Quick Start

Docker 使用

我们提供可以运行模型训练和推理的 docker，便于在新环境下快速使用九格大模型。您也可以使用 Conda 配置运行环境。Conda 配置方式请见下一节。

镜像加载

启动 docker 的 rootless 模式

```shell
module load rootless-docker/default
start_rootless_docker.sh
```

运行成功的话，此时执行 docker ps 可以看到当前没有正在运行的容器，如果有正在运行的容器，说明 rootless 模式没有启动成功，请联系管理员。

加载镜像

```shell
docker load -i cpmlive-flash-0.0.4.tar
docker tag [IMAGE_ID] cpmlive-flash:0.0.4
```

QY 服务器中，镜像的路径在 `/data/public/CPM-9G/docker-images` 。

容器启动

启动容器

```shell
docker run -it -d -v [HOST_PATH1]:[DOCKER_PATH1] -v [HOST_PATH2]:[DOCKER_PATH2] --gpus all cpmlive-flash:0.0.4 bash
```

进入容器

```shell
docker exec -it [CONTAINER_ID] bash
```

退出容器

```shell
Ctrl+d
```

删除容器

```shell
docker stop [CONTAINER_ID]
```

查看正在运行容器

```shell
docker ps
```

Conda 环境配置

训练环境配置

1. 使用 python 3.8.10 创建 conda 环境

```bash
conda create -n cpm-9g python=3.8.10
```

2. 安装 Pytorch

```Bash
conda install pytorch==1.13.1 torchvision==0.14.1 torchaudio==0.13.1 pytorch-cuda=11.6 -c pytorch -c nvidia
```

3. 安装 BMTrain

```Bash
pip install bmtrain==0.2.3.post2
```

4. 安装 flash-attn

```Bash
pip install flash-attn==2.0.8
```

5. 安装其他依赖包

```Bash
pip install einops
pip install pytrie
```

推理环境配置

1. 安装 nvidia-nccl

```Bash
pip install nvidia-nccl-cu11==2.19.3
```

配置环境变量

```bash
nccl_root=`python -c "import nvidia.nccl;import os; print(os.path.dirname(nvidia.nccl.__file__))"`
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$nccl_root/lib
echo $LD_LIBRARY_PATH
```

2. 安装 LibCPM

```bash
pip install /data/public/packages/libcpm-1.0.0-cp38-cp38-linux_x86_64.whl
```

Quick Start

为了帮助您快速了解 CPM-9G 的使用，我们准备了一个快速入门教程，目标是基于 CPM-9G 基座模型通过指令微调的方式构建一个 Chat 模型。

本教程使用的数据是 Alpaca Zh，一个开源中文指令微调数据集。数据集的地址为：https://github.com/Instruction-Tuning-with-GPT-4/GPT-4-LLM/blob/main/data/alpaca_gpt4_data_zh.json

数据预处理

1. 准备 jsonl 文件

首先需要将原始数据处理位 jsonl 形式（每一行一个 json），每行包含两个字段，分别是 `input` 和 `output`，以下是一个例子：

```json
{"input":"", "output": "我爱北京。"}
```

实际预训练时，数据多为 txt 格式，可以参照以下过程将 txt 格式的文件转换为 jsonl 格式：

Python

```python
# convert_txt2jsonl.py
import json
import sys

for line in sys.stdin:
    if line.strip() == "":
        continue
    temp_json = {"input": "", "output": line.strip()}
    print(json.dumps(temp_json, ensure_ascii=False))
```

使用方式为：

Shell

```shell
cat pretrain.txt | python convert_txt2jsonl.py > pretrain.jsonl
```

在本 Quick Start 教程中，已准备好 jsonl 数据，路径为 `raw_data/alpaca_zh.jsonl`，示例如下：

JSON

```
1 {"input": "<用户>保持健康的三个提示。<AI>", "output": "以下是保持健康的三个提示：\n\n1. 保持身体活动。每天做适当的身体运动，如散步、跑步或游泳，能促进心血管健康，增强肌肉力量，并有助于减少体重。\n\n2. 均衡饮食。每天食用新鲜的蔬菜、水果、全谷物和脂肪含量低的蛋白质食物，避免高糖、高脂肪和加工食品，以保持健康的饮食习惯。\n\n3. 睡眠充足。睡眠对人体健康至关重要，成年人每天应保证 7-8 小时的睡眠。良好的睡眠有助于减轻压力，促进身体恢复，并提高注意力和记忆力。"}
2 {"input": "<用户>三原色是什么？<AI>", "output": "三原色通常指的是红色、绿色和蓝色（RGB）。它们是通过加色混合原理创建色彩的三种基础颜色。在以发光为基础的显示设备中（如电视、计算机显示器、智能手机和平板电脑显示屏），三原色可混合产生大量色彩。其中红色和绿色可以混合生成黄色，红色和蓝色可以混合生成品红色，蓝色和绿色可以混合生成青色。当红色、绿色和蓝色按相等比例混合时，可以产生白色或灰色。\n\n此外，在印刷和绘画中，三原色指的是以颜料为基础的红、黄和蓝颜色（RYB）。这三种颜色用以通过减色混合原理来创建色彩。不过，三原色的具体定义并不唯一，不同的颜色系统可能会采用不同的三原色。"}
3 {"input": "<用户>描述原子的结构。<AI>", "output": "原子是物质的基本单位，它由三种基本粒子组成：质子、中子和电子。质子和中子形成原子核，位于原子中心，核外的电子围绕着原子核运动。\n\n原子结构具有层次性。原子核中，质子带正电，中子不带电（中性）。原子核非常小且致密，占据了原子总质量的绝大部分。电子带负电，通常围绕核运动，形成若干层次，称为壳层或电子层。电子数量与质子数量相等，使原子呈电中性。\n\n电子在每个壳层中都呈规律分布，并且不同壳层所能容纳的电子数也不同。在最里面的壳层一般只能容纳2个电子，其次一层最多可容纳8个电子，再往外的壳层可容纳的电子数逐层递增。\n\n原子核主要受到两种相互作用力的影响：强力和电磁力。强力的作用范围非常小，主要限制在原子核内，具有极强的吸引作用，使核子（质子和中子）紧密结合在一起。电磁力的作用范围较大，主要通过核外的电子与原子核相互作用，发挥作用。\n\n这就是原子的基本结构。原子内部结构复杂多样，不同元素的原子核中质子、中子数量不同，核外电子排布分布也不同，形成了丰富多彩的化学世界。"}
```

2. 数据二进制化

为了提升数据读取的效率，方便进行大规模分布式预训练，我们以二进制的形式读取训练数据。因此，在训练开始前，需要将上一步准备好的 jsonl 格式的数据文件二进制化，需要的代码路径为 `quick_start/data_binarize.py`，**使用前需要将环境变量设置为您的本地路径：**

Python

```
1 sys.path.insert(0, "/data/public/CPM-9G/9G-Train")
```

以下是一个使用示例：

假设当前的数据在 raw_data 路径下：`raw_data/alpaca_zh.jsonl`

Shell

```
python data_binarize.py --input [PATH to raw_data] --data_type json --output_path [PATH to raw_data_bin] --output_name data
```

处理完成后，在输出路径（即 `OUTPUT PATH`）下会生成 data 和 meta.bin 两个文件，其中 data 是二进制后的数据文件，meta.bin 则记录了这份数据的规模、大小等信息，示例如下：

JSON

```
{"file_name": "data", "block_begin": 0, "block_end": 45, "nbytes": 738321350, "nlines": 4432310, "mask": false, "block_size": 16777216}
```

请注意，当前的框架需要保证 `block_end` 数大于所用的 GPU 总数。

例如，用 32 卡训练时，需满足 `block_end>32`，如果文件较小，可以在二进制化之前对多个小文件进行拼接，以满足大规模训练的需求。

在本 Quick Start 中，我们为 jsonl 数据到二进制数据的转换过程准备了脚本：

Shell

```
for i in {1..10};do
cat raw_data/alpaca_zh.jsonl >> raw_data/alpaca_zh_repeat.jsonl
done

mkdir raw_data_repeat
mv raw_data/alpaca_zh_repeat.jsonl raw_data_repeat/data.jsonl

python data_binarize.py --input raw_data_repeat --data_type json --output_path bin_data_repeat --output_name data
```

3. 准备数据读取脚本

鉴于不同的预训练数据所包含的字段可能有所差别，我们还兼容了字段转换的环节，如果按照上述标准流程做的数据预处理，那么转换方式将十分简单，代码如下：

Python

```python
# transform_script.py
import random


def rand(n: int, r: random.Random):
    return int(r.random() * n)


def transform(data, num_sample: int, r: random.Random):
    return {"input": data["input"], "output": data["output"]}
```

我们还支持多个数据集的混合读入，并设置不同数据集的比例。为此，需要准备一个数据混合的 json 文件，来指导训练过程中的数据读取策略，示例如下：

JSON

```json
[
    {
        "dataset_name": "alpaca_zh",
        "task_name": "alpaca_zh",
        "weight": 1.0,
        "path": "/data/public/CPM-9G/quick_start/bin_data_repeat",
        "incontext_weight": [
            1.0
        ],
        "transforms": "/data/public/CPM-9G/quick_start/transform_data.py"
    }
]
```

该文件中各字段的解释如下：

- dataset_name: 数据集名称；
- task_name: 数据集所属任务，task_name+dataset_name 将作为训练过程中识别数据集的标签，task_name 则可用于训练过程中针对任务分别汇总 loss 信息、token 吞吐量等；
- weight: 浮点数，采样权重；(注意此权重仅代表每个数据集的样本数配比，实际 token 吞吐量的配比还与每个样本的平均 token 数量有关)
- path:meta.bin、二进制数据的父目录，即前文所述的 raw_data_bin;
- transforms: 数据转换脚本对应的路径；

- incontext_weight: 训练样本叠加方式，[1.0] 表示 100% 的概率采样一个样本，[0.8, 0.2] 表示 80% 的概率采样一个样本，20% 概率采样两个样本进行拼接，[0.75, 0.1, 0.15] 表示 15% 概率采样三个样本、 10% 的概率采样两个样本进行拼接、75% 采样一个样本；
- **数据集的配比(即 weight 参数)需要重点调整，对于模型的训练稳定性和最终在各类数据上的能力表现有重大影响；**
- **我们在此文件中指定了数据文件的路径、转换脚本路径等信息，后续训练仅需要系统该文件的路径即可。**

模型训练

模型训练代码的位置：`9G-Train/apps/cpm9g/pretrain_cpm9g.py`

需要将代码中环境变量设置为您的代码路径：

`CPM-9G/apps/cpm9g/pretrain_cpm9g.py:17`

Python

```
1 sys.path.insert(0, "/data/public/CPM-9G/9G-Train")
```

训练脚本如下：

JSON

```
#! /bin/bash

# use 8 GPU for example, pretrain may need 32 GPU
export MASTER_ADDR=`hostname`
export MASTER_PORT=12345

EXP_PATH=. # 修改为您的实验路径，用于存储训练日志和模型
CODE_PATH=/data/public/CPM-9G/9G-Train # 修改为您的代码路径
DATA_PATH=/data/public/CPM-9G/quick_start/datasets.json # 修改为您的datasets.json路径
CHECKPOINT=/data/public/CPM-9G/models/7b-base/7b.pt # 修改为您的基座模型路径

mkdir -p ${EXP_PATH}/logs/debug
mkdir -p ${EXP_PATH}/logs/tensorboard/cpm9g/
CONFIG_NAME="${CODE_PATH}/apps/cpm9g/config/7b"
# --------------- 运行参数 ---------------
OPTS=""
OPTS+=" --model-config ${CONFIG_NAME}/config.json"
OPTS+=" --vocab ${CONFIG_NAME}/vocab.txt"
OPTS+=" --batch-size 12"
OPTS+=" --train-iters 2000" # 训练步数，达到此步数后，学习率降到最小值
OPTS+=" --save-iters 100" # 存储步数，每隔此步数，存储一个模型文件
OPTS+=" --save-name cpm9g_checkpoint" # 模型名称前缀
OPTS+=" --max-length 4096" # 最多token数量
OPTS+=" --lr 1.5e-5" # 峰值学习率
OPTS+=" --inspect-iters 100" # 检查步数，每隔此步数，输出一次模型梯度的详细信息
OPTS+=" --warmup-iters 50" # 热启动步数
OPTS+=" --lr-decay-style noam" # 学习率变化策略
OPTS+=" --weight-decay 0.1" # 正则化参数
OPTS+=" --clip-grad 1.0" # 正则化参数
OPTS+=" --loss-scale 1048576" # 和训练稳定性相关，一般情况下不需修改
OPTS+=" --loss-scale-steps 32" # 和训练稳定性相关，一般情况下不需修改
OPTS+=" --offload" # 使用cpu offload将优化器参数转移到cpu，一般情况下无需修改
OPTS+=" --flash cuda"
# OPTS+=" --load-grad"

# --------------- 写文件路径 ---------------
## checkpoint
OPTS+=" --save ${EXP_PATH}/checkpoints/cpm9g/"
OPTS+=" --save-model ${EXP_PATH}/models/cpm9g/"

```

```
## logs，/local/logs 等价于 /data/logs（软链）
OPTS+=" --log-dir ${EXP_PATH}/logs/train/"
OPTS+=" --tensorboard ${EXP_PATH}/tensorboard/cpm9g/"`date +"%Y%m%d%H%M%S"`

# ---------------- 读文件路径 ----------------
OPTS+=" --dataset ${DATA_PATH}"
OPTS+=" --load ${CHECKPOINT}"
OPTS+=" --start-step 1"

# ---------------- 透传参数 ----------------
OPTS+=" $@"

# ---------------- 最终指令 ----------------

CMD="torchrun --nnodes=1 --nproc_per_node=8 --rdzv_id=1 --rdzv_backend=c10d --rdzv_endpoint=${MASTER_ADDR}:${MASTER_PORT} ${CODE_PATH}/apps/cpm9g/pretrain_cpm9g.py ${OPTS}"
echo "${CMD}"

$CMD
```

模型推理

Python

```python
import os

from libcpm import CPM9G

import argparse, json, os

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--pt", type=str, help="the path of ckpt")
    parser.add_argument("--config", type=str, help="the path of config file")
    parser.add_argument("--vocab", type=str, help="the path of vocab file")
    args = parser.parse_args()

    model_config = json.load(open(args.config, 'r'))
    model_config["new_vocab"] = True

    model = CPM9G(
        "",
        args.vocab,
        0,
        memory_limit = 30 << 30,
        model_config=model_config,
        load_model=False,
    )
    model.load_model_pt(args.pt)

    datas = [
        '''<用户>马化腾是谁？<AI>''',
        '''<用户>你是谁？<AI>''',
        '''<用户>我要参加一个高性能会议，请帮我写一个致辞。<AI>''',
    ]

    # print(model.inference(datas, max_length=30))  # inference batch

    for data in datas:
        res = model.inference(data, max_length=4096)
        print(res['result'])
        # print(model.random_search(data))

```

```
40 if __name__ == "__main__":
41     main()
```

敏捷部署套件资源

- 9G-Train 代码
- Docker 镜像
- LibCPM whl 文件
- Quick Start 资源包，包括数据、数据预处理脚本
- CPM9G-11B 基座模型