Summary of "Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection"

Alex Zhang (ahzhang), Jiyu Chen (jiyuchen), Kasey Feng (kaixfeng), Yuning Cong (yuningc)

Problem and Motivation

The **problem** addressed in this paper is the issues with state-of-the-art LLMs that they are often struggling with factual errors in their outputs despite their increased model and data scale. One common way to resolve this issue is by Retrieval-Augmented Generation (RAG) methods, which are considered to be able to enhance the accuracy and reliability of generative AI models, as they fetch facts from external sources, and augment the inputs to LLMs with these retrieved passages. However, there are also some problems with these RAG methods. First, sometimes they simply retrieve passages indiscriminately regardless of whether they are helpful or not, which could reduce the versatility of LLMs or introduce unnecessary or off-topic passages that actually harm the quality of the generated outputs. Second, although these RAG methods fetch relevant passages, the generated output is actually not guaranteed to be consistent with retrieved passages. This is because the RAG models are not explicitly trained to leverage and follow facts from provided passages, and they just treat the provided passages the same as other inputs. The **motivation** behind this paper is to solve the aforementioned problems with RAG methods to apply them properly and reduce the factual errors in the outputs generated by LLMs. This is an important problem to solve because reducing factual errors in LLM outputs can help improve LLM's reliability and eliminate misinformation and inaccurate knowledge generated by LLMs while maintaining high-quality and relevant outputs. As for real-world use cases, it can also help speed up the adoption of LLMs in professional domains, as well as applications that require high-accuracy generations.

Related Works

There are two major parts of related works: retrieval-augmented generation (RAG), and training and generating with critics. **Retrieval-augmented generation (RAG)** methods fetch facts from external sources and augment the inputs to LLMs with these retrieved passages. They are considered to be able to improve LLM's correctness in the generation when applied to knowledge-intensive tasks. Recent works applying RAG methods include: Luo et al. instruction-tune an LM with a fixed number of retrieved passages prepended to input; Izacard et al. pre-train a retriever and LM jointly, followed by few-shot fine-tuning on task datasets; Jiang et al. propose to adaptively retrieve passages for generation on top of a proprietary LLM; and Schick et al. train an LM to generate API calls for named entities. However, Mallen et al. pointed out that RAG methods sacrifice run-time efficiency. Shi et al. discussed that these methods make LLMs to be easily distracted by irrelevant context. Liu et al. and Gao et al. also highlighted that these methods suffer from a lack of proper attributions and citations. **Training and generating with critics** takes the idea from reinforcement learning. Instead of seeking feedback from humans like Schulman et al., Ouyang et al., and Wu et al. did, the solution proposed in this paper – Self-RAG seeks feedback from a critic model offline, with a far lower training cost compared to including human reviewers in the training process. Also, unlike previous work by Lu et al. and Korbak et al., which uses control tokens to guide LM generation, Self-RAG uses reflection tokens, which are well-defined to be able to reflect the need for retrieval as well as the generation quality.

Solution Overview

Inference overview:
The generator M will generate a sequence of token yt s as one segment at a time(for example the author uses a sentence as a segment in the paper) based on input x and previously generated sequence [y1, y2, …, yt-1]. During each round, M will first predict retrieval tokens based on (x, yt-1). If retrieval is not required it will directly generate yt like normal LMs. If retrieval is needed, the retriever R will retrieve all related passages D. For each passage d in D, generate a critique token IsREL that evaluates the retrieved passage's relevance based on (x, d), generate new segment yt based on (x, d, yt-1) and a critique token IsSUP to check if the generated segment is supported by d. Finally, a new critique token isUSE evaluates the overall utility of the response.

Training overview:
For critic model C, the training data was collected by prompting GPT-4 to add a reflection token to the original training data. The prompt includes an overall description of what critique tokens mean and a few shot instructions. The author collected 4k-20k supervised training data for each type and combined them to form training data for C. C is initialized as a normal LM and trained with the standard conditional language modeling objective, maximizing likelihood. **For generator model M**, the training data was augmented by pre-trained critic model C. The critic model will follow the inference algorithm to decide whether or not to insert retrieved passages and the following critique tokens. During the training process, it will mask out retrieved passages for loss calculation and expand vocabulary to include critique tokens.

Limitations

The computational overhead introduced by Critic: Self-RAG introduces an additional critique step, which evaluates the quality of generated responses in parallel, essentially implementing a "self-reflection" process. While this step is valuable for improving answer quality, it adds computational overhead, making the model slower and more resource-intensive. This could be problematic in real-time applications or on devices with limited computational resources when retrieval is frequently triggered.

Possible bottleneck of retriever: Self-RAG uses an off-shelf retriever which might not properly align with the generator M and critic C. As the effectiveness of the generator and critic improve, it will become the bottleneck for not retrieving the most relevant passages.

Future Research Directions

Jointly training LM and retriever: as mentioned in the limitations section, we could try to do instruction-tuning of retrieval systems or joint training of retrieval and LM components to avoid potential bottlenecks introduced by the retrieval system.

Optimizing the critique mechanism: The current model follows a fixed algorithm that could be improved. Rather than using static critique methods(isRel, isSup), developing adaptive critique models that change based on context or task requirements could be valuable. For example, introducing multi-modal critic signals—such as external knowledge sources or human feedback loops—could provide richer, more varied perspectives

Summary of Class Discussion

In general RAG, how does the retriever send texts to the generator? It's a context learning problem and it could append the text to the beginning of the prompt. **Is generating based on different sources in parallel the contribution of Self-RAG?** Yes, regular RAG just appends everything to the prompt.

How does Self-RAG retrieve top k sources? Self-RAG utilizes a retriever model to identify the most relevant passages. The process begins with the generator model determining whether more sources are needed. If additional sources are required, the retriever model comes into play to find and rank the top k relevant passages. An important point highlighted was that the retriever model operates independently from what Self-RAG does, meaning that Self-RAG can integrate any retriever model that proves effective.

Which model is generating these reflection tokens? The reflection tokens are generated by a combination of the generator model, retriever model, and critic model. **How to guarantee these tokens are produced?** While the generator model is responsible for producing the "Retrieve" tokens at the sentence level, it cannot entirely guarantee the generation of other critique tokens.

What if you need more than one source for generating the output? How to use multiple sources to generate one response? Self-RAG addresses this by taking another iteration to reassess the necessity of additional retrieval. Specifically, after the initial retrieval and generation, the model can be prompted to ask whether further retrieval is required.

At inference time, the generator model is responsible for executing all tasks, including generating output and reflection tokens. **Why not train two models, i.e. to introduce a separate critic model?** However, bundling both functionalities into one model may make the generator smarter and simplify the system by avoiding the back-and-forth communication between the two models. Nonetheless, using two separate models could still be a reasonable idea.

Is the Self-RAG model fine-tuned or trained from scratch? It is fine-tuned from a pre-trained Llama-2 model, which is then further trained using augmented data. This augmented data includes reflection tokens and examples generated with the assistance of GPT-4.

What are the time and resources needed for Self-RAG to generate its outputs? Self-RAG may actually be faster than traditional RAG models. This efficiency gain is partly because Self-RAG processes multiple sources in parallel, rather than sequentially.

Is there a limit on how many iterations Self-RAG would go if the “Retrieve” token is always “Yes”? Self-RAG does not continue indefinitely, working in a way similar to regular LLMs.

Understanding Self-RAG from a systems perspective: traditional RAG models aim to retrieve as many relevant sources as possible, while Self-RAG optimizes resource usage by wisely choosing which sources to retrieve based on the model's reflection tokens. There could also be a question of **handling trade-offs between finite resources and the accuracy of the outputs**, which could be an interesting future work direction.

Summary of "Fast Vector Query Processing for Large Datasets Beyond GPU Memory with Reordered Pipelining"

Alex Zhang (ahzhang), Jiyu Chen (jiyuchen), Kaixin Feng (kaixfeng), Yuning Cong (yuningc)

Problem and Motivation

The recent advancements in LLMs and the representation of unstructured data enable the emergence of many AI applications. Vector databases are designed to store these high-dimensional vectors efficiently to maintain long-term memory for LLMs. The ability of vector databases enables RAG, a technique that dynamically retrieves information from these databases to supplement what the LLM already knows. It allows build a vector database with documents, and later a query vector from a user to find the most similar vectors from a database. These similar vectors will be sent along with the original query vector to feed into an LLM to return the final result. With the growth of dataset scale and intensive computation of vector query processing, GPUs have become an optimization tool to reduce query overhead. While GPUs are fast, their memory is limited. One of the solutions is to split the dataset into partitions in host memory, sending each part to GPU memory to process vector queries. However, this causes an under-utilization of GPU resources due to the sequential computation. Another one is to utilize CUDA unified memory to handle GPU memory swapping automatically. While this solution keeps the computation parallel, massive GPU page faults may be involved because CUDA unified memory doesn't recognize vector queries.

For vector query processing, there are three primary GPU memory challenges in dealing with large datasets. When processing large batches of vector queries, limited GPU memory results in the re-transmission of the same data subsets for different queries within a batch. Secondly, existing systems suffer from an imbalance in loading when the query processing pipeline divides tasks into groups that don't fully occupy the SMs. Additionally, the differences in data size across clusters cause the issue that some tasks take longer than others. In conventional pipelining, lastly, data transmission and computation processes have limited overlap, with high overhead due to frequent API calls and synchronizations. This issue is compounded by the fact that each pipelining plan must be tailored to the specific query batch, which can only be determined at runtime.

Related Works

Vector Representation and Query Processing: DL models convert unstructured data into high-dimensional vectors, enabling vector databases to store this data as long-term memory for applications such as RAG. RAG utilizes these vector databases for efficient memory retrieval in LLMs, particularly in multimodal applications where datasets like TEXT1B contain billions of entries from diverse data types.
Nearest neighbor search techniques: to find similar data, vector queries rely on nearest neighbor search, particularly top-k nearest neighbors (KNN). However, KNN becomes impractical for billion-scale datasets due to high latency. Instead, the Approximate Nearest Neighbor (ANN) search is widely used, achieving high accuracy with reduced latency by only searching a subset of the dataset. Common representative ANN index methods, such as IVF and graph index, are used for high-dimensional similarity search. IVF is chosen because it's efficient for large-scale datasets and consumes less memory than graph-based indices.

Solution Overview

1. Cluster-based retrofitting: it reorganizes the query plan to reduce redundant transmission in two ways optimization. For intra-batch, it ensures each cluster is transmitted only once by processing all queries that need that cluster immediately after transmission. For inter-batch, it prioritizes clusters already present in GPU memory by moving them to the higher execution order.
2. Dynamic Kernel Padding with Cluster Balancing: cluster balancing divides clusters into equal-sized balanced clusters offline to prevent stragglers by ensuring each thread block processes the same amount of data. Moreover, dynamic kernel padding addresses spatial underutilization at runtime. Instead of trying to share GPU resources between different groups, it dynamically splits individual thread blocks into smaller ones. The kernel controller determines the optimal split factor based on the number of SMs and the query plan.
3. Query-Aware Reordering and Grouping:
    - Reordering determines the execution order of clusters via a greedy algorithm to maximize the overlap between data transmission and computation. It moves computation-heavy clusters forward in the execution order increasing the opportunity for overlap with transmission of subsequent clusters.
    - Grouping: use a dynamic programming algorithm to find the optimal way to group clusters for pipelining. This balances fine-grained pipelining benefits against system overhead costs.

Limitations

RUMMY delivers significant performance improvements for GPU-accelerated vector query processing, particularly in large-scale offline tasks and **large batch sizes**. However, its applicability to smaller batches might be limited, as the performance gains narrow significantly compared to existing solutions like IVF-GPU w/ CUDA unified memory (IVF-CUM). This may potentially offer only limited marginal benefits for real-time, interactive RAG scenarios, as small batch sizes may dominate in many highly interactive systems like chatbots or real-time search, where many RAG use cases may often be applied. Smaller batch sizes < 8 are not explored in this paper.

Future Research Directions

RUMMY is a transformative solution for large-scale, high-throughput tasks but faces diminishing returns in small batch, real-time, or edge-focused scenarios. This scalability toward smaller batches remains an area for further exploration, especially for specific RAG workloads.

The system's reliance on dynamic query plan retrofitting and reordering also introduces questions of fairness, particularly when batching queries of varying sizes or priorities. While dynamic configurations provide efficiency, they may favor certain workloads, creating resource contention or unpredictability in highly variable environments. In contrast, static configurations (i.e. in IVF-CUM, etc.), though less flexible, may offer simpler and more consistent performance for predictable workloads.

RUMMY pipeline scheduling enables RAG on beyond-GPU datasets as is, but perhaps can explore further directions in codesigning with higher level schedulers in order to better optimize resource utilization. By coordinating which batches are scheduled adjacent to each other and how queries are grouped, systems could reduce the frequency of cluster page-ins and page-outs, enhancing memory efficiency. Additionally, better workload coordination could address challenges like balancing latency-sensitive and throughput-driven tasks, ensuring equitable resource allocation while maintaining high overall performance.

Summary of Class Discussion

Q: How are queries converted into vector representations to conduct ANN vector queries?
A: some embedding/attention model

Q: combining clusters reduces transmission time (because only transmitting now, one, cluster for a group?
A: yes

Q: For all of the computations/transformations, they are drawing the arrows going from one circle to another but they actually have no meaning because if they were directed graphs and each of these circles is independent of each other you need all of them.
A: The arrows were to show the order that they're using if they do avoid retransmission. But it's not that the application has to do C2 first and it cannot do C1 before for a particular query.

Q: How is the size to partition the balanced cluster determined?
A: They may not say it in the paper, but they said they make the page size of the balanced clusters. So it's probably based on CUDA page size.

Q: The offline padding algorithm would have to be designed with the particular GPU configuration in order to be distributed among the multiple SMs, is that right?
A: The offline algorithm doesn't aim to fill all the SMs. It's aiming to split them into equal chunks, regardless of how many SMs are used.

Q: It seemed like the transmission and the computation time are on the same order of magnitude. Is it ever the case that there's computation going on but no transmission going on? And why would they not be proportional?
A: The diagram is to demonstrate the benefit of overlapping. The transmission may be smaller and do lots of transmissions first before starting the computation. Transmission is slower because it has to fetch from CPU memory compared with doing computation already in the device. Moreover, it's like the profiler works. We look at how big the cluster is and how much data is there to predict the amount of time to transmit.

Q: intra-batch fairness considerations for queries within a batch? or are the gains so significant on the batch level that it really does not matter
A: Yes probably does not matter.

Q: cluster ordering is NOT an application requirement -> else reordering is not possible

Q: (slide 45) - is transmission always the bottleneck? are there ever places in which computation is the bottleneck? (and thus computation is just executing alone, with no overlapped transmission/gaps)?
A: Yes.