基于ChatGLM3、LangChain、BigDL开发论文助手

1　选题背景和意义

对于本次云计算系统的大作业，我选择了做基于BigDL项目开发一个大语言模型应用。我选择做的是搭建一个论文助手。大语言模型有这很好的语言理解能力，用他们来处理文本是不错的选择。论文通常都比较晦涩难懂，在进行科研工作时，需要阅读大量文献，经常会浪费时间检索一些资料。如果有个程序能快速分析文本，将其内容梳理清楚，那将会大大减小科研人员的压力，节约出宝贵的时间投入到更有意义的研究中。另外，许多科研工作者并不都是英语专业，阅读英文论文也会增加他们的负担。这时，大语言模型就可以很好地发挥作用了。

2　知识和工具准备

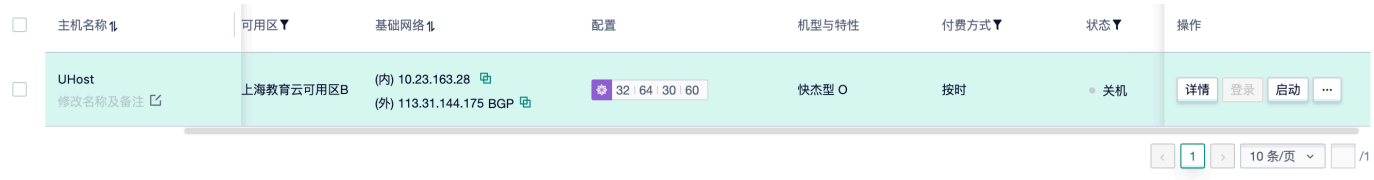
2.1　开发环境

2.1.1　平台

由于BigDL项目需要在Intel芯片上运行，而我的设备是ARM架构的MacBook。所以本次实验全部在Ucloud云主机上完成。

以下是我的配置

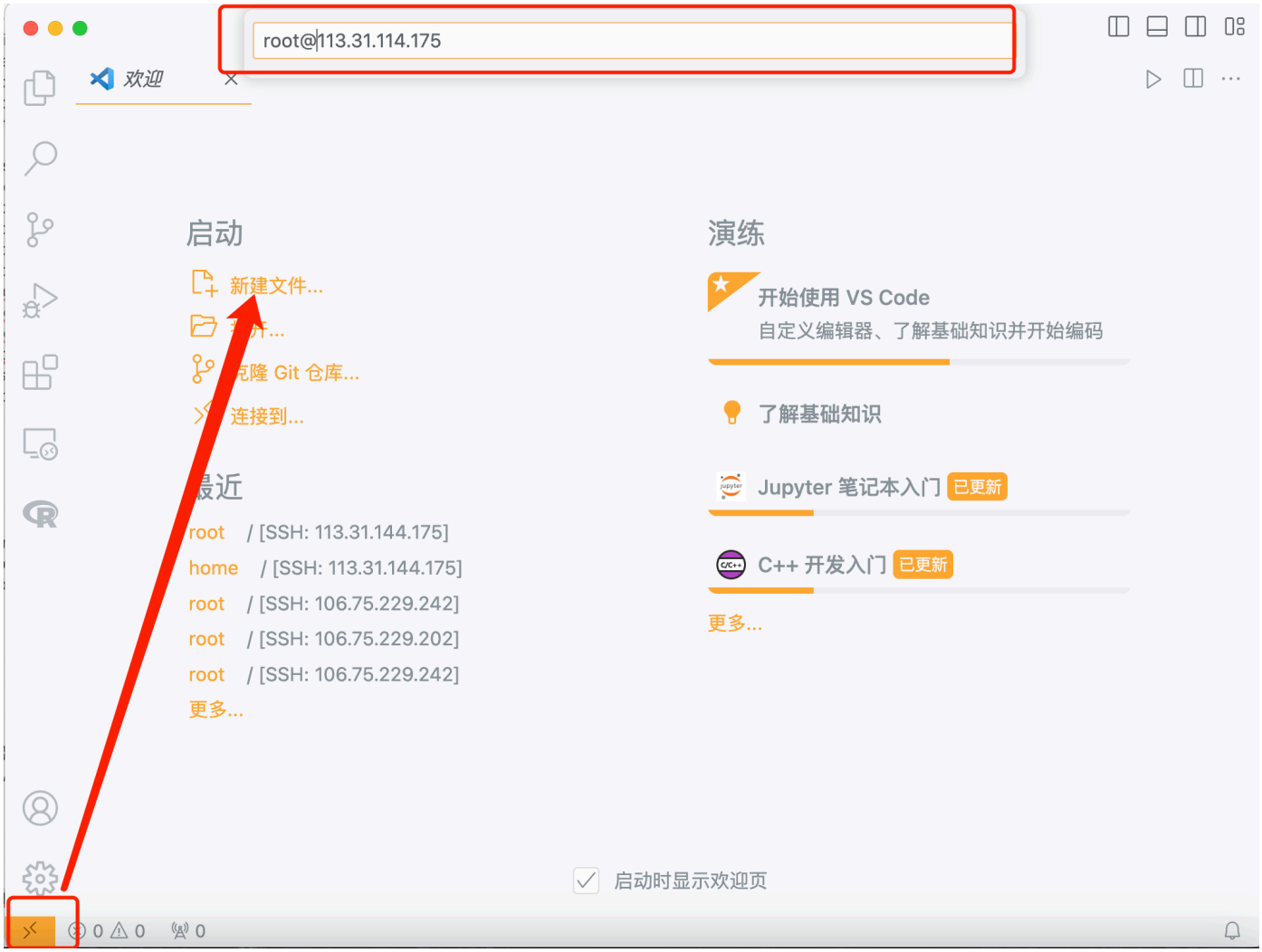
- CPU：Intel/CascadeLake，32核，64G内存
- 系统盘容量：30GB
- 数据盘容量：60GB



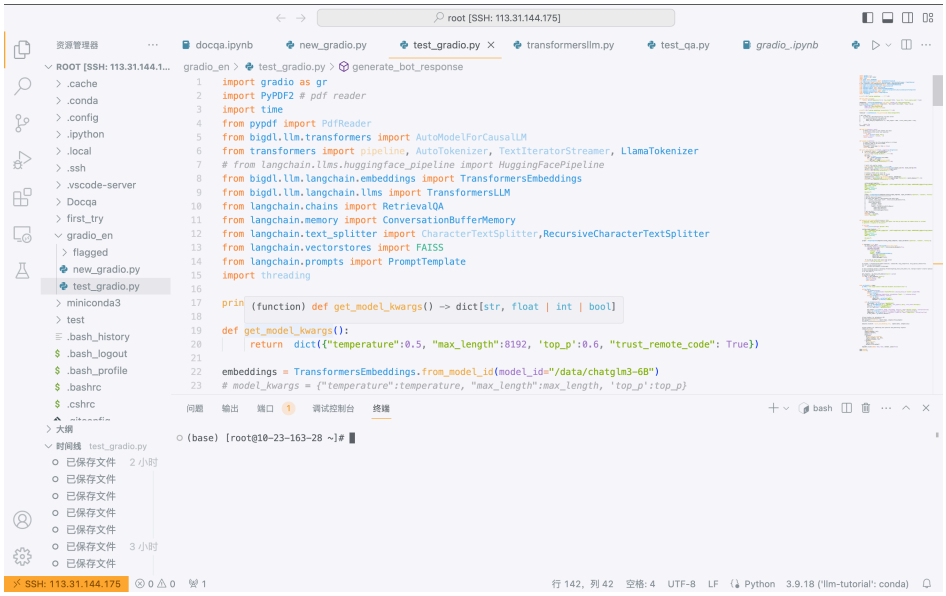
2.1.2　VS Code远程连接

工欲善其事，必先利其器。考虑到在命令行不适合开发环境，于是决定使用VS Code连接远程主机。

点击左下角“远程连接”按钮。然后在中间框内输入`root@113.31.114.175`



然后输入密码，即可连接。



但是，为了有更好的体验，我又利用VS Code在远程主机中安装了 `Pylance, Jupyter` 插件。这样就可以和在本地一样使用代码提示功能。也可以方便创建 `.ipynb` 文件

2.1.3 安装依赖

刚创建好的云主机里面啥也没有。所以安装了常用命令

```
yum install wget
yum vi
```

安装Conda环境

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash ./Miniconda3-latest-Linux-x86_64.sh
conda init
```

创建虚拟环境

```
conda create -n llm-tutorial python=3.9
conda activate llm-tutorial
```

至此，基本环境已经配置完成。现在云端开发和本地开发基本一致。

2.2 BigDL

2.2.1 安装BigDL

```
pip install --pre --upgrade bigdl-llm[all]
```

2.2.2 基本原理

BigDL是将模型的参数进行低精度保存，在保证模型性能前提下对模型进行加速。BigDL提供了方便的API，使得我们在开发时只需要修改一行代码，就可以完成优化。

BigDL提供了加载Hugging Face模型的接口，其使用方式和Hugging Face 的transformers相同。因为bigdl提供的API是继承了Hugging Face写的。下面展示一个优化示例

```
from bigdl.llm.transformers import AutoModelForCausalLM

model_path = 'openlm-research/open_llama_3b_v2'

model = AutoModelForCausalLM.from_pretrained(model_path,
                                             load_in_4bit=True)
```

从`bigdl.llm.transformers`中加载`AutoModelForCausalLM`

优化的步骤就是`load_in_4bit=True`，然后就可以用这个`model`做搭建应用了。

2.3 LangChain

LangChain是一款专门为大语言模型应用而诞生的一个框架。有十分便捷的API。而且BigDL也提供了LangChain的API。所以接下来都是综合利用BigDL、Hugging Face、LangChain。

2.3.1 安装

```
pip install langchain==0.1.1
```

LangChain主要有6个核心模块构成。我之后会在具体代码中做简单介绍。

2.4 Gradio

是一款机器学习框架，可以快速实现前后端。展示机器学习的结果、应用等。随着大语言模型的兴起，使用gradio可以快速搭建应用的前后端。

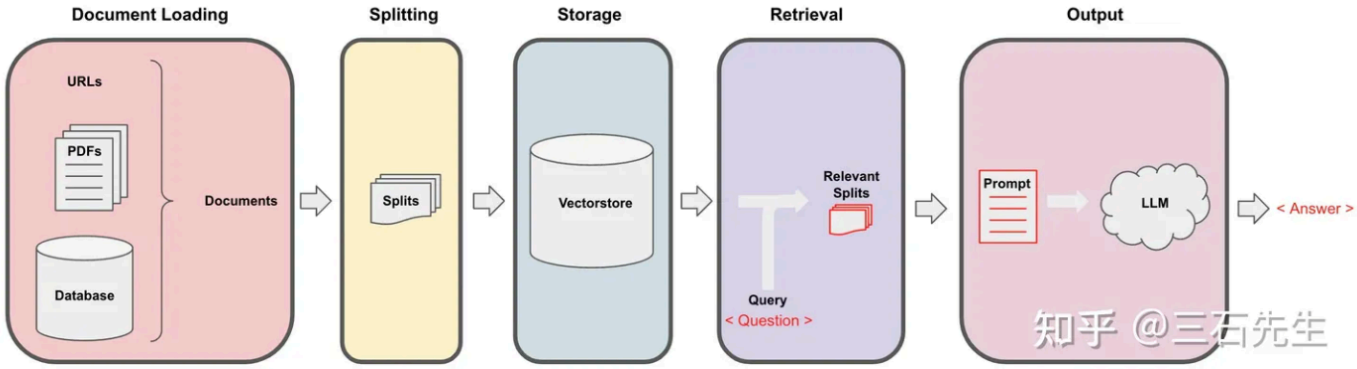
```
pip install gradio==3.39.0
```

3　构建应用

准备好了各种工具后，就可以开始开发应用了。我将构建应用的过程分为**模型加载，文档加载与嵌入、提示工程、构建RetrievalQA链，生成回答**这五个部分

应用流程大致如下，这里用了知乎三石先生的图片。

1. 首先要加载文档；
2. 然后进行分割和嵌入；
3. 将结果保存在向量数据库中；
4. 用户提出问题的时候（query）就会将用户的query和向量数据库中的内容进行比对；
5. 检索出符合条件的文本，最后语言模型根据prompt和文本生成回答。



3.1　模型加载

BigDL提供了与transformers相同功能的API，所以我们我可以根据API输入模型的名字直接从Hugging Face Hub上下载模型必要的文件。但是，应为大语言模型太过于庞大。我是用的ChatGLM3-6B有60亿参数，模型参数的大小接近15GB。直接从官网加载速度会很慢，而且经常出错，于是我选择先将文件下载到本地，然后上传到云主机。

```
scp config.json  pytorch_model-00003-of-00007.bin  pytorch_model.bin.index.json
configuration_chatglm.py  pytorch_model-00004-of-00007.bin  quantization.py
modeling_chatglm.py pytorch_model-00005-of-00007.bin  tokenization_chatglm.py
pytorch_model-00001-of-00007.bin  pytorch_model-00006-of-00007.bin
tokenizer_config.json pytorch_model-00002-of-00007.bin  pytorch_model-00007-of-
00007.bin  tokenizer.model root@113.31.144.175:/data/chatglm3-6B
```

```
(llm-tutorial) [root@10-23-163-28 ~]# cd /data
(llm-tutorial) [root@10-23-163-28 data]# ls
 chatglm3-6B  documents  vicuna-7b-v1.5
(llm-tutorial) [root@10-23-163-28 data]# cd chatglm3-6B/
(llm-tutorial) [root@10-23-163-28 chatglm3-6B]# ls
 config.json                    pytorch_model-00003-of-00007.bin  pytorch_model.bin.index.json
 configuration_chatglm.py       pytorch_model-00004-of-00007.bin  quantization.py
 modeling_chatglm.py            pytorch_model-00005-of-00007.bin  tokenization_chatglm.py
 pytorch_model-00001-of-00007.bin  pytorch_model-00006-of-00007.bin  tokenizer_config.json
 pytorch_model-00002-of-00007.bin  pytorch_model-00007-of-00007.bin  tokenizer.model
(llm-tutorial) [root@10-23-163-28 chatglm3-6B]#
```

然后调用API

```
from bigdl.llm.langchain.embeddings import TransformersEmbeddings
from bigdl.llm.langchain.llms import TransformersLLM

embeddings = TransformersEmbeddings.from_model_id(model_id="/data/chatglm3-6B")
llm = TransformersLLM.from_model_id(
        model_id="/data/chatglm3-6B",
        model_kwargs={"temperature":temperature,
                      "max_length":max_length,
                      'top_p':top_p})
```

这样，模型就加载好了。

- `temperature`表示的是温度系数，它的取值在`0~1`之间，越接近1表示模型的输出越具有多样性。
- `max_length`表示模型最大token的数量
- `top_p`表示这个参数用于控制模型在生成下一个词时考虑的概率分布的大小。具体来说，top_p 决定了在概率分布中保留的最高概率的累积概率的阈值。在生成每个词时，模型会按照概率从高到低对词汇表中的词进行排序，然后选择概率总和达到 top_p 阈值的所有词。这样做可以使得生成的文本更加多样性，因为不再仅仅选择最有可能的词。

3.2 文档加载与嵌入

3.2.1 加载

首先要安装pdf加载的包。

```
pip install pyPDF2
pip install pymupdf
```

```
pdf_text = ""
for file in files:
    pdf = PyPDF2.PdfReader(file.name)
    for page in pdf.pages:
        pdf_text += page.extract_text()
```

这一部分比较简单，函数会读取每个文件中的每一页，提取出文字。

3.2.2 分割

然后是文档分割。因为一篇论文中文本量是很大的，远远超过了模型最大token的输入子限制。所以我们需要将一篇论文分成很多个片段，然后对每个片段进行embedding。存放到向量数据库中。下面是相关的代码

```
from langchain.text_splitter import
CharacterTextSplitter,RecursiveCharacterTextSplitter

loader = PyPDFLoader("/data/documents/深度学习_582660_231113.pdf")
pages = loader.load()
pdf_splitter = RecursiveCharacterTextSplitter(chunk_size=1024, chunk_overlap=165)
pdf_text = pdf_splitter.split_documents(pages)
```

该文本分割器需要一个字符列表。它尝试根据第一个字符分割创建块，但如果任何块太大，则移动到下一个字符，依此类推。默认情况下，它尝试分割的字符是`[" ", "\n", " ", ""]`

- `length_function`：如何计算块的长度。默认情况下只计算字符数，但通常在此处传递令牌计数器。默认是python的`len`函数；
- `chunk_size`：块的最大大小（由长度函数测量）；
- `chunk_overlap`：不同文本块之间的最大重叠部分。保持文本块之间的一定连续性可能非常有用（例如，使用滑动窗口），因此一些重叠是很好的。

展示一下

这里是用的是数据学院兰韵诗老师的教学大纲。分割的结果如下（其中一块的结果）：

```
Document(page_content='《深度学习》教学大纲 \n课程代码 DATA0031132019 课程性质 专业选修 \n课程名称： 深度学习 \n英文名称 Deep Learning \n学时/学分 72/3 其中实验 /\n实践学时 36 \n开课单位 数据科学与工程学院 适用专业： 计算机科学 \n先修课程 无 \n大纲撰写人 兰韵诗 大纲审核\n人 钱卫宁 \n课程网址 无 授课语言 中文 \n \n一、课程说明 \n深度学习的概念源于人工神经网络的研究。深度学习通过组合低层特征形\n成更加抽象的高层表示属性类别或特征，以发现数据的分布式特征表示。深度\n学习是机器学习研究中的一个新的领域，其动机在于建立、模拟人脑进行分析\n学习的神经网络，它模仿人脑的机制来解释数据，例如图像，声音和文本。同\n机器学习方法一样，深度机器学习方法也有监督学习与无监督学习之分．不同\n的学习框架下建立的学习模型很是不同。 \n \n二、课程目标 \n通过为本科生开设《深度学习》这门课程，将达到如下目标： \n \n目标1：掌握深度学习的核心思想，以及主要技术，包括 最小二乘估计、\n随机梯度下降 等。（支撑毕业要求 6） \n目标2：学会用代码实现深度学习的经典算法，包括 前馈神经网络、卷积\n神经网络等 。（支撑毕业要求 9） \n目标3：学会用深度学习的技术解决实际问题，包括计算机视觉、自然语\n言处理等问题。 （支撑毕业要求 13） \n目标4:学会发散性思考，能够多方面分析模型的能力、优点和不足。能够\n尝试在已有的模型基础上进行修改。 （支撑毕业要求 10和14）', metadata={'source': '/data/documents/深度学习_582660_231113.pdf', 'page': 0}),
```

3.2.3 向量存储

分割晚文本之后，对每一段进行embedding，存入向量数据库中。这里我采用的数据库是`FAISS`

```
pip install faiss-cpu
```

同时加载embedding

```
from bigdl.llm.langchain.embeddings import TransformersEmbeddings
from langchain.vectorstores import FAISS

embeddings = TransformersEmbeddings.from_model_id(model_id="/data/chatglm3-6B/")
vectorstore_db = FAISS.from_documents(pdf_text, embeddings)
retriever = vectorstore_db.as_retriever(search_type="similarity", search_kwargs={"k": 6})
```

利用我们得到的多个pdf子文档与embedding，将文档转化为向量，然后将其存储在向量数据库中。这个过程比较花时间。

得到文档数据库之后构建**检索器**，这是之后模型搜搜索文档的工具。

- `search_type`：模型搜索的类别，这里是利用相似度进行搜索。还有很多其他的检索方式。
- `search_kwargs`：这个参数表示找出符合检索类别的文件的数量。例如在本例子中就是找到最相似六个文档

来生成回答。

3.3 提示工程（Prompt Engineering）

提示工程是大语言模型中的重要一环。广义上的提示指的是所有向大语言模型的输入。编写好的提示工程能够更好的发挥大语言模型的功能。同时我们要将提示语句和参考文本、问题清晰的分开。我来介绍一下这个项目中我写的提示工程。

```
custom_prompt_template = """
你是一个优秀的AI助手，现在给你提供了一些材料，请你根据这些材料回答用户提出的问题。如果你无法根据提供的信息回答问题，请直接回答不知道。不要尝试编造答案。
材料：{context}
历史记录：{history}
问题：{question}

有用的回答：
"""
from langchain.prompts import PromptTemplate
prompt = PromptTemplate(template=custom_prompt_template, input_variables=
["question", "context", "history"])
```

因为我们要搭建论文阅读助手，所以我们需要模型仅仅根据论文内容分析。不要收到其他信息的烦扰。同时将它可以参考的内容写清楚。（文本材料、历史记录、问题）。`{}`中的内容可以理解为变量，函数会将对应的内容传入其中。这样可以让模型更好地区分我们给的prompt和参考文本、问题。

3.4 构建RetrievalQA链

Chain是LangChain中的一个组件，用来完成一个任务。chain之间可以连接，完成多个复杂啊任务。`RetrievalQA`链就包含了**检索文本，将文本提交给大模型，大模型生成回答**这三个任务。具体用法如下

```
qa_chain_with_memory = RetrievalQA.from_chain_type(

llm=llm,
chain_type='stuff',
 retriever=vectorstore_db.as_retriever(),
return_source_documents = False,
chain_type_kwargs = {"verbose": False,
                     "prompt": prompt,
                     "memory": ConversationBufferMemory(

input_key="question",
                                                memory_key="history",

return_messages=False)},
                                    )
```

这段代码包括了很多元素。

- `llm`：3.1中加载的模型；
- `chain_type`：链式规则；
- `retriever`：3.2.3中提到的检索器，用来对比文本和查询语句，同时返回相关文本；
- `return_source_documents`:是否需要返回来源。（从文章的那一页找到的）
- `memory`：为模型添加聊天记录，对应于提示工程中的`{history}`。`ConversationBufferMemory`将我们之前和模型的聊天记录和文本一起传入模型中。这样，模型就可以利用聊天记录进行推理。如果没有memory，我们的每一次提问都是独立，模型不会利用之前的记录。

`qa_chain_with_memory`是实例化`RetrievalQA`。之后将用于提问与回答。

3.5 提问与回答

```
query = "这门课程的考核方式有哪些?"
bot_response = qa_chain_with_memory({"query": query})
```

```
query = "这门课程有哪些考核方式? 按照```考核方式1```:{考核内容} \n ```考核方式2```:{考核内容}的形式回答\n"
docs = docsearch.get_relevant_documents(query)
print("-"*20+"number of relevant documents"+"-"*20)
print(len(docs))
```

✓ 1.7s

```
--------------------number of relevant documents--------------------
4
```

模型的输出：

```
result = doc_chain.run(input_documents=docs, question=query)
```

✓ 54.5s

```
/root/miniconda3/envs/llm-tutorial/lib/python3.9/site-packages/transformers/generatio
  warnings.warn(
这门课程的考核方式如下：

1. 考勤和平时发言：学生需要参加课堂讨论并进行平时习题，以及完成期末大作业。
2. 平时习题：学生需要每天进行一定数量的习题，以巩固所学知识。
3. 期末大作业：学生需要完成一个关于深度学习的项目，以展示所学知识的应用。

这门课程的考核内容主要包括：

1. 理解机器学习的概念。
2. 熟悉 numpy和matplotlib 等基本的 python工具包。
3. 能够想到合理的深度 学习解决办法并通过代码把问题给解决掉。
```

真实结果对比：

考核方式 1：考勤和平时发言占 10%。教师在授课过程中将对出勤情况进行不定期查验，以此确定考勤分数。

考核方式 2：平时习题占 50%。教师根据该周的理论课内容，布置一道代码题型。考察学生对算法的掌握和应用能力，并锻炼编程能力。

考核方式 3:期末大作业占 40%。教师出一道实际场景中遇到的问题，让学生利用深度学习的方法自由解题。考察学生对深度学习方法的掌握和应用能力，并激发学生的思考和科研潜力。

可见模型的梳理还是很准确的。

论文的阅读在视频中展示！！！

3.5.1 流式输出

大语言模型一个很明显的问题是比较耗时间。如果我们让它生成全部回答会在一起返回，那么用户就要等待很长时间。所以应该是实时返回生成的内容，我们称之为流式输出。

BigDL的TransformersLLM中使用的就是流式输出。

4 前后端交互

我使用的是Grdio进行前后端交互。

4.1 前端

网页代码如下：

```
with gr.Blocks() as demo:
    gr.HTML("""<h1 align="center">🤗Welcome Documents Assistant🤗</h1>""")

    with gr.Row():
        with gr.Column(scale=4):
            chatbot = gr.Chatbot(label="ChatGLM-6B bot",value=
[],elem_id='chatbot',height=550)
            with gr.Column():
                txt = gr.Text(show_label=False, placeholder="Input...",
container=False)
            with gr.Column(min_width=32, scale=1):
```

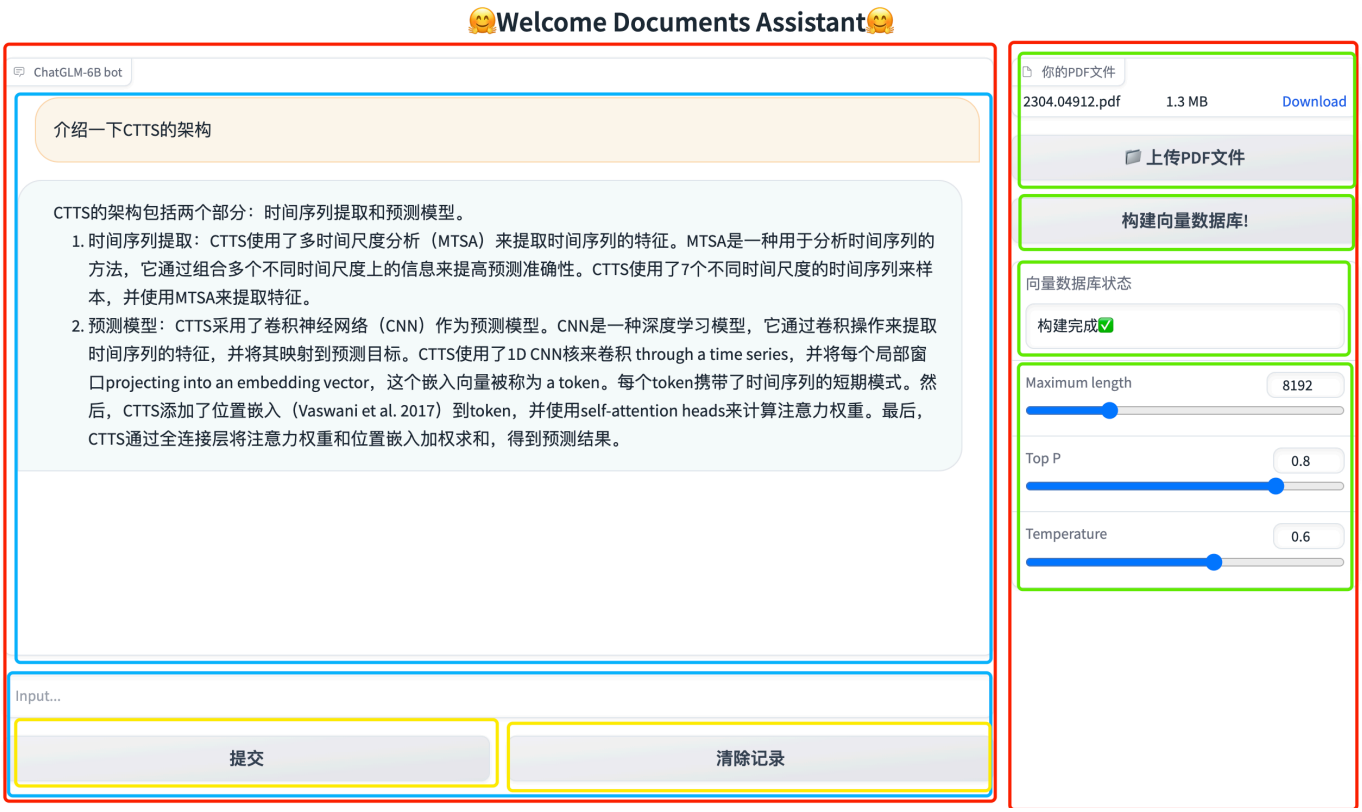
```
                with gr.Row():
                    # 提交
                    submit_btn = gr.Button("提交")

                    emptyBtn = gr.Button("清除记录")

            with gr.Column(scale=1):
                file_output = gr.File(label="你的PDF文件")
                btn = gr.UploadButton("📁 上传PDF文件", file_types=[".pdf"],
file_count="multiple")
                analysis = gr.Button('构建向量数据库!')
                xx = gr.Text(label= '向量数据库状态')

                max_length = gr.Slider(0, 32768, value=8192, step=1.0, label="Maximum
length", interactive=True)
                top_p = gr.Slider(0, 1, value=0.8, step=0.01, label="Top P",
interactive=True)
                temperature = gr.Slider(0.01, 1, value=0.6, step=0.01,
label="Temperature", interactive=True)
```

这个框架的特点是采用缩进、行、列的形式对一个页面进行分割。



- 例如在第一个`with_Row()`下有两个`with_Column()`这就是两个红色的框，表示他们是两个“列”，但是共享“行”。
- 在第一个`with_Column()`下有两个`with_Column()`，这就是两个蓝色的框，表示他们共享“列”，分割“行”

组件

- `submit_btn = gr.Button("提交")` 提交按钮
- `emptyBtn = gr.Button("清除记录")` 清除记录按钮
- `chatbot = gr.Chatbot(label="ChatGLM-6B bot",value[],elem_id='chatbot',height=550)` 聊天框
- `txt = gr.Text(show_label=False, placeholder="Input..."` 输入框
- `file_output = gr.File(label="你的PDF文件")` 上传文件夹
- `btn = gr.UploadButton` 文件上传按钮
- `analysis = gr.Button` 构建向量数据库按钮
- `xx = gr.Text('向量数据库状态')`

4.2 后端逻辑

4.2.1 模型与分词器

```
print("-"*20+"loading embeddings......"+"-"*20)

def get_model_kwargs():
     return  dict({"temperature":0.5, "max_length":8192, 'top_p':0.6,
"trust_remote_code": True})

embeddings = TransformersEmbeddings.from_model_id(model_id="/data/chatglm3-6B")
# model_kwargs = {"temperature":temperature, "max_length":max_length,
'top_p':top_p}
llm = TransformersLLM.from_model_id(
        model_id="/data/chatglm3-6B",
        model_kwargs=get_model_kwargs())

print("-"*20+"loading embeddings successfully!"+"-"*20)
```

首先加载模型和分词器。

4.2.2 向聊天框中添加消息

```
def add_text(history, text):
      # Adding user query to the chatbot and chain
      # use history with curent user question
      if not text:
          raise gr.Error('Enter text')
      history = history + [(text, '')]
      return history
```

4.2.3 上传文件

```
def upload_file(files):
      # Loads files when the file upload button is clicked
      # Displays them on the File window
      # print(type(file))
      file_path = [file.name for file in files]
      return file_path
```

4.2.4 分割文本、嵌入

```
def split_and_embedding_file(files, progress=gr.Progress()):
    #   progress(0, desc="开始构建向量数据库...")
    #   for i in progress.tqdm(range(100)):
        print("-"*20+"loading documents...."+"-"*20)
        pdf_text = ""
        for file in files:
            pdf = PyPDF2.PdfReader(file.name)
            for page in pdf.pages:
                pdf_text += page.extract_text()
        print("-"*20+"loading successfully!"+"-"*20)


        # split into smaller chunks
        print("-"*20+"spliting texts....."+"-"*20)
        text_splitter = RecursiveCharacterTextSplitter(chunk_size=600,
chunk_overlap=200)
        splits = text_splitter.create_documents([pdf_text])
        print("-"*20+"spliting successfully!"+"-"*20)

        # create a FAISS vector store db
        # embedd the chunks and store in the db
        print("-"*20+"embedding...."+"-"*20)
        vectorstore_db = FAISS.from_documents(splits, embeddings)
        retriever_ = vectorstore_db.as_retriever(search_type="similarity",
search_kwargs={"k": 6})
        print("-"*20+"embedding successfully!"+"-"*20)

        return "构建完成✅"
```

4.2.5 生成回答

```
def generate_bot_response(history,query, btn):
    """Function takes the query, history and inputs from the qa chain when the
submit button is clicked
    to generate a response to the query"""
```

```
    if not btn:
        raise gr.Error(message='Upload a PDF')

    custom_prompt_template = """
    你是一个优秀的AI助手，现在给你提供了一些材料，请你根据这些材料回答用户提出的问题。如果你无法根据提供的信息回答问题，请直接回答不知道。\n
        不要尝试编造答案。
        材料：{context}
        历史记录：{history}
        问题：{question}

        有用的回答：
        """
    prompt = PromptTemplate(template=custom_prompt_template, input_variables=["question", "context", "history"])


    if retriever is not None:
        qa_chain_with_memory = RetrievalQA.from_chain_type(
            llm=llm, chain_type='stuff', return_source_documents=True,
            retriever=retriever,
            chain_type_kwargs={
                "verbose": False,
                "prompt": prompt,
                "memory": ConversationBufferMemory(
                    input_key="question",
                    memory_key="history",
                    return_messages=True)})


    bot_response = qa_chain_with_memory({"query": query})
    #

    # 向前端发送消息
    for char in bot_response['result']:
          history[-1][-1] += char
          time.sleep(0.05)
          yield history,''
```

4.3 前后端交互

4.3.1 监听上传文件

```
btn.upload(fn=upload_file, inputs=[btn], outputs=[file_output])
```

点击按钮上传文件。执行`upload_file`函数，输入是`btn`的值（也就是文件），输出到`file_output`,一个上传文件夹（图中右上角）。

4.3.2 构建数据库

```
analysis.click(fn = split_and_embedding_file,  inputs=[btn], outputs=[xx])
```

点击将上传的文件进行数据库构。执行`split_and_embedding_file`。输入仍然是`btn`的值，输出到`[xx]`的一个提示("构建完成✅")，告诉用户构建完成。

4.3.3 清除监听按钮

```
emptyBtn.click(lambda: None, None, chatbot, queue=False)
```

点击将聊天记录清除，同时将模型内保存的和用户的记录删除。

4.3.4 问题提交监听

```
submit_btn.click(
        fn= add_text,
        inputs=[chatbot, txt],
        outputs=[chatbot],
        queue=False
        ).success(
          fn=generate_bot_response,
          inputs=[chatbot, txt, btn],
          outputs=[chatbot, txt]
        ).success(
          fn=upload_file,
          inputs=[btn],
          outputs=[file_output]
        )
```

- 点击提交按钮先将提问装入对话框
- 然后执行回答生成，将回复消息加入到对话框

5 BigDL的一个小问题

在刚开始进行实验的时候，我发现每次没进行几次对话，就会收到提示，4096个token已经用完了。我感到非常奇怪，为什么没说几句话token就超了呢？实现前端后，我发现模型每次都会把我的提示工程输出，这里要注意，提示工程中还嵌入了论文内容，所以输出非常长，占用了很多个token。

```
query = "summarize the paper, please."
qa_chain_with_memory({"query": query})
```

Python

```
Token indices sequence length is longer than the specified maximum sequence length for this model (4873 > 4096). Running this sequence through the model w
/root/miniconda3/envs/llm-tutorial/lib/python3.9/site-packages/transformers/generation/utils.py:1369: UserWarning: Using `max_length`'s default (4096) to
  warnings.warn(
Input length of input_ids is 4873, but `max_length` is set to 4096. This can lead to unexpected behavior. You should consider increasing `max_new_tokens`.
```

模型把提示工程内容全部输出了。

```
{'query': 'what is the main theory of the paper?',
 'result': "You are an assistant for question-answering tasks. Use the following pieces of retrieved context and
answer the question at the end.\nIf you don't know the answer just say you d not know an do not try to make up th
e answer nor try to use outside souces to answer. Keep the answer as concise as possible.\nContext= and long-term
dependencies within a time series, and forecast\nif the price would go up, down or remain the same (flat) in\nthe
future. In our experiments, we demonstrated the success\nof the proposed method in comparison to commonly adopted
\nstatistical and deep learning methods on forecasting intraday\nstock price change of S&P 500 constituents.\nInt
roduction\nTime series forecasting is challenging, especially in the fi-\nnancial industry (Pedersen 2019). It inv
olves statistically\nunderstanding complex linear and non-linear interactions\nwithin historical data to predict
the future. In the finan-\ncial industry, common applications for forecasting include\npredicting buy/sell or posi
tive/negative price changes for\ncompany stocks traded on the market. Traditional statis-\ntical approaches commo
nly adapt linear regression, expo-\nnential smoothing (Holt 2004; Winters 1960; Gardner Jr\nand McKenzie 1985) an
d autoregression models (Makri-\ndakis, Spiliotis, and Assimakopoulos 2020). With the ad-\n\ntistical tools, such
as, exponential smoothing (ETS) (Holt\n2004; Winters 1960; Gardner Jr and McKenzie 1985)\nand autoregressive inte
grated moving average (ARIMA)\n(Makridakis, Spiliotis, and Assimakopoulos 2020), on nu-\nmerical time series data
for making one-step-ahead predic-\ntions. These predictions are then recursively fed into the\nfuture inputs to o
btain multi-step forecasts. Multi-horizon\nforecasting methods such as (Taieb, Sorjamaa, and Bon-\ntempi 2010; Ma
rcellino, Stock, and Watson 2006) directly\ngenerate simultaneous predictions for multiple pre-defined\nfuture tim
e steps.\nMachine learning and deep learning based approaches\nMachine learning (ML) approaches have shown to imp
rove\nperformance by addressing high-dimensional and non-linear\nfeature interactions in a model-free way. These
methods in-\nclude tree-based algorithms, ensemble methods, neural net-\nwork, autoregression and recurrent neura
l networks (Hastie,\nTibshirani, and Friedman 2001). More recent works have ap-\n\nFigure 1: Overview of the prop
osed approach. Quick peak of the transformer encoder architecture on the right (Dosovitskiy\net al. 2020)\nprove
the performance of NLP applications. The commonly\nused approach is to pre-train on a large dataset and then fine-
\ntune on a smaller task-specific dataset (Devlin et al. 2018).\nTransformers leverage from multi-headed self-atte
ntion and\nreplace the recurrent layers most commonly used in encoder-\ndecoder architectures. In contrast to RNN
```

于是我开始阅读源码。在阅读 `from bigdl.llm.langchain.llms import TransformersLLM`

这段源码时发现了问题：

```python
if self.streaming:
    from transformers import TextStreamer
    input_ids = self.tokenizer([prompt], return_tensors="pt")
    input_length = input_ids['input_ids'].shape[1]
    streamer = TextStreamer(self.tokenizer, skip_prompt=True, skip_special_tokens=True)
    # streamer = self.streamer
    if stop is not None:
        from transformers.generation.stopping_criteria import StoppingCriteriaList
        from transformers.tools.agents import StopSequenceCriteria
        # stop generation when stop words are encountered
        # TODO: stop generation when the following one is stop word
        stopping_criteria = StoppingCriteriaList([StopSequenceCriteria(stop,
                                                  self.tokenizer)])
    else:
        stopping_criteria = None

    # for token in streamer:
    #     print(token)
    output = self.model.generate(**input_ids, streamer=streamer,
                                 stopping_criteria=stopping_criteria, **kwargs)
    text = self.tokenizer.batch_decode(output[:, input_length:], skip_special_tokens=True)[0]
```

标红的地方是我添加（第一个框）和修改过的部分

原来的代码是

```
text = self.tokenizer(output[0], skip_special_tokens=True)
```

模型一开始将promt作为输入，这没有问题。但是输出的时候应跳过prompt的对应的部分，因为我们不需要这部分。跳过的方法很简单，就是获取`input_ids['input_ids']`的长度,这个长度就是prompt对应的embedding的长度，我们在解码前跳过这段长度输出才是我们要的答案。而模型将output[0]解码，根据Hugging 函数的规则，`output[0]`包含了所有的信息，及输入和输出都有。所以会出现输出超长的情况。同时要注意将`decode`改为`batch_decode`，`decode`一次只能解码`output`中的一个元素，如果使用了`output[:, input_length:]`将会有多个元素，所以不能使用`decode`。

做了上述修改后，模型输出恢复正常！！！

后来我思考，为什么开发人员会这么写？我在BigDL的github中找到了答案。仓库中给的demo就是**续写故事**，所以他们会这么写。但是这么写必然存在问题，因为不是所有任务都要按照prompt开始生成。而且，如果真的要以某句话开头生成回答，我们完全可以在prompt中写道：“`"""请以{begin}开头，续写故事。"""`”。

6 总结

本次大作业学习了大模型的相关概念，了解了LangChain的许多使用方法。也更加熟悉了Hugging Face transformers API的用法。感受了NLP的魅力！为后续的学习打下基础。

演示视频：https://www.bilibili.com/video/BV1rC4y1r7z5/?vd_source=2fa9348d0245244f19ce3da71f309a92

github代码仓库https://github.com/NaOH678/DocQA-based-on-LangChain-and-BigDL/tree/main，只需要关注

gradio_en中的代码即可，其他的都是草稿。