GRADMEM: LEARNING TO WRITE CONTEXT INTO MEMORY WITH TEST-TIME GRADIENT DESCENT

Yuri Kuratov[1 2 *], **Matvey Kairov**[2], **Aydar Bulatov**[1 2], **Ivan Rodkin**[3 2], **Mikhail Burtsev**[4]
[1]Cognitive AI Systems Lab, Moscow, Russia
[2]MIRAI, Moscow, Russia
[3]MBZUAI, Abu Dhabi, UAE
[4]London Institute for Mathematical Sciences, London, UK
[*]kuratov@cogailab.com

ABSTRACT

Transformers typically process long contexts by storing a large per-layer KV-cache of past activations. A desirable alternative is *compressive memory*: read a context once, store it in a compact state, and answer many queries from that state. We introduce **GradMem**, which writes context into memory via *per-sample test-time optimization*. Given a context, GradMem performs a few steps of gradient descent on a small set of prefix *memory tokens* while keeping model weights frozen. GradMem explicitly optimizes a model-level self-supervised context reconstruction loss, resulting in a loss-driven write operation with iterative error correction, unlike forward-only methods. On associative key-value retrieval, GradMem outperforms forward-only memory writers with the same memory size, and additional gradient steps scale capacity much more effectively than repeated forward writes.

1 INTRODUCTION

Large language models are increasingly deployed in settings where task-relevant information resides in long, external contexts: documents, codebases, tool interactions in agent workflows and dialogue histories spanning multiple sessions (Lewis et al., 2020; Zhang et al., 2023; Team et al., 2024; Team, 2025). In these regimes, the challenge is not only to support long contexts, but to do so efficiently and *reusably*—ideally, the model reads a context once, stores what matters, and answers many queries without repeatedly re-processing the same tokens. The dominant approach is to retain intermediate activations via the KV-cache (and various compression schemes thereof), which reduces recomputation but can impose substantial memory overhead and does not naturally produce a portable representation of the context. A complementary alternative is to provide the model with a *compact memory state* that is constructed from a context and then reused across subsequent queries. Crucially, many applications require incorporating new information *without retraining or fine-tuning the full model*: we want to adapt the model to the current context by writing into a separate memory representation, while keeping the pretrained parameters fixed.

Recent work on *test-time training* shows that a model can adapt to the current context via gradient-based updates during inference (Sun et al., 2025), and that iterative optimization of input embeddings can losslessly encode thousands of tokens given enough steps (Kuratov et al., 2025). Motivated by this observation, we introduce **GradMem**[1]. GradMem writes context into memory by *direct per-sample optimization* at test time (Figure 1). Specifically, GradMem treats embeddings of special memory tokens as *writable state* and performs a small number of *gradient descent* updates on this state for each context. This is *test-time training in the literal sense*: during inference, we execute a short inner-loop optimization on the current example. Crucially, GradMem cleanly separates *memory* from *model weights*: the base model parameters remain fixed, while adaptation to new contexts occurs solely through updates to the memory state. Unlike forward-only writing rules, this loss-driven inner loop provides per-example feedback, enabling GradMem to iteratively correct write errors as it forms a compact memory representation. A key design choice in GradMem is the use of an *explicit, model-level WRITE objective* that is independent of the downstream supervision. In this

[1]Code is available at https://github.com/yurakuratov/gradmem.

paper, we focus on a simple self-supervised WRITE objective—*reconstruction*—computed from the language model's own predictions and backpropagated to the memory tokens. Because the objective is explicit, GradMem provides a direct way to trade compute for compression: additional gradient steps lead to a better memory state.

The intuition behind GradMem is simple. First, standard training with SGD can be viewed as a mechanism that *writes data into parameters* of a model via gradient updates (i.e., train set memorization); analogously, we treat memory as a parameter-like state to store the current context. Second, unlike one-shot forward writing (e.g., with text encoders), optimization provides an explicit signal of *what has not been encoded yet*: the reconstruction loss concentrates on the parts of the context that the model currently predicts poorly. Thus, gradient-based writing naturally prioritizes novel, unpredictable or high-entropy inputs and iteratively reduces reconstruction error. Third, while *lossless* context encoding via iterative optimization is known to be possible, it typically requires *hundreds to thousands* of gradient steps to achieve near-perfect reconstruction (Kuratov et al., 2025). In contrast, GradMem targets the few-step regime: by meta-learning the memory initialization and model parameters, we enable effective context writing with only a small number of test-time gradient steps.

We evaluate GradMem primarily on associative KV-retrieval task under context removal setting, a clean synthetic benchmark that directly measures how much information can be stored in a fixed-size memory. Across a wide range of settings, GradMem stores more key–value pairs than forward-only methods that encode the context into memory with the same memory size. Our results also show that *how* the memory state is updated matters as much as *how many times* it is updated: even a single gradient-based WRITE update can write more information than a single forward-only update, and additional gradient descent steps further increase capacity. In contrast, repeating WRITE using only forward operations (e.g., re-processing the context multiple times) yields much weaker or less consistent gains.

This paper makes the following **contributions**:

1. **GradMem: gradient-based context memorization.** We introduce GradMem, a memory mechanism that encodes a context into a compact memory state by performing a small number of explicit gradient descent steps on memory tokens at test time, while keeping the base model weights fixed.
2. **Few-step gradient writing.** We show that a small set of memory tokens can be meta-trained so that $K \leq 5$ gradient descent steps reliably write task-relevant information into memory, enabling downstream tasks prediction with the original context removed.
3. **Gradient updates outperform forward-only memory updates at fixed memory size and scale with number of memory updates.** We demonstrate that gradient-based updates write substantially more information into a fixed-size memory state than WRITE mechanisms that use only forward computation. As we increase the number of memory-state updates, GradMem consistently gains capacity, whereas repeating forward-only updates (e.g., re-processing the same context multiple times) yields much weaker or less consistent improvements. Furthermore, we provide evidence of transfer of GradMem memory mechanism to pretrained language models on natural language tasks (e.g., bAbI, SQuAD variants, language modeling, see Appendix B).

2 Method: Problem Setup and GradMem

Many sequence modeling problems can be expressed by separating (i) external information that can be used, (ii) a task specification, and (iii) the desired output. We formalize this by representing each task instance as three sequences: *context* C, *query* Q, and *target* Y. Our goal is to enable prediction of Y from Q without direct access to C at inference time, by first compressing C into a small, fixed-size memory $\mathcal{M}$.

Let f_θ be a causal language model parameterized by θ, then standard full-context prediction is $f_\theta\big(Y \mid [C;Q]\big)$. We instead consider a memory-augmented view with a **WRITE/READ** phase decomposition. We introduce a memory representation $\mathcal{M}$ (e.g., KV-cache, or m input vectors of dimension d, or recurrent state) and define two phases:

$$\textbf{WRITE (encode context into memory):} \qquad \mathcal{M} = \mathcal{E}_\theta(C); \tag{1}$$

$$\textbf{READ (decode using memory and query):} \qquad f_\theta(Y \mid \mathcal{M}, Q) \triangleq f_\theta\big(Y \mid [\mathcal{M}; Q]\big). \tag{2}$$

The central evaluation constraint we consider is the *context removal*: during READ phase, the model does not have access to the original context C. All information needed to predict Y must pass

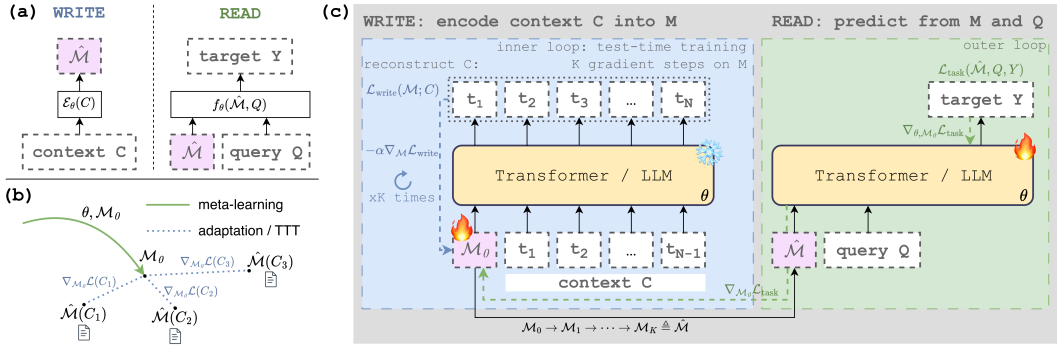

Figure 1: **GradMem overview. (a)** A context encoder E_θ compresses context C into a fixed-size memory $\hat{\mathcal{M}}$ in a WRITE phase, and the model predicts the target Y from $[\hat{\mathcal{M}}; Q]$ in a READ phase, without access to C. **(b)** Meta-learning view: a shared initialization $\mathcal{M}_0$ and model parameters θ are learned across training examples (outer loop), while at test time each context C_i adapts its own memory $\hat{\mathcal{M}}(C_i)$ via a few gradient steps (dotted trajectory). **(c)** Test-time gradient descent on memory. Starting from the meta-learned initialization $\mathcal{M}_0$, GradMem updates the per-sample memory state during WRITE with K steps of gradient descent on the context reconstruction loss $\mathcal{L}_{\text{write}}(\mathcal{M}; C)$. At READ, the model predicts the task target using only $[\hat{\mathcal{M}}; Q]$.

through the memory state $\mathcal{M}$ computed in WRITE phase. Under this setting (Figure 1a), a method is considered successful if the memory $\mathcal{M}$ captures enough task-relevant information from C to solve the task *using memory and query only*.

2.1 GradMem: Test-Time Gradient Descent Memory

GradMem *directly optimizes* the memory representation $\mathcal{M}$: for every example, it performs *test-time training* by running a few gradient descent steps. Crucially, parameters of the model are frozen; instead, only the memory states $\mathcal{M}$ are trained on the current context C, resulting in context-relevant representation for the subsequent READ phase.

Memory parameterization. We represent memory as m vectors of dimension d, $\mathcal{M} \in \mathbb{R}^{m \times d}$. In a decoder-only transformer, these vectors are used as *prefix embeddings* prepended to the model input. GradMem maintains a meta-learned initialization $\mathcal{M}_0$ shared across examples, and produces an example-specific memory $\mathcal{M}_K$ after K WRITE updates. While transformer stores KV-cache of size num_layers $\times\, N \times d \times 2$ for context of length N, GradMem memory $\mathcal{M}$ is independent of context length and requires only m d-dim vectors.

GradMem uses an explicit WRITE objective that is task-agnostic and depends only on the ability of the model to reconstruct the context when conditioned on memory: $\mathcal{L}_{\text{write}}(\mathcal{M}; C) = -\sum_{i=1}^{N} \log f_\theta(t_i \mid [\mathcal{M}; t_{<i}])$.

Starting from the meta-learned initialization $\mathcal{M}_0$, GradMem performs K steps of gradient descent *on the memory parameters only*: $\mathcal{M}_{k+1} = \mathcal{M}_k - \alpha \nabla_{\mathcal{M}_k} \mathcal{L}_{\text{write}}(\mathcal{M}_k; C)$, where α is a WRITE-phase learning rate. We denote the final memory by $\hat{\mathcal{M}} \triangleq \mathcal{M}_K$, and define the context encoder as the composition of these optimization steps: $\hat{\mathcal{M}} = \mathcal{E}_\theta(C) \triangleq \text{GD}_K(\mathcal{M}_0, \mathcal{L}_{\text{write}}(\cdot; C))$. In the READ phase, the model receives only $\hat{\mathcal{M}}$ and the query Q and predicts the target Y. During training, we minimize $\mathcal{L}_{\text{task}}(\hat{\mathcal{M}}, Q, Y)$ w.r.t. θ and $\mathcal{M}_0$ by differentiating through the WRITE phase optimization steps that produce $\hat{\mathcal{M}}$. Importantly, the WRITE objective $\mathcal{L}_{\text{write}}$ is not designed for any specific downstream task; it is a generic reconstruction loss used to form a memory state. GradMem training is summarized on the Figure 1c.

GradMem is closely related to Test-Time Training (TTT) layers (Sun et al., 2025), where an update is performed by gradient descent *online per token* (or small token mini-batches). The self-supervised objective in TTT is typically an ℓ_2 reconstruction loss on the *layer input* x_i. TTT layers reconstruct layer inputs/activations, while GradMem reconstructs the context tokens, and does so once per context rather than at every layer and every token. Conceptually, instead of maintaining and updating

Table 1: **KV-retrieval: gradient-based WRITE outperforms forward-only WRITE.** Exact match retrieval accuracy (mean±std) for predicting a 2-token value from a 2-token key.

Model	Number of KV-pairs 4	8	16	32	64	96
Transformer: $\mathcal{M}$=KV-cache	100.0±0.0	100.0±0.0	99.8±0.0	99.8±0.3	96.5±2.9	98.8±0.0
Mamba: $\mathcal{M}$=per-layer recurrent state	99.9±0.1	98.9±0.4	98.7±0.2	90.2±10.2	95.2±0.1	**92.2**±0.4
ARMT: $\mathcal{M}$=per-layer associative matrix	99.0±0.3	98.5±0.5	97.4±0.3	54.9±2.1	22.6±3.9	15.2±0.2
Forward-only write (RMT): $\mathcal{M}$=8 mem vectors	**100.0**±0.0	**100.0**±0.0	45.5±0.2	44.3±0.0	19.3±3.1	12.9±0.2
x2 memory updates			69.6±28.1	18.7±3.4	–	–
x3 memory updates			60.0±42.0	38.1±0.0	–	–
x4 memory updates			–	31.5±0.0	–	–
x5 memory updates			–	37.0±1.7	–	–
GradMem: $\mathcal{M}$=8 mem vectors	**100.0**±0.0	99.7±0.0	96.3±0.9	86.9±0.5	58.6±0.7	32.6±0.1
x2 memory updates	**100.0**±0.0	**100.0**±0.0	99.6±0.1	98.3±0.1	72.8±0.5	34.2±0.1
x5 memory updates	**100.0**±0.0	**100.0**±0.0	**100.0**±0.0	**99.9**±0.1	**99.1**±0.3	88.4±2.3
x1, w/o meta-learning		12.9±8.1	3.0±0.6	–	–	–
x2, w/o meta-learning		46.7±8.3	4.2±0.7	–	–	–

a separate adaptive state in every layer, GradMem concentrates all test-time adaptation into a single memory state at the model input. We provide a more detail comparison in in Table 2

We overview other related works in Appendix A

3 Experiments and Results

Associative retrieval is our main synthetic and controllable benchmark for comparing different memory mechanisms. Each example contains N key–value pairs (k_i, v_i), where each key and each value consists of 2 symbols from a 62-character vocabulary. The context is a sequence of key–value pairs with special delimiters: $C = !\,k_1 : v_1\,!\,!\,k_2 : v_2\,! \cdots !\,k_N : v_N\,!$. The query Q asks for the value associated with key k_j and the target Y is the corresponding value: $Q = ?\,!\,k_j :, Y = v_j$. The model can answer correctly only if the mapping from keys to values is written into memory during WRITE phase. We report the performance on KV-retrieval in Table 1. We also evaluate our models on natural language tasks with pre-trained language models. These results are reported in Appendix B.

We compare GradMem to a number of baseline architectures with both full contexts and forward-only memory writes. First, we use the full-attention Transformer as an upper bound for memorization, as its memory is uncompressed, training a 4-layer Llama model (Touvron et al., 2023) for KV-retrieval. We also include 4-layer Mamba (Dao & Gu, 2024) as a recurrent baseline. As a forward-only memory write baseline, we use the Recurrent Memory Transformer (RMT) (Bulatov et al., 2022), or, more specifically, its 2-segment version: the first segment contains the context, and the second one the query with the target. Finally, we compare to Associative Recurrent Memory Transformer (ARMT) (Rodkin et al., 2024) as another strong forward-only WRITE baseline.

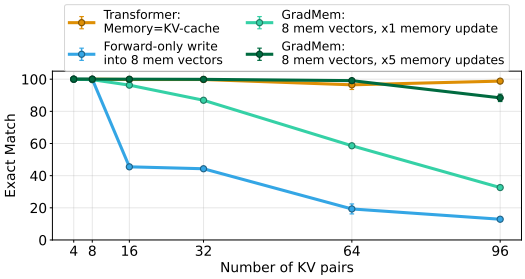

Figure 2: **Gradient-based memory updates (GradMem) outperform forward-only updates at the same memory size.** With a memory state of 8 vectors, GradMem retrieves any of 16 KV-pairs with 95% accuracy, whereas forward-only updates store only 8 pairs with high accuracy. More gradient steps increases the capacity of the same 8-vector memory to 96 pairs at 88% retrieval accuracy. Transformer with KV-cache as non-compressive memory serves as an upper bound.

Figure 2 compares forward-only writing (RMT) with GradMem under the same memory size, and varying the number of WRITE updates. We can see three trends. (i) Gradient-based updates of a model-level memory state are effective: by directly optimizing memory tokens per example at test-time, GradMem writes context information into a compact memory and attains high retrieval accuracy compared to RMT with forward-only memory update. (ii) Even a single gradient-based WRITE step (K=1) substantially outperforms a forward-only write at the same memory size. (iii) Allocating more WRITE compute to gradient updates further improves performance: increasing to K=5 enables accurate retrieval at much larger numbers of key–value pairs. Moreover, repeating the forward-only

WRITE yields weak or inconsistent improvements, whereas additional gradient-based WRITE steps reliably increase performance.

In GradMem, we find that increasing the number of WRITE iterations at evaluation time provides a substantial accuracy lift on KV-retrieval task. Figure 3(a) shows a trend across settings with different numbers of key–value pairs where for some models extrapolating from the training-time value K_{train} to larger K_{eval} yields improvements in EM. Importantly, these gains are obtained with *fixed* model parameters and therefore reflect the effect of allocating additional WRITE refinement steps at inference. We connect this effect to better convergence on the inner task. By decomposing the inner loss into loss on key or value tokens (Figure 3(b)) we observe that inner loss correlates strongly with EM (more results are in Appendix F).

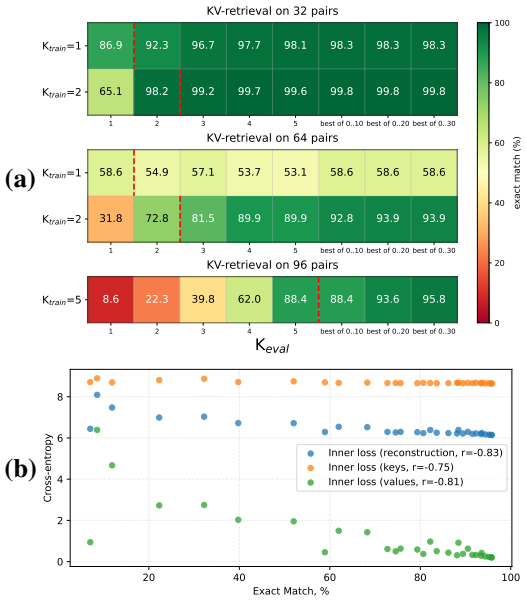

Figure 3: **More gradient steps at test-time leads to better performance. (a)** Results for K_{train} values of GradMem setups that could not achieve 99% exact match during fine-tuning. Red dashed lines denote the beginning of extrapolation. **(b)** Downstream task quality (Exact Match) correlates with WRITE objective in inner loop (reconstruction). Results on 96-pair KV-retrieval.

4 Discussion and Conclusions

Compute and memory overhead. A core cost of GradMem is that it backpropagates through the WRITE inner loop and, during training, differentiates through the unrolled optimization steps. This increases both compute and GPU memory usage compared to forward-only writers and standard full-context training, since the computational graph must be retained across WRITE steps. In practice, this also constrains the choice of attention implementation: common high-performance kernels such as FlashAttention (Dao et al., 2022) and PyTorch SDPA are designed for efficient first-order backpropagation, but do not support the higher-order differentiation required by GradMem training (i.e., taking gradients through the WRITE updates). As a result, we implement a custom double-backward that is both *more memory-efficient* and *faster* than a naive eager implementation, described in Appendix D.

When test-time WRITE compute is worthwhile. At inference time, the WRITE phase with K gradient steps is more expensive than a single forward pass over the context. Nevertheless, in applications where the same context is reused across multiple queries, the additional WRITE compute can be amortized: after encoding a long context C into a compact memory $\mathcal{M}$ once, subsequent READ computations attend only over $[\mathcal{M}; Q]$ rather than $[C; Q]$. When $|C| \gg |\mathcal{M}|$, this can reduce per-query computation and memory, making gradient-based writing a more compute-reasonable option. We analyze this in Appendix E.

Write objectives beyond reconstruction. We use a simple token-level reconstruction objective for $\mathcal{L}_{\text{write}}$, which is task-agnostic and easy to apply across domains. Despite its simplicity, it is already sufficient to yield strong gains on associative retrieval and to transfer to natural language tasks. At the same time, reconstruction is unlikely to be optimal for all downstream tasks. A promising direction is to learn WRITE objectives that better preserve task-relevant information, while retaining the key constraint that WRITE remains self-supervised on the available context.

Conclusions. We introduced GradMem, a WRITE/READ memory mechanism where a model *writes* a context into a small set of memory tokens by running a few steps of test-time gradient descent while keeping model weights fixed. Across controlled KV-retrieval experiments, this direct per-sample optimization gives a stronger WRITE rule than forward-only updates under the same architecture and memory size. Increasing the number of gradient WRITE steps reliably improves retrieval as the number of stored key–value pairs grows, unlike repeating forward WRITE passes. Overall, our results establish gradient-based test-time optimization of a model-level memory state as a promising alternative to forward-only memory writers.

REFERENCES

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663*, 2024.

Ali Behrouz, Zeman Li, Praneeth Kacham, Majid Daliri, Yuan Deng, Peilin Zhong, Meisam Razaviyayn, and Vahab Mirrokni. Atlas: Learning to optimally memorize the context at test time. *arXiv preprint arXiv:2505.23735*, 2025.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pp. 2397–2430. PMLR, 2023.

Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. In Stefan Riezler and Yoav Goldberg (eds.), *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*, pp. 10–21, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/K16-1002. URL https://aclanthology.org/K16-1002/.

Aydar Bulatov, Yury Kuratov, and Mikhail Burtsev. Recurrent memory transformer. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 11079–11091. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/47e288629a6996a17ce50b90a056a0e1-Paper-Conference.pdf.

Mikhail S. Burtsev, Yuri Kuratov, Anton Peganov, and Grigory V. Sapunov. Memory transformer, 2021.

Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder for english. In *Proceedings of the 2018 conference on empirical methods in natural language processing: system demonstrations*, pp. 169–174, 2018.

Vivek Chari, Guanghui Qin, and Benjamin Van Durme. Kv-distill: Nearly lossless learnable context compression for llms. *arXiv preprint arXiv:2503.10337*, 2025.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3829–3846, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.232. URL https://aclanthology.org/2023.emnlp-main.232/.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024. URL https://arxiv.org/abs/2405.21060.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.

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.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1126–1135. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/finn17a.html.

Jun Gao. Selfcp: Compressing long prompt to 1/12 using the frozen large language model itself. *arXiv preprint arXiv:2405.17052*, 2024.

Tao Ge, Hu Jing, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=uREj4ZuGJE.

Geoffrey E Hinton and David C Plaut. Using fast weights to deblur old memories. In *Proceedings of the ninth annual conference of the Cognitive Science Society*, pp. 177–186, 1987.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. Editing models with task arithmetic. In *The Eleventh International Conference on Learning Representations*, 2022.

Mahdi Karami, Ali Behrouz, Praneeth Kacham, and Vahab Mirrokni. Trellis: Learning to compress key-value memory in attention models. In *Second Conference on Language Modeling*, 2025.

Ryan Kiros, Yukun Zhu, Russ R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/file/f442d33fa06832082290ad8544a8da27-Paper.pdf.

Yuri Kuratov, Mikhail Arkhipov, Aydar Bulatov, and Mikhail Burtsev. Cramming 1568 tokens into a single vector and back again: Exploring the limits of embedding space capacity. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19323–19339, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.948. URL https://aclanthology.org/2025.acl-long.948/.

Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In Eric P. Xing and Tony Jebara (eds.), *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pp. 1188–1196, Bejing, China, 22–24 Jun 2014. PMLR. URL https://proceedings.mlr.press/v32/le14.html.

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 (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 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/.

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 H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (eds.), *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)*, pp. 4582–4597, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.353. URL https://aclanthology.org/2021.acl-long.353/.

Zongqian Li, Yixuan Su, and Nigel Collier. 500xcompressor: Generalized prompt compression for large language models. *arXiv preprint arXiv:2408.03094*, 2024.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Byj72udxe.

Yishu Miao, Lei Yu, and Phil Blunsom. Neural variational inference for text processing. In Maria Florina Balcan and Kilian Q. Weinberger (eds.), *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 1727–1736, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/miao16.html.

Jatin Prakash, Aahlad Puli, and Rajesh Ranganath. Attention and compression is all you need for controllably efficient language models. *arXiv preprint arXiv:2511.05313*, 2025.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SylKikSYDH.

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 (eds.), *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL https://aclanthology.org/D16-1264/.

Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Lukas Gruber, Markus Holzleitner, Thomas Adler, David Kreil, Michael K Kopp, et al. Hopfield networks is all you need. In *International Conference on Learning Representations*, 2020.

Ivan Rodkin, Yuri Kuratov, Aydar Bulatov, and Mikhail Burtsev. Associative recurrent memory transformer. *arXiv preprint arXiv:2407.04841*, 2024.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, Tatsunori Hashimoto, and Carlos Guestrin. Learning to (learn at test time): RNNs with expressive hidden states. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=wXfuOj9C7L.

Arnuv Tandon, Karan Dalal, Xinhao Li, Daniel Koceja, Marcel Rød, Sam Buchanan, Xiaolong Wang, Jure Leskovec, Sanmi Koyejo, Tatsunori Hashimoto, et al. End-to-end test-time training for long context. *arXiv preprint arXiv:2512.23675*, 2025.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Kimi Team. Kimi k2: Open agentic intelligence, 2025. URL https://arxiv.org/abs/2507.20534.

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

Jason Weston, Antoine Bordes, Sumit Chopra, and Tomás Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1502.05698.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151/.

A Related Work

Long-context modeling and efficient attention. A large body of work aims to extend the effective context length of transformers by changing the architecture or attention mechanism, thereby reducing the need for an explicit external memory. Compressive Transformers (Rae et al., 2020) augment the standard recurrent memory with a compressed memory stream, retaining a lossy summary of distant past activations. Recurrent memory transformers such as RMT (Bulatov et al., 2022) and their extensions (Chevalier et al., 2023) segment long inputs into chunks and pass a trainable state between segments, enabling processing of sequences far beyond the native context window. Other approaches leverage associative memories and efficient attention mechanisms to scale to long contexts (Rodkin et al., 2024; Ramsauer et al., 2020).

Context compression and reusable representations. In NLP, compression has long been studied for constructing compact sentence- or document-level representations (Le & Mikolov, 2014; Kiros et al., 2015; Cer et al., 2018), often via autoencoding pipelines (Bowman et al., 2016; Miao et al., 2016). In the context of large language models, input compression is used both to reduce the quadratic cost of self-attention and to create persistent representations that can be stored or reused. For compressed data storage in LLMs, one can use dense continuous vectors or memory tokens (Burtsev et al., 2021), but also LoRA parameters (Hu et al., 2022), intermediate hidden states (Li et al., 2024), associative memories (Rodkin et al., 2024; Ramsauer et al., 2020), or the KV-cache directly (Chari et al., 2025; Karami et al., 2025). Some works focus on compressing only part of the input, such as prompts (Lester et al., 2021; Li & Liang, 2021) or in-context examples (Ge et al., 2024; Eyuboglu et al., 2025), while others iteratively compress the entire sequence by splitting it into smaller chunks and processing them sequentially (Rae et al., 2020; Bulatov et al., 2022; Chevalier et al., 2023; Prakash et al., 2025). Approaches such as task vectors (Ilharco et al., 2022) and Cartridges (Eyuboglu et al., 2025) compress information from multiple task samples into persistent vectors that can be reused to steer the model on those tasks, amortizing the cost of processing across many downstream queries.

A straightforward way to compress information into memory is to use the model itself as an encoder, e.g., via segment-level memory tokens in RMT-style architectures (Bulatov et al., 2022; Chevalier et al., 2023; Gao, 2024). ICAE (Ge et al., 2024) and SelfCP (Gao, 2024) train base LLMs to compress context using autoencoding-style objectives and other losses targeted at text understanding (Li et al., 2024).

Fast weights, delta rules, and associative memory offer another path to context-dependent memory. Early work used fast weights as high-level controllers (Schmidhuber, 1992) or as associative storage over past representations (Hinton & Plaut, 1987). Recent formulations adapt fast weights to modern architectures, often in the form of associative memories or gated recurrent states that implement a learned approximation to a delta rule. Associative-memory transformers (Rodkin et al., 2024) and modern Hopfield networks (Ramsauer et al., 2020) can be interpreted as storing and retrieving key–value pairs in a continuous memory, with the write operation implemented purely via forward computation. These mechanisms generally rely on *forward-only* update rules that are applied once per token or timestep, without an explicit per-example optimization objective that is iteratively minimized for a given context.

Test-time training. GradMem is closely related to test-time training (TTT) methods that perform gradient-based adaptation during inference. TTT layers (Sun et al., 2025) introduce lightweight, sequence-dependent state that is updated online using self-supervised objectives, typically by reconstructing layer activations at each timestep. A related line of work uses optimization as a way to compress large amounts of data into a fixed number of weights or embeddings. Early formulations viewed fast weights as auxiliary parameters that are updated via gradient-based rules (Schmidhuber, 1992). More recent work proposes unsupervised objectives for compressing context via test-time optimization (Sun et al., 2025), and adapts these ideas to transformers by introducing additional memory modules, routing mechanisms (Behrouz et al., 2024; 2025) and end-to-end training objectives (Tandon et al., 2025). The study of Kuratov et al. (2025) shows that, by optimizing a simple reconstruction objective with gradient descent, it is possible to achieve extremely high compression ratios (up to $\sim$1500$\times$) into a single vector; however, this requires up to tens of thousands of gradient updates and produces representations that are primarily useful for text reconstruction rather than downstream tasks.

Relation to Test-Time Training (TTT) layers. GradMem is closely related to Test-Time Training (TTT) layers (Sun et al., 2025), where an update is performed by gradient descent *online per token* (or small token mini-batches). , which update a sequence-dependent state by gradient descent on a self-supervised loss *during inference*. producing a trajectory of adapted states $W_1, \ldots, W_N$ for each input sequence. The self-supervised objective in TTT is typically an ℓ_2 reconstruction loss on the *layer input* x_i. , so x_t can correspond to hidden activations in upper layers. TTT layers reconstruct layer inputs/activations, while GradMem reconstructs the context tokens, and does so once per context rather than at every layer and every token. Concretely, GradMem applies the same *training at test time* principle at the *context level*: we optimize a single compact memory $\mathcal{M}$ using the entire context C in a dedicated WRITE phase, and then answer queries in a separate READ phase using only $[\hat{\mathcal{M}}; Q]$ with the original context removed. Conceptually, instead of maintaining and updating a separate adaptive state in every layer, GradMem concentrates all test-time adaptation into a single memory state at the model input. We compare key parts of TTT layers with GradMem in Table 2.

Table 2: Comparison of GradMem to test-time training (TTT) layers. Both approaches perform learning at inference time via a self-supervised loss, but differ in *what* is adapted (layer parameters vs. memory tokens), *when* it is adapted (token-level online vs. context-level WRITE), and *what* the self-supervised signal reconstructs (layer inputs/activations vs. context tokens).

	TTT layers (Sun et al., 2025)	**GradMem (ours)**
Usage pattern	Sequence-modeling layer: updates state online while processing tokens	Explicit two-phase WRITE/READ setting
Inner-loop input	Token x_t (or token mini-batches)	Whole context segment C (WRITE once per context)
Test-time parameters	Layer-specific parameters W_t (updated from the layer's inputs/activations), *per layer*	Prefix memory tokens $\mathcal{M}_k \in \mathbb{R}^{m \times d}$ (single memory state), *per model*
Self-supervised loss	Activation/input reconstruction, e.g. $\ell(W; x_t) = \|f(\tilde{x}_t; W) - x_t\|_2^2$ (or learned multi-view projections)	Context reconstruction $\mathcal{L}_{\text{write}}(\mathcal{M}; C)$
Outer-loop objective	Next-token prediction (LM training)	Downstream task loss with C removed at READ
Outer-loop parameters	Model params θ + reconstruction task/view params	Model params θ + memory init $\mathcal{M}_0$ (optional: memory projections / control tokens)

Connection to meta-learning. GradMem can be viewed through a meta-learning lens (Figure 1b, Finn et al. (2017)): the WRITE phase performs a small number of per-sample optimization steps on $\mathcal{M}$, while the model parameters θ and the shared initialization $\mathcal{M}_0$ are trained so that these few steps reliably produce useful memories. In this view, the WRITE updates in Section 2.1 correspond to an *inner optimization (inner loop)* over per-sample memory variables, and the task loss (Figure 1c) defines an *outer objective (outer loop)* used to learn θ and $\mathcal{M}_0$ across training examples. We backpropagate through the WRITE optimization (yielding second-order gradients). In our setup, first-order approximations were not sufficient to reach strong performance, so we use the second-order variant in all experiments unless noted otherwise. At inference time, the WRITE phase is executed per example to obtain $\hat{\mathcal{M}}$, after which all downstream computation uses only $[\hat{\mathcal{M}}; Q]$.

Comparison to our approach Our work is distinguished from prior memory and TTT approaches along three main axes. First, GradMem uses a *single input-level memory state* that is written once per context, rather than per-layer or per-token states updated online. Second, the WRITE mechanism is *explicitly optimization-based*: memory tokens are treated as parameters and updated by gradient descent on a model-level reconstruction loss, rather than via a learned forward-only update rule. Third, we target the *few-step* regime, meta-training the base model and memory initialization so that a small number of gradient steps ($K \leq 5$) suffices for effective writing, in contrast to hundreds or thousands of iterations used in prior embedding-optimization work.

Table 3: **GradMem remains competitive on downstream language tasks.** Results on bAbI (QA1–5), SQuAD (short) shown as Exact Match (EM) in %, and language modeling (cross-entropy on token ranges). We use 32 mem tokens for SQuAD and LM, and 8 on bAbI. We report mean±std over 3 runs.

	bAbI (EM↑)					SQuAD (EM↑)	LM (CE↓)
	QA1	QA2	QA3	QA4	QA5	short	128–255
Input length (tokens)	∼40	∼100	∼300	∼20	∼20	∼40	256
Full context models (upper bound)							
GPT-2-124m	100.0±0.0	100.0±0.0	99.8±0.1	100.0±0.0	99.4±0.1	64.2±0.3	2.72±0.00
Limit context to 128 tokens							3.20±0.00
Pythia-160m	100.0±0.0	99.7±0.1	95.5±2.7	100.0±0.0	99.0±0.1	48.9±0.4	2.84±0.05
Recurrent models							
Mamba-130m-hf	100.0±0.0	100.0±0.0	96.7±0.2	100.0±0.0	99.7±0.09	63.3±0.2	2.69±0.00
RMT (GPT-2)	100.0±0.0	93.9±0.1	87.9±0.4	100.0±0.0	93.9±6.9	42.6±0.3	2.91±0.00
ARMT (GPT-2)	100.0±0.0	93.8±0.6	92.3±0.8	100.0±0.0	98.9±0.1	39.0±0.2	2.85±0.00
GradMem (GPT-2, $K = 1$)	100.0±0.0	94.2±0.4	80.0±0.2	100.0±0.0	99.2±0.1	38.1±0.1	2.92±0.00
GradMem (GPT-2, increased K)	100.0±0.0	93.9±0.5	79.3±0.2	100.0±0.0	99.2±0.1	54.9±0.4	–

B GradMem with pre-trained models on NLP tasks

We evaluate GradMem with pretrained language models on natural language benchmarks to test whether the gradient-based WRITE method and reconstruction-based WRITE objective can produce useful memory states outside the controlled KV-retrieval setting (see Table 3).

bAbI (Weston et al., 2016) is a question answering benchmark that tests reasoning over stories. We use tasks QA1–QA5, which progressively increase the amount of multi-sentence composition required. **SQuAD** (Rajpurkar et al., 2016) is another question answering dataset, containing tasks based on Wikipedia articles. We construct a short context variant (Short SQuAD) to control context length and isolate whether GradMem's writing mechanism transfers to natural language by extracting sentences containing the annotated answer span only. We also evaluate our approaches on **language modeling** to estimate our model's ability to condition on a preceding context to reduce the cross-entropy loss on subsequent tokens. We split examples from WikiText-103 (Merity et al., 2017) into two consecutive 128-token chunks and use the first chunk as the encoded context C, the second chunk as the target Y and leaving the query empty ($Q = \emptyset$).

For these tasks, we finetune full-attention Transformers, including GPT-2 (Radford et al., 2019) and Pythia (Biderman et al., 2023), as well as a 130M-parameter Mamba. For RMT, ARMT and GradMem we use GPT-2 as the base model.

The bAbI evaluation exhibits significant variability of scores on different reasoning tasks. The context size of QA1, QA4 and QA5 generally does not exceed 40 tokens, and all evaluated methods solve them well, matching the upper bound Transformer performance. QA2 and QA3 require to memorize more facts and detect them among more noise, which poses a challenge to compressive models. Mamba performs the best among recurrent models since the memory operations are already learned during extensive pretraining, unlike GradMem and recurrent Transformers. ARMT follows short, its strong performance on QA2 and QA3 can be attributed to the largest memory state size in its class and added parameters in associative layers. GradMem matches or outperforms forward-only RMT on all QA tasks except QA3 with the highest information density, and performance improves with larger K.

By scaling K, GradMem achieves the best performance among recurrent Transformers on Short SQuAD and even surpasses the uncompressed Pythia model, however GPT-2 still remains the upper bound. On the Wikitext language modeling task, all compressive Transformers learn to use memory and noticeably outperform GPT-2 with context size 128; full-context GPT-2 and Mamba remain the upper bound. ARMT with its more capacious memory takes the lead, followed by RMT and GradMem.

C HYPERPARAMETERS AND TRAINING DETAILS

We provide the source code as supplementary material with the submission.

Table 4 summarizes the memory configuration and training initialization used across tasks. We keep the memory embedding dimension fixed to $d_{\text{mem}} = 64$ in all ARMT experiments, and vary only the number of memory tokens depending on the task. Unless otherwise noted, "from pretrained model" means initializing the base LM weights from a standard pretrained checkpoint (e.g., GPT-2 (124M) / Pythia (160M) / Mamba (130M)) and then fine-tuning with the corresponding memory mechanism.

For associative retrieval (AR), we train models from scratch and use a curriculum over context lengths: training for a larger number of key–value pairs is initialized from the final checkpoint obtained at a smaller length. For GradMem, the curriculum starts from 32 key–value pairs, as we found that the model achieves perfect retrieval on smaller contexts even without curriculum training. For bAbI, most models are fine-tuned from pretrained checkpoints; the exception is RMT, which we found to be sensitive to initialization and therefore train it starting from a GPT-2 (124M) checkpoint already fine-tuned on bAbI. For language modeling, all models are fine-tuned from pretrained checkpoints; additionally, we report a variant where GradMem is initialized from an RMT checkpoint trained on the same language modeling objective.

For GradMem, we tune the inner learning rate $\alpha \in [0.01, 10]$. We observe that model performance does not depend strongly on its value, until it is in a reasonable range and found $\alpha = 0.4$ to be a relatively strong default value for experiments on NLP tasks. For the KV-retrieval task, we still perform small search over α and report the scores for the best setup with every K, but we do not tune α exhaustively in every experiment. We also augment both GradMem and RMT with (i) a learned linear layer applied to the memory before/after the updates and (ii) separate prediction heads (output embeddings) for the WRITE and READ phases. These augmentations are used in all experiments unless stated otherwise. We backpropagate through the unrolled WRITE optimization by default; we explicitly note experiments that disable this meta-learning path (e.g., Table 1, w/o meta-learning).

Table 4: Task-specific memory hyperparameters and training initialization. In experiments ARMT uses $d_{\text{armt\_mem}} = 64$ for associative matrix memory. Memory tokens in RMT, ARMT, and GradMem have the same dimension as models input embeddings.

Task	**num_mem_tokens**	**Training technique**
Associative retrieval (AR)	8	Curriculum learning by number of KV-pairs (4, 8, 16, ...); from randomly init model (4 layers, 4 heads, 128 hid)
Short SQuAD	32	From pretrained model; GradMem trained without learned memory projection
bAbI	8	From pretrained model
Language modeling	32	From pretrained model (all models); GradMem trained without separate WRITE head;

D ACCELERATING DOUBLE BACKWARDS THROUGH ATTENTION.

In our meta-learning setting, the dominant computational challenge is that the inner-loop optimization requires backwards-over-backwards through attention, which substantially increases both runtime and GPU memory compared to standard training. We implement an efficient double-backward for attention that significantly reduces this overhead on longer sequences: for L=1024 tokens, backward time drops from $\sim$1000 ms to $\sim$600 ms and peak GPU memory from $\sim$60 GB to $\sim$30 GB in our setup. This improvement is critical for scaling GradMem to longer contexts and larger WRITE step counts. In order to accelerate our experiments, we evaluated a few approaches to make the double backward of GradMem more efficient:

- **Eager**: a baseline which fully relies on PyTorch's autograd for first- and second-order differentiation.

- **Fast forward → manual backward**: the forward pass is computed using PyTorch's SDPA kernel. The first-order backward is written analytically, and the second-order derivatives are obtained by differentiating the analytical backward with autograd.
- **Fast forward → autograd**: the forward pass also uses SDPA, but constructs the backward by recomputing the attention forward inside the backward and letting autograd differentiate it, enabling second-order derivatives without storing forward intermediates at the cost of recomputation.
- **Manual HVP**: a fully analytical implementation of forward, backward, and double backward in pure PyTorch.
- **Flash HVP**: fused forward and backward kernels, combined with an analytical double backward.

We compare the speed of backward methods in Figure 4. While on shorter sequences eager attention is the most practical solution, longer contexts benefit from our optimizations, with Fast forward → autograd being the fastest and Manual HVP by far the most memory-efficient. In general, Flash HVP is the most balanced approach, coming in second in both speed and memory requirements.

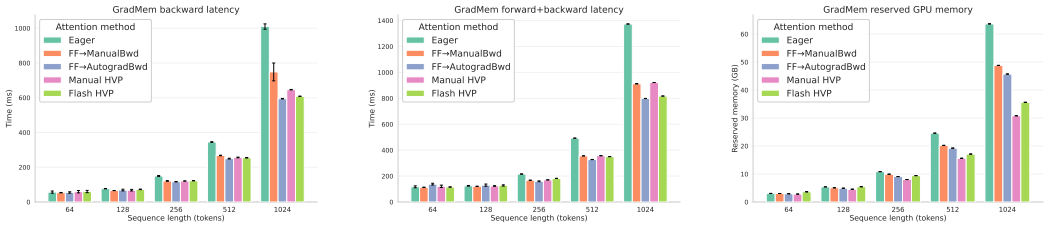

Figure 4: **Comparison of speed and memory consumption for attention backwards-over-backwards in GradMem.** These results were obtained using an A100 GPU, with GPT-2 as the base model for GradMem with 8 memory tokens, a query size of 24, 1 inner SGD step, and a batch size of 16.

E Computational analysis: when GradMem is compute-efficient

GradMem introduces additional WRITE-time compute (test-time optimization) in exchange for cheaper READ-time inference, since subsequent queries attend only over a short memory prefix rather than the full context. Here we characterize when this trade-off is favorable.

Let c be the context length, q the query length with $q \ll c$ is negligible, m the number of memory tokens, N the number of queries asked about the *same* context, and K the number of gradient updates in the WRITE phase. We use R to denote the ratio between the cost of one memory update step and the cost of one forward pass over the context. We compare (i) standard full-context transformer inference that reuses the context for each query, and (ii) GradMem, which pays a WRITE cost once and then answers queries using only memory.

For a transformer-style model, self-attention over a sequence of length L costs $\mathcal{O}(L^2)$, and cross-attention from a query of length q to a context of length c costs $\mathcal{O}(cq)$. If we cache the context representations, standard transformer inference incurs a one-time cost to process C, plus a per-query cross-attention cost:

$$T_{\text{full}} \approx c^2 + cqN. \tag{3}$$

For GradMem, the WRITE phase runs K gradient descent steps on the context. Each step costs R times a forward pass over C, giving a WRITE cost of Rc^2K. At READ time, we process m memory tokens once and each query attends only over the memory tokens and query q tokens. In total giving following cost of both WRITE and multiple reads:

$$T_{\text{GradMem}} \approx R(c+m)^2K + m^2 + mqN. \tag{4}$$

Break-even condition. GradMem is compute-efficient when the total cost is lower than full-context inference: $T_{\text{full}}/T_{\text{GradMem}} > 1$. Equivalently,

$$N > \frac{c^2(RK-1) + (1+RK)m^2 + 2cmRK}{q(c-m)}. \tag{5}$$

In the regime where q is treated as a small constant factor and , this reduces to the simpler heuristic threshold $N \gtrsim \big(c(RK-1)\big)/q$, which matches the form used in our empirical discussion.

Thus, when the same context is reused for more than the threshold in Equation (5) and $c > m$, GradMem yields lower total compute than repeatedly answering from the full context (Figure 5). In practice, the regime $c \gg m$ and large N (many queries per context) is the most favorable for GradMem, since the amortized savings in READ grow linearly with N while the WRITE cost is paid only once per context.

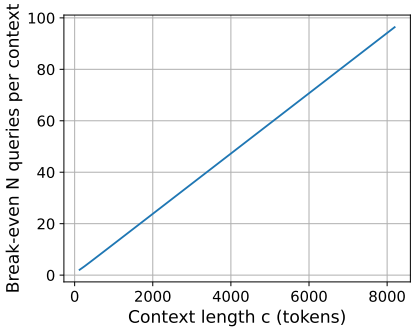

Figure 5: **Break-even number of queries for GradMem compute efficiency.** The curve shows the minimum number of queries per context N required for GradMem to use less total compute than full-context inference with cached context representations. Estimates use query length q=128 and memory size m=32; points above the curve favor GradMem.

Real-use READ/WRITE latency. On a GPU, models can be bound by factors other than theoretical model complexity, i.e. memory bandwidth or kernel efficiency. In order to control for those factors, we evaluate GradMem against GPT-2 (124M) and Mamba-130m baselines in terms of measured READ and WRITE operation latencies. We treat building the KV-cache in GPT-2 and the recurrent state in Mamba as WRITE operations, while subsequent forward passes using the caches correspond to READ operations.

Figure 6 reports the total latency of a WRITE phase followed by N READ (subsequent queries to the same context) operations from the cached representations for contexts of length 64, 256 and 1024 tokens. GradMem has a large initial cost due to the complexity of gradient-based WRITE, which appears as a higher initial offset. However, the complexity of repeating the READ phase is smaller for GradMem compared to other models, since the underlying transformer attends only to a small set of memory tokens instead of the entire context. In contrast, GPT-2 must repeatedly process its KV-cache, resulting in a higher latency per query.

As the number of READ operations rises, the initial overhead of GradMem WRITE operation becomes less pronounced. GradMem consistently outperforms Mamba across all evaluated context lengths, and breaks even with GPT-2 after approximately 64 READ phases for the same context (for context size 256 and 1024).

F Relation between Exact Match and inner loss

To better understand why extrapolating the number of WRITE iterations improves downstream performance, we analyze the behavior of the inner objective during the WRITE phase. Figure 7 shows that increasing K yields a reduction in the inner loss, indicating that additional WRITE iterations produce a more accurate memory state under the reconstruction objective. Moreover, the reduction in inner loss correlates with improvements in Exact Match when evaluating with larger K_{eval}. These

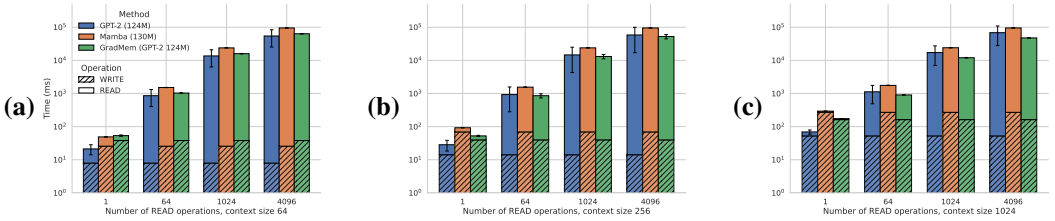

Figure 6: **GradMem is competitive in latency when the same context is reused across many READ operations.** Results obtained on an A100 GPU. WRITE+READ latency is shown for contexts of length 64 (a), 256 (b), and 1024 (c) tokens. The query length is 24 tokens and the batch size is 16. GradMem uses GPT-2 as the base model with $K = 1$. For GPT-2 and Mamba-130M, the WRITE operation corresponds to building the KV-cache and the recurrent state, respectively. The y-axis uses a logarithmic scale; the absolute latency difference (ms) increases with the number of queries per context.

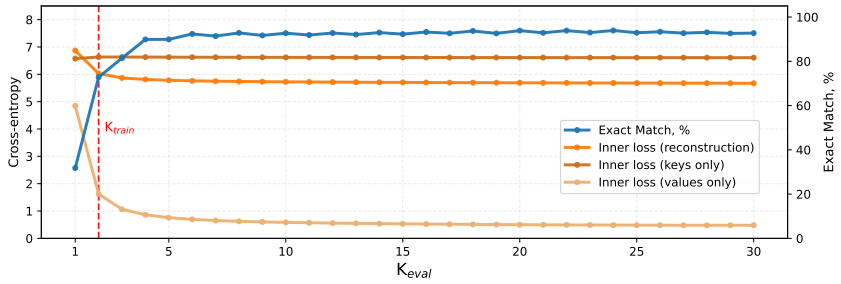

Figure 7: **On KV-retrieval, GradMem learns to decrease inner loss on value tokens only.** This figure was obtained for GradMem fine-tuned on 64-pair KV-retrieval with $K_{\text{train}} = 2$.

results provide evidence that improved context retention—as measured by the inner objective—translates into better task-level accuracy in the context-removal setting.

We further analyze *what* information is being retained by decomposing the reconstruction loss over context tokens into contributions from **key** tokens and **value** tokens in the associative retrieval data. Notably, the loss on key tokens remains comparatively stable across WRITE iterations, while the loss on value tokens decreases as K_{eval} increases. This behavior suggests that the learned memory is *selective*: rather than attempting to store the full context verbatim, the model primarily refines those parts of the representation that are useful for answering queries, i.e., the values that must be produced at READ time. In this regime, keys function mainly as retrieval cues, so improving value reconstruction only is sufficient for increasing Exact Match. This supports two conclusions: (i) better memory fidelity under the inner objective leads to higher downstream accuracy, and (ii) the memory mechanism learns to allocate representational capacity toward answer-relevant content, producing a structured compression in which values are retained more precisely than the keys that index them.