LLM Fine-tuning For Rag Applications

Those involved with the development and use of Large Language Models (LLMs) know that Retrieval-Augmented Generation (RAG) is all the **rag**e :).

RAG (Lewis et al., 2021) was proposed with the objective of enhancing the output of LLMs for complicated tasks otherwise susceptible to generating hallucinations.

Why have RAG applications exploded?

The popularity of RAG can be attributed to its many advantages.

- **RAG eliminates the need to retrain LLMs for domain-specific applications.** Foundation models are trained in very general domain corpora, affecting their performance in domain-specific tasks. Additionally, they are trained offline and thus are limited to only data created before the model had finished training. With RAG, data from external sources could augment user prompts to the language model.
- **RAG exhibits high practicality and low barriers to entry.** As opposed to training a model from scratch, one can simply implement RAG applications on top of a pre-trained LLM. This effectively lowers the barriers to entry for developers and organizations.
- **RAG enables LLM integration with custom data and backend systems.** As a result, organizations are able to personalize their chat assistants with their proprietary data.
- **RAG allows for end-to-end control.**

RAG Paradigms

RAG's development can be categorized into three stages: Naive RAG, Advanced RAG, and Modular RAG.

Naive RAG:

Naive RAG consists of the following steps: indexing, retrieval, and generation.

Indexing: This stage involves encoding vector representations using an embedding model of chunked plain text data and storing these representations in a vector database.

Retrieval: For additional context, the prompt is augmented with the top K retrieved chunks scoring the highest similarity between the user's query and vectorized chunks of the indexed corpus. It is worth noting that the user's query was encoded into a vector representation by the same encoding model from the indexing phase.

Generation: An LLM generates a response for the prompt consisting of the posed query and selected documents. Existing conversational history can also be integrated to facilitate multi-turn dialogue.

There are many **drawbacks** observed with Naive RAG including selection of irrelevant chunks during retrieval as well as incoherent outputs that lack insightful information and contain biases, toxicity, hallucinations and redundancy. Additional challenges with Naive RAG implementation include limited memory capability and context window.

Advanced RAG:

The objective of advanced RAG is to overcome the limitations of Naive RAG. Strategies to improve upon Naive RAG include pre-retrieval and post-retrieval strategies.

Pre-retrieval strategies aim to improve the quality of the indexed text data and the original query. Strategies such as enhancing data granularity, optimizing index structures, adding metadata, alignment optimization, and mixed retrieval can be employed.

Post-retrieval strategies are for effective integration of the relevant context with the query. The main methods in the post-retrieval process include rerank chunks and context compressing.

Modular RAG:

Modular RAG focuses on improving modular components. This allows for improved adaptability through module substitution or reconfiguration.

- **Search module** for similarity searches
- **RAG Fusion** to expand user queries
- **Memory module** where RAG is iteratively employed, allowing the model to use its own output (aka self-memory) for higher quality generation
- **Predict module** employs strategies like GenRead where the LLM is first prompted to generate context and then reads the generated document to produce a final answer. This allows for a reduction in redundancy and noise.
- **Task Adapter module** tailors RAG to downstream tasks

Questions to Consider

When using RAG, it is worthwhile to consider the following questions:

1) What to retrieve?
2) When to retrieve?
3) How to use the retrieved information?

Evolution of RAG-related Research:

Initially, research efforts towards RAG focused on the inference stage, but as time went on, researchers have directed more attention to fine-tuning.

RAG and Fine-Tuning

Fine-tuning is likened to a student internalizing knowledge over time. In contrast, RAG is compared to providing a model with a customized textbook for information retrieval.

Advantages of Fine-tuning

After fine-tuning, the LLM is capable of responding without retrieval and therefore, there is lower latency. Customization (style, tone, increased domain focus) is also possible with fine-tuning.

Disadvantages of Fine-tuning

Fine-tuning requires retraining for knowledge and data updates since the data being stored is static. While training the model on data pertaining to a specific domain can reduce hallucinations, unfamiliar input can still elicit hallucinations. Additionally, the performance of fine-tuned models is reliant on the creation of high-quality datasets.

Advantages of RAG

RAG consistently outperforms fine-tuning for knowledge-intensive tasks. Furthermore, RAG is less susceptible to hallucinations, generates current information that can be traced back to specific data sources, and involves minimal data processing

.

Disadvantages of RAG

Increased latency, limited customization, and ethical considerations during data retrieval are all examples of concerns that arise with RAG implementation.

References:

Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks
Retrieval-Augmented Generation for Large Language Models: A Survey
https://docs.aws.amazon.com/sagemaker/latest/dg/jumpstart-foundation-models-customize-rag.html