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：

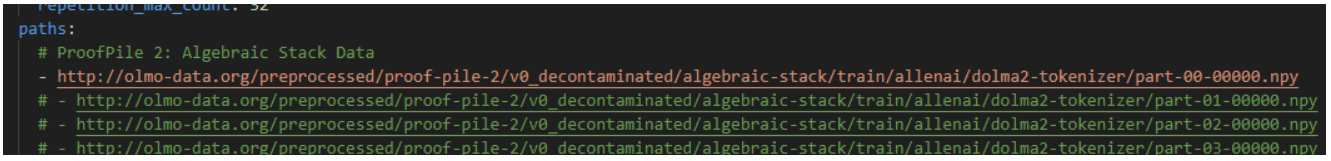
        ```
        import torch
        print(torch.__version__)
        ```

    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}
```

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
from hf_olmo.modeling_olmo import OLMoForCausalLM
from transformers import AutoTokenizer

# --- 添加全局安全定义，避免 torch.load 报错 ---
torch.serialization.add_safe_globals([
    pathlib.PosixPath,
    numpy.core.multiarray._reconstruct,
    numpy.ndarray,
    numpy.dtype,
    numpy.dtypes.UInt32DType,
])

def load_model_and_merge_ckpt():
    # 加载预训练模型结构和 tokenizer
    tokenizer_path = "allenai/OLMo-1B"
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_path)
    model = OLMoForCausalLM.from_pretrained(tokenizer_path)

    # 加载 checkpoint 路径（换成你自己的路径）
    ckpt_base_dir = pathlib.Path("tmp/Checkpoints/OLMo-small/OLMo2-1B-
stage1/latest/train")
    rank_files = sorted(ckpt_base_dir.glob("rank*.pt"))

    # 加载每个 rank 的 state_dict
    state_dicts = []
    for f in rank_files:
        raw = torch.load(f, map_location="cpu")
        state_dicts.append(raw["model"] if "model" in raw else raw)
```

```
    # 合并参数
    merged_state_dict = {}
    for key in state_dicts[0].keys():
        tensors = []
        all_tensor = True
        for sd in state_dicts:
            v = sd[key]
            if not isinstance(v, torch.Tensor):
                all_tensor = False
            tensors.append(v)
        if all_tensor and tensors[0].dim() == 2:
            try:
                merged_state_dict[key] = torch.cat(tensors, dim=0)
            except Exception:
                merged_state_dict[key] = tensors[0]
        else:
            merged_state_dict[key] = tensors[0]

    # 添加 'model.' 前缀
    if not any(k.startswith("model.") for k in merged_state_dict.keys()):
        merged_state_dict = {("model." + k): v for k, v in merged_state_dict.items()}

    # 加载合并后的 state_dict
    model.load_state_dict(merged_state_dict, strict=False)
    model.eval()
    model.to("cuda")  # ✅ 移动模型到 GPU

    return model, tokenizer

def generate_reply(model, tokenizer, prompt, max_length=100):
    inputs = tokenizer(prompt, return_tensors="pt")
    if "token_type_ids" in inputs:
        del inputs["token_type_ids"]
    inputs = {k: v.to("cuda") for k, v in inputs.items()}  # ✅ 输入也放到 GPU
```

```
    with torch.no_grad():
        output_ids = model.generate(**inputs, max_length=max_length, do_sample=False)
    return tokenizer.decode(output_ids[0], skip_special_tokens=True)

def interactive_loop(model, tokenizer):
    print("🔁 进入 OLMo 一问一答模式（输入 q 退出）")
    while True:
        user_input = input("\n👱‍♀️💻 你：")
        if user_input.lower() 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...")
    model, tokenizer = load_model_and_merge_ckpt()
    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。

参数	说明
`--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`

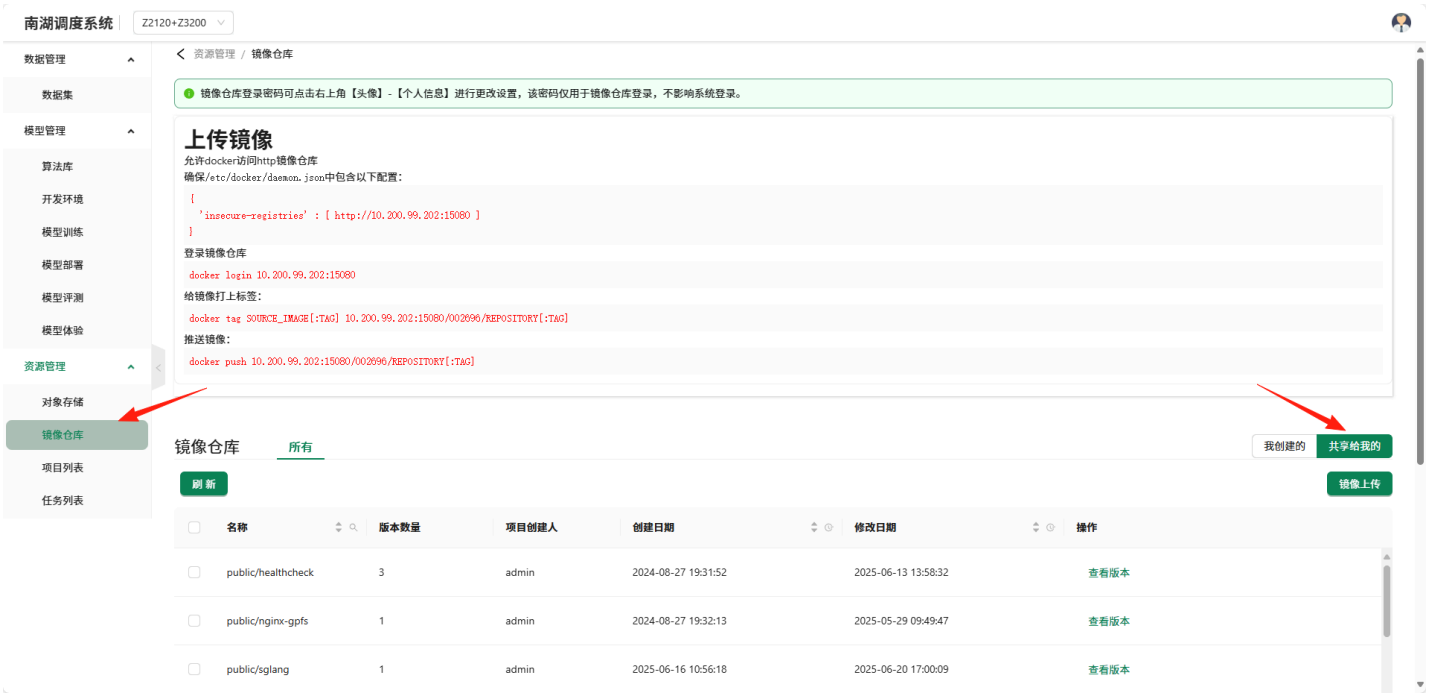
8. 查看清洗好的数据

```
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
```

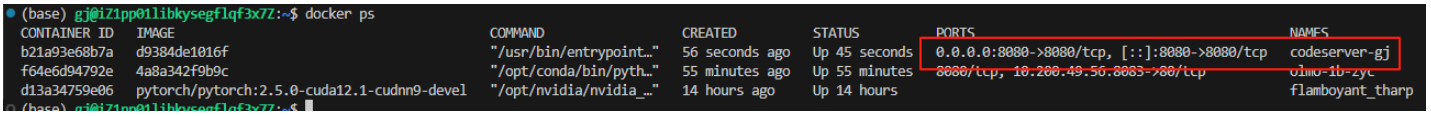
这里我们拉取的是有 vscode web-server 的镜像，这种镜像相对于基础镜像而言，可以用 web 端的vscode 对远程容器进行调试，使用更方便。此外，这个镜像是被拉取到本地的，需要我们对它进行加工，安装 olmo 必要的环境并测试能否运行

启动镜像并进入容器

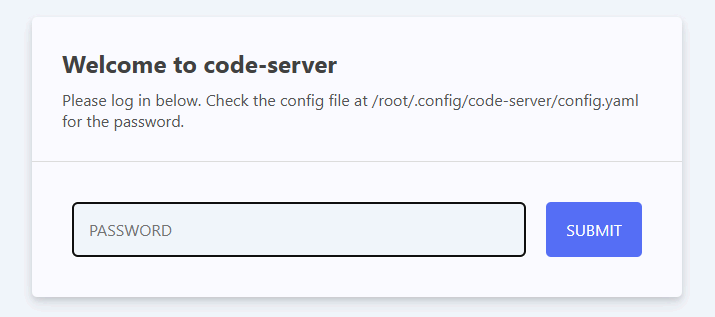
因为拉取的公共镜像需要先在本地进行修改和调试，然后保存修改并打包传送到集群，所以我们需要先在本地启动它。相对于普通镜像直接 `docker run -it 镜像名字 /bin/bash` ，有vscode web-server的镜像需要将容器端口映射到本地才能进行连接。**此外，记住还要挂载自己的工作区！**

```
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

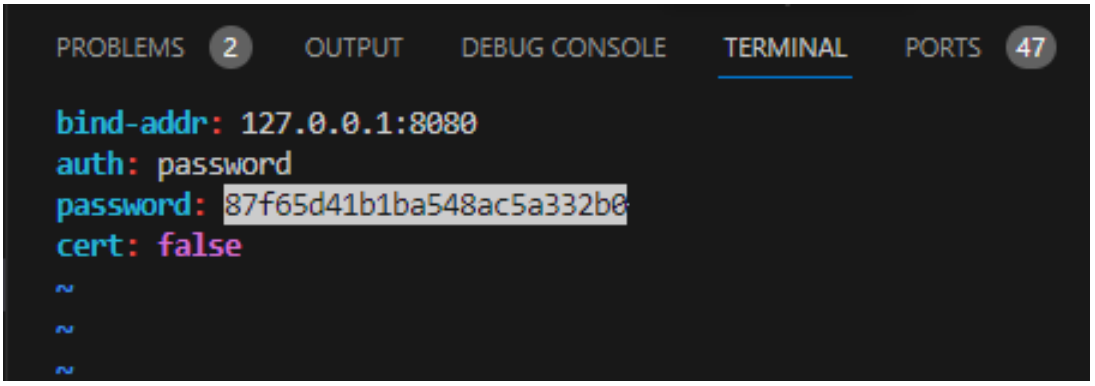
回到本地 vscode，进入到正在运行的容器中

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

打开存放密码的 `yaml` 文件

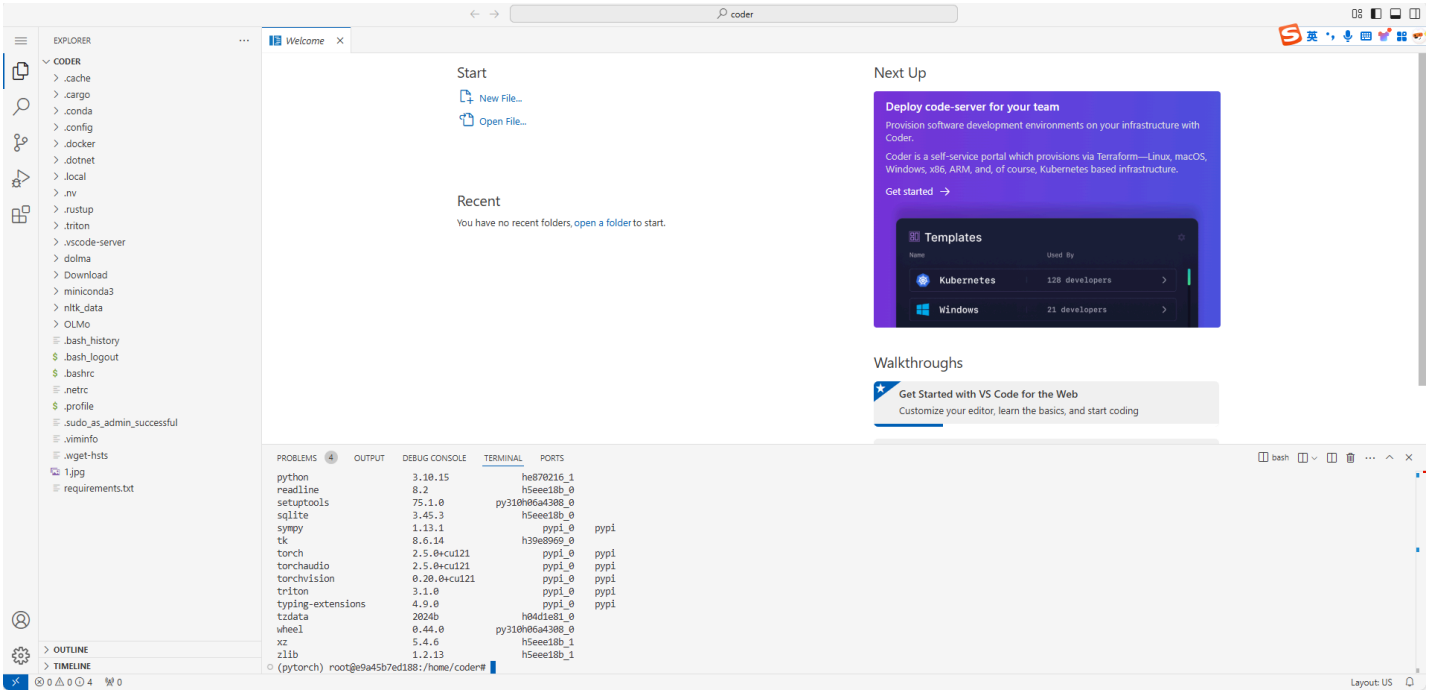
```
vi /root/.config/code-server/config.yaml
```

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
```

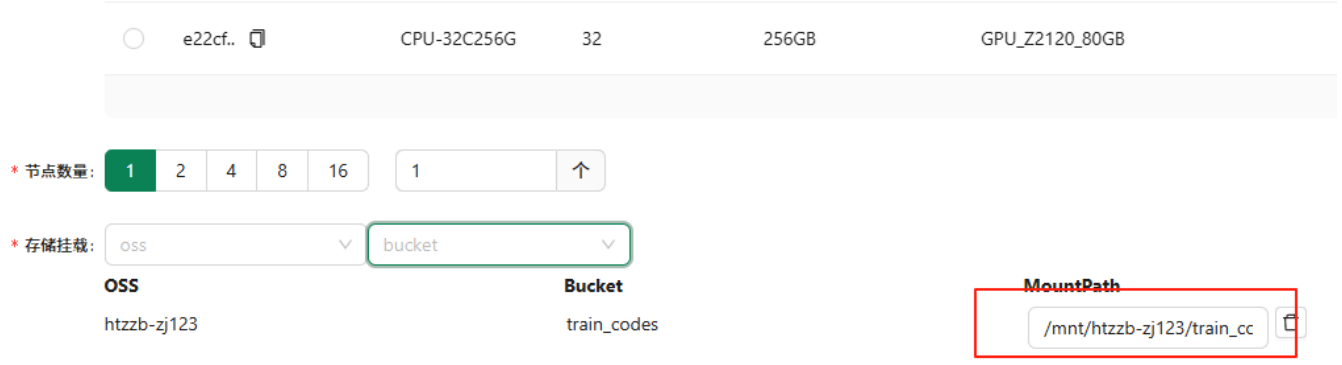
这里需要说明，10.200.99.202:15080是服务器的ip和端口，需要更换的是002696这个工号（换成你自己的，这里和后面的gj需要换成你自己的名字）。注：如果登录出了问题，检查一下是不是要在 `docker login` 前加 `sudo`，可以输入 `groups` 查看自己所在组，如果输出 `你的用户名 docker sudo`，则不需要加 `sudo`

> 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