【RAG技术】MemoRAG-迈向 RAG 2.0 的关键一步

原创 方方 方方的算法花园 2024年11月05日 08:49 北京

写在前面

2024年9月，北京智源人工智能研究院与中国人民大学高瓴人工智能学院联合推出基于长期记忆的下一代检索增强大模型框架MemoRAG，旨在推动RAG技术从仅能处理简单QA任务向应对复杂一般性任务拓展。相关链接如下：

- 论文：
  https://arxiv.org/pdf/2409.05591v1
- Github：
  https://github.com/qhjqhj00/MemoRAG

接下来，将对MemoRAG进行学习和解读。

01 研究背景

LLMs 的局限与 RAG 的兴起：尽管LLMs能力不断提升，但仍面临如知识缺乏导致的幻觉或过时内容、上下文窗口有限难以处理大量历史交互等问题。检索增强生成（RAG）成为解决这些挑战的有前途的范式，它通过检索工具引入外部数据库知识，使 LLMs 能基于知识生成更准确的回复。

现有 RAG 系统的局限：传统 RAG 系统通常要求明确的信息需求和结构化知识，其应用大多局限于简单的问答任务。然而，现实世界中的许多问题信息需求模糊且外部知识非结构化，如理解书中人物关系等，这些问题对现有 RAG 系统构成挑战。

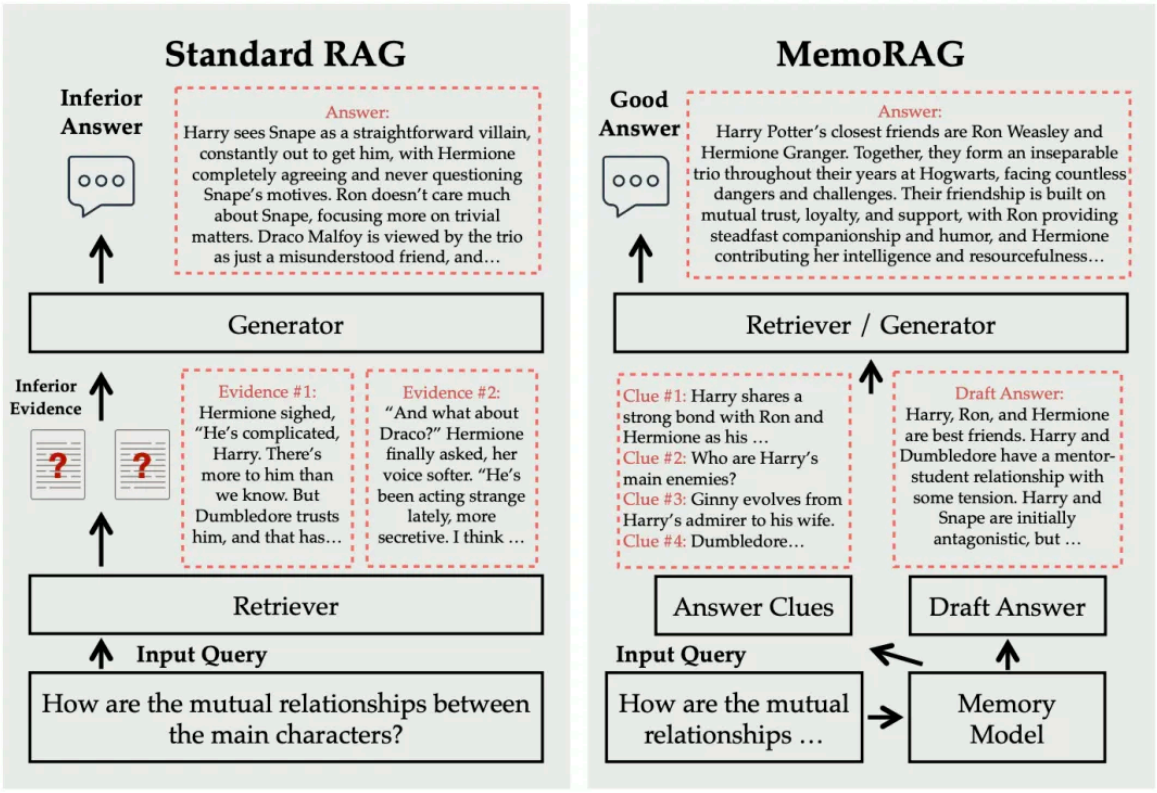
01 MemoRAG 框架介绍

MemoRAG 是一种基于记忆的检索增强生成范式，旨在解决现有 RAG 系统在处理模糊信息需求和非结构化知识方面的局限性，通过引入长期记忆模块，生成线索以指导检索，从而提高生成答案的质量，适用于复杂和常规的问答任务。

（1）总体框架：MemoRAG 采用双系统架构，由轻量级但长距离的 LLM 形成数据库的全局记忆，接收任务后生成草稿答案（线索），引导检索工具在数据库中定位有用信息，再由昂贵但表达能力强的 LLM 基于检索到的信息生成最终答案。

（2）记忆模块：提出灵活的模型架构促进记忆形成，将原始输入token逐步压缩为记忆token，同时保留关键语义信息。在训练方面，分预训练和监督微调两个阶段，预训练使用随机采样的长上下文学习从原始上下文形成记忆，监督微调使模型能基于形成的记忆生成任务特定线索。

（3）MemoRAG 框架流程：输入序列先转换为紧凑的记忆表示，全局记忆用于生成任务特定线索，这些线索帮助确定期望答案，然后基于线索用独立检索器在输入序列中定位证据文本，最后由生成模型基于检索到的证据文本生成最终答案，默认使用记忆模型的底层 LM 进行生成。



以上图为例，清晰展示了其如何处理输入查询并生成最终答案，体现了记忆模块、检索器和生成器之间的协同工作机制：

Step1: 当输入query（如“主要人物之间的相互关系如何？”）进入 MemoRAG 系统后，首先由记忆模块（Memory Model）进行处理。

Step2: 记忆模块基于其对整个数据库的全局记忆，生成一系列线索（Clues），这些线索以草稿答案的形式呈现，例如“哈利与罗恩和赫敏有着深厚的情谊，因为他……”等。

Step3: 这些线索作为检索器（Retriever）的输入，检索器根据线索在数据库（Database）中查找相关的证据（Evidence），如“赫敏叹了口气，‘他很复杂，哈利。他身上有很多我们不知道的东西。但是邓布利多信任他，而这有……’”等。

Step4: 最后，生成器（Generator）利用检索到的证据以及输入查询，生成最终的答案（Answer），如“哈利·波特最亲密的朋友是罗恩·韦斯莱和赫敏·格兰杰。他们在霍格沃茨的岁月里组成了一个不可分割的三人组，一起面对无数的危险和挑战。他们的友谊建立在相互信任、忠诚和支持之上，罗恩提供坚定的陪伴和幽默，赫敏贡献出她的智慧和足智多谋……”。

MemoRAG的“记忆-回忆-检索-生成”正是对人类记忆机制的仿生模拟。

02 MemoRAG 技术亮点

全局记忆：在单个上下文中处理多达100万个tokens，提供对海量数据集的全面理解。

可优化和灵活：轻松适应新任务，只需几个小时的额外训练即可实现优化性能。

上下文线索：从全局记忆中生成精确的线索，将原始输入桥接到答案，并从复杂数据中解锁隐藏的见解。

高效缓存：将上下文预填充速度提高多达30倍，支持缓存分块、索引和编码。

上下文重用：对长上下文进行一次编码，并支持重复使用，提高需要重复数据访问的任务的效率。

03 MemoRAG 的应用场景

1. 处理模糊信息需求：对于标准 RAG 难以处理的具有模糊信息需求的查询，MemoRAG 通过创建全局记忆推断隐含query的意图，生成如更具体的答案线索等阶段性答案，从而检索相

关内容，适用于如理解书籍主题等任务。

2. 应对分布式证据查询：针对需要分布式证据收集的查询，MemoRAG 利用全局记忆连接和整合数据库中多步骤的相关信息，通过生成阶段性答案指导检索相互关联的数据点，有效处理多跳查询，适用于分析多年财务数据确定峰值收入年份等任务。

3.信息聚合任务：在总结长文档等信息聚合任务中，MemoRAG 借助全局记忆捕获和合成整个数据集中的关键点，生成代表内容关键元素的中间阶段性答案，用于检索详细信息并汇总生成最终总结，适用于总结政府报告等任务。

4.个性化助手任务：个性化助手任务如根据用户偏好推荐歌曲，MemoRAG 通过分析用户对话历史形成的全局记忆，识别如音乐偏好、知识背景等关键线索，生成高度个性化的推荐，符合用户口味。

5.终身对话搜索任务：在对话搜索中，MemoRAG 利用全局记忆维护和利用完整的对话历史上下文，解释具有省略语义的查询，通过参考先前交互填补语义空白，准确回应依赖先前对话上下文的后续查询，如在讨论研究论文后的相关提问。

04 MemoRAG 实践

1 环境安装

要使用Memorizer和MemoRAG，需要安装Python以及所需的库。可以使用以下命令安装必要的依赖项：

```
# 安装依赖项
pip install torch==2.3.1
conda install -c pytorch -c nvidia faiss-gpu=1.8.0

# 从源代码安装
git clone https://github.com/qhjqhj00/MemoRAG.git
cd MemoRAG
pip install -e .

# 通过pip安装
pip install memorag

```

2 用法

MemoRAG的精简模式

建议从具有 24GiB 内存的 GPU 开始，但在大多数情况下，16GiB GPU 也可以在默认设置下处理pipelines。

```
from memorag import MemoRAGLite
```

```
pipe = MemoRAGLite()
context = open("examples/harry_potter.txt").read()
pipe.memorize(context, save_dir="harry_potter", print_stats=True)

query = "What's the book's main theme?"
print(pipe(query))
```

MemoRAG Lite使用简单，支持多达数百万个tokens的英文或中文上下文。虽然它可能适用于其他语言，但由于默认prompt是英文的，性能可能会下降。

MemoRAG的基本用法

MemoRAG易于使用，可以直接使用HuggingFace模型进行初始化。通过使用MemoRAG. Memize()方法，记忆模型在长输入上下文上构建全局记忆。经验上，通过默认参数设置，TommyChien/memorag-qwen2-7b-inst可以处理多达400K个tokens的上下文，而TommyChien/memorag-mistral-7b-inst可以管理多达128K个tokens的上下文。通过增加beacon_ratio参数，可以扩展模型处理更长上下文的能力。例如，TommyChien/memorag-qwen2-7b-inst可以处理多达一百万个tokens，beacon_ratio=16。

```
from memorag import MemoRAG

# 初始化 MemoRAG pipeline
pipe = MemoRAG(
    mem_model_name_or_path="TommyChien/memorag-mistral-7b-inst",
    ret_model_name_or_path="BAAI/bge-m3",
    gen_model_name_or_path="mistralai/Mistral-7B-Instruct-v0.2", # Opt
    cache_dir="path_to_model_cache",  # Optional: specify local model
    access_token="hugging_face_access_token",  # Optional: Hugging Fac
    beacon_ratio=4
)

context = open("examples/harry_potter.txt").read()
query = "How many times is the Chamber of Secrets opened in the book?"

# 记住上下文并保存到缓存中。
pipe.memorize(context, save_dir="cache/harry_potter/", print_stats=Tru
# 使用记忆的上下文生成响应。
res = pipe(context=context, query=query, task_type="memorag", max_new_
print(f"MemoRAG generated answer: \n{res}")
```

运行上述代码时，encoded key-value (KV) cache、Faiss index和chunked passages存储在指定的save_dir中。之后，如果再次使用相同的上下文，可以从磁盘

快速加载数据：

```
pipe.load("cache/harry_potter/", print_stats=True)
```

通常，加载缓存权重效率很高。例如，使用TommyChien/memorag-qwen2-7b-inst作为记忆模型，对200Ktokens上下文进行编码、分块和索引大约需要35秒，但从缓存文件加载时只需1.5秒。

使用长LLM作为记忆模型

由于其不断扩展的上下文窗口，LLM已经成为有效的记忆模型。MemoRAG现在支持利用这些长上下文LLM作为记忆模型，利用MInect来优化上下文预填充。已经测试了Meta-Llama-3.1-8B-Instruct和Llama3.1-8B-Chines-Chat作为记忆模型，两者都原生支持128K的上下文长度。目前正在探索其他合适的LLM并优化策略，以进一步增强内存机制和上下文长度。

```
from memorag import MemoRAG
model = MemoRAG(
    mem_model_name_or_path="shenzhi-wang/Llama3.1-8B-Chinese-Chat",
    # mem_model_name_or_path="meta-llama/Meta-Llama-3.1-8B-Instruct",
    ret_model_name_or_path="BAAI/bge-m3",
    # cache_dir="path_to_model_cache",  # to specify local model cache
    # access_token="hugging_face_access_token"  # to specify local model
    )
```

之后，你可以像往常一样使用MemoRAG的功能。

摘要任务

要执行摘要任务，请使用以下脚本：

```
res = pipe(context=context, task_type="summarize", max_new_tokens=512)
print(f"MemoRAG summary of the full book:\n {res}")
```

使用API作为生成器

```
from memorag import Agent, MemoRAG

# API configuration
```

```
api_dict = {
    "endpoint": "",
    "api_version": "2024-02-15-preview",
    "api_key": ""
}
model = "gpt-35-turbo-16k"
source = "azure"

# Initialize Agent with the API
agent = Agent(model, source, api_dict)
print(agent.generate("hi!"))  # Test the API

# Initialize MemoRAG pipeline with a customized generator model
pipe = MemoRAG(
    mem_model_name_or_path="TommyChien/memorag-qwen2-7b-inst",
    ret_model_name_or_path="BAAI/bge-m3",
    cache_dir="path_to_model_cache",  # Optional: specify local model
    customized_gen_model=agent,
)

# Load previously cached context
pipe.load("cache/harry_potter_qwen/", print_stats=True)

# Use the loaded context for question answering
query = "How are the mutual relationships between the main characters?
context = open("harry_potter.txt").read()

res = pipe(context=context, query=query, task_type="memorag", max_new_
print(f"MemoRAG with GPT-3.5 generated answer: \n{res}")
```

内置的Agent对象支持来自openai和deepseek的模型。以下是初始化这些模型的配置：

```
# Using deepseek models
model = ""
source = "deepseek"
api_dict = {
    "base_url": "",
    "api_key": ""
}

# Using openai models
model = ""
source = "openai"
api_dict = {
    "api_key": ""
```

```
}
```

记忆模型的用法

记忆模型可以独立用于存储、调用和与上下文交互。

```python
from memorag import Memory

# Initialize the Memory model
memo_model = Memory(
    "TommyChien/memorag-qwen2-7b-inst",
    cache_dir="path_to_model_cache",  # Optional: specify local model
    beacon_ratio=4  # Adjust beacon ratio for handling longer contexts
)

# Load and memorize the context
context = open("harry_potter.txt").read()
memo_model.memorize(context)

# Save the memorized context to disk
memo_model.save("cache/harry_potter/memory.bin")

# Query the model for answers
query = "How are the mutual relationships between the main characters?
res = memo_model.answer(query)
print("Using memory to answer the query:\n", res)

# Recall text clues for evidence retrieval
res = memo_model.recall(query)
print("Using memory to recall text clues to support evidence retrieval

# Rewrite the query into more specific surrogate queries
res = memo_model.rewrite(query)
print("Using memory to rewrite the input query into more specific surr
```

记忆增强检索的使用

除了独立的内存模型之外，MemoRAG还提供内存增强检索功能。这允许根据记忆中回忆的线索改进证据检索。

```python
from memorag import MemoRAG

# Initialize MemoRAG pipeline
pipe = MemoRAG(
    mem_model_name_or_path="TommyChien/memorag-qwen2-7b-inst",
    ret_model_name_or_path="BAAI/bge-m3",
    cache_dir="path_to_model_cache",  # Optional: specify local model
    access_token="hugging_face_access_token"  # Optional: Hugging Face
)

# Load and memorize the context
test_txt = open("harry_potter.txt").read()
pipe.memorize(test_txt, save_dir="cache/harry_potter/", print_stats=Tr

# Define the query
query = "How are the mutual relationships between the main characters?

# Recall clues from memory
clues = pipe.mem_model.recall(query).split("\n")
clues = [q for q in clues if len(q.split()) > 3]  # Filter out short o
print("Clues generated from memory:\n", clues)

# Retrieve relevant passages based on the recalled clues
retrieved_passages = pipe._retrieve(clues)
print("\n======\n".join(retrieved_passages[:3]))
```

3 其他

1. **MemoRAG 项目已开源或其他可使用的记忆模型：**
   a. 基于 Qwen2-7B-inst 的模型：
   https://huggingface.co/TommyChien/memorag-qwen2-7b-inst
   b. 基于 Mistral-7B-inst 的模型：
   https://huggingface.co/TommyChien/memorag-mistral-7b-inst
   c. https://huggingface.co/shenzhi-wang/Llama3.1-8B-Chinese-Chat
   d. https://huggingface.co/meta-llama/Meta-Llama-3.1-8B-Instruct
2. **数据集**
   a. UltraDomain基准测试：
   https://huggingface.co/datasets/TommyChien/UltraDomain
   b. 其他评测数据：https://huggingface.co/datasets/TommyChien/MemoRAG-data/
3. **代码案例**
   a. MemoRAG的使用示例：
   https://github.com/qhjqhj00/MemoRAG/blob/main/examples/example.i

pynb

b. 使用Long LLM作为记忆模型：
https://github.com/qhjqhj00/MemoRAG/blob/main/examples/longllm_as_memory.ipynb

c. 初始化MemoRAGLite：
https://github.com/qhjqhj00/MemoRAG/blob/tommy-dev-lite/examples/memorag_lite.ipynb

END