Pretraining with hierarchical memories: separating long-tail and common knowledge

Hadi Pouransari, **David Grangier**, **C Thomas**, **Michael Kirchhof**, **Oncel Tuzel**

Apple

The impressive performance gains of modern language models currently rely on scaling parameters: larger models store more world knowledge and reason better. Yet compressing all world knowledge into parameters is unnecessary, as only a fraction is used per prompt, and impractical for edge devices with limited inference-time memory and compute. We address this shortcoming by a memory-augmented architecture and a pretraining strategy aligned with existing hardware paradigms. We introduce small language models that access large hierarchical parametric memory banks encoding world knowledge. During pretraining and inference, we fetch a small, context-dependent memory block and add it to the model. Our pretraining learns to store long-tail world knowledge in the memory parameters, while the small language model acts as an anchor capturing common knowledge and general reasoning abilities. Through trillion-token-scale experiments, we show significant gains: a 160M-parameters model augmented with an 18M-parameters memory fetched from a 4.6B memory bank obtains comparable performance to a regular model with more than $2\times$ the parameters. Through extensive experiments, we study the optimal type and size of parametric memories in transformers, scaling them to over 21B parameters. We find that our proposed hierarchical feed-forward memories work robustly across transformer architectures, whether added during pretraining or post-hoc.

Correspondence: Hadi Pouransari: mpouransari@apple.com

Date: October 7, 2025

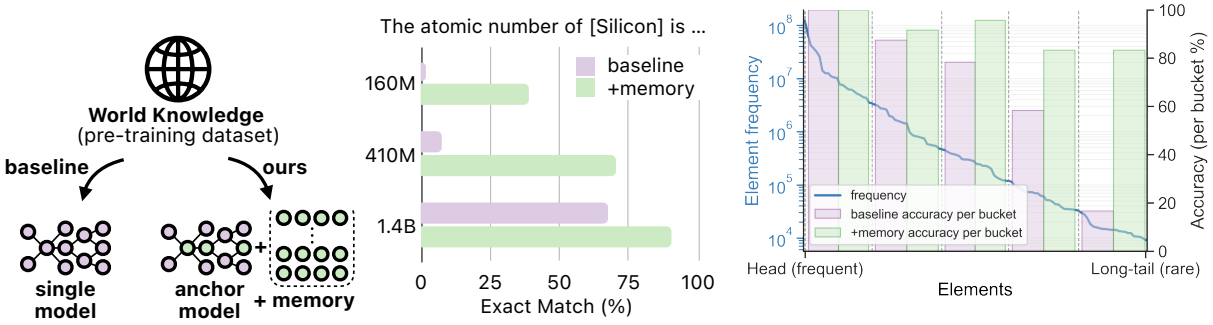

Figure 1 **Left:** Schematic of pretraining-with-memories: some parameters are always used (anchor parameters), others are fetched per input document (memory parameters). **Middle:** Accuracy improvement over baseline when $\simeq 10\%$ of parameters are allocated as memories for a knowledge-intensive task (predicting the atomic numbers of elements), using models with 160M, 410M, and 1.4B parameters, corresponding to rows A2, B2, and C2 in Table 1. **Right:** Elements sorted by their frequency of appearance in the DCLM-Baseline dataset (5 buckets, each with $\simeq 24$ elements). With the proposed *pretraining-with-memories*, we observe significant improvements, especially on long-tail data. While the baseline 1.4B model has only 17% accuracy on the least frequent element bucket, augmenting it with only 10% memory parameters increases the accuracy to 83%.

1 Introduction

Frontier large language models (LLMs) have advanced significantly in recent years, demonstrating strong capabilities across both world knowledge and reasoning tasks. These improvements have largely been driven by scaling the number of parameters and training tokens. However, limited results exist on the role of model parameter count in each specific aspect, and on whether knowledge and reasoning can be disentangled.

LLMs memorize many facts in their parameters during pretraining (Roberts et al., 2020), most of which are long-tail knowledge, overly specific, and unnecessary for the intended use. For example, part of the parameters of the Qwen3-2B (Yang et al., 2025) model store the fact that *Albert Einstein was born on March 14, 1879*, a detail that is not essential for executing on-device personal assistant tasks, yet is permanently loaded into RAM and considered in each computation. Ideally, this capacity would be utilized for reasoning and commonsense abilities.

We propose to use a base model as an *anchor* to capture common knowledge and reasoning capabilities and augment it with a *memory* bank that dedicates a large set of memory parameters to long-tail knowledge. In Figure 1, we illustrate this through a representative knowledge-intensive task: predicting the atomic number of elements. Baseline pretraining performance degrades for samples with lower frequency in the pretraining data. This degradation is attributed to catastrophic forgetting (Toneva et al., 2018), which arises from destructive gradient updates caused by dissimilar content (Ghosal et al., 2025) applied to the same set of parameters. In contrast, in the proposed approach, memory parameters are activated and updated only on sequences of similar topics, thereby reducing susceptibility to forgetting. Besides training dynamics advantages, separating long-tail knowledge into dedicated memory parameters offers several additional benefits, as discussed below.

Runtime efficiency: For on-device deployment of LLMs, the main bottleneck is the availability of large and fast memory. Methods such as mixture-of-experts (Shazeer et al., 2017) (MoEs) improve compute efficiency by activating only a fraction of the feed-forward modules. Still, MoEs require on-demand random access to the full set of experts at every layer and for every token, such that all model parameters remain loaded, not just the active ones. This makes on-device deployment of MoEs challenging. In our proposed method, depending on the context, only the required knowledge parameters are attached to the anchor model (see Figure 2). This allows fast on-device memory to be used primarily for anchor model parameters, while knowledge parameters are stored in a slower but larger storage. Furthermore, we learn knowledge in the form of *hierarchical* memories, allowing inference to naturally align with existing device memory hierarchies (RAM, flash storage, external disk) and benefit from their compositionality over a session of interaction with the model (see Figure 5).

Training efficiency: A key bottleneck in large-scale distributed training of standard LLMs is the need to communicate large gradient tensors between compute nodes. In the proposed approach, during pretraining, based on the content of documents in a batch, only a small fraction of memory bank parameters is retrieved and updated together with the anchor model parameters. As a result, the gradients are highly sparse, which substantially reduces node-to-node communication. This property makes the proposed method well-suited for co-locating training data and compute in massively distributed setups, similar to branch-train-merge (BTM) methods (Li et al., 2022; Gururangan et al., 2023a). For instance, pretraining an anchor model with 160 million parameters and 4.6 billion memory parameters requires less than 1.7× the compute budget of training a 160M model alone.

Privacy and knowledge editing: Separating knowledge and reasoning abilities enables a direct mapping between training tokens and specific subsets of parameters (memories). Ownership of training data tokens can therefore be linked to ownership of the corresponding memories. This makes it possible to remove or edit certain data from the model by deleting or updating the associated memory parameters, or to restrict access to specific memories while keeping the anchor model public. We show the effect of memory blocking in Section 4. The proposed approach also allows creating new memories from new data after pretraining, as shown in Section 4. Through this mechanism, a strong public reasoning model can be readily augmented with memories built from large private user data or organizational databases, providing a more efficient alternative to very long contextual memories.

Our main contributions are transformer architectures augmented with large hierarchical memories (Figure 2)

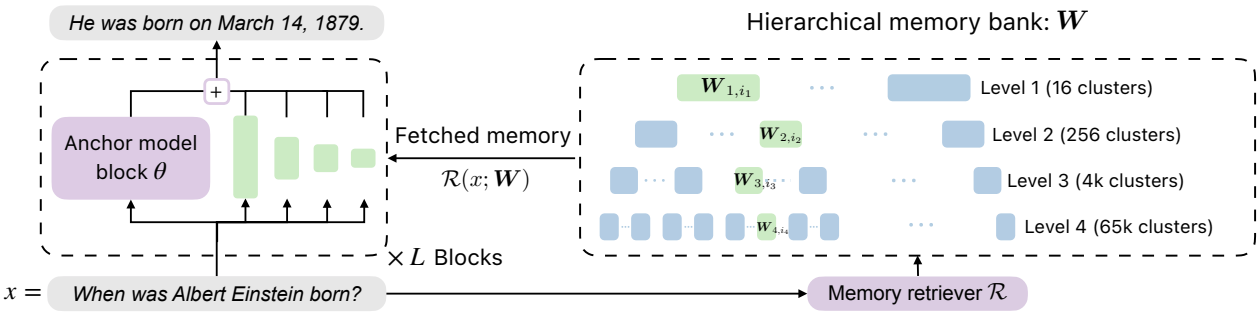

Figure 2 Proposed architecture: For a given context x (such as a question text), the memory retriever module selects relevant parameters from a large set of memory bank parameters. These memory parameters are organized hierarchically based on the hierarchical clustering of the pretraining data. The anchor model, together with the retrieved memories, then responds to the question.

and a clustering-based pretraining method (Section 2.1), through which long-tail knowledge is automatically incorporated into the memory bank. We systematically examine how different memory types (Figure 3), depths (Figure 3d), sizes (Figure 4a), positions (Table 9), and memory-to-model ratios (Figure 4b) affect performance. We further analyze the runtime efficiency of the proposed method in relation to hardware storage hierarchies (Section 3). Memory learning is scaled to banks containing up to 21B parameters. Results demonstrate consistent improvements over baseline models (Table 1) and retrieval-augmented generation strategies (Table 3), and robust applicability to arbitrary transformer architectures (Table 2). For example, with approximately 10% additional parameters retrieved from the memory bank, we observe gains on knowledge-specific tasks of 34.1% $\rightarrow$ 40.3% for a 160M model and 40.9% $\rightarrow$ 45.9% for a 410M model.

2 Pretraining with memories

Consider a language model with parameters $\boldsymbol{\theta}$, called anchor parameters, a large bank of memory parameters $\boldsymbol{W}$, and a retriever module $\mathcal{R}$ that, given a context x, fetches only the relevant memory parameters $\mathcal{R}(x; \boldsymbol{W})$ from the memory bank. During pretraining, we optimize the next token prediction loss for each document x in the dataset $\mathcal{D}$:

$$\mathcal{L}(x) = -\sum_t \log \mathbb{P}_{\boldsymbol{\theta}, \mathcal{R}(x; \boldsymbol{W})}(x_t \mid x_{<t}), \tag{2.1}$$

where $\mathbb{P}_{\boldsymbol{\theta}, \mathcal{R}(x; \boldsymbol{W})}(x_t \mid x_{<t})$ is the model's output probability distribution over the vocabulary for token t in document x, given its previous tokens, using anchor parameters $\boldsymbol{\theta}$ augmented with the retrieved memory parameters $\mathcal{R}(x; \boldsymbol{W})$.

In terms of parameter counts, we generally assume $|\mathcal{R}(x; \boldsymbol{W})| \ll |\boldsymbol{\theta}| \ll |\boldsymbol{W}|$. Therefore, with the training objective in Equation (2.1), the memory bank parameters $\boldsymbol{W}$ are updated only sparsely. Intuitively, we expect $\mathcal{R}(x; \boldsymbol{W})$ to retrieve the same subset of parameters from $\boldsymbol{W}$ for inputs x that are semantically similar. As a result, memory parameters receive gradients primarily from documents with related content, mitigating forgetting (Ghosal et al., 2025) and enabling them to efficiently *memorize* long-tail world knowledge. In contrast, $\boldsymbol{\theta}$ receives gradients for all $x \in \mathcal{D}$ and is expected to primarily capture common knowledge and general reasoning capabilities.

At inference time, the model uses only $|\boldsymbol{\theta}| + |\mathcal{R}(x; \boldsymbol{W})|$ parameters, introducing only a minor overhead compared to a model that relies solely on $\boldsymbol{\theta}$. Figure 2 presents the overall architecture. We next describe the construction of each component in this framework.

2.1 Clustering-based memory retriever

Hierarchical clustering: Given a pretraining dataset $\mathcal{D}$, we use an off-the-shelf text embedding model ϕ to map each document $x \in \mathcal{D}$ to $\phi(x) \in \mathbb{R}^c$. We then perform hierarchical clustering using the document embeddings: first, we divide the documents into k clusters using k-means; next, we further subdivide each

cluster into k sub-clusters, and continue this process for p levels. Finally, we obtain k^l nested clusters at each level. In this paper, we cluster the DCLM-baseline dataset (Li et al., 2024) with 3.2 billion documents into a hierarchy with $p = 4$ levels and dividing factor of $k = 16$ resulting in $16, 16^2, 16^3$, and 16^4 clusters at levels 1, 2, 3, and 4, respectively (see Figure 2). We use Sentence-BERT `all-MiniLM-L6-v2` embedding model (Reimers and Gurevych, 2019) with dimension $c = 384$. See more details in Appendix C.

To retrieve a cluster, we map a document x to an index tuple $\mathcal{I}(x) = (i_1, i_2, \ldots, i_p)$ by traversing the clustering tree greedily: at each level l, $\phi(x)$ is compared to k centroids and the sub-tree corresponding to the closest (via L2 distance) centroid (i_l) is then visited. This greedy traversal leads to a fast and scalable retrieval within $\mathcal{O}(pk)$ comparisons. For our tree of depth $p = 4$, we denote the cluster index as $\mathcal{I}(x) = (i_1, i_2, i_3, i_4)$. We pre-compute the cluster index for each document in the pretraining dataset offline, resulting in no training-time overhead. See details in Section A.2. At test time, we use the same traversal to get the cluster index for the task context (e.g., question text).

Hierarchical memories: We assign a memory parameter block to each cluster in a given hierarchical clustering tree, denoted by $\boldsymbol{W}_{l,i_l} \in \mathbb{R}^{s_l}$, where $l \in \{1, 2, 3, 4\}$ denotes the level, $i_l \in \{1, \ldots, k^l\}$ is the cluster index at level l, and s_l is the size of the memory parameter blocks at level l. The memory bank $\boldsymbol{W}$ consists of all memory parameter blocks $\boldsymbol{W}_{l,i_l}$. The memory retriever is then:

$$\mathcal{R}(x; \boldsymbol{W}) = [\boldsymbol{W}_{1,i_1}, \boldsymbol{W}_{2,i_2}, \boldsymbol{W}_{3,i_3}, \boldsymbol{W}_{4,i_4}], \quad \text{where } \mathcal{I}(x) = (i_1, i_2, i_3, i_4). \tag{2.2}$$

The total number of parameters in the memory bank is $|\boldsymbol{W}| = s_1 k + s_2 k^2 + s_3 k^3 + s_4 k^4$, and the number of retrieved parameters (*fetched memory* size) is $|\mathcal{R}(x; \boldsymbol{W})| = s_1 + s_2 + s_3 + s_4$.

2.2 Inference with parametric memories

There are multiple ways to add parametric memories $\mathcal{R}(x; \boldsymbol{W})$ to an anchor model $\boldsymbol{\theta}$, i.e., to practically model $\mathbb{P}_{\boldsymbol{\theta}, \mathcal{R}(x; \boldsymbol{W})}(x_t \mid x_{<t})$. We limit our discussion to decoder-only, transformer-based (Vaswani et al., 2017) language models. For all memory types, we instantiate them such that at the beginning of training they have no effect on the anchor model. See details in Appendix I.

LoRa-Memories: A popular approach to augment a model with a (small) set of extra parameters is to patch its linear layers with low-rank adaptation matrices (LoRa, Hu et al., 2022). We consider three types of LoRa memories: LoRa-QK adapts the query and key projection layers (see Figure 14), LoRa-VO adapts the value and output projection layers (see Figure 15), and LoRa-FFN adapts all three linear layers in the SwiGLU feed-forward network (FFN) (Shazeer, 2020) with rank r matrices (see Figure 16).

KV-Memories: The knowledge in the context tokens a transformer attends to is ultimately a sequence of KV-caches. A natural extension of providing context knowledge is thus to learn KV-cache parameters directly. This can also be seen as a generalization of prefix tuning (Li and Liang, 2021; Lester et al., 2021), where only input token embeddings are learned. At each transformer layer, data-dependent query tokens cross-attend to r fetched KV memories (see Figure 17).

FFN-Memories: Previous works have argued that transformer-based language models mainly store their knowledge in the FFN layers (Geva et al., 2020; Dai et al., 2022; Yao et al., 2022). Inspired by this, we introduce FFN memories: we expand the inner dimension of the SwiGLU FFN layers by r through concatenation with the fetched memory parameters, which is equivalent to a fast addition (see Figure 18).

The number of parameters s_l in a memory block assigned to clusters at level l depends on the memory type, the anchor model's architecture (e.g., hidden dimension, depth, etc.), and the memory block size multiplier r (rank for LoRa, number of KV memory tokens, or FFN dimension expansion size). Therefore, a hierarchical memory configuration (s_1, s_2, s_3, s_4) can be written as $c_0(r_1, r_2, r_3, r_4)$, where c_0 is a constant determined by the anchor model architecture and memory type, and r_l is the memory block size multiplier for memories at level l that we control. For simplicity, we drop the constant c_0 in the rest of the paper and provide details in Appendix I. In practice, we generally set these multipliers so that coarser levels have larger parameter blocks, i.e., $r_1 \geq r_2 \geq r_3 \geq r_4$, or set $r_l = 0$ when no memories are assigned to clusters at level l.

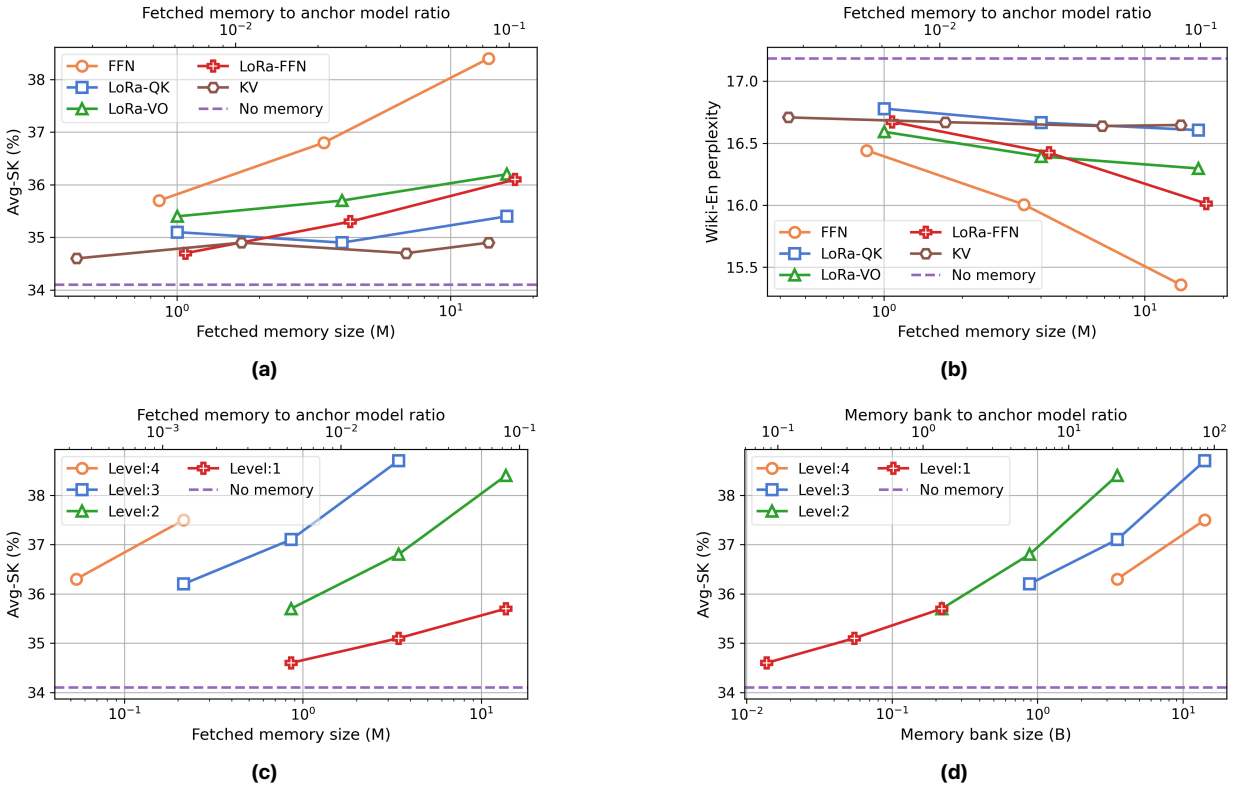

Figure 3 Effect of **memory type** on Specific-Knowledge benchmarks (a) and Wikipedia perplexity (b). Effect of **memory level** on performance as a function of fetched memory size (c) and bank size (d).

3 Design choices for pretraining with memories

We use DCLM-Baseline (Dai et al., 2022) for training. The dataset contains $\simeq 3.2$ billion documents ($\simeq 4.3$ trillion tokens). We cluster the dataset to a tree with 4 levels, having $16, 16^2, 16^3$, and 16^4 clusters at each level. See Appendix A for training details. For evaluation, we consider 13 frequently used benchmarks, including multiple-choice and generative tasks. We divide these tasks into two groups based on the level of specific knowledge required (see Appendices B and G for details):

Common-Knowledge (Avg-CK): *Lambada-OpenAI* (Paperno et al., 2016), *BoolQ* (Clark et al., 2019), *SQuAD* (Rajpurkar et al., 2016), *Winograd* (Levesque et al., 2012), *CoQA* (Reddy et al., 2019), and *WinoGrande* (Sakaguchi et al., 2021).

Specific-Knowledge (Avg-SK): *Hellaswag* (Zellers et al., 2019), *ArcEasy/Challenge* (Clark et al., 2018), *TriviaQA* (Joshi et al., 2017), *NaturalQuestions-Open* (Lee et al., 2019; Kwiatkowski et al., 2019), *PIQA* (Bisk et al., 2019), and *OpenBookQA* (Mihaylov et al., 2018).

For evaluation, we retrieve the memories based only on the question text. As an open-ended generation task, we also track the average perplexity on the 2022 English Wikipedia (**Wiki-En**) with $\simeq 6.5$M samples (4B tokens), where memories are retrieved based on the full Wikipedia document.

FFN-Memories outperform LoRa and KV memories: In Figure 3, we compare different memory types introduced in Section 2.2. We attach the memories to a pretrained model with 160M parameters (row A1 in Table 1) and train them from scratch for 275B tokens with the loss objective in Equation (2.1), while the anchor model parameters ($\boldsymbol{\theta}$) are frozen. For this set of experiments, we use a single-level memory configuration in the form of $(0, s_2, 0, 0)$, corresponding to a total of $16^2 = 256$ memories, each with size s_2, for different values of s_2. We observe that FFN-Memories have a significant advantage over other forms of memory across all memory sizes. Based on this observation, in the rest of the paper we use only FFN-Memories.

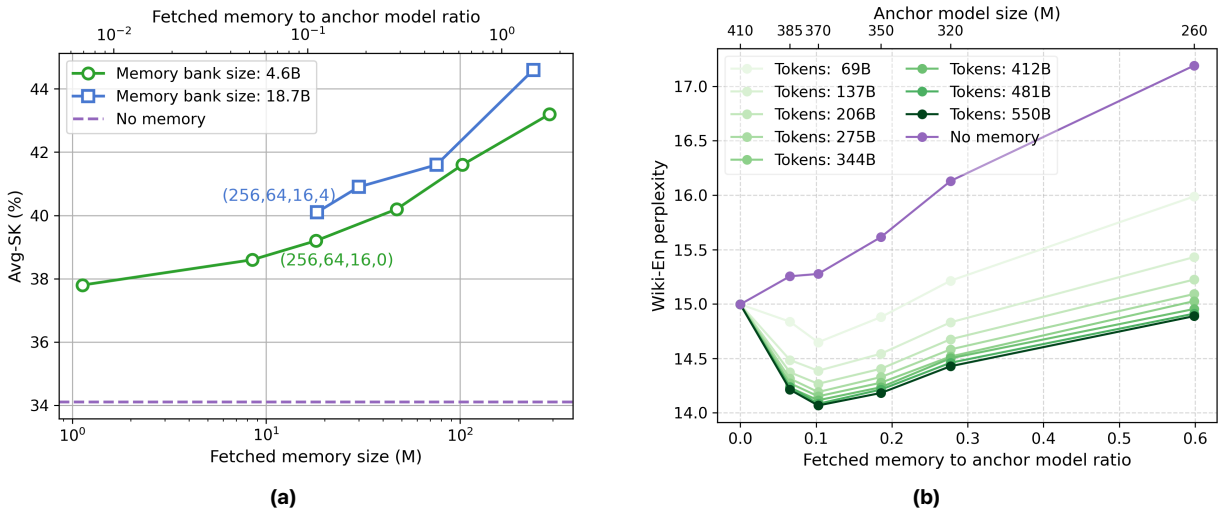

Figure 4 (a) Avg-SK accuracy for different hierarchical memories, demonstrating **performance gain with larger bank size and fetched memory size**. (b) Wiki-En perplexity for different **fetched memory–to–anchor model size ratios, with the optimal point at 1:10**. The purple curve shows the perplexity of anchor models without memory. The green curves show the perplexity of models with memory, with different shades of green corresponding to the progress of memory training.

Accuracy improves with deeper and larger memories: Here, we explore the design space of different single-level memory configurations (s_1, s_2, s_3, s_4), where only one of the s_l's is non-zero. The anchor model is pretrained and frozen (row A1 in Table 1) during memory training (see Section A.3 for details). As shown in Figure 3c, for a constant fetched memory size (i.e., s_l), deeper memories yield greater accuracy improvements, as they capture more relevant and detailed information for a given query. In Figure 3d, we show that performance is a strictly increasing function of total memory bank size for all memory configurations, as more capacity becomes available to capture long-tail knowledge. Shao et al. (2024) recently made a similar observation for regular RAG setups by increasing the size of the datastore from which documents are retrieved. Note that for a fixed total memory bank size, a shallower memory corresponds to a larger fetched memory size. Therefore, in Figure 3d, at a fixed memory bank size, shallower memories achieve higher accuracy.

Hierarchical memories enable an optimal design: Unlike single-level memories used in previous experiments, a general hierarchical memory configuration (s_1, s_2, s_3, s_4) with possibly multiple non-zero values allows independent control of the total memory bank size ($\sum_l 16^l s_l$) and the size of fetched memory parameters at inference time ($\sum_l s_l$). For a large total memory bank size with a small number of fetched parameters at inference, we can use larger level-3 and level-4 memories. Conversely, for a smaller total bank size with more fetched parameters at inference, we can increase the sizes of level-1 and level-2 memories.

From a learning dynamics perspective, in regular language modeling, all parameters are updated at every iteration, receiving gradients from a wide range of dissimilar documents in the dataset. As a result, long-tail information is often forgotten (Toneva et al., 2018). When training with hierarchical memories, however, memory bank parameters at level l are activated 16^l times less frequently compared to anchor model parameters (which can be considered level-0). Consequently, deeper memory bank parameters receive fewer gradient updates, and from more similar content, shielding them from forgetting (Ghosal et al., 2025). This leads to effective learning of a hierarchy of memories, ranging from the most common knowledge at level 1 to the most specific knowledge at level 4.

Above, we showed that single-level memories benefit from larger fetch size and bank size. We now systematically demonstrate this for a general hierarchical configuration. We evaluate two groups of hierarchical memories added to our frozen pretrained 160M model (row A1 in Table 1): one with a memory bank size of 4.6B and another with 18.7B, both with configurations spanning between 1M and 300M fetched parameters (see Section A.4 for details). Results in Figure 4a confirm that performance increases strictly with fetched memory size (while keeping bank size fixed) and with bank size (while keeping fetched memory size fixed) for the

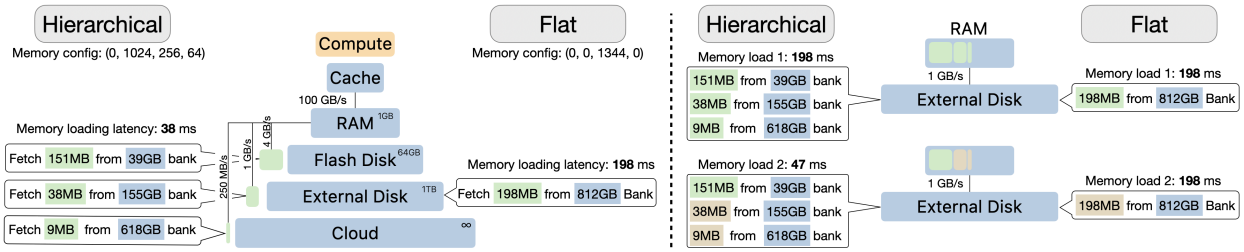

Figure 5 Deployment advantages of hierarchical memories. **Left:** Memory loading latency is reduced by using the hardware hierarchy. **Right:** Latency is reduced by exploiting compositionality over time: larger memories for low-level clusters, once loaded, are less likely to need reloading.

general hierarchical configurations. For example, changing the memory configuration from `(256,64,16,0)` to `(256,64,16,4)` barely changes the fetched memory size but corresponds to bank sizes of 4.6B and 18.7B, respectively, and results in an Avg-SK improvement from 39.1% to 40.1%.

The point with highest accuracy in Figure 4a corresponds to augmenting a 160M anchor model with $\simeq$ 240M fetched memory parameters, achieving 44.5% accuracy on Avg-SK. This is a notable result, showing that an anchor+memory model with a total of 400M runtime parameters outperforms a regularly trained 410M model (row B1 in Table 1) by 3.6 points. Building on this observation, we ask the following question: *For a given runtime parameters budget, how many parameters should be allocated to the anchor model and how many to the fetched memories?*

To explore this, we train a sequence of anchor models with 260M, 320M, 350M, 370M, 385M, and 410M parameters, freeze them, and pair them with fetched memories of sizes 150M, 90M, 60M, 40M, 25M, and 0, respectively, such that the total runtime parameter count for anchor+memory remains fixed at 410M. All configurations use a 6.3B memory bank (see Section A.7). In Figure 4b, a 1:10 ratio of fetched memory to anchor model size appears optimal and guides our next experiments. However, this observation may not generalize to different runtime, memory bank, and training budgets, or when the anchor model and memory parameters are co-trained.

Models with hierarchical memories have on-device deployment advantages. Assuming a von Neumann architecture with hardware organized in increasing size and decreasing speed (RAM, flash disk, external disk), we can store shallower memory levels (with larger fetch size but smaller bank size) on faster components, while offloading deeper levels to slower storage (Alizadeh et al., 2024). In the example in Figure 5 (left), with **a hypothetical hardware setup**, a hierarchical memory can be loaded in 38ms, whereas loading a memory of the same fetch size from a flat bank (stored on the external disk due to its excessive total size) takes 198ms—more than 5× longer. Additionally, even if both hierarchical and flat memory banks are stored exclusively on the external disk, hierarchical memories still provide a loading speed advantage due to their *compositionality*. For example, when generating different atomic numbers in Figure 1, the level-1 and level-2 memories remain mostly unchanged, and only deeper memories need to be swapped. This takes 47ms, compared to 198ms when the entire flat memory must be reloaded, as shown in Figure 5 (right).

4 Results

In this section, building on the findings from Section 3, we provide a comprehensive set of results for different-sized models and compare them with baselines. See Appendix A for training details.

Starting from a 160M anchor model pretrained regularly (row A1 in Table 1), we add memories with the $(256, 64, 16, 0)$ configuration, corresponding to a fetched memory size of $\simeq$ 18M parameters and a total memory bank size of $\simeq$ 4.6B parameters. When memories are learned with a frozen anchor model (row A3), we observe a +5.1 points improvement on Avg-SK compared to A1 and a 2 points reduction in Wiki-En perplexity, demonstrating the effectiveness of memories for tasks requiring specific knowledge, with only $\simeq$ 10% additional runtime parameters from fetched memories.

To ensure a fair comparison, we also train a *generic* memory with the same size as the fetched memories (18M parameters) together with the memory bank parameters. When evaluating with generic memory, unlike

Row	Anchor model	Init.	Seen tokens	Cotrain anchor	Memory config	Bank size	Fetch size	Avg-CK (%) ↑		Avg-SK (%) ↑		WikiEn Pplx ↓	
								Generic	Fetched	Generic	Fetched	Generic	Fetched
A1	160M	Scratch	1.1T	n/a	(0,0,0,0)	0	0	45.9		34.1		17.2	
A2		A1	+1.1T	Yes	(256,64,16,0)	4.6B	18M	47.9	**48.7**	35.7	**40.3**	16.7	**14.2**
A3		A1	+1.1T	No				46.6	47.4	34.7	39.2	16.7	15.2
A4		Scratch	2.2T	Yes				46.6	46.7	33.8	39.6	17.8	15.6
B1	410M	Scratch	1.1T	n/a	(0,0,0,0)	0	0	52.3		40.9		13.9	
B2		B1	+1.1T	Yes	(512,128,32,0)	12.7B	50M	55.5	**56.1**	41.8	**45.9**	13.8	**12.4**
B3		Scratch	2.2T	n/a	(0,0,0,0)	0	0	53.2		41.1		13.8	
C1	1.4B	Scratch	1.1T	n/a	(0,0,0,0)	0	0	61.2		49.7		10.8	
C2		C1	+1.1T	Yes	(768,256,16,0)	21.1B	153M	64.4	**64.5**	51.3	**54.9**	11.0	**10.2**

Table 1 Summary of results for pretraining with memories at different parameter scales.

fetched memories that are retrieved based on context, we simply use the anchor+generic memory parameters. This isolates the effect of merely increasing the number of parameters and training tokens in the anchor model. Anchor+generic memory scores 34.7% on Avg-SK, 4.5 points below fetched memories, showing the benefit of context-based retrieval in isolation.

We next explore co-training the anchor model parameters ($\boldsymbol{\theta}$) and the memory parameters ($\boldsymbol{W}$) together, as in Equation (2.1). For a fair comparison, during training we use the generic memory with probability $1/(16+1)$ and the fetched memory with probability $16/(16+1)$, where 16 is the clustering division factor. This ensures there is no training bias in favor of the memory bank parameters.

Row A2 in Table 1 shows co-training results, with Figure 6a illustrating training curves for A2 and A3 experiments. A key observation is that when we allow the anchor parameters to be co-trained with the memory bank, we obtain greater improvement compared to the case where the anchor model is frozen (Avg-SK 39.2% → 40.3%). This gain can be attributed to two factors: 1) when co-training, the anchor model learns to utilize the memories more effectively compared to when it is frozen, and 2) the anchor model performance improves simply due to more training. The latter effect should be minor if the anchor model is already converged. We provide additional discussion in Appendix E.

We also explore co-training the anchor model and memory bank together *from scratch*. Results are shown in row A4 of Table 1. Despite using the same total training budget as row A2, A4 shows lower performance. This result suggests that memories are learned more effectively after the anchor model has been trained to some extent (i.e., the setup of row A2). This is analogous to human memory, which develops only after the brain gains semantic understanding, around age 3 (Shaw, 2016).

Next, we scale the co-training setup of row A2 to anchor models of sizes 410M and 1.4B, corresponding to rows B2 and C2 in Table 1. These models are paired with memory configurations such that the fetched memory is approximately 10% of the anchor model size, corresponding to memory bank sizes of 12.7B and 21.1B for the 410M and 1.4B anchor models, respectively. We observe similar improvements for these larger models: for Avg-SK, fetched memories outperform generic memories by +4.1 points for the 410M and +3.6 points for the 1.4B model.

Finally, for the 410M model, we train a model regularly (without memories) with the same total training budget as row B2 (2.2T tokens). We observe that Avg-CK performance is worse than that of the anchor+generic setup in row B2. These preliminary results suggest that when the anchor model is co-trained with a large memory bank (as in row B2), long-tail knowledge is offloaded to the memories, enabling the anchor model to perform better on common-knowledge tasks.

Blocking part of the memory bank: In Figure 6b, we show the performance of the 410M model with memories (row B2 in Table 1) on predicting atomic numbers of elements (see Figure 1) when the best-matching parts of the memory bank are adversarially blocked during retrieval. We observe a significant performance drop, from 70% to 20%, when blocking 1/16 of the bank. This preliminary result highlights the potential of the proposed approach for applications with privacy goals.

Adding memory to other pretrained models: We explore the post-hoc addition of parametric memories to open-weight models. We span multiple sizes and architectures, namely Gemma 3 270M (Gemma Team

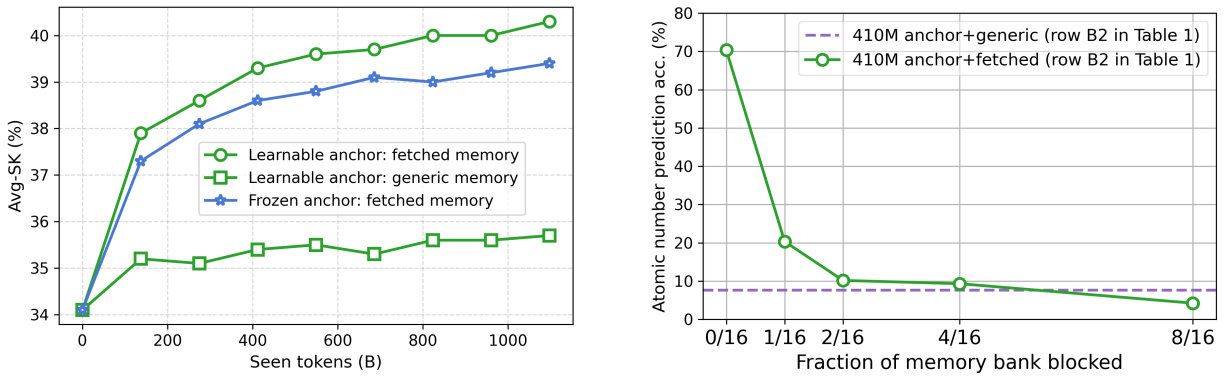

Figure 6 (a) Performance improvements on Avg-SK when co-training memories and anchor model parameters jointly during training. (b) Effect of blocking parts of the memory bank from retrieval.

Pretrained model	Bank size	Fetch size	Avg-CK (%) ↑	Avg-SK (%) ↑	Atomic Number Acc. (%) ↑
Gemma 3 270M	0	0	44.2	34.3	1.7
+ memory	5.9B	23.2M	**44.8**	**38.2**	**49.2**
Qwen 2.5 0.5B	0	0	**53.9**	40.6	53.4
+ memory	11.1B	43.4M	52.1	**44.5**	**90.4**
Llama 3.2 1B	0	0	**58.9**	46.6	**96.6**
+ memory	14.1B	102.2M	57.6	**50.5**	**96.6**

Table 2 Learning memory on top of pretrained open-weight models post-hoc. All trainings use 1.1 trillion tokens from DCLM.

Anchor model	Inference setup	Bank size	FLOPs	Avg-CK 0-shot	Avg-SK 0-shot
160M	Baseline	0	×1	43.6	32.8
	RAG-DCLM	70 TB	×2.3	42.4	32.6
	RAG-Wiki	21 GB	×1.7	42.4	35.0
	10% Memory	9 GB	×1.11	**45.3**	**38.4**
410M	Baseline	0	×1	48.6	38.5
	RAG-DCLM	70 TB	×2.3	48.2	38.1
	RAG-Wiki	21 GB	×1.7	47.3	41.6
	10% Memory	25 GB	×1.1	**52.0**	**44.5**
1.4B	Baseline	0	×1	56.0	46.9
	RAG-DCLM	70 TB	×2.3	55.8	46.1
	RAG-Wiki	21 GB	×1.7	55.5	49.2
	10% Memory	42 GB	×1.1	**59.3**	**52.4**

Table 3 Comparison with vanilla RAG.

et al., 2025), Qwen 2.5 0.5B (Yang et al., 2024), and Llama 3.2 1B (Meta AI, 2024). We add hierarchical FFN memories of $\simeq 10\%$ the model size post-hoc to pretrained (and frozen) anchor models and train the memories for 1.1T tokens on DCLM, see Section A.6 for additional details. Results are shown in Table 2. As above, the specific knowledge accuracy improves with memories. This is consistent across all architectures, showing the generality of adding hierarchical memories across all models. Common knowledge remains the same or slightly decreases; potentially because these models were pretrained on more tuned data mixtures than the simple open-source data mixture DCLM that we use in these experiments.

Comparing with vanilla retrieval-augmented generation: An alternative, yet complementary, approach to parametric memory is retrieval-augmented generation (RAG), where relevant texts are retrieved from a datastore and prepended to the context (Lewis et al., 2020; Ram et al., 2023; Izacard et al., 2023) to improve performance on knowledge-intensive tasks. Using our Sentence-BERT embedding model as the retriever and DCLM training data as the datastore, we evaluate RAG on the baseline models from rows A1, B1, and C1. See implementation details in Appendix H.

As shown in Table 3, vanilla retrieval from DCLM performs poorly relative to the baseline models, while adding more than 2× runtime FLOPs and requiring large storage for the raw-document datastore. This is likely due to the low quality of DCLM (a pretraining dataset) when used as a RAG datastore. To give RAG an advantage, we also retrieve from the higher-quality Wiki-En. RAG-Wiki improves baseline performance on SK (e.g., from 46.9 to 49.2 for the 1.4B model) while remaining slightly below baseline on CK. By contrast, learned memories (with $\simeq 10\%$ extra parameters) improve both CK and SK, with lower FLOPs overhead. We note that high-quality RAG is complementary to the proposed learned memory approach and can be combined with it for further gains.

5 Related works

Databases. (Ahn et al., 2016) use a symbolic knowledge graph, and Borgeaud et al. (2022) introduce *Retro*, augmenting language-model predictions with a large raw-text memory bank. More recently, Zhao et al. (2025) propose replacing long-tail knowledge with retrieval to an external knowledge base by masking retrieved tokens during pretraining. Limitations of these approaches are scalability to large pretraining corpora and low compression rate because the memory stores raw text.

Parametric memories. Wu et al. (2022) propose memorizing transformers, which use nearest-neighbor lookup to sparsely retrieve cached key–value pairs. For efficiency, Eyuboglu et al. (2025) introduce *Cartridges*: KV-memories (similar to what we discussed in Section 2.2) that learn a specific long document as a more runtime-efficient alternative to in-context learning. We find that KV-memories underperform compared to FFN-memories, at least for large-scale memorization. MemSinks (Ghosal et al., 2025) uses a type of FFN-memories, where they dedicate a fraction (e.g., 30%) of FFN neurons per layer to memorization. However, their goal is to throw those parameters away at inference time for privacy and generalization. Our context-dependent memory retrieval is also conceptually related to instruction-following pruning Hou et al. (2025), which selects the most suitable parameters from a larger model based on the task description.

Mixture of experts. Our approach is related to MoEs (Shazeer et al., 2017). Jelassi et al. (2024) show that increasing the number of experts improves memorization while reasoning saturates when active parameters are fixed, aligning with our anchor-memory decomposition to balance reasoning and knowledge. For privacy, Shi et al. (2025) propose *FlexOlmo*, combining a publicly trained anchor expert with exchangeable domain experts trained on private data. Product key memory (PKM) (Lample et al., 2019) can be seen as a sparser MoE that combines two selected expert sets; Huang et al. (2024) improve PKM via tensor decomposition, and Berges et al. (2024) augment subsets of FFN layers with such memory to boost factual benchmarks. These MoE approaches are similar in kind to ours, and we expect that some of our insights may carry over. However, our memory architecture is vastly different, allowing to offload inactive parameters based on the context, give explicit control over memories during both training and inference, and add memories post-hoc.

6 Discussion

Conclusion. We propose pretraining language models with memories to automatically capture long-tail world knowledge in large hierarchical memory banks. Small language models augmented with memory banks match regular transformers with 2× more parameters. Moreover, we propose and analyze several additional potential benefits of this design, including more efficient hardware implementation and enhanced data privacy.

Limitations and future directions. One unexplored area is the study of optimal scaling laws for learning memories. Compute-optimal scaling laws developed for dense training (Hoffmann et al., 2022) are not necessarily applicable to pretraining with memories, since memory parameters are updated less frequently. We also mainly focused on architecture design for memories, and leave architecture search and design of anchor models to future work. Finally, pretraining with memories can benefit multilingual setups (not considered in this work) or other modalities beyond text, such as vision. We leave this investigation for future work.

Acknowledgments

We would like to thank Cheng-Yu Hsieh, Samira Abnar, Stephen Pulman, Karen Khatamifard, Chun-Liang Li, and Rick Chang for their feedback.

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.

Sungjin Ahn, Heeyoul Choi, Tanel Pärnamaa, and Yoshua Bengio. A neural knowledge language model. *ArXiv*, abs/1608.00318, 2016. URL https://api.semanticscholar.org/CorpusID:2600027.

Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12562–12584, 2024.

David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. 2024.

Vincent-Pierre Berges, Barlas Oğuz, Daniel Haziza, Wen-tau Yih, Luke Zettlemoyer, and Gargi Ghosh. Memory layers at scale. *arXiv preprint arXiv:2412.09764*, 2024.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *AAAI Conference on Artificial Intelligence*, 2019. URL https://api.semanticscholar.org/CorpusID:208290939.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

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*, pages 2206–2240. PMLR, 2022.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2924–2936, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1300. URL https://aclanthology.org/N19-1300/.

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.

Damai Dai, Li Dong, Yaru Hao, Zhifang Sui, Baobao Chang, and Furu Wei. Knowledge neurons in pretrained transformers. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8493–8502, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.581. URL https://aclanthology.org/2022.acl-long.581/.

Sabri Eyuboglu, Ryan Ehrlich, Simran Arora, Neel Guha, Dylan Zinsley, Emily Liu, Will Tennien, Atri Rudra, James Zou, Azalia Mirhoseini, et al. Cartridges: Lightweight and general-purpose long context representations via self-study. *arXiv preprint arXiv:2506.06266*, 2025.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

Mor Geva, R. Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. *ArXiv*, abs/2012.14913, 2020. URL https://api.semanticscholar.org/CorpusID:229923720.

Gaurav R Ghosal, Pratyush Maini, and Aditi Raghunathan. Memorization sinks: Isolating memorization during llm training. *arXiv preprint arXiv:2507.09937*, 2025.

David Grangier, Simin Fan, Skyler Seto, and Pierre Ablin. Task-adaptive pretrained language models via clustered-importance sampling. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025. URL https://doi.org/10.48550/arXiv.2410.03735.

Suchin Gururangan, Margaret Li, Mike Lewis, Weijia Shi, Tim Althoff, Noah A. Smith, and Luke Zettlemoyer. Scaling expert language models with unsupervised domain discovery. *ArXiv*, abs/2303.14177, 2023a. URL https://api.semanticscholar.org/CorpusID:257756896.

Suchin Gururangan, Mitchell Wortsman, Samir Yitzhak Gadre, Achal Dave, Maciej Kilian, Weijia Shi, Jean Mercat, Georgios Smyrnis, Gabriel Ilharco, Matt Jordan, Reinhard Heckel, Alex Dimakis, Ali Farhadi, Vaishaal Shankar, and Ludwig Schmidt. OpenLM: a minimal but performative language modeling (lm) repository, 2023b. URL https://github.com/mlfoundations/open_lm/. GitHub repository.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. Training compute-optimal large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Bairu Hou, Qibin Chen, Jianyu Wang, Guoli Yin, Chong Wang, Nan Du, Ruoming Pang, Shiyu Chang, and Tao Lei. Instruction-following pruning for large language models. *arXiv preprint arXiv:2501.02086*, 2025.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Zihao Huang, Qiyang Min, Hongzhi Huang, Defa Zhu, Yutao Zeng, Ran Guo, and Xun Zhou. Ultra-sparse memory network. *arXiv preprint arXiv:2411.12364*, 2024.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Atlas: few-shot learning with retrieval augmented language models. *J. Mach. Learn. Res.*, 24(1), January 2023. ISSN 1532-4435.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

Samy Jelassi, Clara Mohri, David Brandfonbrener, Alex Gu, Nikhil Vyas, Nikhil Anand, David Alvarez-Melis, Yuanzhi Li, Sham M Kakade, and Eran Malach. Mixture of parrots: Experts improve memorization more than reasoning. *arXiv preprint arXiv:2410.19034*, 2024.

Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1147. URL https://aclanthology.org/P17-1147/.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Herve Jegou. *Large memory layers with product keys*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. Latent retrieval for weakly supervised open domain question answering. *arXiv preprint arXiv:1906.00300*, 2019.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.243. URL https://aclanthology.org/2021.emnlp-main.243/.

Hector J. Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'12, page 552–561. AAAI Press, 2012. ISBN 9781577355601.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, Saurabh Garg, Rui Xin, Niklas Muennighoff, Reinhard Heckel, Jean Mercat, Mayee Chen, Suchin Gururangan, Mitchell Wortsman, Alon Albalak, Yonatan Bitton, Marianna Nezhurina, Amro Abbas, Cheng-Yu Hsieh, Dhruba Ghosh, Josh Gardner, Maciej Kilian, Hanlin Zhang, Rulin Shao, Sarah Pratt, Sunny Sanyal, Gabriel Ilharco, Giannis Daras, Kalyani Marathe, Aaron Gokaslan, Jieyu Zhang, Khyathi Chandu, Thao Nguyen, Igor Vasiljevic, Sham Kakade, Shuran Song, Sujay Sanghavi, Fartash Faghri, Sewoong Oh, Luke Zettlemoyer, Kyle Lo, Alaaeldin El-Nouby, Hadi Pouransari, Alexander Toshev, Stephanie Wang, Dirk Groeneveld, Luca Soldaini, Pang Wei Koh, Jenia Jitsev, Thomas Kollar, Alexandros G. Dimakis, Yair Carmon, Achal Dave, Ludwig Schmidt, and Vaishaal Shankar. Datacomp-lm: In search of the next generation of training sets for language models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 14200–14282. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/19e4ea30dded58259665db375885e412-Paper-Datasets_and_Benchmarks_Track.pdf.

Margaret Li, Suchin Gururangan, Tim Dettmers, Mike Lewis, Tim Althoff, Noah A. Smith, and Luke Zettlemoyer. Branch-train-merge: Embarrassingly parallel training of expert language models. *ArXiv*, abs/2208.03306, 2022. URL https://api.semanticscholar.org/CorpusID:251371375.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, 2021. URL https://api.semanticscholar.org/CorpusID:230433941.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

Xinxi Lyu, Michael Duan, Rulin Shao, Pang Wei Koh, and Sewon Min. Frustratingly simple retrieval improves challenging, reasoning-intensive benchmarks. *arXiv preprint arXiv:2507.01297*, 2025.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in gpt. *Advances in neural information processing systems*, 35:17359–17372, 2022.

Meta AI. Llama 3.2 model card. 2024. URL https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_2/.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *EMNLP*, 2018.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.

Hadi Pouransari, Chun-Liang Li, Jen-Hao Rick Chang, Pavan Kumar Anasosalu Vasu, Cem Koc, Vaishaal Shankar, and Oncel Tuzel. Dataset decomposition: faster llm training with variable sequence length curriculum. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2025. Curran Associates Inc. ISBN 9798331314385.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In Jian Su, Kevin Duh, and Xavier Carreras, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL https://aclanthology.org/D16-1264/.

Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics*, 11: 1316–1331, 2023. doi: 10.1162/tacl_a_00605. URL https://aclanthology.org/2023.tacl-1.75/.

Siva Reddy, Danqi Chen, and Christopher D. Manning. CoQA: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266, 2019. doi: 10.1162/tacl_a_00266. URL https://aclanthology.org/Q19-1016/.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL https://arxiv.org/abs/1908.10084.

Adam Roberts, Colin Raffel, and Noam Shazeer. How much knowledge can you pack into the parameters of a language model? In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5418–5426, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.437. URL https://aclanthology.org/2020.emnlp-main.437/.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: an adversarial winograd schema challenge at scale. *Commun. ACM*, 64(9):99–106, August 2021. ISSN 0001-0782. doi: 10.1145/3474381. URL https://doi.org/10.1145/3474381.

Rulin Shao, Jacqueline He, Akari Asai, Weijia Shi, Tim Dettmers, Sewon Min, Luke Zettlemoyer, and Pang Wei W Koh. Scaling retrieval-based language models with a trillion-token datastore. *Advances in Neural Information Processing Systems*, 37:91260–91299, 2024.

Julia Shaw. *The memory illusion: Remembering, forgetting, and the science of false memory*. Random House, 2016.

Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Weijia Shi, Akshita Bhagia, Kevin Farhat, Niklas Muennighoff, Pete Walsh, Jacob Morrison, Dustin Schwenk, Shayne Longpre, Jake Poznanski, Allyson Ettinger, et al. Flexolmo: Open language models for flexible data use. *arXiv preprint arXiv:2507.07024*, 2025.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Mariya Toneva, Alessandro Sordoni, Rémi Tachet des Combes, Adam Trischler, Yoshua Bengio, and Geoffrey J. Gordon. An empirical study of example forgetting during deep neural network learning. *ArXiv*, abs/1812.05159, 2018. URL https://api.semanticscholar.org/CorpusID:55481903.

V0.9. Llm foundry v0.7.0. https://github.com/mosaicml/llm-foundry.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

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

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

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.

Yunzhi Yao, Shaohan Huang, Li Dong, Furu Wei, Huajun Chen, and Ningyu Zhang. Kformer: Knowledge injection in transformer feed-forward layers. In Wei Lu, Shujian Huang, Yu Hong, and Xiabing Zhou, editors, *Natural Language Processing and Chinese Computing*, pages 131–143, Cham, 2022. Springer International Publishing. ISBN 978-3-031-17120-8.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

Linxi Zhao, Sofian Zalouk, Christian K Belardi, Justin Lovelace, Jin Peng Zhou, Kilian Q Weinberger, Yoav Artzi, and Jennifer J Sun. Pre-training large memory language models with internal and external knowledge. *arXiv preprint arXiv:2505.15962*, 2025.

Appendix Contents

A Training details

In this section, we provide the training details of all experiments. All experiments in the paper are conducted using the OpenLM (Gururangan et al., 2023b) repository[1]. The training data is DCLM-Baseline (Li et al., 2024). Rotary Positional Embedding (RoPE) (Su et al., 2024) is used to encode positions in queries and keys before the attention module, with a frequency of 100,000. For all training jobs, anchor and memory parameters are stored in BFloat16 precision and trained with PyTorch Fully Sharded Data Parallelism (FSDP) using AdamW optimizer ($\beta_1 = 0.9, \beta_2 = 0.95$) and gradient clipping to 1.0.

A.1 Baseline model training

The baseline models, corresponding to rows A1, B1, and C2 in Table 1, are transformers with the block shown in Figure 13 and configurations detailed in Table 4, Table 5, and Table 6, respectively. Each model is trained from scratch with $2^{40} \simeq 1.1$T tokens, using a context length of 8,192 and the dataset decomposition approach (Pouransari et al., 2025) with a global batch size of 1M tokens. The learning rate follows a cosine schedule with 10k warmup steps, a maximum value of 5×10^{-3}, and a cooldown value of 5×10^{-5}. Weight decay is set to 0.05.

Model	OpenLM-160M
Num layers	35
Hidden dim	512
Num heads	12
Per head dim	32
FFN inner dim	2,048
Head-Embedding	tied
Num params	163,510,016

Table 4 OpenLM-160M

Model	OpenLM-410M
Num layers	24
Hidden dim	1,024
Num heads	16
Per head dim	64
FFN inner dim	2,816
Head-Embedding	separate
Num params	411,665,408

Table 5 OpenLM-410M

Model	OpenLM-1B
Numb layers	24
Hidden dimension	2,048
Num heads	16
Per head dim	128
FFN inner dim	5,632
Head-Embedding	separate
Num params	1,439,893,504

Table 6 OpenLM-1B

A.2 Memory model training

For all memory learning jobs, we first perform clustering as described in Appendix C. After identifying the corresponding level 4 cluster ID of each document (a number between 1 and 16^4), we pack documents from the same cluster into sequences of length 2,048, globally shuffle them, and add the cluster ID as a prefix to each sequence. During training, the cluster ID is simply obtained by separating the first token from the rest of the tokens, which represent the actual data. Note that shallower-level cluster IDs can be inferred from the level-4 cluster ID due to the nested structure of our clustering. Each sequence of length 2,048 can contain subsequences from different documents (within the same cluster), separated by a special `EOT` token. The attention mask is restricted to each document to avoid cross-document attention. The global batch size for all jobs is 2M tokens (1,024 sequences of length 2,048 each), except for the 1.4B model (row C2 in Table 1), where we use a global batch size of 4M to improve GPU utilization.

When the anchor model is frozen, we train memories asynchronously (one job for the memories of each level-1 subtree, resulting in a total of 16 jobs) and merge the checkpoints afterward.

In addition, we use a cosine learning rate schedule with a maximum value of 10^{-4} and a cool-down value of 10^{-5}. When training for 2^{40} tokens, we use 10k warmup steps, and for all other trainings with different numbers of tokens, we keep the same warmup ratio. We found that warmup has minimal effect on the performance of memory learning. We use a weight decay value of 10^{-3}, which we found to improve the performance of memory learning, consistent with its goal of memorization.

A.3 Additional details of memory type and size ablations

The total number of seen tokens for the memory type and size experiments in Figure 3 is 2^{38}. For all types of memories considered, the memories correspond to level-2 clusters.

[1] https://github.com/mlfoundations/open_lm

For the experiments in Figure 3d, the total seen tokens are 2^{36}, 2^{38}, 2^{40}, and 2^{41} for memories at levels 1, 2, 3, and 4, respectively. Since deeper memories are updated less frequently, we increased the total seen token budget for those cases.

A.4 Additional details of ablation on memory configuration

For the experiments in Figure 4 (corresponding to Table 7), we use 2^{40} and 2^{41} total seen tokens, corresponding to memory bank sizes of 4.6B and 18.7B, respectively.

Memory config				Bank size	Fetched size (M)	Common-Knowledge (%)	Specific-Knowledge (%)
level 1	level 2	level 3	level 4				
0	0	0	0	0	0	45.9	34.1
0	16	4	1	4.6B	1.1	46.5	37.8
98	42	18	0		8.5	46.5	38.6
256	64	16	0		18.1	47.4	39.2
768	96	12	0		47.1	47.6	40.2
1792	112	7	0		102.7	48.3	41.6
5376	0	0	0		289	49.7	43.2
256	64	16	4	18.7B	18.1	47.2	40.1
328	156	74	0		30	46.3	40.9
1216	164	22	3		75.5	47.8	41.6
4096	320	17	2		238.4	49.2	44.6

Table 7 Ablation on different hierarchical memory configurations corresponding to results shown in Figure 4a. For all experiments, the anchor model is the 160M model (row A1 in Table 1), frozen during memory learning. Memories are trained for 1.1T and 2.2T tokens when the memory bank size is 4.6B and 18.7B, respectively.

A.5 Additional details of co-training memories and anchor model

For the experiments in Table 1, we denote the total number of seen tokens, which is either 2^{40} or 2^{41}. For the A4 model in Table 1, where both memory and anchor model parameters are trained together from scratch, we use a cosine learning rate schedule with a maximum value of 10^{-3} and a cool-down value of 10^{-5}, with 10k warmup steps and weight decay set to 0.1.

A.6 Additional details of memory learning on open-weight models

For all experiments in Table 2, we use a total of 2^{40} tokens. All models are initialized from their public pretrained checkpoints, and these anchors remain frozen in all experiment except cotraining. We use a memory config of (512, 128, 32, 0) for Gemma and Qwen (which results in higher memory parameters for Qwen because it has a higher internal dimension and more transformer blocks) and (768, 256, 32, 0) for Llama. These numbers result in a roughly 10% memory to anchor model parameter ratio. We follow the optimizer setup of Section A.2, except that we do not restrict cross-attention because the baseline models were not all trained this way and reduce warmup steps to 5k. When we cotrain the Qwen anchor with memories, we reduce the batchsize to 1M tokens (512 sequences of length 2,048 each) to accomodate for GPU VRAM and increase warmup steps to 10k to reduce the risk of interference with the pretrained parameters.

A.7 Additional details of memory-to-model size experiment

Here we provide implementation details for the experiments in Figure 4b.

We summarize the architecture of six anchor models and their corresponding memories for this experiment in Table 8. The anchor models are first trained for 2^{38} tokens, then frozen, and their memories are trained for another 2^{39} tokens. For the anchor models, we use a cosine learning rate schedule with a maximum learning rate of 10^{-3} and a cool-down value of 10^{-5}. Weight decay is set to 0.1. We then freeze the anchor model to

train the memories. For this particular experiment, memories are trained with a constant learning rate of 10^{-4} to demonstrate convergence of memory parameters, even without learning rate annealing.

Anchor model			Memory			Runtime params
Name	Num layers	Num params	Config	Fetch size	Bank size	
260M	12	257,475,584	`(3840,336,6,0)`	154,165,248	6,341,787,648	411,640,832
320M	17	321,721,344	`(1445,256,8,0)`	89,250,816	6,341,246,976	410,972,160
350M	19	347,419,648	`(870,226,9,0)`	64,496,640	6,341,099,520	411,916,288
370M	21	373,117,952	`(384,200,10,0)`	38,320,128	6,341,787,648	411,438,080
385M	22	385,967,104	`(264,94,16,0)`	25,276,416	6,341,001,216	411,243,520
410M	24	411,665,408	`(0,0,0,0)`	0	0	411,665,408

Table 8 Anchor models and their corresponding memory configurations for the memory-to-model size experiment in Figure 4b. The anchor model architecture and memory configuration are designed to keep the total runtime number of parameters and the memory bank size (almost) fixed. The anchor model architecture uses a hidden dimension of 1,024 and 16 heads for all rows.

B Evaluation

We use LLM-Foundry (V0.9) as the evaluation framework. To obtain a stable signal, we only include benchmarks for which the baseline 410M model (B1 in Table 1) performs better than random and the ratio of standard deviation to mean performance is less than 0.5. Below is a detailed description of the 13 tasks we use to evaluate pretrained language models in this work:

- Lambada-OpenAI (Paperno et al., 2016) with 5,153 samples, evaluated 0-shot, is of type `language modeling`, with a random baseline performance equal to 0%.
- BoolQ (Clark et al., 2019) with 3,270 samples, evaluated 0-shot, is of type `multiple choice`, with a random baseline performance equal to 50%.
- SQuAD (Rajpurkar et al., 2016) with 10,570 samples, evaluated 3-shots, is of type `language modeling`, with a random baseline performance equal to 0%.
- Winograd (Levesque et al., 2012) with 273 samples, evaluated 3-shots, is of type `schema`, with a random baseline performance equal to 50%.
- WinoGrande (Sakaguchi et al., 2021) with 1,267 samples, evaluated 5-shots, is of type `schema`, with a random baseline performance equal to 50%.
- CoQA (Reddy et al., 2019) with 7,983 samples, evaluated 0-shot, is of type `language modeling`, with a random baseline performance equal to 0%.
- Hellaswag (Zellers et al., 2019) with 10,042 samples, evaluated 0-shot, is of type `multiple choice`, with a random baseline performance equal to 25%.
- ArcEasy (Clark et al., 2018) with 2,376 samples, evaluated 3-shots, is of type `multiple choice`, with a random baseline performance equal to 25%.
- ArcChallenge (Clark et al., 2018) with 1,172 samples, evaluated 3-shots, is of type `multiple choice`, with a random baseline performance equal to 25%.
- TriviaQA (Joshi et al., 2017) with 3,000 samples, evaluated 3-shots, is of type `generation task with answers`, with a random baseline performance equal to 0%.
- PIQA (Bisk et al., 2019) with 1,838 samples, evaluated 0-shot, is of type `multiple choice`, with a random baseline performance equal to 50%.

- OpenBookQA (Mihaylov et al., 2018) with 500 samples, evaluated 10-shots, is of type `multiple choice`, with a random baseline performance equal to 25%.
- NaturalQuestions (Lee et al., 2019; Kwiatkowski et al., 2019) with 2,655 samples used by Liu et al. (2024), evaluated 0-shot, is of type `generation task with answers`, with a random baseline performance equal to 0%.

In Appendix G, we present our analysis dividing the above tasks into two groups: common-knowledge (CK) and specific-knowledge (SK).

When evaluating models with fetched memory, we retrieve memory based on the *question* for multiple choice tasks, the *context* for language modeling tasks, and the portion of the text that is common across all choices for the schema tasks. To compute perplexity (for the Wiki-En dataset) we use the full document to retrieve memory.

Elements atomic number: We include a task on predicting the atomic number of different elements. The model completes a prompt in the form of "The atomic number of {...} is," where {...} is replaced with the name of an element (from a total of 118). The model's generation is then processed to extract integer numerical values, and a response is accepted if it matches the actual atomic number. Random baseline performance for this task is 0%. We use the prompt, including the element name, for memory retrieval.

The elements atomic number evaluation is particularly interesting because each query has a specific keyword: the element name. This allows us to count the frequency of each element's name in the dataset (as shown in Figure 1) to analyze model performance as a function of knowledge scarcity in the pretraining dataset. We note, however, that this analysis has some minor caveats. For example, the word "lead" is both the name of an element and an English verb. In addition, some elements have multiple names; for instance, "Natrium" is an accepted alias for "Sodium". In our frequency calculations, we count all acceptable aliases of each element name.

C Data clustering details

We cluster the training set with hierarchical clustering (Grangier et al., 2025). We build a clustering tree: each node in the tree is associated with a cluster centroid. The examples traverse the tree from top to bottom, selecting the node corresponding to the closest centroids among the current node's children.

Before training the clustering tree, we segment our dataset into non-overlapping 2,048 token windows and compute sentence-BERT embedding for every window. We rely on the sentence-BERT `MiniLM-L6-v2` model (Reimers and Gurevych, 2019). This process associates each segment of text with a 384 dimensional vector.

The training of the tree proceed from root to leaves. Iteratively, a new level is built by applying k-means to a subset of the examples belonging to each node. We built a tree of depth up to 4, always splitting nodes in 16 clusters. For k-means, we normalize the Euclidean norm of the vectors prior to clustering. We train the model via Expectation Maximization using k-means++ initialization (Arthur and Vassilvitskii, 2006). At each step, we sample 6,400 new examples. With 20 steps, we visit 128k examples. To ensure a cluster distribution close to uniform, we monitor the cluster sizes at each assignment steps. If a cluster is larger than our balancing limit ($0.094 \simeq 1.5 \times 1/16$), we split evenly at random its assignments with the smallest cluster, as suggested by Jegou et al. (2010).

D Tokens per parameter in memory learning

While scaling laws exist for regular language model pretraining that identify the training budget–optimal choice for token-per-parameter (TPP) (Hoffmann et al., 2022), these laws may not hold for memory learning due to its different learning dynamics. Here, we experiment with the effect of TPP when learning memories.

We consider the `(0,0,16,0)` memory configuration on top of a pretrained 160M anchor model (row A1 in Table 1). This corresponds to 4,096 memories, each with 860,160 parameters, for a total of 3.5B parameters.

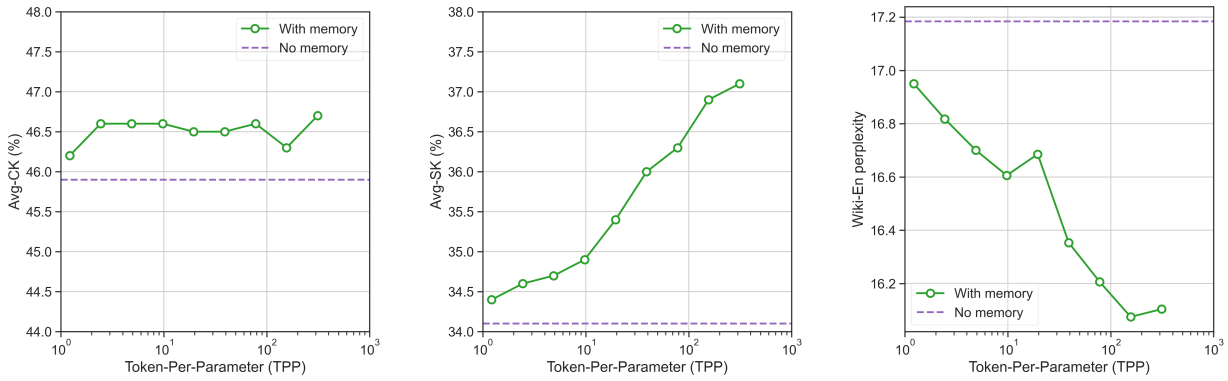

Figure 7 Common-Knowledge (Avg-CK), Specific-Knowledge (Avg-SK), and perplexity on Wiki-En as a function of tokens-per-parameter (TPP). The anchor model is a pretrained 160M model (row A1 in Table 1) and is kept frozen during memory learning.

We freeze the anchor model and train the memories with $2^{32}, 2^{33}, \ldots, 2^{40}$ total tokens, corresponding to TPP values ranging from $\simeq 1$ to 312. Results are shown in Figure 7.

In regular language modeling, knowledge (as in conditional likelihoods) is picked up into parameters when it, and similar content, is repeated often (with many TPP) and receives constructive gradient updates, and forgotten if it is rare and gets destructive gradient updates by dissimilar content (Ghosal et al., 2025). This means that long-tail information is forgotten after it is seen in a batch because the following batches send destructive gradient updates. What lasts in joint parameters is common knowledge, which occurs more often and with more aligned gradients. We demonstrate this effect in Figure 1.

In the proposed memory learning method, however, memory parameters are shielded in that they are activated and updated only on sequences of a similar topic. This both reduces the times where knowledge can be overwritten and the dissimilarity of possible other gradients. On average, memory parameters corresponding to level l are updated 16^l times less often than anchor parameters. For the setup considered above with memory configuration `(0,0,16,0)`, where all memories are at level 3, memory parameters receive one update for every 4,096 sequences in the training set.

Specific knowledge stored in deep memories is shielded from catastrophic forgetting (Toneva et al., 2018), because, unlike regular parameters, it is not overwritten frequently by destructive gradients from unrelated content. Instead, deep memory parameters are only activated when there are constructive gradients of similar content in their clusters, so that they *memorize* this specific knowledge. This behavior is shown in Figures 7b and 7c, where specific-knowledge benchmark accuracy and wikipedia perplexity steadily improve with increasing TPP. The last point corresponds to a total of 1.1 trillion seen tokens (TPP = 312). Due to computational constraints, we did not scale TPP further.

E Fetched memory vs generic memory (additional results)

In Figure 6a, we showed the performance of the model using fetched and generic memories throughout training, both when anchor parameters are frozen and when they are learnable. A key observation is that when we allow the anchor parameters to be co-trained with the memory bank, we obtain greater improvement compared to the case where the anchor model is frozen (e.g., Avg-SK improves from 39.2% to 40.3% as shown in Table 1). This improvement can be attributed to two factors: 1) when co-training, the anchor model learns to adapt to the memories more effectively compared to when it is frozen, and 2) the anchor model is exposed to additional tokens, so its performance improves simply due to more training. The latter effect should be minor if the anchor model is already over-trained (i.e., trained with a high TPP) during pretraining and has reached performance saturation, meaning it no longer benefits from additional training.

We also track the gap between the performance of the anchor model using generic memories (a fixed set of

memory parameters independent of the input context) and fetched memories throughout co-training, as shown in Figure 8b and Figure 8d for the Avg-SK and Wiki-En perplexity metrics, respectively. We observe that the performance gap between fetched and generic memories grows over time (despite using a cosine learning rate schedule), indicating that longer co-training benefits fetched memories more than generic ones. This is expected, as the memory bank has significantly more parameters (4.6B in this example) compared to the anchor model (160M) and a single generic memory (18M), and thus benefits more from extended training.

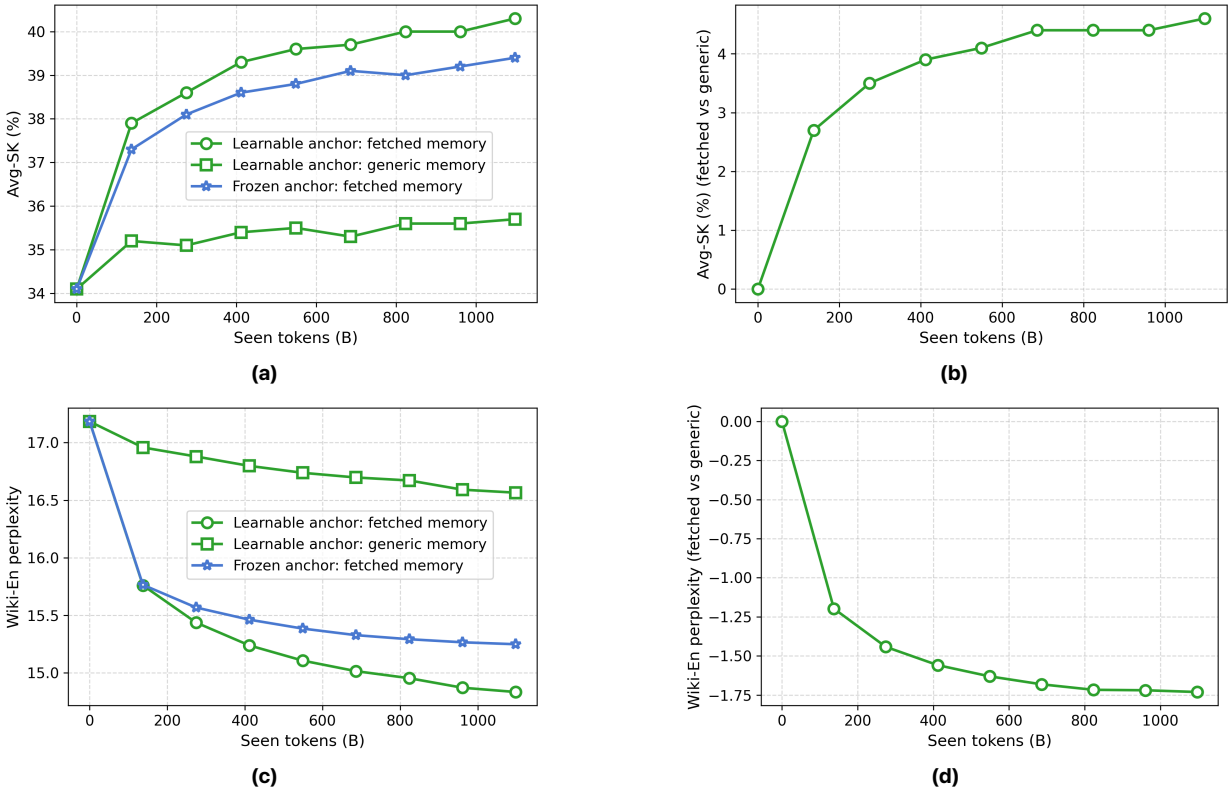

Figure 8 Additional results for the experiment setup of Figure 6a. Performance improvements from co-training the anchor model and memories, corresponding to row A2 in Table 1. We also show performance when the anchor model is frozen, corresponding to row A3 in Table 1. **(a)** Avg-SK using fetched and generic memories, **(b)** Avg-SK gap between fetched and generic memories, which grows as training progresses, **(c)** Wiki-En perplexity using fetched and generic memories, **(d)** Wiki-En perplexity gap between fetched and generic memories, which widens as training progresses.

F Memory location

So far, we have considered memory parameters to be uniformly distributed across the layers of the model. Meng et al. (2022) showed that model knowledge is mainly captured in the middle layers. We use our memory learning setup to further explore this hypothesis by considering different distributions of memory parameters across layers, as shown in Figure 9.

Starting from a baseline memory with configuration `(0,64,0,0)`, trained on top of a pretrained and frozen 160M model (row A1 in Table 1), corresponding to the level-2 models in Figure 3d, we study the effect of distributing memory parameters non-uniformly across the model. We consider three setups—early, mid, and late—where the same number of memory parameters as the uniform baseline are applied only to the first, middle, or last 10 layers of the anchor model (see Figure 9). As discussed in Section A.3, for these experiments the anchor model is frozen, and memories are trained for a total of 2^{38} tokens.

Results for each memory placement are shown in Table 9. Consistent with the observations of Lample et al. (2019), using memories in the early layers of the anchor model is less effective than using them uniformly (the default setup) or in the deeper layers.

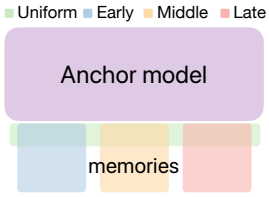

Figure 9 Different distribution of memory along the layers of the anchor model.

Memory location	Avg-CK (%) ↑	Avg-SK (%) ↑	Elements atomic number (%) ↑	Wiki-En Pplx ↓
No Memory	45.9	34.1	1.7	17.2
Uniform	46.7	**36.8**	14.4	**16.0**
Early	45.8	34.9	2.5	16.6
Middle	46.1	35.4	1.7	16.5
Late	**46.9**	**36.8**	**20.3**	16.1

Table 9 Comparing effectiveness of memories with different distribution over the depth of anchor model, which is kept frozen.

G What tasks benefit more from memories

In Figure 1, using the atomic number prediction task, we showed that memories can significantly improve performance on tasks requiring long-tail knowledge. To extend this concept to commonly used benchmarks for evaluating pretrained models, we introduce an approximate, quantitative measure of the degree of knowledge specificity for each of the 13 evaluation benchmarks described in Appendix B. For each dataset, we randomly sample 100 entries and prompt GPT-4 (Achiam et al., 2023) to estimate the education level (as a proxy for knowledge specificity) at which a typical person would have acquired the knowledge needed to answer each question. The average of these ratings is used as the knowledge specificity score of the task.

Specifically, we ask the model to rate each question based on the amount of knowledge required, using the following prompt:

> Given the following prompt and answer, what facts should a human know in order to answer the question correctly? What phase of life will a typical person know all required facts? Your response should be in the format of an integer between 0 and 5 based on the following scale:
> 0: Only language understanding, all required information is in the context
> 1: Commonsense facts learned through sensory experiences in childhood
> 2: Facts learned in elementary school
> 3: Facts learned in middle school
> 4: Facts learned in high school
> 5: More specific facts learned later in life
> Respond with only a single integer and nothing else.

In Figure 10, we plot the accuracy difference between fetched memories and generic memories against the knowledge specificity score for each benchmark, using the 1.4B model trained with memories (row C2 in Table 1). The plot shows a clear positive correlation between knowledge specificity and performance improvement from fetched memories.

We further group the datasets into six common-knowledge (purple) and seven specific-knowledge (green) tasks using a knowledge specificity score threshold of 2.0. On average, fetched memories improve specific-knowledge (Avg-SK) task performance by 3.6 points, while performance on common-knowledge (Avg-CK) tasks remains comparable (64.4 vs. 64.5). Notably, both CK and SK performance show even greater improvement compared to the baseline model (row C1 in Table 1): Avg-CK improves from 61.2% to 64.5%, and Avg-SK improves from 49.7% to 54.9%.

Additionally, we show the same analysis for the 160M (row A2 in Table 1) and 410M (row B2 in Table 1) models trained with memories in Figure 11 and Figure 12, respectively.

H Retrieval augmented generation detail

In this section, we provide further details for the experiments in Section 4. We consider two datasets to retrieve documents from: DCLM-Baseline and English Wikipedia 2022, with 6.5 million and 3.2 billion documents, respectively. The average document length in DCLM, using our `EleutherAI/gpt-neox` (Black et al., 2022)

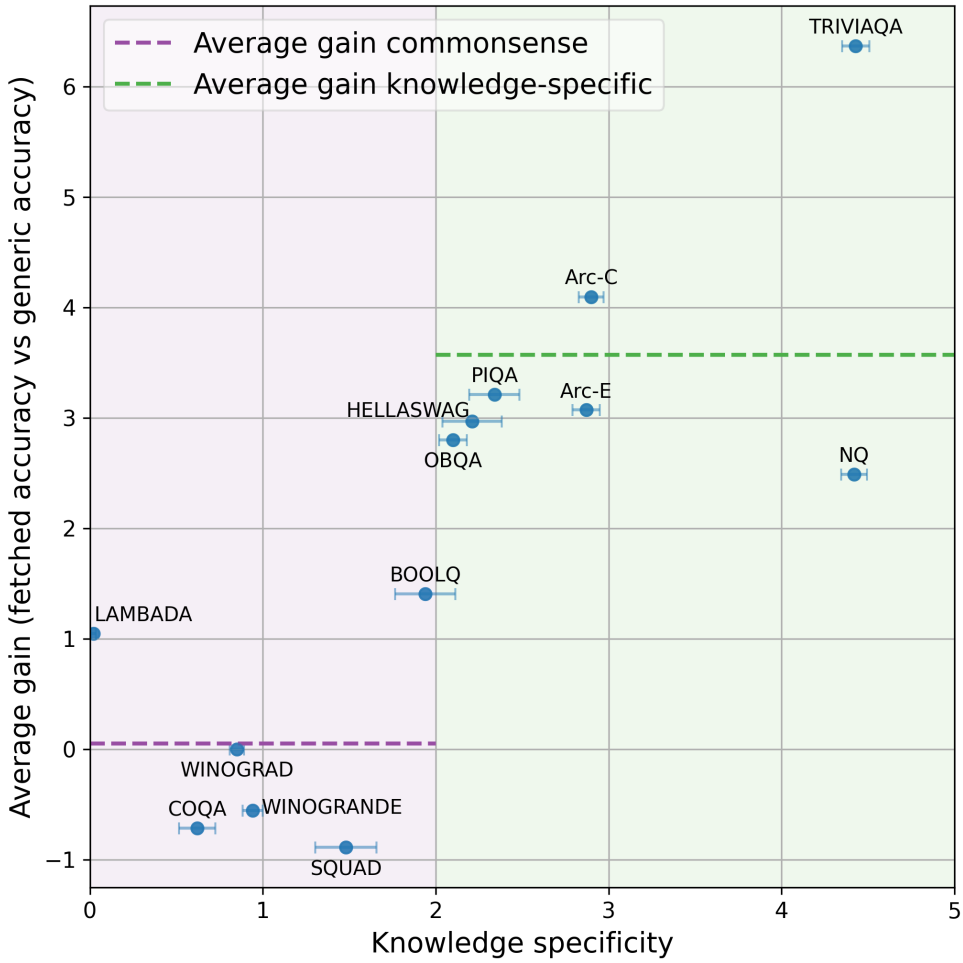

Figure 10 Fetched memories improve performance on knowledge-intensive benchmarks. Accuracy gain (fetched memory vs. generic memory) for the 1.4B model (row C2 in Table 1) as a function of the knowledge specificity score of each benchmark. Knowledge specificity is determined by GPT-4 ratings of 100 sampled entries per dataset, and error bars reflect the standard error of the mean. The positive correlation highlights the value of fetched memories for knowledge-intensive tasks. Note that this plot shows the improvement of fetched memories compared to generic memories; the improvement is even greater when comparing fetched memories with the baseline model without memory (row C1 in Table 1).

tokenizer, is 1,309 tokens (computed from a random 10% subset of the dataset), while for Wikipedia it is 723 tokens. We set the max sequence length at evaluation to 2,560.

As a reference point, we report FLOPs associated with each approach. For RAG models, we use the average sequence length and compute the additional FLOPs on top of a 1024-token context. When using memory, the context length does not increase, but additional memory parameters are fetched and added to the anchor model, increasing runtime FLOPs by $\simeq 10\%$.

For these experiments, we compare augmenting the anchor model with: 1) fetched memory from the learned bank of memories, versus 2) raw documents retrieved from the same dataset the memories were trained on. For high-quality RAG performance, many factors matter, including the instruction-following and long-context capability of the LLM, the retriever's quality, the quality of the datastore, and additional techniques such as self-reflection (Asai et al., 2024).

In this study, we mainly aim to compare learned memories (ours) against contextual memories without additional confounders. We therefore call our retrieval-augmented generations *vanilla* RAG. We use the same

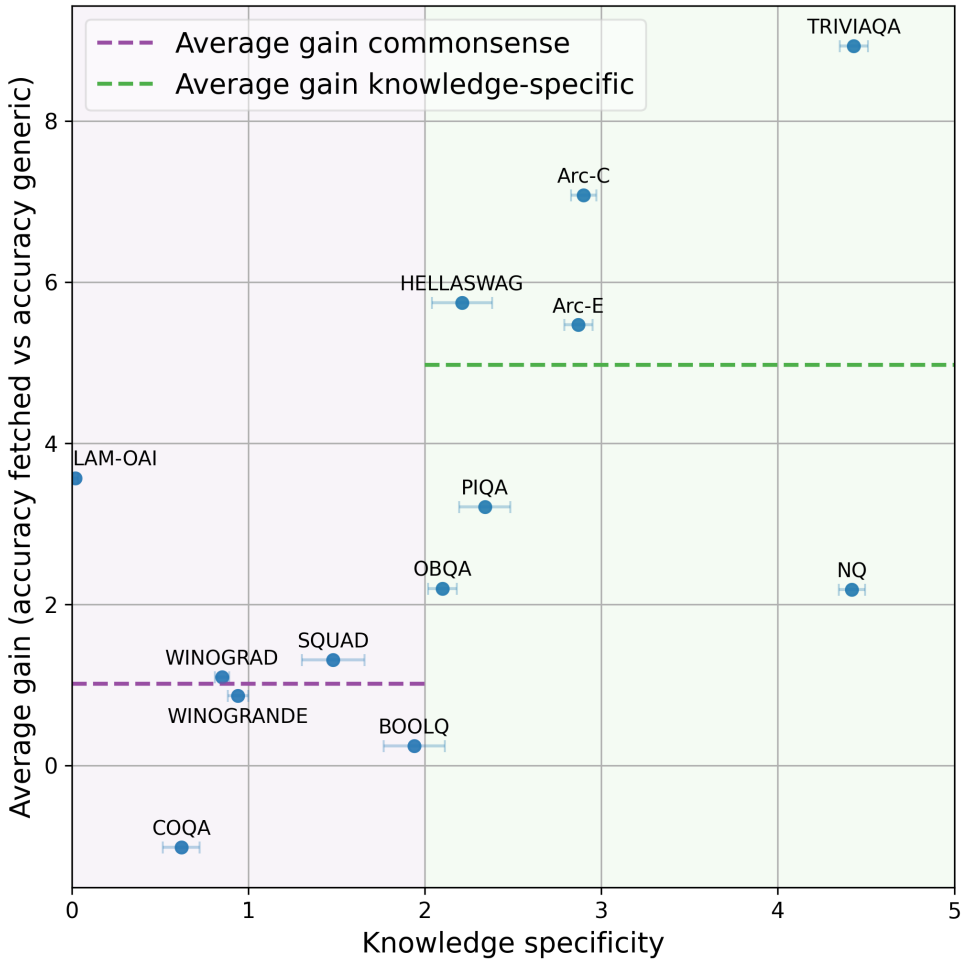

Figure 11 Fetched memories improve performance on knowledge-intensive benchmarks. Accuracy gain (fetched memory vs. generic memory) for the 410M model (row B2 in Table 1) as a function of the knowledge specificity score of each benchmark. Knowledge specificity is determined by GPT-4 ratings of 100 sampled entries per dataset, and error bars reflect the standard error of the mean. The positive correlation highlights the value of fetched memories for knowledge-intensive tasks. Note that this plot shows the improvement of fetched memories compared to generic memories; the improvement is even greater when comparing fetched memories with the baseline model without memory (row B1 in Table 1).

retrieval mechanism for both learned memories and RAG: given a query, we identify the context as described in Appendix B. Using the Sentence-BERT (Reimers and Gurevych, 2019) `all-MiniLM-L6-v2` embedding model, we first determine the closest level-3 cluster in DCLM (comparing against 4096 centroids). Note that the models with memory in Table 3 are also trained with hierarchical memory configurations up to level 3. We then retrieve the nearest-neighbor (NN) document from within that level 3 cluster (on average $\simeq$ 750k documents, given 3B/4096). This document is then added to the context. To avoid confounders from few-shot complexity in RAG, all tasks are run in a 0-shot setup. We experiment with the 160M, 410M, and 1.4B models, corresponding to rows A1, B1, and C1 in Table 1.

As shown in Table 3, a vanilla NN retrieval from DCLM does not improve baseline model performance for either common-knowledge or specific-knowledge benchmarks. We argue this is mainly due to the low quality of DCLM (a pretraining dataset) when used as a datastore for RAG. Recently, Lyu et al. (2025) showed that with careful filtering, RAG quality can be improved when using web-scale datastores.

To demonstrate the effect of datastore quality here, we also use English Wikipedia to retrieve higher-quality

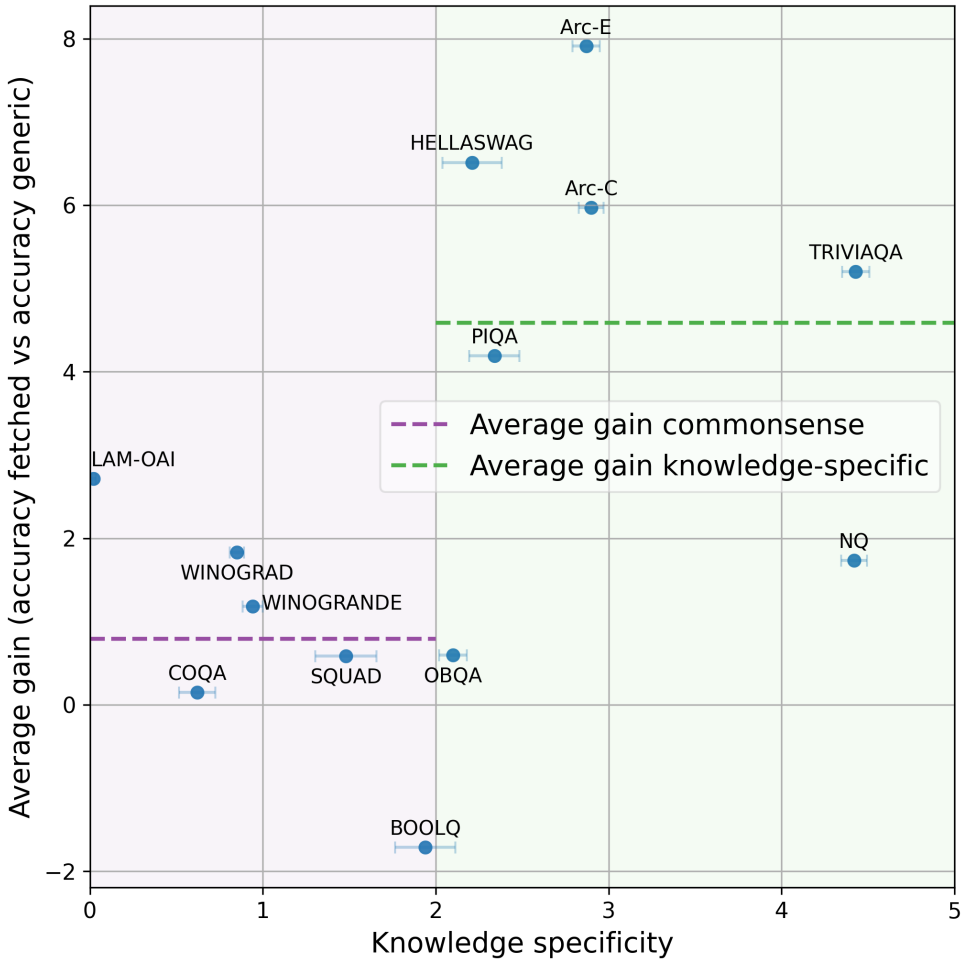

Figure 12 Fetched memories improve performance on knowledge-intensive benchmarks. Accuracy gain (fetched memory vs. generic memory) for the 160M model (row A2 in Table 1) as a function of the knowledge specificity score of each benchmark. Knowledge specificity is determined by GPT-4 ratings of 100 sampled entries per dataset, and error bars reflect the standard error of the mean. The positive correlation highlights the value of fetched memories for knowledge-intensive tasks. Note that this plot shows the improvement of fetched memories compared to generic memories; the improvement is even greater when comparing fetched memories with the baseline model without memory (row A1 in Table 1).

documents for the given context with the same Sentence-BERT model. Results in Table 3 show that RAG-Wiki improves baseline performance on specific-knowledge benchmarks (e.g., from 46.9 to 49.2 for the 1.4B model). However, for common-knowledge benchmarks, RAG-Wiki does not improve over baseline. By contrast, using learned memories (with $\simeq 10\%$ additional parameters relative to baseline) improves performance on both common-knowledge and specific-knowledge benchmarks with lower runtime FLOPs overhead.

Finally, we note that high-quality RAG is complementary to the proposed learned memories and can further enhance performance.

I Memory augmented transformer architecture

In this section, we provide additional detail on different memory augmented transformer architectures that we considered.

LoRa-Memories: The transformer block with SwiGLU FFN has seven linear layers, as shown in Figure 13.

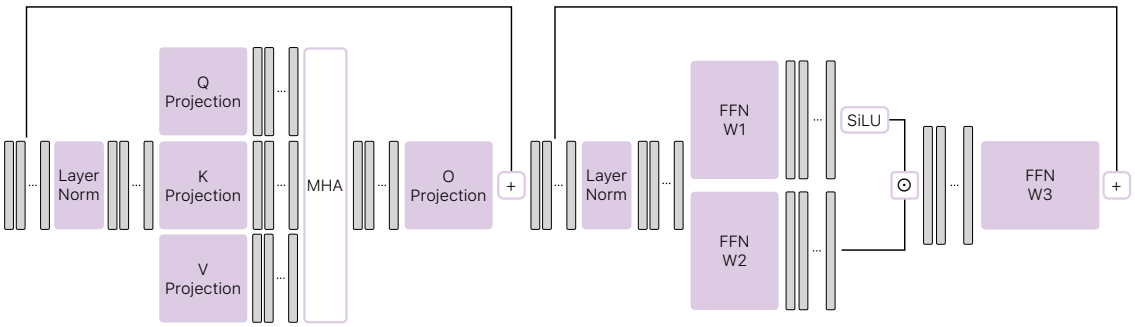

Figure 13 Base architecture of a transformer block with SwiGLU FFN layer.

We can augment any subset of these with low-rank memories. To avoid exhaustive search, we group the linear layers into three categories based on their role in the transformer block: 1) query and key projection layers, 2) value and output projection layers, and 3) FFN linear layers.

Accordingly, we define three types of LoRa memories: LoRa-QK (shown in Figure 14), LoRa-OV (shown in Figure 15), and LoRa-FFN (shown in Figure 16). Each LoRa consists of two low-rank matrices, A and B. The target linear layer W is patched additively as $W + BA$. The size of memories is determined by the rank r of matrices A and B. Note that for a model with hidden dimension d, inner FFN dimension d_f, per attention head dimension d_h, h heads, and l layers the size of fetched memory with LoRa type is as follows:

- LoRa-QK memory size: $2rl(d + hd_h)$
- LoRa-OV memory size: $2rl(d + hd_h)$
- LoRa-FFN memory size: $3rl(d + d_f)$

For graceful initialization, so that memories initially have no effect, we initialize B with zeros, as suggested in the original work (Hu et al., 2022), and A with a uniform Kaiming distribution in all cases. Additionally, we found that a scaling factor of $\alpha = 2$ works best for the LoRa memories considered here.

Aligned with previous observations that knowledge in transformers is primarily stored in FFN layers (Geva et al., 2020; Dai et al., 2022; Yao et al., 2022), we also find that LoRa-FFN mostly outperforms alternative LoRa memories for the same number of learnable parameters, as shown in Figure 3.

KV-Memories: With these memories, we learn additional key and value vectors to augment the input-dependent keys and values. The input-dependent query vectors cross-attend to the learned key and value vectors, and their results are added to the output of multi-head attention. Note that we do not apply causal masking when attending to the learned keys and values. Additionally, we found that KV memories are slightly more effective when used without positional encoding. To ensure no memory effect at initialization, we initialize the learned value vectors with zeros and the learned key vectors with a truncated normal distribution, consistent with other model parameters.

The size of KV memories is determined by the number of key-value vectors (r), as shown in Figure 17, and can be calculated as follows:

- KV memory size: $2rlhd_h$

FFN-Memories: for FFN memories, we directly expand the inner dimension of the three linear layers in the SwiGLU FFN as shown in Figure 18. Similar to other memory types, we initialize memories such that at the beginning of training they have no effect. Therefore, we initialize W1 and W2 FFN memories with truncated normal, and W3 FFN memory with zeros. The size of the FFN memory is determined by their inner dimension and can be calculated as follows:

- FFN memory size: $3rld$

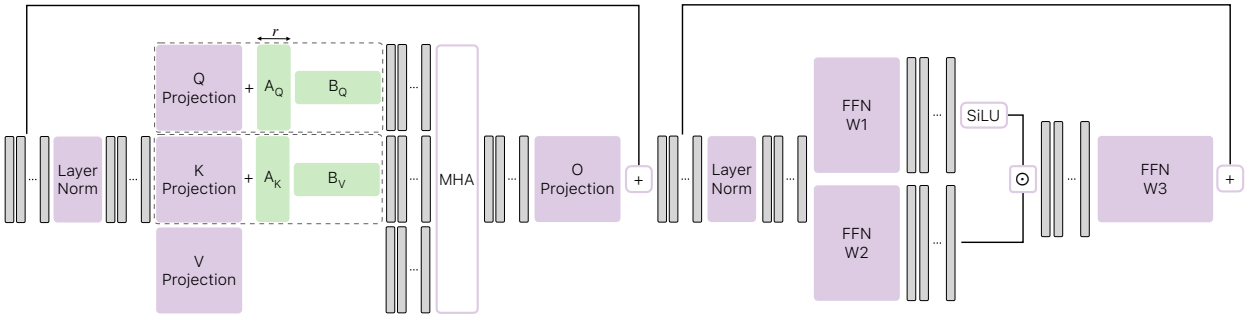

Figure 14 A transformer block with LoRa memories on queries and keys projection layers.

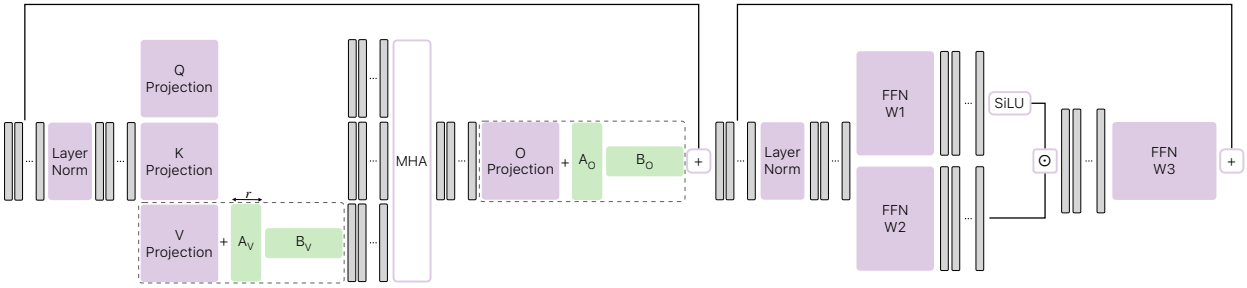

Figure 15 A transformer block with LoRa memories on values and output projection layers.

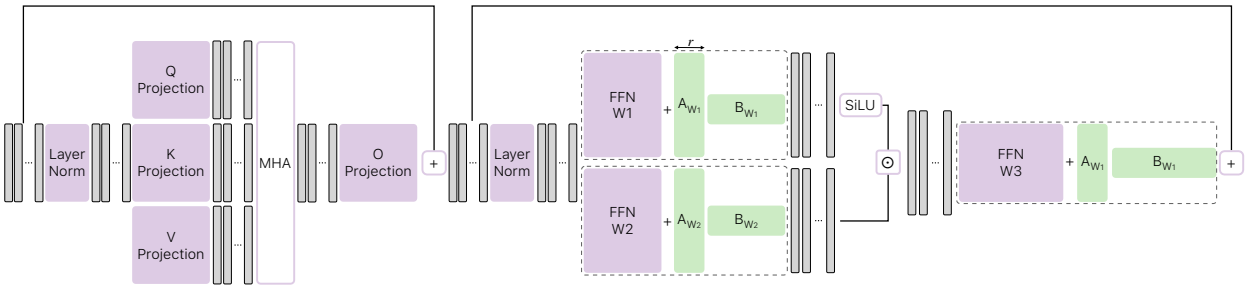

Figure 16 A transformer block with LoRa memories on SwiGLU-FFN linear layers.

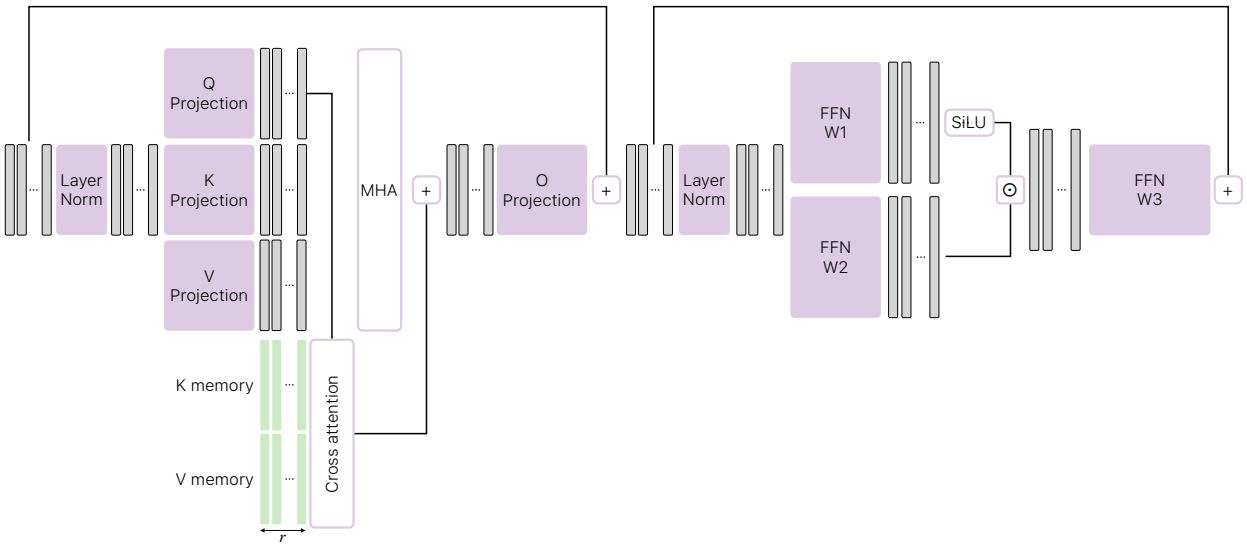

Figure 17 A transformer block with learned KV memories.

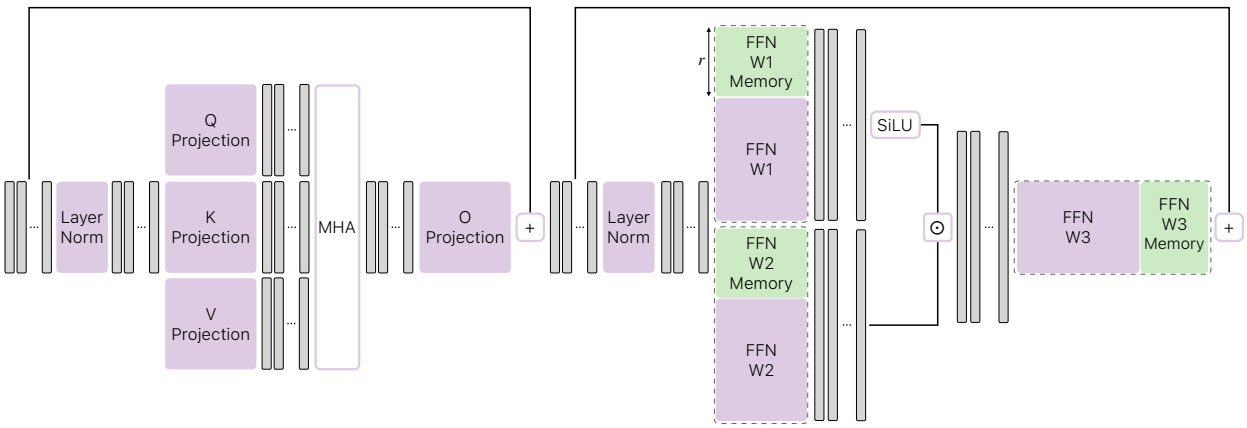

Figure 18 A transformer block with FFN memories.