1. 搭建自己的聊天模型 langchain+qwen+gradio

LangChain库的概述
LangChain是一个专为大型语言模型设计的Python库，它旨在简化模型与各类数据源的交互、增强模型的推理能力，并扩展模型的实际应用范围。这个库的初衷是让开发者能够更加轻松地搭建复杂的模型应用场景，而不需要处理繁琐的底层数据操作。

LangChain的核心功能
LangChain提供了多个核心模块，支持模型的动态调用、数据处理和知识管理，主要包括：

- **链式调用（Chains）**：通过将多个小步骤组合成一个工作流，使得用户可以创建端到端的问答系统或对话系统。
- **内存（Memory）**：支持会话记忆，使得模型在对话中可以记住上下文，从而实现连续对话。
- **工具（Tools）**：可以让模型与外部工具（如数据库、API等）交互，从而增强模型的功能和实用性。
- **知识库（Knowledge Base）**：支持创建知识库，使模型可以从外部知识源中获取信息，提高回答的准确性。

LangChain的应用场景
LangChain的设计目标之一是提升模型在实际应用中的适应性，因此它广泛应用于各类场景：

- **问答系统**：利用链式调用创建的问答系统可以在复杂问题上进行深度推理。
- **对话机器人**：利用内存模块增强对话连续性，实现更加人性化的交互。
- **知识驱动的任务**：通过知识库，模型能够更好地处理涉及大量背景知识的问题。

1.1. 核心推理代码

可参考：Langchain文档

以下代码为结合Langchain框架实现的LLM调用，具备上下文能力。

```
### 文件名为Qwen2.py，自定义文件名也可以，只要 from xxx import xxx 对应即可
# 一个简单的对话应用，能够保存对话历史
from langchain.llms.base import LLM
from transformers import AutoModelForCausalLM, AutoTokenizer
from typing import List, Optional
import torch

class Qwen2(LLM):

    # 模型参数
    max_new_tokens: int = 1920
    temperature: float  = 0.9
    top_p: float = 0.8
    tokenizer: object = None
    model: object = None
    history: List = []
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
    def __init__(self,max_new_tokens = 1920,
                     temperature = 0.9,
                     top_p = 0.8):
        super().__init__()
        self.max_new_tokens = max_new_tokens
        self.temperature = temperature
        self.top_p = top_p

    @property
    def _llm_type(self) -> str:
        return "Qwen2"

    # 载入模型，max_memory代表在载入模型阶段该显卡最多使用显存的大小，AWQ量化版本不支持模型载入到CPU内存中
    def load_model(self, model_name_or_path=None):
        self.tokenizer = AutoTokenizer.from_pretrained(
            model_name_or_path,
            trust_remote_code=True
        )
        self.model =AutoModelForCausalLM.from_pretrained(
            model_name_or_path,
            torch_dtype="auto",
            device_map="sequential",       #sequential/auto/balanced_low_0
            max_memory={0:"25GB",1: "25GB", 2: "5GB", 3: "5GB",4: "5GB", 5: "5GB", 6: "5GB"}
        )

        # 模型主要的chat功能实现
    def chat_stream(self, model, tokenizer, query:str, history:list):
        with torch.no_grad():
            # 历史整理
            messages = [
                {'role': 'system', 'content': '###角色\n你是一位临床心血管医生，你在心血管领域有非常深入的知识。你非常擅长使用通俗易懂的方式去准确回答心血管问题。 ###目标\n我希望你根据用户提出的心血管相关临床问题，提供准确、专业且易于理解的回答。\n用户将提供问题相关的文档，这些文档按与问题的相关性从高到低排列。你需要结合这些文档内容以及你本身的医学知识，正确回答题目并提供详细解释。\n你的所有回答都应有医学依据，确保回答的准确性和可靠性。'},
            ]
            # 将之前的history内容重新组合
            for item in history:
                if item['role'] == 'user':
                    if item.get('content'):
                        messages.append({'role': 'user', 'content': item['content']})
                if item['role'] == 'assistant':
                    if item.get('content'):
                        messages.append({'role': 'assistant', 'content': item['content']})
            # 最新的用户问题
            messages.append({'role': 'user', 'content': query})
            # 模型推理
            text = tokenizer.apply_chat_template(
                messages,
                tokenize=False,
                add_generation_prompt=True
            )
```

```
            model_inputs = tokenizer([text], return_tensors="pt").to(self.device)

            generated_ids = model.generate(
                **model_inputs,
                max_new_tokens=self.max_new_tokens,
                top_p=self.top_p,
                temperature=self.temperature
            )
            generated_ids = [
                output_ids[len(input_ids):] for input_ids, output_ids in
zip(model_inputs.input_ids, generated_ids)
            ]
                        # 模型根据messages的内容后的输出
            response = tokenizer.batch_decode(generated_ids,
skip_special_tokens=True)[0]
                        # 将模型输出组合到messages中
            messages.append({'role': 'assistant', 'content': response})

        return response ,messages

        # Langchain调用
    def _call(self, prompt: str ,stop: Optional[List[str]] = ["<|user|>"]):
        # 主要调用chat_stream实现
        response, self.history = self.chat_stream(self.model, self.tokenizer,
prompt, self.history)

        return response

    # 当使用RAG技术时会出现用户输入存在大量的参考资料，导致模型难以理解整体上下文内容。当LLM生
成回复后，使用该函数可将history的用户输入转换为不含参考资料的内容
    def query_only(self, query):
        if self.history[-2]['role'] == 'user':
            self.history[-2]['content'] = query

    # 返回模型history
    def get_history(self) -> List:
        return self.history

    # 删除模型所有history
    def delete_history(self):
        del self.history
        self.history = []
```

模型调用

```
from Qwen2 import Qwen2
model_name = "./qwen/Qwen2___5-7B-Instruct"
llm = Qwen2()
llm.load_model(model_name)

query = '你好'
response = llm.invoke(query)
print(response)
```

模型history

```
# 查看模型history
llm.history
# 删除模型所有history
llm.delete_history()
```

1.2. Gradio+Qwen

Gradio是大模型常用的前端界面库，下面代码将创建一个简单的前端界面以调用Qwen

Gradio的文档：https://www.gradio.app/guides/creating-a-chatbot-fast

pip install gradio==4.27.0

```
import gradio as gr
# 为上述“结合Langchain调用Qwen” 中的 class Qwen2
from Qwen2 import Qwen2
# 模型所在的地址
model_path = "./qwen/Qwen2___5-7B-Instruct"
# 载入模型
llm = Qwen2()
llm.load_model(model_path)

# 定义一个简单的predict
def predict(query,history):
    response = llm(query)
    # 将LLM中的history替换为只要query，避免上下文太长导致模型无法理解
    llm.query_only(query)

    return response
# 调用 Gradio 中的预制前端界面
gr.ChatInterface(predict).launch(share=True)
```

如果提示：

```
Please check your internet connection. This can happen if your antivirus software
blocks the download of this file. You can install manually by following these
steps:

1. Download this file: https://cdn-media.huggingface.co/frpc-gradio-
0.3/frpc_linux_amd64
2. Rename the downloaded file to: frpc_linux_amd64_v0.3
3. Move the file to this location:
/mnt/diskb5/zhangbo_private/anaconda/envs/llm_course/lib/python3.10/site-
packages/gradio
```

按照提示操作即可