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 Code Llama 实现代码生成
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 8-3 基于 Code Llama 实现代码生成

1　实验目的

- 本实验旨在介绍基于 Code Llama 实现代码生成的基本原理和操作步骤；
- 学生将深入了解基于 Code Llama 实现代码生成关键步骤，包括基于 transformers 库的 Codel Llama 推理模块、Transformer 模型构建模块、Llama 代码生成模块等；
- 掌握如何在 DLP 平台上部署基于 Code Llama 的代码生成模型，能够实现 Code Llama 代码生成模块、Code Llama 代码补全模块、Code Llama 指令微调模块。

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：能够正确实现基于 transformers 库的 Codel Llama 推理模块。
- 70 分标准：在 60 分标准基础上，能够正确实现 Transformer 模型构建模块、Llama 代码生成模块。
- 80 分标准：在 70 分标准基础上，能够正确实现 Code Llama 代码生成模块。
- 90 分标准：在 80 分标准基础上，能够正确实现 Code Llama 代码补全模块。
- 100 分标准：在 90 分标准基础上，能够正确实现 Code Llama 指令微调模块。

4　实验内容及步骤

4.1　补全 llama_mlu/model.py

在该文件中，ModelArgs 定义了模型的参数。RMSNorm 实现了 RMS 归一化层，用于对输入张量进行归一化处理等。

该文件主要定义了一个 Transformer 模型的结构和功能，包括模型参数、归一化层、多头注意力机制、前馈神经网络层、Transformer 块以及整个 Transformer 模型的实现。还包括了一些辅助函数，用于处理张量的形状和频率嵌入。

由于篇幅限制，补全详情见代码

4.2　补全 llama_mlu/generation.py

给该文件主要实现了基于 Llama 模型的文本生成系统，包括文本补全、文本填充和对话生成等多种任务。通过定义 Llama 类封装了模型构建、参数加载、分布式初始化以及生成过程。同时提供了辅助函数（如 sample_top_p、infilling_prompt_tokens 和 dialog_prompt_tokens）用于处理采样、提示编码和对话格式化，从而支持多种生成场景。

由于该文件是实现推理的核心功能，我将补全代码部分展示出。

代码 1: build 函数

```
1    class Llama:
2    @staticmethod
```

```
    def build(
        ckpt_dir: str,
        tokenizer_path: str,
        max_seq_len: int,
        max_batch_size: int,
        model_parallel_size: Optional[int] = None,
    ) -> "Llama":
        if not torch.distributed.is_initialized():
            if device == "mlu":
                # TODO: 使用 MLU 设备初始化分布式进程组
                torch.distributed.init_process_group("cncl")
            else:
                torch.distributed.init_process_group("gloo")
        if not model_parallel_is_initialized():
            if model_parallel_size is None:
                model_parallel_size = int(os.environ.get("WORLD_SIZE", 1))
            initialize_model_parallel(model_parallel_size)

        local_rank = int(os.environ.get("LOCAL_RANK", 0))
        if device == "mlu":
            # TODO:  如果设备为 MLU, 则设置当前进程的 MLU 设备
            torch.mlu.set_device(local_rank)

        # seed must be the same in all processes
        torch.manual_seed(1)

        if local_rank > 0:
            sys.stdout = open(os.devnull, "w")

        start_time = time.time()
        checkpoints = sorted(Path(ckpt_dir).glob("*.pth"))
        assert len(checkpoints) > 0, f"no checkpoint files found in {ckpt_dir}"
        assert model_parallel_size == len(
            checkpoints
        ), f"Loading a checkpoint for MP={len(checkpoints)} but world size is {
            model_parallel_size}"
        ckpt_path = checkpoints[get_model_parallel_rank()]
        # TODO: 加载模型的检查点文件, 并将模型加载到 CPU 上。
        checkpoint = torch.load(ckpt_path, map_location="cpu")
        with open(Path(ckpt_dir) / "params.json", "r") as f:
            params = json.loads(f.read())

        model_args: ModelArgs = ModelArgs(
            max_seq_len=max_seq_len,
            max_batch_size=max_batch_size,
            **params,
        )
```

```
        # TODO: 调用Tokenizer函数
        tokenizer = Tokenizer(tokenizer_path)
        model_args.vocab_size = tokenizer.n_words
        # support for mac
        # print(device)
        if device == "mlu":
            torch.set_default_tensor_type(torch.HalfTensor)
        else:
            torch.set_default_tensor_type(torch.HalfTensor)
        # TODO: 调用Transformer 模型
        model = Transformer(model_args)
        # TODO: 加载模型的参数字典
        model.load_state_dict(checkpoint, strict=False)
        print("TRANSFORMER MODEL PASS!")
        # add start
        # print(device)
        if device == "cpu":
            model = model.float()

        # add end
        model.to(device)
        print(f"Loaded in {time.time() - start_time:.2f} seconds")

        print("LLAMA BUILD PASS!")
        return Llama(model, tokenizer)
```

该 build 函数主要是构建并初始化 Llama 模型，包括初始化分布式进程组，根据环境变量设置本地设备，加载指定检查点目录下的预训练模型和模型配置，根据检查点文件数与模型并行大小进行校验，调用 Tokenizer 加载分词器，根据分词器设置模型词汇表大小，构建 Transformer 模型，并加载检查点中的权重，最后将模型转移到指定设备上。返回封装了模型与分词器的 Llama 生成器实例。

代码 2: generate 函数

```
        def generate(
        self,
        prompt_tokens: List[List[int]],
        max_gen_len: int,
        temperature: float = 0.6,
        top_p: float = 0.9,
        logprobs: bool = False,
        echo: bool = False,
        stop_token: Optional[int] = None,
    ) -> Tuple[List[List[int]], Optional[List[List[float]]]]:
        # print("调用了generate")
        if stop_token is None:
            stop_token = self.tokenizer.eos_id
        params = self.model.params
        bsz = len(prompt_tokens)
```

```
        assert bsz <= params.max_batch_size, (bsz, params.max_batch_size)
        # TODO:  获取提示文本序列中最短的长度
        min_prompt_len = min(len(t) for t in prompt_tokens)
        # TODO: 获取提示文本序列中最长的长度
        max_prompt_len = max(len(t) for t in prompt_tokens)
        assert max_prompt_len <= params.max_seq_len
        # TODO: 计算生成的总长度，需考虑提示文本和最大生成长度
        total_len = max_prompt_len + max_gen_len
        pad_id = self.tokenizer.pad_id
        tokens = torch.full((bsz, total_len), pad_id, dtype=torch.long, device=
            device)
        for k, t in enumerate(prompt_tokens):
            # TODO: 将提示文本编码添加到张量中
            tokens[k, : len(t)] = torch.tensor(t, dtype=torch.long, device=
                device)
        if logprobs:
            # TODO: 创建一个与tokens张量具有相同形状的全零张量
            token_logprobs = torch.zeros_like(tokens, dtype=torch.float)
        prev_pos = 0
        stop_reached = torch.tensor([False] * bsz, device=device).to(device)
        input_text_mask = tokens != pad_id
        # print("generate第189行")
        # k = 1
        # print(f"min_prompt_len={min_prompt_len},total_len={total_len}")
        for cur_pos in range(min_prompt_len, total_len):
            logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos)
            # print(f"开始{k}次for循环")
            if logprobs:
                token_logprobs[:, prev_pos + 1 : cur_pos + 1] = -F.cross_entropy
                    (
                    input=logits.transpose(1, 2),
                    target=tokens[:, prev_pos + 1 : cur_pos + 1],
                    reduction="none",
                    ignore_index=pad_id,
                )
            if temperature > 0:
                # TODO: 对模型的输出进行 softmax 归一化，以得到每个可能的下一个
                    token的概率分布，其中 temperature 用于控制模型输出的多样性
                probs = F.softmax(logits[:, -1, :] / temperature, dim=-1)
                # TODO: 根据概率分布采样出下一个 token
                next_token = torch.multinomial(probs, num_samples=1)
            else:
                # TODO: 直接选择logits最大的位置作为下一个token，不进行随机采样
                next_token = torch.argmax(logits[:, -1, :], dim=-1, keepdim=True
                    )
            next_token = next_token.reshape(-1)
            # only replace token if prompt has already been generated
```

```
            next_token = torch.where(
                input_text_mask[:, cur_pos], tokens[:, cur_pos], next_token
            )
            tokens[:, cur_pos] = next_token
            stop_reached |= (~input_text_mask[:, cur_pos]) & (next_token ==
                stop_token)
            prev_pos = cur_pos
            # print(f"经历了{k}次for循环")
            # k += 1
            if all(stop_reached):
                break
        # print("for循环结束了")
        if logprobs:
            # TODO: 将张量转换为列表格式
            token_logprobs = token_logprobs.tolist()
        out_tokens, out_logprobs = [], []
        for i, toks in enumerate(tokens.tolist()):
            # cut to max gen len
            start = 0 if echo else len(prompt_tokens[i])
            # TODO: 截取生成的标记序列，直到达到最大生成长度
            toks = toks[start : start + max_gen_len]
            probs = None
            if logprobs:
                probs = token_logprobs[i][start : len(prompt_tokens[i]) +
                    max_gen_len]
            # cut to stop token if present
            if stop_token in toks:
                stop_idx = toks.index(stop_token)
                toks = toks[:stop_idx]
                probs = probs[:stop_idx] if logprobs else None
            # TODO: 将截取后的标记序列添加到输出列表中
            out_tokens.append(toks)
            # TODO: 将截取后的log概率列表添加到输出列表中
            if logprobs:
                out_logprobs.append(probs)
        print("LLAMA GENERATE PASS!")
        return (out_tokens, out_logprobs if logprobs else None)
```

该 generate 函数负责根据输入的提示文本，迭代生成后续 token，直到达到最大生成长度或遇到停止 token 为止。其流程是：对输入的 prompt_tokens 进行 padding，并设置生成序列的总长度。循环调用模型的 forward 方法，基于当前生成的序列预测下一个 token。根据温度参数对模型输出进行 softmax 归一化，然后采样得到下一个 token（或直接取最大概率 token）。使用 mask 确保已存在的 prompt token 不被覆盖，并检查是否达到停止条件，最后将生成的 token 序列（以及可选的 log 概率）截取后返回。

代码 3: text_completion 函数

```
        def text_completion(
        self,
```

```
        prompts: List[str],
        temperature: float = 0.6,
        top_p: float = 0.9,
        max_gen_len: Optional[int] = None,
        logprobs: bool = False,
        echo: bool = False,
    ) -> List[CompletionPrediction]:
        if max_gen_len is None:
            max_gen_len = self.model.params.max_seq_len - 1
        prompt_tokens = [
            self.tokenizer.encode(str(x), bos=True, eos=False) for x in prompts
        ]
        # TODO: 调用 generate 方法生成文本
        generation_tokens, generation_logprobs = self.generate(
            prompt_tokens, max_gen_len, temperature, top_p, logprobs, echo
        )
        if logprobs:
            assert generation_logprobs is not None
            return [
                {
                    "generation": self.tokenizer.decode(t),
                    "tokens": [self.tokenizer.token_piece(x) for x in t],
                    "logprobs": logprobs_i,
                }
                for t, logprobs_i in zip(generation_tokens, generation_logprobs)
            ]
        print("LLAMA TEXTCOMPLETION PASS!")
        return [{"generation": self.tokenizer.decode(t)} for t in
            generation_tokens]
```

该函数实现了文本补全任务的具体流程：首先将输入的字符串提示使用分词器进行编码生成 token 序列；然后调用 generate 函数基于这些 token 序列生成后续文本，生成序列长度由 max_gen_len 决定；最后根据是否需要返回 log 概率，将生成的 token 序列解码为文本输出，并包装成包含生成文本、token 碎片和 log 概率（可选）的字典列表返回。

代码 4: text_infilling 函数

```
        def text_infilling(
        self,
        prefixes: List[str],
        suffixes: List[str],
        temperature: float = 0.6,
        top_p: float = 0.9,
        max_gen_len: Optional[int] = None,
        logprobs: bool = False,
        suffix_first: bool = False,
    ) -> List[InfillingPrediction]:
        assert self.tokenizer.eot_id is not None
```

```
        # print("调用了text_infilling")
        if max_gen_len is None:
            max_gen_len = self.model.params.max_seq_len - 1
        prompt_tokens = [
            # TODO: 调用函数对每个前缀和后缀进行处理，生成填充问题的编码
            infilling_prompt_tokens(self.tokenizer, prefix, suffix, suffix_first
                )
            for prefix, suffix in zip(prefixes, suffixes)
        ]
        # TODO: 调用 generate 方法生成文本
        generation_tokens, generation_logprobs = self.generate(
            prompt_tokens,
            max_gen_len,
            temperature,
            top_p,
            logprobs,
        )

        generations = [self.tokenizer.decode_infilling(t) for t in
            generation_tokens]
        print("LLAMA TEXTINFILLING PASS!")
        if logprobs:
            assert generation_logprobs is not None
            return [
                {
                    "generation": generation,
                    "logprobs": logprobs_i,
                    "tokens": [self.tokenizer.token_piece(x) for x in t],
                    "full_text": prefix + generation + suffix,
                }
                for prefix, suffix, generation, t, logprobs_i in zip(
                    prefixes,
                    suffixes,
                    generations,
                    generation_tokens,
                    generation_logprobs,
                )
            ]
        else:
            return [
                {
                    "generation": generation,
                    "full_text": prefix + generation + suffix,
                }
                for prefix, suffix, generation in zip(prefixes, suffixes,
                    generations)
            ]
```

该函数用于在文本中插入缺失内容。首先将前缀和后缀编码为提示序列，然后调用 generate 函数生成中间填充文本，最后将生成的文本与前后缀拼接并返回，可选择输出每个 token 的 log 概率信息。

代码 5: chat_completion 函数

```
        def chat_completion(
        self,
        dialogs: List[Dialog],
        temperature: float = 0.6,
        top_p: float = 0.9,
        max_gen_len: Optional[int] = None,
        logprobs: bool = False,
    ) -> List[ChatPrediction]:
        if self.tokenizer.step_id is not None:
            ## 如果模型支持 step_id, 则使用另一种chat_completion的方法
            return self._chat_completion_with_step_id(dialogs, temperature,
                top_p, max_gen_len, logprobs)
        if max_gen_len is None:
            max_gen_len = self.model.params.max_seq_len - 1
        prompt_tokens = []
        unsafe_requests = []
        for dialog in dialogs:
            unsafe_requests.append(
                any([tag in msg["content"] for tag in SPECIAL_TAGS for msg in
                    dialog])
            )
            if dialog[0]["role"] == "system":
                dialog = [  # type: ignore
                    {
                        "role": dialog[1]["role"],
                        "content": B_SYS
                        + dialog[0]["content"]
                        + E_SYS
                        + dialog[1]["content"],
                    }
                ] + dialog[2:]
            assert all([msg["role"] == "user" for msg in dialog[::2]]) and all(
                [msg["role"] == "assistant" for msg in dialog[1::2]]
            ), (
                "model only supports 'system', 'user' and 'assistant' roles, "
                "starting with 'system', then 'user' and alternating (u/a/u/a/u
                    ...)"
            )
            dialog_tokens: List[int] = sum(
                [
                    self.tokenizer.encode(
                        f"{B_INST} {prompt['content'].strip()} {E_INST} {answer
                            ['content'].strip()} ",
```

```
                        bos=True,
                        eos=True,
                    )
                    for prompt, answer in zip(
                        dialog[::2],
                        dialog[1::2],
                    )
                ],
                [],
            )
            assert (
                dialog[-1]["role"] == "user"
            ), f"Last message must be from user, got {dialog[-1]['role']}"
            dialog_tokens += self.tokenizer.encode(
                f"{B_INST} {dialog[-1]['content'].strip()} {E_INST}",
                bos=True,
                eos=False,
            )
            prompt_tokens.append(dialog_tokens)
        # TODO: 调用 generate 方法生成文本
        generation_tokens, generation_logprobs = self.generate(
            prompt_tokens,
            max_gen_len,
            temperature,
            top_p,
            logprobs,
        )
        print("LLAMA CHATCOMPLETION PASS!")
        if logprobs:
            assert generation_logprobs is not None
            return [
                {
                    "generation": {  # type: ignore
                        "role": "assistant",
                        "content": (
                            self.tokenizer.decode(t) if not unsafe else
                                UNSAFE_ERROR
                        ),
                    },
                    "tokens": [self.tokenizer.token_piece(x) for x in t],
                    "logprobs": logprobs_i,
                }
                for t, logprobs_i, unsafe in zip(
                    generation_tokens, generation_logprobs, unsafe_requests
                )
            ]
        return [
```

```
            {
                "generation": {  # type: ignore
                    "role": "assistant",
                    "content": self.tokenizer.decode(t) if not unsafe else
                        UNSAFE_ERROR,
                }
            }
            for t, unsafe in zip(generation_tokens, unsafe_requests)
        ]
```

此函数用于生成聊天对话的补全内容。它会将输入的对话消息转换为提示 tokens，调用 generate 函数生成回复，然后将生成的结果解码为文本返回。它还会检测对话中是否包含非法标记，遇到这些内容则输出错误提示。

代码 6: _chat_completion_turns 函数

```
        def _chat_completion_turns(
        self,
        dialogs: List[Dialog],
        temperature: float = 0.6,
        top_p: float = 0.9,
        max_gen_len: Optional[int] = None,
        logprobs: bool = False,
    ) -> List[ChatPrediction]:
        if self.tokenizer.step_id is None:
            raise RuntimeError("Model not suitable for chat_completion_step()")
        if max_gen_len is None:
            max_gen_len = self.model.params.max_seq_len - 1

        prompt_tokens = []
        unsafe_requests = []
        for dialog in dialogs:
            unsafe_requests.append(
                any([tag in msg["content"] for tag in SPECIAL_TAGS for msg in
                    dialog])
            )

            # Insert system message if not provided
            if dialog[0]["role"] != "system":
                dialog = [{"role": "system", "content": ""}] + dialog  # type:
                    ignore
            # TODO:调用函数将对话格式化为模型可处理的对话提示编码
            dialog_tokens = dialog_prompt_tokens(self.tokenizer, dialog)
            prompt_tokens.append(dialog_tokens)
        # TODO: 调用 generate 方法生成文本
        generation_tokens, generation_logprobs = self.generate(
            prompt_tokens,
            max_gen_len,
```

```
            temperature,
            top_p,
            logprobs,
        )
        if logprobs:
            assert generation_logprobs is not None
            return [
                {
                    "generation": {
                        "role": "assistant",
                        "destination": "user",
                        "content": (
                            self.tokenizer.decode(t) if not unsafe else
                                UNSAFE_ERROR
                        ),
                    },
                    "tokens": [self.tokenizer.token_piece(x) for x in t],
                    "logprobs": logprobs_i,
                }
                for t, logprobs_i, unsafe in zip(
                    generation_tokens, generation_logprobs, unsafe_requests
                )
            ]
        return [
            {
                "generation": {
                    "role": "assistant",
                    "destination": "user",
                    "content": self.tokenizer.decode(t) if not unsafe else
                        UNSAFE_ERROR,
                }
            }
            for t, unsafe in zip(generation_tokens, unsafe_requests)
        ]
```

该函数用于在支持 step_id 的场景下生成多轮对话的回复：如果对话中缺少 system 角色，会自动插入一条空的 system 消息。调用 dialog_prompt_tokens 将对话转为可处理的 token 序列。使用 generate 函数生成后续内容，解码成文本并封装为 JSON 结构返回。若对话包含非法标记，则输出相应错误提示。

4.3 补全 Codellama-inference.py

该文件主要功能是测试 Code Llama 模型在代码补全任务中的性能。
由于篇幅限制，补全详情见代码
运行7以测试代码效果

代码 7: run-cll.sh

```
export MLU_VISIBLE_DEVICES='0,1'
```

```
python Codellama-infer.py
```

测试效果如图1所示。

值得注意的是，在运行过程中，不断提示寒武纪的 mlu 不支持 64 位的数据类型，因此需要将代码中的数据类型改为 32 位。

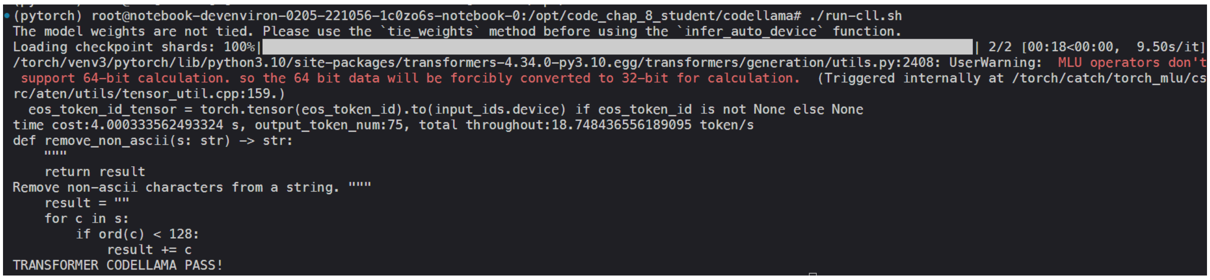

图 1: Codellama-inference.py 测试结果

4.4 补全 example_completion_mlu.py

该文件主要功能是使用 Llama 模型进行文本生成任务，使用了 generate.py 文件中 Llama 类的 text_completion 方法。

由于篇幅限制，补全详情见代码

运行8脚本后，能显示测试结果。

代码 8: run_completion.sh

```
torchrun --nproc_per_node 1 example_completion_mlu.py \
    --ckpt_dir /workspace/model/favorite/large-scale-models/model-v1/CodeLlama-7b/ \
    --tokenizer_path /workspace/model/favorite/large-scale-models/model-v1/CodeLlama-7b/tokenizer.model \
    --max_seq_len 128 --max_batch_size 4
```

测试结果如图2所示。

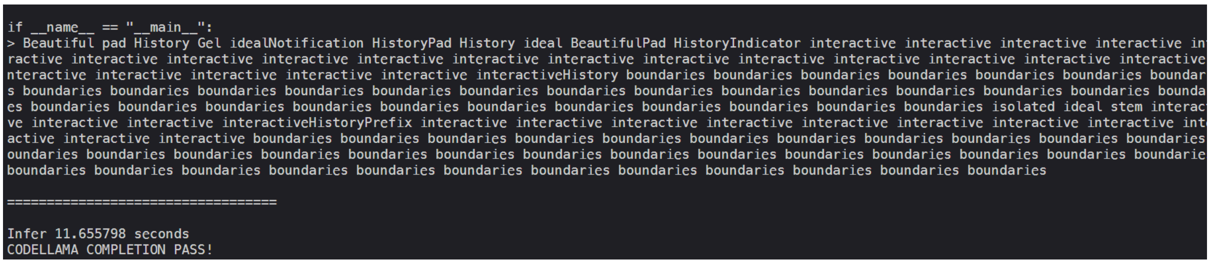

图 2: example_completion_mlu.py 测试结果

内容生成逻辑正确。个人猜测可能由于 generate.py 中，对应的文本推理方法中，默认参数没有调整，也没有重复惩罚措施，导致生成的文本出现重复内容。

4.5 补全 example_infilling_mlu.py

该文件使用 Llama 模型进行文本填充任务，通过加载模型分词器，处理包含占位符的提示语，然后使用 Llama 模型生成填充文本，该功能主要使用了 generate.py 文件中 Llama 类的 text_infilling 方法。

由于篇幅限制，补全详情见代码

运行9脚本后，能显示测试结果。

代码 9: run_completion.sh

```
torchrun --nproc_per_node 1 example_infilling_mlu.py \
    --ckpt_dir /workspace/model/favorite/large-scale-models/model-v1/CodeLlama-7b/ \
    --tokenizer_path /workspace/model/favorite/large-scale-models/model-v1/CodeLlama-7b/tokenizer.model \
    --max_seq_len 192 --max_batch_size 4
```

运行后结果如图3。

图 3: example_infilling_mlu.py 测试结果

4.6 补全 example_instructions_mlu.py

该文件使用 Llama 模型生成对话文本，同故宫加载模型和分词器，处理用户提供的指令，使用 Llama 模型生成相应的回复，并打印完整的对话内容，该功能主要使用了 generate.py 文件中 Llama 类的 chat_completion 方法。

由于篇幅限制，补全详情见代码

运行10脚本后，能显示测试结果。

代码 10: run_completion.sh

```
torchrun --nproc_per_node 1 example_instructions_mlu.py \
    --ckpt_dir /workspace/model/favorite/large-scale-models/model-v1/CodeLlama-7b-Instruct/ \
    --tokenizer_path /workspace/model/favorite/large-scale-models/model-v1/CodeLlama-7b-Instruct/tokenizer.model \
    --max_seq_len 512 --max_batch_size 4
```

运行后结果如图4。

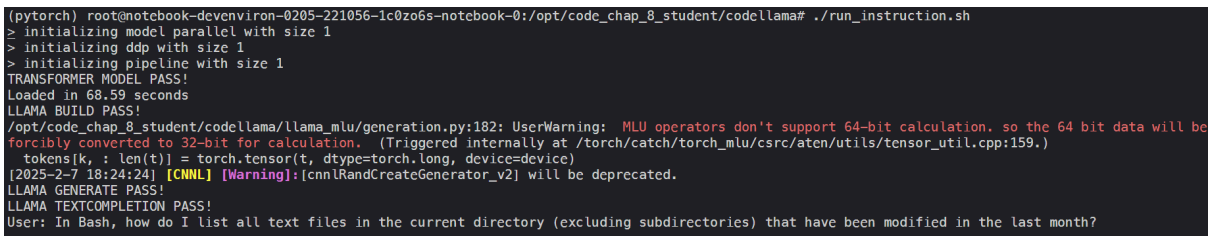

图 4: example_instructions_mlu.py 测试结果

可见，代码正确跑出结果, 但还是由于方法中的 temperature、top_p 等默认参数没有调整，或所给的模型缺乏重复惩罚措施，导致生成的文本出现重复内容。

给出希冀平台评测结果如图5。

transformer_codellama	llama_build	llama_generate	llama_textcompletion	codellama_completion	llama_textinfiling	codellama_infiling	llama_chatcompletion	codellama_instruction
PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS

图 5: 8-3 测试结果

可见，实验结果符合预期，实验完成。

5 实验总结

实验初期，面对复杂的实验任务和众多的代码文件，我感到有些不知所措。但通过仔细阅读代码，逐步理解代码的功能和结构，我逐渐了解了一些代码的作用和实现原理。这些都促使我不断思考，不断尝试，最终完成了实验任务。

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。例如模型生成的质量可能并不十全十美，我也意识到大模型的开发过程中的复杂性和困难性。因此，我会继续努力，不断学习，提高自己的技术水平，为未来的研究和实践打下坚实的基础。