MOE

1. MoE

1.1 什么是 MoE

MoE，全称 Mixture of Experts，混合专家模型。

什么是 MoE？从 Transformer 模型的角度来说，MoE 包含两个要素：**稀疏 MoE 层**和**门控网络**。

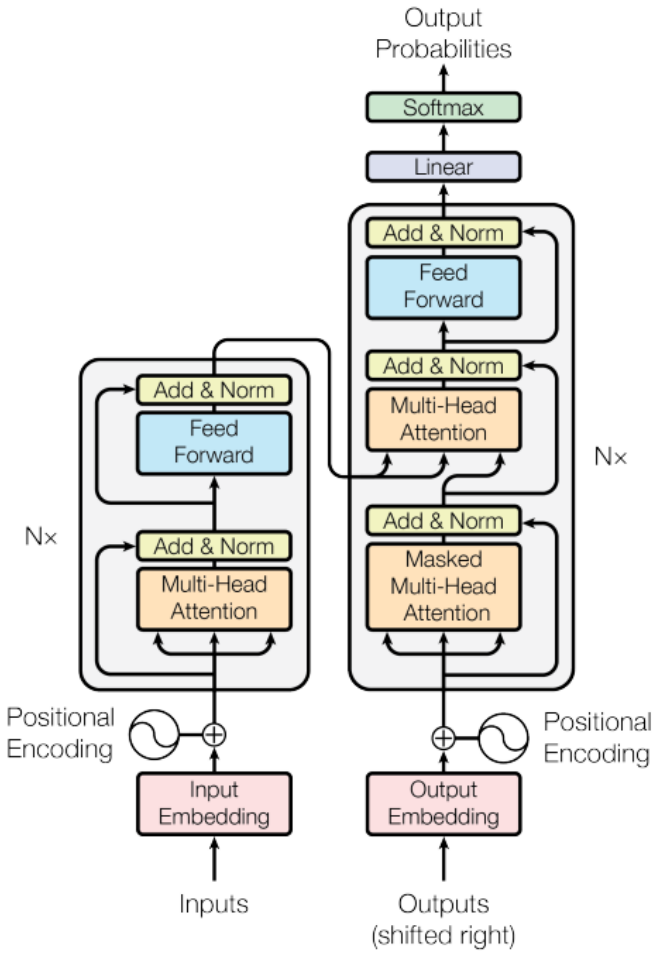
简单理解，**稀疏 MoE 层就是专家层，由 N 个专家网络构成，门控网络用于将 token 分发给不同的专家**。在 MoE 模型运行过程中，在每个时间步由门控网络选择 n 个（n < N）专家执行计算，也就是每个时间步仅使用一部分参数执行计算。

我们之前学习的 Transformer 系列模型都是**稠密模型**，在处理数据时，**所有的参数都会对所有输入数据进行处理**。

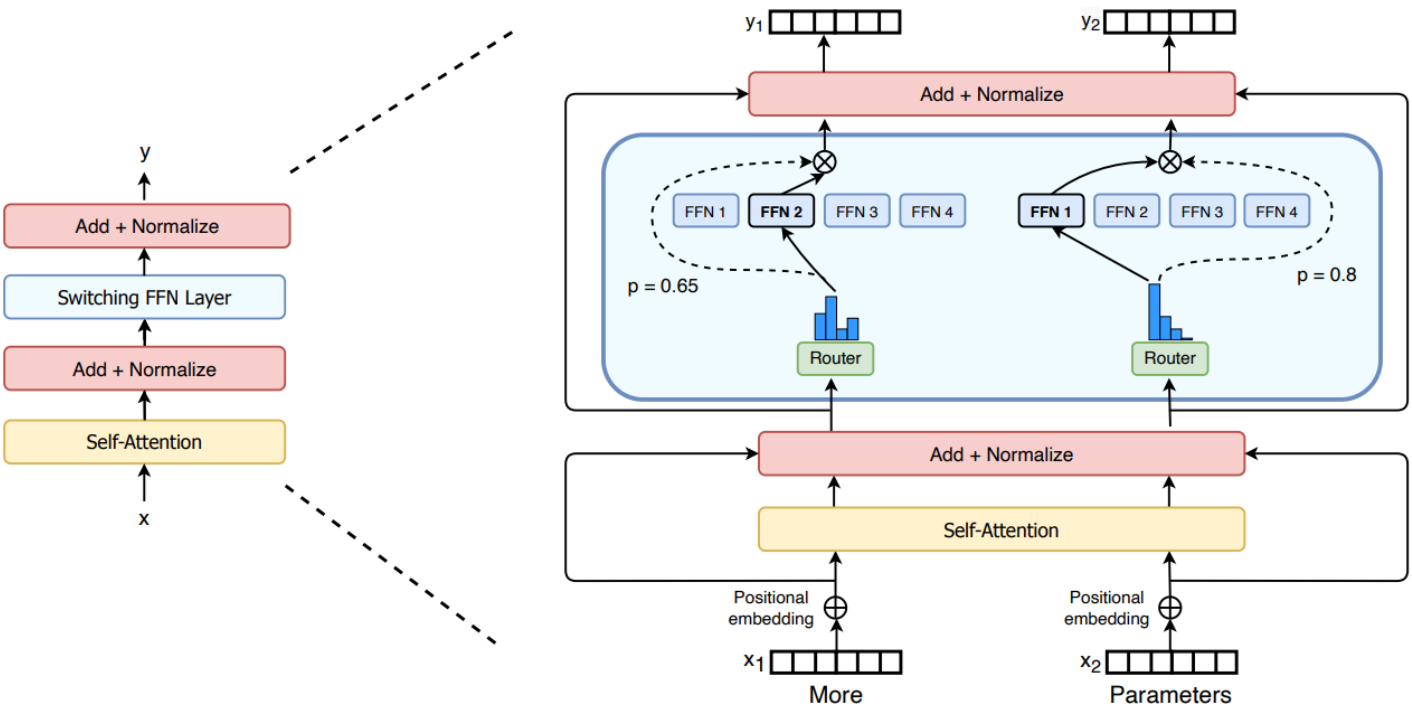
稀疏性的概念采用了**条件计算**的思想。相比于稠密模型，**稀疏性允许我们仅针对整个系统的某些特定部分执行计算**。这意味着**并非所有参数都会在处理每个输入时被激活或使用，而是根据输入的特定特征或需求，只有部分参数集合被调用和运行**。

1.2 MoE 的结构

Transformer架构：



典型MoE架构：



MoE 基于 Transformer 架构，主要由两部分组成：

- **门控网络/路由**：这个部分用于决定哪些 token 被发送到哪个专家。token 的路由方式是 MoE 使用中的一个关键点，因为路由器由学习的参数组成，并且与网络的其他部分一同进行预训练。

- **稀疏 MoE 层**：这些层代替了传统 Transformer 模型中的前馈网络 (FFN) 层。MoE 层包含若干“专家”(例如 8 个)，每个专家本身是一个独立的神经网络。在实际应用中，这些专家通常是前馈网络 (FFN)，但它们也可以是更复杂的网络结构。

1.3 不同专家传递了什么信息

MoE 通过门控网络，在每个时间步，将 input 中的 token 分发给不同的专家。

1.4 专家是否真的实现了“分工”

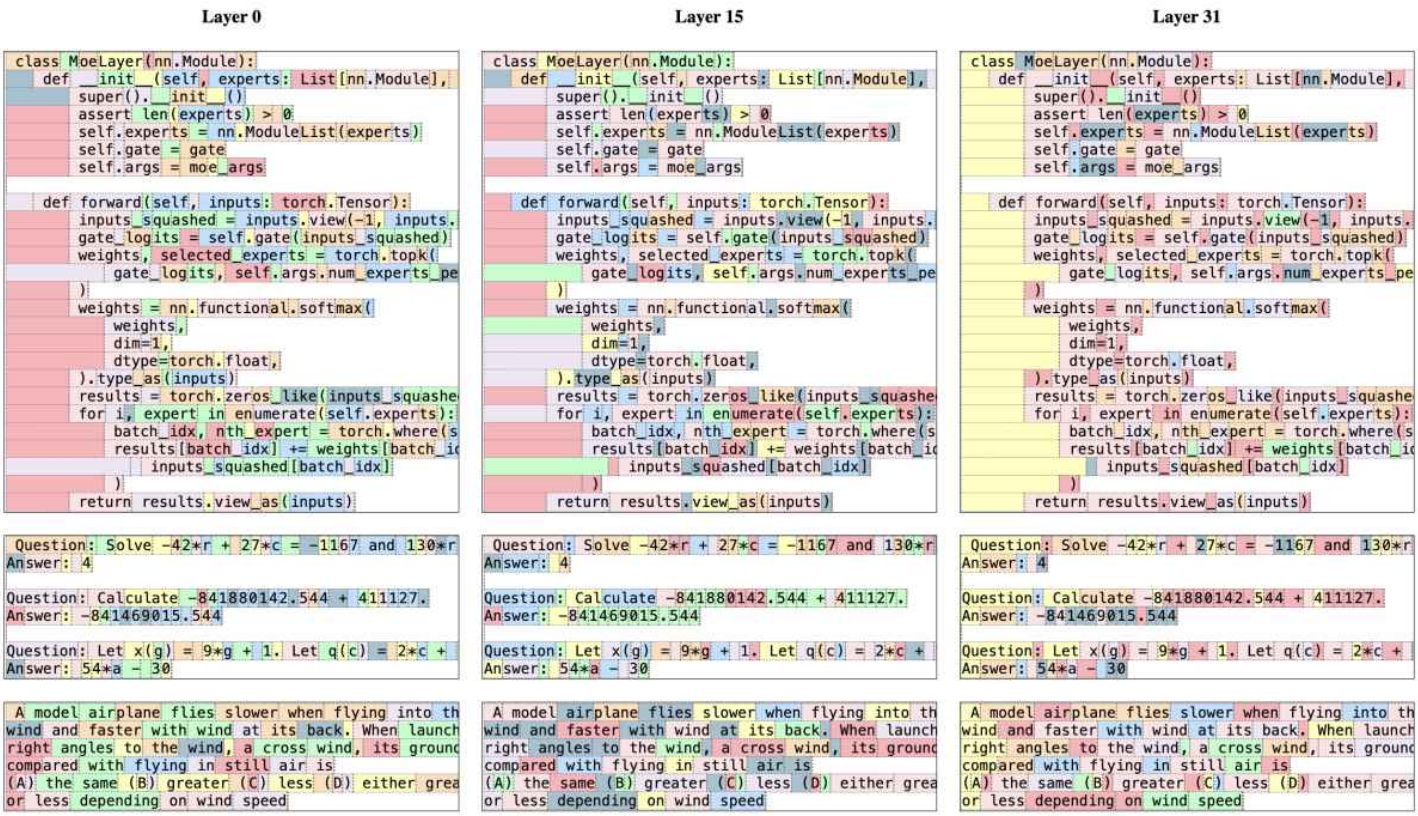
以 ST MoE 为例，在 ST MoE 的研究中，发现有专门处理标点、连词和冠词、动词、视觉描述、专有名词、计数和数字的专家，同样也有对特殊token（例如 <extra id x>）进行操作的专家。

Expert specialization	Expert position	Routed tokens
Sentinel tokens	Layer 1	been <extra_id_4><extra_id_7>floral to <extra_id_10><extra_id_12><extra_id_15> <extra_id_17><extra_id_18><extra_id_19>...
	Layer 4	<extra_id_0><extra_id_1><extra_id_2> <extra_id_4><extra_id_6><extra_id_7> <extra_id_12><extra_id_13><extra_id_14>...
	Layer 6	<extra_id_0><extra_id_4><extra_id_5> <extra_id_6><extra_id_7><extra_id_14> <extra_id_16><extra_id_17><extra_id_18>...
Punctuation	Layer 2	, , , , , , , , , - , , , , ,).)
	Layer 6	, , , , , : . : , & , & & ? & - , , ? , , , . <extra_id_27>
Conjunctions and articles	Layer 3	The the the the the the the the the The the the the the the The the the the
	Layer 6	a and and and and and and and or and a and . the the if ? a designed does been is not
Verbs	Layer 1	died falling identified fell closed left posted lost felt left said read miss place struggling falling signed died falling designed based disagree submitted develop
Visual descriptions *color, spatial position*	Layer 0	her over her know dark upper dark outer center upper blue inner yellow raw mama bright bright over open your dark blue
Proper names	Layer 1	A Mart Gr Mart Kent Med Cor Tri Ca Mart R Mart Lorraine Colin Ken Sam Ken Gr Angel A Dou Now Ga GT Q Ga C Ko C Ko Ga G
Counting and numbers *written and numerical forms*	Layer 1	after 37 19. 6. 27 I I Seven 25 4, 54 I two dead we Some 2012 who we few lower each

此外，研究者们还对 ST MoE 进行了多语言训练，尽管预期每个专家处理一种特定语言，但实际上并非如此，由于令牌路由和负载均衡的机制，没有任何专家被特定配置以专门处理某一特定语言。

1.5 专家根据什么进行“分工”

以 Mixtral 8x7B 为例，下图显示了不同领域（Python 代码、数学和英语）的文本示例，其中每个 token 都用与其所选专家相对应的背景色突出显示：



在 Mixtral 8x7B 中，**专家的选择似乎更符合语法而非领域**，尤其是在初始层和最终层。

目前，“**专家专门化**” 是MoE技术的重要研究方向，在陈丹琦团队今年5月份发布的MoE架构Lory中，通过在训练数据中分组相似文档，鼓励“专家专门化”。

2. MoE的代码实现原理

2.1 门控网络

门控网络，也称为路由，确定哪个专家网络接收来自多头注意力的 token 的输出。

2.1.1 基础门控网络

举个例子解释路由的机制，假设有 4 个专家，token 需要被路由到前 2 个专家中。首先需要通过线性层将 token 输入到门控网络中。该层将对应于 [batch_size, max_seq, model_dim] 的输入张量从 [2, 4, 32] 维度，投影到对应于[batch_size, max_seq, num_expert] 的新形状：[2, 4, 4]。其中 n_embed 是输入的通道维度，num_experts 是专家网络的计数。

```
# 定义模型超参数
batch_size=2
context_length=4
```

```
n_embed=32
# 创建随机向量表示MHA输出
mh_output = torch.randn(batch_size, context_length, n_embed)
# 定义线性变换层
topkgate_linear = nn.Linear(n_embed, num_experts)
# [batch_size, max_seq, num_expert]
logits = topkgate_linear(mh_output)
# 获取TopK的Logits和对应的Indices
top_k_logits, top_k_indices = logits.topk(top_k, dim=-1)
top_k_logits, top_k_indices
```

```
# output
(tensor([[[ 0.0246, -0.0190],
          [ 0.1991,  0.1513],
          [ 0.9749,  0.7185],
          [ 0.4406, -0.8357]],
         [[ 0.6206, -0.0503],
          [ 0.8635,  0.3784],
          [ 0.6828,  0.5972],
          [ 0.4743,  0.3420]]]),
tensor([[[2, 3],
         [2, 1],
         [3, 1],
         [2, 1]],
        [[0, 2],
         [0, 3],
         [3, 2],
         [3, 0]]])
```

获得稀疏门控的输出后，用 -inf （负无穷）填充其余部分，再使用 softmax 激活函数，目的是经过softmax处理时负无穷会被映射至零，而最大的前两个值会更加突出，且和为 1。**要求和为 1 是为了对专家输出的内容进行加权**。

```
# 使用 -inf （负无穷）填充其余部分，恢复 num_experts 维度
zeros = torch.full_like(logits, float('-inf'))
sparse_logits = zeros.scatter(-1, top_k_indices, top_k_logits)
sparse_logits
```

```
#output
tensor([[[   -inf,    -inf,  0.0246, -0.0190],
         [   -inf,  0.1513,  0.1991,    -inf],
```

```
          [   -inf,  0.7185,    -inf,  0.9749],
          [   -inf, -0.8357,  0.4406,    -inf]],
         [[ 0.6206,    -inf, -0.0503,    -inf],
          [ 0.8635,    -inf,    -inf,  0.3784],
          [   -inf,    -inf,  0.5972,  0.6828],
          [ 0.3420,    -inf,    -inf,  0.4743]]], grad_fn=<ScatterBackward0>)
```

进行softmax处理，Top-2的权重被归一化，其余专家对应权重被归0：

```
gating_output= F.softmax(sparse_logits, dim=-1)
gating_output
```

```
#ouput
tensor([[[0.0000, 0.0000, 0.5109, 0.4891],
         [0.0000, 0.4881, 0.5119, 0.0000],
         [0.0000, 0.4362, 0.0000, 0.5638],
         [0.0000, 0.2182, 0.7818, 0.0000]],
        [[0.6617, 0.0000, 0.3383, 0.0000],
         [0.6190, 0.0000, 0.0000, 0.3810],
         [0.0000, 0.0000, 0.4786, 0.5214],
         [0.4670, 0.0000, 0.0000, 0.5330]]], grad_fn=<SoftmaxBackward0>)
```

综上，基础的门控机制如下：

```
# First define the top k router module
class TopkRouter(nn.Module):
    def __init__(self, n_embed, num_experts, top_k):
        super(TopkRouter, self).__init__()
        self.top_k = top_k
        self.linear =nn.Linear(n_embed, num_experts)
    def forward(self, mh_ouput):
        # [batch_size, max_seq, model_dim] -> [batch_size, max_seq, num_expert]
        logits = self.linear(mh_output)
        # 获取 Top-K
        top_k_logits, indices = logits.topk(self.top_k, dim=-1)
        # 填充 "-inf"
        zeros = torch.full_like(logits, float('-inf'))
        sparse_logits = zeros.scatter(-1, indices, top_k_logits)
        # 生成权重
        router_output = F.softmax(sparse_logits, dim=-1)
        return router_output, indices
```

测试结果：

```
batch_size=2
context_length=4
n_embed=32
mh_output = torch.randn(batch_size, context_length, n_embd)
# 示例输出
inputtop_k_gate = TopkRouter(n_embd, num_experts, top_k)
gating_output, indices = top_k_gate(mh_output)
gating_output.shape, gating_output, indices
```

```
#output
(torch.Size([2, 4, 4]),
 tensor([[[0.5284, 0.0000, 0.4716, 0.0000],
          [0.0000, 0.4592, 0.0000, 0.5408],
          [0.0000, 0.3529, 0.0000, 0.6471],
          [0.3948, 0.0000, 0.0000, 0.6052]],
         [[0.0000, 0.5950, 0.4050, 0.0000],
          [0.4456, 0.0000, 0.5544, 0.0000],
          [0.7208, 0.0000, 0.0000, 0.2792],
          [0.0000, 0.0000, 0.5659, 0.4341]]], grad_fn=<SoftmaxBackward0>),
 tensor([[[0, 2],
          [3, 1],
          [3, 1],
          [3, 0]],
         [[1, 2],
          [2, 0],
          [0, 3],
          [2, 3]]]))
```

2.1.2 带噪声的门控网络

在通常的混合专家模型 (MoE) 训练中，**门控网络往往倾向于主要激活相同的几个专家。这种情况可能会自我加强，因为受欢迎的专家训练得更快，因此它们更容易被选择**。但是，如果所有的令牌都被发送到只有少数几个受欢迎的专家，那么训练效率将会降低。因此，为了负载平衡，从门控的线性层向 logits 激活函数添加标准正态噪声。

```
class NoisyTopkRouter(nn.Module):
    def __init__(self, n_embed, num_experts, top_k):
        super(NoisyTopkRouter, self).__init__()
```

```
        self.top_k = top_k
        self.topkroute_linear = nn.Linear(n_embed, num_experts)
        self.noise_linear =nn.Linear(n_embed, num_experts)
    def forward(self, mh_output):
        # dim -> num_experts
        logits = self.topkroute_linear(mh_output)
        # 生成noise
        noise_logits = self.noise_linear(mh_output)
        # 生成高斯白噪声
        noise = torch.randn_like(logits)*F.softplus(noise_logits)
        # 加入高斯白噪声
        noisy_logits = logits + noise
        # 获取top_k
        top_k_logits, indices = noisy_logits.topk(self.top_k, dim=-1)
        # 填充 -inf
        zeros = torch.full_like(noisy_logits, float('-inf'))
        sparse_logits = zeros.scatter(-1, indices, top_k_logits)
        # 权重结果
        router_output = F.softmax(sparse_logits, dim=-1)
        return router_output, indices
```

测试结果：

```
batch_size=2
context_length=4
n_embed=32
num_experts=8

mh_output = torch.randn(2, 4, n_embd)
# 示例输出
noisy_top_k_gate = NoisyTopkRouter(n_embd, num_experts, top_k)
gating_output, indices = noisy_top_k_gate(mh_output)
gating_output.shape, gating_output, indices
```

```
#output
(torch.Size([2, 4, 8]),
tensor([[[0.4181, 0.0000, 0.5819, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],

         [0.4693, 0.5307, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],

         [0.0000, 0.4985, 0.5015, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
```

```
          [0.0000, 0.0000, 0.0000, 0.2641, 0.0000, 0.7359, 0.0000, 0.0000]],

          [[0.0000, 0.0000, 0.0000, 0.6301, 0.0000, 0.3699, 0.0000, 0.0000],

          [0.0000, 0.0000, 0.0000, 0.4766, 0.0000, 0.0000, 0.0000, 0.5234],

          [0.0000, 0.0000, 0.0000, 0.6815, 0.0000, 0.0000, 0.3185, 0.0000],

          [0.4482, 0.5518, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]],
grad_fn=<SoftmaxBackward0>),
 tensor([[[2, 0],
          [1, 0],
          [2, 1],
          [5, 3]],
         [[3, 5],
          [7, 3],
          [3, 6],
          [1, 0]]]))
```

除了为路由增加噪声，还可以通过随机路由、专家容量、辅助损失等策略实现 token 负载均衡。

2.2 混合专家模块

2.2.1 专家定义

每个专家都是一个**简单的多层感知器**。

```
class Expert(nn.Module):
""" An MLP is a simple linear layer followed by a non-linearity i.e. each
Expert """
    def __init__(self, n_embd):
        super().__init__()
        # 创建多层容器
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
            )
    def forward(self, x):
        return self.net(x)
```

2.2.2 混合专家模块

在获得门控网络的输出结果之后，对于给定的 token，将前 k 个值选择性地与来自相应的前 k 个专家的输出相乘。这种选择性乘法的结果是一个加权和，该加权和构成 SparseMoe 模块的输出。

这个过程的关键和难点是避免不必要的乘法运算，只为前 k 名专家进行正向转播。为每个专家执行前向传播将破坏使用稀疏 MoE 的目的，因为这个过程将不再是稀疏的。

```
class SparseMoE(nn.Module):
    def __init__(self, n_embed, num_experts, top_k):
        super(SparseMoE, self).__init__()
        self.router = NoisyTopkRouter(n_embed, num_experts, top_k)
        self.experts = nn.ModuleList([Expert(n_embed) for _ in
range(num_experts)])
        self.top_k = top_k

    def forward(self, x):
        # x [4, 8, 16]
        # 每个token被分发给2个专家，所以 indices [4, 8, 2]
        # gating_output 是每个token选定专家的加权权重 [4, 8, 8]
        gating_output, indices = self.router(x)
        final_output = torch.zeros_like(x)
        # flat_x [32, 16]
        flat_x = x.view(-1, x.size(-1))
        # flat_gating_output [32, 8]
        flat_gating_output = gating_output.view(-1, gating_output.size(-1))
        # 执行专家处理逻辑
        for i, expert in enumerate(self.experts):
            # indices与当前expert的i一致的位置设置为True，代表该token会被分发给当前
expert
            # expert_mask 代表输入的所有token是否要经过该expert处理
            # expert_mask [4, 8]
            expert_mask = (indices == i).any(dim=-1)
            # 将 Mask 拉直为扁平向量（[4, 8] -> [32]）
            flat_mask = expert_mask.view(-1)
            # 如果有token需要被expert处理
            if flat_mask.any():
                # 取要被该expert处理的token
                # expert_input [N, 16]
                expert_input = flat_x[flat_mask]
                # 执行expert处理逻辑
                # expert_output [N, 16]
                expert_output = expert(expert_input)
                # 获取每个token的权重
                # gating_scores [N, 1]
                gating_scores = flat_gating_output[flat_mask, i].unsqueeze(1)
                # 计算加权值
                # weighted_output [N, 16]
```

```
                weighted_output = expert_output * gating_scores
                # 将当前加权值加入总值中
                # final_output [4, 8, 16]
                final_output[expert_mask] += weighted_output.squeeze(1)
                print('************************************************')
        return final_output
```

代码测试：

```
import torchimport torch.nn as nn
batch_size=4
context_length=8
n_embed=16

mh_output = torch.randn(batch_size, context_length, n_embd)
# 创建MoE
outputsparse_moe = SparseMoE(n_embd, num_experts, top_k)
# 获取MoE结果
final_output = sparse_moe(mh_output)
print("Shape of the final output:", final_output.shape)
```

```
Shape of the final output: torch.Size([4, 8, 16])
```

3. 总结

目前，MoE 技术在大模型中的应用还面临一些挑战，如模型训练复杂、权重分配不平衡、分布式训练成本高等。但随着技术的不断进步，这些问题将逐渐得到解决。未来，MoE 技术有望在更多领域得到应用，为人工智能的发展带来新的机遇。