Understanding and Alleviating Memory Consumption in RLHF for LLMs

Jin Zhou, Hanmei Yang, Steven (Jiaxun) Tang, Mingcan Xiang, Hui Guan
University of Massachusetts Amherst, MA, USA
{jinzhou, hanmeiyang, jtang, mingcanxiang, huiguan}@umass.edu

Tongping Liu
ByteDance Inc., San Jose, CA, USA
tongping.liu@bytedance.com

Abstract

Fine-tuning with Reinforcement Learning with Human Feedback (RLHF) is essential for aligning large language models (LLMs). However, RLHF often encounters significant memory challenges. This study is the first to examine memory usage in the RLHF context, exploring various memory management strategies and unveiling the reasons behind excessive memory consumption. Additionally, we introduce a simple yet effective approach that substantially reduces the memory required for RLHF fine-tuning.

1 Introduction

Reinforcement Learning from Human Feedback (RLHF) helps align large language models (LLMs) with human values and expectations [1, 2, 3], ensuring that the models produce more accurate, relevant, and contextually appropriate outputs. The RLHF process includes multiple inference and training phases, which involve a total of four models, leading to high memory consumption [4, 5, 6]. Efficient memory management techniques are necessary to enable the practical deployment of RLHF.

Improving memory efficiency during training and inference is a well-studied topic. Previous studies have proposed different memory management policies to reduce memory consumption, such as Zero Redundancy Optimizers (ZeRO) [7, 8], gradient checkpointing [9], and CPU offloading [10], as discussed in §2.2. Additionally, some work focuses on reducing the memory consumption of inference, especially for key-value caching [11, 12]. These memory management strategies are typically combined together to optimize the memory consumption, as they are orthogonal in theory. However, *based on our experiments, some strategies actually introduce higher memory consumption, instead of reducing it.*

Therefore, it is crucial to understand the reason behind this, and how each memory management strategy may inadvertently affect the overall memory consumption. This knowledge will enable the development of optimized memory management strategies and impact the cost of computational resources, making the RLHF fine-tuning of LLMs more accessible and sustainable.

For this purpose, this work provides the first study on memory usage in the RLHF scenario. During this study, we focus on two open-source RLHF frameworks, including DeepSpeed-Chat [13] and ColossalChat [14]. Our study also includes two types of LLM models, aiming to answer the following technical questions:

- **R1:** What causes high memory consumption during RLHF?
- **R2:** How effective are different memory management strategies?
- **R3:** How can we effectively reduce memory consumption in RLHF?

Overall, this study explains why enabling certain strategies may introduce counter-intuitive effects in memory reduction, and further proposes a simple yet effective method that helps reduce memory consumption in RLHF training. The proposed approach minimizes memory consumption without requiring substantial code changes or redesigns, thus offering an efficient solution with minimal effort.

2   Background

2.1   RLHF Training

RLHF [1] is a widely used technique to enhance LLMs by incorporating human preferences into the training process. It involves three stages, each requiring fine-tuning of the LLM: (1) supervised fine-tuning (SFT) on an instruction-following dataset; (2) training a reward model on human preference data; and (3) fine-tuning via proximal policy optimization (PPO) [15]. Our work focuses on the third stage, which is resource-intensive due to the need to manage several large models: the SFT reference model to prevent reward divergence, the reward model for calculating sequence returns, the actor model (final RLHF-aligned LLM) initialized from the reference model, and the critic model initialized from the reward model to estimate returns. In this stage, the actor generates responses to prompts, which are then evaluated by multiple inferences (actor, reference, critic, and reward) to produce experience data. Finally, the actor model is trained to maximize rewards while minimizing policy deviation, and the critic model is trained to ensure alignment with human preferences.

2.2   Memory Management in Training

Various strategies can reduce memory consumption during model training. ZeRO [7, 8] minimizes data redundancy in distributed training by partitioning optimizer states, gradients, and model parameters. CPU Offloading [10] moves some data to CPU memory, thereby reducing GPU memory consumption. Gradient checkpointing [9] trades computation for reduced memory usage by storing only partial activations and recomputing the rest. Existing RLHF training systems often rely on open-source LLM training frameworks that incorporate these memory optimization techniques. For instance, ColossalChat [14] uses Colossalai [16], while DeepSpeed-Chat [13], trlX [17], APP [18], and LLaMA-Factory [19] use DeepSpeed [20].

These training frameworks are typically built on top of PyTorch, relying on its CUDA caching allocator for managing memory usage. Pytorch allocator exposes two parameters that help to understand memory usage: **reserved memory** refers to the total amount of GPU memory that has been reserved by the PyTorch CUDA caching allocator from the CUDA driver, and **allocated memory** refers to the amount of GPU memory that is currently being used by active tensors.

3   Detailed Studies

This section develops the experiments to answer the three research questions in §1, based on two open-source RLHF projects. In the remainder of this section, the size of fragmentation is equal to the difference between reserved memory and allocated memory when the PyTorch allocator attempts to allocate more memory from the CUDA driver.

Hardware Platform: Our experiments was performed on a machine with 2 Intel(R) Xeon(R) Silver 4214R CPUs and 376GB DRAM, and 4 NVIDIA GeForce RTX 3090 GPUs, each with 24GB of HBM memory.

Workload and Setting: For DeepSpeed-Chat, we evaluated OPT [21], where the Actor and Reference model are OPT-1.3b, and the Critic and Reward are OPT-350m model. For ColossalChat, we tested OPT with the same size. For GPT-2 [22], the Actor and Reference are GPT2-xl, and the Critic and Reward are GPT2-medium.

For each framework, we use its default input data, and we set the LoRA [23] dimension to 128. We set the batch size to 2 for DeepSpeed-Chat, and 32 for ColossalChat. ColossalChat offloads the inference models to the CPU during actor and critic training.

3.1 What Causes High Memory Consumption during RLHF?

We ran DeepSpeed-Chat with ZeRO, CPU offloading, and gradient checkpointing enabled – hereafter referred to as memory management strategies – and profiled the memory usage as shown in Figure 1. In this figure, "reserved memory w/o fragmentation" is calculated by subtracting the size of fragmentation from the reserved memory (the yellow line). The difference between the peak reserved memory and "reserved memory w/o fragmentation" is referred to as **memory fragmentation overhead**.
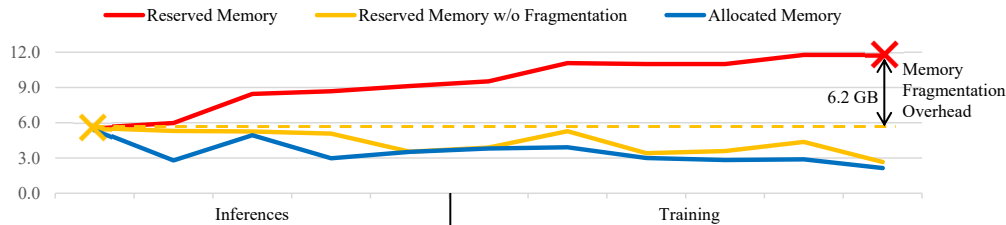

Figure 1: Memory usage (GB) of DeepSpeed-Chat running OPT with multiple memory management strategies enabled. The red cross marks the peak of reserved memory, while the yellow cross and dotted yellow line mark the theoretical peak of reserved memory after subtracting the size of memory fragmentation.

In this experiment, the peak memory usage appears in the training phase where the reserved memory size is much larger than the allocated memory size. We further found that the gap between the reserved and allocated memory is due to external memory fragmentation (hereafter referred to as fragmentation). In this example, the memory fragmentation overhead is 6.2 GB, increasing the memory consumption by 46%.

To determine whether significant fragmentation accumulates from the prior tasks, we compared the memory consumption of the three scenarios: (1) running both inferences and training; (2) training the actor and critic models with pre-collected data; (3) training only the actor model with pre-collected data. When only performing the training phases, we observed smaller fragmentation and reserved memory. Therefore, most fragmentation is accumulated from inferences to the training. In §3.3, we further confirmed that the inferences generate the most fragmentation, which introduces most of the memory consumption to RLHF.

> **Insight:** RLHF introduces high memory consumption because of memory fragmentation. The memory consumption reaches its peak during the training, but mostly due to a large size of memory fragmentation accumulated from inferences.

3.2 How Effective are Different Memory Management Strategies?

We profiled the memory usage of DeepSpeed-Chat and ColossalChat with different strategies enabled, as shown in Table 1. Note that ColossalChat does not support ZeRO-1. In addition, ColossalChat fails in the gradient synchronization when all strategies are enabled, so we excluded those cases. For each of the strategies, we have the following observations:

ZeRO-1: Based on our investigation, ZeRO-1 does not increase the memory fragmentation overhead, and it stably reduces memory consumption (the reserved memory).

ZeRO-2: ZeRO-2 of DeepSpeed-Chat can slightly increase the fragmentation. However, it still reduces memory consumption.

ZeRO-3: ZeRO-3 increases the memory fragmentation overhead. In DeepSpeed-Chat, it causes the memory consumption even greater than those with ZeRO-1 and ZeRO-2, mostly due to memory fragmentation.

CPU Offloading: Based on our observation, CPU offloading can affect the size of fragmentation. However, it can effectively reduce the memory consumption.

Gradient Checkpointing: It may slightly increase the memory fragmentation overhead, but it is still effective in reducing memory consumption, except for ColossalChat with GPT-2. We further figured

Table 1: Memory usage under different memory management strategies. "Reserved" shows the peak of reserved memory size, "Frag." column shows the size of memory fragmentation, and "Allocated" column lists the peak size of allocated memory. We highlighted the strategies (with red color) that increase the memory fragmentation overhead, and the cases (with bold format) where `empty_cache()` is effective in reducing the fragmentation.

Framework	Model	Strategy	Original		Allocated	Using `empty_cache()`	
			Reserved	Frag.		Reserved	Frag.
DeepSpeed-Chat	OPT	None	18.8	0.2	18.2	19.4	<0.1
		ZeRO-1	15.6	0.1	14.4	15.9	0.1
		ZeRO-2	14.5	0.6	12.8	**14.3**	**<0.1**
		ZeRO-3	17.3	3.7	12.0	**13.7**	**0.3**
		ZeRO-3 + CPU Offloading	15.4	4.0	9.8	**11.7**	**0.3**
		Gradient Checkpointing	15.4	0.6	14.8	15.4	0.1
		All Enabled	11.8	6.2	5.4	**5.9**	**0.1**
ColossalChat	OPT	None	17.5	0.2	17.0	17.8	0.4
		ZeRO-3	16.5	0.5	15.6	**16.4**	**0.4**
		ZeRO-3 + CPU Offloading	13.1	0.4	12.3	13.1	0.2
		Gradient Checkpointng	14.8	0.7	12.1	**12.5**	**0.1**
	GPT-2	None	22.9	6.9	14.0	**15.0**	**0.1**
		ZeRO-3	22.1	7.6	13.2	**16.6**	**0.2**
		ZeRO-3 + CPU Offloading	15.0	2.6	10.3	**11.5**	**0.1**
		Gradient Checkpointing	22.9	6.9	14.0	**15.0**	**0.1**
		All Enabled	15.0	2.6	10.3	**11.5**	**0.1**

out that for GPT-2, the memory consumption reaches its peak during the inference phases, where gradient checkpointing has no effect.

> **Insights:** Not all memory management strategies can reduce memory consumption: ZeRO-3 may increase the fragmentation and memory consumption; ZeRO-2 and CPU offloading may increase the fragmentation, but they still effectively reduce memory consumption; gradient checkpointing only reduces the memory consumption when peaks occur during training phases, and ZeRO-1 consistently lowers memory consumption.

3.3 How Can We Effectively Reduce Memory Consumption in RLHF?

We find that the `empty_cache()` API exposed by the PyTorch allocator can help significantly alleviate memory fragmentation. Invocation of the API will release all cached memory blocks back to the GPU [24]. To reduce memory consumption, we propose to insert `empty_cache()` after each inference and training phase to release cached memory. Our results show that the proposed approach effectively reduces memory fragmentation, as shown in the bold part of Table 1. For these cases, it reduces the memory consumption by 25% on average. Additionally, the approach only increases the end-to-end time overhead by 2% on average.

We compared the memory consumption of invoking `empty_cache()` at different phases: (1) after each inference and training phase (2) only after each inference phase (3) only after the training phases. Based on our evaluation, invoking `empty_cache()` after inferences is almost as effective as invoking upon each inference and training phase, while invoking only after the training phases is not very effective. The observation echoes our previous insight in §3.1 that the inference phase introduces most of the memory fragmentation and has the largest impact on RLHF's memory consumption.

> **Insight:** Invoking `empty_cache()` after each inference phase can significantly reduce the memory fragmentation overhead and memory consumption, with only increasing 2% end-to-end time on average.

4 Conclusion

RLHF is an important stage for LLM alignment, but it often has high memory consumption. This paper provides the first study on RLHF memory usage. We identified the cause of high memory consumption, and investigated the effectiveness of different memory management strategies in the RLHF scenario. We also found a simple yet effective approach to reduce memory consumption by using the `empty_cache()` API, which can reduce 25% of the memory consumption with only 2% end-to-end time overhead on average.

References

[1] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[2] OpenAI. Introducing chatgpt. `https://openai.com/index/chatgpt/`, 2022.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] DeepSpeedExamples Contributors. Deepspeedexamples (deepspeed-chat) issue #482: A100 40 gb: Oom on step-3 for opt-6.7b. `https://github.com/microsoft/DeepSpeedExamples/issues/482`, 2023.

[5] Colossal-AI Contributors. Colossalai (colossalchat) issue #3832: Out of memory error encountered while running rlhf. `https://github.com/hpcaitech/ColossalAI/issues/3832`, 2023.

[6] DeepSpeedExamples Contributors. Deepspeedexamples (deepspeed-chat) issue #447: training 12b model seems to require more memory than expected. `https://github.com/microsoft/DeepSpeedExamples/issues/447`, 2023.

[7] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[8] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–14, 2021.

[9] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[10] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[11] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[12] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vattention: Dynamic memory management for serving llms without pagedattention, 2024.

[13] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, et al. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023.

[14] Yang You. Colossalchat: An open-source solution for cloning chatgpt with a complete rlhf pipeline. `https://medium.com/pytorch/colossalchat-an-open-source-solution-for-cloning-chatgpt-with-a-complete-rlhf-pipeline-5edf08fb538b`, 2023.

[15] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[16] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 766–775, 2023.

[17] Alexander Havrilla, Maksym Zhuravinskyi, Duy Phung, Aman Tiwari, Jonathan Tow, Stella Biderman, Quentin Anthony, and Louis Castricato. trlx: A framework for large scale reinforcement learning from human feedback. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 8578–8595, 2023.

[18] Youshao Xiao, Weichang Wu, Zhenglei Zhou, Fagui Mao, Shangchun Zhao, Lin Ju, Lei Liang, Xiaolu Zhang, and Jun Zhou. An adaptive placement and parallelism framework for accelerating rlhf training. *arXiv preprint arXiv:2312.11819*, 2023.

[19] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*, 2024.

[20] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[21] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[22] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[23] 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. *arXiv preprint arXiv:2106.09685*, 2021.

[24] PyTorch Documentation. Pytorch documentation - torch.cuda.empty_cache. `https://pytorch.org/docs/stable/generated/torch.cuda.empty_cache.html#to rch.cuda.empty_cache`, 2024.

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

A   Appendix: PyTorch Allocator and empty_cache()

The PyTorch memory allocator manages a pool of free memory blocks to minimize the overhead associated with frequent allocations and deallocations directly from the GPU. When memory is requested, the allocator first checks this pool for suitable blocks, invoking the costly `cudaMalloc()` operation only when necessary. Upon deallocation, memory blocks are not immediately returned to the GPU but are instead cached for future reuse. In RLHF training, different tasks often have different memory allocation patterns with varying object sizes, leaving smaller memory blocks in the pool that are difficult to reuse, leading to memory fragmentation. When memory management strategies are used, the difference in allocation sizes can become even more noticeable, making the fragmentation worse.

The PyTorch API function, `empty_cache()`, releases all unused cached blocks in the pool, reducing memory fragmentation. The function `empty_cache()` is not commonly used in traditional training because it has a limited impact on memory savings and significant time overhead. However, it is effective in RLHF training, likely because `empty_cache()` can release most of the cached blocks from the previous task, preventing fragmentation. Additionally, since the last task has already finished, these memory blocks are no longer being used by any stream, allowing them to be released without waiting.

B   Appendix: Other Implementation Details

We implemented a profiler that collects the sizes of reserved and allocated memory by calling the API of PyTorch's allocator. External memory fragmentation is computed at each `cudaMalloc()` invocation. It represents the difference between reserved and allocated memory when the allocator cannot satisfy the requested size due to non-contiguous freed objects.

For ColossalChat, we observed that the memory consumption of `generation()` was exceptionally high in the original implementation. We replaced its implementation of the function with HuggingFace's to decrease memory consumption.

C   Appendix: Additional Experiment Results

Table 2: Memory usage with and without ZeRO-3 on a node with 4 A100 GPUs. "Reserved" shows the peak of reserved memory size, "Frag." column shows the size of memory fragmentation, and "Allocated" column lists the peak size of allocated memory. We highlighted the strategies (with red color) that increase the memory fragmentation overhead, and the cases (with bold format) where `empty_cache()` is effective in reducing the fragmentation.

Framework	Model	Strategy	Original		Allocated	Using empty_cache()	
			Reserved	Frag.		Reserved	Frag.
ColossalChat	OPT-1.3b	None	46.4	2.4	43.5	**45.5**	**0.3**
		ZeRO-3	46.4	2.9	43.2	**45.0**	**0.3**
	OPT-6.7b	None	53.4	9.2	31.4	**44.3**	**0.1**
		ZeRO-3	55.3	20.6	25.6	**50.3**	**0.8**
	Llama-2-7b	None	56.2	8.8	39.2	**44.9**	**0.2**
		ZeRO-3	60.5	13.4	32.3	**54.5**	**1.7**

We also tested more examples on machines with A100s, as shown in Table 2. We found that the observations discussed in the main text also hold true across different platforms and models.

D   Appendix: Limitations

Our studies include just two open-sourced RLHF frameworks and two models. Our findings might vary when applied to other frameworks or models. Due to page constraints, we did not list the underlying reasons behind the varying effectiveness of each strategy. Additionally, we did not cover all existing memory management strategies, such as PagedAttention [25].