【从零训练Steel-LLM】模型设计

原创 战士金 炼钢AI 2024年09月03日 23:20 北京

这是从零训练Steel-LLM的第三篇文章，于24年7月9日首发于我的zhi hu帐号：“战士金”，略有修改。目前正在进行模型微调和评估的相关工作，近期已经将训练过程中的多个checkpoint上传到HuggingFace，最终一共训练了1060k个step，1.1T个token（2个epoch）。

① 从零训练Steel-LLM目录

② 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。相关资源链接如下：

```
github链接：https://github.com/zhanshijinwat/Steel-LLM/tree/main
huggingface链接：https://huggingface.co/gqszhanshijin/Steel-LLM
```

本篇文章是该系列的第三篇文章，主要分享一下笔者在模型设计上的思考与探索。

③ 关于Scaling Law

考虑到算力以及训练完的模型最终包含的知识量，Steel-LLM在开始时就已经确定了要训练的是1B的模型，使用1T左右的数据（最终用来训练的数据为1.6T数据，400B个token）。但在项目过程中，仍然简单的根据scaling law计算了一下在我们拥有的算力的情况下，模型尺寸和数据规模的“最优”（能达到最低的loss）值。计算scaling law时，Steel-LLM项目并未使用Chinchilla等早起工作拟合出来的参数，而是使用DeepSeek技术报告中给出的参数，如下所示，M为模型规模，D为数据规模，C为预计使用的计算里量：

$$M_{\text{opt}} = M_{\text{base}} \cdot C^{a}, \quad M_{\text{base}} = 0.1715, \quad a = 0.5243$$
$$D_{\text{opt}} = D_{\text{base}} \cdot C^{b}, \quad D_{\text{base}} = 5.8316, \quad b = 0.4757$$

通过在wandb上的打点来看，训练1.1B模型时我们单卡A100实际的算力是 $1.88*10^{14}$ flops/s左右（因为是数据并行，各卡是独立消费token的，因此算scaling law时候用单卡的算力算），在假设训练25天的情况下，能达到最低loss的模型大小为10B左右，单卡数据消费量为36B左右。（但如果真换成10B模型，25天应该消费不了36B数据，因为mfu会下降，实际的每秒的flops不会有 $1.88*10^{14}$ 这么高，所以这块的具体数值并不准

确，计算的最优的模型和数据规模是偏大的。这篇文章撰写时候，我们的最终模型已经开始训练，暂时不会停下来测scaling law这块。并且，单卡也撑不下10B模型的训练）。

```
C = 1.8*10**14* 3600 *24 * 25
M_opt = 0.1715*C**0.5243 # 10701818976
D_opt = 5.8316*C**0.4757 # 36334610364
```

我们的训练目标是，在有限算力并且模型不太小的情况下，尽量消费更多的数据，让模型学到更多的东西，因此并不严格遵守scaling law。并且，目前开源的LLM训练的数据量通常也是比通过scaling law计算出来的"最优数据量"大的多的多的。

④ 模型设计

对于LLM的模型结构来讲，有两大部分可以进行改造：self attention和FFN。self attention目前被广泛使用的工程实现是Flash Attention V2，下表给出了使用Flash Attenion V2与Pytorch实现的self attention的速度和显存对比，序列长度为2048，batch size为2：

	训练效率(tokens/s/gpu)	每个gpu显存使用量
pytorch self attention	7500	64G
Flash Attention V2	10100	43G

可见使用Flash Attention V2在训练速度（减少了从显存上读取的数据量）和显存占用（节约了注意力矩阵的显存）上都有显著的优势。我们如果对self attention结构进行改造，也需要实现配套的高效计算算子才能满足训练速度和显存要求，但我们并没有这方面的经验以及精力，因此self attention这块就沿用最原始的self attention，并使用Flash Attention V2。pytorch的scaled_dot_product_attention接口内置了Flash Atention的实现，使用起来还是比较方便的。需要注意的是，**pytorch需要升级到2.2以上版本才是Flash Atention V2，否则是Flash Atention V1，会影响计算速度。**

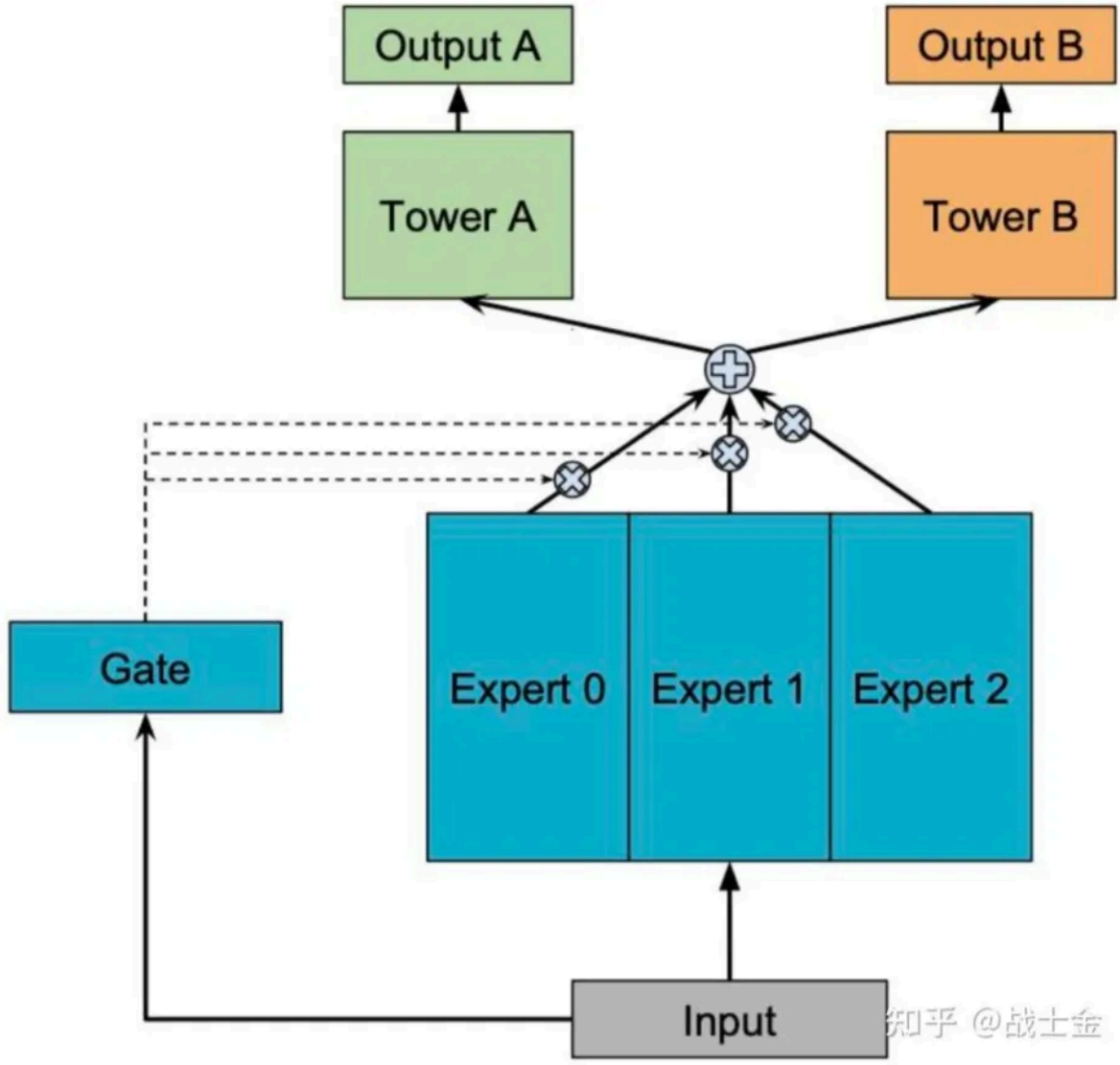
Steel LLM项目在模型结构上的改进主要聚焦于FFN层，有两点改进：SoftMOE以及双层SENET。

4.1 SoftMOE

MOE其实是个很古早的技术，起源于1991年的论文《Adaptive Mixtures of Local Experts》。在LLM中，MOE结构出现在FFN层，且使用的是hard MOE，即每个token仅由部分expert进行计算。大型LLM使用hard MOE的原因在于因为每次前向传播只激活FFN层的部分expert，可以减少训练和推理时的计算量。需要注意的是，**LLM中使用hard MOE并不会节约非常多的显存**，因为在训练和推理时仍然需要将完整的模型加载到显存中。带有hard MOE结构的LLM在进行效果比较时普遍只会在**相同激活参数量的层面**上进行比较。举个例子，Qwen1.5-MoE-A2.7B效果声称和Qwen1.5-7B相当，前者训练和推理时仅激活了2.7B参数，后者则激活了7B的参数。但实际上Qwen1.5-MoE-A2.7B整体上有14B左右的参数量。而具有hard MOE结构的14B LLM效果是不如激活全部14B参数的LLM的。

具体情况具体分析，我们的显存有限，想充分利用显存中的每一份参数进行计算，同时也想将MOE结构纳入到模型中，因此最终选择使用Soft MOE结构，即每个token由所有expert进行计算，最后进行加权求和。在工业界中，其实SoftMOE几年前就在搜广推领域被

应用了，如下图所示。（两年前随便在知乎上一搜MOE，出来的都是搜广推模型，现在出来的都是LLM的东西了hhh）。input就是一个向量，通过不同的专家网络计算出结果，然后再通过input计算出来的权重进行加权求和。在LLM中的FFN层使用SoftMOE原理类似，只不过有多个input（即多个token），分别通过各个专家并进行加权求和。



Steel LLM项目中，笔者尝试了3种Soft MOE的实现。

SoftMoe V1

v1 版本的Soft MOE实现如下所示。即按照最朴素的Soft MOE原理进行实现，每个token都会被每个expert进行处理。该种实现方法训练速度较慢。

```
class SteelSoftMoeV1(nn.Module):
    def __init__(self, config, layer=None):
        super().__init__()
        self.config = config
        self.experts = nn.ModuleList([layer(config) for _ in range(config.n_e
        self.gating = nn.Linear(config.hidden_size, config.n_experts)
    def forward(self, x):
        weights = self.gating(x)
        weights = nn.functional.softmax(weights, dim=-1, dtype=torch.float32)
        outputs = torch.stack(
```

```
            [expert(x) for expert in self.experts], dim=2)
        weights = weights.unsqueeze(-1)
        return torch.sum(outputs * weights, dim=2)
```

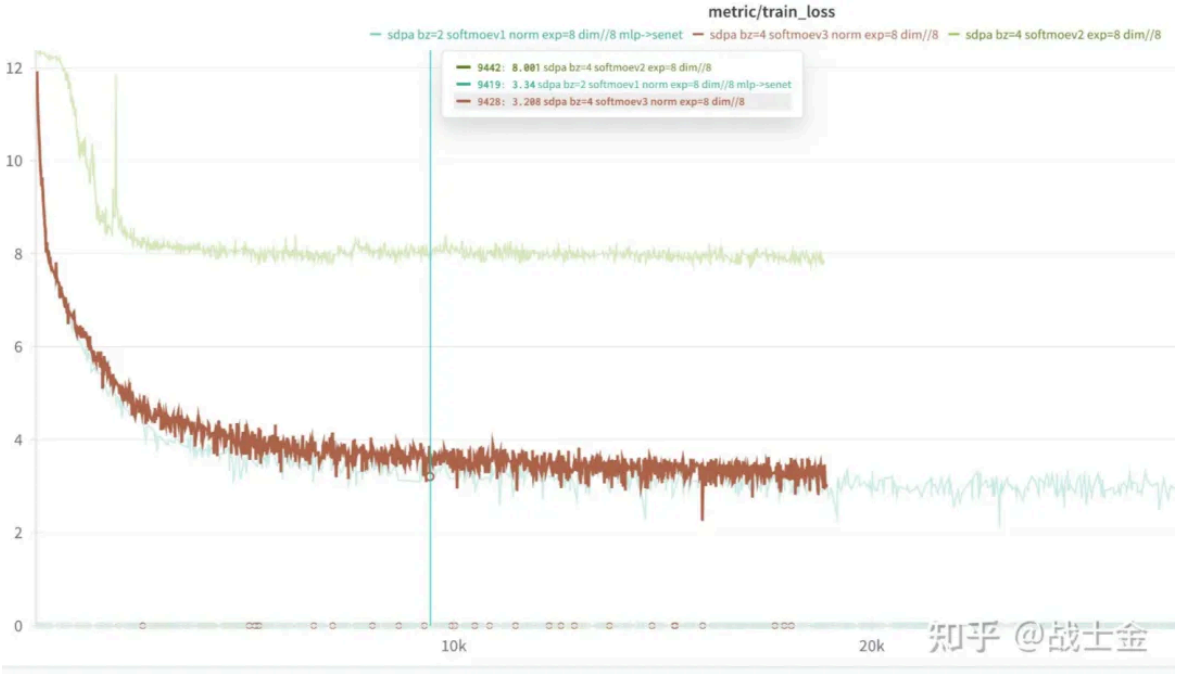
SoftMoe V2

该版本Soft MOE的实现来自于如下github项目：https://github.com/lucidrains/soft-moe-pytorch
是论文《From Sparse to Soft Mixtures of Experts》的复现。和Soft MOE v1相比，有更高的计算效率。之所以有更高的计算效率，是因为“torch.bmm(dispatch_weights.transpose(1, 2), x)”这行代码相当于把输入在序列长度(seq_len)维度进行了加权求和，最后再通过“torch.bmm(combine_weights, ys)”将序列长度维度恢复出来。

```
class SteelSoftMoeV2(nn.Module):
    def __init__(self, config, layer):
        super().__init__()
        self.config = config
        self.experts = nn.ModuleList([layer(config) for _ in range(config.n_e
        self.score = nn.Parameter(torch.randn(config.hidden_size, config.n_ex

    def forward(self, x):
        logits = torch.matmul(x, self.score) # (batch_size, seq_len, n_expert
        dispatch_weights = F.softmax(logits, dim=-1,dtype=torch.float32).to(x
        combine_weights = F.softmax(logits, dim=1,dtype=torch.float32).to(x.d
        # 序列维度加权求和
        xs = torch.bmm(dispatch_weights.transpose(1, 2), x) # (batch_size, n_
        ys = torch.cat(
            [expert(xs[:, i * self.config.slots_per_expert : (i + 1) * self.c
                        for i, expert in enumerate(self.experts)],
            dim=1
            )
        # 恢复序列长度维度
        y = torch.bmm(combine_weights, ys)
        return y
```

这一版Soft MOE实现的有点问题，loss无法收敛到正常水平，笔者目前不太清楚原因。下图给出了3种Soft MOE在训练前期loss的收敛情况。基于Qwen1.5-1.8B，8个expert，同时FFN层的dim缩小8倍，保持1.8B的总参数量不变。



SoftMoe V3

v3版本的soft MOE依然是论文《From Sparse to Soft Mixtures of Experts》的复现，来自如下github项目，具体代码就不贴了，比较长。

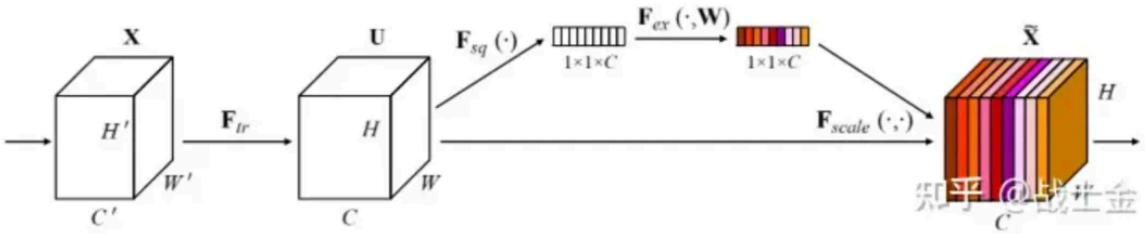
```
https://github.com/bwconrad/soft-moe/blob/main/soft_moe/soft_moe.py。
```

最后放上Soft MOE v1 v2和v3训练效率，以及显存使用情况。在batch size=2的情况下，soft moe v1的显存直接OOM了，因此放上一个batch size=2情况下的训练效率和显存。可见，softmoe v1的显存需求比较大，最终Steel LLM选择使用softmoe v3。

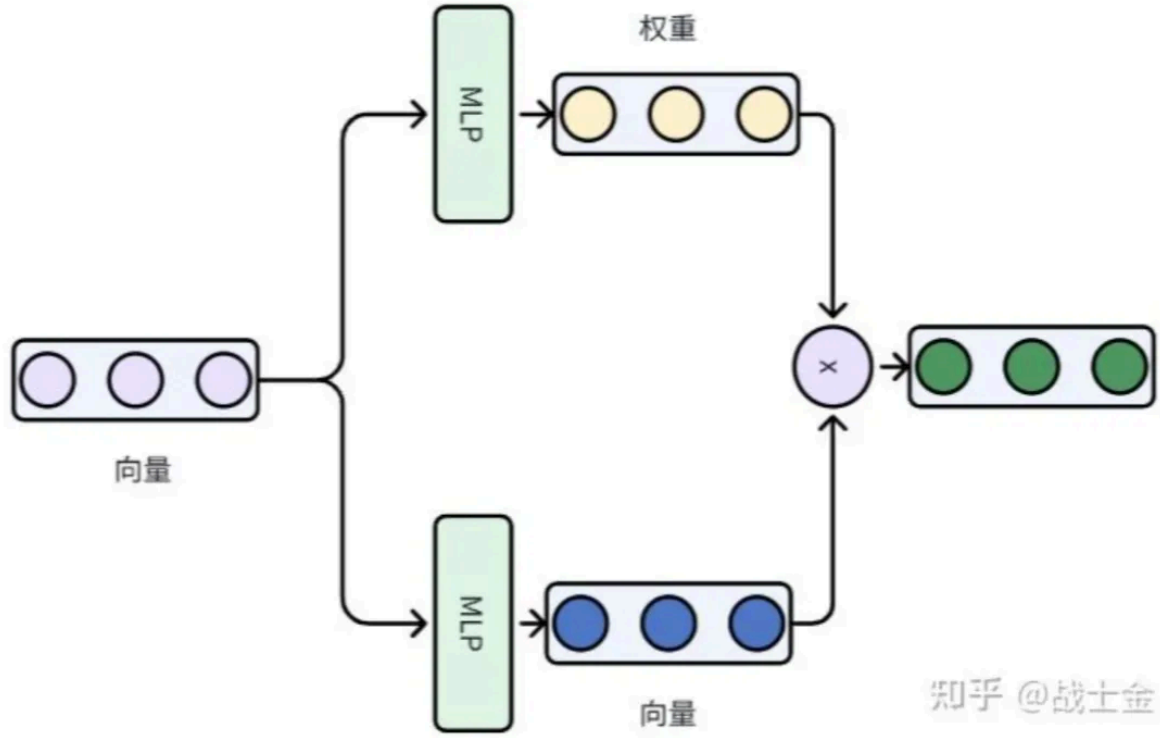
	batch size	训练效率 (tokens/s/gpu)	每个gpu显存使用量
softmoe v1	2	6588G	67G
softmoe v2	4	16300G	54G
softmoe v3	4	14100G	56G

4.2 SeNet

SENet（Squeeze-and-Excitation Networks）来自于计算机视觉领域ImageNet 2017竞赛的冠军方案，目的是通过学习的方式来获取到每个图像特征通道的重要程度，通过重要程度加强或抑制特征通道，示意图如下图所示，C表示通道维度（给不太了解图像领域的同学简单提一嘴，最原始的一个像素点需要有R、G、B三种颜色数值表示，可以理解为有3个通道。网络中间层的feature map维度由卷积核数量决定），W、H表示表示图像宽和高维度。每个通道维度的权重是一个数值。



接下来我们对原始的SENet的形式进行一下简单的变换，假设W和H都等于1，那么一个3维的特征矩阵就能用一个1维向量来表示。此时计算SENet的示意图如下所示，即将输入分别通过两个MLP变换出向量和权重，并逐位相乘。**相当于让模型自动去学习对哪些神经元进行增益或衰减。**



我们再来看一下qwen的FFN层的具体实现，如下所示。FFN层大致可以分为两层，可以发现第一层的实现和SENet的思想是一致的。

```
class Qwen2MoeMLP(nn.Module):
    def __init__(self, config, intermediate_size=None):
        super().__init__()
        self.config = config
        self.hidden_size = config.hidden_size
        self.intermediate_size = intermediate_size
        self.gate_proj = nn.Linear(self.hidden_size, self.intermediate_size,
        self.up_proj = nn.Linear(self.hidden_size, self.intermediate_size, bi
        self.down_proj = nn.Linear(self.intermediate_size, self.hidden_size,
        self.act_fn = ACT2FN[config.hidden_act]

    def forward(self, x):
        return self.down_proj(self.act_fn(self.gate_proj(x)) * self.up_proj(x
```

原始的Transformer FFN层的第一层并不是SENet，而是普通的MLP，后人将这块进行了改动应该是有一些收益的。Steel LLM选择将第二层MLP也替换成SENet，代码如下所示：

```
class SteelSENet(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.hidden_size = config.hidden_size
        self.intermediate_size = config.intermediate_size //config.mlp_div_ra
        self.gate_up_proj = nn.Linear(self.hidden_size, self.intermediate_siz
        self.up_proj = nn.Linear(self.hidden_size, self.intermediate_size, bi
        self.gate_down_proj = nn.Linear(self.intermediate_size,self.hidden_si
        self.down_proj = nn.Linear(self.intermediate_size, self.hidden_size,
        self.act_fn1 = ACT2FN[config.hidden_act]
        self.act_fn2 = ACT2FN[config.hidden_act]

    def forward(self, x):
        x = self.act_fn1(self.gate_up_proj(x)) * self.up_proj(x)
        return self.act_fn2(self.gate_down_proj(x)) * self.down_proj(x)
```

Steel LLM Soft MOE结构最终选择使用6个专家，每个专家是一个SteelSENet。

最后来一个灵魂拷问，你在结构上“猴戏”了这么多，能证明在LLM上有用吗？很遗憾，我没法证明，LLM的评估标准太难量化了，而且训练成本太高，无法做充分的消融研究。这也就是为啥主流LLM改结构的工作比较少，一是尝试成本太高，二是很难衡量是不是真的是模型结构有收益。作为从事过搜广推行业的工程师，我还是比较相信模型结构能够带来一定收益的。同时Steel LLM最终的模型效果也不会影响我的收入，所以还是比较乐意尝试一些自己的新想法的hhh

4.3 其他细节

rms norm精度问题

之前本打算用flash attention提供的rms_norm算子的，能提高一点点训练效率。但是简单测了一下，结果没法和pytorch实现的 rms norm完全对上，如下所示。虽然数值差距也不是很大，但是为了带来不必要的麻烦，还是没有用。

```
tensor([-1.1797e+00,  1.3438e+00, -3.3594e-01, -7.8125e-01, -9.9219e-01,
        -1.3672e+00,  5.5075e-05, -9.9121e-02,  1.6562e+00, -1.0234e+00],
       device='cuda:0', dtype=torch.bfloat16, grad_fn=<SliceBackward0>)
#================
tensor([-1.1797e+00,  1.3359e+00, -3.3398e-01, -7.7734e-01, -9.8828e-01,
        -1.3594e+00,  5.4836e-05, -9.8633e-02,  1.6484e+00, -1.0234e+00],
       device='cuda:0', dtype=torch.bfloat16, grad_fn=<SliceBackward0>)
```

softmax使用float32精度

在soft MOE的实现上，涉及到用softmax计算各个专家权重的环节，softmax的计算对精度要求相对高，因此将数值转换为float32后计算（模式使用bf16混合精度方式训练，因此模型的默认参数类型是bf16）。

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

从零预训练LLM 2

从零预训练LLM · 目录

下一篇 · 个人从零预训练1B LLM心路历程