MOE前馈神经网络

基本概念

MOE(Mixture of Experts)即专家混合模型，是一种特殊的神经网络架构，旨在通过结合多个不同的“专家”网络的预测来提高性能和效率。每个“专家”通常是一个前馈神经网络(feedforward neural network)，它们各自专注于数据的不同部分或者不同任务。这些专家网络的输出随后被一个门控网络(gating network)加权组合，以产生最终的输出。这也是我们将其称之为MOE前馈神经网络的原因。

核心组件

专家(前馈神经网络)模块

一个专家模块就是一个前馈神经网络。它通常由线性层、激活函数、Dropout层等组成。从网络结构上看，每个专家的网络结构可以不同，但通常都是设置为相同的。例如，在TIAOYU中，假设输入为 x ，我们这样基于 SwiGLU 神经网络架构构建专家网络:

$$y = \mathrm{Dropout}\{W_2 \cdot [\mathrm{SiLU}(W_1 \cdot x) \odot (W_3 \cdot x)]\}$$

即， x 的处理经过 2 条路径，然后合并输出：

- 第一条路径: x 经过 W_1 线性变换，然后经过 SiLU 激活函数；
- 第二条路径: x 经过 W_3 线性变换；
- 两条路径的输出逐元素相乘，再经过 W_2 线性变换，最后经过 $\mathrm{Dropout}$ 层。

这种操作有这样的优点：

- 非线性与门控机制，增强的表达能力：通过结合激活函数和门控机制来增强模型的表达能力。相比于简单的线性变换，这种组合允许网络学习更复杂的特征映射，并能动态地控制信息流，从而捕捉输入数据中的细微差异。
- 平滑的梯度特性，改进的学习动态：SiLU(也称为Swish)是一种平滑且连续可导的激活函数，相较于ReLU等激活函数，它拥有更好的梯度特性，有助于缓解深度网络中常见的梯度消失或爆炸问题，进而改善训练过程中的学习动态。
- 提升模型性能：研究表明，在许多自然语言处理任务中，使用SwiGLU的模型（如PaLM、LLaMA等）表现出了优于使用传统线性层模型的性能。这主要是因为SwiGLU能够更好地捕捉序列数据中的长距离依赖关系和复杂模式。

- 高效的计算结构：尽管SwiGLU引入了额外的乘法操作，但其整体计算架构仍然是轻量级的，特别是在现代硬件（如GPU/TPU）上执行时，这种设计非常高效，能够在不显著增加计算成本的情况下提高模型的表现力。

门控模块

对于给定的输入，门控模块可以选择性地激活某些专家而忽略其他专家，这样做的好处是可以减少计算成本。假设总的专家数量为 $expert\_num$ ，每次需要激活使用的专家数量为 $expert\_use$ 一个典型的门控模块接受一个输入张量 h (可能是某个注意力机制的输出，维度为 (b, s, d))，输出选出的专家索引 $expert\_ids$ 、专家权重 $expert\_weight$ 和辅助损失 aux_loss 。其具体计算流程如下：

- 将输入张量 h 展平为二维张量，维度为 $(b \times s, d)$ ；
- 将展平之后的 h 输入到一个线性层(输入维度为 d ，输出维度为 $expert_num$)，得到形状为 $(b \times s, expert\_num)$ 的张量；然后进一步通过一个softmax层，得到形状为 $(b \times s, expert\_num)$ 的专家得分张量。即每个token对每个专家的得分，用来决定在计算一个token时，哪些专家会被激活。
- 基于 torch.topk 函数，选出得分最高的 $expert\_use$ 个专家，得到其索引 $expert\_ids$ 和得分张量 $expert\_scores$ (形状都为 $(b \times s, expert\_use)$)。
- 根据得分张量 $expert\_scores$ 计算(比例归一化，张量的每个元素除以其所在的最后1维的总和)被选中的专家权重 $expert\_weight$ ，确保权重和为1。
- 计算辅助损失：如果是训练模式，并且辅助损失系数 $\lambda > 0$ ，则计算负载均衡损失 aux_loss 作为辅助损失。负载均衡损失会惩罚那些被过度使用的专家，鼓励更均匀的负载分布，目标是让每个专家处理的样本数量尽可能均匀。具体计算方法是：
  - 将得分张量 $scores$ 从形状 $(b \times s, expert\_num)$ 变为 $(b, s, expert\_num)$ ，便于后续计算每个序列的专家概率分布；
  - 计算每个序列的专家概率分布(沿sequence维度求平均)，结果得到形状为 $(b, expert\_num)$ 的张量；
  - 在最后一维操作，计算每个序列的L2损失(平方和)，得到形状为 (b) 的张量；
  - 对batch求平均并缩放，得到最终的负载均衡损失 aux_loss 。

  具体代码实现如下：

```
def _compute_load_balance_loss(self, scores, b, s):
    """计算基于L2范数的负载均衡损失"""
    # 将scores从(b*s, expert_num)变为(b, s, expert_num)，便于计算每个序列的损失
    scores = scores.view(b, s, -1)
    # 计算每个序列的专家概率分布（沿sequence维度平均）
    seq_probs = scores.mean(dim=1)  # shape: [b, expert_num]
    # 计算每个序列的L2损失（平方和）
    seq_losses = torch.sum(seq_probs ** 2, dim=-1)  # shape: [b]
    # 对batch求平均并缩放(expert_num和aux_loss_lambda是预定义的超参数)
    return seq_losses.mean() * expert_num * aux_loss_lambda
```

如果不是在训练模式，则辅助损失为0。

- 返回 选出的专家索引 $expert\_ids$ 、专家权重 $expert\_weight$ 和辅助损失 aux_loss 。

实现细节

MOE前馈神经网络在TIAOYU的实现细节主要包括：

- 初始化
  - 基于前面提及的专家(前馈神经网络)模块，创建 1 个共享专家——\text{Shared_expert} 。
  - 基于前面提及的专家(前馈神经网络)模块，创建 $expert\_num$ 个混合专家，并将这些专家保存在一个 nn.ModuleList 列表 —— Mixture_experts 中。
  - 基于前面提及的门控模块，创建 1 个门控网络—— Gate ，用来选择专家和计算权重、输出辅助损失。
- 前向传播
  - 假设输入为 h， 是一个形状为 (b, s, d) 的张量。
  - 首先将 h 输入到共享专家中，得到共享专家的输出: $shared\_y = \text{Shared\_expert}(h)$。
  - 然后将 h 输入到门控网络中—— $\text{Gate}(h)$ ，得到选出的专家索引 $expert\_ids$ 、专家权重 $expert\_weight$ 和辅助损失 aux_loss 。其中， $expert\_ids$ 和 $expert\_weight$ 的形状都为 $(b \times s, expert\_use)$ 。 aux_loss 被存在MOE前馈神经网络对象中，以便在反向传播时计算梯度。
  - 将输入 h 展平为一个 $(b \times s, d)$ 的张量，以便后续计算。
  - 在训练模式下
    - 将选择专家的索引 $expert\_ids$ 展平到1维，即 $(b \times s \times expert\_use)$ 的1维张量。
    - 在 0 维上将 h 重复 $expert\_use$ 次，得到形状为 $(b \times s \times expert_use, d)$ 的张量。
    - 创建一个空的张量 y ，形状与 h 相同 ($(b \times s \times expert_use, d)$)，用于存储每个专家的输出。
    - 遍历每个专家，对输入数据应用对应的专家前馈神经网络，并将结果存储到 y 中:

```
for i, expert in enumerate(Mixture_experts):
    # 对第 i 个专家，选择 expert_ids 中等于 i 的索引，将对应的输入数据传入专家前馈神经网络，
    y[expert_ids == i] = expert(h[expert_ids == i]).to(y.dtype)  # 确保类型一致
# i. y.view(*expert_weight.shape, -1) 将 y 的形状从 (b * s * expert_use, d) 变形为
# ii. expert_weight.unsqueeze(-1) 对 expert_weight 添加一个维度，变形为 (b * s, exp
# iii. y.view(*expert_weight.shape, -1) * expert_weight.unsqueeze(-1) 进行乘法操作
y = (y.view(*expert_weight.shape, -1) * expert_weight.unsqueeze(-1)).sum(dim=1)
# 恢复原始形状
y = y.view(b, s, d)
```

- 在推理模式下
  - 创建元素值全为0输出张量 y，形状与 h 相同(为 $(b \times s, d)$)，用于存储输出结果。
  - 对 h 中的 $b \times s$ 个 token 对应的嵌入向量依次进行处理，并将结果累加到 y 中:

```
for i in range(h.size(0)):
    # a. 获取当前 token 选择的专家ID和权重
    this_experts = expert_ids[i]    # expert_ids 的形状为 (b * s, expert_use), th
    this_weights = expert_weight[i] # expert_weight 的形状为 (b * s, expert_use),

    # b. 计算加权输出，对每个专家，将输入数据传入对应的专家前馈神经网络，将结果乘以对应的权重，
    for j in range(expert_use):
        # i. 获取当前专家的索引
        this_expert_id = this_experts[j].item()
        # ii. 将当前 token 对应的嵌入向量数据输入到对应的专家前馈神经网络中，得到专家的输出
        this_expert_out = self.Mixture_experts[this_expert_id](h[i].unsqueeze(0
        # iii. 同一个 token 的 expert_use 个不同专家的输出会被加权，并累加到 y 中，结果约
        y[i] += this_expert_out.squeeze(0) * this_weights[j]
```

- 将混合专家结果与共享专家层的结果相加： $y = y + shared\_y$，并返回 y。