



上海大学  
SHANGHAI UNIVERSITY

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

# 目录

— 实验 1 基于 Code Llama 实现代码生成	1
1    实验目的 . . . . .	1
2    实验环境 . . . . .	1
3    评分标准 . . . . .	1
4    实验内容及步骤 . . . . .	1
4.1    补全 llama_mlu/model.py . . . . .	1
4.2    补全 llama_mlu/generation.py . . . . .	1
4.3    补全 Codellama-inference.py . . . . .	11
4.4    补全 example_completion_mlu.py . . . . .	12
4.5    补全 example_infilling_mlu.py . . . . .	12
4.6    补全 example_instructions_mlu.py . . . . .	13
5    实验总结 . . . . .	14

# 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 Code 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 Code 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
```

```

3     def build(
4         ckpt_dir: str,
5         tokenizer_path: str,
6         max_seq_len: int,
7         max_batch_size: int,
8         model_parallel_size: Optional[int] = None,
9     ) -> "Llama":
10        if not torch.distributed.is_initialized():
11            if device == "mlu":
12                # TODO: 使用 MLU 设备初始化分布式进程组
13                torch.distributed.init_process_group("cnc1")
14            else:
15                torch.distributed.init_process_group("gloo")
16        if not model_parallel_is_initialized():
17            if model_parallel_size is None:
18                model_parallel_size = int(os.environ.get("WORLD_SIZE", 1))
19            initialize_model_parallel(model_parallel_size)
20
21        local_rank = int(os.environ.get("LOCAL_RANK", 0))
22        if device == "mlu":
23            # TODO: 如果设备为 MLU, 则设置当前进程的 MLU 设备
24            torch.mlu.set_device(local_rank)
25
26            # seed must be the same in all processes
27            torch.manual_seed(1)
28
29        if local_rank > 0:
30            sys.stdout = open(os.devnull, "w")
31
32        start_time = time.time()
33        checkpoints = sorted(Path(ckpt_dir).glob("*.pth"))
34        assert len(checkpoints) > 0, f"no checkpoint files found in {ckpt_dir}"
35        assert model_parallel_size == len(
36            checkpoints
37        ), f"Loading a checkpoint for MP={len(checkpoints)} but world size is {
38            model_parallel_size}"
39        ckpt_path = checkpoints[get_model_parallel_rank()]
39        # TODO: 加载模型的检查点文件, 并将模型加载到 CPU 上。
40        checkpoint = torch.load(ckpt_path, map_location="cpu")
41        with open(Path(ckpt_dir) / "params.json", "r") as f:
42            params = json.loads(f.read())
43
44        model_args: ModelArgs = ModelArgs(
45            max_seq_len=max_seq_len,
46            max_batch_size=max_batch_size,
47            **params,
48        )

```

```

49     # TODO: 调用 Tokenizer 函数
50     tokenizer = Tokenizer(tokenizer_path)
51     model_args.vocab_size = tokenizer.n_words
52     # support for mac
53     # print(device)
54     if device == "mlu":
55         torch.set_default_tensor_type(torch.HalfTensor)
56     else:
57         torch.set_default_tensor_type(torch.HalfTensor)
58     # TODO: 调用 Transformer 模型
59     model = Transformer(model_args)
60     # TODO: 加载模型的参数字典
61     model.load_state_dict(checkpoint, strict=False)
62     print("TRANSFORMER MODEL PASS!")
63     # add start
64     # print(device)
65     if device == "cpu":
66         model = model.float()
67
68     # add end
69     model.to(device)
70     print(f"Loaded in {time.time() - start_time:.2f} seconds")
71
72     print("LLAMA BUILD PASS!")
73     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

```

1  def generate(
2      self,
3      prompt_tokens: List[List[int]],
4      max_gen_len: int,
5      temperature: float = 0.6,
6      top_p: float = 0.9,
7      logprobs: bool = False,
8      echo: bool = False,
9      stop_token: Optional[int] = None,
10 ) -> Tuple[List[List[int]], Optional[List[List[float]]]]:
11     # print("调用了 generate")
12     if stop_token is None:
13         stop_token = self.tokenizer.eos_id
14     params = self.model.params
15     bsz = len(prompt_tokens)

```

```

16     assert bsz <= params.max_batch_size, (bsz, params.max_batch_size)
17     # TODO: 获取提示文本序列中最短的长度
18     min_prompt_len = min(len(t) for t in prompt_tokens)
19     # TODO: 获取提示文本序列中最长的长度
20     max_prompt_len = max(len(t) for t in prompt_tokens)
21     assert max_prompt_len <= params.max_seq_len
22     # TODO: 计算生成的总长度，需考虑提示文本和最大生成长度
23     total_len = max_prompt_len + max_gen_len
24     pad_id = self.tokenizer.pad_id
25     tokens = torch.full((bsz, total_len), pad_id, dtype=torch.long, device=
26                         device)
26     for k, t in enumerate(prompt_tokens):
27         # TODO: 将提示文本编码添加到张量中
28         tokens[k, : len(t)] = torch.tensor(t, dtype=torch.long, device=
29                                           device)
30     if logprobs:
31         # TODO: 创建一个与tokens张量具有相同形状的全零张量
32         token_logprobs = torch.zeros_like(tokens, dtype=torch.float)
33     prev_pos = 0
34     stop_reached = torch.tensor([False] * bsz, device=device).to(device)
35     input_text_mask = tokens != pad_id
36     # print("generate第189行")
37     # k = 1
38     # print(f"min_prompt_len={min_prompt_len},total_len={total_len}")
39     for cur_pos in range(min_prompt_len, total_len):
40         logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos)
41         # print(f"开始{k}次for循环")
42         if logprobs:
43             token_logprobs[:, prev_pos + 1 : cur_pos + 1] = -F.cross_entropy(
44                 (
45                     input=logits.transpose(1, 2),
46                     target=tokens[:, prev_pos + 1 : cur_pos + 1],
47                     reduction="none",
48                     ignore_index=pad_id,
49                 )
50             )
51         if temperature > 0:
52             # TODO: 对模型的输出进行 softmax 归一化，以得到每个可能的下一个
53             # token的概率分布，其中 temperature 用于控制模型输出的多样性
54             probs = F.softmax(logits[:, -1, :], dim=-1) / temperature, dim=-1)
55             # TODO: 根据概率分布采样出下一个 token
56             next_token = torch.multinomial(probs, num_samples=1)
57         else:
58             # TODO: 直接选择logits最大的位置作为下一个token，不进行随机采样
59             next_token = torch.argmax(logits[:, -1, :], dim=-1, keepdim=True
60                                     )
61             next_token = next_token.reshape(-1)
62             # only replace token if prompt has already been generated

```

```

58     next_token = torch.where(
59         input_text_mask[:, cur_pos], tokens[:, cur_pos], next_token
60     )
61     tokens[:, cur_pos] = next_token
62     stop_reached |= (~input_text_mask[:, cur_pos]) & (next_token ==
63         stop_token)
64     prev_pos = cur_pos
65     # print(f"经历了{k}次for循环")
66     # k += 1
67     if all(stop_reached):
68         break
69     # print("for循环结束了")
70     if logprobs:
71         # TODO: 将张量转换为列表格式
72         token_logprobs = token_logprobs.tolist()
73         out_tokens, out_logprobs = [], []
74         for i, toks in enumerate(tokens.tolist()):
75             # cut to max gen len
76             start = 0 if echo else len(prompt_tokens[i])
77             # TODO: 截取生成的标记序列，直到达到最大生成长度
78             toks = toks[start : start + max_gen_len]
79             probs = None
80             if logprobs:
81                 probs = token_logprobs[i][start : len(prompt_tokens[i]) +
82                                         max_gen_len]
83             # cut to stop token if present
84             if stop_token in toks:
85                 stop_idx = toks.index(stop_token)
86                 toks = toks[:stop_idx]
87                 probs = probs[:stop_idx] if logprobs else None
88             # TODO: 将截取后的标记序列添加到输出列表中
89             out_tokens.append(toks)
90             # TODO: 将截取后的log概率列表添加到输出列表中
91             if logprobs:
92                 out_logprobs.append(probs)
93     print("LLAMA GENERATE PASS!")
94     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

```

1     def text_completion(
2         self,

```

```

3     prompts: List[str],
4     temperature: float = 0.6,
5     top_p: float = 0.9,
6     max_gen_len: Optional[int] = None,
7     logprobs: bool = False,
8     echo: bool = False,
9 ) -> List[CompletionPrediction]:
10     if max_gen_len is None:
11         max_gen_len = self.model.params.max_seq_len - 1
12     prompt_tokens = [
13         self.tokenizer.encode(str(x), bos=True, eos=False) for x in prompts
14     ]
15     # TODO: 调用 generate 方法生成文本
16     generation_tokens, generation_logprobs = self.generate(
17         prompt_tokens, max_gen_len, temperature, top_p, logprobs, echo
18     )
19     if logprobs:
20         assert generation_logprobs is not None
21     return [
22         {
23             "generation": self.tokenizer.decode(t),
24             "tokens": [self.tokenizer.token_piece(x) for x in t],
25             "logprobs": logprobs_i,
26         }
27         for t, logprobs_i in zip(generation_tokens, generation_logprobs)
28     ]
29     print("LLAMA TEXTCOMPLETION PASS!")
30     return [{"generation": self.tokenizer.decode(t)} for t in
31             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**

```

1     def text_infilling(
2         self,
3         prefixes: List[str],
4         suffixes: List[str],
5         temperature: float = 0.6,
6         top_p: float = 0.9,
7         max_gen_len: Optional[int] = None,
8         logprobs: bool = False,
9         suffix_first: bool = False,
10    ) -> List[InfillingPrediction]:
11        assert self.tokenizer.eot_id is not None

```

```

12     # print("调用了text_infilling")
13
14     if max_gen_len is None:
15         max_gen_len = self.model.params.max_seq_len - 1
16
17     prompt_tokens = [
18         # 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
19         infilling_prompt_tokens(self.tokenizer, prefix, suffix, suffix_first
20             )
21         for prefix, suffix in zip(prefixes, suffixes)
22     ]
23
24     # TODO: 调用generate方法生成文本
25     generation_tokens, generation_logprobs = self.generate(
26         prompt_tokens,
27         max_gen_len,
28         temperature,
29         top_p,
30         logprobs,
31     )
32
33     generations = [self.tokenizer.decode_infilling(t) for t in
34         generation_tokens]
35     print("LLAMA TEXTINFILLING PASS!")
36
37     if logprobs:
38         assert generation_logprobs is not None
39
40         return [
41             {
42                 "generation": generation,
43                 "logprobs": logprobs_i,
44                 "tokens": [self.tokenizer.token_piece(x) for x in t],
45                 "full_text": prefix + generation + suffix,
46             }
47             for prefix, suffix, generation, t, logprobs_i in zip(
48                 prefixes,
49                 suffixes,
50                 generations,
51                 generation_tokens,
52                 generation_logprobs,
53             )
54         ]
55     else:
56         return [
57             {
58                 "generation": generation,
59                 "full_text": prefix + generation + suffix,
60             }
61             for prefix, suffix, generation in zip(prefixes, suffixes,
62                 generations)
63         ]

```

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

```
1      def chat_completion(
2          self,
3          dialogs: List[Dialog],
4          temperature: float = 0.6,
5          top_p: float = 0.9,
6          max_gen_len: Optional[int] = None,
7          logprobs: bool = False,
8      ) -> List[ChatPrediction]:
9          if self.tokenizer.step_id is not None:
10              ## 如果模型支持 step_id, 则使用另一种chat_completion的方法
11              return self._chat_completion_with_step_id(dialogs, temperature,
12                  top_p, max_gen_len, logprobs)
13          if max_gen_len is None:
14              max_gen_len = self.model.params.max_seq_len - 1
15          prompt_tokens = []
16          unsafe_requests = []
17          for dialog in dialogs:
18              unsafe_requests.append(
19                  any([tag in msg["content"] for tag in SPECIAL_TAGS for msg in
20                      dialog]))
21
22          if dialog[0]["role"] == "system":
23              dialog = [  # type: ignore
24                  {
25                      "role": dialog[1]["role"],
26                      "content": B_SYS
27                      + dialog[0]["content"]
28                      + E_SYS
29                      + dialog[1]["content"],
30                  }
31              ] + dialog[2:]
32          assert all([msg["role"] == "user" for msg in dialog[::2]]) and all(
33              [msg["role"] == "assistant" for msg in dialog[1::2]]
34          ), (
35              "model only supports 'system', 'user' and 'assistant' roles, "
36              "starting with 'system', then 'user' and alternating (u/a/u/a/u
37              ...
38          )
39          dialog_tokens: List[int] = sum(
40              [
41                  self.tokenizer.encode(
42                      f"{B_INST} {prompt['content'].strip()} {E_INST} {answer
43                      ['content'].strip()} ",
```

```

40                 bos=True,
41                 eos=True,
42             )
43             for prompt, answer in zip(
44                 dialog[::2],
45                 dialog[1::2],
46             )
47         ],
48     [],
49 )
50 assert (
51     dialog[-1]["role"] == "user"
52 ), f"Last message must be from user, got {dialog[-1]['role']}"
53 dialog_tokens += self.tokenizer.encode(
54     f"B_INST {dialog[-1]['content'].strip()} E_INST",
55     bos=True,
56     eos=False,
57 )
58 prompt_tokens.append(dialog_tokens)
59 # TODO: 调用 generate 方法生成文本
60 generation_tokens, generation_logprobs = self.generate(
61     prompt_tokens,
62     max_gen_len,
63     temperature,
64     top_p,
65     logprobs,
66 )
67 print("LLAMA CHATCOMPLETION PASS!")
68 if logprobs:
69     assert generation_logprobs is not None
70     return [
71         {
72             "generation": { # type: ignore
73                 "role": "assistant",
74                 "content": (
75                     self.tokenizer.decode(t) if not unsafe else
76                     UNSAFE_ERROR
77                 ),
78             },
79             "tokens": [self.tokenizer.token_piece(x) for x in t],
80             "logprobs": logprobs_i,
81         }
82         for t, logprobs_i, unsafe in zip(
83             generation_tokens, generation_logprobs, unsafe_requests
84         )
85     ]
86     return [

```

```

86         {
87             "generation": { # type: ignore
88                 "role": "assistant",
89                 "content": self.tokenizer.decode(t) if not unsafe else
90                     UNSAFE_ERROR,
91             }
92         for t, unsafe in zip(generation_tokens, unsafe_requests)
93     ]

```

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

```

1 def _chat_completion_turns(
2     self,
3     dialogs: List[Dialog],
4     temperature: float = 0.6,
5     top_p: float = 0.9,
6     max_gen_len: Optional[int] = None,
7     logprobs: bool = False,
8 ) -> List[ChatPrediction]:
9     if self.tokenizer.step_id is None:
10         raise RuntimeError("Model not suitable for chat_completion_step()")
11     if max_gen_len is None:
12         max_gen_len = self.model.params.max_seq_len - 1
13
14     prompt_tokens = []
15     unsafe_requests = []
16     for dialog in dialogs:
17         unsafe_requests.append(
18             any([tag in msg["content"] for tag in SPECIAL_TAGS for msg in
19                  dialog])
20         )
21
22         # Insert system message if not provided
23         if dialog[0]["role"] != "system":
24             dialog = [{"role": "system", "content": ""}] + dialog # type:
25                         ignore
26         # TODO: 调用函数将对话格式化为模型可处理的对话提示编码
27         dialog_tokens = dialog_prompt_tokens(self.tokenizer, dialog)
28         prompt_tokens.append(dialog_tokens)
29         # TODO: 调用 generate 方法生成文本
30         generation_tokens, generation_logprobs = self.generate(
31             prompt_tokens,
32             max_gen_len,
33         )

```

```

31         temperature,
32         top_p,
33         logprobs,
34     )
35     if logprobs:
36         assert generation_logprobs is not None
37         return [
38             {
39                 "generation": {
40                     "role": "assistant",
41                     "destination": "user",
42                     "content": (
43                         self.tokenizer.decode(t) if not unsafe else
44                         UNSAFE_ERROR
45                     ),
46                     "tokens": [self.tokenizer.token_piece(x) for x in t],
47                     "logprobs": logprobs_i,
48                 }
49                 for t, logprobs_i, unsafe in zip(
50                     generation_tokens, generation_logprobs, unsafe_requests
51                 )
52             ]
53         return [
54             {
55                 "generation": {
56                     "role": "assistant",
57                     "destination": "user",
58                     "content": self.tokenizer.decode(t) if not unsafe else
59                         UNSAFE_ERROR,
60                 }
61                 for t, unsafe in zip(generation_tokens, unsafe_requests)
62             ]

```

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

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

```
2 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。

```
(pytorch) root@notebook-devenvironment-0205-221056-1c0zo6s-notebook-0:/opt/code_chap_8_student/codellama# ./run-cll.sh
The model weights are not tied. Please use the `tie_weights` method before using the `infer_auto_device` function.
Loading checkpoint shards: 100%[=====] 2 / 2 [00:18<00:00,  9.50s/it]
/torch/venv3/pytorch/lib/python3.10/site-packages/transformers-4.34.0-py3.10.egg/transformers/generation/utils.py:2408: UserWarning: MLU operators don't
support 64-bit calculation, so the 64 bit data will be forcibly converted to 32-bit for calculation. (Triggered internally at /torch/catch/torch_mlu/cs
rc/aten/utils/tensor_util.cpp:159.)
    eos_token_id = torch.tensor(eos_token_id).to(input_ids.device) if eos_token_id is not None else None
time cost:4.00033562493324 s, output_token_num:75, total throughout:18.748436556189095 token/s
def remove_non_ascii(s: str) -> str:
    """
        return result
    Remove non-ascii characters from a string.
    result = ""
    for c in s:
        if ord(c) < 128:
            result += c
TRANSFORMER CODELLAMA PASS!
```

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

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

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

运行后结果如图3。

```
(pytorch) root@notebook-devenviron-0205-221056-1c0zo6s-notebook-0:/opt/code_chap_8_student/codellama# ./run_infilling.sh
> initializing model parallel with size 1
> initializing ddp with size 1
> initializing pipeline with size 1
TRANSFORMER MODEL PASS!
Loaded in 61.27 seconds
LLAMA BUILD PASS!
/opt/code_chap_8_student/codellama/llama_mlu/generation.py:182: UserWarning: MLU operators don't support 64-bit calculation. so the 64 bit data will be
forcibly converted to 32-bit for calculation. (Triggered internally at /torch/catch/torch_mlu/csrc/aten/utils/tensor_util.cpp:159.)
    tokens[k, : len(t)] = torch.tensor(t, dtype=torch.long, device=device)
LLAMA GENERATE PASS!
LLAMA TEXTINFILLING PASS!
```

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

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

运行后结果如图4。

```
(pytorch) root@notebook-devenviron-0205-221056-1c0zo6s-notebook-0:/opt/code_chap_8_student/codellama# ./run_instruction.sh
> initializing model parallel with size 1
> initializing ddp with size 1
> initializing pipeline with size 1
TRANSFORMER MODEL PASS!
Loaded in 68.59 seconds
LLAMA BUILD PASS!
/opt/code_chap_8_student/codellama/llama_mlu/generation.py:182: UserWarning: MLU operators don't support 64-bit calculation. so the 64 bit data will be
forcibly converted to 32-bit for calculation. (Triggered internally at /torch/catch/torch_mlu/csrc/aten/utils/tensor_util.cpp:159.)
    tokens[k, : len(t)] = torch.tensor(t, dtype=torch.long, device=device)
[2025-2-7 18:24:24] [CNNL] [Warning]: [cnnlRandCreateGenerator_v2] will be deprecated.
LLAMA GENERATE PASS!
LLAMA TEXTCOMPLETION PASS!
User: In Bash, how do I list all text files in the current directory (excluding subdirectories) that have been modified in the last month?
```

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_code_llama	llama_build	llama_generate_rate	llama_textcompletion	codellama_completion	llama_textfilling	codellama_infilling	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。