2024-2025 学年冬季学期

《智能计算系统》(08696037)

实验报告

姓　　名	汪江豪
学　　号	22121630
实验名称	基于 Llama 2 实现聊天机器人
日　　期	2025 年 2 月 8 日

实验 (70 分)	目标 1	目标 2	目标 3	得分
报告 (30 分)	代码 (10 分)	结果 (10 分)	格式 (10 分)	得分
批阅人		**批阅日期**		**总得分**

上海大学 计算机工程与科学学院

目录

一　选修实验 8-2 基于 Llama 2 实现聊天机器人

1　实验目的

- 熟悉 Llama 2 大语言模型的算法原理，掌握在 DLP 平台上移植优化聊天机器人的方法和流程，具体包括：
- 本实验旨在向学生介绍基于大型语言模型 Llama 2 构建聊天机器人的基本原理和操作步骤，深入理解其在对话生成中的应用；
- 学生将深入了解关键概念，包括模型加载和适配、对话模板的应用、文本生成的基本流程、模型推理的关键步骤，以及使用命令行进行聊天的具体过程；
- 学生将了解如何在 DLP 平台上部署模型，实现人机快速聊天应用，为将来在实际场景中应用模型提供实用经验。

2　实验环境

- 硬件平台：DLP 云平台环境。
- 软件环境：编程框架 Pytorch1.13.1、CNNL 高性能 AI 运算库，CNRT 运行时库，以及 python 环境及相关的扩展库。

3　评分标准

- 60 分标准：能够正确实现模型适配模块、对话模板模块，其中无论是否提供对话模板都能正确地实现程序。
- 80 分标准：在 60 分标准基础上，能够正确实现文本生成模块。
- 100 分标准：在 80 分标准基础上，能够采用 Simple 和 Rich 两种聊天模式与机器人进行交互，实现 FastChat 模型推断模块，可以完成聊天应用。

4　实验内容及步骤

4.1　补全 fastchat/utils.py

该文件主要包含了一些辅助工具函数和类，用于日志记录，重定向标准输出和错误输出，获取 GPU 信息等。

作业主要任务是补全 get_gpu_memory() 函数，代码补全如下：

代码 1: utils.py

```
def get_gpu_memory(max_gpus=None):
    """Get available memory for each GPU."""
    #TODO: 存储每个GPU的可用内存信息
    gpu_memory = []
    #TODO: 获取MLU设备的数量，如果未指定最大GPU数（max_gpus为None），则使用所有
        可用设备;否则，使用max_gpus和实际设备数量中的较小值
    num_gpus = (
        min(max_gpus, torch.mlu.device_count())
        if max_gpus is not None
        else torch.mlu.device_count()
```

```
    )

    for gpu_id in range(num_gpus):
        #TODO: 将当前MLU设备设置为gpu_id
        torch.mlu.set_device(f"mlu:{gpu_id}")
        #TODO: 获取当前MLU设备
        device = torch.mlu.current_device()
        #TODO: 获取当前MLU设备的属性
        gpu_properties = torch.mlu.get_device_properties(device)
        #TODO: 获取总内存，单位转换为GB
        total_memory = gpu_properties.total_memory / (1024 ** 3) # GB
        #TODO: 获取已分配内存，单位转换为GB
        allocated_memory = torch.mlu.memory_allocated() / (1024 ** 3) # GB
        #TODO: 计算可用内存，单位转换为GB
        available_memory = total_memory - allocated_memory
        #TODO:将可用内存信息添加到列表中
        gpu_memory.append(available_memory)
    return gpu_memory
```

4.2 补全 fastchat/model/model_adapter.py

该文件主要用于模型适配器的注册和管理，它定义了一个基础模型适配器类 BaseAdapter 及其多个子类。每个子类正对不同的模型类型实现了特定的加载和匹配逻辑。文件还包含了用于注册模型适配器的函数、加载模型的函数以及获取对话模板的函数。通过这些适配器，可以根据模型路径动态选择和加载合适的模型和分词器。

作业主要任务是补全 load_model() 函数和 get_conversation_template() 函数。

由于篇幅限制，这里只展示部分补全的代码

代码 2: load_model() 函数

```
def load_model(
    model_path: str,
    device: str,
    num_gpus: int,
    max_gpu_memory: Optional[str] = None,
    load_8bit: bool = False,
    cpu_offloading: bool = False,
    debug: bool = False,
):
    """Load a model from Hugging Face."""

    # TODO: 处理设备映射，调用函数，检查并更新CPU offloading的配置
    cpu_offloading = raise_warning_for_incompatible_cpu_offloading_configuration
        (
        device, load_8bit, cpu_offloading
    )
    # TODO: 如果设备是CPU
```

```
    if device == "cpu":
        kwargs = {"torch_dtype": torch.float32}
    # TODO: 如果设备是MLU
    elif device == "mlu":
        kwargs = {"torch_dtype": torch.float16}
        if num_gpus != 1:
            kwargs["device_map"] = "auto"
            if max_gpu_memory is None:
                kwargs["device_map"] = (
                    "sequential"  # This is important for not the same VRAM
                        sizes
                )
                # TODO:获取每个GPU的可用内存
                available_gpu_memory = get_gpu_memory(num_gpus)
                kwargs["max_memory"] = {
                    i: str(int(available_gpu_memory[i] * 0.85)) + "GiB"
                    for i in range(num_gpus)
                }
            else:
                kwargs["max_memory"] = {i: max_gpu_memory for i in range(
                    num_gpus)}
    # TODO: 如果设备是mps
    elif device == "mps":
        kwargs = {"torch_dtype": torch.float16}
        # Avoid bugs in mps backend by not using in-place operations.
        replace_llama_attn_with_non_inplace_operations()
    else:
        raise ValueError(f"Invalid device: {device}")

    # TODO: 如果启用了CPU offloading
    if cpu_offloading:
        # raises an error on incompatible platforms
        from transformers import BitsAndBytesConfig

        if "max_memory" in kwargs:
            kwargs["max_memory"]["cpu"] = (
                str(math.floor(psutil.virtual_memory().available / 2**20)) + "
                    Mib"
            )
        kwargs["quantization_config"] = BitsAndBytesConfig(
            load_in_8bit_fp32_cpu_offload=cpu_offloading
        )
        kwargs["load_in_8bit"] = load_8bit
    # TODO: 如果启用了8位量化但未启用CPU offloading
    elif load_8bit:
        if num_gpus != 1:
            warnings.warn(
```

```
                "8-bit quantization is not supported for multi-gpu inference."
            )
        else:
            # TODO:调用压缩模型的函数
            return load_compress_model(model_path, device=device, **kwargs)

    # TODO:调用函数，根据给定的模型路径获取适配器，遍历已注册的模型适配器列表，返回第一个匹配的适配器实例。
    adapter = get_model_adapter(model_path)
    # TODO: 使用适配器adapter加载模型和分词器
    model, tokenizer = adapter.load_model(model_path, kwargs)

    if (device == "mlu" and num_gpus == 1 and not cpu_offloading) or device == "mps":
        model.to(device)

    if debug:
        print(model)
    print("MODEL ADAPTER PASS!")
    return model, tokenizer
```

4.3 补全 fastchat/conversation.py

该文件主要定义了对话提示模板和对话类。它包含了不同分隔符样式的枚举类和用于存储对话历史的类。

文件还定义了一些全局对话模板，并提供了注册和获取对话模板的函数。通过这些模板，可以方便地管理和生成不同风格的对话。

作业主要为补全 get_prompt() 函数和 append_message() 函数。

由于篇幅限制，且此部分代码不是重点，补全代码不在此展示，详细内容见代码文件。

4.4 补全 fastchat/model/chatglm_model.py

该文件主要包含两个函数。

stream_chat_token_num() 函数用于计算给定对话历史和当前问题的 token 数量。

chatglm_generate_stream() 函数使用给定的模型和分词器，根据输入参数生成文本流，支持多轮对话，并在生成过程中返回生成的文本和相关的 token 数量。

同理篇幅限制，补全详情见代码文件。

4.5 补全 fastchat/serve/inference.py

该文件是实现推理的核心功能，主要用于 FastChat 模型的推理，它定义了生成文本流的函数 generate_stram, 并导入了 chatglm_generate_stream 函数，用于根据输入参数生成对话文本。

chatloop 负责加载模型和分词器，管理对话循环，并调用相应的生成函数生成和输出对话文本。

这里需要注意的是，在 generate_stream 函数中，要及时关闭梯度计算，否则会导致内存泄漏。

由于此部分是实现推理的核心功能，我将详细展示各补全的接口函数。

代码 3: generate_stream() 函数

```python
def generate_stream(
    model, tokenizer, params, device, context_len=2048, stream_interval=2
):
    #TODO:# 关闭梯度计算
    with torch.no_grad():
        prompt = params["prompt"]
        #TODO: 获取prompt的长度
        len_prompt = len(prompt) if isinstance(prompt, str) else sum(len(s) for
            s in prompt)
        #TODO: 从参数中获取生成文本时的temperature（控制文本生成的多样性），如果
            参数中未指定，则默认为 1.0。
        temperature = float(params.get("temperature", 1.0))
        #TODO: 从参数中获取生成文本时的 repetition penalty（抑制文本中重复的程
            度），如果参数中未指定，则默认为 1.0。
        repetition_penalty = float(params.get("repetition_penalty", 1.0))
        #TODO:从参数中获取生成文本时的 top_p（控制生成文本的多样性），如果参数中
            未指定，则默认为 1.0。
        top_p = float(params.get("top_p", 1.0))
        #TODO: 从输入的参数中获取 top_k 的值，如果参数中没有设置，则默认为 -1。
        top_k = int(params.get("top_k", -1))  # -1 means disable
        #TODO 从参数中获取生成文本时的max_new_tokens数，如果参数中未指定，则默认
            为 256。
        max_new_tokens = int(params.get("max_new_tokens", 256))
        #TODO: 从参数中获取 stop_str，如果未设置，则默认为 None
        stop_str = params.get("stop", None)
        #TODO: 从参数中获取 echo，如果未设置，默认为 True。并将其转换为布尔值。
        echo = bool(params.get("echo", True))
        stop_token_ids = params.get("stop_token_ids", None) or []
        #TODO: 将文本生成停止的标记添加到已有的停止标记列表stop_token_ids中。
        if stop_str and isinstance(stop_str, str):
            # 这里仅取第一个token id作为示例，可根据需要调整
            stop_token_ids.append(tokenizer.encode(stop_str, add_special_tokens=
                False)[0])

        #TODO: 创建一个 logits 处理器列表
        logits_processor = prepare_logits_processor(temperature,
            repetition_penalty, top_p, top_k)

        #TODO: 使用tokenizer将输入文本转换为模型可接受的输入张量
        input_ids = tokenizer.encode(prompt, add_special_tokens=False)
        #TODO: 记录输入文本的长度
        input_echo_len = len(input_ids)
        #TODO: 创建一个名为 output_ids 的列表，其初始值等于 input_ids 列表的内容
        output_ids = input_ids.copy()

        if model.config.is_encoder_decoder:
```

```
            max_src_len = context_len
        else:
            max_src_len = context_len - max_new_tokens - 8

        #TODO: 截取源文本的最后一部分，以适应模型的上下文长度
        if len(input_ids) > max_src_len:
            input_ids = input_ids[-max_src_len:]

        if model.config.is_encoder_decoder:
            encoder_output = model.encoder(
                input_ids=torch.as_tensor([input_ids], device=device)
            )[0]
            start_ids = torch.as_tensor(
                [[model.generation_config.decoder_start_token_id]],
                dtype=torch.int64,
                device=device,
            )

        past_key_values = out = None
        for i in range(max_new_tokens):
            if i == 0:
                if model.config.is_encoder_decoder:
                    #TODO: 使用解码器对起始标记进行处理
                    out = model.decoder(
                        input_ids=start_ids,
                        encoder_hidden_states=encoder_output,
                        use_cache=True,
                    )
                    logits = model.lm_head(out[0])
                else:
                    #TODO: 非编码解码器类型，直接使用模型进行处理
                    out = model(
                        input_ids=torch.as_tensor([input_ids], device=device),
                        use_cache=True,
                    )
                    logits = out.logits
                #TODO: 记录过去的键值
                past_key_values = out.past_key_values
            else:
                if model.config.is_encoder_decoder:
                    #TODO: 使用解码器对当前标记进行处理
                    out = model.decoder(
                        input_ids=torch.as_tensor([[output_ids[-1]]], device=
                            device),
                        encoder_hidden_states=encoder_output,
                        use_cache=True,
                        past_key_values=past_key_values,
```

```
                    )
                    #TODO: 获取logits（预测的标记分布）
                    logits = model.lm_head(out[0])
                else:
                    out = model(
                        #TODO:非编码解码器类型，直接使用模型对当前标记进行处理
                        input_ids=torch.as_tensor([[output_ids[-1]]], device=
                            device),
                        use_cache=True,
                        past_key_values=past_key_values,
                    )
                    logits = out.logits
                #TODO: 更新过去的键值
                past_key_values = out.past_key_values

            if logits_processor:
                if repetition_penalty > 1.0:
                    tmp_output_ids = torch.as_tensor([output_ids], device=logits
                        .device)
                else:
                    tmp_output_ids = None
                #TODO: 使用logits_processor处理最后一个标记的logits
                last_token_logits = logits_processor(tmp_output_ids, logits[:,
                    -1, :])
            else:
                #TODO: 没有logits处理器，直接获取最后一个标记的logits
                last_token_logits = logits[:, -1, :]

            if device == "mps":
                # Switch to CPU by avoiding some bugs in mps backend.
                last_token_logits = last_token_logits.float().to("cpu")

            if temperature < 1e-5 or top_p < 1e-8:  # greedy
                #TODO:# 通过argmax获取概率最高的标记索引，并将其转换为整数类型。
                token = int(torch.argmax(last_token_logits, dim=-1).item())
            else:
                #TODO: 使用softmax将概率分布归一化
                probs = torch.softmax(last_token_logits / temperature, dim=-1)
                #TODO: 使用torch.multinomial函数根据概率分布采样生成标记,并转换
                    为整数类型。
                token = int(torch.multinomial(probs, num_samples=1).item())

            # 将生成的token添加到输出序列output_ids中
            output_ids.append(token)

            if token in stop_token_ids:
                stopped = True
```

```
129             else:
130                 stopped = False
131 
132             #TODO: 判断是否达到生成结果的间隔、已完成生成最大标记数、或者已经停止生成
133             if (i % stream_interval == 0) or (i == max_new_tokens - 1) or stopped:
134                 if echo:
135                     tmp_output_ids = output_ids
136                     rfind_start = len_prompt
137                 else:
138                     tmp_output_ids = output_ids[input_echo_len:]
139                     rfind_start = 0
140 
141                 output = tokenizer.decode(
142                     tmp_output_ids,
143                     skip_special_tokens=True,
144                     spaces_between_special_tokens=False,
145                 )
146                 if stop_str:
147                     if isinstance(stop_str, str):
148                         #TODO: 在输出字符串中从右向左搜索停止标记的位置，rfind_start参数指定了搜索的起始位置
149                         pos = output.rfind(stop_str, rfind_start)
150                         if pos != -1:
151                             #TODO: 将输出字符串截断，仅保留停止标记位置之前的部分
152                             output = output[:pos]
153                             stopped = True
154                     elif isinstance(stop_str, Iterable):
155                         for each_stop in stop_str:
156                             #TODO: 从指定位置（rfind_start）向前搜索每个停止标记在输出字符串中的最后出现位置
157                             pos = output.rfind(each_stop, rfind_start)
158                             if pos != -1:
159                                 #TODO: 将输出字符串截断，仅保留停止标记位置之前的部分
160                                 output = output[:pos]
161                                 stopped = True
162                                 break
163                     else:
164                         raise ValueError("Invalid stop field type.")
165 
166                 yield {
167                     "text": output,
168                     "usage": {
169                         "prompt_tokens": input_echo_len,
```

```
                            "completion_tokens": i,
                            "total_tokens": input_echo_len + i,
                        },
                        "finish_reason": None,
                    }

            if stopped:
                break

        # finish stream event, which contains finish reason
        if i == max_new_tokens - 1:
            finish_reason = "length"
        elif stopped:
            finish_reason = "stop"
        else:
            finish_reason = None

        #TODO: 返回生成的文本、使用情况和结束原因的字典
        yield {
            "text": tokenizer.decode(output_ids[input_echo_len:],
                skip_special_tokens=True, spaces_between_special_tokens=False),
            "usage": {
                "prompt_tokens": input_echo_len,
                "completion_tokens": i if not stopped else i+1,
                "total_tokens": input_echo_len + (i if not stopped else i+1),
            },
            "finish_reason": finish_reason,
        }

        # clean
        del past_key_values, out
        gc.collect()
        #TODO:释放MLU设备上的缓存空间
        torch.mlu.empty_cache()
```

该函数用于在生成模型上执行流式文本生成。其流程包括：

从输入参数中提取生成配置（如温度、重复惩罚、top_p、top_k、最大 token 数、停止字符串/标记等）。

利用 tokenizer 将提示文本转为模型输入，截断以适应模型的最大上下文长度。

根据模型类型（编码器-解码器或普通生成模型），分别调用相应的模型接口进行首次生成和后续 token 生成，同时利用 past_key_values 缓存先前的计算结果以提高效率。

使用 logits 处理器和概率采样方法选取下一个 token，将其拼接到输出 token 列表中。

根据设定的输出间隔、达到最大 token 或遇到停止标记，实时解码并返回当前生成的文本，同时记录 token 使用情况。

最终，当生成完成或触发停止条件时，返回最终的生成结果以及相关的使用信息，并清理缓存。

这里注意，最后的清理缓存，toch_mlu 中不存在相关缓存清理函数（曾困扰了我很久）。且要及时关闭梯度计算，否则会爆显存。

代码 4: chat_loop() 函数

```python
def chat_loop(
    model_path: str,
    device: str,
    num_gpus: int,
    max_gpu_memory: str,
    load_8bit: bool,
    cpu_offloading: bool,
    conv_template: Optional[str],
    temperature: float,
    max_new_tokens: int,
    chatio: ChatIO,
    debug: bool,
):
    #TODO: 调用load_model函数加model和tokenizer
    model, tokenizer = load_model(
        model_path,
        device=device,
        num_gpus=num_gpus,
        max_gpu_memory=max_gpu_memory,
        load_8bit=load_8bit,
        cpu_offloading=cpu_offloading,
    )
    is_chatglm = "chatglm" in str(type(model)).lower()

    #TODO: 如果提供了对话模板, 使用提供的模板,调用get_conv_template创建会话对象
    if conv_template:
        conv = get_conv_template(conv_template)
    else:
        #TODO:否则使用默认的对话模板,调用get_conversation_template创建会话对象
        conv = get_conversation_template("default")
    print("GENERATE STEAM PASS!")
    while True:
        try:
            #TODO: 尝试获取用户输入, 传入 conv.roles[0] 作为角色标识
            inp = chatio.prompt_for_input(conv.roles[0])
        except EOFError:
            inp = ""
        if not inp:
            print("exit...")
            break

        conv.append_message(conv.roles[0], inp)
        conv.append_message(conv.roles[1], None)

        if is_chatglm:
            generate_stream_func = chatglm_generate_stream
```

```
            prompt = conv.messages[conv.offset :]
        else:
            generate_stream_func = generate_stream
            prompt = conv.get_prompt()

        gen_params = {
            "model": model_path,
            "prompt": prompt,
            "temperature": temperature,
            "max_new_tokens": max_new_tokens,
            "stop": conv.stop_str,
            "stop_token_ids": conv.stop_token_ids,
            "echo": False,
        }

        #TODO: 获取机器对话输出, 传入 conv.roles[1] 作为角色标识
        chatio.prompt_for_output(conv.roles[1])
        output_stream = generate_stream_func(model, tokenizer, gen_params,
            device)
        #TODO: # 输出对话的流式输出
        outputs = chatio.stream_output(output_stream)

        # NOTE: strip is important to align with the training data.
        conv.messages[-1][-1] = outputs.strip()
        if debug:
            print("\n", {"prompt": prompt, "outputs": outputs}, "\n")
        print("FASTCHAT INFERENCE PASS!")
```

该函数是实现多轮交互对话的关键功能。

通过加载模型和 tokenizer，根据传入的对话模板参数创建会话对象，（可使用自定义或默认模板）。在循环中，每次获取用户输入，并将用户信息添加到对话记录中。根据模型类型选择生成函数（chatglm 或普通生成），然后构造生成参数，并调用模型进行流式文本生成，然后实时显示生成结果。

4.6 补全 fastchat/serve/cli.py

该文件提供了一个命令行界面与模型进行对话。还定义了两个类，SimpleChatIO 和 RichChatIO，分别用于简单对话和富文本对话，都继承自 ChatIO 基类。

主函数 main 解析命令行参数，选择合适的输入输出类，并调用 chat_loop 函数启动聊天循环，实现与模型的交互。

其中还含有命令行参数的定义和选项。

该文件是对话程序的主逻辑，我将详细展示补全代码：

代码 5: SimpleChatIO 类

```
    #TODO:构建SimpleChatIO类, 它继承自ChatIO类
class SimpleChatIO(ChatIO):
    def prompt_for_input(self, role) -> str:
        return input(f"{role}: ")
```

```
5
6       def prompt_for_output(self, role: str):
7           print(f"{role}: ", end="", flush=True)
8
9       def stream_output(self, output_stream):
10          pre = 0
11          for outputs in output_stream:
12              output_text = outputs["text"]
13              #TODO: 移除文本两端的空白字符，然后按空格分割
14              output_text = output_text.strip().split(" ")
15              #TODO: 获取处理后的文本中单词的数量
16              now = len(output_text)
17              if now > pre:
18                  # 输出不同于前次的部分,其中，新增的内容以空格分隔的形式显示，确
                        保每次新增的内容都在同一行，并及时刷新输出缓冲区。
19                  print(" ".join(output_text[pre:now]), end=" ", flush=True)
20                  pre = now
21          print(" ".join(output_text[pre:]), flush=True)
22          print("CHATIO PASS!")
23          return " ".join(output_text)
```

该类继承自 inference.py 文件中的 ChatIO 类，用于实现简单的命令行交互输入输出。主要功能包括 prompt_for_input()，根据指定角色提示用户输入，并返回输入内容；prompt_for_output()，在命令行中打印角色标识，准备输出生成的回答；stream_output()，流式处理生成器返回的文本，在处理过程中，函数会对文本空白字符去除和按空格分割，然后只输出新生成的部分，确保输出及时刷新，直至完成整个生成过程，最终函数返回完整的输出文本。

代码 6: RichChatIO 类

```
1       #TODO:构建RichChatIO类，它继承自ChatIO类
2   class RichChatIO(ChatIO):
3       def __init__(self):
4           #TODO: 创建PromptSession实例，用于获取用户输入，并设置输入历史记录
5           self._prompt_session = PromptSession(history=InMemoryHistory())
6           #TODO: 创建自动补全器，用于用户输入的自动完成
7           self._completer = WordCompleter(words=["!exit", "!reset"], pattern=re.
                compile(r"$"))
8           #TODO:创建Console 实例，在命令行界面中以更丰富的样式显示文本
9           self._console = Console()
10
11      def prompt_for_input(self, role) -> str:
12          self._console.print(f"[bold]{role}:")
13          # TODO(suquark): multiline input has some issues. fix it later.
14          prompt_input = self._prompt_session.prompt(
15              completer=self._completer,
16              multiline=False,
17              #TODO: 启用自动建议功能
18              auto_suggest=AutoSuggestFromHistory(),
```

```
            key_bindings=None,
        )
        self._console.print()
        return prompt_input

    def prompt_for_output(self, role: str):
        self._console.print(f"[bold]{role}:")

    def stream_output(self, output_stream):
        """Stream output from a role."""
        # TODO(suquark): the console flickers when there is a code block
        #  above it. We need to cut off "live" when a code block is done.

        # Create a Live context for updating the console output

        #TODO: 创建Live上下文管理器，用于实现在命令行中实时更新显示。其中，需要指定要在其上执行实时更新的console实例,每秒刷新的次数设为4
        with Live(console=self._console, refresh_per_second=4) as live:
            # Read lines from the stream
            for outputs in output_stream:
                if not outputs:
                    continue
                # 如果输出是字典，则获取字典中的"text"键对应的值，否则直接获取输出
                if isinstance(outputs, dict):
                    text = outputs.get("text", "")
                else:
                    text = outputs

                lines = []
                for line in text.splitlines():
                    if "### Human" in line:
                        continue
                    lines.append(line)
                    if line.startswith("```"):
                        # Code block marker - do not add trailing spaces, as it would
                        #  break the syntax highlighting
                        lines.append("\n")
                    else:
                        lines.append("  \n")
                markdown = Markdown("".join(lines))
                #TODO: 将渲染后的 markdown 文本实时更新到控制台
                live.update(markdown)
        self._console.print()
        print("CHATIO PASS!")
        return text
```

该类继承自 inference.py 文件中的 ChatIO 类，用于实现富文本的命令行交互输入输出。同样包括在控制台以加粗样式输出角色标识，并使用 PromptSession 获取用户输入, 同时启用历史自动建议。在 stream_output() 函数中，使用 Rich 的 Live 上下文管理器实现实时刷新，将流式获取的输出文本以 Markdown 格式渲染，然后实时更新显示，确保生成过程在终端平滑显示。最终稿返回输出文本。提供了更加美观的交互界面。

代码 7: 主函数部分

```python
def main(args):
    if args.gpus:
        if len(args.gpus.split(",")) < args.num_gpus:
            raise ValueError(
                f"Larger --num-gpus ({args.num_gpus}) than --gpus {args.gpus}!"
            )
        os.environ["CUDA_VISIBLE_DEVICES"] = args.gpus

    if args.style == "simple":
        chatio = SimpleChatIO()
    elif args.style == "rich":
        chatio = RichChatIO()
    else:
        raise ValueError(f"Invalid style for console: {args.style}")
    try:
        #TODO:调用 chat_loop 函数, 启动聊天循环
        chat_loop(
            args.model_path,
            args.device,
            args.num_gpus,
            args.max_gpu_memory,
            args.load_8bit,
            args.cpu_offloading,
            args.conv_template,
            args.temperature,
            args.max_new_tokens,
            chatio,
            args.debug,
        )
    except KeyboardInterrupt:
        print("exit...")


if __name__ == "__main__":
    #TODO:  创建一个ArgumentParser对象, 用于解析命令行参数
    parser = argparse.ArgumentParser()
    #TODO: 向ArgumentParser对象中添加模型相关的参数, 这些参数由add_model_args函数定义
    add_model_args(parser)
    parser.add_argument(
```

```
        "--conv-template", type=str, default=None, help="Conversation prompt
            template."
    )
    parser.add_argument("--temperature", type=float, default=0.7)
    parser.add_argument("--max-new-tokens", type=int, default=512)

    #TODO:试一下simple模式，同时也试一下rich模式。
    parser.add_argument(
        "--style",
        type=str,
        default="simple",
        choices=["simple", "rich"],
        help="Display style.",
    )
    parser.add_argument("--debug", action="store_true", help="Print debug
        information")
    args = parser.parse_args()
    main(args)
```

这段主函数部分，实现了命令行应用程序的入口。主要包括：根据命令行参数配置 GPU，根据用户指定的显示风格（simple 或 rich）实例化对应的 ChatIO 类，以确定交互界面的显示方式。解析并设置模型相关与对话生成的参数，例如模型路径，对话模板，温度，最大 token 数等。调用 inference.py 文件中 chat_loop 函数，启动多轮对话循环，进行用户输入，文本生成与输出展示。同时有捕获异常，实现用户中断时的退出。

整体上这一部分负责初始化应用环境，解析参数，并启动聊天交互过程。同时也是测试环境中，脚本测试的对象文件。

在开发环境中，通过运行当前目录下的的 run_scripts 文件夹中的 run_mlu_infer.sh 脚本，可以对 cli.py 进行测试，从而测试整个模型推理效果。

脚本命令如下：

代码 8: run_completion.sh

```
set -e

model_path=/workspace/model/favorite/large-scale-models/model-v1/Llama-2-7b-hf
echo "**********************************"
echo "**命令行终端形式 仅支持单轮对话**"
echo "**********************************"
# 单卡
# 配置板卡
export MLU_VISIBLE_DEVICES=0
python3 -m fastchat.serve.cli --model-path ${model_path} --style=rich
```

在脚本命令后输入相应参数可实现多种功能，如调试模式，选择富文本或简单文本等。

运行结果如图1-3所示：

在测试中，我询问了 AI，“你是谁”，“询问了质能方程”，“询问了牛顿第二定律”。AI 均能正确回答问题。

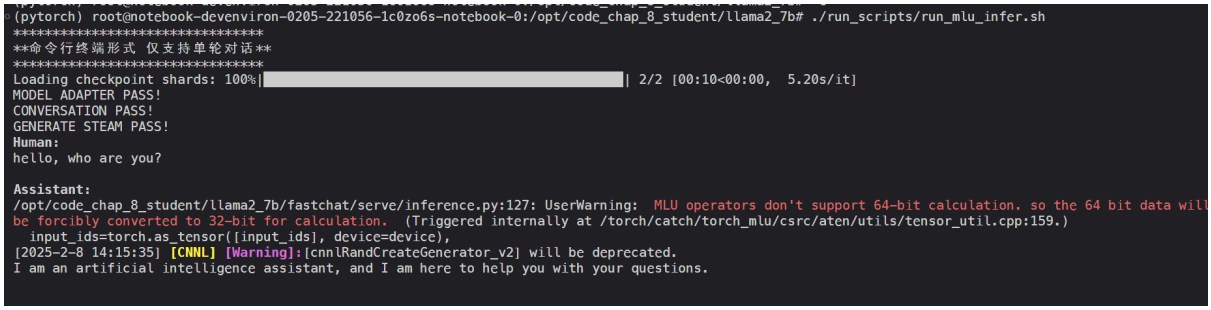

图 **1:** 多轮对话结果展示 1

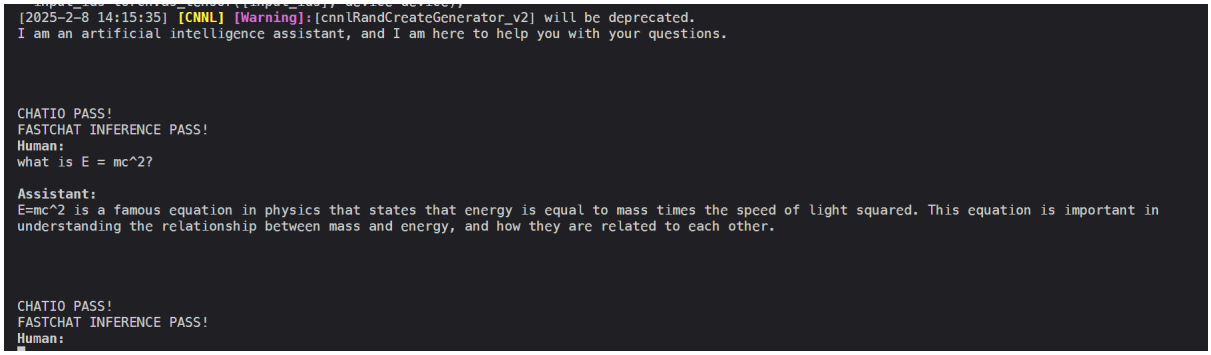

图 **2:** 多轮对话结果展示 2

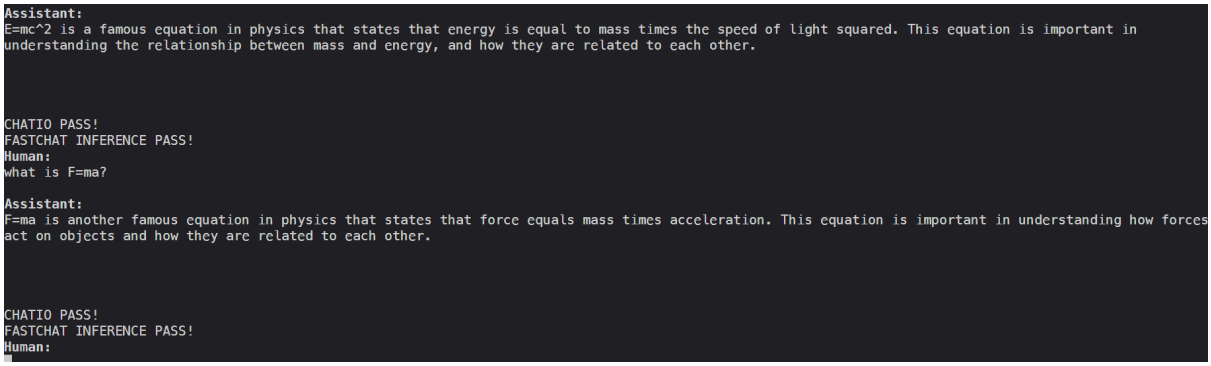

图 **3:** 多轮对话结果展示 3

可见，正确实现了对话功能，可以与模型进行多轮对话交互。

之前在希冀平台提交的结果中，最后两项 chatio 和 fastchat_inference 始终未通过。后来经过助教通知，了解到，希冀平台的评测机机制中，没法处理多轮对话交互的情况。解决办法是将 inference.py 中的 chat_loop 函数中的 while(true) 循环逻辑移除，并为 inp 赋值为一个固定的内容，即可。修改的代码部分如下, 见注释部分和 inp 变量：

代码 9: 修改交互部分

```
def chat_loop(
    model_path: str,
    device: str,
    num_gpus: int,
    max_gpu_memory: str,
    load_8bit: bool,
    cpu_offloading: bool,
    conv_template: Optional[str],
    temperature: float,
    max_new_tokens: int,
    chatio: ChatIO,
    debug: bool,
):
    # TODO: 调用load_model函数加model和tokenizer
    model, tokenizer = load_model(
        model_path,
        device=device,
        num_gpus=num_gpus,
        max_gpu_memory=max_gpu_memory,
        load_8bit=load_8bit,
        cpu_offloading=cpu_offloading,
    )
    is_chatglm = "chatglm" in str(type(model)).lower()

    # TODO: 如果提供了对话模板, 使用提供的模板,调用get_conv_template创建会话对象
    if conv_template:
        conv = get_conv_template(conv_template)
    else:
        # TODO:否则使用默认的对话模板,调用get_conversation_template创建会话对象
        conv = get_conversation_template("default")
    print("GENERATE STEAM PASS!")
    # while True:
    #     try:
    #         # TODO: 尝试获取用户输入, 传入 conv.roles[0] 作为角色标识
    #         inp = chatio.prompt_for_input(conv.roles[0])
    #     except EOFError:
    #         inp = ""
    #     if not inp:
    #         print("exit...")
    #         break
```

```
    inp = "hello, who are you?"
    conv.append_message(conv.roles[0], inp)
    conv.append_message(conv.roles[1], None)

    if is_chatglm:
        generate_stream_func = chatglm_generate_stream
        prompt = conv.messages[conv.offset :]
    else:
        generate_stream_func = generate_stream
        prompt = conv.get_prompt()

    gen_params = {
        "model": model_path,
        "prompt": prompt,
        "temperature": temperature,
        "max_new_tokens": max_new_tokens,
        "stop": conv.stop_str,
        "stop_token_ids": conv.stop_token_ids,
        "echo": False,
    }

    # TODO: 获取机器对话输出, 传入 conv.roles[1] 作为角色标识
    chatio.prompt_for_output(conv.roles[1])
    output_stream = generate_stream_func(model, tokenizer, gen_params, device)
    # TODO: # 输出对话的流式输出
    outputs = chatio.stream_output(output_stream)

    # NOTE: strip is important to align with the training data.
    conv.messages[-1][-1] = outputs.strip()
    if debug:
        print("\n", {"prompt": prompt, "outputs": outputs}, "\n")
    print("FASTCHAT INFERENCE PASS!")
```

最后，希冀平台评测结果如图4：
可见，成功通过了所有测试点，实验完成。

5 实验总结

5.1 实验过程回顾

实验初期，面对复杂的实验环境和众多的代码文件，我感到有些无从下手。但在逐步梳理实验步骤后，我开始按照计划补全各个文件中的代码。我一边查阅资料学习相关知识，一边仔细推敲每一行代码的逻辑，确保其功能的正确实现。

5.2 实验成果与不足

经过不懈努力，我成功实现了基于 Llama 2 的聊天机器人，并通过命令行界面与模型进行了多轮对话交互。在开发环境中测试时，结果显示了最终的 pass 信息，表明实验功能基本实现。然而，在希冀平台

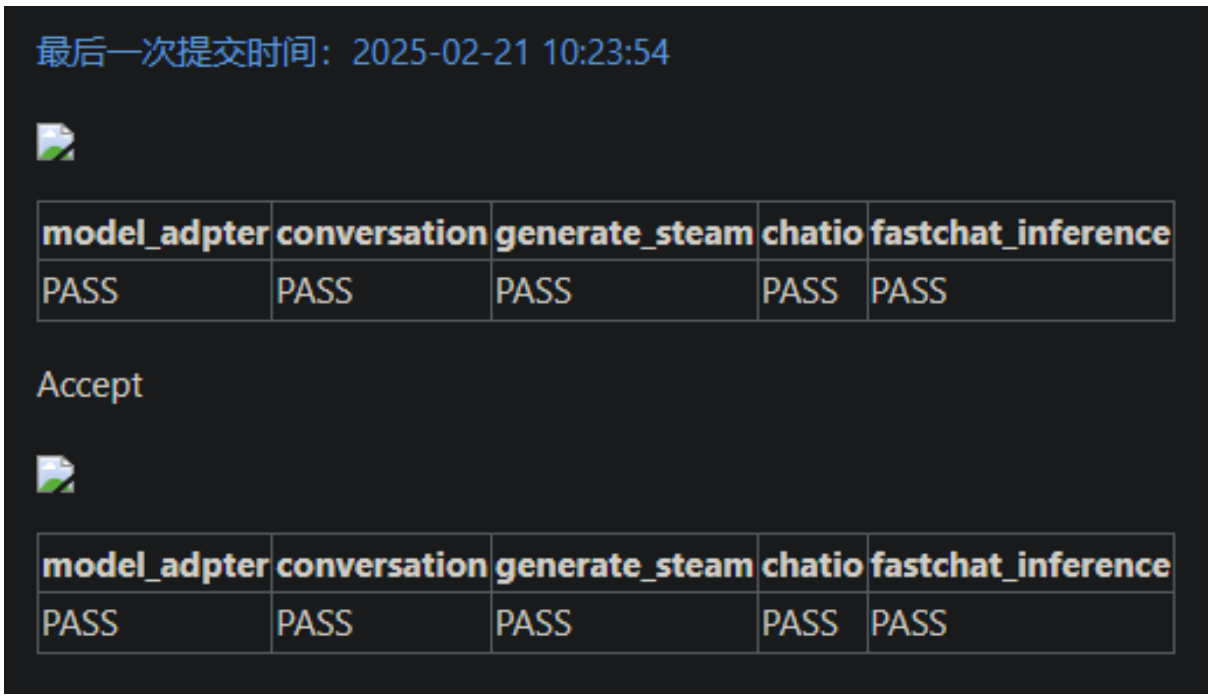

图 4: 8-2 测试结果

的判题系统中，却未通过最后两项 chatio 和 fastchat_inference 测试，这让我感到有些遗憾。尽管我尝试了多种方法进行调试，但仍未找到具体原因，可能是接口功能未被测试捕捉到，也可能是代码本身存在一些隐藏的 BUG。

5.3 实验收获

技术能力提升： 通过本次实验，我深入学习了 Llama 2 模型的相关知识，掌握了在 DLP 平台上进行模型移植优化的方法，熟悉了 Pytorch 框架下模型的加载、适配、推理等操作流程，提升了自己在智能计算领域的技术能力。

问题解决能力锻炼： 在实验过程中遇到的各种问题，促使我不断查阅资料、分析代码、调试程序，锻炼了我的问题解决能力。面对复杂的代码和未知的错误，我学会了保持冷静，从不同角度思考问题，逐步排查并解决问题。

理论与实践结合： 本次实验将课堂上学到的理论知识与实际操作紧密结合，让我更加深刻地理解了智能计算系统的原理和应用。通过亲自动手实现聊天机器人，我体会到了理论指导实践、实践检验理论的重要性。

5.4 未来展望

虽然本次实验取得了一定的成果，但也暴露出了一些不足之处。在今后的学习和实践中，我将继续深入研究智能计算领域的相关技术，不断提升自己的技术水平。同时，我也会更加注重细节，努力提高代码的质量和稳定性，避免类似的问题再次出现。此外，我还希望能够将所学知识应用到更多的实际项目中，为智能计算技术的发展贡献自己的一份力量。