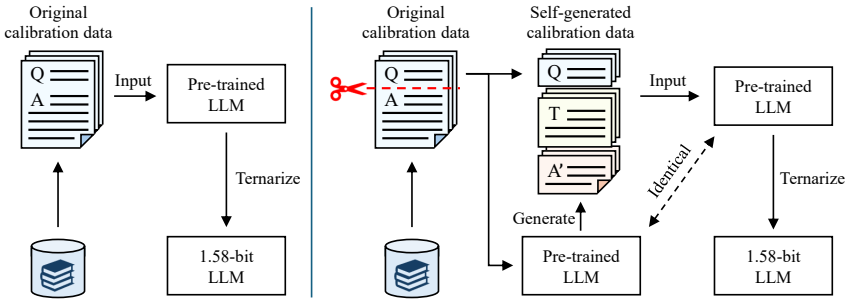

Figure 2: Post-training ternarization of LLMs under CoT-agnostic calibration (left) vs. CoT-aware calibration AYOT (right). For the question Q, A denotes the ground-truth answer, and T and A' denote the reasoning traces and final answer generated by the pre-trained target LLM, respectively.

introduces a soft ternarization function coupled with a learnable weight redistribution strategy:

$$T_i = f(\hat{W}_i; s, t, \Delta) = \frac{\tanh(ts(\hat{W}_i - \Delta)) + \tanh(ts(\hat{W}_i + \Delta))}{2\tanh(ts)}. \tag{2}$$

Here, following the notations in Eq. 1, $\hat{W}_i$ denotes the ith element of the redistributed pre-trained weights $\hat{\mathbf{W}}$ obtained by a linear transformation $\hat{\mathbf{W}} = (\mathbf{W} - \mu)/\alpha$ parameterized by two learnable variables: the weight mean μ and the scaling factor α. In formulation, μ is used only for weight redistribution and is removed during weight reconstruction, leading to improved performance compared to using it in weight reconstitution while retaining the original hardware-friendly property of TWN (Li et al., 2016). s is a constant sharpness factor, and t denotes the current calibration time-step normalized to $[0, 1]$. The denominator $2\tanh(ts)$ controls the shape of the output curve, progressively bounding the output to $[-1, 1]$ during calibration. The weight threshold Δ determines the width of its zero-output region. In principle, with $f(\cdot)$, CAT-Q starts from the identity mapping $f(\cdot) = \hat{\mathbf{W}}$ and asymptotically converges to a desired ternary output in a differentiable manner as t increases. In implementation, instead of the commonly used layer-wise weight reconstruction, CAT-Q adopts cross-layer output reconstruction based on a sliding window (Wang et al., 2026a). In this paper, for all experiments with ScaleQ-1.58, we use the same hyperparameter settings as in CAT-Q.

2.2 Attend to Your Own Thoughts

Regarding the calibration scheme, existing PTQ methods typically rely on a small set of calibration samples randomly drawn from web text corpora, e.g., C4 and WikiText2 to optimize quantization parameters. CAT-Q follows this de facto calibration practice. By introducing a new calibration method, Attend to Your Own Thoughts (AYOT) illustrated in Fig. 2, ScaleQ-1.58 simply combines AYOT with CAT-Q to achieve the primary goal of this paper: ternarizing pre-trained high-precision reasoning LLMs into 1.58-bit models that retain their CoT reasoning abilities across diverse and complex tasks. *AYOT is built upon two simple design principles: (1) selecting domain-specific calibration samples and (2) leveraging self-generated CoT demonstrations*, which are motivated by an empirical study exploring the scaling behavior of CAT-Q across five challenging mathematics (Math-500, GSM8K and Omni-MATH) and coding (HumanEval+ and MBPP+) tasks.

In this study, using 4M calibration tokens throughout, we first evaluate CAT-Q on the pre-trained Qwen3-4B using the de facto calibration scheme. Surprisingly, we observe severe performance collapse, yielding zero or near-zero accuracy on most tasks, regardless of whether the calibration samples are drawn from C4 or WikiText2. We conjecture that such catastrophic degradation arises because web text corpora primarily encode generic word-to-word relationships but lack domain-specific reasoning content. To investigate this hypothesis, we instead construct the calibration set from two public domain-specific datasets comprising question–answer pairs, MetaMathQA(Yu et al., 2024) for mathematics and OpenCodeInstruct (Ahmad et al., 2025) for coding. Calibration samples are drawn evenly in each domain, and have no sample overlaps with our five evaluation datasets. The average accuracy of the resulting 1.58-bit model across five tasks improves to 17.43%, highlighting the importance of selecting domain-specific calibration samples, which forms the first design principle of AYOT. However, the performance remains far from satisfactory. Considering that CoT interprets the step-by-step problem-solving process of pre-trained reasoning LLMs, we hypothesize that the absence of CoT demonstrations during ternarization is a major cause of the poor performance obtained when

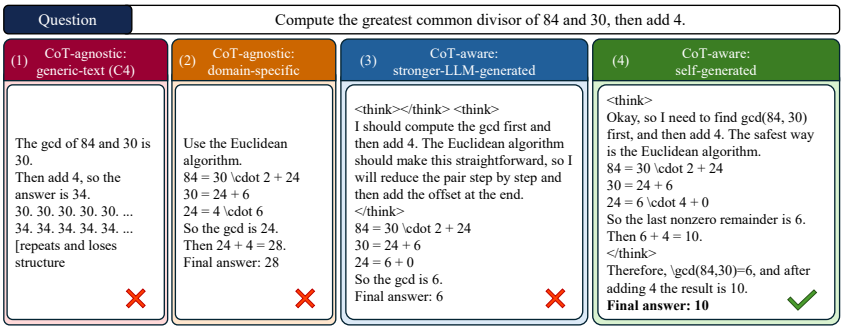

Figure 3: Reasoning examples of 1.58-bit Qwen3-4B models obtained via CAT-Q under different calibration schemes. Our AYOT significantly improves the capability of the resulting 1.58-bit model on complex mathematics tasks. *More examples are provided in Section H of the Appendix.*

domain-specific calibration data is used in a straightforward manner, i.e., taking [*question, answer*] pairs as the context input. Driven by this insight, we employ two strategies to generate reasoning traces and final answer for each sampled question, forming [*question, reasoning traces, generated answer*] triplets as the context input. Specifically, we first use DeepSeek-R1-671B (Guo et al., 2025), a sufficiently large and strong LLM, to generate CoT demonstrations for sampled questions, which serve as proxies for ground-truth reasoning traces and answers that are originally unavailable in MetaMathQA and OpenCodeInstruct. It further improves the mean accuracy from 17.43% to 20.06%, but it still remains low. This is likely because, under extreme ternary setting in the PTQ regime, the 1.58-bit model quantized from the pre-trained target LLM is unable to mimic the reasoning capability of a significantly larger and stronger LLM, due to differences in architecture, model scale and knowledge density. Therefore, the second design principle of AYOT is established: using the pre-trained target LLM itself to generate reasoning traces and final answer for each sampled question. As a result, AYOT substantially boosts the mean accuracy to 45.60%, yielding a remarkable absolute gain of 25.54% over using DeepSeek-R1-671B. Compared to the prior best 1.58-bit reasoning LLM, BitNet b1.58 2B4T trained from scratch on a massive amount of 4T tokens, our 1.58-bit Qwen3-4B ternarized with 4M calibration tokens shows an absolute gain of 8.97%, while requiring 1,000,000$\times$ fewer calibration tokens for quantization. Detailed results and illustrative reasoning examples of 1.58-bit Qwen3-4B models produced by CAT-Q under the above four calibration schemes are shown in Fig. 1 and Fig. 3, respectively. To better differentiate these four calibration schemes, we term (1) the first calibration scheme, the default setting of CAT-Q, as *CoT-agnostic: generic-text*; (2) the second calibration scheme, which uses domain-specific samples, as *CoT-agnostic: domain-specific*; (3) the third calibration scheme, which leverages CoT demonstrations and final answers generated by DeepSeek-R1-671B, as *CoT-aware: stronger-LLM-generated*; (4) AYOT, using the pre-trained target LLM itself to generate CoT demonstrations and final answers, as *CoT-aware: self-generated*.

3 Experiments

3.1 Setup

In experiments, we first focus on complex reasoning tasks including mathematics and coding, and then consider scientific logic reasoning, basic language generation, and commonsense reasoning for broader generalization. Specifically, mathematics is evaluated on Math-500 (Hendrycks et al., 2021), GSM8K (Cobbe et al., 2021) and Omni-MATH (Gao et al., 2025); coding is evaluated on HumanEval+ and MBPP+ (Liu et al., 2023); scientific logic reasoning is evaluated on ProofWriter (Tafjord et al., 2021); basic language generation is measured by perplexity on WikiText2 (Merity et al., 2017) and C4 (Raffel et al., 2020); and commonsense reasoning is reported as the average accuracy over PIQA (Bisk et al., 2020), ARC-e, ARC-c (Clark et al., 2018), HellaSwag (Zellers et al., 2019), and Winogrande (Sakaguchi et al., 2021). To validate the scaling properties of ScaleQ-1.58 across diverse model scales and architectures, we instantiate ScaleQ-1.58 on the Qwen3 family, including dense models of 1.7B, 4B, 8B, 14B, and 32B parameters, as well as the MoE variants Qwen3-30B-A3B and Qwen3-235B-A22B. To assess generalization across different model families, we additionally include DeepSeek-R1-Distill-Llama-70B (Guo et al., 2025), a strong distilled reasoning model. The calibration set is constructed by sampling prompts from MetaMathQA (Yu et al., 2024) and OpenCodeInstruct (Ahmad et al., 2025), generating responses with the pre-trained model to be

Table 1: Performance of eight LLMs ternarized by ScaleQ-1.58 on mathematics and coding tasks. We include the best prior 1.58-bit reasoning LLM, BitNet b1.58 2B4T (Ma et al., 2025), for a comparison. The metric is accuracy (%). #Tokens denotes the number of tokens for quantization.

Model	#Bits	#Tokens	Math-500 ↑	GSM8K ↑	Omni-MATH ↑	HumanEval+ ↑	MBPP+ ↑
BitNet b1.58 2B4T	W1.58A16	4T	43.40	58.38	-	38.40	37.03
Qwen3-1.7B	W16A16	-	90.80	84.08	27.76	76.22	53.97
	W1.58A16	**4M**	38.60	54.35	12.23	34.14	33.33
Qwen3-4B	W16A16	-	96.80	88.10	34.64	85.37	61.90
	W1.58A16	**4M**	58.40	61.56	14.93	53.98	39.15
Qwen3-8B	W16A16	-	96.80	88.85	34.67	87.20	63.49
	W1.58A16	**4M**	63.00	71.72	16.96	54.88	44.71
Qwen3-14B	W16A16	-	97.20	91.35	37.78	86.59	64.29
	W1.58A16	**4M**	76.80	79.19	22.49	74.39	50.26
Qwen3-32B	W16A16	-	96.80	93.63	44.08	87.20	66.93
	W1.58A16	**4M**	85.20	87.34	28.84	84.15	54.50
Qwen3-30B-A3B	W16A16	-	97.60	94.84	41.10	86.59	62.96
	W1.58A16	**4M**	76.80	84.99	23.62	60.37	44.97
DeepSeek-R1-Distill-Llama-70B	W16A16	-	96.60	93.03	44.26	88.41	61.90
	W1.58A16	**4M**	87.60	88.85	30.16	85.74	56.88
Qwen3-235B-A22B	W16A16	-	98.60	96.28	45.51	89.02	69.31
	W1.58A16	**4M**	78.80	85.97	26.88	67.07	48.51

quantized, and concatenating each prompt with its corresponding reasoning trace and answer to form context input. *By default, the number of calibration tokens is 4M and we use W1.58A16 quantization.*

3.2 Main Results

Scaling across Different Models and Tasks. Table 1 summarizes the results of ScaleQ-1.58 across a wide range of models (from 1.7B to 235B parameters, covering both dense and MoE architectures, and including Qwen and Llama families) on mathematics and coding tasks of varying difficulty levels, under the default budget of 4M calibration tokens. ScaleQ-1.58 performs consistently well across all models and tasks. We find that smaller models are more sensitive to quantization than larger ones, and under comparable parameter counts, MoE models exhibit higher sensitivity than dense ones. Notably, compared to the best prior 1.58-bit reasoning LLM, BitNet b1.58 2B4T (Ma et al., 2025), our ternary Qwen3-1.7B achieves competitive performance (over 90.52% of the average score of BitNet b1.58 2B4T on four math/coding tasks) while using 1,000,000× fewer tokens for quantization. Moreover, our ternary Qwen3-4B surpasses BitNet b1.58 2B4T by an absolute 8.97%, highlighting the extreme efficiency of ScaleQ-1.58 in terms of token count for ternarization.

Scaling with Increasing Calibration Token Count. We next examine the scaling behavior of ScaleQ-1.58 with respect to the calibration-token budget. As shown in Table 2, increasing the calibration token count from 256K to 16M tokens consistently improves the performance of the ternary Qwen3-4B across all evaluated tasks. Fig. 4 further shows that accuracy continues to increase with more calibration tokens, and the trend does not appear to plateau. This highlights the importance of the calibration token budget. Even with only 4M tokens our method already outperforms BitNet b1.58 2B4T, and as the calibration budget grows, this advantage becomes even more pronounced.

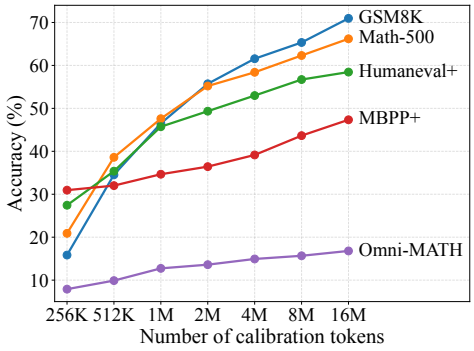

Figure 4: Effect of increasing calibration tokens evaluated on ternarizing Qwen3-4B.

3.3 Ablation Studies

We also perform a lot of ablation studies on Qwen3-4B under W1.58A16 to identify the key factors for ScaleQ-1.58, choosing this model for its balanced size and decent performance. Additional experiments under W2A16 and W4A16 confirm the generality ability of our AYOT.

Table 2: Performance of Qwen3-4B ternarized by ScaleQ-1.58 with different numbers of calibration tokens on mathematics and coding tasks. Underlines denote our default configuration. All experiments use the same number of epochs and identical hyperparameters, isolating the effect of token budget.

Model	#Bits	#Tokens	Math-500 ↑	GSM8K ↑	Omni-MATH ↑	HumanEval+ ↑	MBPP+ ↑
BitNet b1.58 2B4T	W1.58A16	4T	43.40	58.38	-	38.40	37.03
Qwen3-4B	W16A16	-	96.80	88.10	34.64	85.37	61.90
	W1.58A16	256K	20.90	15.84	7.90	27.44	30.95
	W1.58A16	512K	38.60	34.53	9.89	35.37	32.01
	W1.58A16	1M	47.60	46.50	12.74	45.73	34.66
	W1.58A16	2M	55.20	55.72	13.60	49.34	36.51
	W1.58A16	4M	58.40	61.56	14.93	53.98	39.15
	W1.58A16	8M	62.31	65.34	15.67	56.71	43.65
	W1.58A16	16M	**66.20**	**70.96**	**16.80**	**58.45**	**47.35**

Table 3: Performance of Qwen3-4B ternarized by ScaleQ-1.58 with different calibration set sizes and training iterations on mathematics and coding tasks.

#Bits	#Tokens	#Iterations	Math-500 ↑	GSM8K ↑	Omni-MATH ↑	HumanEval+ ↑	MBPP+ ↑
W16A16	-	-	96.80	88.10	34.64	85.37	61.90
W1.58A16	256K	2560	20.90	15.84	7.90	27.44	30.95
W1.58A16	256K	5120	27.40	22.74	8.99	29.27	31.75
W1.58A16	256K	10240	31.20	26.54	9.71	30.49	32.54
W1.58A16	2M	2560	36.80	30.17	9.96	36.59	33.60
W1.58A16	2M	5120	48.40	36.77	11.88	44.51	34.92
W1.58A16	2M	10240	**52.80**	**50.42**	**13.03**	**47.56**	**35.98**

Table 4: Performance of Qwen3-4B ternarized by CAT-Q with different calibration schemes.

Calibration Scheme	Math-500 ↑	GSM8K ↑	Omni-MATH ↑	HumanEval+ ↑	MBPP+ ↑
CoT-agnostic: generic-text (Wikitext2)	0.00	2.65	0.43	0.00	0.00
CoT-agnostic: generic-text (C4)	0.00	14.48	2.71	0.00	0.00
CoT-agnostic: domain-specific	24.20	28.65	5.04	17.88	11.38
CoT-aware: stronger-LLM-generated	31.80	30.78	6.98	16.46	14.29
AYOT (CoT-aware: self-generated)	**58.40**	**61.56**	**14.93**	**53.98**	**39.15**

Calibration Set Size vs. Quantization Budget. Table 2 shows that more calibration tokens improve the performance of ScaleQ-1.58 under the same number of calibration epochs. To dispel any concern that this gain might be due solely to more iterations, we further conduct a controlled analysis in Table 3. With the same number of calibration tokens, increasing the number of optimization iterations yields consistent gains. For 256K tokens, raising the number of optimization iterations from 2560 to 10240 improves the average accuracy across the five tasks by 5.49%. Yet under the same 10240 optimization iterations, scaling the calibration token count from 256K to 2M tokens produces a much larger gain of 13.86% on average. This indicates that data diversity, not just the number of optimization iterations, drives the performance improvement.

Different Calibration Schemes. Having established the importance of calibration set size, we now study the impact of different calibration schemes. As a qualitative illustration, Fig. 3 presents how the ternary models respond with four different calibration schemes. In Table 4, we present a more detailed numerical comparison of their performance. Two main observations stand out. First, selecting domain-specific calibration samples leads to better performance than generic-text data (Wikitext2 and C4). Second, when using domain-specific data, self-generated calibration samples prove to be more effective than either the CoT-agnostic responses from the original dataset or CoT-aware responses generated by a stronger model. These results validate the two key principles proposed by our AYOT.

Task-Specific and Long-Context Calibration. Table 5 examines two factors: task relevance of calibration data and sequence length. Replacing part of domain-specific calibration data (general mathematics/code) with task-specific samples from GSM8K and MBPP training sets substantially improves performance on the corresponding target tasks while slightly improving others. Extending the sequence length from 2048 to 4096 consistently boosts results across different tasks, with larger gains on long-form reasoning benchmarks such as Math-500 and HumanEval+. These results confirm that task-specific data provides finer-grained alignment beyond domain matching, and longer context captures more complete reasoning traces that better match the model's own inference generation.

Table 5: Effect of calibration sequence length and introducing task-specific calibration data for ScaleQ-1.58 on Qwen3-4B. When task-specific data is introduced, a part of the calibration data is replaced with samples from the training sets of GSM8K and MBPP while keeping total budget fixed at 4M tokens. Shaded cells denote target tasks aligned with the task-specific calibration data.

Task-Specific	SeqLen	Mathematics			Coding	
		Math-500↑	GSM8K↑	Omni-MATH↑	HumanEval+↑	MBPP+↑
	2048	58.40	61.56	14.93	53.98	39.15
✓	2048	59.40	70.45	15.21	55.37	43.65
	4096	68.60	62.33	18.11	64.63	42.06
✓	4096	**68.80**	**73.62**	**18.77**	**66.46**	**45.24**

Table 6: Generalization of ScaleQ-1.58 on Qwen3-4B to the scientific logic reasoning task ProofWriter. We compare calibration with and without a portion of its training samples.

#Bits	Calibration Data Domains	Scientific Logic Reasoning (ProofWriter)				
		Depth=1↑	Depth=2↑	Depth=3↑	Depth=5↑	Average↑
W16A16	-	92.83	89.33	84.83	72.67	84.92
W1.58A16	Mathematics, Code	47.33	42.33	36.00	32.33	39.50
W1.58A16	Mathematics, Code, ProofWriter	91.33	85.83	80.00	71.67	82.21

Table 7: Generalization of ScaleQ-1.58 on Qwen3-4B to a wide range of tasks. We consider four types of calibration data: Generic (C4), Math/Code (MetaMathQA, OpenCodeInstruct), Scientific (ProofWriter), and Task-specific (GSM8K, MBPP). The table reports performance on mathematics & coding (average accuracy of Math-500, GSM8K, Omni-MATH, HumanEval+ and MBPP+), language modeling (average perplexity of WikiText2 and C4), commonsense reasoning (average accuracy of PIQA, ARC-e, ARC-c, HellaSwag and Winogrande), and scientific logic reasoning (ProofWriter).

Calibration Data Domains				Mathematics & Coding					Language Modeling Avg.↓	Commonsense Reasoning Avg.↑	Scientific Logic Reasoning Avg.↑
Generic	Math/Code	Scientific	Task-Specific	Math-500↑	GSM8K↑	Omni-MATH↑	HumanEval+↑	MBPP+↑			
✓				0.00	14.48	2.71	0.00	0.00	**25.40**	**58.17**	1.38
	✓			58.40	61.56	14.93	53.98	39.15	35.34	53.23	39.50
✓	✓			57.60	60.80	14.89	53.65	40.74	28.18	57.27	47.46
✓	✓	✓		58.00	64.37	15.11	53.05	41.27	28.64	57.05	**83.92**
✓	✓	✓	✓	**59.10**	**70.36**	**16.71**	**54.27**	**42.06**	33.14	52.81	83.54

Table 8: Generalizing AYOT from 1.58-bit to other bit-width settings on Qwen3-4B. Specifically, we apply AYOT to CAT-Q for W1.58A16, AWQ, GPTQ, and SliderQuant for W2A16, W3A16, W4A16.

#Bits	Quantizer	C4 ↓		Math-500 ↑		GSM8K ↑		Omni-MATH ↑		HumanEval+ ↑		MBPP+ ↑	
		Generic	AYOT	Generic	AYOT	Generic	AYOT	Generic	AYOT	Generic	AYOT	Generic	AYOT
W1.58A16	CAT-Q	**24.40**	31.45	0.00	**58.40**	14.48	**61.56**	2.71	**14.93**	0.00	**53.98**	0.00	**39.15**
W2A16	AWQ	**2.9e7**	3.1e7	0.00	0.00	0.00	0.00	0.23	**0.34**	0.00	0.00	0.00	0.00
	GPTQ	**1.5e2**	8.2e2	0.00	0.00	1.06	**1.74**	0.32	**0.36**	0.00	0.00	0.00	0.00
	SliderQuant	**27.20**	40.36	1.40	**60.20**	16.48	**68.39**	3.39	**16.30**	1.22	**54.27**	0.79	**42.59**
W3A16	AWQ	**26.76**	28.70	76.00	**86.60**	74.83	**77.10**	22.40	**23.01**	68.90	**73.17**	53.70	**57.41**
	GPTQ	**24.11**	28.10	80.00	**92.80**	73.62	**79.53**	18.41	**28.86**	70.73	**76.83**	54.23	**58.73**
	SliderQuant	**22.40**	28.13	81.60	**95.00**	75.28	**80.82**	23.31	**31.07**	74.39	**81.71**	57.41	**59.52**
W4A16	AWQ	23.38	**23.33**	92.00	**94.60**	78.24	**81.05**	29.86	**32.84**	79.27	**82.93**	60.05	**62.43**
	GPTQ	**21.95**	22.62	91.80	**94.80**	78.08	**81.80**	28.81	**33.19**	78.65	**82.32**	60.85	**62.16**
	SliderQuant	**17.05**	18.22	92.80	**95.00**	78.47	**83.02**	29.92	**33.33**	80.49	**83.54**	60.85	**63.49**

Generalizing ScaleQ-1.58 to Scientific Logic Reasoning. As shown in Table 6, ScaleQ-1.58 generalizes effectively to the scientific logic reasoning task, ProofWriter, when using task-specific calibration data (256 examples from the training set of ProofWriter). That means our method has good potential to generalize to other challenging reasoning tasks beyond mathematics and coding.

Generalizing ScaleQ-1.58 to a Wide Range of Tasks via Diverse Calibration Data Composition. Table 7 provides a comprehensive view of how different calibration compositions affect ternary model's performance on various downstream tasks including mathematics and coding, scientific logic reasoning, language modeling (perplexity) and commonsense reasoning. The calibration data choice strongly shapes downstream performance, data from each domain most benefits its corresponding tasks. Mixing data from multiple domains yields balanced performance across tasks. Finally, adding task-specific training samples from the target benchmarks further improves the corresponding tasks.

Generalizing AYOT to Other Low-Bit Quantization Settings. Finally, we study the generalization ability of AYOT from ternary quantization to other low-bit quantization settings. As shown in Table 8, besides CAT-Q for the extreme W1.58A16 quantization, we additionally integrate AYOT with GPTQ (Frantar et al., 2023), AWQ (Lin et al., 2024b) and SliderQuant (Wang et al., 2026a), and evaluate their performance under W2A16, W3A16 and W4A16 settings. We can see that our

AYOT mostly brings obvious performance gains to four baseline methods under all bit-width settings compared to the de facto calibration scheme. Comparatively, the performance gains are especially pronounced in extremely low-bit quantization settings such as W1.58A16 and W2A16. This indicates that AYOT is a general calibration scheme to PTQ with different bit-width settings.

4 Related Work

In PTQ research for LLMs, some prior studies have explored the impact of calibration data from different perspectives. Most existing methods routinely use a small set of fixed-length samples for calibration, e.g., 128 samples randomly drawn from C4 or WikiText2, each comprising 2048 tokens. Williams and Aletras (Williams & Aletras, 2024) present the first systematic study on the role of calibration data in post-training compression for commonsense reasoning tasks, considering two quantization methods for 4-bit weights, GPTQ (Frantar et al., 2023) and SpQR (Dettmers et al., 2024), and two pruning methods, SparseGPT (Frantar & Alistarh, 2023) and Wanda (Sun et al., 2024). They show that the choice of calibration data sampled from different web text corpora or pre-training data will affect their performance. Many subsequent works also focus on commonsense reasoning tasks, but investigate different aspects of calibration data for post-training compression. (Williams et al., 2025) evaluates GPTQ and AWQ (Lin et al., 2024b) for 4-bit weight quantization, together with SparseGPT and Wanda for pruning, using either real or synthetic data. Their results indicate that, for the tested two pruning methods, synthetic text generated by the target model itself usually yields higher accuracy than randomly sampled web text, which the authors attribute to its closer approximation of the pre-training data distribution. In contrast, they find that the tested two PTQ methods are much less sensitive to the choice of calibration data. The authors of (Oh & Oh, 2025) study the impact of calibration sequence length using six post-training compression methods, including two PTQ methods SmoothQuant (Xiao et al., 2023) and AWQ (Lin et al., 2024b), and show that using shorter calibration sequences outperforms the commonly used fixed-length of 2048 tokens. (Paglieri et al., 2024) finds that, newer LLMs such as Llama-2 and Llama3 are significantly less sensitive to varying calibration data than the older OPT model, when using GPTQ, AWQ, and SmoothQuant. (Lee et al., 2023) proposes a sequence-length-aware calibration strategy to improve PTQ in 4-bit weight and 8-bit activation quantization. RSQ (Sung et al., 2025) reveals that, for fixed-length calibration data, a small subset of tokens, particularly those at the beginning and end of sequences, are more important for layer-wise quantization methods such as GPTQ (Frantar et al., 2023) and QuaRot (Ashkboos et al., 2024). MBS (Zeng et al., 2024) introduces a calibration scheme tailored for improving LLM compression methods including GPTQ, whose core insight is to sample the calibration data of different languages proportionally to the language distribution of the model-specific pre-training data.

In summary, these prior studies primarily focus on conventional low-bit quantization settings (e.g., 4-bit weights or 4-bit weights and 8-bit activations) and simple commonsense reasoning tasks. Recently, (Liu et al., 2025a) benchmarks the impact of quantization on reasoning LLMs, considering five PTQ methods: GPTQ, AWQ, SmoothQuant, FlatQuant (Sun et al., 2025) and KVQuant (Hooper et al., 2024) under 8-bit, 4-bit and 3-bit quantization settings. It reveals that these PTQ methods generally incur serious performance degradation when model precision is reduced to 3-bit. In addition, it finds that the domain of calibration data has little effect on the tested quantization methods, except for GPTQ. In contrast, this work differs from them clearly in focus and method formulation.

5 Conclusion

In this paper, we present ScaleQ-1.58, a post-training ternarization framework for reasoning LLMs. Built upon CAT-Q in the PTQ regime, its core contribution is AYOT, a simple and effective calibration method, which enables ScaleQ-1.58 to generate 1.58-bit reasoning LLMs capable of handling complex reasoning tasks at substantially reduced costs than state-of-the-art QAT-based methods. Through extensive experiments, we show that ScaleQ-1.58 exhibits favorable scaling properties across different LLM architectures, model scales and calibration data sizes, as well as a wide range of tasks with varying difficulty levels. Besides, AYOT generalizes well from the 1.58-bit setting to other quantization bit widths. We hope this work will inspire future research on building 1.58-bit reasoning LLMs in a cost-efficient and accurate manner.

Reproducibility Statement

To facilitate reproducibility, all experiments are conducted using publicly available models and datasets. Implementation details regarding our method and baselines, experimental settings, hyperparameters, and evaluation protocols are provided in the main paper and appendix. We will release our code and models upon publication.

AI Use Statement

We used generative AI tools solely to assist with proofreading the manuscript. The authors reviewed the AI-assisted edits and take responsibility for the final content of this paper.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. Opencodeinstruct: A large-scale instruction tuning dataset for code llms. *arXiv preprint arXiv:2504.04030*, 2025.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. In *NeurIPS*, 2024.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *AAAI*, 2020.

Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher M De Sa. Quip: 2-bit quantization of large language models with guarantees. In *NeurIPS*, 2023.

Tianqi Chen, Zhe Li, Weixiang Xu, Zeyu Zhu, Dong Li, Lu Tian, Emad Barsoum, Peisong Wang, and Jian Cheng. Ternaryllm: Ternarized large language model. *arXiv preprint arXiv:2406.07177*, 2024.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. In *NeurIPS*, 2022.

Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. Spqr: A sparse-quantized representation for near-lossless llm weight compression. In *ICLR*, 2024.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.

Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. Learned step size quantization. In *ICLR*, 2020.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *ICML*, 2023.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. In *ICLR*, 2023.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, et al. Omni-math: A universal olympiad level mathematic benchmark for large language models. In *ICLR*, 2025.

Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *ICCV*, 2019.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. In *NeurIPS*, 2024.

Xing Hu, Yuan Cheng, Dawei Yang, Zukang Xu, Sifan Zhou, Jiangyong Yu, Chen Xu, Zhixuan Chen, Zhe Jiang, and Zhihang Yuan. Ostquant: Refining large language model quantization with orthogonal and scaling transformations for better distribution fitting. In *ICLR*, 2025.

Hong Huang, Decheng Wu, Rui Cen, Guanghua Yu, Zonghang Li, Kai Liu, Jianchen Zhu, Peng Chen, Xue Liu, and Dapeng Wu. Tequila: Trapping-free ternary quantization for large language models. *arXiv preprint arXiv:2509.23809*, 2025.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *NeurIPS*, 2016.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. In *Findings of EMNLP*, 2020.

Ayush Kaushal, Tejas Vaidhya, Arnab Kumar Mondal, Tejas Pandey, Aaryan Bhagat, and Irina Rish. Surprising effectiveness of pretraining ternary language model at scale. In *ICLR*, 2025.

Janghwan Lee, Minsoo Kim, Seungcheol Baek, Seok Joong Hwang, Wonyong Sung, and Jungwook Choi. Enhancing computation efficiency in large language models through weight and activation quantization. In *EMNLP*, 2023.

Fengfu Li, Bin Liu, Xiaoxing Wang, Bo Zhang, and Junchi Yan. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

Haokun Lin, Haobo Xu, Yichen Wu, Jingzhi Cui, Yingtao Zhang, Linzhan Mou, Linqi Song, Zhenan Sun, and Ying Wei. Duquant: Distributing outliers via dual transformation makes stronger quantized llms. In *NeurIPS*, 2024a.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. In *MLSys*, 2024b.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *NeurIPS*, 2023.

Ruikang Liu, Yuxuan Sun, Manyi Zhang, Haoli Bai, Xianzhi Yu, Tiezheng Yu, Chun Yuan, and Lu Hou. Quantization hurts reasoning? an empirical study on quantized reasoning models. In *COLM*, 2025a.

Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant: Llm quantization with learned rotations. In *ICLR*, 2025b.

Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit llms: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764*, 2024.

Shuming Ma, Hongyu Wang, Shaohan Huang, Xingxing Zhang, Ying Hu, Ting Song, Yan Xia, and Furu Wei. Bitnet b1.58 2b4t technical report. *arXiv preprint arXiv:2504.12285*, 2025.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *ICLR*, 2017.

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

Jaehoon Oh and Dokwan Oh. Beyond fixed-length calibration for post-training compression of llms. In *Findings of EMNLP*, 2025.

Davide Paglieri, Saurabh Dash, Tim Rocktäschel, and Jack Parker-Holder. Outliers and calibration sets have diminishing effect on quantization of modern llms. In *ICML Workshop*, 2024.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 2020.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Computer Vision – ECCV 2016*, 2016.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 2021.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. In *ICLR*, 2024.

Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *AAAI*, 2020.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. In *ICLR*, 2024.

Yuxuan Sun, Ruikang Liu, Haoli Bai, Han Bao, Kang Zhao, Yuening Li, Jiaxin Hu, Xianzhi Yu, Lu Hou, Chun Yuan, et al. Flatquant: Flatness matters for llm quantization. In *ICML*, 2025.

Yi-Lin Sung, Prateek Yadav, Jialu Li, Jaehong Yoon, and Mohit Bansal. Rsq: Learning from important tokens leads to better quantized llms. *arXiv preprint arXiv:2503.01820*, 2025.

Oyvind Tafjord, Bhavana Dalvi, and Peter Clark. Proofwriter: Generating implications, proofs, and abductive statements over natural language. In *Findings of the ACL*, 2021.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1.5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025.

Qwen Team. Qwen3.5-omni technical report. *arXiv preprint arXiv:2604.15804*, 2026.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Hongyu Wang, Shuming Ma, and Furu Wei. Bitnet v2: Native 4-bit activations with hadamard transformation for 1-bit llms. *arXiv preprint arXiv:2504.18415*, 2025.

Shigeng Wang, Chao Li, Yangyuxuan Kang, Jiawei Fan, Zhonghong Ou, and Anbang Yao. Sliderquant: Accurate post-training quantization for llms. In *ICLR*, 2026a.

Shigeng Wang, Chao Li, Yangyuxuan Kang, Jiawei Fan, and Anbang Yao. Cat-q: Cost-efficient and accurate ternary quantization for llms. In *ICML*, 2026b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

Miles Williams and Nikolaos Aletras. On the impact of calibration data in post-training quantization and pruning. In *ACL*, 2024.

Miles Williams, George Chrysostomou, and Nikolaos Aletras. Self-calibration for language model quantization and pruning. In *NAACL*, 2025.

Xun Wu, Shaohan Huang, Wenhui Wang, Ting Song, Li Dong, Yan Xia, and Furu Wei. Bitnet distillation. *arXiv preprint arXiv:2510.13998*, 2025.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *ICML*, 2023.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In *NeurIPS*, 2022.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models. In *ICLR*, 2024.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *ACL*, 2019.

Hongchuan Zeng, Hongshen Xu, Lu Chen, and Kai Yu. Multilingual brain surgeon: Large language models can be compressed leaving no language behind. In *LREC-COLING*, 2024.

Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. Ternarybert: Distillation-aware ultra-low bit bert. In *EMNLP*, 2020.

Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. In *ICLR*, 2017.

SUPPLEMENTARY MATERIALS CONTENTS

- Section A: Limitations and broader impact.
- Section B: Datasets used in experiments.
- Section C: Implementation details of ScaleQ-1.58.
- Section D: More experimental results.
- Section E: Ternarization time cost.
- Section F: Runtime acceleration on hardware.
- Section G: Reasoning context length statistics.
- Section H: Qualitative reasoning samples.

A LIMITATIONS AND BROADER IMPACT

As a PTQ method, our ScaleQ-1.58 still uses a relatively small set of proper samples for sequential calibration, quantizing one window containing multiple layers at a time, without the need of re-training or fine-tuning on task data. Regarding efficiency, ScaleQ-1.58 uses more calibration samples and is slower than conventional efficient PTQ methods such as GPTQ and AWQ tailored for conventional low-bit quantization (e.g., 8-bit and 4-bit). However, owing to its extreme low-bit format $\{1, 0, -1\}$, 1.58-bit quantization for reasoning LLMs suffers suffers from serious information loss and instability, making it considerably more difficult to converge than conventional low-bit quantization. Previous state-of-the-art 1.58-bit LLMs (e.g., BitNet, TriLM and Tequila) predominantly rely on prohibitively expensive QAT to combat these issues. Compared to BitNet b1.58 2B4T, the prior best reasoning LLM trained from scratch on 4T tokens, ScaleQ-1.58 with the default 4M calibration tokens only requires 4 and 8 hours on 8 A100-80GB GPUs to quantize pre-trained Qwen3-1.7B and Qwen3-4B into 1.58-bit models while achieving very competitive performance on complex mathematics and coding tasks, respectively. In addition, while we show that ScaleQ-1.58 exhibits favorable scaling properties in addressing pre-trained reasoning LLMs of diverse model architectures (dense and MoE) and scales (up to 235B) as well as various tasks of varying difficulty levels through extensive experiments, there is still considerable room to reduce the performance gap to the FP16 baseline, especially on complex reasoning tasks that remain underexplored in LLM quantization research. Aligned with the research gap in 1.58-bit quantization for reasoning LLMs and the motivation of this work, ScaleQ-1.58 already offers a substantially more practical trade-off between quantization training cost and performance. For broader impact, ScaleQ-1.58 can lower the deployment cost of pre-trained reasoning LLMs on edge, client and cloud devices, generating diverse 1.58-bit high-performance reasoning LLMs in a cost-efficient PTQ manner and making them accessible to the vast majority of researchers.

B DATASETS USED IN EXPERIMENTS

WikiText2 (Merity et al., 2017) is a popular language modeling benchmark consisting of over 2 million tokens from verified Wikipedia articles.

C4 (Raffel et al., 2020)(Colossal Clean Crawled Corpus) is a large-scale dataset primarily used for language modeling tasks, comprising 156 billion clean tokens. It is sourced from cleaned web pages, originally from Common Crawl.

PIQA (Bisk et al., 2020) contains 16,000 training and 3,084 test samples. It focuses on physical reasoning through multiple-choice questions, where models select the most appropriate solution from two options, with exactly one correct answer.

ARC (Clark et al., 2018) is a dataset of 7,787 genuine grade-school level, multiple-choice science questions, assembled to encourage research in advanced question-answering. The dataset is partitioned into a Challenge Set and an Easy Set, where the former contains only questions answered incorrectly by both a retrieval-based algorithm and a word co-occurrence algorithm.

HellaSwag (Zellers et al., 2019) contains 70,000 training and 10,042 validation samples. It focuses on commonsense reasoning by predicting the most plausible sentence continuation, sourced from crowdsourced captions and activity descriptions.

Winogrande (Sakaguchi et al., 2021) is a collection of 44,000 problems, which is formulated as a fill-in-a-blank task with binary options, the goal is to choose the right option for a given sentence which requires commonsense reasoning.

MATH-500 (Hendrycks et al., 2021) comprises 500 challenging competition-level mathematics problems sampled from the MATH dataset. These problems span various topics such as algebra, geometry, number theory, and probability, and are designed to test a model's ability to perform complex mathematical reasoning and generate step-by-step solutions.

Omni-MATH (Gao et al., 2025) is a large-scale benchmark for olympiad-level mathematical reasoning, containing 4,428 problems collected from diverse national and international mathematics competitions. The dataset covers a broad range of advanced topics and is designed to evaluate models on difficult multi-step problem solving beyond standard school-level math benchmarks.

GSM8K (Cobbe et al., 2021) is a dataset of 8,792 high-quality, linguistically diverse grade school math word problems created by human problem writers. The dataset is segmented into 7,473 training problems and 1,319 test problems, each requiring multi-step reasoning and basic arithmetic operations to solve.

HumanEval+ (Liu et al., 2023) is an extension of the HumanEval dataset, consisting of 164 original programming problems designed to assess the functional correctness of code generated by language models. Each problem includes a function signature, a docstring specifying the intended functionality, and multiple test cases for evaluation.

MBPP+ (Liu et al., 2023) is an augmented version of the Mostly Basic Programming Problems (MBPP) dataset, comprising approximately 378 crowd-sourced Python programming tasks. Each task includes a natural language description, a reference solution, and three test cases, aiming to evaluate models' abilities in basic programming and problem-solving.

ProofWriter (Tafjord et al., 2021) is a scientific logic reasoning dataset built from synthetic natural language theories consisting of facts and rules, paired with hypotheses labeled as true, false, or unknown. It also provides accompanying proofs and varying reasoning depths, making it suitable for evaluating multi-step deductive inference.

MetaMathQA (Yu et al., 2024) is a large-scale mathematical instruction dataset constructed by bootstrapping and rewriting existing math problems into diverse question variants. It is designed to improve mathematical reasoning in language models and provides rich supervision spanning both grade-school and competition-style mathematics.

OpenCodeInstruct (Ahmad et al., 2025) is a large-scale code instruction dataset released by NVIDIA, containing millions of instruction-response pairs for code generation and related programming tasks. The dataset also includes unit tests and quality annotations, making it useful for training or calibrating models on code-centric workloads.

C Implementation Details of ScaleQ-1.58

C.1 Details of Calibration Data

Our default calibration setup uses 2048 samples, each truncated or padded to 2048 tokens. For the default mathematics/coding configuration, we randomly sample 1024 examples from MetaMathQA and 1024 examples from OpenCodeInstruct, which serve as the mathematics-domain and code-domain calibration data, respectively. For the task-specific experiments in Table 5, we replace one eighth of the mathematics-domain samples with examples from the GSM8K training set, and one eighth of the code-domain samples with examples from the MBPP training set, which does not overlap with the MBPP+ test set. When using ProofWriter as additional calibration data (shown in Table 6), we sample from the ProofWriter training set and replace one eighth of the mathematics-domain samples and one eighth of the code-domain samples accordingly. Unless otherwise stated, all calibration experiments are run for 60 epochs with a batch size of 3. Other implementation details follow CAT-Q (Wang et al., 2026b).

C.2 Quantization Details

The weights are quantized in a group-wise manner with a fixed group size of $g = 128$, and the ternary reconstruction uses $\mathbf{W} \approx \alpha\mathbf{T}$, discarding the weight mean μ learned to redistribute the pre-trained weights $\mathbf{W}$. As we have clarified in the Method section, this design retains the standard ternary format originally proposed in TWN and its hardware-friendly property. For MoE models, we leave the router layers in full precision and quantize the remaining modules.

D More Experimental Results

D.1 Ablation on Calibration Schemes with Qwen3-30B-A3B

Building on Table 4 in the main paper which reports results on Qwen3-4B, we further perform ablation of the calibration scheme on the larger Qwen3-30B-A3B model. As shown in Table A, the generic-text calibration scheme again leads to model collapse on most mathematics and coding tasks, whereas AYOT remains effective on this larger MoE model.

Table A: Performance of 1.58-bit Qwen3-30B-A3B models produced by CAT-Q with different calibration schemes.

Calibration Schemes	**Math-500 ↑**	**GSM8K ↑**	**Omni-MATH ↑**	**HumanEval+ ↑**	**MBPP+ ↑**
CoT-agnostic: generic-text (C4)	0.00	16.45	5.69	0.00	0.00
AYOT (CoT-aware: self-generated)	**76.80**	**84.99**	**23.62**	**60.37**	**44.97**

Table B: Detailed language modeling and commonsense reasoning results corresponding to Table 7.

Calibration Data Domains					**Language Modeling**		**Commonsense Reasoning**					
#Bits	**Generic**	**Math/Code**	**Scientific**	**Task-Specific**	**WikiText2↓**	**C4↓**	**PIQA↑**	**ARC-e↑**	**ARC-c↑**	**HellaSwag↑**	**Winogrande↑**	**Avg.↑**
W16A16					13.67	16.53	75.19	78.32	53.84	68.41	65.51	68.25
W1.58A16	✓				**26.40**	**24.40**	**71.44**	**64.19**	**38.93**	**55.26**	**61.04**	**58.17**
W1.58A16		✓			39.23	31.45	64.17	58.22	37.48	49.64	56.62	53.23
W1.58A16	✓	✓			29.74	26.61	68.77	63.83	38.23	54.79	60.75	57.27
W1.58A16	✓	✓	✓		30.10	27.17	70.61	63.87	36.58	54.54	59.67	57.05
W1.58A16	✓	✓	✓	✓	37.38	28.89	66.59	54.76	34.39	48.15	60.14	52.81

Table C: Detailed scientific logic reasoning (ProofWriter) results corresponding to Table 7.

Calibration Data Domains					**Scientific Logic Reasoning (ProofWriter)**				
#Bits	**Generic**	**Math/Code**	**Scientific**	**Task-Specific**	**Depth=1↑**	**Depth=2↑**	**Depth=3↑**	**Depth=5↑**	**Average↑**
W16A16					92.83	89.33	84.83	72.67	84.92
W1.58A16	✓				1.67	1.17	1.33	1.33	1.38
W1.58A16		✓			47.33	42.33	36.00	32.33	39.50
W1.58A16	✓	✓			57.67	51.67	42.50	38.00	47.46
W1.58A16	✓	✓	✓		92.50	**88.67**	**80.33**	**74.17**	**83.92**
W1.58A16	✓	✓	✓	✓	**92.83**	87.17	80.00	**74.17**	83.54

D.2 Detailed Results of Calibration Data Composition

Tables B and C provide the detailed results of Table 7 in the main paper. Specifically, they report the full results for language modeling, the individual commonsense reasoning results, and the per-depth scientific logic reasoning (ProofWriter) results that are averaged in the main table.

D.3 Effect of Mathematics/Code Calibration Ratio

Table D compares different proportions of mathematics and code calibration samples for Qwen3-4B at W1.58A16, keeping the total budget fixed at 2048 samples (4M tokens). Increasing the mathematics proportion from 25% to 75% consistently improves results on Math-500, GSM8K, and Omni-MATH, while decreasing performance on HumanEval+ and MBPP+. Conversely, a larger code proportion

Table D: Effect of the mathematics/code calibration ratio on ScaleQ-1.58 with Qwen3-4B. MetaMathQA and OpenCodeInstruct columns report the number of calibration samples from each domain. All quantized configurations use 2048 samples of 2048 tokens each (4M tokens). Bold values denote the best results among the quantized configurations.

Calibration Data		Model	#Bits	#Tokens	Math-500 ↑	GSM8K ↑	Omni-MATH ↑	HumanEval+ ↑	MBPP+ ↑
MetaMathQA	OpenCodeInstruct								
–	–	Qwen3-4B	W16A16	–	96.80	88.10	34.64	85.37	61.90
1024	1024	Qwen3-4B	W1.58A16	4M	58.40	61.56	14.93	53.98	39.15
512	1536	Qwen3-4B	W1.58A16	4M	56.80	56.86	13.78	**56.10**	**41.01**
1536	512	Qwen3-4B	W1.58A16	4M	**60.40**	**64.06**	**15.45**	51.22	37.30

improves both coding benchmarks. The default 1:1 mixture gives intermediate results on all five tasks.

E Ternarization Time Cost

We also report the wall-clock cost of our ternarization procedure under the default settings used in the paper, namely 60 epochs and 4M calibration tokens. Table E summarizes the total ternarization time of ScaleQ-1.58 for different models, measured on a server with 8×A100-80G GPUs.

Table E: Ternarization time cost of ScaleQ-1.58 under the default settings of 60 epochs and 4M calibration tokens measured on a server with 8×A100-80G GPUs.

Model	Training Time (h)
Qwen3-1.7B	4
Qwen3-4B	8
Qwen3-8B	20
Qwen3-14B	32
Qwen3-32B	65
Qwen3-30B-A3B	56
DeepSeek-R1-Distill-Llama-70B	128
Qwen3-235B-A22B	240

F Runtime Acceleration on Hardware

We also evaluate the deployment benefits of 1.58-bit reasoning LLMs produced by ScaleQ-1.58, using `llama.cpp`. Table F compares the deployed 1.58-bit Qwen3-4B model with standard 4-bit and 2-bit baselines. For the real deployment of our 1.58-bit Qwen3-4B model, we use the `TQ1_0` implementation on CPU and the `TQ2_0` implementation on GPU. The 1.58-bit model achieves the smallest memory footprint and the highest decoding throughput on both CPU and GPU. These results confirm that the proposed ternary PTQ method can generate high-performance 1.58-bit reasoning LLMs with practical inference gains.

Table F: Real deployment cases of Qwen3-4B models in different low-bit formats with `llama.cpp`. We report profiling results on both CPU and GPU for standard 4-bit and 2-bit models, together with the 1.58-bit model obtained by ScaleQ-1.58, which is implemented as `TQ1_0` on CPU and `TQ2_0` on GPU. We set the batch size to 1 and the generation length to 512 tokens.

Model	Deployment Format	CPU Memory (GB)	CPU Throughput (Tokens/s)	GPU Memory (GB)	GPU Throughput (Tokens/s)
Qwen3-4B	W4A16 (Q4_K_M)	2.32	19.31	2.32	225.91
	W2A16 (Q2_K_M)	1.55	23.84	1.55	268.51
	ScaleQ-1.58 (TQ1_0 / TQ2_0)	**1.01**	**37.35**	**1.17**	**320.34**

G Reasoning Context Length Statistics

We report dataset statistics to characterize the distributions of context lengths across different evaluation tasks. All statistics are computed on the corresponding test or validation sets, and context

lengths are measured from the complete inference context produced by the high-precision Qwen3-4B model, including the question, the reasoning traces, and the generated answer. Table G summarizes the per-sample context lengths for mathematics, coding, and scientific logic reasoning tasks, while Table H reports the corresponding statistics for commonsense reasoning tasks, which are typically formulated as multiple-choice problems and do not require generating long-form outputs. Complex reasoning tasks have much longer contexts than commonsense reasoning tasks.

Table G: Context length statistics for mathematics, coding, and scientific logic reasoning tasks.

Dataset	#Samples	Context Length		
		Min	Max	Avg.
Math-500	500	1024	30045	5188.34
Omni-MATH	4428	61	9611	7013.57
GSM8K	1319	547	4576	1484.81
HumanEval+	164	406	32760	3358.81
MBPP+	378	372	32760	3277.46
ProofWriter depth = 1	600	265	4421	1518.42
ProofWriter depth = 2	600	388	4405	1883.96
ProofWriter depth = 3	600	507	4432	2275.15
ProofWriter depth = 5	600	569	4424	2674.88
ProofWriter (all depths)	2400	265	4432	1911.34

Table H: Context length statistics for commonsense reasoning tasks.

Dataset	#Samples	Context Length		
		Min	Max	Avg.
PIQA	3084	9	244	35.43
ARC-e	2376	9	169	31.42
ARC-c	1172	10	179	36.63
HellaSwag	10042	17	147	78.41
Winogrande	1267	16	37	21.97

H QUALITATIVE REASONING SAMPLES

Extending beyond the single example in Fig. 3 of the main paper, here we provide additional qualitative reasoning samples to further illustrate the effectiveness of our method. Specifically, we select four mathematics problems (as shown in Section H.1) from Math-500 and four coding problems (as shown in Section H.2) from HumanEval+. These examples highlight how different calibration schemes affect the CoT reasoning ability of the resulting 1.58-bit models. Using AYOT, the model produces more stable reasoning traces and more accurate final answers across both mathematics and coding tasks.

H.1 MATHEMATICS REASONING

Question

```
What is the result when the greatest common factor of 6432 and 132 is increased by 11?
```

CoT-agnostic: generic-text (C4)

```
1. The greatest common factor (GCF) of
6432 and 132 is 132. So, the GCF is 132.
Then, adding 11 to that GCF would be 143.
But that's not the GCF.

2. The GCF of 6432 and 132 is 132. So
adding 11 to 132 is 143, but that's not
the GCF. So the answer is still 132.

3. The GCF of 6432 and 132 is 132. So
adding 11 to 132 is 143, but that's not
the GCF. So the answer is still 132.

4. The GCF is 132, and increasing it by 11
 gives 143, but the GCF is 132. So the
answer is 132.

...
332. 332. 332. 332. 332. 332. 332. ...
[truncated repetitive content]
```

CoT-agnostic: domain-specific

```
...
Okay, let's solve this problem step by
step.
First, we need to find the greatest common
 factor (GCF) of 6432 and 132. To find the
 GCF, we can use the Euclidean algorithm.

Step 1: Divide 6432 by 132.
6432 / 132 = 49.0... (since 132 x 49 =
6432). So the remainder is 0. That means
that 132 is a factor of 6432, and the GCF
is 132.

Wait, that's not correct. Let me check
again. Wait, 132 x 49 = 6432. So the
remainder is 0, which means that 132 is a
factor of 6432. Therefore, the GCF of 6432
 and 132 is 132.

Now, we need to add 11 to the GCF.
132 + 11 = 143.

The answer is $\boxed{143$.
```

CoT-aware: stronger-LLM-generated

```
<think>
Let me first compute the greatest common
factor and only then adjust the result.
This should be easy, so I only need to be
careful not to skip the divisibility step.
</think>

We need the greatest common factor of 6432
 and 132, then we add 11.

Since 132 goes into 6432 exactly, the gcd
is 132.

Indeed,

$$
6432 \div 132 = 48
$$

with remainder 0, so the common factor is
132.

Then

$$
132 + 11 = 143
$$

So the result is

$$
\boxed{143}
$$

The greatest common factor of 6432 and 132
 increased by 11 is $\boxed{143}$.
```

CoT-aware: self-generated

```
<think>
Okay, so I need to find the greatest
common factor of 6432 and 132, and then
add 11. Let me slow down and do the gcd
carefully, because this is exactly the
kind of place where it is easy to
accidentally confuse ``132 divides 6432''
with ``132 is the gcd.''
The cleanest way is the Euclidean
algorithm.
First,
6432 = 132 \cdot 48 + 96
because 132 \cdot 48 = 6336, and 6432 -
6336 = 96.

So
\gcd(6432,132) = \gcd(132,96).

Now continue:
132 = 96 + 36,
96 = 2 \cdot 36 + 24,
36 = 24 + 12,
24 = 2 \cdot 12 + 0.

So the last nonzero remainder is 12, which
 means the gcd is 12.
Then 12 + 11 = 23.
</think>

Using the Euclidean algorithm:
1. $\gcd(6432,132)=\gcd(132,96)$
2. $\gcd(132,96)=\gcd(96,36)$
3. $\gcd(96,36)=\gcd(36,24)$
4. $\gcd(36,24)=\gcd(24,12)$
5. $\gcd(24,12)=12$

Final Answer: $\boxed{23}$
```

Figure A: Reasoning examples of 1.58-bit Qwen3-4B models obtained via differentiable ternarization under different calibration schemes. The mathematics problem involves calculating the greatest common factor.

Question

```
For what value of $c$ will the circle with equation $x^2 - 10x + y^2 + 6y + c = 0$ have a
radius of length 1?
```

CoT-agnostic: generic-text (C4)

```
</think>
</think>
Okay, let's try to figure out this problem.
 The equation given is x^2 - 10x + y^2 + 6
y + c = 0.
We need to find the value of c such that
the circle has a radius of 1.

First, I need to understand what this
equation represents. It's a circle
equation, but it's not the standard one.
The standard equation for a circle is x^2
+ y^2 + c = 0, but this one has more terms.


Wait, maybe this equation is an equation
of a circle, but with some extra terms.
Let me think about how equations like this
 can be used to describe a circle.

Maybe I should move terms around first. Or
 maybe I should compare it directly with
the radius formula.
...
The. The answer. The. The. The. The. ...
[truncated repetitive content]
```

CoT-agnostic: domain-specific

```
<think></think> <think>
I should convert this quadratic equation
into the standard form of a circle.
Completing the square in both variables
will expose the radius term directly.

We are given $x^2 - 10x + y^2 + 6y + c = 0.
$

Completing the square gives $ (x-5)^2 + (y
+3)^2 = 34 - c. $

For a circle of radius 1, the right-hand
side should be 1, so the intended value is
 33.

But let me restate the transformed
equation once more:

$$
(x-5)^2 + (y+3)^2 = 34
$$

...
33
33
33
[truncated repetitive content]
```

CoT-aware: stronger-LLM-generated

```
<think>
Okay, let's solve the problem step by step.


First, we need to find the value of $c$
such that
$x^2 - 10x + y^2 + 6y + c = 0$ has a
radius of 1.

For the $x$ terms:
$$x^2 - 10x = (x - 5)^2 - 25$$

For the $y$ terms:
$$y^2 + 6y = (y + 3)^2 - 9$$

So
$$(x - 5)^2 + (y + 3)^2 = 34 - c.$$

Since the radius is 1:
$$34 - c = 1,$$ so $c = 33$.

Let me check the steps again...
Actually, after checking, I should make
sure whether the constant is 34 or 33 in
the final answer box.
</think>

Wait, I must check the answer. The answer
is $\boxed{34}$.
[truncated repetitive content]
```

CoT-aware: self-generated

```
<think>
Okay, so I need to determine $c$ so that
the circle has radius 1. The main step is
clearly to complete the square in $x$ and
$y$ and compare with the standard circle
form.

Starting from

$x^2 - 10x + y^2 + 6y + c = 0$.

For the x-part:
$x^2 - 10x = (x - 5)^2 - 25$.

For the y-part:
$y^2 + 6y = (y + 3)^2 - 9$.

Substituting those back in gives

$(x - 5)^2 + (y + 3)^2 = 34 - c$.

Now compare this with $(x-h)^2 + (y-k)^2 =
 r^2$.
Since the radius is 1, we need $r^2 = 1$,
so $34 - c = 1$.
That gives $c = 33$.
</think>

Thus,
$$34 - c = 1$$
so
$$c = 33.$$

Final Answer: $\boxed{33}$
```

Figure B: Reasoning examples of 1.58-bit Qwen3-4B models obtained via differentiable ternarization under different calibration schemes. The mathematics problem involves solving for an unknown in a circle equation.

Question

```
Find the unique odd integer $t$ such that $0<t<23$ and $t+2$ is the inverse of $t$ modulo
$23$.
```

CoT-agnostic: generic-text (C4)

```
</think>
Okay, so I need to find an odd integer t
that is less than 23 and such that t + 2
is the inverse of t modulo 23.
The question is asking for an odd integer
t that is less than 23, and when you add 2
 to it, the result is the inverse of t
modulo 23.

Wait, but how does that work? Let me think.

If the inverse of t is the number that
when you add 2 to it, it's the inverse of
t...
Maybe the inverse is the number that when
you add 2 to it, the result is the inverse
 of t.

For example, if t is 1, then the inverse
would be 21, but that's not really the
inverse.
Maybe I should try small odd values and
see what happens.

...
the to the to to the to to the to to the
to ...
[truncated repetitive content]
```

CoT-agnostic: domain-specific

```
...
Okay, let's solve the problem step by step.

We need to find an odd integer $t$ such
that $0 < t < 23$ and $t + 2$ is the
inverse of $t$ modulo 23.

This means
$$(t + 2) \cdot t \equiv 1 \mod 23.$$

So
$$t^2 + 2t - 1 \equiv 0 \mod 23.$$

Now, we can check odd integers between 1
and 21.

1. $t = 1$ gives 2
2. $t = 3$ gives 14
3. $t = 5$ gives 11
4. $t = 7$ gives 16
5. $t = 9$ gives 6
6. $t = 11$ gives 4
...

So after checking, the answer should be 15.


So, the answer is $\boxed{15$.
```

CoT-aware: stronger-LLM-generated

```
<think>
Let me translate the inverse statement
into a modular equation first.
</think>

<think></think>

The condition says that $t+2$ is the
inverse of $t$ modulo 23.

So I interpret that as

$$
t + 2 \equiv 1 \pmod{23}.
$$

Then

$$
t \equiv -1 \pmod{23},
$$

which means

$$
t \equiv 22 \pmod{23}.
$$

So the only possible value in the range is
 22, but that is even.

...
\mod 23 \mod 23 \mod 23 \mod 23 ...
[truncated repetitive content]
```

CoT-aware: self-generated

```
<think>
Okay, so I need to find an odd integer $t$
 with $0 < t < 23$ such that $t + 2$ is
the multiplicative inverse of $t$ modulo
23. So the first thing to do is translate
that sentence into an equation modulo 23.

If $t + 2$ is the inverse of $t$, then
their product should be congruent to 1
modulo 23:

$t(t+2) \equiv 1 \pmod{23}$.

That expands to

$t^2 + 2t - 1 \equiv 0 \pmod{23}$.

Now I can solve this quadratic modulo 23.
Formally,

$t = \frac{-2 \pm \sqrt{8}}{2}$.

Since $10^2 \equiv 8 \pmod{23}$, the two
solutions are $t = 4$ and $t = 17$.
Among these, only 17 is odd and satisfies
the range constraint.
</think>

Therefore the unique odd integer is

$$
\boxed{17}
$$
```

Figure C: Reasoning examples of 1.58-bit Qwen3-4B models obtained via differentiable ternarization under different calibration schemes. The mathematics problem involves solving a modular inverse equation.

Question

```
If $\frac{\sqrt{2x}}{\sqrt{3x-1}}=\frac32$, solve for $x$. Express your answer in simplest
 fractional form.
```

CoT-agnostic: generic-text (C4)

```
</think>
</think>
Okay, so I need to solve this problem
where the ratio of sqrt(2x) over sqrt(3x
-1) equals 3/2.

First, let me understand what the problem
is asking.
The equation is sqrt(2x)/sqrt(3x-1) = 3/2.

So, maybe I need to think about the ratio
as being 2x over 3x-1, but then multiplied
 by...
Wait, maybe that's not the right way to
think about it.

Let me think about the equation again.
Maybe if I square it directly, something
simple happens.
Or maybe I should isolate one square root
first.

...
Let. Let. Let. Let. Let. Let. ...
[truncated repetitive content]
```

CoT-agnostic: domain-specific

```
To solve the equation
$\frac{\sqrt{2x}}{\sqrt{3x - 1}} = \frac
{3}{2}$, we follow these steps.

Step 1: Cross-multiply
$$2\sqrt{2x} = 3\sqrt{3x - 1}$$

Step 2: Square both sides
$$8x = 27x - 9$$

Step 3: Solve
$$x = \frac{9}{19}$$

Step 4: Check the solution
Substituting back gives the correct ratio.

So the solution is 9/19.
Let me write the final answer.

...

The answer is $\boxed{99}$.
The answer is $\boxed{99}$.
[truncated repetitive content]
```

CoT-aware: stronger-LLM-generated

```
<think>
I can remove the radicals by squaring, but
 I should still check the resulting value
against the domain and the original
equation.

If the algebra stays clean, this should
reduce to a single linear equation.
</think>

We are given
$$ \frac{\sqrt{2x}}{\sqrt{3x - 1}} = \frac
{3}{2}. $$

Squaring both sides gives
$$ \frac{2x}{3x - 1} = \frac{9}{4}. $$

Cross-multiplying:
$$ 8x = 27x - 9, $$

so
$$ x = \frac{9}{19}. $$
This satisfies the domain condition $x > \
frac{1}{3}$.

### Final Answer
$$ \boxed{\frac{9}{19}} $$

...
79
80
81
[truncated repetitive content]
```

CoT-aware: self-generated

```
<think>
Okay, so I need to solve

$\frac{\sqrt{2x}}{\sqrt{3x-1}}=\frac{3}{2}
$.

The most natural thing is to square both
sides, but because square roots are
involved I should also check the final
result in the original equation.

Squaring gives
$\frac{2x}{3x-1} = \frac{9}{4}$.

Now cross-multiply:
$8x = 27x - 9$.

So $-19x = -9$, hence
$x = \frac{9}{19}$.

Now I should verify it, not just stop
there.

Substituting in gives $\frac{\sqrt
{18/19}}{\sqrt{8/19}} = \frac{\sqrt{18}}{\
sqrt{8}} = \frac{3}{2}$.
</think>

Final Answer:
$$
\boxed{\frac{9}{19}}
$$
```

Figure D: Reasoning examples of 1.58-bit Qwen3-4B models obtained via differentiable ternarization under different calibration schemes. The mathematics problem involves solving a radical equation.

H.2 CODING REASONING

Question

```
def largest_prime_factor(n: int):
    """Return the largest prime factor of n. Assume n > 1 and is not a prime.
    >>> largest_prime_factor(13195)
    29
    >>> largest_prime_factor(2048)
    2
    """
```

CoT-agnostic: generic-text (C4)

```
</think>
</think>
</think>
</think>
```

CoT-agnostic: domain-specific

```
let's think about how to find the largest
prime factor of a given number.
One approach is to start by dividing the
number by 2 until it's odd.
Then check for odd factors starting from 3
 upwards.

Now, let's implement this in code.

def largest_prime_factor(n: int):
    if n % 2 == 0:
        n = n // 2
        while n % 2 == 0:
            n = n // 2
    i = 3
    while i * i <= n:
        while n % i == 0:
            n = n // i
        i += 2
    return n

For n = 13195, the largest prime factor is
 239.
[truncated repetitive content]
```

CoT-aware: stronger-LLM-generated

```
<think>
I probably want trial division here.
</think>

### Solution Code

def largest_prime_factor(n: int) -> int:
    if n <= 1:
        return -1
    max_prime = 2
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            max_prime = i

At this point I realize the loop is only
collecting divisors, not proving primality.

I also have not reduced n while factoring,
 so this can easily return the wrong thing.


Maybe I should divide by i repeatedly. Or
maybe I should check whether i itself is
prime first.

Wait, this approach may not be correct.
```

CoT-aware: self-generated

```
<think>
Okay, so I need to write a function called
 largest_prime_factor that returns the
largest prime factor of n. Since n is
guaranteed to be composite and greater
than 1, trial division should be enough
here.
The main idea is:
1. remove all factors of 2,
2. then scan odd candidates,
3. divide them out when found,
4. keep track of the largest factor I have
 seen.
I should be careful with powers of 2 and
with the case where a large prime factor
remains at the end.
</think>

def largest_prime_factor(n: int):
    largest = None
    while n % 2 == 0:
        largest = 2
        n //= 2
    factor = 3
    while factor * factor <= n:
        while n % factor == 0:
            largest = factor
            n //= factor
        factor += 2
    if n > 1:
        largest = n
    return largest
```

Figure E: Reasoning examples of 1.58-bit Qwen3-4B models obtained via differentiable ternarization under different calibration schemes. The coding problem involves finding the largest prime factor.

Question

```
def count_upper(s):
    """
    Given a string s, count the number of uppercase vowels in even indices.
    count_upper('aBCdEf') returns 1
    count_upper('abcdefg') returns 0
    count_upper('dBBE') returns 0
    """
```

CoT-agnostic: generic-text (C4)

```
<think>
Okay, I need to implement the function
count_upper(s) that counts the number of
uppercase vowels in even indices.

First, I need to determine which vowels
are considered uppercase.
But in this case, we're looking at
uppercase vowels and even indices.
</think>

...
the the the the the the ...
[truncated repetitive content]
```

CoT-agnostic: domain-specific

```
...
So, the function can be implemented as
follows:

def count_upper(s):
    count = 0
    vowels = {'A', 'E', 'I', 'O', 'U'}
    for i in range(0, len(s), 2):
        if s[i] in vowels:
            count += 1
    return count

The function correctly counts the number
of uppercase vowels in even indices.
```

CoT-aware: stronger-LLM-generated

```
<think>
</think>

The task is simple enough that a direct
loop over indices should work.

<think>
I only need to check even positions and
see whether the character is an uppercase
vowel.
But maybe normalizing the character with .
upper() will make the condition cleaner.
</think>

def count_upper(s):
    vowels = {'A', 'E', 'I', 'O', 'U'}
    count = 0
    for i in range(len(s)):
        if i % 2 == 0 and s[i].upper() in
vowels:
            count += 1
    return count

For example, 'aBCdEf' gives 2.
```

CoT-aware: self-generated

```
<think>
Okay, so I need to write a function called
 count_upper that counts uppercase vowels
appearing at even indices. This one is
short, so the main risk is not the loop
but the predicate: it must count only
characters that are already uppercase
vowels, not lowercase vowels converted to
uppercase.

So the clean structure is:
- iterate through even indices only,
- check membership in {'A', 'E', 'I', 'O',
 'U'},
- increment the counter.
</think>

def count_upper(s):
    count = 0
    vowels = {'A', 'E', 'I', 'O', 'U'}
    for i in range(0, len(s), 2):
        if s[i] in vowels:
            count += 1
    return count
```

Figure F: Reasoning examples of 1.58-bit Qwen3-4B models obtained via differentiable ternarization under different calibration schemes. The coding problem involves counting uppercase vowels at even indices.

Question

```
def count_nums(arr):
    """
    Return the number of elements whose signed digit sum is greater than 0.
    For a negative number, only the first digit is negative:
    e.g., -123 has signed digits -1, 2, and 3.
    """
```

CoT-agnostic: generic-text (C4)

```
So, the function is written as follows:
def count_nums(arr):
    "Write a function count_nums ..."

For each element in the array, we need to
check if it is a negative number.
If it is, then we need to compute the sum
of its digits, but the first digit is
negative.

...
the the the the the ...
[truncated repetitive content]
```

CoT-agnostic: domain-specific

```
I need to write a function called
count_nums that takes an array of integers
 and returns the number of elements where
the sum of their digits is greater than 0.

Let's implement this in code.


def count_nums(arr):
    count = 0
    for num in arr:
        if num < 0:
            sum_digits = sum(abs(num))
        else:
            sum_digits = sum(abs(num))
        if sum_digits > 0:
            count += 1
    return count

Wait, that would make the sample answer 3,
 not 1.
```

CoT-aware: stronger-LLM-generated

```
<think>
Maybe I should convert each number to a
string and handle the negative sign
separately.
</think>

<think>
The first signed digit is negative, so the
 sign only affects the first character
after the minus sign. I think I can keep
the digits as a string and then
reconstruct the sum.
</think>

def count_nums(arr):
    count = 0
    for num in arr:
        s = str(num)
        if s[0] == '-':
            sign = -1
            digits = s[1:]
        else:
            sign = 1
            digits = s
        sum_digits = sum(int(d) for d in
digits)
        if sum_digits > 0:
            count += 1
    return count

Wait, but for -123, the signed digits
should be -1, 2, 3, so this is still not
quite right.
```

CoT-aware: self-generated

```
<think>
Okay, so I need to count how many numbers
have positive signed digit sum. The
annoying part is the negative case: for
-123, the sum is not -(1+2+3), but rather
-1 + 2 + 3.

So I should be careful not to
overgeneralize from ordinary digit sums.

One clean way is:
1. turn the absolute value into digits,
2. compute the normal sum,
3. if the original number was negative,
subtract twice the first digit to flip
only that digit's sign.
</think>

def count_nums(arr):
    count = 0
    for num in arr:
        if num < 0:
            abs_num = abs(num)
            digits = list(map(int, str(
abs_num)))
            first_digit = digits[0]
            sum_digits = sum(digits)
            sum_original = sum_digits - 2
* first_digit
        else:
            digits = list(map(int, str(num
)))
            sum_original = sum(digits)
        if sum_original > 0:
            count += 1
    return count
```

Figure G: Reasoning examples of 1.58-bit Qwen3-4B models obtained via differentiable ternarization under different calibration schemes. The coding problem involves counting numbers with positive signed digit sum.

Question

```
from typing import List, Optional
def longest(strings: List[str]) -> Optional[str]:
    """Return the longest string. Return the first one in case of ties.
    Return None if the input list is empty."""
```

CoT-agnostic: generic-text (C4)

```
</think>
Okay, I need to implement the function
called "longest strings"
that takes a list of strings and returns
the longest one.

The function should return the first one
if there are multiple strings
of the same length. If the list is empty,
it should return None.

...
So. So. So. So. So. ...
[truncated repetitive content]
```

CoT-agnostic: domain-specific

```
I need to implement the function longest(
strings: List[str]) -> Optional[str].

The function should return the longest
string in the list.
If there are multiple strings of the same
maximum length, it should return the first
 one.
If the list is empty, it should return
None.


def longest(strings: List[str]) ->
Optional[str]:
    if not strings:
        return None
    max_length = 0
    result = None
    for string in strings:
        if len(string) > max_length:
            max_length = len(string)
            result = string
    return result

The code works for all the cases. The code
 works for all the cases.
[truncated repetitive content]
```

CoT-aware: stronger-LLM-generated

```
<think>
</think>

<think>
This one is just a left-to-right scan with
 a running best answer.
</think>

from typing import List, Optional

def longest(strings: List[str]) ->
Optional[str]:
    if not strings:
        return None
    max_length = 0
    result = None
    for s in strings:
        if len(s) > max_length:
            max_length = len(s)
            result = s
        elif len(s) == max_length:
            pass
    return result

The function handles the edge case of an
empty list and returns the first longest
string.
```

CoT-aware: self-generated

```
<think>
Okay, so I need to write a function called
 longest that returns the longest string
in a list, with the tie-breaking rule that
 if multiple strings have the same maximum
 length I keep the first one.

This is actually simple, so I should avoid
 overthinking it:
- return None if the list is empty,
- scan from left to right,
- only update the answer when I see a
strictly longer string.

That way ties automatically keep the
earlier string.
</think>

from typing import List, Optional

def longest(strings: List[str]) ->
Optional[str]:
    if not strings:
        return None
    max_length = -1
    result = None
    for s in strings:
        if len(s) > max_length:
            max_length = len(s)
            result = s
    return result
```

Figure H: Reasoning examples of 1.58-bit Qwen3-4B models obtained via differentiable ternarization under different calibration schemes. The coding problem involves finding the longest string in a list.