OLMo踩坑日记

环境搭建

1. `adduser` 添加个人账户，`nvidia-smi` 和 `nvcc -V` 查看 cuda 版本；
2. 网速较慢就：
    1. 添加代理服务器：`export https_proxy=https://代理服务器IP:端口` ，需要取消的话：`unset https_proxy`
    2. 配置清华镜像：pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
3. 安装 miniconda 配置环境，conda环境可以放在 mnt 里
    1. 下载 miniconda：curl -O https://repo.anaconda.com/miniconda/Miniconda3-latest-Linuxx86_64.sh、
    2. 更新环境：`source ~/.bashrc`
    3. 创建环境：`conda create -n OLMo python=3.10`
    4. 激活环境：`conda activate OLMo`
    5. 安装 pytorch：`conda install pytorch torchvision torchaudio pytorch-cuda=12.1 -c pytorch -c nvidia`
    6. 检查 pytorch 是否安装成功：

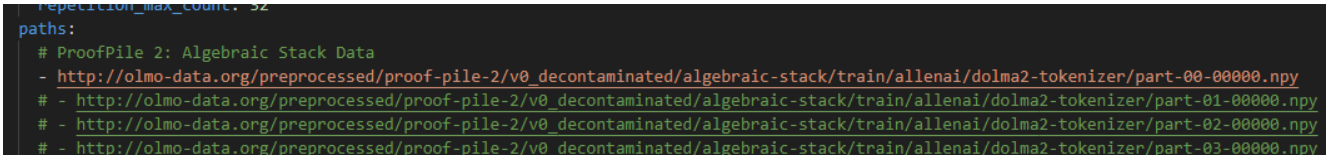
        ```
        import torch
        print(torch.__version__)
        ```

    7. 记录几个坑：
        - 出现缺少 nms 组件的时候，是 torchvision 版本出了问题，我的 requrement.txt 大致如下

```
torch==2.7.1
torchaudio==2.5.1
torchmetrics==1.7.3
torchvision==0.22.1
```

- 需要将 OLMo-1B.yaml 文件中的 `flash_attention` 设定为 `false`，否则后面运行会报错
- 训练的时候如果出现连接超时，记得关闭代理（翻墙软件）

4. 安装模型框架：`git clone https://github.com/allenai/OLMo.git`
5. 对 OLMo 文件夹添加权限：`sudo chmod 777 OLMo`
6. cd 到 OLMo 文件夹中配置环境 `pip install -e .[all]`
7. 配置文件设置
   1. 将 /OLMo/configs/official里的OLMo-1B.yaml配置文件复制一份，将最后本地数据配置的内容替换为数据盘里的部分数据。如果没有本地数据，就用 olmo 自带的。考虑到仅仅先跑通流程，可以先把除第一个之外的都注释掉，参考如下：



   2. 在OLMo文件目录下，新建Checkpoints、results文件夹

```
mkdir tmp
mkdir tmp/Checkpoints
mkdir tmp/results
export SCRATCH_DIR=./tmp
```

   3. 如果是 V100 显卡，`flash_attention: false`
   4. 考虑到显存不够用，对下面代码做注释或更改

```
# softmax_auxiliary_loss: true
# auxiliary_loss_multiplier: 1e-5
```

```
# fused_loss: true

# d_model: 2048
 d_model: 1024

# max_sequence_length: 4096
max_sequence_length: 2048

# global_train_batch_size: 512
global_train_batch_size: 4
# device_train_microbatch_size: 4
device_train_microbatch_size: 1
```

5. 设置存储文件夹为上述 Checkpoints 文件夹

```
save_folder: tmp/Checkpoints/OLMo-small/${run_name}
save_interval: 200
save_interval_ephemeral: 200
save_num_checkpoints_to_keep: -1
sharded_checkpointer: olmo_core

# save_interval_unsharded: null
save_interval_unsharded: 200 # 每两百个step保存一个合并模型
save_num_unsharded_checkpoints_to_keep: 3 # 保存最近三个合并模型
```

6. 设置加载文件夹为 Checkpoints 中的 latest 文件夹(如果是第一次训练，路径为 `null`)

```
load_path: tmp/Checkpoints/OLMo-small/OLMo2-1B-stage1/latest
# 第一次训练，这里是 null
load_path: null
```

8. wandb 设置

1. https://wandb.ai/home官网注册账户，记录下API key
2. train.py 中的98～101行注释掉

```
    # Maybe start W&B run.
    if cfg.wandb is not None and (get_global_rank() == 0 or not
cfg.wandb.rank_zero_only):
        wandb_dir = Path(cfg.save_folder) / "wandb"
        wandb_dir.mkdir(parents=True, exist_ok=True)
        wandb.init(
            dir=str(wandb_dir),
            project=cfg.wandb.project,
            # entity=cfg.wandb.entity,
            # group=cfg.wandb.group,
            # name=cfg.wandb.name,
            # tags=cfg.wandb.tags,
            config=cfg.asdict(exclude=["wandb"]),
        )
```

开始训练

1. cd 到 OLMo 文件夹中
2. 设置使用哪些显卡(第一次训练需要加 `export`)

```
export CUDA_VISIBLE_DEVICES=0,1,2,3
```

3. 开始训练

```
torchrun --nproc_per_node=4 --rdzv_endpoint=localhost:XXXX scripts/train.py
configs/official-0425/OLMo2-1B-stage1.yaml --save_overwrite
```

   这里需要注意，`--nproc_per_node=4` 代表使用的显卡数量，因为前面指定了 cuda:0,1,2,3 显卡可见，这里会将四张显卡全部用上。XXXX代表端口号，为避免与他人重复，需要额外设定，否则导致 wandb 连接失败，无法开始训练

4. 提示 wandb 时，选择新建账户，然后输入API key

问答测试脚本

```
import torch
import pathlib
import numpy as np
from transformers import AutoTokenizer
from omegaconf import OmegaConf

from hf_olmo.modeling_olmo import OLMoForCausalLM
from hf_olmo.configuration_olmo import OLMoConfig

# 避免 torch.load 出错
torch.serialization.add_safe_globals([
    pathlib.PosixPath,
    np.core.multiarray._reconstruct,
    np.ndarray,
    np.dtype,
    np.dtypes.UInt32DType,
])

def load_model_and_ckpt_unsharded():
    # 1. 加载 Qwen tokenizer
    tokenizer_path = "Qwen/Qwen2.5-0.5B"
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_path)

    # 2. 读取 YAML 配置
    yaml_path = "/mnt/zzb/workspace/three_data/train_codes/gj/OLMo/configs/official-0425/OLMo2-1B-stage1.yaml"
    cfg = OmegaConf.load(yaml_path)

    # 3. 计算 mlp_hidden_size 并构造 config
    cfg.model["mlp_hidden_size"] = cfg.model.d_model * cfg.model.mlp_ratio
    model_config_dict = OmegaConf.to_container(cfg.model, resolve=True)
    config = OLMoConfig.from_dict(model_config_dict)
```

```
    # 4. 初始化模型结构
    model = OLMoForCausalLM(config)

    # 5. 直接加载 unsharded checkpoint 文件（修改路径为你的实际路径）
    ckpt_path =
"/mnt/zzb/workspace/three_data/train_codes/gj/OLMo/tmp/Checkpoints/Qwen/OLMo2-
1B-stage1/step13600-unsharded/model.pt"
    state_dict = torch.load(ckpt_path, map_location="cpu")
    if "model" in state_dict:
        state_dict = state_dict["model"]

    # 6. 如果缺失 model. 前缀，加上
    if not any(k.startswith("model.") for k in state_dict.keys()):
        state_dict = {"model." + k: v for k, v in state_dict.items()}

    # 7. 加载权重到模型
    model.load_state_dict(state_dict, strict=False)
    model.eval()
    model.to("cuda")

    return model, tokenizer

def generate_reply(model, tokenizer, prompt, max_new_tokens=50):
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=1024)
    if "token_type_ids" in inputs:
        del inputs["token_type_ids"]
    inputs = {k: v.to("cuda") for k, v in inputs.items()}

    with torch.no_grad():
        output_ids = model.generate(
            **inputs,
            do_sample=True,
            temperature=0.2,
```

```
            top_p=0.9,
            repetition_penalty=1.1,
            max_new_tokens=max_new_tokens,
            eos_token_id=tokenizer.eos_token_id,
            pad_token_id=tokenizer.pad_token_id,
            no_repeat_ngram_size=2
        )

    full_text = tokenizer.decode(output_ids[0], skip_special_tokens=True)

    # 去除重复 prompt
    if full_text.startswith(prompt):
        full_text = full_text[len(prompt):].strip()

    # 如果第一个字符是标点，去掉它
    if full_text and full_text[0] in ['，', '。', '！', '？', ',', '.', '!', '?']:
        full_text = full_text[1:].strip()

    # 截断到第一个终止标点
    END_TOKENS = ['。', '！', '？', '.', '!', '?']
    for token in END_TOKENS:
        if token in full_text:
            idx = full_text.index(token)
            return full_text[:idx+1].strip()

    return full_text[:max_new_tokens].strip()

def interactive_loop(model, tokenizer):
    print("🔁 进入 OLMo 对话模式，输入 q 退出。")
    while True:
        user_input = input("\n👴 • 💻 你：")
        if user_input.lower().strip() in {"q", "quit", "exit"}:
            print("👋 再见！")
            break
```

```
        reply = generate_reply(model, tokenizer, user_input)
        print(f"\n🤖 OLMo: {reply}")

if __name__ == "__main__":
    print("🚀 正在加载模型和 tokenizer，请稍候...")
    model, tokenizer = load_model_and_ckpt_unsharded()
    interactive_loop(model, tokenizer)
```

dolma 安装

新建一个 dolma 的环境

`pip install dolma`

这个可能会很慢，如果安装有问题，可以先安装 rust

```
# 安装 Rust 工具链
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

# 加载环境变量
source $HOME/.cargo/env
```

然后验证：

```
rustc --version
cargo --version
```

如果出现 `punkt` 报错，删除它并重新下载 `punkt` 分词器

```
rm -f /home/你的账户/nltk_data/tokenizers/punkt.zip
python -c "import nltk; nltk.download('punkt')"
```

验证

```
(Dolma) gj@iZ1pp01libkysegflqf3x7Z:~/dolma$ dolma --help
usage: dolma {global options} [command] {command options}

Command line interface for the Dolma processing toolkit

positional arguments:
  {dedupe,mix,tag,list,stat,tokens,warc}
    dedupe              Deduplicate documents or paragraphs using a bloom filter.
    mix                 Mix documents from multiple streams.
    tag                 Tag documents or spans of documents using one or more taggers.
                        For a list of available taggers, run `dolma list`.
    list                List available taggers.
    stat                Analyze the distribution of attributes values in a dataset.
    tokens              Tokenize documents using the provided tokenizer.
    warc                Extract documents from WARC files and parse HTML out.

options:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        Path to configuration optional file
```

dolma 例子

获取维基百科的数据

1. 首先安装依赖

```
pip install git+https://github.com/santhoshtr/wikiextractor.git requests smart_open tqdm
```

2. 下载维基百科某天的数据(设置比较近日子的数据)

```
python scripts/make_wikipedia.py \
  --output wikipedia \
  --date 20250601 \
  --lang simple \
  --processes 16
```

3. 运行标记器(每个标记的意思可在第6步了解)

```
dolma tag \
    --documents "wikipedia/v0/documents/*.gz" \
    --experiment exp \
    --taggers random_number_v1 \
            cld2_en_paragraph_with_doc_score_v2 \
            char_length_with_paragraphs_v1 \
            whitespace_tokenizer_with_paragraphs_v1 \
    --processes 16
```

这里如果报错,大概率是需要安装 `pycld2`

```
pip install pycld2
```

4. 去重 标记后,在段落级别对数据集进行重复数据删除

```
dolma dedupe \
    --documents "wikipedia/v0/documents/*" \
    --dedupe.paragraphs.attribute_name 'bff_duplicate_paragraph_spans' \
    --dedupe.skip_empty \
    --bloom_filter.file tmp/deduper_bloom_filter.bin \
    --no-bloom_filter.read_only \
    --bloom_filter.estimated_doc_count '6_000_000' \
    --bloom_filter.desired_false_positive_rate '0.0001' \
    --processes 188
```

参数	说明
`dolma dedupe`	启动 deduplication 去重模块，对文档内容进行重复检测。
`--documents "wikipedia/v0/documents/*"`	处理所有路径下的 `.gz` 文档。可以是多个 JSONL（Gzip）文件，格式如：每行一个文档对象。
`--dedupe.paragraphs.attribute_name 'bff_duplicate_paragraph_spans'`	结果将写入 `attributes.bff_duplicate_paragraph_spans` 字段，记录哪些段落是重复的。
`--dedupe.skip_empty`	跳过空段落或无效文档（如纯空格段）。提升处理效率。
`--bloom_filter.file tmp/deduper_bloom_filter.bin`	布隆过滤器数据将存储到这个路径，后续可复用。布隆过滤器是一种高效检测重复的哈希结构。
`--no-bloom_filter.read_only`	表示**允许写入**布隆过滤器（默认是只读），此选项用于构建或更新布隆过滤器内容。
`--bloom_filter.estimated_doc_count '6_000_000'`	预计去重处理的文档量，用于预估布隆过滤器大小和结构。影响内存使用和准确率。
`--bloom_filter.desired_false_positive_rate '0.0001'`	设置期望的误判率为 0.01%（即允许 1 万个中最多有 1 个误判为重复）。数值越低，准确性越高，内存也会占用更多。
`--processes 188`	并行进程数，这里为 188，意味着会并行开启 188 个进程加速处理。建议根据你机器的 CPU 核数和内存设置此值。

查看重复文档的内容

```
zcat wikipedia/v0/attributes/bff_duplicate_paragraph_spans/wiki_00.gz | head -n 5
# 文档 id=1：没有重复段落（即干净的文档）
{"attributes":{"bff_duplicate_paragraph_spans":[]},"id":"1"}
# 文档 id=2：段落 397 ~ 408 被认为是重复段落，flag = 1：表示这些段落是被判定为重复的
{"attributes":{"bff_duplicate_paragraph_spans":[[397,408,1]]},"id":"2"}
{"attributes":{"bff_duplicate_paragraph_spans":[]},"id":"6"}
{"attributes":{"bff_duplicate_paragraph_spans":[[1836,1848,1],
[1848,1892,1]]},"id":"8"}
{"attributes":{"bff_duplicate_paragraph_spans":[[2973,2985,1],
[2985,3029,1]]},"id":"9"}
```

5. 运行 Mixer 运行标记器并标记哪些段落是重复的之后，我们可以运行混合器来创建包含语言和文档子集的数据集，要开始清洗数据了（这里在前面加了个doc/是因为 `wikipedia-mixer.json` 文件在这里）

```
dolma -c docs/examples/wikipedia-mixer.json mix --processes 16
```

查看 `wikipedia-mixer.json` 文件，这里面是混合器 Mixer 对数据集处理的配置信息

首先看顶层结构

```
{
  "streams": [ ... ], // "streams"：处理数据的任务流数组（可以配置多个任务）。
  "processes": 1 // "processes": 1：使用 1 个进程处理任务。你可以改为更多以加速。
}
```

streams[0] 配置项含义（主任务）

```
"name": "getting-started" //给这个数据处理任务起个名字(任意字符串)。
"documents": ["wikipedia/v0/documents/*.gz"] //输入数据路径:读取路径中所有 .gz 格式的 Wikipedia 文档(JSON Lines 格式)。是上游如 make_wikipedia.py 或 dolma tag 生成的结果。
"output": {
 "path": "wikipedia/example0/documents",
 "max_size_in_bytes": 1000000000
} // 输出路径:保存清洗后的文档(例如 .gz 格式的 JSONL 文件)到 wikipedia/example0/documents/。每个输出文件最大不超过 1GB(单位:字节)。
"attributes": [
 "exp",
 "bff_duplicate_paragraph_spans"
] //声明你在文档中会使用哪些属性字段。"exp" 是你在 tag 时使用 --experiment exp 时生成的标签。"bff_duplicate_paragraph_spans" 是 dedupe 去重时生成的字段。
```

`"filter"` 筛选条件

```
"filter": {
  // 只保留满足以下条件的内容
 "include": [
   // 文档总 token 数少于 100,000 才保留。
  "$.attributes[?(@.exp__whitespace_tokenizer_with_paragraphs_v1__document[0][2]<100000)]"
 ],
  // 以下文档被丢弃
 "exclude": [
   // token 数太少(<50)的文档删掉。
  "$.attributes[?(@.exp__whitespace_tokenizer_with_paragraphs_v1__document[0][2]<50)]",
   // 英文段落置信度低于 0.5 的文档删掉(说明可能不是英语内容)。
  "$.attributes[?(@.exp__ft_lang_id_en_paragraph_with_doc_score_v2__doc_en[0][2]<=0.5)]",
   // 有重复段落(比如去重后标记为 [..., ..., 1.0])的文档删掉。
```

```
    "$@.attributes[?(@.bff_duplicate_paragraph_spans &&
@.bff_duplicate_paragraph_spans[0] && @.bff_duplicate_paragraph_spans[0][2] >=
1.0)]"
  ]
}
```

`"span_replacement"`（可选清洗：替换非英文段落）

```
{
  // 找到属性中标记为“不是英文”的段落（CLD2 检测）。
  "span": "$.attributes.exp__cld2_en_paragraph_with_doc_score_v2__not_en",
  // 如果 score > 0.1，就用空字符串替换掉（相当于删掉）。
  "min_score": 0.1,
  "replacement": ""
}
```

从 Wikipedia 抽取后的文档中：

- 保留长度适中（token 在 50~100000）的；
- **去掉**非英文内容、重复段落；
- 清理掉低质量段落；
- 最终保存到指定目录，每个文件最大 1GB。

6. 查看清洗后的数据 📜 **顶层字段解析：**

```
{
 "id": "1", // 文档唯一 ID。
 "source": "wikipedia", // 数据来源标签。
 "text": "...", // 文档原始文本，按段落排列。
 "attributes": {...}, // Taggers（标签器）和 Deduper（去重器）生成的结构化数据，是 dolma 清洗/筛选的核心依据。
 "metadata": {...}, // 文档的元信息，如来源文件、版本、URL、修订号。
 "created": "2025-06-01T00:00:00.000Z", // 创建时间戳。
 "added": "2025-06-19T13:03:20.457Z", // 混入（mixer）数据流的时间戳
 "version": "v0"
}
```

🧠 **核心部分：attributes 字段** 这个字段由多个 tagger 模块生成的“属性”组成，键名遵循：

```
exp__{tagger_name}__{scope}
```

例如：

```
exp__whitespace_tokenizer_with_paragraphs_v1__document
[[0,2843,609.0]] // 文档总共 2843 个字符，包含 609 个 token（按空格分词统计）。

exp__cld2_en_paragraph_with_doc_score_v2__doc_en
[[0,2843,0.98311]] // 文档整体英文概率为 98.31%。

exp__cld2_en_paragraph_with_doc_score_v2__not_en
[[0,6,1.0], [7,166,0.01], ...] // 每个段落被判断为非英文的概率。第一个段落（长度 6 个字符）几乎可以确定不是英文（1.0）。后续段落大多为英文（< 0.01）。

 exp__char_length_with_paragraphs_v1__paragraph
 [[0,6,6.0], [7,166,159.0], ...] // 每段文字的字符长度。例如第二段长度是 159 个字符。

exp__random_number_v1__random
```

```
[[0,2843,0.47887]] // 为每个文档打上一个伪随机浮点数(用于抽样、排序等)。

bff_duplicate_paragraph_spans
[] // 表示该文档没有重复段落(由 dolma dedupe 的 Bloom Filter 模块判断)如果存在类似 [[start, end, score]] 的条目,表示对应字符范围内段落有重复。
```

🔗 metadata 字段

```
"metadata": {
  "length": 599, // 表示总 token 数或字节数(用于判断文档大小)。
  "provenance": "wiki_00.gz:1", // 来源文件名及行号(即来源于 wiki_00.gz 的第 1 条记录)。
  "revid": "396686", // Wikipedia 的修订 ID。
  "url": "https://simple.wikipedia.org/wiki?curid=1" // 对应 Wikipedia 原文链接。
}
```

🧩 text 字段:原始内容

这是 Wikipedia 文档的实际正文内容。

分段内容已经通过 dolma 的段落划分器处理好了,可以直接用来训练语言模型、分类器或过滤无意义段。

✅ 总结

模块	使用字段
数据过滤器(convert)	`attributes`, `id`, `text`
去重器(deduper)	`bff_duplicate_paragraph_spans`, `id`
多语言筛选	`cld2`、`ft_lang_id` 类属性
清洗器(span_replacement)	`not_en`,替换非英文段落
数据统计/采样	`random_number`, `token`、`char_length`

7. 标记数据集 使用命令对数据集进行分词 `tokens` 。在本例中,我们使用 EleutherAI 优秀的GPT Neo-X 20B分词器。

```
dolma tokens \
    --documents "wikipedia/example0/documents/*.gz" \
    --tokenizer.name_or_path "EleutherAI/gpt-neox-20b" \
    --tokenizer.eos_token_id 1 \
    --destination wikipedia/example0/tokens \
    --processes 16
```

`eos_token_id`：表示“End of Sequence”token 的 ID，用于表示每个文档的结束。必须显式提供，告诉模型，这段话结束了。

`bos_token_id`：可选，表示文档的开始（Begin Of Sequence）。大多数模型并不要求它。

分完词后，可以在 `wikipedia/example0/tokens` 目录下看到两个文件，一个 `.csv.gz` 和一个 `.npy`：

文件类型	作用	查看方式
`.csv.gz`	每个文档在 `.npy` 中的位置和元数据	`pandas.read_csv`
`.npy`	所有 tokens 序列，拼成一个数组	`numpy.load`

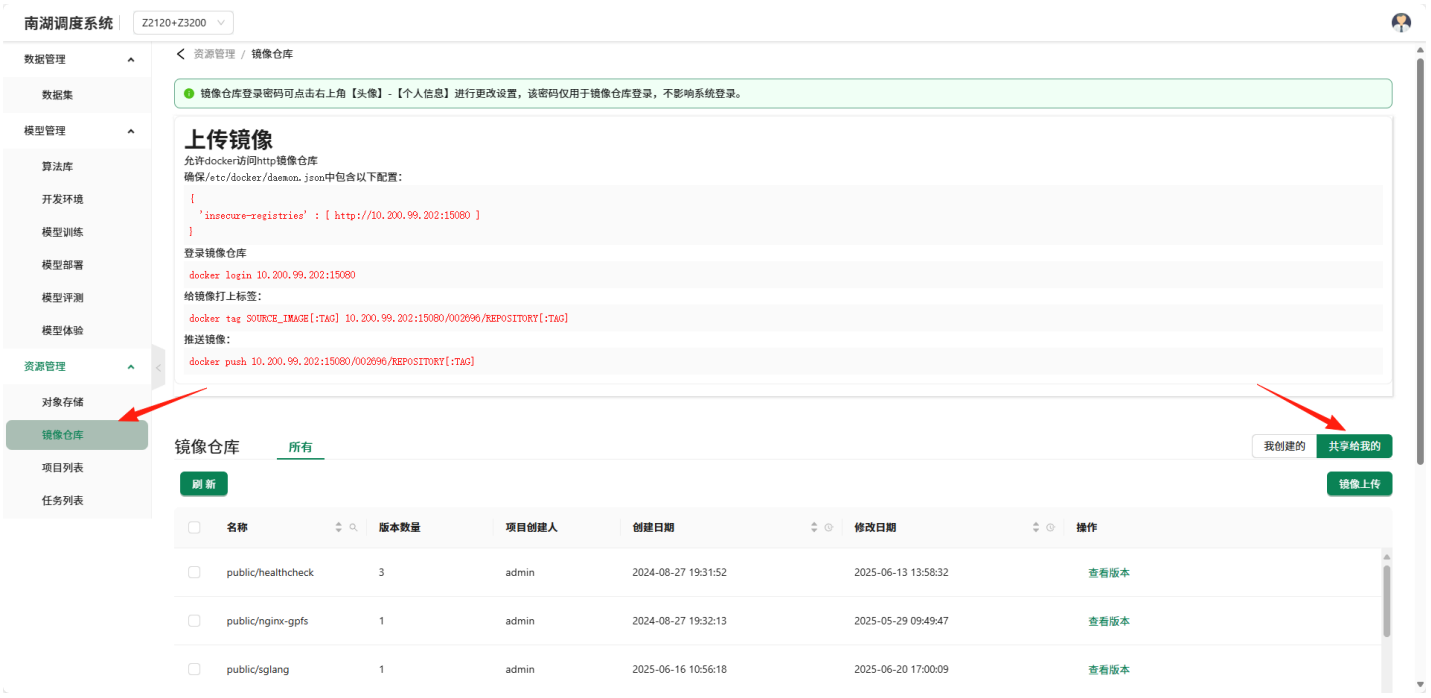
8. 查看清洗好的数据

```
import numpy as np
from transformers import AutoTokenizer

# 加载tokenizer
tokenizer = AutoTokenizer.from_pretrained("EleutherAI/gpt-neox-20b")

# 读取token ID流（假设之前已经用 np.fromfile 读入了 tokens）
tokens = np.fromfile("wikipedia/example0/tokens/part-0-00000.npy", dtype=np.uint16)

# 把token ID列表转换成文本
text = tokenizer.decode(tokens.tolist(), clean_up_tokenization_spaces=False)
print(text[0:100]) # 输出前101个字符
```

拉取并修改镜像

拉取基础镜像

> 为了方便，直接拉取南湖调度平台的公共镜像



在共享给我栏目中挑选合适的镜像，查看版本并复制镜像地址

在 vscode 里登录账号

```
docker login 10.200.99.202:15080
```

拉取一个公共镜像，给镜像打标签（相当于改了个名字）

```
docker pull 10.200.99.202:15080/public/pytorch:2.5.0-cuda12.1-cudnn9-vscode

docker tag 10.200.99.202:15080/public/pytorch:2.5.0-cuda12.1-cudnn9-vscode
10.200.99.202:15080/002696/olmo-gj:v1.0
```

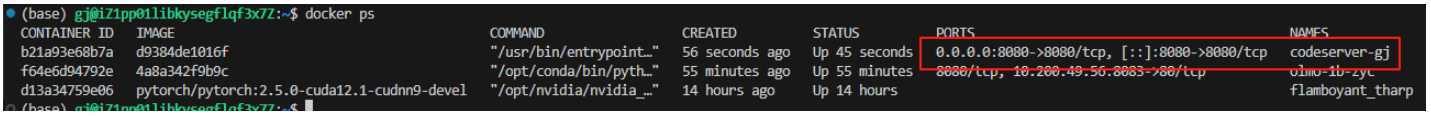
这里我们拉取的是有 vscode web-server 的镜像，这种镜像相对于基础镜像而言，可以用 web 端的vscode 对远程容器进行调试，使用更方便。此外，这个镜像是被拉取到本地的，需要我们对它进行加工，安装 olmo 必要的环境并测试能否运行

启动镜像并进入容器

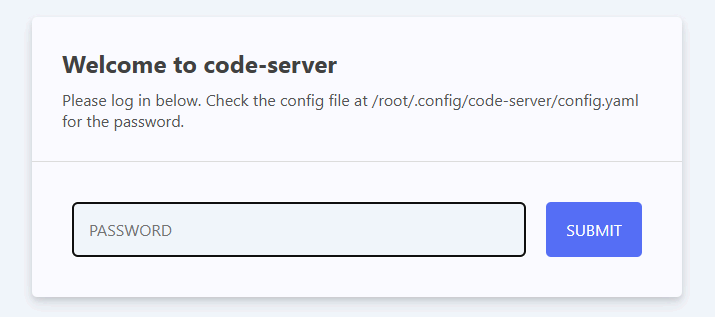
因为拉取的公共镜像需要先在本地进行修改和调试，然后保存修改并打包传送到集群，所以我们需要先在本地启动它。相对于普通镜像直接 `docker run -it 镜像名字 /bin/bash` ，有vscode web-server的镜像需要将容器端口映射到本地才能进行连接。**此外，记住还要挂载自己的工作区！**

```
docker run -d \
  --gpus all \
  -p 8080:8080 \
  --name codeserver-gj \
  -v /home/gj:/home/coder \
  -v /mnt/:/mnt \
  d9384de1016f
```

`docker ps` 查看当前运行着的容器



可以看到容器成功启动了，接下来我们可以通过浏览器输入地址 `localhost:8080` 进入到该容器



这里需要我们输入密码，密码存放在 `/root/.config/code-server/config.yaml` 文件中，所以我们得先进到容器中将密码修改成我们自己的

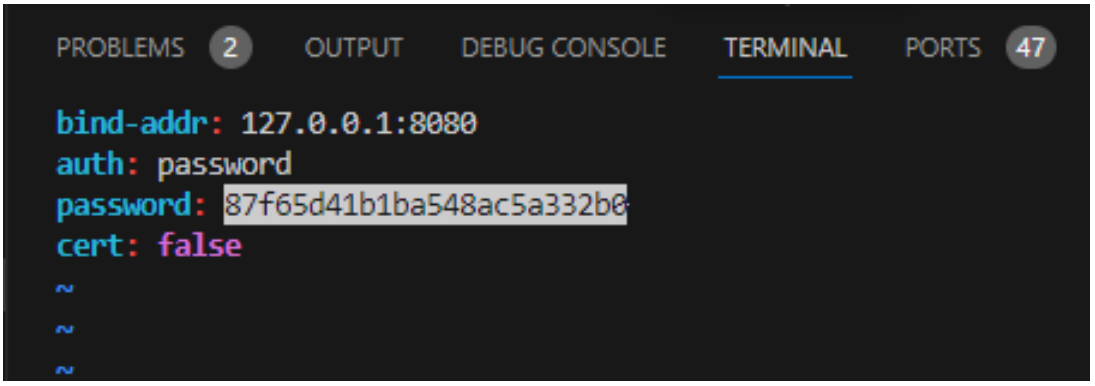
回到本地 vscode，进入到正在运行的容器中

```
docker exec -it b21a93e68b7a /bin/bash
```

打开存放密码的 `yaml` 文件

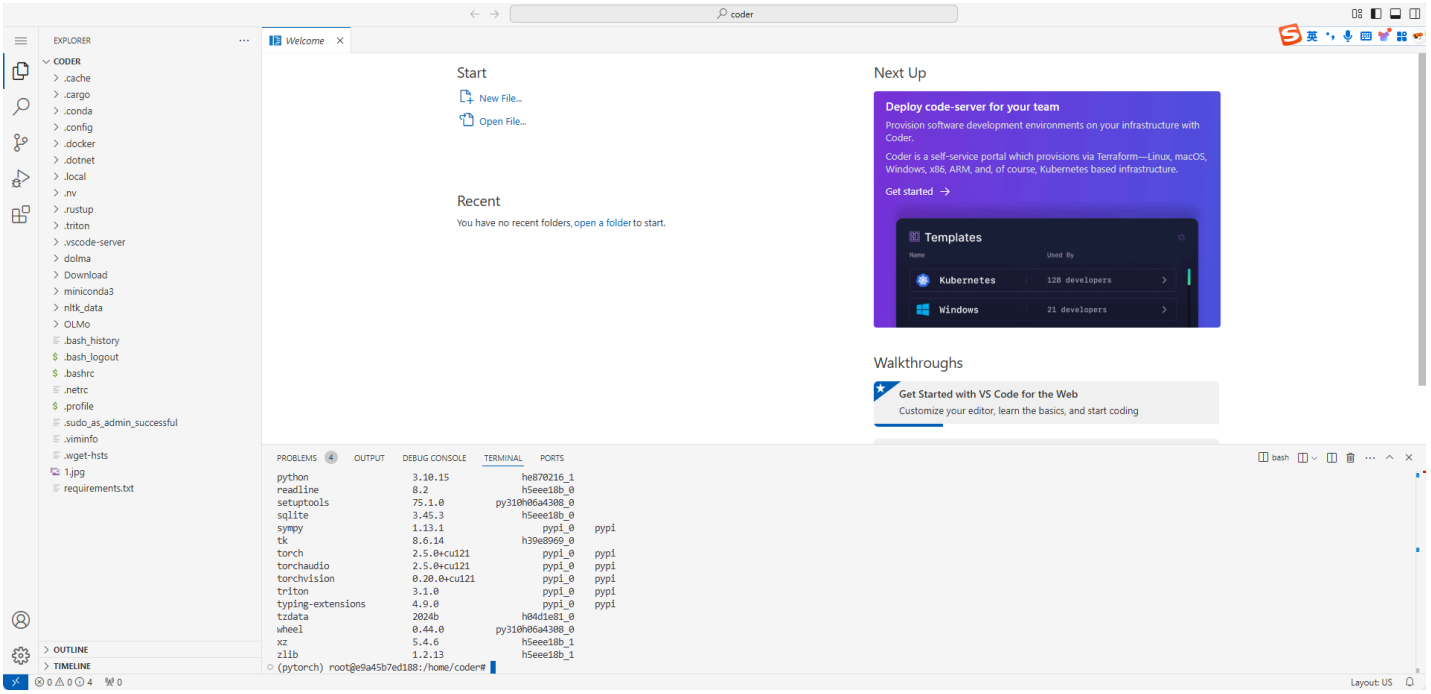
```
vi /root/.config/code-server/config.yaml
```

可以看到当前密码



将改密码设置为自己喜欢的后，`:wq` 保存并退出（这里用的vi编辑器，不会的可以搜下基本指令）

`exit` 退出容器并 `docker restart 容器名`，回到浏览器，输入修改后的密码即可登录



通过这种方式登录容器，可以像使用本地vscode一样对远程服务器进行代码调试，并且更容易传输文件，查看图片（直接拖拉拽）。

安装依赖并打包镜像

`pip list` 查看当前环境，`nvcc -V` 查看cuda版本（可能没有），配置清华源 `pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple` ，查看conda `conda --version` 和 `conda env list`

我这边能看到有个pytorch的环境，而我们进入容器的时候就已经在这个环境中了，因此不需要额外激活，可以直接安装olmo。这里由于我把本地OLMo的项目挂载进来了，就不需要在容器中再放一个OLMo项目。

在容器中pip依赖

```
cd OLMo
pip install -e .[all]
```

容器的环境是装在 `/opt/miniconda3/envs` 里的，而非挂载的文件夹中，因此实现了环境与脚本&数据的分离。如果安装中报错，可能是torchvision的版本不对，`pip install torchvision==0.22`

试着跑一下OLMo，先 `nvidia-smi` 找一下gpu，能找到再

```
export CUDA_VISIBLE_DEVICES=0,1,2,3
torchrun --nproc_per_node=4 --rdzv_endpoint=localhost:29600 scripts/train.py
configs/official-0425/OLMo2-1B-stage1.yaml --save_overwrite
```

能跑通OLMo，就commit容器保存为新的镜像

```
docker commit codeserver-gj olmo-images-gj:latest
```

退出刚装好依赖的容器，从刚建好的镜像创建新的容器，查看镜像是否修改好了

```
docker run -d \
  --gpus all \
  -p 8080:8080 \
  --name olmo-gj \
  -v /home/gj:/home/coder \
  -v /mnt/:/mnt \
  50764bf7c1a2
```

cd到OLMo文件夹中跑一下，能跑起来就说明镜像没问题

将镜像上传到调度中心

```
# 登录账号（账户是你工号，初始密码为Zjlab123，想改密码需要先登录南湖调度中心）
# 调度中心地址：https://zjic.zhejianglab.com/portal/navigation
docker login 10.200.99.202:15080 #
# 打标签
docker tag olmo-images-gj:latest 10.200.99.202:15080/002696/olmo-for-group2:latest
# 上传镜像至个人镜像仓库
docker push 10.200.99.202:15080/002696/olmo-for-group2:latest
```

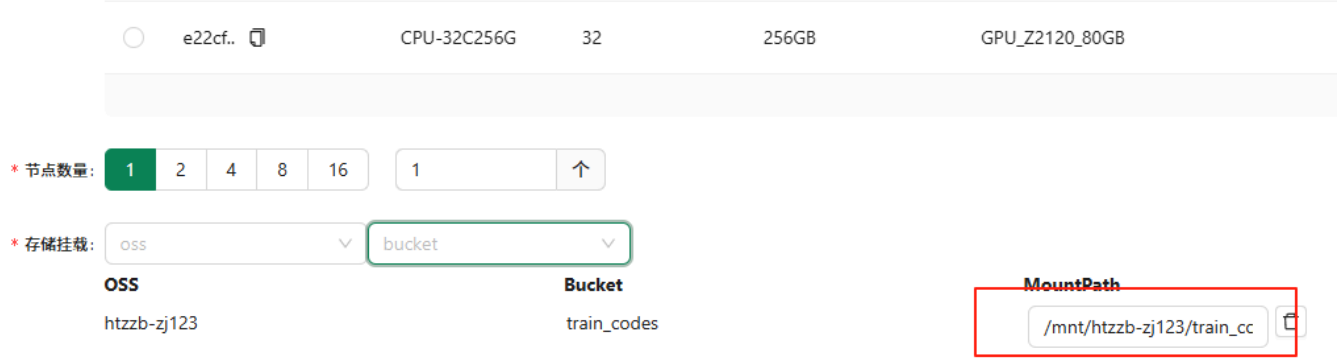
这里需要说明，10.200.99.202:15080是服务器的ip和端口，需要更换的是002696这个工号（换成你自己的，这里和后面的gj需要换成你自己的名字）。注：如果登录出了问题，检查一下是不是要在 `docker login` 前加 `sudo`，可以输入 `groups` 查看自己所在组，如果输出 `你的用户名 docker sudo`，则不需要加 `sudo`

> 镜像名一律改成这种格式方能推送 `10.200.99.202:15080/002696/olmo-for-group2:latest`
>
> olmo-for-group2:latest是你上传到个人镜像仓库后的名称：tag

调度中心操作：

1. 因为前面已经推送了镜像到个人仓库，这里点击**模型管理⇒开发环境⇒新建**后就能看到自己的上传的镜像。选择该镜像后，填写vscode密码，供之后登录vscode的web-server使用

2. 挂载数据以供程序脚本训练。你要找到自己的OLMo文件夹和数据集，因此将开发机上 `/home/你用户名` 下的数据放到 `/mnt/你的挂载路径` 下，然后这里会把你的数据挂载到集群上



3. 首次创建开发环境需要申请，申请好之后就可以拉起任务。点击右边的vscode可以直接登录到web-server端

4. cd到你挂载路径下的OLMo文件夹中开始训练

训练开始前的操作

因为我们的OLMo是通过 `pip install -e .[all]` 安装的，有个olmo的包不在env路径下，所以需要将OLMo文件夹路径添加到环境变量中（**这一步在南湖平台的容器里完成**）

```
export PYTHONPATH=/mnt/htzzb-zj123/train_codes/gj/OLMo:$PYTHONPATH
```

在本地vscode或Xshell进入到挂载的个人OLMo文件夹(如果自己的文件夹在/home里,建议拷贝到挂载目录/train_codes中,并 `chmod -R 777 gj` ,**这一步在本地做,因为南湖平台的容器权限不够**)

```
cd /mnt/htzzb-zj123/train_codes/gj/OLMo
```

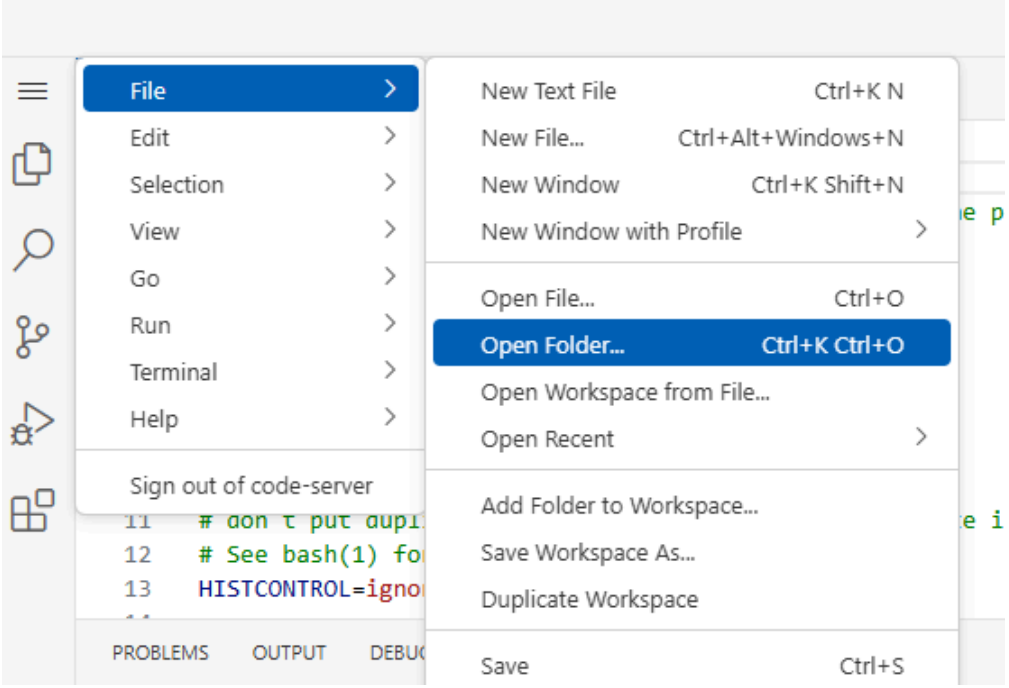
设置wandb重新登录(**这里一定要设置,否则日志会自动传到我的wandb记录里**)

```
wandb login --relogin
```

`nvidia-smi` 查看gpu,没问题就可以跑训练了

> 调度中心申请算力资源,需要选择有gpu版本的服务器,任务运行后点击 VS Code 打开网页
>
> 在输入vscode密码的时候,可以考虑给一个ssh密码,以供ssh登录的方式

如果想在浏览器的 vscode 中打开个人 OLMo 文件夹,如下所示



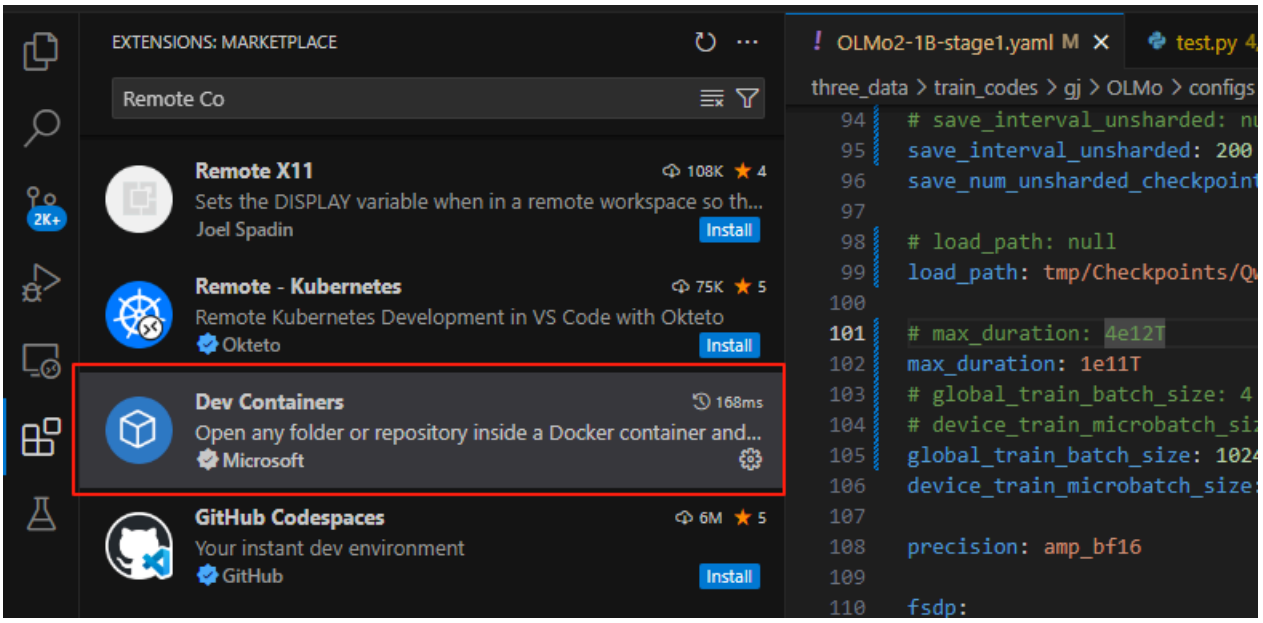
模型微调

LLaMA-Factory

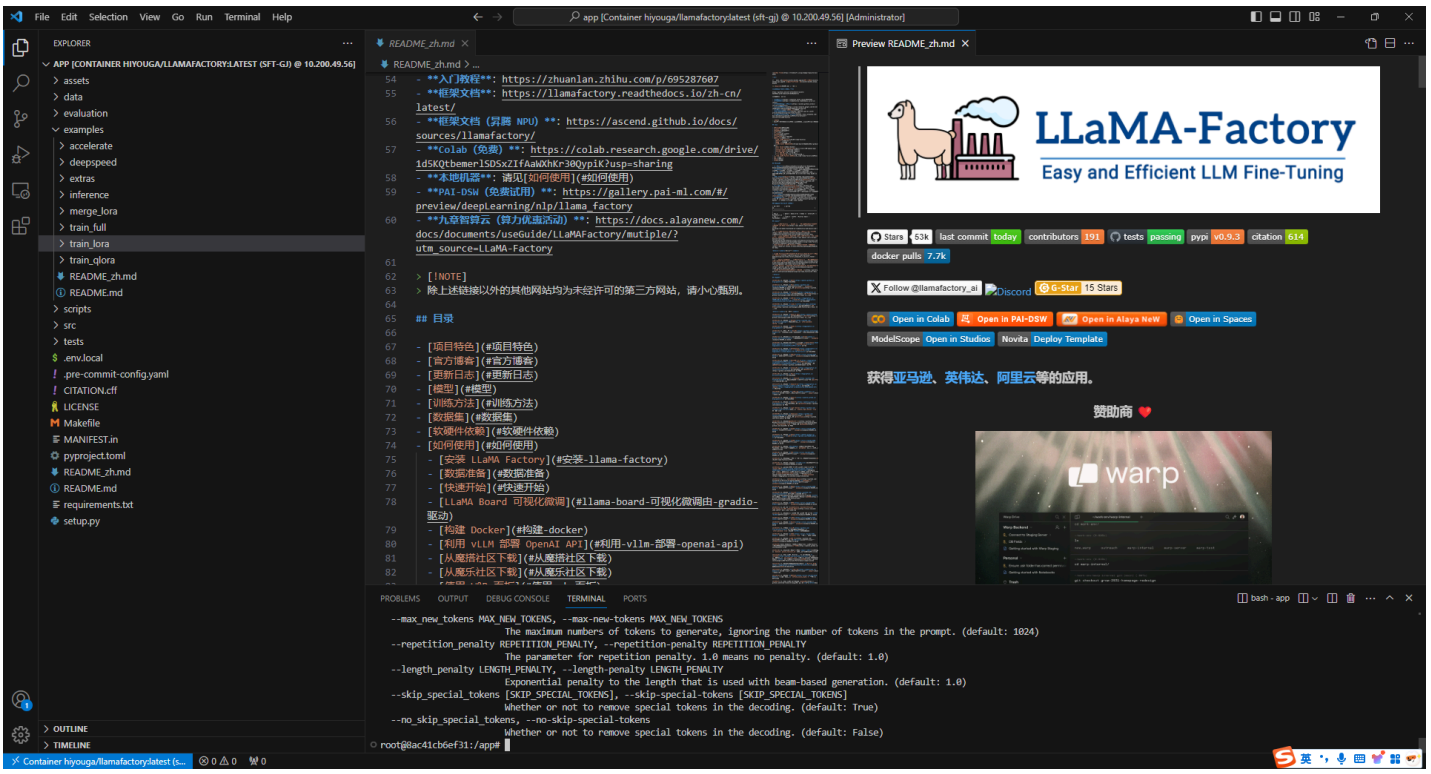
拉取官方镜像文件

```
docker run -it --gpus=all -v /mnt:/mnt --name sft-gj --ipc=host hiyouga/llamafactory:latest
```

拉取完之后就已经启动并进入到容器内部了，退出容器，在本地 vscode 扩展程序中安装 Dev Containers，然后 ctrl+shift+p，搜 `Remote-Containers: Attach to Running Container...` 进入到容器内部



由于挂载的/mnt里已经存放好了我们的脚本和预训练数据，所以我们成功偷懒不需要配环境了



微调简介

LLaMA-Factory 的微调指令很简洁，就三行

```
llamafactory-cli train examples/train_lora/llama3_lora_sft.yaml
llamafactory-cli chat examples/inference/llama3_lora_sft.yaml
llamafactory-cli export examples/merge_lora/llama3_full_sft.yaml
```

第一行指的是用 `examples/train_lora/llama3_lora_sft.yaml` 中的配置在预训练模型上进行微调

第二行指的是用 `examples/inference/llama3_lora_sft.yaml` 中的配置和微调后的模型进行对话

第三行指的是将微调模型的adapter与原模型合并，输出符合hf格式的单一模型

- **在预训练模型上进行微调**

打开 `examples/train_lora/llama3_lora_sft.yaml` 文件逐行分析

---

1. 模型配置（`model`）

```yaml
model_name_or_path: /mnt/zzb/workspace/three_data/train_codes/gj/OLMo/tmp/result # 预训练模型或微调检查点的本地路径
trust_remote_code: true # 允许加载自定义模型代码（如非HuggingFace官方模型）
resize_vocab: true # 调整词表大小（如果添加了新token需设为true）
```

---

2. 微调方法（`method`）

```yaml
stage: sft # 训练阶段：监督微调（Supervised Fine-Tuning）
do_train: true # 是否执行训练
finetuning_type: lora # 微调方法：LoRA（低秩适配）
lora_rank: 8 # LoRA矩阵的秩（影响参数量，值越小计算量越小）
lora_target: all # 对哪些层应用LoRA（all表示所有线性层）
```

---

3. 数据集配置（`dataset`）

```yaml
dataset: distill_r1_110k_sft # 数据集名称（需在dataset_info.json中定义）
template: olmo # 模板类型（与OLMo模型匹配的对话格式）
cutoff_len: 2048 # 输入文本的最大token长度（超出部分截断）
max_samples: 10000 # 使用的最大样本数（-1表示全部使用）
overwrite_cache: true # 强制重新生成预处理缓存
preprocessing_num_workers: 16 # 数据预处理的并行线程数
dataloader_num_workers: 4 # 数据加载的子进程数
```

---

4. 输出配置（`output`）

```yaml
output_dir: /mnt/zzb/workspace/three_data/train_codes/gj/OLMo/tmp/sft-result/ # 模型和日志保存路径
logging_steps: 10 # 每10步记录一次日志
save_steps: 500 # 每500步保存一次模型
plot_loss: true # 是否绘制损失曲线
overwrite_output_dir: true # 覆盖已有输出目录
save_only_model: false # 是否仅保存模型(false会保存优化器状态等)
report_to: none # 禁用第三方日志(如WandB/TensorBoard)
```

---

5. 训练参数（`train`）

```yaml
per_device_train_batch_size: 1 # 每个GPU的批次大小
gradient_accumulation_steps: 8 # 梯度累积步数(等效批次大小=1*8=8)
learning_rate: 1.0e-4 # 初始学习率
num_train_epochs: 3.0 # 训练总轮次
lr_scheduler_type: cosine # 学习率调度器(余弦退火)
warmup_ratio: 0.1 # 预热步数占总步数的比例(前10%步数线性增加LR)
bf16: true # 使用bfloat16混合精度训练
ddp_timeout: 180000000 # DDP超时时间(毫秒,用于多卡训练)
resume_from_checkpoint: null # 从检查点恢复训练(null表示不恢复)
```

通常最需要修改的是模型路径`model_name_or_path`和输出路径`output_dir`

> 注意：这里模型路径下的模型，是满足hf标准的模型，这就需要我们对OLMo的pt模型进行转化。我这里用的olmo2的转化脚本(olmo训练的yaml文件里attention_layer_norm: true)

```
#!/bin/bash
/home/gj/miniconda3/envs/OLMo/bin/python \ # OMLo环境下python的位置
scripts/convert_olmo2_to_hf.py \ # 需要执行转化脚本
--input_dir "你的多合一模型的pt路径/step10600-unsharded" \
--output_dir "输出路径/OLMo/tmp/result" \
--tokenizer_json_path "tokenizers/Qwen2.5-0.5B/tokenizer.json"
```

这里可能会出问题，转化后需要chat一下看看输出是否出现中英文交杂或者语法混乱的情况。如果出现上述问题，那么转化的脚本可能选错了，试一下 `convert_olmo_to_hf_new.py`

- **与微调后的模型进行对话**

打开 `examples/inference/llama3_lora_sft.yaml` 文件进行逐行分析

---

1. 模型路径配置

```
model_name_or_path: /mnt/zzb/workspace/three_data/train_codes/gj/OLMo/tmp/result

# 主模型路径（必填），可以是以下两种之一：
# - 预训练模型的本地目录（如`/path/to/olmo-7b`）
# - 微调后的全模型检查点路径

adapter_name_or_path:
/mnt/zzb/workspace/three_data/train_codes/gj/OLMo/tmp/sft-result/
# LoRA适配器路径（可选），若使用LoRA微调且未合并权重，需指定适配器目录
# 注：与`model_name_or_path`配合使用（主模型+适配器）
```

---

2. 模板与推理后端

```
template: olmo
# 对话模板类型(必填),需与模型匹配:
# - `olmo`:OLMo模型的指令模板(如"### Instruction:...### Response:...")
# - 其他选项:`llama3`、`qwen`等(取决于模型架构)

infer_backend: huggingface
# 推理引擎(可选),可选值:
# - `huggingface`:原生Transformers推理(兼容性好)
# - `vllm`:高性能推理(支持连续批处理,吞吐量高)
# - `sglang`:特定场景优化后端
```

---

3. 安全与兼容性

```
trust_remote_code: true
# 是否信任远程代码(必填),加载自定义模型时需设为`true`:
# - 若模型含非HuggingFace官方代码(如OLMo的定制层)
# - 设置为`false`可能引发加载错误
```

---

4. 生成参数控制

```
max_new_tokens: 100
# 生成文本的最大token数(防无限生成),根据任务调整:
# - 短回答:50-100
# - 长文本生成:512-2048

temperature: 0.7
# 温度系数(控制随机性),范围(0, 1]:
# - 值越低输出越确定(如0.2适合事实性回答)
# - 值越高输出越多样(如0.8适合创意任务)

top_k: 50
# Top-k采样(限制候选词数量),仅考虑概率最高的k个token:
# - 设为0禁用,通常50-100
```

```
top_p: 0.9
# Top-p采样(核采样),从累计概率达p的最小集合中采样:
# - 与top_k二选一,通常0.7-0.95

repetition_penalty: 1.1
# 重复惩罚系数,>1时降低重复词概率:
# - 1.0表示无惩罚
# - 1.1-1.5可有效减少重复
```

通常最需要修改的是模型路径`model_name_or_path`和adapter路径`adapter_name_or_path`,通常adapter路径就是模型输出路径

- **原模型与微调模型合并**

打开`examples/merge_lora/llama3_full_sft.yaml`文件进行逐行分析

```
model_name_or_path: /mnt/zzb/workspace/three_data/train_codes/gj/OLMo/tmp/result
#【必填】主模型路径,可以是:
# - 原始预训练模型目录
# - 已微调的全模型检查点路径
# 注意:当与adapter_name_or_path同时使用时,表示"基座模型+LoRA适配器"的组合

adapter_name_or_path:
/mnt/zzb/workspace/three_data/train_codes/gj/OLMo/tmp/sft-result
#【可选】LoRA适配器路径,需满足:
# - 必须是通过LoRA微调生成的输出目录
# - 目录内应包含adapter_model.bin和adapter_config.json
# 若此项为空,则直接导出model_name_or_path的完整模型

template: olmo
#【必填】指定模型对话模板,必须与模型架构匹配:
# - 'olmo' 表示使用OLMo模型的指令模板(如"### Instruction:...### Response:...")
# - 错误设置会导致生成格式混乱
```

```
trust_remote_code: true
#【必填】是否允许加载自定义代码:
# - OLMo等非HuggingFace官方模型需设为true
# - 若设为false可能导致加载失败

export_dir: /mnt/zzb/workspace/three_data/train_codes/gj/OLMo/tmp/sft-export/
#【必填】导出模型的保存目录:
# - 如果是LoRA适配器,此处会保存合并后的完整模型
# - 目录不存在时会自动创建

export_size: 5
#【可选】分片保存的模型文件数量:
# - 5 表示将模型参数分成5个文件(如pytorch_model-00001-of-00005.bin)
# - 设为1表示保存为单个文件
# - 大模型建议分片(便于传输和加载)

export_device: cpu
#【可选】导出时使用的计算设备:
# - 'cpu':强制在CPU上执行合并/转换(内存需求大但兼容性好)
# - 'auto':自动选择可用设备(如有GPU则优先使用)

export_legacy_format: false
#【可选】是否导出为旧格式:
# - false:默认保存为新版HuggingFace格式(含index.json)
# - true:保存为PyTorch传统bin文件(不推荐)
```

下载使用sft数据集

在魔搭或抱抱脸上下载数据集并微调