MEMORYLLM: Towards Self-Updatable Large Language Models

Yu Wang[*][1] **Yifan Gao**[2] **Xiusi Chen**[3] **Haoming Jiang**[2] **Shiyang Li**[2] **Jingfeng Yang**[2] **Qingyu Yin**[2] **Zheng Li**[2] **Xian Li**[2] **Bing Yin**[2] **Jingbo Shang**[1] **Julian McAuley**[1]

Abstract

Existing Large Language Models (LLMs) usually remain static after deployment, which might make it hard to inject new knowledge into the model. We aim to build models containing a considerable portion of self-updatable parameters, enabling the model to integrate new knowledge effectively and efficiently. To this end, we introduce MEMO-RYLLM, a model that comprises a transformer and a fixed-size memory pool within the latent space of the transformer. MEMORYLLM can self-update with text knowledge and memorize the knowledge injected earlier. Our evaluations demonstrate the ability of MEMORYLLM to effectively incorporate new knowledge, as evidenced by its performance on model editing benchmarks. Meanwhile, the model exhibits long-term information retention capacity, which is validated through our custom-designed evaluations and long-context benchmarks. MEMORYLLM also shows operational integrity without any sign of performance degradation even after nearly a million memory updates. Our code and model are open-sourced at https://github.com/wangyu-ustc/MemoryLLM.

1. Introduction

Despite the impressive performance LLMs demonstrate, a pivotal issue persists: *How should we update the model with the latest knowledge?* Previous solutions can be broadly categorized into three classes: **(1) Retrieval-Based Methods:** These methods rely on information retrieval in a knowledge base (Khandelwal et al., 2019; Zhong et al., 2023). They can yield strong results, but face challenges when redundancy in the knowledge base presents and suffer the logistical issue of managing an ever-expanding repository of knowledge.

In multi-modality scenarios, retrieval-based methods might require enormous storage space to store all image data (24 images per second for humans) for retrieval purposes. **(2) Model Editing:** This class of methods involves making targeted edits to the model to adapt to new facts while preserving other desired capabilities (Yao et al., 2023). Existing methods primarily focus on fact-based editing, which is typically limited to single sentences. This limitation becomes more severe when one attempts to inject new knowledge in the form of longer and more complicated contexts. **(3) Long Context Methods:** Another alternative solution is to incorporate all the knowledge into the model's context, which essentially makes the context into a knowledge base. This differs from retrieval-based methods in that the context directly informs the inference of the model. Methods in this category involve reducing the complexity of attention operations (Child et al., 2019; Beltagy et al., 2020; Wang et al., 2020), and modifying positional embeddings (Press et al., 2021; Sun et al., 2023) to handle longer contexts. However, as complex reasoning tasks are thirsty for massive up-to-date knowledge, the inevitable context overload with long context methods becomes infeasible, as long as the context length is finite.

In response to the challenges identified above, we introduce MEMORYLLM, a model that embeds a substantial, fixed-size memory pool within its latent space, which serves as the self-updatable parameters. Specifically, we build the memory pool as hidden vectors within each layer of the transformer. At each layer, the memory pool contains *memory tokens* representing compressed knowledge. This design results in a memory pool that is less redundant than traditional knowledge bases in retrieval-based methods or contexts in long-context methods. To update the memory pool, we devise a *self-update* mechanism to propagate the new knowledge to every layer of the memory. During self-update, MEMORYLLM only updates a proportion of memory in each layer to absorb the incoming knowledge. This allows previously stored knowledge to slowly phase out. These designs ensure MEMORYLLM remains up-to-date while the old knowledge is slowly forgotten. After curated training, we update MEMORYLLM nearly a million times without observing any performance deterioration.

The evaluation of MEMORYLLM focuses on several key

---

[*]Work done during the internship at Amazon. [1]UC, San Diego [2]Amazon [3]UC, Los Angeles. Correspondence to: Yu Wang <yuw164@ucsd.edu>, Yifan Gao <yifangao@amazon.com>.

aspects: (1) **Integration of New Knowledge**: The model's performance is assessed with model editing benchmarks and QA tasks (long context QA benchmarks), where MEM-ORYLLM demonstrates substantial improvements over existing methods. (2) **Knowledge Retention Ability**: MEMO-RYLLM is evaluated on long context benchmarks and our knowledge retention experiments, showcasing its ability to recall knowledge. (3) **Robustness**: To test the integrity of the model, we subject MEMORYLLM to almost a million update steps. The results show that our model is functioning properly even after extreme updates.

In summary, our contributions are as follows:

- We introduce MEMORYLLM, which features an integrated memory pool within the latent space of an LLM. This memory pool is designed to manage new knowledge integration and encourage minimal information forgetting while being fixed-sized to circumvent the issue of uncontrolled growth.
- We augment a 7B parameter model with an extensive memory pool comprising 1B parameters.
- MemoryLLM demonstrates strong performance across various benchmarks, including model editing, long-context evaluation, and our knowledge retention experiments, showcasing its versatility and effectiveness in diverse applications.

2. Preliminaries

2.1. Problem Statement

The primary challenge addressed in this paper is: *How should we design a large language model that is capable of efficiently integrating new knowledge while minimizing the degradation of previously learned knowledge?* To make the challenge more specific, we outline several essential properties that we hope to integrate into the new model: **(1) Effiency**: The process of knowledge injection into the model should be streamlined, potentially eliminating the need for back-propagation for efficiency. **(2) Efficacy**: It is crucial to ensure that the knowledge is effectively injected into the model, guaranteeing its impact on the model's performance. **(3) Knowledge Retention**: Our model has a fixed-sized memory pool, implying a constant memorization capacity. This necessitates a mechanism for gradually phasing out older knowledge. **(4) Integrity**: The model must maintain full functionality regardless of the number of updates made to the memory pool. **(5) Non-redundancy**: We aim for more compact storage of knowledge, reducing redundancy, and optimizing memory usage.

2.2. Sketch of MEMORYLLM

To address the above challenges, our rough idea is to design a model denoted as $\mathcal{M}_{\theta,\phi}$ consisting of two sets of

parameters: ϕ and θ. Once we obtain the model, the ϕ parameters should be static, while θ dynamically evolves when encountering new knowledge. This aligns with the intuition that some knowledge within an LLM should never change (persistent truths, encoded by ϕ) and some knowledge is being updated continuously (fresh information, modeled by θ). Specifically, we use an existing large language model (Llama2) to model ϕ, while θ is modeled by the memory pool with the detailed structure in Section 3.1.1. Here we need to design the *self-updating* mechanism of θ that is pivotal to this process. Denoting the new knowledge as x, a text paragraph, the *self-updating* process refers to updating θ in a way that does not compromise the general capabilities of the model while injecting the latest knowledge x into the memory pool θ to obtain a new memory pool θ':

$$\theta' = U(\theta, x) \tag{1}$$

Here U is the update function which takes the memory pool θ and the new knowledge x as input and outputs the new memory pool θ'. Extending this process to multistep updating, consider a scenario with a never-ending context or a series of conversation histories, represented as $(x_1, \cdots, x_n)$, where $x_i, i \in \{1, \cdots, n\}$ is a text paragraph. The model requires the integration of all these contexts, which can be accomplished using the update function I defined in Eq.(1):

$$\theta_n = U(\cdots(U(\theta, x_1), x_n). \tag{2}$$

We define the process *self-updating* as modifying the parameters θ with newly encountered knowledge x, essentially enabling the model to read and assimilate knowledge. This design presents two primary challenges: (1) **Parameter and Interaction Design**: We need to determine the structure for θ and how it should interact with ϕ, The goal is to allow the LLM to effectively use the knowledge from the θ in the generation process. (2) **Update function design**: It is crucial to design the update function U such that θ can be updated without disturbing the old knowledge and undermining the overall capabilities of the model.

3. MEMORYLLM

3.1. Structure Design

3.1.1. MEMORY POOL

We choose to instantiate ϕ with an off-the-shelf LLM, specifically Llama2 (Touvron et al., 2023). ϕ consists of multiple transformer layers, denoted as $\phi = \{\phi_l\}_{l=1}^{L}$, where L represents the total number of layers. To facilitate the transformer ϕ to understand the memory pool θ, we conceptualize θ as hidden vectors within each transformer layer, symbolized as $\theta = \{\theta_l\}_{l=1}^{L}$. Each θ_l is of dimension $N \times d$, corresponding to N hidden states and the word embedding dimension d in ϕ. We term θ_l *memory tokens*. The *memory tokens*
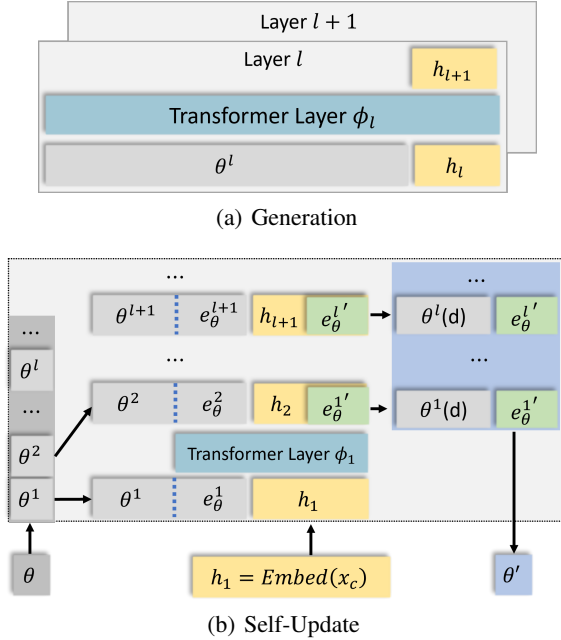
(a) Generation

(b) Self-Update

Figure 1. **The framework of MEMORYLLM.** (a) During generation, all memory tokens in the l-th layer of memory pool θ^l are attended by the hidden states h_l. (b) During self-update, The last k memory tokens from θ_l are taken to be concatenated with the hidden states h_l as the input to ϕ_l. The output h_{l+1} goes to the next layer. The last K tokens of h_{l+1} serve as the new memory tokens $e_\theta^{l\,'}$. We randomly drop K tokens in θ_l and concatenate the left θ_l (denoted as $\theta^l(d)$) with $e_\theta^{l\,'}$ to obtain new memory θ_l'.

serve as the representation of previous knowledge that the model has seen in a more compressed manner. We intend to maximize the memory size, so we assign the memory pool to every layer to significantly enlarge the memory pool. During the generation phase, all memory tokens are used, as illustrated in Figure 1(a). The attention map is designed to enable every token in x to attend to all memory tokens. If x comprises n_x tokens, the attention map assumes a shape of $n_x \times (n_x + N)$, yielding a linear complexity w.r.t. the size of the memory pool.

3.1.2. SELF-UPDATE PROCESS

Figure 1(b) illustrates the self-update process. The goal of self-update is to ensure that MEMORYLLM can always digest the latest knowledge and memorize the previously learned knowledge at its best. We discuss the self-update process in this subsection and prove in section 3.1.3 that MEMORYLLM only forgets stale knowledge at an exponential decay rate with a theoretical guarantee. When introducing new knowledge x_c (in the following, we denote the new knowledge as *context* x_c to distinguish it from x in the last section), the model must integrate x_c into θ as per Eq.(1). To avoid additional modules and complexities, we use the transformer ϕ for the update. Ideally, the input to ϕ_l should be the memory pool θ_l and the hidden states h_l (where h_1 are

the word embeddings of tokenized x_c). We find it essential to maintain the gradient flow from both the self-update and the generation to achieve better performance (see Section 3.2.1). However, it is much more costly to feed the entire pool θ_l to ϕ_l during self-update. To solve this problem, we extract the last K tokens of θ_l where $K << N$ and denote these extracted tokens as e_θ^l. e_θ^l is then concatenated with h_l to form the input of ϕ_l, where h_l can attend to the preceding context e_θ^l. The attention also employs an attention map of dimension $\max(n_{x_c}, K) \times (n_{x_c} + K)$, where n_{x_c} is the number of tokens in x_c (Note that in Figure 1(b) we show the case when $n_{x_c} > K$. The case when $k > n_{x_c}$ is shown in Figure 8 in Appendix). The last K hidden states of the output h_{l+1} are designated as $e_\theta^{l\,'}$. Then we drop K memory tokens from the current memory pool θ^l and squeeze θ^l to the left side of the newly formed memory pool $\theta^{l\,'}$ where the new memory tokens $e_\theta^{l\,'}$ fill the right side. In this way, $e_\theta^{l\,'}$ is still of dimension d, with the new knowledge injected into the last K dimensions.

3.1.3. ANALYSIS OF FORGETTING

The design of the self-update process draws inspiration from the concept of *exponential forgetting* in human cognition, as described by the Ebbinghaus Forgetting Curve, and analogous to the intuition in MemoryBank (Zhong et al., 2023). Through this structure, we aim to simulate exponential forgetting. In each update we drop K tokens from the memory pool; statistically, we drop K/N of the knowledge from the existing memory pool, which means that knowledge within the memory pool would be exponentially forgotten at a rate of K/N. Here, N denotes the total memory size, while K denotes the number of tokens that are used to incorporate knowledge in x_c. Thus, N denotes the total capacity of the memory while K represents the compression ratio. The smaller K, the more compressed the knowledge.

After self-update, the latest knowledge is preserved entirely without any random dropping. After N/K update steps, the retention ratio for knowledge injected N/K steps earlier can be calculated as:

$$(1 - \frac{K}{N})^{N/K}. \tag{3}$$

In Eq. 3, with the memory pool θ_l getting larger (N becomes greater) and the knowledge in x_c getting more compressed (K becomes smaller), we can approach the following limit:

$$\lim_{\frac{N}{K} \to \infty} (1 - \frac{K}{N})^{N/K} = 1/e, \tag{4}$$

where e is the natural constant, therefore, to achieve minimal forgetting, the strategy involves reducing the compression ratio (by minimizing K, as we essentially compress knowledge from h_l into $e_\theta^{l\,'}$) and increasing the memory size (by maximizing N).
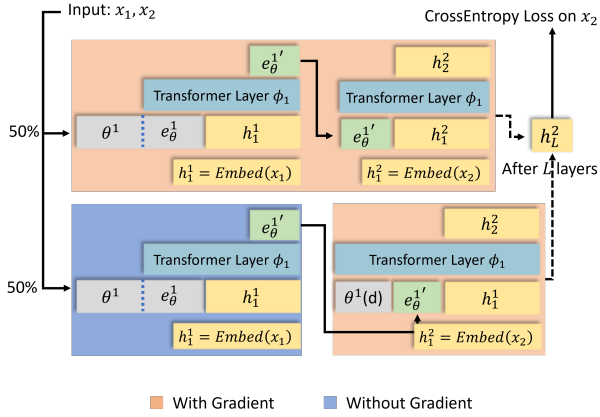
Figure 2. **Training Process for new knowledge incorporation.** During training, we randomly choose one of two shown processes to proceed with 50% probability each. The description pertains to the first layer, and the subsequent layers share an analogous procedure. After sampling (x_1, x_2) from the dataset, we first perform self-update with x_1 as depicted in the left side of both processes. Subsequently, the modified memory $e_\theta^{1\,\prime}$ is employed to predict x_2. Of the two processes, the upper one maintains gradient flow throughout the entire process, optimizing the knowledge compression from x_1 to $e_\theta^{l\,\prime}$ ($l \in \{1, \cdots, L\}$). In contrast, the lower process executes the self-update without gradient. Both processes are designed to encourage the use of the knowledge in the memory pool for the prediction.

3.2. Training Strategy

We adopt the next word prediction task to pretrain our model. Our training methodology for MEMORYLLM is strategically designed to optimize towards three core objectives discussed as follows:

3.2.1. NEW KNOWLEDGE INCORPORATION

The training process begins by selecting a document d from the dataset, which is then divided into two segments (x_1, x_2). Then we update the memory pool θ with x_1, followed by using the updated memory pool to predict x_2. The whole process is described in Figure 2. Ideally, we would design the whole process shown in the lower part of the figure with gradient enabled (see figure 9). However, this approach incurs prohibitive memory demands, especially when the memory pool is large. To mitigate this issue, in l-th layer, we propose to only use $e_\theta^{l\,\prime}$ for the prediction of x_2 rather than the whole updated memory θ_l' when keeping the gradient flow, and use θ_l' when the self-update process with x_1 is performed without gradient. In each iteration, the two aforementioned processes are randomly selected, to ensure that our model can absorb the knowledge in x_1 into θ and use the memory pool θ during the generation.
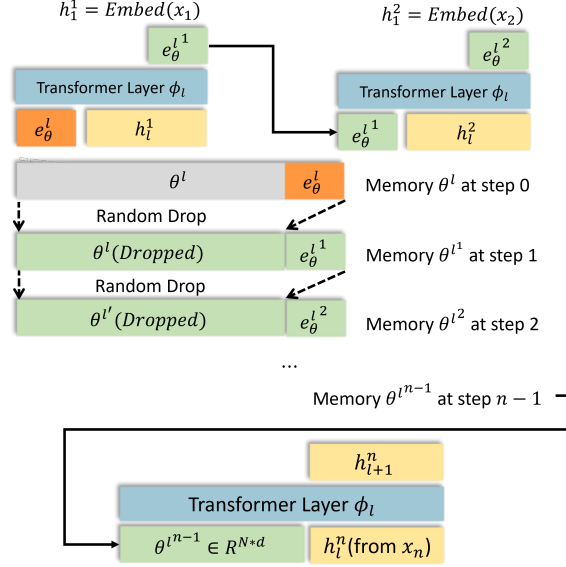
Figure 3. **Training process for continuous contexts understanding.** We only draw two self-update steps here with x_1, x_2 though there should be $n - 1$ self-updates in this training iteration. We show the procedure of l-th layer here. At the bottom of the figure, h_1^n refers to the word embeddings of x_n, and h_L^n is used for loss value calculation. Essentially we are compressing the knowledge from $x_1, \cdots, x_{n-1}$ into $\theta^{l^{n-1}}$ to predict x_n.

3.2.2. ENHANCING CONTINUOUS CONTEXTS UNDERSTANDING

In Section 3.2.1, We encourage the model to understand the latest knowledge injected, where the model can make predictions based on the new memory pool θ'. However, the model only needs the last K tokens of each layer θ_l' since only $e_\theta^{l\,\prime}$ (the last K tokens of θ_l') contains the knowledge of the last injected context. Thus our model may suffer from predicting the next token based on multiple injected contexts, which is essentially the long context problem. We propose a training routine illustrated in Figure 3 to address this problem. In Figure 3, a long document is sampled and segmented into n parts $(x_1, \cdots, x_n)$, with each segment being shorter than a predefined maximum length. The first $n - 1$ segments are then sequentially injected into the memory pool θ using Eq.(2), resulting in θ_{n-1}. Note that this whole injection process of $(x_1, \cdots, x_{n-1})$ is executed with gradient disabled. Upon obtaining θ_{n-1}, we calculate the cross-entropy loss on segment x_n. With this training procedure, we wish to enhance the model's ability to understand and process continuous contexts.

3.2.3. MITIGATING FORGETTING PROBLEMS

To address the forgetting issue, we design a task that involves contexts across multiple documents. Specifically, we sample one main document d and multiple side documents

d' (we take one side document as an example) and split them into segments $(x_1, \cdots, x_n)$ and $(x_1', \cdots, x_n')$. The first $n-1$ segments of the main document $(x_1, \cdots, x_{n-1})$ and the side document $(x_1', \cdots, x_n')$ are then injected into the model sequentially. To force the model to recall the related context injected a long time ago, we make the model predict the last segment of the main document x_n. Similarly, the gradient is disabled during all the injections. We encourage the model to use the knowledge from long ago to make the prediction, thereby mitigating the forgetting problem effectively. The implementation details of this part are described in Appendix B.1.

To maintain the integrity of our model, i.e., to avoid the issue that the model may start malfunctioning after updating θ too many times, we update θ with the context after backpropagation. Specifically, we update θ with x_1 in Section 3.2.1 and with $\{x_1, \cdots, x_{n-1}\}$ in Section 3.2.2 at the end of each training iteration. Intuitively, we are regularizing the distribution of ${e_\theta^l}'$ to be the same as that of θ_l to maintain integrity after arbitrarily many updates.

3.3. Model Instantiation

We use Llama2-7b as ϕ, consisting of 32 layers, with a hidden dimension of $4,096$. The model we propose has $7,680$ memory tokens in every layer, meaning that $\theta \in \mathbb{R}^{32 \times 7680 \times 4096}$, comprising 1.066B parameters.

3.4. Discussions

Extension to Other Architectures: Our experimental framework involves the use of Llama2-7b as the instantiation for the function ϕ. This selection was driven by the popularity and performance of Llama2-7b as a large language model during the development phase of our project. It is important to note, however, that the framework of our model is broadly applicable across various large language models (LLMs) that have transformer architectures with full attention mechanisms.

Scalability of the Memory Size: In our main experiments, we expand the memory size to approximately 1 billion parameters. We wish to emphasize that the efficiency of the self-update process (discussed in Section 3.1.2) remains unaffected by increases in the memory pool size. This efficiency is due to the model's design, which only adopts the most recent K tokens from the memory pool as the input during self-updates. Consequently, the primary scalability constraint arises from the attention mechanism between the memory tokens and the input tokens during generation (as depicted in Figure 1(a)). As the memory pool enlarges, the computational complexity of these attention mechanisms increases **linearly** with respect to the number of tokens N in the memory pool, which is because the complexity of the attention is $N \times K$. With distributed training, our framework has the potential to be scaled to significantly larger memory sizes.

The design of Random Dropping: Random dropping is a fairly straightforward way to keep the size of the memory pool fixed while maintaining an exponential forgetting mechanism. Other possible strategies include applying an exponential decay factor to the memory pool from the previous step and aggregating the decayed memory pool with the new memory. We have experimented with aggregating existing memory and new memory instead of using random dropping. However, we found that maintaining the integrity of hidden states for tokens seems to be beneficial. Aggregating hidden states often disrupts both the original and new knowledge, resulting in situations where even the knowledge injected into the memory during the last self-update process cannot be fully extracted. In contrast, while random dropping carries the risk of forgetting previous information, it allows for the full recovery of information from the context injected during the last self-update step, as there is no random dropping applied to the new memory tokens at the last update. Therefore, we choose random dropping as we believe it provides a more natural way to integrate existing hidden states with new hidden states.

4. Experiments

4.1. Evaluation Protocols

As illustrated in Section 1, we need to evaluate MEMORYLLM in the following three aspects: (1) Integration of New Knowledge: this evaluation is conducted with the model editing tasks (Section 4.3) and QA tasks (long context QA benchmarks, Section 4.4); (2) Knowledge Retention Ability: the model is evaluated with long context QA benchmarks (Section 4.4) and our knowledge retention experiments (Section 4.5); (3) Robustness: we make nearly a million updates to our memory pool and then test the functionality of our model (Section 4.6).

4.2. Implementation Details

We train our model on the processed version of the C4 dataset (Raffel et al., 2020) from Red-Pajama (Computer, 2023). For the training processes in Section 3.2.1, we sample documents from the entire dataset, while the training process in Section 3.2.2 is based on a subset of C4 (we call this the long context subset) where all documents are of length greater than 2048. For the last part, Section 3.2.3, the documents are sampled randomly from the original C4 dataset and the long context subset. The training is performed on 8 A100-80GB GPUs for three days.

4.3. Model Editing

4.3.1. EXPERIMENTAL SETUP

We follow the experimental setup in (Meng et al., 2022). The benchmarks are:

Table 1. **Quantitative Editing Results on Llama2-7B** for ZsRE and CounterFactual Datasets. "w/EF" means "with editing facts", indicating the model after updating the memory pool with the new fact.

Editor	ZsRE Dataset				CounterFactual Dataset			
	Score	Efficacy	Generalization	Specificity	Score	Efficacy	Generalization	Specificity
Llama2-7B	55.6	55.9	54.7	56.3	20.7	13.7	16.6	83.4
MemoryLLM-7B	51.2	50.0	49.1	54.8	22.6	15.7	17.6	82.1
FT	50.3	78.6	80.6	29.0	10.0	99.7	96.9	3.6
FT-L	69.8	81.4	76.8	56.6	33.8	47.2	18.0	83.3
ROME	69.3	88.7	70.2	56.3	69.2	82.6	75.2	55.8
IKE	-	-	-	-	70.7	99.8	96.2	45.4
MemoryLLM-7B (w/ EF)	**79.2**	99.8	96.7	57.1	**75.3**	98.5	82.2	57.0

zsRE (Levy et al., 2017): Zero-Shot Relation Extraction(zsRE) is first used in Mitchell et al. (2022); Cao et al. (2021) for model editing evaluations. we use the first 10, 000 records in the dataset as the evaluation set, with each record containing one factual statement.

CounterFactual (Meng et al., 2022): A set of more difficult false facts in which the LLMs would have low scores when prompted with these facts. Then after editing the model, the model is queried again with these facts. Each example includes the question, the original fact, and the false fact, where we aim to inject the false fact into the model. The first 2, 000 examples in this dataset are used for evaluation (following the evaluation of GPT-J in (Meng et al., 2022)).

Evaluation metrics include **Efficiency** (the post-edit accuracy), **Generalization** (post-edit accuracy of the paraphrased version of the factual statement), and **Specificity** (the post-edit accuracy on unrelated facts). The harmonic mean of the three metrics is reported in column **Score**.

We compare our model with the following baselines[2]: **FT**, **FT-L** (Zhu et al., 2020), **IKE** (Zheng et al., 2023), **ROME** (Meng et al., 2022). The details of the baselines are described in Appendix C.1.

4.3.2. OVERALL PERFORMANCE COMPARISON

The experimental results on dataset ZsRE and CounterFactual are shown in Table 1. From the table, We observe (1) Our model outperforms all baseline models in both datasets, achieving the highest overall performance metrics. (2) While Fine-Tuning (FT) performs better in terms of Efficacy and Generalization, which means the model absorbs the knowledge, but it tends to lag in "Specificity", meaning the model's knowledge of other facts is affected by the tuning. (3) With enforced constraints (FT-L), the model performs better in terms of Specificity while compromising

Efficacy and Generalization, representing that the knowledge is not absorbed by the model effectively. (4) ROME, striking a reasonable balance between efficacy and specificity, compared with MEMORYLLM, may fall short in the overall performance measurement. (5) IKE, conceptually similar to our approach by incorporating the information into the context, faces limitations in specificity, which could be ascribed to the complexity of the prompts used in the implementation, potentially disrupting the accuracy.

4.4. Long Context Evaluation

4.4.1. EXPERIMENTAL SETUP

In this section, we evaluate the long-context modeling capabilities of our model. Our assessment utilizes the Long-Bench benchmark (Bai et al., 2023), specifically designed to test performance in long-context scenarios. Since our model has not undergone instruction fine-tuning, the baselines for comparison are also not instruction finetuned. The baselines include **Llama2-7B**: our backbone; **Longllama-3B-v1-1**(Tworkowski et al., 2023): The model employs contrastive learning to extend the effective context length of existing models. We adopt the 3B model here as only the 3B model is open-sourced, derived from Openllama-V2; **Openllama-V2** (Geng & Liu, 2023): An open-sourced reproduction of Llama. **Llama2-LongLora-7B-16k** (Chen et al., 2023b): A novel attention mechanism, Shift Short Attention, is proposed and used for longer context pertaining. This model, based on Llama2-7B, is extended to accommodate a 16k context length; **Llama2-LongLora-7B-100k** (Chen et al., 2023b). The same method but context length is extended to 100k. For 7B models, we omit the results of maximum length being 16, 384 as we encounter the out-of-memory (OOM) error even when using eight A100-80GB GPUs. This shows another advantage of MEMORYLLM, as our model needs one 48 GB GPU or two 40GB GPUs to run inference regardless of the input length.

4.4.2. OVERALL PERFORMANCE COMPARISON

The results in Figure 4 reveal that: (1) MEMORYLLM outperforms baselines in four out of six datasets when provided

---

[2]We tried MEND (Mitchell et al., 2022), but the existing code is for GPT-style models. Our re-implementation of MEND on Llama2 based on the published code constantly encounters `nan`. In addition, MEND is inferior in the experiments in (Meng et al., 2022), thus we omit MEND in our experiments.

Table 2. The performance comparison on long context QA benchmarks of our model with and without BM25 retriever.

	MEMORYLLM-16k	MEMORYLLM-all-BM25
narrativeqa	**20.64**	15.60
qasper	19.57	**20.30**
multifieldqa_en	29.56	**33.08**
hotpotqa	**34.03**	32.27
2wikimqa	**27.22**	24.17
musique	13.47	**15.36**

with extended contexts. However, a notable exception is observed in the Qasper dataset, where MEMORYLLM exhibits suboptimal performance. This could be attributed to the model's training predominantly on the C4 dataset, without incorporating the arxiv dataset. Thus, the training may affect the model's ability on scientific datasets (such as Qasper). (2) As the context length grows, the performance of MEMORYLLM continues to improve, demonstrating the knowledge retention ability of MEMORYLLM, where the knowledge from multiple updates earlier could boost performance. The performance of MEMORYLLM, when the context length is less than 4k, is not the same as that of Llama2-7B, which can be attributed to the subset we used for training MEMORYLLM, as we do not need to use the entire dataset for pertaining Llama2-7B for our model and a subset would inevitably have distribution shift from the original dataset.

4.4.3. COMPARISON WITH RAG METHODS

In this section, we aim to explore the role of RAG methods in QA tasks which we argue is orthogonal to MEMORYLLM. The primary goal of MemoryLLM is to achieve self-updatable LLM where the memory module serves as the parameters that could keep updating along the inference process, whereas RAG methods aim to retrieve the most relevant piece of information from the history. Intuitively, RAG is used to conduct coarse retrieval from millions of documents, while MemoryLLM can process the retrieved documents. We use BM25 retriever to extract 4k tokens from the whole context and use MemoryLLM to process these 4k tokens to generate the answer. The results are shown in Table 2. Here MemoryLLM-7b-16k corresponds to the results in Figure 4, and MemoryLLM-7b-all-BM25 means retrieving 4k tokens from the whole given context and using MemoryLLM to process the retrieved 4k tokens. The results show that using the BM25 retriever could enhance the model performance on certain datasets while not universally beneficial.

4.5. Knowledge Retention Experiments

4.5.1. EXPERIMENTAL SETUP

The datasets are prepared as below:
SQuAD: Formatted as `(context, question, answer)`, where `context` and `question` are sen-
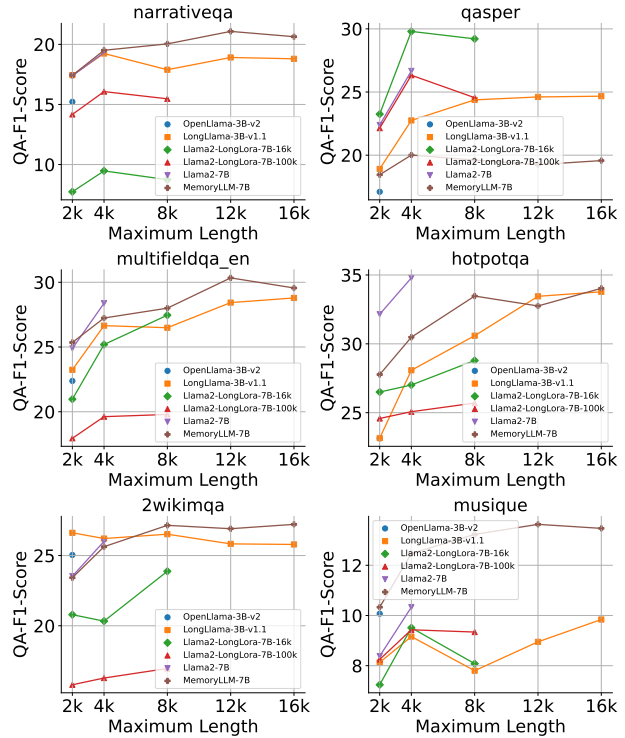

Figure 4. **Experimental Results on LongBench**. The x-axis is the maximum context length for the QA task. For instance, with a maximum length of 4096, we truncate 4096 tokens from the given context as input to the model. The y-axis is the F1 score.

tences, `answer` refers to the first answer in the list of ground-truth acceptable answers. Then we extract all the samples with `answer` shorter or equal to 3 tokens. The model generates 10 new tokens from the prompt "Question: `Question` Answer:". Correct predictions cover the 3-token answer within the 10 generated tokens. A total of $2,250$ samples are used for the accuracy calculation.
NaturalQA: Formatted as `(context, question, answer)`, using the long answer as the context and the short answer as the ground truth. Samples with answers of 4 tokens or less are selected. Like SQuAD, the model generates 10 new tokens, and the correct predictions cover the 4-token answer. This yields 1,004 samples for analysis.
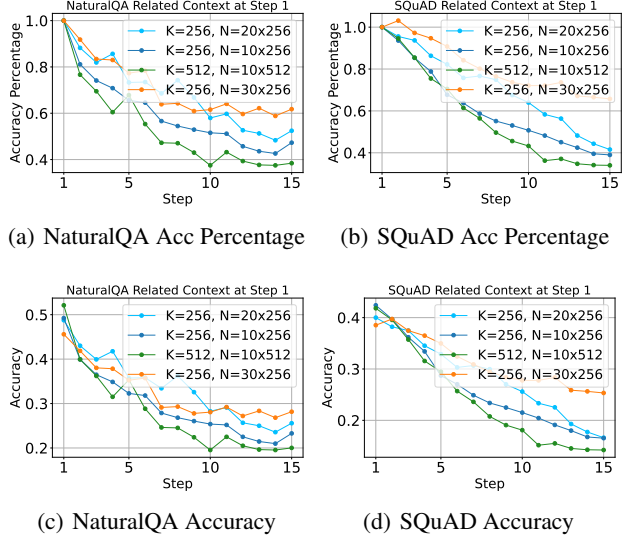
The results are shown in Figure 5. We assess MEMORYLLM's forgetting rate, comparing it against a baseline (accuracy without context injected into the memory) and a theoretical upper bound. Denote the accuracy at step 1 as a_u, and the borderline accuracy as a_b. Then at step t, we calculate the point on the curve with the following equation:

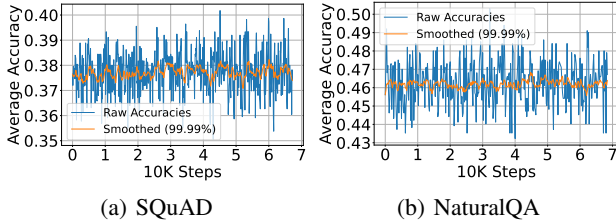$$a_t = (a_u - a_b) * \left(\frac{N - K}{N}\right)^{t-1} \qquad (5)$$

In our instantiation, $N = 7,680$ and $K = 256$. Our findings indicate that the model retains knowledge even after 20 updates. However, it falls short of the exponential decay curve
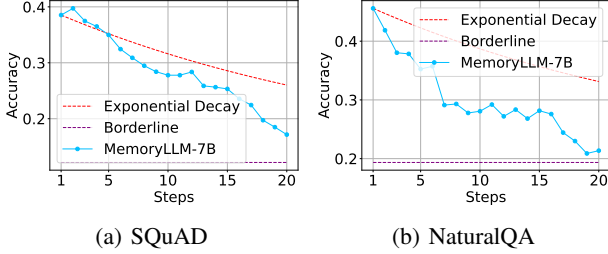
(a) SQuAD (b) NaturalQA

Figure 5. **Performance Comparison on SQuAD and NaturalQA.** The x-axis shows the number of updates we perform on the model, where the context that contains the knowledge to answer the question is injected in Step 1. The y-axis reveals the accuracy of the model's prediction after a certain number of updates. The accuracy is higher than the borderline indicating that the knowledge is not completely forgotten, while we wish the model to be more aligned with the exponential decay, i.e., the theoretical upper bound.

(a) SQuAD (b) NaturalQA

Figure 6. **Model Integrity Check** with SQuAD and NaturalQA. We plot accuracy along the updating process as well as the exponential moving average as the *Smoothed* (99.99%) value. **We do not observe any decrease over 650k updates.**

representing the upper bound. This gap can be attributed to the fact that even if the knowledge is partially corrupted after 20 steps of updating, it might be hard for the model to reveal the exact answer. The performance exceeding the upper bound at step 2 on the dataset SQuAD might be due to (a) the variation of inference and (b) dropping a small part of the memory may not affect the model predicting the words, while the exponential curve would drop.

4.6. Model Integrity Analysis

To illustrate the integrity of our model, We update our model with NaturalQA and SQuAD mentioned in Section 4.5. Each time we go through the whole dataset, we shuffle the dataset and inject it into the memory again. In this way, we can simulate infinite updates. Then during the updating process, we track if our model could answer the question related to the most recent context, obtaining a long binary array that indicates whether our model successfully answers the question related to the most recently injected context. With this binary array, we calculate the average accuracy of the last $2,250$ samples for SQuAD and the last $1,004$ samples for NaturalQA. The results are shown in Figure

(a) NaturalQA Acc Percentage (b) SQuAD Acc Percentage

(c) NaturalQA Accuracy (d) SQuAD Accuracy

Figure 7. **Ablation Study** with our knowledge retention experiments on NaturalQA and SQuAD. All models are trained with the same setting, 30×256 is our main model. The relevant knowledge for answering the question is injected in step 1, and the x-axis means the number of updates (steps) performed. The top figures show the ratio of the accuracy at each step compared with the accuracy at step 1 for better visualization of knowledge retention.

6. We continue running for up to $650,000$ steps for 3 days. As shown in the figure, there is no sign of decreasing in accuracy even after the $650,000$ steps, demonstrating the integrity of our model. From this observation, we argue that our model could be potentially updated for arbitrarily many times without affecting the functioning ability.

4.7. Ablation Study

4.7.1. ABLATION STUDY OF DIFFERENT K AND N

In this section, we study the effects of different K and N in Eq.(3) with our knowledge-retention experiments. Our primary goal is to explore the forgetting ratio when the model has different memory sizes (N) and numbers of tokens to store the new knowledge (K). We vary N to be $\{10 \times 256, 20 \times 256, 30 \times 256\}$, and K to be $\{256, 512\}$. We also tried $K = 128$, but to find that the accuracy is much worse than the other settings (the step 1 accuracy of NQA and SQuAD under the setting $K = 128$ are 0.34 and 0.25, respectively), we omit this setting here. The results shown in Figure 7 reveal that (1) When K is fixed, with greater N, the forgetting ratio is smaller; (2) When N is fixed, with smaller K (10×512 vs. 20×256, the latter yields better knowledge-retention ability), the forgetting ratio becomes smaller. These experiments support our intuition and show that with the improvement of N/K, we can enable better knowledge-retention ability.

4.7.2. ABLATION STUDY OF THE MODEL STRUCTURES

In our main experiments, we train the model with the memory tokens augmented in every layer. To study the necessity of this design, we tried the following several settings: (1) Augment only one layer in the model with memory tokens; (2) Augment the last half of the layers in the model with the memory tokens (this design is inspired by Figure 6(a) in Fang et al. (2024)). Then we find that design (1) leads to almost zero improvements with the context compared to the performance without the context, which means augmenting only one layer is almost useless. For design (2), We record the accuracies after injecting the context for one step: NaturalQA: 0.39, SQuAD: 0.22. For reference, the accuracy of NaturalQA and SQuAD in Figure 7 at step 1 is 0.46 and 0.39 respectively. This shows that having the memory tokens in both the first half and the second half layers is necessary for better performance.

5. Related Work

5.1. Memory based methods

Previous memory-based methods share certain similarities with MEMORYLLM. Among these methods, some use an external encoder to inject knowledge into the memory pool, such as the Memory Network (Weston et al., 2014), which focuses on rectifying the forgetting problems in RNNs. Follow-up work Sukhbaatar et al. (2015) computes the weighted sum of the entire memory pool as the representative vector of the memory. Others use the language model itself as the encoder to update the memory. Memory Transformer (Burtsev & Sapunov, 2020) and RMT (Bulatov et al., 2022) propose to add memory tokens when reading the contexts, where the memory pool is up to 20 tokens. EMMA (Moro et al., 2023) has a slightly larger memory pool, which is the size of the chunk when injecting the contexts into the memory. These fixed-sized memory pools show promising results, although performance is limited by the size of the memory pool. This also shows the challenges of expanding memory and incorporating information without disturbing the original capability of the model.

Other memory-based methods integrate the memory pool with unfixed size, where different forgetting mechanisms are adopted to handle the ever-growing problem. In this case, the memory pool would be in the form of (1) hidden states, such as (Adel, 2022) and MemoryBank (Zhong et al., 2023); (2) key-value pairs, represented by KNN-LM (Khandelwal et al., 2019), LONGMEM (Wang et al., 2023). (3) vectors in hidden space. This involves the image captioning task (Cornia et al., 2020) and Memformer (Wu et al., 2022). (4) raw texts. RET-LLM (Modarressi et al., 2023) proposes to save the knowledge with triplets into the memory and then use API query to retrieve related information in the

memory given the context. These methods have a more flexible memory pool. However, the memory pool might be redundant in terms of the stored knowledge.

5.2. Downsteam Tasks

As MEMORYLLM has a large memory pool that can be used to store knowledge, it could be used for downstream tasks such as model editing and long context tasks.

For model editing tasks (Yao et al., 2023), MEND (Mitchell et al., 2022) and ROME (Meng et al., 2022) propose to modify the parameters of the LLM with the new given fact. During inference, MEND needs back-propagation and ROME requires the optimization for new MLP weights, while MEMORYLLM, regarding the memory pool as part of the model parameters, could directly update the memory pool to store new facts. IKE (Zheng et al., 2023) proposes to simply put the new facts in context, which is straightforward and intuitively similar to MEMORYLLM in terms of this task. However, IKE would encounter the same problem as long context methods, i.e., the ever-growing contexts.

For Long context tasks, representative methods can be categorized as follows: (1) Efficient Attention such as Longformer (Beltagy et al., 2020), Linformer (Wang et al., 2020), LongNet (Jiayu et al., 2023), (2) Positional Encoding like Alibi (Press et al., 2021), Positional Interpolation (Chen et al., 2023a) and Extrapolation (Sun et al., 2023), (3) Fine-tuning with longer context (Xiong et al., 2023; Tworkowski et al., 2023), (4) Memory-based methods (Wang et al., 2023; Bulatov et al., 2022; Wu et al., 2022). Among all these categories, MEMORYLLM could fit into the fourth category where long contexts are absorbed into the memory, which is used for future prediction.

6. Conclusion and Future Work

In this paper, we propose MEMORYLLM, a language model consisting of a transformer and a huge memory pool within the latent space of the transformer, which serves as the self-updatable parameters of the model. MEMORYLLM can perform self-updates on the memory with new knowledge, enabling effective knowledge incorporation and slow forgetting of previous knowledge. Comparisons against baselines for model editing and long context, together with a dedicated customized evaluation for knowledge retention analysis, demonstrate the superiority of MEMORYLLM in effectively absorbing new knowledge and knowledge retention ability. In the future, it is of interest to extend the memory size as well as increase the compression rate, i.e., using fewer memory tokens during self-update to store the new knowledge. In addition, we aim to extend MEMORYLLM to be multimodal, as the memory tokens of MEMORYLLM may be suitable for storing multimodal knowledge.

Impact Statement

This paper presents work that aims to advance the field of Natural Language Processing, specifically the Large Language Models. There are many potential societal consequences of our work associated with LLMs, such as AI safety and reliability. Beyond LLMs, we feel no other consequences must be highlighted here.

References

Adel, A. A. Global memory transformer for processing long documents. *CoRR*, abs/2212.01650, 2022.

Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020. URL https://arxiv.org/abs/2004.05150.

Bulatov, A., Kuratov, Y., and Burtsev, M. S. Recurrent memory transformer. In *NeurIPS*, 2022.

Burtsev, M. S. and Sapunov, G. V. Memory transformer. *CoRR*, abs/2006.11527, 2020. URL https://arxiv.org/abs/2006.11527.

Cao, N. D., Aziz, W., and Titov, I. Editing factual knowledge in language models. In *EMNLP (1)*, pp. 6491–6506. Association for Computational Linguistics, 2021.

Chen, S., Wong, S., Chen, L., and Tian, Y. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023a.

Chen, Y., Qian, S., Tang, H., Lai, X., Liu, Z., Han, S., and Jia, J. Longlora: Efficient fine-tuning of long-context large language models. *arXiv preprint arXiv:2309.12307*, 2023b.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019. URL http://arxiv.org/abs/1904.10509.

Computer, T. Redpajama: an open dataset for training large language models, 2023. URL https://github.com/togethercomputer/RedPajama-Data.

Cornia, M., Stefanini, M., Baraldi, L., and Cucchiara, R. Meshed-memory transformer for image captioning. In *CVPR*, pp. 10575–10584. Computer Vision Foundation / IEEE, 2020.

Fang, J., Tang, L., Bi, H., Qin, Y., Sun, S., Li, Z., Li, H., Li, Y., Cong, X., Yan, Y., et al. Unimem: Towards a unified view of long-context large language models. *arXiv preprint arXiv:2402.03009*, 2024.

Geng, X. and Liu, H. Openllama: An open reproduction of llama, May 2023. URL https://github.com/openlm-research/open_llama.

Jiayu, D., Shuming, M., Li, D., Xingxing, Z., Shaohan, H., Wenhui, W., and Wei†, F. Longnet: Scaling transformers to 1,000,000,000 tokens. *arXiv preprint arXiv:2307.02486*, 2023.

Khandelwal, U., Levy, O., Jurafsky, D., Zettlemoyer, L., and Lewis, M. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172*, 2019.

Levy, O., Seo, M., Choi, E., and Zettlemoyer, L. Zero-shot relation extraction via reading comprehension. In Levy, R. and Specia, L. (eds.), *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017), Vancouver, Canada, August 3-4, 2017*, pp. 333–342. Association for Computational Linguistics, 2017. doi: 10.18653/V1/K17-1034. URL https://doi.org/10.18653/v1/K17-1034.

Meng, K., Bau, D., Andonian, A., and Belinkov, Y. Locating and editing factual associations in gpt. *Advances in Neural Information Processing Systems*, 35:17359–17372, 2022.

Mitchell, E., Lin, C., Bosselut, A., Finn, C., and Manning, C. D. Fast model editing at scale. In *ICLR*. OpenReview.net, 2022.

Modarressi, A., Imani, A., Fayyaz, M., and Schütze, H. Retllm: Towards a general read-write memory for large language models. *arXiv preprint arXiv:2305.14322*, 2023.

Moro, G., Ragazzi, L., Valgimigli, L., Frisoni, G., Sartori, C., and Marfia, G. Efficient memory-enhanced transformer for long-document summarization in low-resource regimes. *Sensors*, 23(7):3542, 2023.

Press, O., Smith, N. A., and Lewis, M. Train short, test long: Attention with linear biases enables input length extrapolation. *arXiv preprint arXiv:2108.12409*, 2021.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

Sukhbaatar, S., Weston, J., Fergus, R., et al. End-to-end memory networks. *Advances in neural information processing systems*, 28, 2015.

Sun, Y., Dong, L., Patra, B., Ma, S., Huang, S., Benhaim, A., Chaudhary, V., Song, X., and Wei, F. A length-extrapolatable transformer. In *ACL (1)*, pp. 14590–14604. Association for Computational Linguistics, 2023.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton-Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

Tworkowski, S., Staniszewski, K., Pacek, M., Wu, Y., Michalewski, H., and Miłoś, P. Focused transformer: Contrastive training for context scaling. *arXiv preprint arXiv:2307.03170*, 2023.

Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Wang, W., Dong, L., Cheng, H., Liu, X., Yan, X., Gao, J., and Wei, F. Augmenting language models with long-term memory. *arXiv preprint arXiv:2306.07174*, 2023.

Weston, J., Chopra, S., and Bordes, A. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

Wu, Q., Lan, Z., Qian, K., Gu, J., Geramifard, A., and Yu, Z. Memformer: A memory-augmented transformer for sequence modeling. In *AACL/IJCNLP (Findings)*, pp. 308–318. Association for Computational Linguistics, 2022.

Xiong, W., Liu, J., Molybog, I., Zhang, H., Bhargava, P., Hou, R., Martin, L., Rungta, R., Sankararaman, K. A., Oguz, B., et al. Effective long-context scaling of foundation models. *arXiv preprint arXiv:2309.16039*, 2023.

Yao, Y., Wang, P., Tian, B., Cheng, S., Li, Z., Deng, S., Chen, H., and Zhang, N. Editing large language models: Problems, methods, and opportunities. *CoRR*, abs/2305.13172, 2023.

Zheng, C., Li, L., Dong, Q., Fan, Y., Wu, Z., Xu, J., and Chang, B. Can we edit factual knowledge by in-context learning? *arXiv preprint arXiv:2305.12740*, 2023.

Zhong, W., Guo, L., Gao, Q., and Wang, Y. Memorybank: Enhancing large language models with long-term memory. *arXiv preprint arXiv:2305.10250*, 2023.

Zhu, C., Rawat, A. S., Zaheer, M., Bhojanapalli, S., Li, D., Yu, F. X., and Kumar, S. Modifying memories in transformer models. *CoRR*, abs/2012.00363, 2020.

A. Details in Methodology

A.1. Self-Update Process

In Section 3.1.2, we illustrate the self-update process with the scenario of the input context x_c having more than K tokens. As for the case when x_c has less than K tokens, we draw the process in Figure 8. As shown in this figure, we input e_θ^l and h_l into the transformer layer ϕ_l to obtain $e_\theta^{l'}$ where the last n_{x_c} tokens are passed into the next layer.
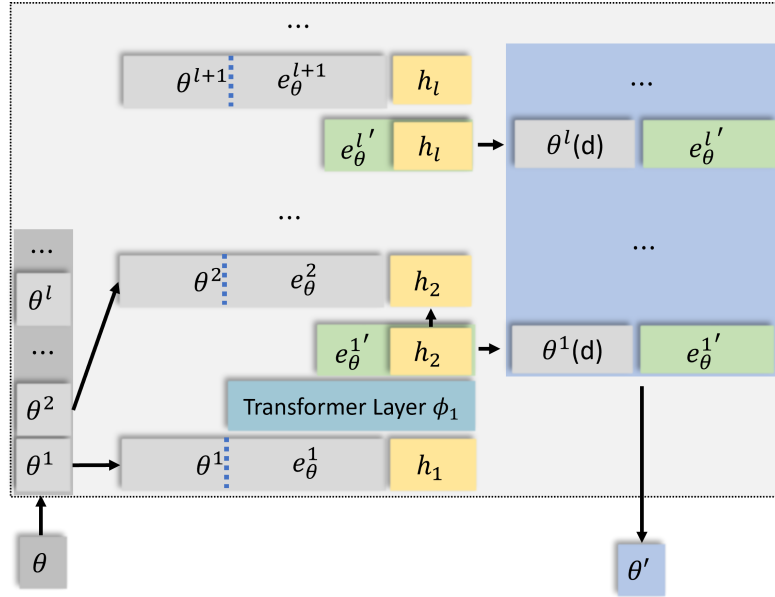

Figure 8. Self-Update process when the number of tokens is smaller than the number of memory tokens needed.

A.2. Training Strategy for New Knowledge Incorporation

As shown in Figure 9, compared with Section 3.2.1, the ideal case is to perform the whole process, i.e., self-update and the prediction on x_2, with gradient flow, so that the cross-entropy loss could be backpropagated to x_1. However, this would induce unaffordable memory consumption, thus we decompose this process into two processes in Figure 2.
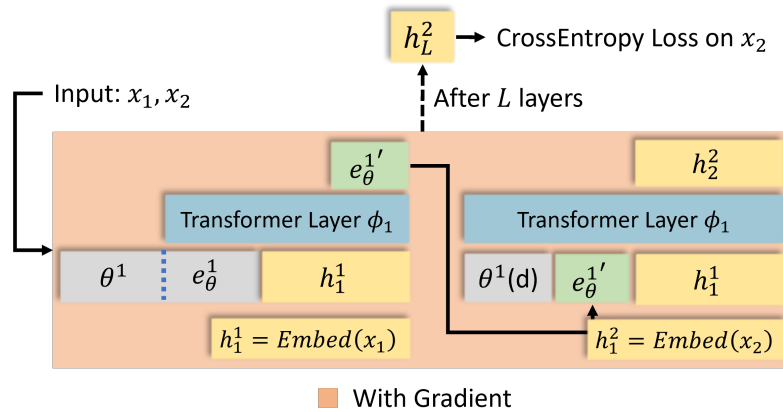

Figure 9. Ideal Training Routine for Latest Knowledge Incorporation

B. Implementation Details

B.1. Details for Mitigating Forgetting Problems

As mentioned in Section 3.2.3, we need to sample one main document $d = \{x_1, \cdots, x_n\}$ and multiple side documents and inject all the side documents into the memory after the injection of $\{x_1, \cdots, x_{n-1}\}$, then we calculate the loss on x_n to update the model. However, instead of sampling multiple documents at each step, we develop a more efficient strategy during training. We provide the pseudo-code in Algorithm 1.

Algorithm 1 Training Strategy for Mitigating Forgetting Problems

---

Require: Training data $\mathcal{D}$;

1: Initialize the indicator $r_0 = 1$, $l = 0$;
2: Initialize the cache x_{cache} = None;
3: **for** $d \in \mathcal{D}$ **do**
4:     n = the number of contexts in d;
5:     $\{x_1, \cdots, x_n\} = d$;
6:     **if** $r_0 == 1$ or $l == 0$ **then**
7:        $r = 0$;
8:     **else**
9:        r = Random(0, 2);
10:     **end if**
11:     **if** $r == 0$ and $r_0 == 0$ **then**
12:        Inject $\{x_1, \cdots, x_{n-1}\}$ into the memory pool;
13:        Calculate the cross-entropy loss on x_n and update the model;
14:        $l += n$;
15:     **else if** $r == 0$ and $r_0 == 1$ **then**
16:        Inject $\{x_1, \cdots, x_{n-1}\}$ into the memory pool;
17:        Calculate the cross-entropy loss on x_n and update the model;
18:        $x_{cache} = x_n$;
19:        $l += n$;
20:     **else if** $r == 1$ **then**
21:        Calculate the cross-entropy loss on x_{cache} and update the model;
22:        $l = 0$;
23:     **end if**
24:     $r_0 = r$;
25: **end for**

Note that at every step, we inject the knowledge into the memory pool, thus after a random number of steps, the useful knowledge for predicting x_{cache} must be somewhere in the memory pool, we need to encourage the model to extract the relevant knowledge. If the model could extract the knowledge from the memory that was injected long ago, we could mitigate the forgetting problems.

C. Additional Experiments

C.1. Baselines for Model Editing

We introduce the details of the baselines for the model editing experiments here:

FT (Finetuning): which applies Adam with early stopping at one layer to finetune the model on the given fact.

FT-L (Constrained Finetuning) (Zhu et al., 2020): a parameter-space L_∞ norm constraint is imposed on the weight changes.

IKE (In-context knowledge editing) (Zheng et al., 2023): The facts used to edit the model are saved in the contexts, which are inputted into the model during inference. This method is only implemented on CounterFactual so we compare our model with it on the CounterFactual benchmark.

ROME (Rank-One Model Editing) (Meng et al., 2022): After identifying that MLPs in LLMs are the major modules for saving knowledge, ROME proposes to alter the MLP matrix by regarding the matrix as a key-value store and then insert a new key-value pair into the matrix, obtaining a new one that contains the injected information.