Conditional Memory via Scalable Lookup: A New Axis of Sparsity for Large Language Models

@Deepseek.com

Junran. Yang[1]

[1]School of AI
Tianjin University

Week Report, Feb.3rd 2026

Table of Contents

Table of Contents

What is knowledge

For many years, there have been 2 ways to represent "knowledge": explicit ones such as embedding models and implicit ones, such as Transformers.

- **Embedding model**: A de-facto database that stores vectors which represent "knowledge".
- **Transformer**: A parametric model that implicitly encodes "knowledge" within its trained weights.

Much practice(e.g., face recognition, RAG) has demonstrated that explicit knowledge representation is critical for storing knowledge in a sound manner.

What Engram offers us

- An efficient framework combining Knowledge Retrieval and Reasoning
- a systematic-level design example for algorithm design
- An end-to-end framework that naturally supports incremental learning avoiding catastrophic forgetting

Table of Contents

Balance Between Retrieval and Computation

What they found:

- Standard Transformers lack **a native knowledge lookup primitive**, current LLMs are forced to simulate retrieval through computation.
- sparse computation $\rightarrow$ MoE architecture, they need a sparse lookup operations to retrieve static embeddings for fixed knowledge
- More computational resources could be allocated to reasoning if the network makes full use of lookup operations.
- lookup operations (memory-load) could be hidden since Streaming Multiprocessors(SMs) cost time on computation.

Engram relieves the backbone from reconstructing static knowledge in early layers, thereby increasing the effective depth available for complex reasoning. $\rightarrow$ They prove it in the experiments.

Table of Contents

N-gram

The equation is critical for understanding how the system work. So lets dive into it

What is $N-gram$

given a sequence of tokens: $s = (x'_1, \ldots, x'_t)$ N-Gram is the N-suffix tokens on the tokens which can be formulated as $g_{t,n} = (x'_{t-n+1}, \ldots, x'_t)$, t is the last token's index, n we call the **ORDER**.

N-gram

The equation is critical for understanding how the system work. So lets dive into it

What is $N-gram$

given a sequence of tokens: $s = (x'_1, \ldots, x'_t)$ N-Gram is the N-suffix tokens on the tokens which can be formulated as $g_{t,n} = (x'_{t-n+1}, \ldots, x'_t)$, t is the last token's index, n we call the **ORDER**.

How do we use $N-gram$

Core question now can be parsed as:
$\rightarrow$ how to connect a vector to the Embedding table
$\rightarrow$ how to connect a vector with an index(integer) of the Embedding table
Their answer is: a **HASH** function.

Hashing

Recall that a $n-gram$ writes: $g_{t,n} = (x'_{t-n+1}, \ldots, x'_t)$. We formulate a function as $\varphi_{n,k}(g_{t,n})$, where k means the k^{th} hash function for a $n-gram$.

Hashing

Recall that a $n-gram$ writes: $g_{t,n} = (x'_{t-n+1}, \ldots, x'_t)$. We formulate a function as $\varphi_{n,k}(g_{t,n})$, where k means the k^{th} hash function for a $n-gram$.

a hash function project the vector into a scalar: $z_{t,n,k} \triangleq \varphi_{n,k}(g_{t,n})$, and we look up a embedding in a Embedding table $\mathbf{E}_{n,k}$: $\mathbf{e}_{t,n,k} = \mathbf{E}_{n,k}[z_{t,n,k}]$

Hashing

Recall that a $n-gram$ writes: $g_{t,n} = (x'_{t-n+1}, \ldots, x'_t)$. We formulate a function as $\varphi_{n,k}(g_{t,n})$, where k means the k^{th} hash function for a $n-gram$.

a hash function project the vector into a scalar: $z_{t,n,k} \triangleq \varphi_{n,k}(g_{t,n})$, and we look up a embedding in a Embedding table $\mathbf{E}_{n,k}$: $\mathbf{e}_{t,n,k} = \mathbf{E}_{n,k}[z_{t,n,k}]$

the final memory embedding is **concatenated** by all retrieved embeddings:

Final embedding

$$\mathbf{e}_t \triangleq \Big\Vert_{n=2}^{N} \Big\Vert_{k=1}^{K} \mathbf{e}_{t,n,k}, \quad z_{t,n,k} \triangleq \varphi_{n,k}(g_{t,n}).$$

Fusion

After we retrieve the embedding(knowledge), our task is to fuse it with the hidden latent h_t passed from reasoning. The authors' design-choice here is using a cross-attention like mechanism.

$$\mathbf{k}_t = \mathbf{W}_K \mathbf{e}_t, \quad \mathbf{v}_t = \mathbf{W}_V \mathbf{e}_t$$

and the attention is represented as:

$$\alpha_t = \sigma\left(\frac{\text{RMSNorm}(\mathbf{h}_t)^\top \text{RMSNorm}(\mathbf{k}_t)}{\sqrt{d}}\right), \quad \tilde{\mathbf{v}}_t = \alpha_t \cdot \mathbf{v}_t.$$

so as to suppress noise($a_t = 0$). Notice, v_t is just a vector $\in \mathbb{R}^d$, the whole sequnce is represented as $\tilde{\mathbf{V}} \in \mathbb{R}^{T \times d}$
Finally, they use a 1-D convolution to filter out the noise again:

$$\mathbf{Y} = \text{SiLU}(\text{Conv1D}(\text{RMSNorm}(\tilde{\mathbf{V}}))) + \tilde{\mathbf{V}}, \quad \mathbf{H}^{(\ell)} \leftarrow \mathbf{H}^{(\ell)} + \mathbf{Y}$$

A pause here

Until now, what have we done?

A pause here

Until now, what have we done?

We merged the knowledge retrieve from a embedding table, and use a cross-attention like mechanism to fuse it into the hidden state.

Data Flow

x'_t(input tokens) $\rightarrow$ $z_{t,n,k}$(embedding table index) $\rightarrow$ e_t(concatenated embedding) $\rightarrow$ H(hidden state)

Integrate with mHC

A revision:

$$\alpha_t = \sigma\left(\frac{\text{RMSNorm}(\mathbf{h}_t)^\top \text{RMSNorm}(\mathbf{k}_t)}{\sqrt{d}}\right), \quad \tilde{\mathbf{v}}_t = \alpha_t \cdot \mathbf{v}_t.$$

We notice that compared a standard Transformer Attention, it **lacks multi-head mechanism**, which limits the representation ability.

Integrate with mHC

A revision:

$$\alpha_t = \sigma\left(\frac{\text{RMSNorm}(\mathbf{h}_t)^\top \text{RMSNorm}(\mathbf{k}_t)}{\sqrt{d}}\right), \quad \tilde{\mathbf{v}}_t = \alpha_t \cdot \mathbf{v}_t.$$

We notice that compared a standard Transformer Attention, it **lacks multi-head mechanism**, which limits the representation ability.

Instead, they use a clever way to solve this problem.

- M distinct K projection $\{\mathbf{W}_\mathbf{K}^{(\mathbf{m})}\}_{m=1}^{M}$
- A value projection matrix $\mathbf{W}_V$ shared across M branches
- This design allows the linear projections (one $\mathbf{M_V}$ and M distinct $\mathbf{W}_K^{(m)}$) to be fused into a single dense FP8 matrix multiplication, maximizing the compute utilization of modern GPUs.

Integrate with mHC

the Final Formulation

$$\alpha_t^{(m)} = \sigma\left(\frac{\text{RMSNorm}(\mathbf{h}_t^{(m)})^\top \text{RMSNorm}(\mathbf{W}_K^{(m)}\mathbf{e}_t)}{\sqrt{d}}\right), \mathbf{u}_t^{(m)} = \alpha_t^{(m)} \cdot (\mathbf{W}_V \mathbf{e}_t).$$

Pipeline

The Engram module is placed at specific layers, leveraging the computation of preceding layers to prevent GPU stalls

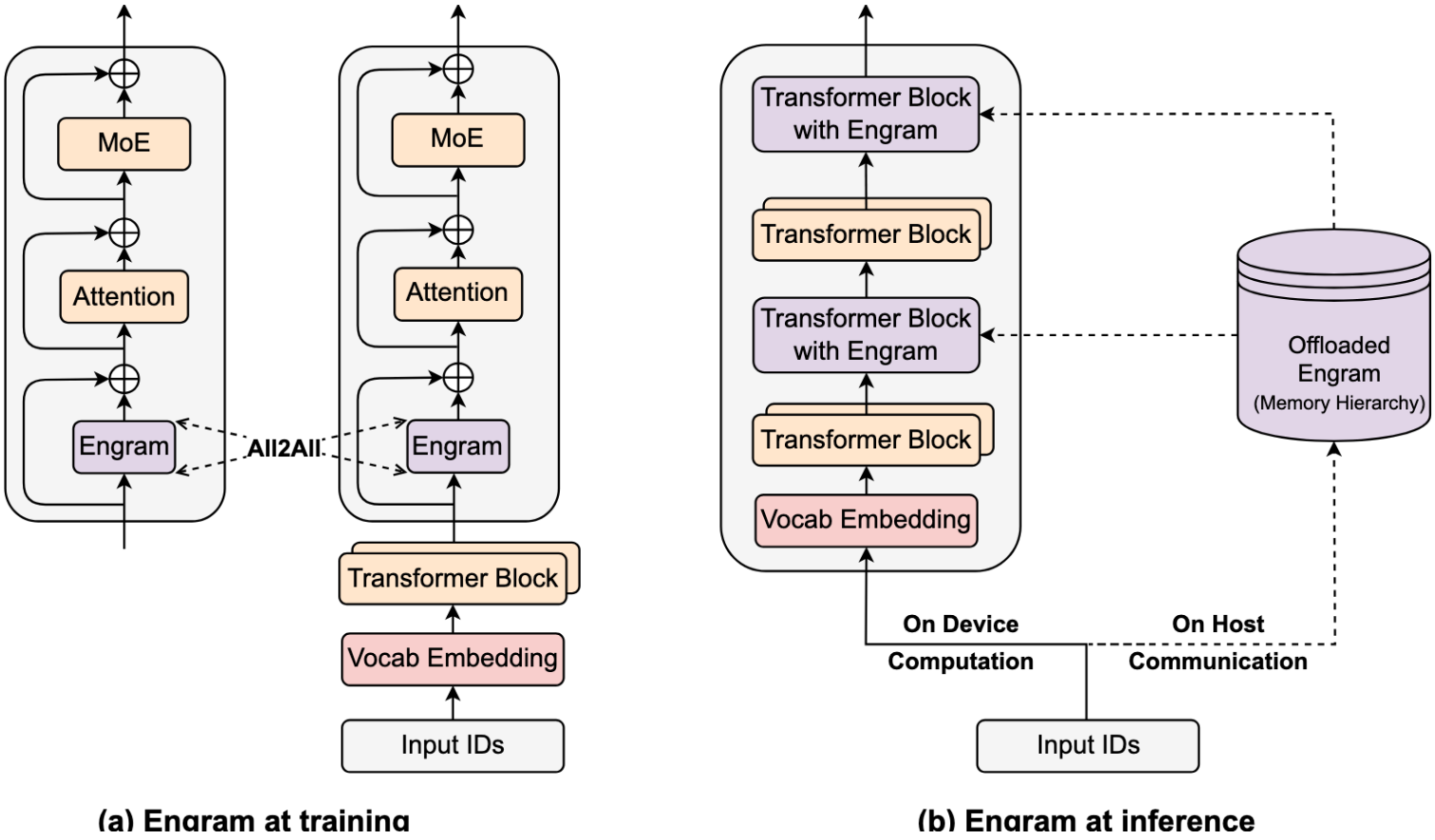

Figure: The embedding table is offloaded to host memory in inference

Table of Contents

Symbol Table

The symbols used in the experiment

P_{tot}: total trainable parameters
P_{act}: activated parameters per token
$P_{sparse} \triangleq P_{tot} - P_{act}$, the budget allocated for MoE expert number and Engram

Allocation ratio $\rho \in [0, 1]$:

$$P_{\text{MoE}}^{(\text{sparse})} = \rho P_{\text{sparse}}, \qquad P_{\text{Engram}} = (1 - \rho) P_{\text{sparse}}.$$

Scaling

Left: Fix the P_{sparse} parameters number ($P_{tot} = 5.7B/9.9B, \quad P_{act} = 568M/993M$), there is a U-shape relationship between validation loss and the allocation ratio ρ

Right: Fix the parameter size in MoE backbone $P_{act} = 568M, \quad P_{tot} = 3B$, increase Engram table.

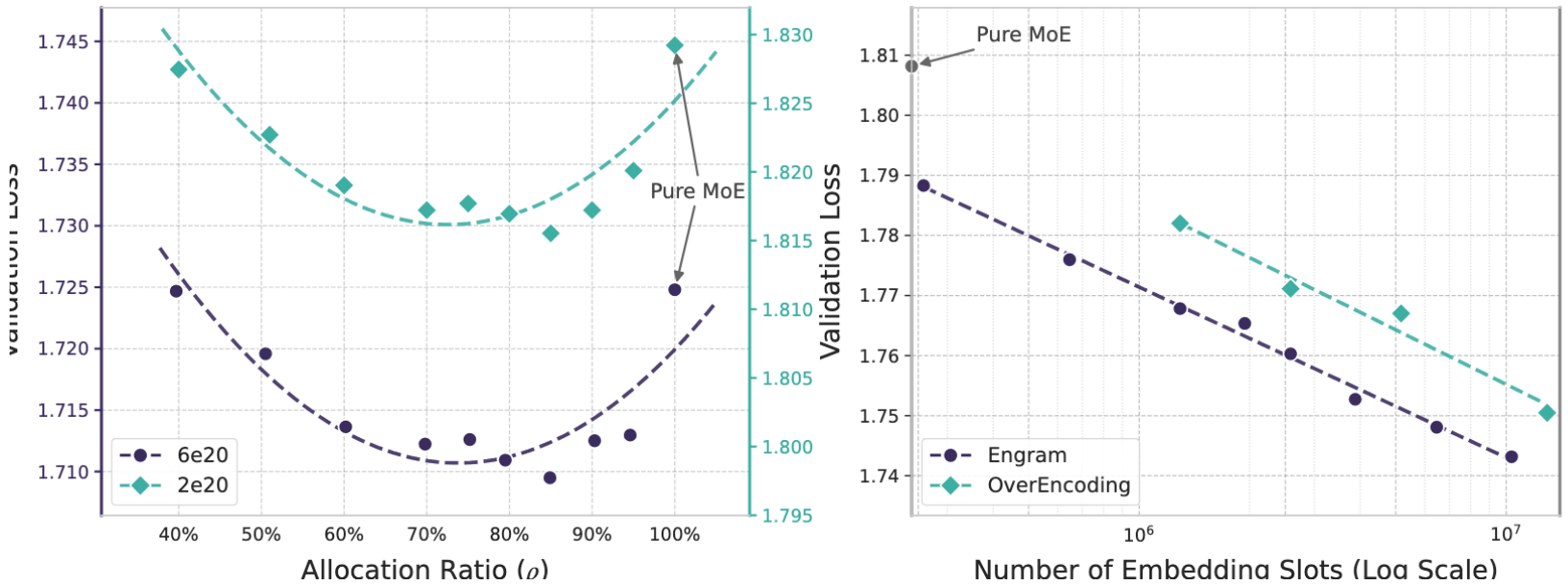

Figure: Scaling plot

Large-scale Pretraining

Pre-training Data: 262 billion tokens.
Tokenizer: DeepSeek-v3 (Vocabulary size: 128k).
Model Architecture: 30-block Transformer with a hidden size of 2560.
Layer Components: Multi-head Latent Attention (MLA) with 32 heads; connected to FFNs via mHC (Expansion rate: 4).
Optimizer: Muon.
Experimental Control: Consistent default settings are applied across all models for comparison.

Key Benchmark Results: Engram vs. Baselines

Engram-27B consistently outperforms MoE-27B across **part** of tasks in the paper.

Table: Comparison under constant activated parameters (3.8B)

Benchmark (Metric)	# Shots	Dense-4B	MoE-27B	Engram-27B	Engram-40B
Model Specs					
# Total Params	-	4.1B	26.7B	26.7B	39.5B
# Activated Params	-	3.8B	3.8B	3.8B	3.8B
Knowledge & Reasoning					
MMLU (Acc.)	5-shot	48.6	57.4	60.4	**60.6**
ARC-Challenge (Acc.)	25-shot	59.3	70.1	73.8	**76.4**
HellaSwag (Acc.)	0-shot	64.3	71.8	72.7	**73.1**
Code & Math					
HumanEval (Pass@1)	0-shot	26.8	37.8	**40.8**	38.4
GSM8K (EM)	8-shot	35.5	58.4	60.6	**62.6**
MATH (EM)	4-shot	15.2	28.3	**30.7**	30.6

Why 27B beats 40B?

The explanation in the paper

"Although it does not yet strictly dominate Engram-27B on every task, this is likely an artifact of under-training. We observe that the training loss gap between Engram-40B and the baselines continues to widen towards the end of training, suggesting that the expanded memory capacity has not yet fully saturated within the current token budget."

Table of Contents

Engram equivalent to increasing the model's depth

They calculate the KL-divergence and the semantic similarity between each layers on MoE and Engram

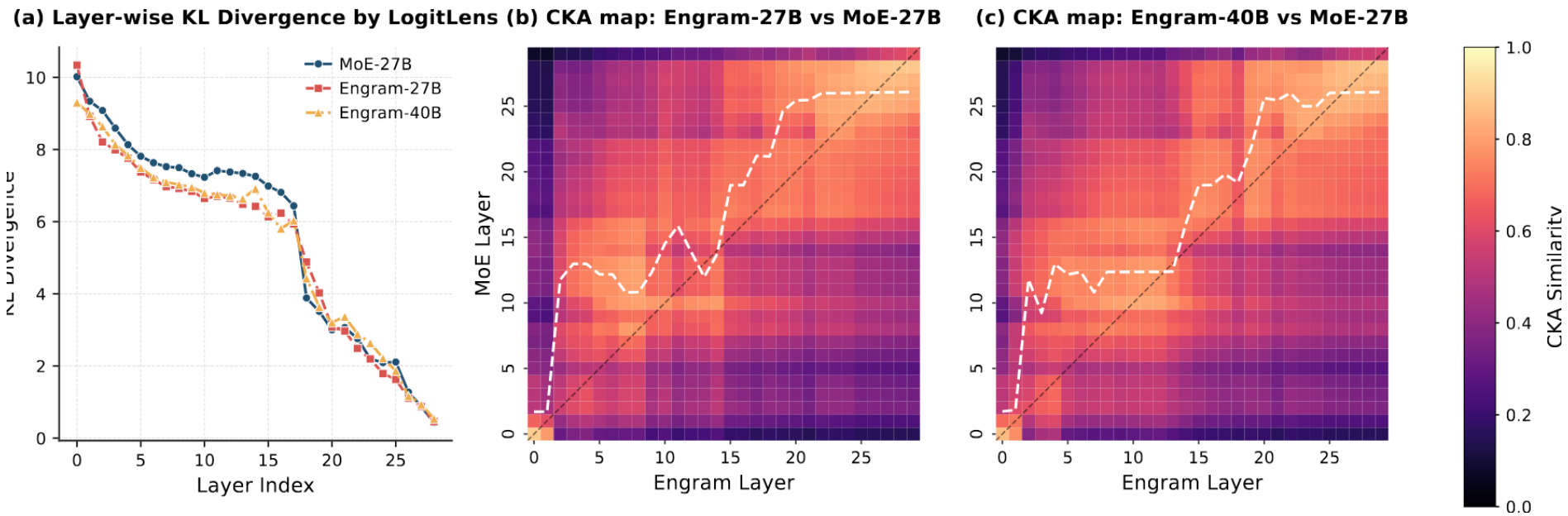

Figure: Left: Layer-wise KL Divergence via LogitLens Right: Similarity heatmap computed by CKA

System Efficiency

Table: Impact of 100B Engram Layer (CPU Offload) on Throughput

Base Model	Configuration	Throughput (tok/s)
Experimental Setup: NVIDIA H800, 512 Seq, Len 100-1024		
4B-Dense	Baseline	9,031.62
	+ 100B Engram (Offload)	**8,858.28**
8B-Dense	Baseline	6,315.52
	+ 100B Engram (Offload)	**6,140.02**

System Advantage: Deterministic Memory Access

Unlike MoE, Engram uses **static hash IDs**, allowing the system to prefetch embeddings asynchronously via PCIe. This overlaps memory transfer with GPU computation, resulting in $<$**3% throughput overhead** despite adding 100B external parameters.