NGM: A PLUG-AND-PLAY TRAINING-FREE MEMORY MODULE FOR LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent studies have introduced memory-augmented architectures that decouple knowledge storage from neural computation, enabling more direct access to stored representations. However, existing approaches often rely on learned memory embeddings or explicit memory and retrieval components, requiring additional training or increasing architectural complexity. To address these limitations, we propose **N-gram Memory (NGM)**, a training-free, plug-and-play module that repurposes a backbone model's pretrained token embedding space as memory. NGM consists of a **Causal N-Gram Encoder** and a **Cosine-Gated Memory Injector**. The former constructs N-gram representations by directly aggregating pretrained token embeddings, while the latter uses cosine similarity with ReLU gating to adaptively control the amount of retrieved memory injected into contextual representations. Unlike conventional memory-augmented architectures, NGM requires neither additional embedding training nor a separately learned memory bank or external retrieval infrastructure. We evaluate NGM on ten benchmarks across six language-model backbones spanning two model families and multiple scales, including four Qwen3 models (0.6B, 1.7B, 4B, and 8B) and two Llama-3.2-Instruct models (1B and 3B). NGM consistently improves the ten-task average across all six backbones. Notably, it improves **HumanEval by 4.26**, **HumanEval+ by 2.41**, and **GPQA-Diamond by 6.56** percentage points. Furthermore, NGM extends beyond text-only language models to vision-language models, with Qwen3-VL-2B achieving improvements across all five VLM benchmarks.

1 INTRODUCTION

Transformer-based large language models (LLMs) (Vaswani et al., 2017; Zheng et al., 2023; Guo et al., 2025; Jiang et al., 2026; Zhu et al., 2024; Zhao et al., 2026) excel at contextual modeling and compositional reasoning, but not all patterns encountered during language modeling require the same form of computation. Named entities, repeated identifiers, units, terminology, and formulaic phrases often exhibit stable local dependencies that could potentially be recovered through inexpensive lookup rather than repeatedly reconstructed by deep neural computation (Erman, 2000; Constant et al., 2017; Brants et al., 2007; Liu et al., 2024a; Nguyen, 2024). Standard Transformers, however, lack a native lookup primitive for such local lexical and symbolic patterns, and therefore rely on attention and feed-forward computation to reconstruct them at inference time (Wu et al., 2022; Cheng et al., 2026).

Lookup-style memory provides a natural mechanism for separating static pattern reuse from dynamic Transformer computation (Khandelwal et al., 2019; Wu et al., 2022). Recent approaches have explored this direction by introducing explicitly learned memory representations. Engram (Cheng et al., 2026), for example, formulates conditional memory using learned N-gram lookup tables and context-dependent gating, while embedding-scaling methods increase memory capacity through additional token-level or N-gram embedding parameters (Yu et al., 2026; Tseng & De Sa, 2026; Liu et al., 2026; Ding et al., 2026). These results show that local lookup can aid language models, but typically relies on separately learned memory representations, adding parameters, training, and sometimes dedicated storage or retrieval infrastructure.

This observation raises a more fundamental question:

Can a pretrained LLM embedding space serve as local memory without retraining or a separate memory table?

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.

We challenge the assumption that useful memory representations must be separately learned. Our central insight is to reinterpret the pretrained token embedding space not merely as an input representation, but as a *non-parametric memory substrate*. If nearby tokens form stable lexical, symbolic, or phrase-level patterns, their pretrained embeddings may be aggregated into local N-gram representations that provide reusable signals without introducing new memory parameters. Specifically, we construct causal N-gram representations by averaging pretrained token embeddings within local trailing windows. This simple operation converts the backbone's existing embedding space into a lightweight local-memory representation without learning a separate N-gram embedding space.

A key question, however, is whether these embedding-derived N-gram representations are compatible with the contextual representation space of the decoder. Figure 1 provides evidence that they are. Across model depth in Qwen3-8B, aggregated N-gram embeddings exhibit substantially stronger cosine alignment with hidden states than both position-shuffled N-gram controls and random-token controls. At the two default injection layers, the mean cosine similarities are 0.312 and 0.137 for the actual N-gram representations, compared with 0.172 and 0.084 for shuffled controls and 0.014 and 0.008 for random controls, respectively. These results suggest that the pretrained embedding space preserves local N-gram structure that is measurably compatible with contextual representations, motivating a similarity-based mechanism for selectively injecting such memory signals.

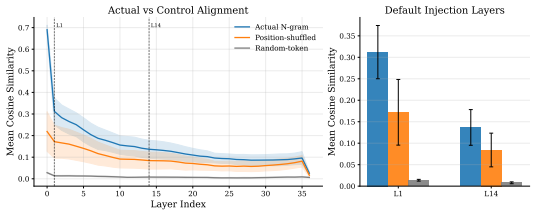


Figure 1: Alignment between hidden states and aggregated N-gram embeddings in the Qwen3-8B model.

Motivated by this observation, we propose **NGM (N-gram Memory)**, a training-free, plug-and-play memory module for frozen decoder-only LLMs, as illustrated in Figure 2. The central design principle of NGM is to reuse the backbone's pretrained representation space rather than introducing a separately learned memory space. NGM consists of two non-parametric components: a **Causal N-gram Encoder** and a **Cosine-Gated Memory Injector**. Given an input sequence, the Causal N-gram Encoder constructs causal multi-scale N-gram representations by aggregating pretrained token embeddings over local trailing windows, capturing local patterns at different granularities without learning additional memory entries. The Cosine-Gated Memory Injector then measures the cosine similarity between these N-gram representations and decoder hidden states, applies a ReLU-filtered gate, and injects the resulting memory signal through a scaled residual connection. Consequently, only positively aligned local-memory signals contribute to the contextual representation, while unrelated or negatively aligned signals are suppressed. This design has two complementary implications. Practically, NGM can be added to a pretrained LLM without updating the backbone, learning a separate embedding space, or using a memory bank or external retrieval system. Conceptually, it offers a simple framework for testing whether pretrained representations contain reusable local-memory structure.

We evaluate NGM on ten benchmarks across six language-model backbones spanning two model families and multiple scales: four Qwen3 models (0.6B, 1.7B, 4B, and 8B) and two Llama-3.2-Instruct models (1B and 3B). Across all six backbones, NGM variants consistently improve the unweighted ten-task average over their corresponding base models. The improvements are particularly pronounced on code generation and complex reasoning, including gains of 4.26 points on HumanEval, 2.41 points on HumanEval+, and 6.56 points on GPQA-Diamond. We further test the same design beyond text-only language models by applying NGM only to the language decoder of Qwen3-VL-2B, leaving the visual encoder unchanged. NGM improves performance across all five evaluated VLM benchmarks.

In summary, our contributions are threefold: (1) we show that a pretrained token embedding space can serve as a non-parametric substrate for local N-gram memory, challenging the assumption that memory representations must be separately learned; (2) we introduce NGM, a training-free and plug-and-play memory module that combines causal multi-scale N-gram aggregation with cosine-gated memory injection; and (3) we provide extensive empirical evaluation across six LLM backbones, ten benchmarks, and a vision-language model, demonstrating consistent improvements across model families and scales and substantial gains in code generation and complex reasoning.

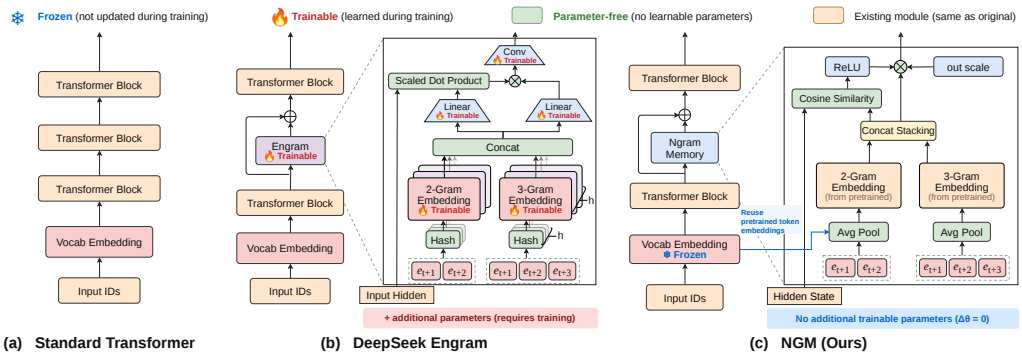


Figure 2: Overview of NGM. **(a)** Standard Transformer without an explicit N-gram memory. **(b)** DeepSeek Engram introduces trainable N-gram embeddings and additional learned modules. **(c)** NGM reuses pretrained vocabulary embeddings to construct N-gram representations through parameter-free pooling and similarity-based retrieval, introducing no additional trainable parameters ($\Delta\theta = 0$).

2 Related Work

Conditional memory and learned lookup representations. Recent work augments language models with learned lookup representations to increase capacity or enable conditional access to stored patterns. SCONE (Yu et al., 2026) introduces an auxiliary Transformer for producing contextualized f-gram embeddings, while L^3 (Tseng & De Sa, 2026), LongCat-Flash-Lite (Liu et al., 2026), and MeKi (Ding et al., 2026) explore token- or N-gram-level memory through learned embeddings, conditional routing, or memory experts. Engram (Cheng et al., 2026) is most closely related to our work. It combines hashed N-gram lookup, context-aware gating, and sparsity allocation to provide conditional memory at intermediate layers. Qwen3.8-Flash-Next (Qiu et al., 2026) similarly uses hashed N-gram embeddings with contextual gating. These methods demonstrate the utility of lookup-based memory, but they introduce a separately learned memory space and require additional training. Several also rely on specialized routing, caching, host-memory prefetching, or offloading mechanisms. As a result, their performance improvements are coupled with increased model capacity and system complexity. In contrast, NGM reuses the backbone's existing token embedding space and constructs causal multi-scale N-gram representations at inference time, without additional training or learned memory parameters.

Retrieval-based memory. A complementary line of work retrieves stored model representations, as in kNN-LMs and Memorizing Transformers (Khandelwal et al., 2019; Wu et al., 2022), or text from external corpora, as in REALM and RETRO (Guu et al., 2020; Borgeaud et al., 2022). These methods provide access to information beyond the current computation but require explicit datastores, indexing, and retrieval procedures. NGM instead operates entirely within the frozen backbone and uses its pretrained embeddings as an internal, non-parametric source of local memory.

3 Methodology

NGM is a training-free, plug-and-play memory module for frozen decoder-only language models. It is based on the following hypothesis:

if the pretrained token embedding space preserves reusable local regularities, then these regularities can be exposed as memory without learning a separate memory representation.

This hypothesis leads to a two-stage design. First, the ***Causal N-gram Encoder*** constructs local-memory candidates by aggregating pretrained token embeddings over causal windows of multiple scales. Second, the ***Cosine-Gated Memory Injector*** uses the current decoder state to measure the compatibility of each candidate and writes only positively aligned signals into the residual stream.

This design separates two roles that are often entangled in memory-augmented architectures: the encoder specifies *what local information is available*, while the injector determines *when that*

information should affect computation. All backbone parameters remain frozen, and NGM introduces no separately learned memory table, external datastore, or trainable projection.

3.1 Problem Setup and Zero-Parameter Memory Reuse

Let $\boldsymbol{X} = \{x_1, \ldots, x_T\}$ denote an input token sequence of length T. Let $\boldsymbol{E} \in \mathbb{R}^{V \times d}$ be the frozen token embedding matrix of the backbone, where V is the vocabulary size and d is the embedding dimension. The embedding of token x_t is $\boldsymbol{e}_t = \boldsymbol{E}[x_t] \in \mathbb{R}^d$.

Unlike conventional memory-augmented models, NGM does not introduce a new parameterized memory space. Instead, it treats local transformations of $\boldsymbol{E}$ as memory candidates and uses the decoder hidden state as a context-dependent compatibility query. The resulting memory operation is therefore zero-parameter with respect to the memory mechanism itself.

3.2 Causal Multi-Scale Memory Construction

Let $\mathcal{N}$ denote a predefined set of window sizes. For each $n \in \mathcal{N}$, we construct a causal local representation using the current token and its preceding $n-1$ tokens. To preserve the sequence length, we prepend $n-1$ zero vectors:

$$\tilde{\boldsymbol{e}}_t = \begin{cases} \boldsymbol{0}, & 2-n \le t \le 0, \\ \boldsymbol{e}_t, & 1 \le t \le T. \end{cases} \tag{1}$$

The causal N-gram representation at position t is

$$\boldsymbol{g}_{t,n} = \frac{1}{n} \sum_{k=0}^{n-1} \tilde{\boldsymbol{e}}_{t-k}. \tag{2}$$

The averaging operation is intentionally parameter-free. It preserves the backbone dimension, introduces no additional memory entries, and exposes local patterns at multiple context scales. More importantly, it defines a common representation space in which memory construction and contextual compatibility can be evaluated without learned alignment layers.

For each position, the multi-scale memory candidates are stacked as

$$\boldsymbol{G}_t = \left[\boldsymbol{g}_{t,n}\right]_{n \in \mathcal{N}} \in \mathbb{R}^{|\mathcal{N}| \times d}. \tag{3}$$

The construction is causal because every $\boldsymbol{g}_{t,n}$ depends only on tokens up to position t. Thus, $\boldsymbol{G}_t$ can be computed during both full-sequence prefill and incremental autoregressive decoding.

3.3 Cosine-Gated Memory Injection

The Causal N-gram Encoder provides candidate local memories, but not every candidate is useful for every context. The Cosine-Gated Memory Injector addresses this selection problem without introducing a learned controller.

Let $\boldsymbol{h}_t^l \in \mathbb{R}^d$ be the hidden state at position t and layer l. We measure the compatibility between the hidden state and each candidate using cosine similarity:

$$s_{t,n} = \cos\left(\boldsymbol{h}_t^l, \boldsymbol{g}_{t,n}\right) = \frac{\left\langle \boldsymbol{h}_t^l, \boldsymbol{g}_{t,n} \right\rangle}{\left\|\boldsymbol{h}_t^l\right\|_2 \left\|\boldsymbol{g}_{t,n}\right\|_2}. \tag{4}$$

The score serves as a geometric compatibility test between the contextual representation and the local-memory candidate. We then apply a ReLU gate:

$$\alpha_{t,n} = \max(0, s_{t,n}). \tag{5}$$

The ReLU is important for the intended memory behavior. Positively aligned candidates can reinforce the current representation, whereas negatively aligned candidates are treated as incompatible and are

prevented from contributing to the update. The memory signal injected at layer l is

$$\boldsymbol{m}_t^l = \sum_{n \in \mathcal{N}} \alpha_{t,n} \boldsymbol{g}_{t,n}. \tag{6}$$

We write this signal into the residual stream using

$$\boldsymbol{h}_t^{l'} = \boldsymbol{h}_t^l + \lambda_l \boldsymbol{m}_t^l, \tag{7}$$

where λ_l is a fixed injection scale for layer l. Equations (4)–(7) define a content-addressed residual write. The pretrained embedding space provides the candidate signal, cosine similarity determines its contextual compatibility, and the residual connection controls how the selected signal enters the decoder computation. No learned key-value projection, retrieval index, or additional memory parameters are required. The complete algorithmic implementation is provided in Algorithm 1.

Algorithm 1 NGM: N-gram Memory (*Causal N-gram Encoder + Cosine-Gated Memory Injector*)

Require: Token IDs $\boldsymbol{X} = \{x_1, \ldots, x_T\}$; hidden states $\boldsymbol{H}^l = \{\boldsymbol{h}_t^l\}_{t=1}^L$; backbone token embedding matrix $\boldsymbol{E} \in \mathbb{R}^{V \times d}$; N-gram sizes $\mathcal{N}$; output scale λ; boolean use_relu.

Ensure: Updated hidden states $\boldsymbol{H}^{l'} = \{\boldsymbol{h}_t^{l'}\}_{t=1}^L$.

% — Causal N-gram Encoder (Eqs. 1–4) —

$\boldsymbol{G} \leftarrow \textsc{CausalNgramEncoder}(\boldsymbol{X}, \boldsymbol{E}, \mathcal{N})_{T-L+1:T}$ ▷ $\boldsymbol{G} \in \mathbb{R}^{L \times |\mathcal{N}| \times d}$, aligned to last L positions

% — Cosine-Gated Memory Injector —

$\hat{\boldsymbol{H}}^l \leftarrow \text{L2Norm}(\boldsymbol{H}^l)$ ▷ Row-wise ℓ_2 normalization; $\hat{\boldsymbol{H}}^l \in \mathbb{R}^{L \times d}$

$\hat{\boldsymbol{G}} \leftarrow \text{L2Norm}(\boldsymbol{G})$ ▷ Normalize along d; $\hat{\boldsymbol{G}} \in \mathbb{R}^{L \times |\mathcal{N}| \times d}$

$S_{t,n} \leftarrow \sum_j \hat{H}_{t,j}^l \, \hat{G}_{t,n,j}$ ▷ Position-wise cosine similarity; $\boldsymbol{S} \in \mathbb{R}^{L \times |\mathcal{N}|}$

if use_relu **then**

  $\boldsymbol{S} \leftarrow \max(\boldsymbol{0}, \boldsymbol{S})$ ▷ Suppress negatively aligned entries

end if

$M_{t,j} \leftarrow \sum_n S_{t,n} \, G_{t,n,j}$ ▷ Gated weighted sum over N-gram scales; $\boldsymbol{M} \in \mathbb{R}^{L \times d}$

$\boldsymbol{H}^{l'} \leftarrow \boldsymbol{H}^l + \lambda \boldsymbol{M}$ ▷ Scaled residual injection

return $\boldsymbol{H}^{l'}$

3.4 Layer Placement and Cached Decoding

We apply the injector after the MLP block in a small set of selected decoder layers. Let $\mathcal{L}_{\text{inj}}$ denote the injection-layer set. The original self-attention and MLP parameters are unchanged; NGM only modifies the layer representation through Eq. (7). The layer IDs and injection scales are fixed hyperparameters, whose default values are reported in Appendix Table 5. Motivated by the layer-sensitivity analysis of Engram (Cheng et al., 2026), we use a sparse placement as a shared practical default.

During prefill, the encoder constructs $\boldsymbol{G}_t$ for all input positions. During autoregressive decoding with a KV cache, the representation at position t depends only on the current token and at most the preceding $\max(\mathcal{N}) - 1$ token embeddings. We therefore maintain a short causal embedding cache and update the memory candidates incrementally.

If the active hidden-state segment contains L positions, we align the corresponding memory candidates by selecting the final L positions:

$$\boldsymbol{G}_{T-L+1:T} = [\boldsymbol{G}_{T-L+1}, \ldots, \boldsymbol{G}_T]. \tag{8}$$

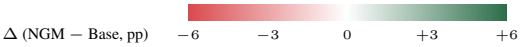
This alignment preserves the causal correspondence between hidden states and memory candidates while avoiding recomputation over the entire prefix.

3.5 Computational and Parameter Complexity

For a sequence of length T, embedding dimension d, and window-size set $\mathcal{N}$, the causal memory construction requires $O(T|\mathcal{N}|d)$ additional computation during prefill. Let $K = |\mathcal{L}_{\text{inj}}|$ be the number of injection layers. The cosine scoring, gating, and multi-scale aggregation add $O(KT|\mathcal{N}|d)$

Table 1: Performance across six backbones and ten benchmarks (%; ↑). For each backbone and benchmark, the baseline and NGM variants were evaluated under the same decoding configuration and run protocol; the table reports the mean and sample standard deviation across three runs for NGM. Δ denotes the difference between the NGM mean and the baseline score, in percentage points. Avg is the unweighted average across the ten benchmarks, computed before rounding.

		Mathematics and code generation					Knowledge, truthfulness, and reasoning					
Model		**GSM8K**	**MATH**	**HE**	**HE+**	**LCB**	**MMLU-R**	**GPQA**	**TQA**	**PopQA**	**BBH**	**Avg**
Qwen3-0.6B	Baseline	62.09	50.20	33.54	29.27	4.19	45.63	25.25	12.61	10.53	31.30	30.46
	+NGM	**62.75**±0.51	**52.40**±1.56	**36.18**±2.53	**31.51**±2.31	**6.59**±0.00	**46.04**±0.29	**26.94**±2.49	**13.91**±0.14	**10.76**±0.03	**31.70**±0.13	**31.88**
	Δ	+0.66	+2.20	+2.64	+2.24	+2.40	+0.41	+1.69	+1.30	+0.23	+0.40	+1.42
Qwen3-1.7B	Baseline	80.97	**73.80**	59.15	57.32	8.98	66.60	26.77	52.02	13.70	54.71	49.40
	+NGM	**83.19**±0.94	73.60±0.53	**63.41**±1.22	**59.73**±0.61	**9.78**±0.35	**67.09**±0.37	**33.33**±1.82	**52.51**±0.21	13.70±0.06	**54.87**±0.36	**51.12**
	Δ	+2.22	-0.20	+4.26	+2.41	+0.80	+0.49	+6.56	+0.49	+0.00	+0.16	+1.72
Qwen3-4B	Baseline	91.58	83.20	78.66	73.78	18.56	79.21	42.42	51.29	15.80	79.47	61.40
	+NGM	**91.81**±0.20	**85.07**±0.58	**81.51**±0.35	**75.81**±0.70	**19.56**±0.69	**79.83**±0.38	**47.47**±1.34	**52.10**±0.49	**15.88**±0.02	**80.03**±0.35	**62.91**
	Δ	+0.23	+1.87	+2.85	+2.03	+1.00	+0.62	+5.05	+0.81	+0.08	+0.56	+1.51
Qwen3-8B	Baseline	93.18	84.00	83.54	82.32	19.76	80.72	42.93	72.95	19.98	82.94	66.23
	+NGM	**93.66**±0.29	**85.00**±0.92	**85.17**±0.93	82.32±0.61	**21.16**±0.69	**80.98**±0.20	**47.14**±2.87	**73.19**±0.13	**20.28**±0.05	**83.25**±0.17	**67.21**
	Δ	+0.48	+1.00	+1.63	+0.00	+1.40	+0.26	+4.21	+0.24	+0.30	+0.31	+0.98
Llama-1B	Baseline	35.71	20.80	25.61	25.61	0.00	35.44	19.19	33.90	11.42	23.16	23.08
	+NGM	**37.15**±0.46	**22.73**±0.46	**29.68**±0.93	**27.03**±1.41	**0.80**±0.35	**36.74**±0.53	**20.88**±2.59	**35.25**±0.43	**11.52**±0.02	**23.61**±0.20	**24.54**
	Δ	+1.44	+1.93	+4.07	+1.42	+0.80	+1.30	+1.69	+1.35	+0.10	+0.45	+1.46
Llama-3B	Baseline	69.98	36.60	48.78	46.34	3.59	62.19	26.26	58.75	17.28	42.11	41.19
	+NGM	**70.38**±0.84	**37.53**±1.10	**50.61**±1.22	**47.36**±0.35	**6.59**±0.60	**62.72**±0.16	**29.63**±1.77	**60.18**±0.43	**17.40**±0.03	**42.68**±0.15	**42.51**
	Δ	+0.40	+0.93	+1.83	+1.02	+3.00	+0.53	+3.37	+1.43	+0.12	+0.57	+1.32



Green: improvement | Red: degradation
Color encodes the difference, not the absolute score.

computation during prefill. With incremental embedding caching, the additional cost per autoregressive decoding step is $O\left(K|\mathcal{N}|d\right)$, which does not grow with the prefix length. The method introduces no trainable parameters. During streaming decoding, the additional state consists only of the recent token embeddings needed to update the causal windows, giving an auxiliary memory cost of $O\left(\max(\mathcal{N})d\right)$.

4 Experiments

4.1 Experimental setup

Setup. We evaluate NGM on six backbones: Qwen3 at 0.6B, 1.7B, 4B, and 8B (Yang et al., 2025), and Llama-3.2-Instruct at 1B and 3B (Grattafiori et al., 2024; Meta, 2024). For each backbone, we compare the original checkpoint with its NGM-augmented counterpart, keeping all backbone weights frozen. For each backbone, the NGM configuration is fixed across tasks; the output scales and insertion layers are reported in Appendix Table 5. We use EvalScope (Team, 2024) for these evaluations. For each benchmark, the baseline and NGM share identical inference settings, including any benchmark-specific overrides. The default settings are a maximum context length of 32,768 tokens, temperature $= 0.7$, top-$p = 0.8$, top-$k = 20$, and presence penalty $= 1.5$.

Benchmarks. We evaluate mathematical reasoning on GSM8K (Cobbe et al., 2021) and MATH500 (Hendrycks et al., 2021); code generation on HumanEval (Chen et al., 2021), HumanEval+ (Liu et al., 2023), and LiveCodeBench v5 (Jain et al., 2025); knowledge and reasoning on MMLU-Redux (Hendrycks et al., 2020; Gema et al., 2025), GPQA-Diamond (Rein et al., 2023), and BBH (Suzgun et al., 2023); and truthfulness and factual recall on TruthfulQA (Lin et al., 2022) and PopQA (Mallen et al., 2023).

4.2 MAIN RESULTS

Table 1 reports the results across six backbones and ten benchmarks. NGM improves the unweighted ten-task average for every model, with gains of 0.98–1.72 percentage points across both the Qwen3 and Llama-3.2-Instruct families. The improvement is not monotonic with model size: Qwen3-1.7B obtains the largest average gain (+1.72), whereas Qwen3-8B gains +0.98. This suggests that NGM acts as a complementary signal to the frozen backbone rather than providing a fixed improvement that scales uniformly with model size.

The most consistent gains occur on code-generation and knowledge-oriented tasks: HumanEval, LiveCodeBench, and GPQA-Diamond improve for all six models, with the largest gain observed on GPQA-Diamond for Qwen3-1.7B (+6.56 percentage points). GSM8K and MMLU-Redux also improve for all six models. HumanEval+ improves for five models and remains unchanged on Qwen3-8B, while TruthfulQA, PopQA, and BBH do not decrease.

Overall, NGM provides gains across model families and scales, but the magnitude of the improvement varies across tasks, with the clearest benefits appearing on code- and knowledge-oriented benchmarks. These results support the compatibility of the local-memory signal across the evaluated models without updating backbone weights, adding trainable parameters, or using external retrieval.

4.3 ABLATION STUDIES

Table 2: Component ablations on Qwen3-1.7B. The reference configuration uses ReLU gating, stack fusion, and the standard tokenizer. Each variant changes one component while keeping the remaining settings fixed.

Configuration	GSM8K	MATH500	HE	HE+	GPQA-D	TQA
Full NGM	**83.19**	73.60	**63.41**	**59.73**	**33.33**	**52.51**
− ReLU gating	83.62	71.60	**63.41**	58.54	30.81	52.39
Alternative fusion (concat)	82.34	72.60	62.80	59.15	27.78	52.14
Compressed tokenization	82.64	**74.20**	60.98	56.71	31.31	52.26

We study the key design choices of the Cosine-Gated Memory Injector and the Causal N-gram Encoder on Qwen3-1.7B. The reference configuration uses $\mathcal{N} = \{2, 3\}$, ReLU gating, stack fusion, and the standard tokenizer. All variants use the same output scales and insertion layers (see Appendix Table 5), while the backbone weights remain frozen. HE and HE+ denote the pass@1 scores of HumanEval and HumanEval+, respectively; GPQA-D denotes GPQA-Diamond, and TQA denotes TruthfulQA. Table 2 reports three component variants. The full configuration achieves the best or tied-best result on four of the six metrics, indicating a favorable balance across tasks.

ReLU gating. For the Cosine-Gated Memory Injector, removing ReLU improves GSM8K by 0.43 percentage points, but decreases MATH500 and GPQA-D by 2.00 and 2.52 points, respectively. This suggests that ReLU filters candidates whose directions oppose the current decoder state, allowing the injector to focus on positively aligned memory signals, although its effect is not uniformly beneficial across tasks.

Alternative fusion. For the Causal N-gram Encoder, stack fusion outperforms concat fusion on all six metrics, with the largest difference on GPQA-D. Stack fusion computes separate gates for different N-gram scales before aggregating the candidates, whereas concat fusion uses a shared gate and mean aggregation.

Compressed tokenization. We also introduce the compressed tokenization scheme from Engram (Cheng et al., 2026) into the Causal N-gram Encoder. The compressed token IDs are used to index the original embedding matrix, while the backbone tokenizer and sequence length remain unchanged. Compressed tokenization improves MATH500 by 0.60 points but performs worse than the standard tokenizer on the other five metrics, resulting in a 1.28-point lower average across the six tasks. This indicates that the Causal N-gram Encoder depends not only on the surface form of tokens, but also on the geometric structure preserved by the original token embeddings. Merging token IDs changes the local candidate representations and consequently affects their alignment with the decoder state.

N-gram window sizes. Finally, we examine the window sizes used by the Causal N-gram Encoder. Table 3 shows that $\mathcal{N} = \{2, 3\}$ achieves the highest scores on GSM8K, MATH500, and GPQA-D, and ties for the best HumanEval score. Single-scale configurations perform better on some individual metrics, such as HumanEval+ or TruthfulQA, but adding a fourth scale does not provide a consistent benefit. We therefore use $\mathcal{N} = \{2, 3\}$ as the default multi-scale configuration for capturing complementary local patterns.

Table 3: Effect of N-gram window sizes on Qwen3-1.7B. All other components follow the reference configuration.

$\mathcal{N}$	GSM8K	MATH500	HE	HE+	GPQA-D	TQA
$\{2\}$	82.64	72.60	61.59	**60.37**	30.30	**52.63**
$\{3\}$	82.64	72.80	62.80	59.15	31.82	52.39
$\{4\}$	83.09	73.40	**63.41**	58.54	30.81	52.14
$\{2, 3, 4\}$	82.87	72.40	62.80	59.15	31.31	52.51
$\{2, 3\}$	**83.19**	**73.60**	**63.41**	59.73	**33.33**	52.51

4.4 Extension to multimodal models

Table 4: Preliminary multimodal results on Qwen3-VL-2B-Instruct. MMBench (Liu et al., 2024b), MMStar (Chen et al., 2024), OCRBench (Liu et al., 2024c) (scored out of 1000), TQA = TruthfulQA (Lin et al., 2022), and MMLU-Redux (Gema et al., 2025). NGM is applied only to the language decoder; the visual encoder is unchanged.

Model	MMBench	MMStar	OCRBench	TQA	MMLU-R
Qwen3-VL-2B	76.32	54.67	826	52.26	67.79
+NGM	**76.63**	**56.20**	**829**	**52.63**	**67.82**

To test whether NGM transfers beyond text-only LLMs, we apply NGM to Qwen3-VL-2B-Instruct (Bai et al., 2025), leaving the visual encoder and vision-language fusion modules unchanged. The N-gram operates exclusively on text token embeddings; vision tokens are excluded from the sliding-window pooling so that the local memory signal remains purely linguistic. Using VLMEvalKit (Duan et al., 2024) under identical decoding settings, NGM improves or matches the baseline on all five reported benchmarks, with the largest gain on MMStar (+1.53; Table 4). The retained TruthfulQA point estimates use a legacy evaluator whose metric has not been verified as MC2; they are not directly comparable with the generative TruthfulQA results in the main and ablation tables. This single-scale result suggests that the same training-free local-memory mechanism can transfer to multimodal models without architectural changes, but we leave comprehensive multimodal evaluation to future work.

4.5 Mechanistic analysis

We examine two mechanistic questions: whether the cosine gate reflects meaningful aligned structure rather than a generic embedding prior, and whether the resulting interactions are local in position.

Interactions are predominantly local. We next examine the full cross-position matrix $\cos(\boldsymbol{h}_i, \boldsymbol{g}_j)$ on the same model (Figure 3). Across representative code, math, and knowledge samples, the diagonal mean consistently exceeds the off-diagonal mean, with diagonal/off-diagonal ratios of 1.27×–2.42× at the injected layers. The pattern is strongest for the knowledge sample and remains clear for code, indicating that the most useful memory signal is concentrated near the aligned position rather than uniformly distributed across the sequence.

Implication for the default gate. Together, these results support the default design. Since alignment remains well above shuffled and random controls, raw cosine similarity is informative without a learned projection. Since the interaction pattern is diagonal-dominant, the token-wise gate captures most of the useful signal while preserving linear-time cost. This interpretation is consistent with the strongest gains in Table 1, especially on tasks where short-range pattern stability matters. Combined with our intentionally restrictive evaluation setting—fixed backbone weights, no additional trainable parameters, and no external knowledge—this also yields a relatively controlled comparison in which the observed gains can be attributed more directly to the core NGM mechanism.

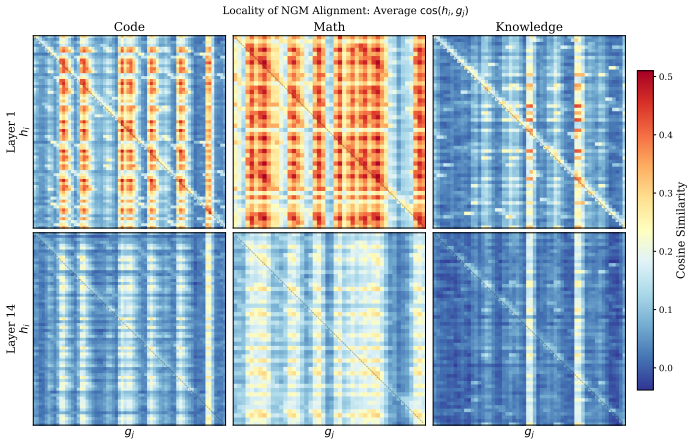


Figure 3: Cross-position locality of NGM interactions in the default Qwen3-8B-NGM model. Heatmaps show the average cross-position cosine matrix $\cos(\boldsymbol{h}_i, \boldsymbol{g}_j)$ at the two default injection layers for representative code, math, and knowledge samples. The diagonal structure dominates, indicating that useful memory interactions are predominantly local.

4.6 Wall-clock overhead

We measure prefill and decode latency for Qwen3-8B with and without NGM on a single RTX 5090 (batch size 1, `bfloat16`, prompt lengths 128–2048, 20 runs; Figure 4). Overhead is reported as the relative latency increase over the original Qwen3-8B baseline, i.e., $(\text{latency}_{NGM} - \text{latency}_{\text{base}})/\text{latency}_{\text{base}}$.

For 256–1024-token prompts, prefill overhead is 3.4–7.3% and decode overhead 1.9–2.3%; at 2048 tokens the figures rise to 16.0% and 9.9%, respectively. These numbers reflect the released code path, which recomputes N-gram features over the full prefix; a streaming cache would reduce the cost to $O(|\mathcal{N}|d)$ per step.

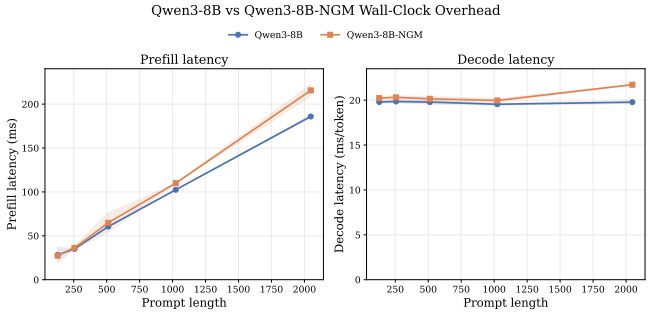


Figure 4: Prefill and per-token decode latency for Qwen3-8B vs. Qwen3-8B-NGM on a single RTX 5090 (mean $\pm$ std over 20 runs). The gap widens at 2048 tokens because the current implementation recomputes N-gram features over the full prefix.

5 Conclusion

We presented NGM, a training-free and plug-and-play memory module built on the observation that pretrained token embeddings can serve as a non-parametric substrate for local memory. Rather than learning a separate memory representation, NGM directly constructs causal multi-scale N-gram memories from the backbone embedding space and injects them through a non-parametric cosine gate, without modifying backbone parameters. Across multiple Qwen3 and Llama backbones, ten benchmarks, and a vision-language model, NGM consistently improves performance across model families and scales, with notable gains in code generation and complex reasoning. These results suggest that useful local lexical and compositional structure is already encoded in pretrained representation spaces and can be directly reused as memory. More broadly, our findings motivate treating pretrained representations not only as inputs to computation, but also as reusable computational resources for augmenting language models without additional training.

AI USE STATEMENT

Generative AI tools were used solely to assist with language editing and wording refinement during manuscript preparation. The authors reviewed the AI-assisted text and take full responsibility for the final content of this work.

ETHICS STATEMENT

This work studies inference-time augmentation of pretrained language models using existing model checkpoints and evaluation benchmarks. The resulting models may retain biases, factual errors, or other undesirable behaviors of the underlying models. Our evaluation focuses on task performance and does not establish improvements in safety or fairness. The reported benchmark gains should therefore not be interpreted as guarantees of reliability or suitability for sensitive applications. Downstream use should consider these limitations and respect the licenses and usage restrictions of the underlying models and datasets.

REPRODUCIBILITY STATEMENT

We describe the Causal N-gram Encoder and Cosine-Gated Memory Injector in Section 3, with the overall procedure summarized in Algorithm A. The evaluated backbones, benchmarks, and inference settings are described in Section 4. Appendix A provides a simplified PyTorch implementation, and Table 5 reports the output scales and insertion layers for each backbone. The hardware configuration and measurement protocol for the latency evaluation are specified in Section 4.6.

REFERENCES

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pp. 2206–2240. PMLR, 2022.

Thorsten Brants, Ashok Popat, Peng Xu, Franz Josef Och, and Jeffrey Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pp. 858–867, 2007.

Lin Chen, Jinsong Li, Xiaoyi Dong, Pan Zhang, Yuhang Zang, Zehui Chen, Haodong Duan, Jiaqi Wang, Yu Qiao, Dahua Lin, et al. Are we on the right way for evaluating large vision-language models? *Advances in Neural Information Processing Systems*, 37:27056–27087, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xin Cheng, Wangding Zeng, Damai Dai, Qinyu Chen, Bingxuan Wang, Zhenda Xie, Kezhao Huang, Xingkai Yu, Zhewen Hao, Han Zhang, et al. Conditional memory via scalable lookup: A new axis of sparsity for large language models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4968–4990, 2026.

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.

Mathieu Constant, Gülşen Eryiğit, Johanna Monti, Lonneke Van Der Plas, Carlos Ramisch, Michael Rosner, and Amalia Todirascu. Multiword expression processing: A survey. *Computational Linguistics*, 43(4):837–892, 2017.

Ning Ding, Fangcheng Liu, Kyungrae Kim, Linji Hao, Kyeng-Hun Lee, Hyeonmok Ko, and Yehui Tang. Meki: Memory-based expert knowledge injection for efficient llm scaling. *arXiv preprint arXiv:2602.03359*, 2026. URL https://arxiv.org/abs/2602.03359.

Haodong Duan, Junming Yang, Yuxuan Qiao, Xinyu Fang, Lin Chen, Yuan Liu, Xiaoyi Dong, Yuhang Zang, Pan Zhang, Jiaqi Wang, et al. Vlmevalkit: An open-source toolkit for evaluating large multi-modality models. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pp. 11198–11201, 2024.

Britt Erman. The idiom principle and the open choice principle. *Text-Interdisciplinary Journal for the Study of Discourse*, 2000.

Aryo Pradipta Gema, Joshua Ong Jun Leang, Giwon Hong, Alessio Devoto, Alberto Carlo Maria Mancino, Rohit Saxena, Xuanli He, Yu Zhao, Xiaotang Du, Mohammad Reza Ghasemi Madani, et al. Are we done with mmlu? In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 5069–5096, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Yanan Guo, Wenhui Dong, Jun Song, Shiding Zhu, Xuan Zhang, Hanqing Yang, Yingbo Wang, Yang Du, Xianing Chen, and Bo Zheng. Fila-video: Spatio-temporal compression for fine-grained long video understanding. *arXiv preprint arXiv:2504.20384*, 2025.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pp. 3929–3938. PMLR, 2020.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

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.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Changjiang Jiang, Wenhui Dong, Zhonghao Zhang, Fengchang Yu, Wei Peng, Xinbin Yuan, Yifei Bi, Ming Zhao, Zian Zhou, Chenyang Si, et al. Ivy-fake: A unified explainable framework and benchmark for image and video aigc detection. In *Proceedings of the 2026 International Conference on Multimedia Retrieval*, pp. 2438–2447, 2026.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172*, 2019.

Stephanie Lin, Jacob Hilton, and Owain Evans. Truthfulqa: Measuring how models mimic human falsehoods. In *Proceedings of the 60th annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 3214–3252, 2022.

Hong Liu, Jiaqi Zhang, Chao Wang, Xing Hu, Linkun Lyu, Jiaqi Sun, Xurui Yang, Bo Wang, Fengcun Li, Yulei Qian, Lingtong Si, Yerui Sun, Rumei Li, Peng Pei, Yuchen Xie, and Xunliang Cai. Scaling embeddings outperforms scaling experts in language models. *ArXiv*, abs/2601.21204, 2026. URL https://api.semanticscholar.org/CorpusID:285140484.

Jiacheng Liu, Sewon Min, Luke Zettlemoyer, Yejin Choi, and Hannaneh Hajishirzi. Infini-gram: Scaling unbounded n-gram language models to a trillion tokens. *arXiv preprint arXiv:2401.17377*, 2024a.

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 *Advances in Neural Information Processing Systems*, volume 36, pp. 21558–21572, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract.html.

Yuan Liu, Haodong Duan, Yuanhan Zhang, Bo Li, Songyang Zhang, Wangbo Zhao, Yike Yuan, Jiaqi Wang, Conghui He, Ziwei Liu, et al. Mmbench: Is your multi-modal model an all-around player? In *European conference on computer vision*, pp. 216–233. Springer, 2024b.

Yuliang Liu, Zhang Li, Mingxin Huang, Biao Yang, Wenwen Yu, Chunyuan Li, Xu-Cheng Yin, Cheng-Lin Liu, Lianwen Jin, and Xiang Bai. Ocrbench: on the hidden mystery of ocr in large multimodal models. *Science China Information Sciences*, 67(12):220102, 2024c.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9802–9822. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.acl-long.546. URL https://aclanthology.org/2023.acl-long.546/.

Meta. Llama 3.2 model card. https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/MODEL_CARD.md, 2024.

Timothy Nguyen. Understanding transformers via n-gram statistics. *Advances in neural information processing systems*, 37:98049–98082, 2024.

Zihan Qiu, Zekun Wang, Xiao Li, Yanpeng Li, Yang Xu, Yixuan Wang, Huaqing Zhang, Rui Men, Bochao Mao, Chengruidong Zhang, et al. On the design of Qwen3.8-Next architecture: Evaluation, efficiency, and training stability. *arXiv preprint arXiv:2608.30320*, 2026.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, and Jason Wei. Challenging BIG-Bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 13003–13051. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-acl.824. URL https://aclanthology.org/2023.findings-acl.824/.

ModelScope Team. EvalScope: Evaluation framework for large models, 2024. URL https://github.com/modelscope/evalscope.

Albert Tseng and Christopher De Sa. L^3: Large lookup layers. *arXiv preprint arXiv:2601.21461*, 2026.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Yuhuai Wu, Markus N Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. *arXiv preprint arXiv:2203.08913*, 2022.

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.

Da Yu, Edith Cohen, Badih Ghazi, Yangsibo Huang, Pritish Kamath, Ravi Kumar, Daogao Liu, and Chiyuan Zhang. Scaling embedding layers in language models. *Advances in Neural Information Processing Systems*, 38:31969–31995, 2026.

Ming Zhao, Wenhui Dong, Yang Zhang, Zhonghao Zhang, Zian Zhou, YUNZHI GUAN, Liukun Xu, Wei Peng, Zhaoyang Gong, Zhicheng Zhang, et al. Spinebench: a clinically salient, level-aware benchmark powered by the spinemed-450k corpus. In *International Conference on Learning Representations*, volume 2026, pp. 15028–15061, 2026.

Dehua Zheng, Wenhui Dong, Hailin Hu, Xinghao Chen, and Yunhe Wang. Less is more: Focus attention for efficient detr. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6651–6660. IEEE, 2023.

Shiding Zhu, Wenhui Dong, Jun Song, Yingbo Wang, Yanan Guo, and Bo Zheng. Fila: Fine-grained vision language models. *arXiv preprint arXiv:2412.08378*, 2024.

A NGM IMPLEMENTATION

Listing 1 gives a simplified PyTorch implementation of NGM.

```
def ngm_forward(hidden_states, input_ids, embed_matrix,
                ngram_sizes, output_scale, use_relu):
    """
    hidden_states: [B, L, D]   (layer-l hidden states)
    input_ids:     [B, T]      (full input token ids)
    embed_matrix:  Embedding    (token embeddings from backbone model)
    """
    B, L, D = hidden_states.shape
    T = input_ids.shape[1]

    # Step 1: causal n-gram construction
    token_emb = embed_matrix(input_ids)          # [B, T, D]
    ngram_list = []
    for n in ngram_sizes:
        padded = F.pad(token_emb, (0, 0, n-1, 0))
        pooled = F.avg_pool1d(
            padded.transpose(1, 2), kernel_size=n, stride=1
        ).transpose(1, 2)                        # [B, T, D]
        ngram_list.append(pooled)
    ngram_emb = torch.stack(ngram_list, dim=2)   # [B, T, N, D]

    # Step 2: KV-cache alignment
    ngram_emb = ngram_emb[:, -L:, :, :]          # [B, L, N, D]

    # Step 3: cosine gating
    h_norm = F.normalize(hidden_states, dim=-1)  # [B, L, D]
    g_norm = F.normalize(ngram_emb, dim=-1)      # [B, L, N, D]
    sim = torch.einsum('bld,blnd->bln', h_norm, g_norm)
    if use_relu:
        sim = F.relu(sim)

    # Step 4: residual injection
    out = torch.einsum('bln,blnd->bld', sim, ngram_emb)
    return hidden_states + output_scale * out
```

Listing 1: NGM core implementation (simplified).

A.1 MODEL-SPECIFIC NGM SETTINGS

The main evaluation uses $\mathcal{N} = \{2, 3\}$, ReLU gating, and stack fusion. The output scale is fixed at 0.1. Every scale remains frozen during evaluation. Table 5 reports inserted decoder layers in 1-based numbering; the implementation uses zero-based indices $\{1, 14\}$.

Table 5: NGM settings for the six backbones in the main evaluation.

Model	Backbone layers	Inserted decoder layers	Output scale
Qwen3-0.6B-NGM	28	2, 15	0.1
Qwen3-1.7B-NGM	28	2, 15	0.1
Qwen3-4B-NGM	36	2, 15	0.1
Qwen3-8B-NGM	36	2, 15	0.1
Llama-1B-NGM	16	2, 15	0.1
Llama-3B-NGM	28	2, 15	0.1

These layer placements are heuristic. Engram's layer-sensitivity analysis (Cheng et al., 2026) motivates balancing early access to local features with enough contextualization for gating. We use the 2nd and 15th decoder layers as a shared practical default across the six backbones; their relative depths differ, especially for 16-layer Llama-1B, so the shared indices are not equivalent relative positions or universally optimal placements.

B CASE STUDIES: NGM VS. BASE MODEL ON GSM8K

We present three representative examples from the GSM8K benchmark where Qwen3-8B-NGM produces the correct answer while the base Qwen3-8B model fails. Each case highlights a different failure mode that NGM helps mitigate.

Case 1: Arithmetic error correction

Question. In a candy machine, there are 22 more than four times the number of pink gumballs as there are blue gumballs. If there are 12 blue gumballs, how many pink ones are there? **Gold answer:** 70

Base model (Qwen3-8B) ✗

The number of pink gumballs is 22 more than four times the number of blue gumballs.
Given that there are 12 blue gumballs, we calculate:
$4 \times 12 = \mathbf{28}$ (arithmetic error)
$28 + 22 = 50$
`ANSWER: 50`

Qwen3-8B-NGM ✓

The number of pink gumballs is 22 more than four times the number of blue gumballs.
There are 12 blue gumballs.
$4 \times 12 = \mathbf{48}$ (correct)
$48 + 22 = 70$
`ANSWER: 70`

Analysis. The base model produces an incorrect intermediate product ($4 \times 12 = 28$), causing the final answer to be wrong. NGM maintains correct arithmetic in the short-range computation chain.

Case 2: Semantic misinterpretation

Question. Finn watches 11 tadpoles swimming in the pond. Suddenly he sees 6 of them come out of hiding from under a lily pad, then he sees 2 of them hide under a rock. How many tadpoles can Finn see in the pond now? **Gold answer:** 15

Base model (Qwen3-8B) ✗

Finn initially sees 11 tadpoles.
Then, 6 come out of hiding from under a lily pad.
After that, 2 hide under a rock.
Ignores the 6 newcomers:
$11 - 2 = 9$
`ANSWER: 9`

Qwen3-8B-NGM ✓

Finn initially watches 11 tadpoles.
Then, 6 come out of hiding $\rightarrow$ visible count increases by 6.
Next, 2 hide under a rock $\rightarrow$ visible count decreases by 2.
$11 + 6 - 2 = 15$
`ANSWER: 15`

Analysis. The base model misinterprets "come out of hiding" and ignores the additional 6 tadpoles, computing only $11 - 2$. NGM correctly accumulates all quantity changes mentioned in the prompt.

Case 3: Reading comprehension error

Question. Three friends spent \$20.25 on 3 tickets to the fair. They also spent \$4.50 less on food than on the tickets. They also went on 2 different rides which cost \$33 for each ride. If they agreed to split all the costs evenly, how much did each of them pay? **Gold answer:** 34

Base model (Qwen3-8B) ✗

Misreads \$20.25 as per-ticket price:
Tickets: $3 \times \$20.25 = \$60.75$
Food: $\$60.75 - \$4.50 = \$56.25$
Rides: $2 \times \$33 = \$66$
Total: $\$60.75 + \$56.25 + \$66 = \$183$
Each: $\$183 \div 3 = \$61$
`ANSWER: 61`

Qwen3-8B-NGM ✓

Tickets: \$20.25 (total for 3)
Food: $\$20.25 - \$4.50 = \$15.75$
Rides: $2 \times \$33 = \$66$
Total: $\$20.25 + \$15.75 + \$66 = \$102$
Each: $\$102 \div 3 = \$34$
`ANSWER: 34`

Analysis. The base model misreads "spent \$20.25 on 3 tickets" as \$20.25 *per ticket*, inflating all downstream totals. NGM correctly interprets \$20.25 as the total ticket expenditure, preserving local textual coherence across the reasoning chain.