批量调用大语言模型 API 服务中的技巧

BITSE

2025-06-11

Outline

今天

~~分享某些我研究方向的论文~~

我想分享一些技术性内容

2024 年，大家的研究方向大多需要用到**大语言模型**：

- 使用**大语言模型**的重构检测、重构推荐
- 基于**大语言模型**的单元测试生成
- 代码审查意见的**大语言模型**评估
- **大语言模型**实现的代码生成、提交消息分类、测试输入变异
- …

使用大模型 API

做实验时，我们一定会需要调用 LLM 服务 API，而它们往往是收费的

于是你注册了某些账号，拿到了一串 API 秘钥，开始尝试写一个 Python 脚本

比如要做一个分类任务，准备了提示词模版和一批样本，期望模型输出类别标签

为了能获取模型预测的结果，你在提示词里说：“请你只输出回答，不需要额外解释”

使用大模型 API

初次接触 OpenAI API，你可能会写出这样的代码：

```python
import openai

def invokeGPT(sample_text):
    messages = [
        {"role": "user", "content": f"{prompt}\n{sample_text}"}
    ]
    response = openai.chat.completions.create(
        model='gpt-4o',
        messages=messages, ... )
    return response.choices[0].message.content
```

使用大模型 API

然后你写了一个循环，来对所有的样本调用这个函数：

```python
def main():
    for sample in my_samples:
        sample_text = 处理样本为字符串(sample)
        result = invokeGPT(sample_text)

        with open(f'./GPT第一次实验结果.txt', 'a') as f:
            f.write(str(result) + "\n")
```

看上去这个脚本可以正常工作，但是……

可能的问题

假设你的数据量非常大，10 万以上样本量，全部请求总量超过 10 million tokens，涉及很多 API 费用 …

这在实践中可能会有很多问题，比如：

1. **如何获得百分百可靠的输出结果？** 任务需要解析生成的文本，得到分类标签，但 LLM 不一定听话，纯文本解析显得不是很可靠。
2. **如何实现并发？** 无论是 OpenAI 等 API 服务商还是本地部署的 LLM 推理，都支持并发处理，且效率会高很多。这时我们就需要在客户端实现优雅的并发请求逻辑！
3. **如何缓存调用结果避免浪费 API 费用？** 大模型真的很贵啊，不想浪费钱。万一脚本跑到一半报错断了，或者想暂停，岂不是浪费好多钱？
4. **如何处理各类异常？** 从最常见的网络错误，到请求构建和结果解析，都有可能抛出异常。我们不希望批量执行被任何错误中断

针对以上问题，我今天想分享一些简单有效的解决方案

Outline

开发框架 LangChain

LangChain 是目前最知名的 LLM 开发框架，提供了大量组件

我们使用 LangChain 来简化一些功能的实现

```python
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
model = ChatOpenAI(model="gpt-4o")
prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant."),
    ("user", "Tell me about {input_text}."),
])
chain = prompt_template | model
chain.invoke({"input_text": "Large Language Model"})
```

API 服务商

大模型服务方面，OpenAI API 格式是最典型的，因为许多厂商都会与之兼容：

- OpenAI 官方的 GPT 系列
- DeepSeek 平台
- 其他 API 代理商或转发平台
- 通过 Ollama 本地部署的开源模型
- 通过 vLLM, SGLang 本地部署的开源模型

对于我们的脚本，LangChain 已经提供了 OpenAI API 的抽象

Outline

概念

让 LLM 输出 JSON 文本，然后再解析出 JSON 对象的字段，这非常可靠

API 服务平台一般都支持 JSON 输出，保证百分百可靠的结果解析

markdown

```
Your response should be in JSON format. Here is an example:

{
  "reasoning_steps": [
    "first ... ",
    "then ... "
  ],
  "category": <label>
}
```

代码实现

使用 LangChain 创建一个 LLM 抽象：

```python
llm = ChatOpenAI(
    model="...",      # 模型名称
    api_key="...",    # 秘钥
    base_url="...",   # 服务地址
    temperature=0.0,
    top_p=0.001,
).bind(response_format={"type": "json_object"})
```

这里我们需要填好 API 信息、模型名称；

同时指定 `temperature` 和 `top_p` 能最大程度上保证结果无随机性

代码实现

创建提示词模版对象

我们把提示词放在一个本地文件 `prompt.md` 方便编辑，注意文本中的 `{field}` 是可被替换的模版占位符

```python
from pathlib import Path
prompt = Path("./prompt.md").read_text()
prompt_template = ChatPromptTemplate.from_messages([
  ("system", "You are a helpful assistant."),
  ("user", prompt),
])
```

代码实现

然后我们需要编写处理输出的逻辑。ChatOpenAI 被调用后会返回一个 AIMessage 对象，最简单的处理方式如下：

```python
def parse_response(response: AIMessage) → dict:
    parsed = json.loads(response.content)
    return {"category": parsed["category"]}

chain = prompt_template | llm | RunnableLambda(parse_response)
```

这样我们就用 LangChain 描述了我们调用 LLM 的逻辑，从提示词模版到解析输出标签

代码中使用 `chain.invoke({"input_text": ...})` 来调用该逻辑

Outline

网络异常和 API 方错误

例如：

- 网络暂时断掉
- 达到速率限制需要等待重试

实际上 LangChain 提供了相关逻辑，它默认会在网络问题发生时进行重试，重试次数可以通过 `BaseChatOpenAI.max_retries` 参数控制。而速率限制处理行为可以通过指定 `rate_limiter` 来控制。

```python
llm = ChatOpenAI(
    ...
    max_retries=5,
    rate_limiter=<xxx>
)
```

处理输出解析部分的异常

首先要考虑的是输出解析部分的异常情况。当输出解析部分出现任何问题时，我们记录该问题，然后返回解析前的原始消息：

```python
def parse_response(response: AIMessage) → dict:
  try:
    parsed = json.loads(response.content)
    answer = str(parsed["category"])
    return {"category:": answer, "steps": steps}
  except Exception as e:
    logger.warning(f"LLM response parsing error: {e}")
    return {"category:": None, "raw": response.content}
```

这样，程序不会因为模型输出部分的问题而中断，后续也可以追溯这些意外情况。

模型缓存

难免会重新运行，当数据量大时，调用 LLM 的费用就显得浪费不起了。

我们可以在 API 调用处增加一层缓存，这样我们的批量任务可以随时中断调试，不用担心任何浪费。

实际上是把每一次请求内容和结果都做键值对缓存

- 每次发生请求，会在缓存中查找有没有请求参数+提示词完全相同的历史请求
- 如果没有，通过 HTTP 调用 API
- 如果有，直接返回结果

从而实现节省时间和金钱

模型缓存

所幸的是 LangChain 已经提供了相关功能：

```python
from langchain_community.cache import SQLiteCache
llm_response_cache_dir = Path(".langchain.db").as_posix()
llm_cache = SQLiteCache(database_path=llm_response_cache_dir)
llm = ChatOpenAI(
    ...
    cache=llm_cache,
)
```

缓存使用一个本地 sqlite 文件实现。

Outline

还是存在效率问题

假设已经定义好 chain 对象，我们的任务执行起来是这样：

```python
tasks: list[dict]
results = list()
for task in tasks:
    result = chain.invoke(task)
    results.append(result)
```

这种串行方式会挨个执行等待服务端生成，而 LLM 的生成速度是很慢的。要解决这个问题，OpenAI 提供了 Batches API，但有些情况下我们希望快速得到结果，或者部署服务不支持 Batch 调用形式，这时我们就非常需要自行实现并发请求。

并发请求的实现思路

思考一下，该任务的本质是：数量庞大的网络请求，每一个都极为耗时…

我们的脚本只负责发起 HTTP 请求，收集结果，绝大部分时间都在等网络

所以这个场景下，多进程不是理想的并发方式，而协程是个不错的选择

```python
await asyncio.gather(*[chain.ainvoke(task) for task in tasks])
```

实现时需要加以考虑：

1. 要避免一次性把整个数据集的每个任务都创建出协程对象，过度占用内存
2. 虽然 LangChain 已经帮我们处理了 rate limits，但并发数需要自己做控制
3. 怎么收集协程结果做进度条显示，毕竟比较耗时的任务没进度条还是不放心

代码实现

首先把 chain 逻辑写成异步函数，使用其异步方法，并添加异常处理：

```python
async def run_task(task: dict[str, str]) -> dict[str, str]:
    try:
        result = await chain.ainvoke(task)
        return {**task, **result}  # 把任务信息和模型结果一起返回
    except Exception as e:
        logger.exception(f"Error: {e}")
        return {**task, "error": str(e)}  # 返回错误信息
```

这个异步函数代表单个样本的处理，它的输入输出均为字典对象，非常灵活

并且这个异步函数保证不会抛异常，脚本不会因为有问题的样本而中断

代码实现

假设数据格式为 JSON lines，输入部分要分块进行，以避免过大的内存开销

```python
def read_file_in_chunks(filepath: str, chunk_size: int):
    current_batch = []
    with open(filepath, 'r', encoding="utf-8") as file:
        for line in file:
            current_batch.append(line.rstrip('\n'))
            # 每次迭代读取chunk_size行并返回一个列表
            if len(current_batch) ⩾ chunk_size:
                yield current_batch
                current_batch = []

    if current_batch:
        yield current_batch
```

```python
def read_file_in_chunks(filepath, chunk_size):
    ...
```

调用这个生成器进行迭代，将 JSON 字符串解析为字典对象

```python
for chunk in read_file_in_chunks(filepath, 1000):
    task_dicts = [json.loads(json_text) for json_text in chunk]
```

这时，对于这一组任务 `task_dicts: list[dict[str, str]]`，我们可以创建一批协程

然后使用 `asyncio.gather()`来并发执行协程：

```python
results = await asyncio.gather(*[run_task(d) for d in task_dicts])
```

代码实现

为了控制并发数，很显然这还不够

我们可以使用信号量机制来控制同一时刻能够并发执行的协程数目，为此我们需要在 run_task 外面套一层异步函数：

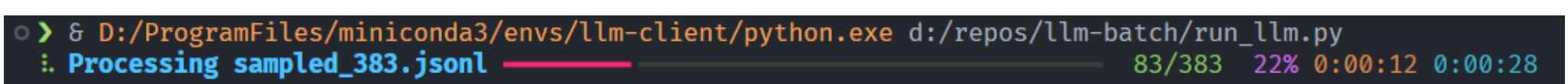
```python
semaphore = asyncio.Semaphore(max_concurrency)

async def do(data):
    async with semaphore:
        return await run_task(data)
```

在 asyncio.gather()调用时，所有 do()协程都被启动，但只有 max_concurrency 个能够进入 with 块，真正执行 run_task()异步函数，其他所有都会卡在 with 入口等待信号量释放

代码实现

进度条显示，使用 rich 库中的 rich.progress.Progress 对象，无需担心 race condition

```python
progress_bar = progress.Progress()
with progress_bar:
    task_id = progress_bar.add_task()
    async def do(data):
        async with semaphore:
            result = await run_task(data)
            progress_bar.advance(task_id)
            return result
```



Outline

小结

今天我们讨论了批量调用大模型做实验的技巧：

1. 我们用了 LangChain 来简化 OpenAI API 逻辑，包括请求重试和模型缓存
2. 我们编写了思维链提示词、JSON 结构化输出及其解析
3. 我们添加了异常处理和缓存，保证脚本不中断、钱不浪费
4. 我们使用协程实现了并发，并且使用信号量实现并发数控制

通过这些处理，相比于原始实现，我们可以做到：可靠的输出解析，并发执行万级以上样本，对各种异常情况比较健壮