【从零训练Steel-LLM】预训练代码讲解、改进与测试

原创 战士金 炼钢AI 2024年08月21日 10:09 北京

此篇文章24年5月5日首发于我的zhi hu帐号“战士金”，内容有略微改动。预训练工作已经做完，后续还会做一些sft和评测的工作。

① 前言

我们的目标是从0预训练一个1B左右的LLM，使用T级别的数据，模型被称为Steel-LLM。我们会分享预训练过程中的关于数据收集、清洗、模型设计、训练程序等内容的所有细节和代码，更详细的项目介绍请见本系列的第一篇文章：【从零训练Steel-LLM】 预训练数据收集与处理。也欢迎各位进群交流，群超过200人了，加vx：a1843450905拉群。

```
github：https://github.com/zhanshijinwat/Steel-LLM/tree/main
```

本篇文章是该系列的第二篇文章，主要讲解预训练代码的改进与测试。我们选择在TinyLlama预训练代码的基础上进行改进，主要有如下两点考虑：

1. 代码封装简单：TinyLlama的代码很简洁，便于阅读学习且易于改造。
2. 基础功能齐全：对于训练1B模型来说，单卡即可训练，使用Transformers库或者原始的pytorch DDP都能训练。但是T级别的数据量是不小的，基本无法一次读入内存。**TinyLlama提供了基础的动态读取数据方法，不需要将数据一次性读入内存。**除此之外，TinyLlama还集成了wandb模块，方便观察实验结果。

② 代码讲解

首先，本文先对Steel-LLM项目中复用了TinyLlama的代码逻辑进行讲解，分为“数据预处理与读取”和“训练流程”两部分介绍。之后再介绍我们对其进行的改进。

2.1数据预处理与读取

因为预训练通常涉及到大量数据，将文本转化为token id的时间会比较长，因此TinyLlama有个数据预处理的步骤提前将文本转化为token id保存到bin格式的文件中供后续训练时使用，而不是在训练过程中即时的将文本转为token id（非常多的LLM训练项目都是这种方式）。保存到bin格式文件中的数据格式如下图所示，是一个形状为(blocks_in_a_chunck，block_size+1)（blocks_in_a_chunck：文件中的数据条数，block_size：每条数据的长度，block_size之所以要+1是因为训练方式为预测next token，句子需要移一个位置当作标签）的二维矩阵。实际处理时， 先用pad_token_id预填充这个二维矩阵,然后再往进塞文本的token id。二维矩阵每一行存放训练时的一段文本对应的token id，文本如果太长就直接截断，放到下一行。一个大的数据集最终会被处理成多个bin格式的文件，每个文件大小是一样的，最后一个文件的二维矩阵基本不会被完全填满，有很多行只有pad_token_id。在Steel-LLM项目中，Steel-LLM/pretrain_modify_from_TinyLlama/scripts/prepare_steel_llm_data.py文件实现了这部分数据预处理的逻辑，更具体的数据预处理细节请看我的上一篇文章。

321	222	1345	4704	666	377
377	eos_token_id	2235	7427	618	866
73	52	33	eos_token_id	pad_token_id	pad_token_id
pad_token_id	pad_token_id	pad_token_id	pad_token_id	pad_token_id	pad_token_id
pad_token_id	pad_token_id	pad_token_id	pad_token_id	pad_token_id	pad_token_id
pad_token_id	pad_token_id	pad_token_id	pad_token_id	pad_token_id	pad_token_id

接下来，介绍一下数据读取的逻辑。我们之所将文本转化为token id后存储到每个part大小都一样的二进制bin文件中是为了方便**在不将数据实际读取到内存的情况下，能够为数据建立索引，并进行样本的shuffle**。数据量大的情况下，我们没必要将每块数据都实际读到内存中，而仅仅需要读取数据的“视图”。读取数据的“视图”可以简单理解为仅读取数据的地址，TinyLlama中具体的实现如下所示，self._buffers保存了所有bin格式数据的视图。

```
mmap = np.memmap(file_dir, mode="r", order="C", offset=HDR_SIZE)
self._mmaps.append(mmap)
self._buffers.append(memoryview(mmap))
```

因为每个bin格式的文件大小都是提前定义好的，且每个文件大小都一样，所以可以很轻松的得到这些bin文件中一共的数据条数n_total，然后生成0~n_total-1的list，当作数据索引，进行shuffle。当想抽取第block_idx 条数据时，因为每块bin文件中包含的数据条数_n_blocks都是一样的，可以通过block_idx //_n_blocks的方式快速的定位到第block_idx条数据在哪个bin文件中。由于我们维护了每个bin文件的数据视图，并且每条数据的长度_block_size都一样，也可以通过block_idx %_n_blocks的方式快读定位到第idx条数据在bin文件中的偏移地址，具体实现逻辑如下所示，在Steel-LLM/pretrain_modify_from_TinyLlama/lit_gpt/packed_dataset.py中实现。

```
block_idx = self._block_idxs[self._curr_idx]
chunk_id = block_idx // self._n_blocks
buffer = self._buffers[chunk_id]
elem_id = (block_idx % self._n_blocks) * self._block_size
offset = np.dtype(self._dtype).itemsize * elem_id
arr = np.frombuffer(buffer, dtype=self._dtype, count=self._block_size, offset=
```

2.2 训练流程

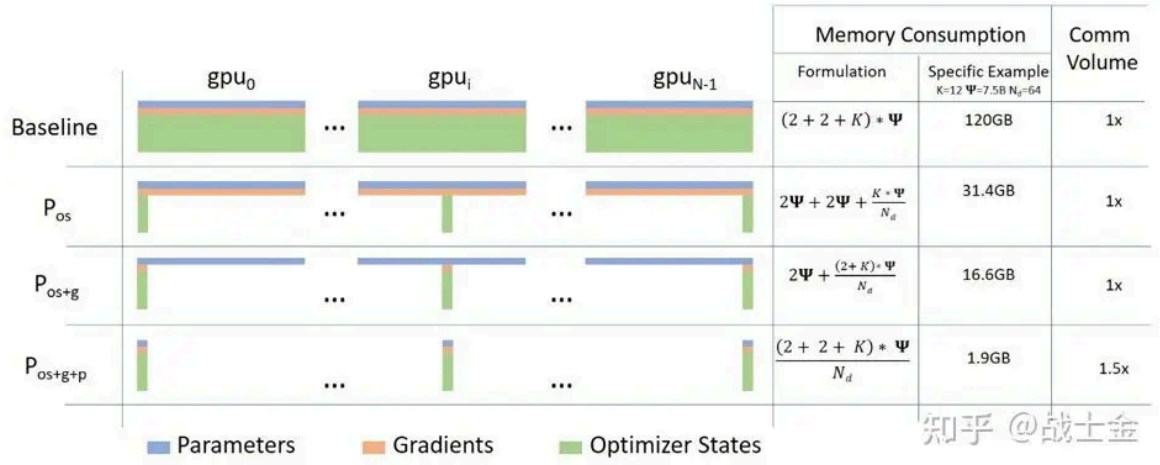
TinyLlama使用lightning.frabic工具包来管理训练流程，本质上就是对pytorch进一步进行封装，让用户更容易使用，lightning生态在国内并不是很流行。在进行数据并行训练模型时，frabic有如下特点：

a.不用大改代码逻辑，轻松从CPU转换到GPU甚至扩展到多结点GPU。

b.支持LLM生态下流行的训练工具，如FSDP、DeepSpeed、混合精度训练等。

c.自动协调各worker的职能，如只让各结点的worker0保存模型参数（通过fabric_rank_zero_only装饰器实现）

具体的训练策略上，TinyLlama使用了FSDP（Fully-sharded data-parallel，完全分片数据并行）技术，以在多卡训练情况下节约显存。其本质上和DeepSpeed的ZeRO stage3技术相同。LLM生态下，和FSDP相比，DeepSpeed的ZeRO是更加广为人知的，因此以DeepSpeed为例，介绍一下这类方法的原理。



	Memory Consumption		Comm Volume
	Formulation	Specific Example K=12 Ψ=7.5B N_d=64	
Baseline	$(2+2+K)*\Psi$	120GB	1x
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

ZeRO有3种模式（stage），如上图所示：
stage 1(p_{os})：优化器状态（如Adam的一阶矩估计和二阶矩估计）分片存储到每个worker，模型参数和梯度每个worker保存一份
stage2 (p_{os+g})：优化器状态和梯度分片存储到每个梯度，模型参数每个worker保存一份
stage3(p_{os+g+p})：优化器状态、梯度、和模型参数都分片存储到每个worker

图中的baseline指的原始的DDP，每个worker都要保存完整的模型参数、梯度和优化器状态。ZeRO和FSDP技术因为分片存储训练时的静态参数，可以节约显存，且训练中使用的卡越多省显存也就越多。和原始的DDP相比，使用stage 1和stage2不会增加训练的通信量，因为原始的DDP也需要做all reduce同步梯度和优化器状态，使用stage3会在训练中增加传输模型参数的通信量。

具体实现上，TinyLlama使用了lightning库的FSDPStrategy接口实现FSDP，底层调用的是pytorch实现的FullyShardedDataParallel类。虽说FSDP最开始实现的是ZeRO stage3，但为了减少通信量，也支持ZeRO stage2，通过设置sharding_strategy参数为“SHARD_GRAD_OP”即可。笔者使用的是单机8卡 A100，在NVLink加持下单机上卡间通信速度还是比较快的，使用ZeRO stage2和ZeRO stage3对训练速度影响其实并不大，后文也会给出具体的实验对比。因为Steel-LLM项目训练的模型仅会有1B左右大小，显存不是很缺，不考虑使用cpu offload技术，这会降低训练速度。

TinyLlama还使用了flash attention库提供的FusedCrossEntropyLoss类，使用cuda编程实现了高效的交叉熵损失函数，进一步提高训练效率。

③ 代码改进

基于TinyLlama代码，进行了如下改进：

3.1 兼容HuggingFace格式的模型

TinyLlama使用的模型结构是定义在项目里边的，如果使用TinyLlama项目训练其他模型不是很方便。我们将其修改为使用Huggingface的Transformers库的模式定义模型（截至到2024.5.1，因为还未完成后续实际训练的模型设计工作，项目中的模型使用的是qwen1 1.8B，用来先调通训练流程）。

我也曾尝试将模型保存/加载的接口也替换为Transformers库提供的方式，但是在多卡模式下，保存模型时会卡住，怀疑是fabric存在一些bug，目前还没有解决，故继续沿用

TinyLlama提供的模型保存/加载接口。等模型训练完毕后，转为Huggingface模型格式即可。

3.2 数据训练进度恢复

如果训练过程中断，想要继续训练的话，不仅要恢复模型和优化器的状态，还需要恢复数据训练的进度。目前TinyLlama提供的恢复数据训练进度的方法比较简单粗暴，具体逻辑如下代码所示。

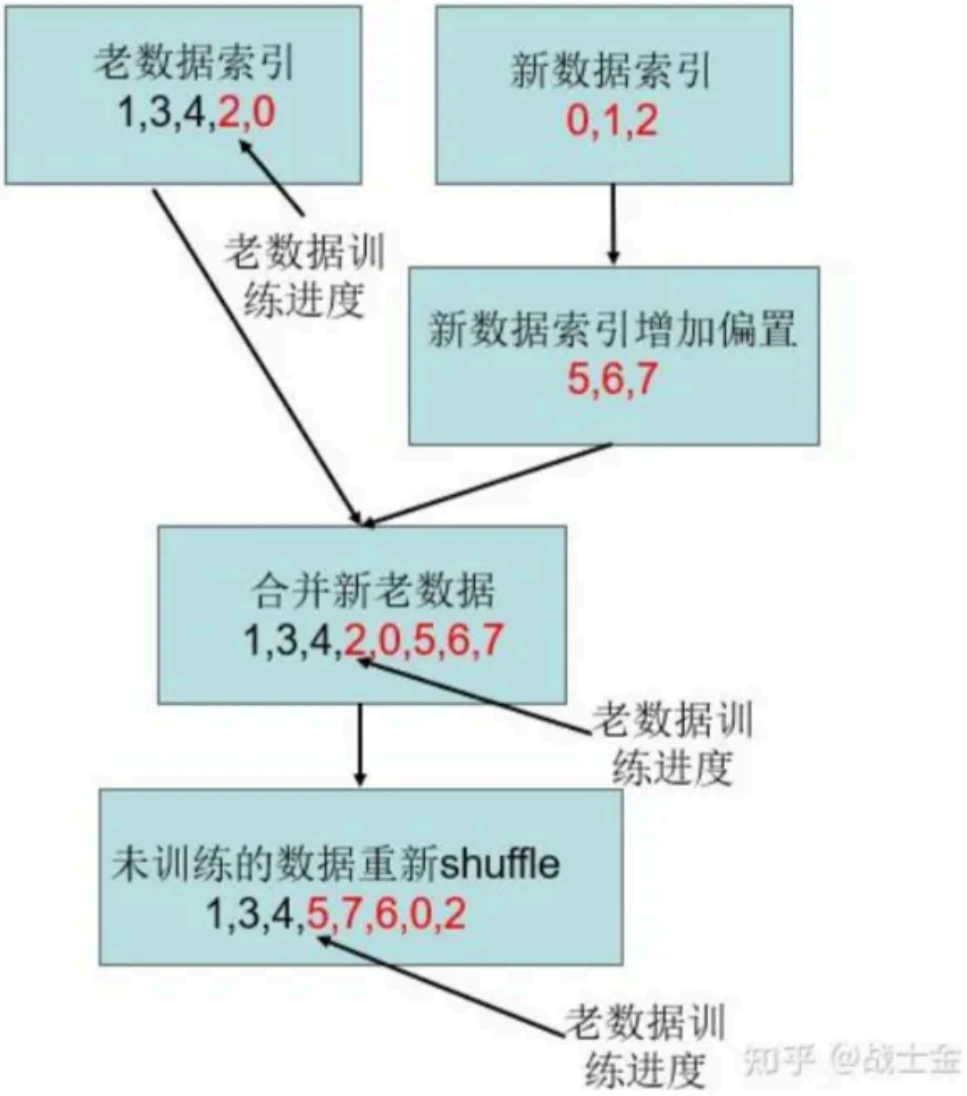
```
for  train_data in train_dataloader:
        # resume loader state. This is not elegant but it works. Should rewrit
        if resume:
            if curr_iter < initial_iter:
                curr_iter += 1
                continue
```

记录模型checkpoint时候也记录下迭代轮数，加载checkpoint之后直接跳过之前迭代过的轮数的数据。使用这种方法能成功恢复数据训练进度的前提是：训练过程中数据不能有变动。往训练数据里追加数据或者删除数据都会使恢复的训练进度是错误的。

TinyLlama中，PackedDatasetIterator类保存了数据训练进度的所有信息，包括所有训练数据的文件名、每条数据的索引等。我利用pickle库，将PackedDatasetIterator对象的内容序列化后并保存，以实现在加载模型checkpoint时也完全恢复数据训练进度。类对象中_mmaps和_buffers变量分别np.memmap和memoryview格式的变量（和文件地址相关）是没办法序列化的，因此不保存，在加载PackedDatasetIterator类时重建即可。

3.1 训练过程中追加数据

LLM训练时常通常多达几十天，在训练过程中（加载checkpoint时）追加数据是很常见的需求。基于对PackedDatasetIterator对象的保存和加载能力，在add_new_data函数中实现了训练过程中追加数据的功能。具体原理如下图所示。新追加的数据索引会和老数据中未训练的数据索引（图中红色数字表示）重新shuffle，**防止新老数据分布差异过大，影响后续的模型训练效果。**



为了防止数据块被意外的重复追加到训练过程中，Steel-LLM还实现了使用hash值检测数据内容是否重复的功能，具体使用的是MD5哈希算法。对一个1.6g的文件进行哈希大约要9s，效率太低。因此选择只用一块数据的头部和尾部数据计算哈希值。

④ 性能测试

代码修改后，也简单测试了下使用这套代码的训练效率，实验设置如下：

a. 硬件：8*A100 80g

b. 模型：qwen1 1.8B（吐槽一下，调试还是用qwen1好，模型代码就直接在huggingface下载的文件夹里，方便修改，qwen1.5已经集成到Transformers库里边了）

c.和训练速度有关的超参数：序列长度=2048；micro_batch_size=4（每次前向传播的batch size）；

qwen1模型代码中除了使用了flash-attention库中flash attention v2算子，也集成了cuda版本的旋转位置编码和RMSNorm算子，我也测试了这些算子对训练效率的影响，整体上来讲，测试了如下因素对训练效率的影响：

a. 是否使用flash-attention库提供的flash attention算法实现：本次测试使用Qwen1模型也支持pytorch实现的flash attention算法（F.scaled_dot_product_attention接口实现），但是效率不如flash-attention库的实现，flash-attention库实现了flash attention v2算法。

b. 是否使用flash-attention库提供的旋转位置编码(RoPE)的cuda实现（apply_rotary_emb_func 函数）
c.是否使用flash-attention库提供的RMSNorm的cuda实现（rms_norm 函数）
d. 是否使用flash-attention库提供的cuda版本交叉熵损失函数（FusedCrossEntropyLoss）
e.在数据并行训练时，是否将模型参数拆分到各个gpu上保存（对应deepspeed zero stage2和stage3）

实验结果如下表所示，baseline的实验设置为**flash attention v2+cuda版本RoPE+torch版本RMSNorm+cuda版本交叉熵损失函数+将模型参数拆分到各个gpu上（zero stage3）**。实验名称的命名规则举个例子，“baseline+torch flash attention”表示在baseline的实验设置下，将flash attention v2替换为torch提供的flash attention。从实验结果中可以看出，使用flash-attention库提供的cuda版本的RMSNorm、RoPE、crossentropy也可提高训练训练效率，并且大都可以节约显存。

实验名称	训练效率(tokens/s/gpu)	每个gpu显存使用量
baseline	13400	65G
baseline+torch RoPE	12500	65G
baseline+torch flash attention	10600	69.5G
baseline+zero2	13800	69G
baseline+cuda RMSNorm	14600	61G
baseline+torch crossentropy loss	13000	75G

最后，我们来对比一下使用所有训练提速手段和不使用任何训练提速手段的差距：
a. 使用所有提速手段：flash attention v2+cuda RoPE+cuda RMSNorm+cuda crossentropy loss+zero stage 2
b. 不使用任何提速手段：torch flash attention+torch RoPE+ torch RMSNorm+ torch crossentropy loss+ zero stage3

实验名称	训练效率(tokens/s/gpu)	每个gpu显存使用量	MFU
使用所有提速手段	15000	66G	63.0%
不使用任何提速手段	10500	75G	43.7%

可见，使用各种提速首段后，训练效率提高了50%，显存节约了12%。模型算力利用率（Model FLOPs Utilization， MFU）指模型一次前反向计算消耗的算力与机器能够提供的算力的比值。使用了flash attention等优化手段后，因为显存读取时间降低，MFU也有所提升，更有效地使用了GPU。

⑤ 环境安装

从性能测试小节可以看出，使用flash-attention库可以大大提升训练效率，但该库不能直接用pip安装，请按照flash-attention gihub仓库的指示进行安装。如果想用flash-attention库提供的cuda版本的RoPE、RMSNorm、交叉熵损失函数，请下载flash-attention源码，在csrc文件下找到想要安装的算子，进到对应的文件夹后，执行如下指令进行安装：

```
pip install .
```

安装过程中你有可能遇到如下错误：

```
subprocess.CalledProcessError: Command '['ninja', '-v']' returned non-zero exi
```

有可能时你的环境变量没有设置，请按照如下模板进行设置（注意替换成你自己的cuda版本以及gcc版本）：

```
CUDAVER=cuda-11.8
export PATH=/usr/local/$CUDAVER/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/$CUDAVER/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/usr/local/$CUDAVER/lib64:$LD_LIBRARY_PATH
export CUDA_PATH=/usr/local/$CUDAVER
export CUDA_ROOT=/usr/local/$CUDAVER
export CUDA_HOME=/usr/local/$CUDAVER
export CUDA_HOST_COMPILER=/usr/bin/gcc-xxxx
```

点个关注再走吧~

炼钢AI

个人公众号，首本RAG相关书籍《大模型RAG实战》、开源预训练项目Steel-LLM作者，...

7篇原创内容

公众号