MOE

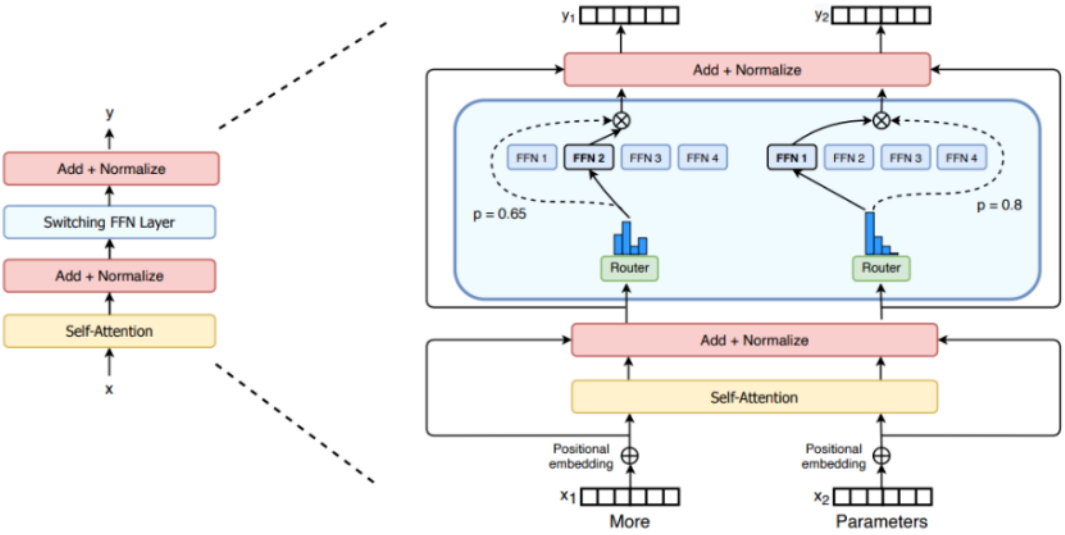
参考：Tim在路上

1 什么是MOE（混合专家模型）？

依据之前提到的scaling law，假设数据集足够大的情况下，随着模型大小的增加，性能也会逐渐增加。但这不是无限的，算力和数据目前已经成为训练更好LLM的主要瓶颈，于是，MOE出现了，**它实质上将神经网络的某些部分（通常是LLM）“分解”为不同的部分，我们将这些被分解的部分称为“专家”。**

MOE主要由两个关键部分组成:

- **专家**: 这些专家代替了传统 Transformer 模型中的前馈网络 (FFN) 层，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，甚至可以是 MoE 层本身，从而形成层级式的 MoE 结构。
- **Router（门控网络）**: 这个部分用于决定哪些token 被发送到哪个专家。例如，在下图中，“More”这个token被发送到第二个专家，而“Parameters”这个令牌被发送到第一个专家。有时，一个令牌甚至可以被发送到多个专家。令牌的路由方式是 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。



混合专家模型 (MoE) 的引入使得训练具有数千亿甚至万亿参数的模型成为可能，如开源的 1.6 万亿参数的 Switch Transformers 等。

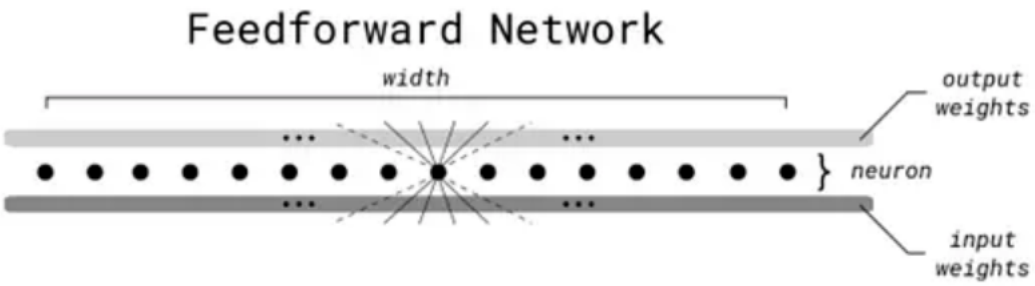
2 MOE提出的动机

- **神经网络的稀疏性：** 在特定层中，神经网络可能会变得非常稀疏，即某些神经元的激活频率远低于其他神经元。换句话说，很多神经元并非每次都会被使用到，这和人类大脑中的神经元是类似的。
- **神经元的多语义性：** 神经元的设计使其具有多语义性，这意味着它们可以同时处理多个主题或概念。比如，一个神经元可能对"苹果"、"香蕉"和"橙子"都有反应，这些词代表着不同的实体。
- **计算资源的有限性：** 模型规模是提升模型性能的关键因素之一。而不管在什么阶段，资源一定是有限的，在有限的计算资源预算下，用更少的训练步数训练一个更大的模型，往往比用更多的步数训练一个较小的模型效果更佳。

3 MOE的优点

- **计算效率**：MoE模型能够在保持参数数量巨大的情况下，以恒定的计算成本运行，因为每次前向传播只激活一部分专家。
- **预训练速度**：MoE模型在预训练阶段可以更快地达到相同的质量水平，与稠密模型相比，它们通常能够更快地完成训练。
- **推理速度**：在推理时，MoE模型由于其稀疏性，可以比具有相同参数数量的稠密模型更快地完成任务。

我们来以当今神经网络中最常见的层之一的替代方案：**前馈层 (FFN)，**来举例分析。



像 ChatGPT 这样的 LLM 由两种类型的层组成：

1. Self-Attention 注意力层。
2. FFN 前馈层。

FFN（前馈神经网络）在 Transformer 模型中的作用是将输入向量投射到更高维度的空间中，以便发掘数据中原本隐藏的细微差别。这一步骤是模型成功的关键之一，但也伴随着高昂的处理成本。

尽管FFN的作用是重要的，但根据论文的发现，它们的激活确实表现出极端的稀疏性。这意味着在处理大量参数的过程中，需要进行大量的计算。

> 事实上，根据 Meta 的说法，它们可以占 LLM 前向传递（预测）中高达 98% 的总计算量。

除此之外，虽然FFN层虽然拥有大量的参数，但在每次预测过程中，只有一小部分参数会被实际激活并参与到计算中。

这种现象揭示了FFN的一个关键特性：尽管模型构建了庞大的参数网络，实际上只有其中的一个子集对于特定的预测任务有实质性的贡献。

这种**稀疏性激活**的特点，是混合专家模型（Mixture of Experts, MoE）的核心概念。在MoE架构中，FFN层被分解为多个“专家”（experts），每个专家实际上是FFN参数的一个子集。这些专家可以根据其特定的功能被设计成处理不同类型的输入或特征。在模型进行预测时，一个路由器（router）机制会决定哪些专家将被激活并参与到当前的计算中。

这种设计的优势在于它能够显著提高模型的效率和可扩展性。由于只有相关的专家被激活，因此可以减少不必要的计算，从而加快模型的推理速度并降低运算成本。同时，这也使得模型能够更加灵活地适应不同的任务，因为不同的任务可能需要不同专家的组合来达到最优的预测效果。

此外，MoE模型还提供了一种细粒度的方式来研究和理解模型内部的工作机制。通过观察哪些专家被激活以及它们如何随着时间变化，研究人员可以更深入地洞察模型是如何学习和泛化知识，以及它是如何处理不同的输入特征的。

4 MOE存在的问题

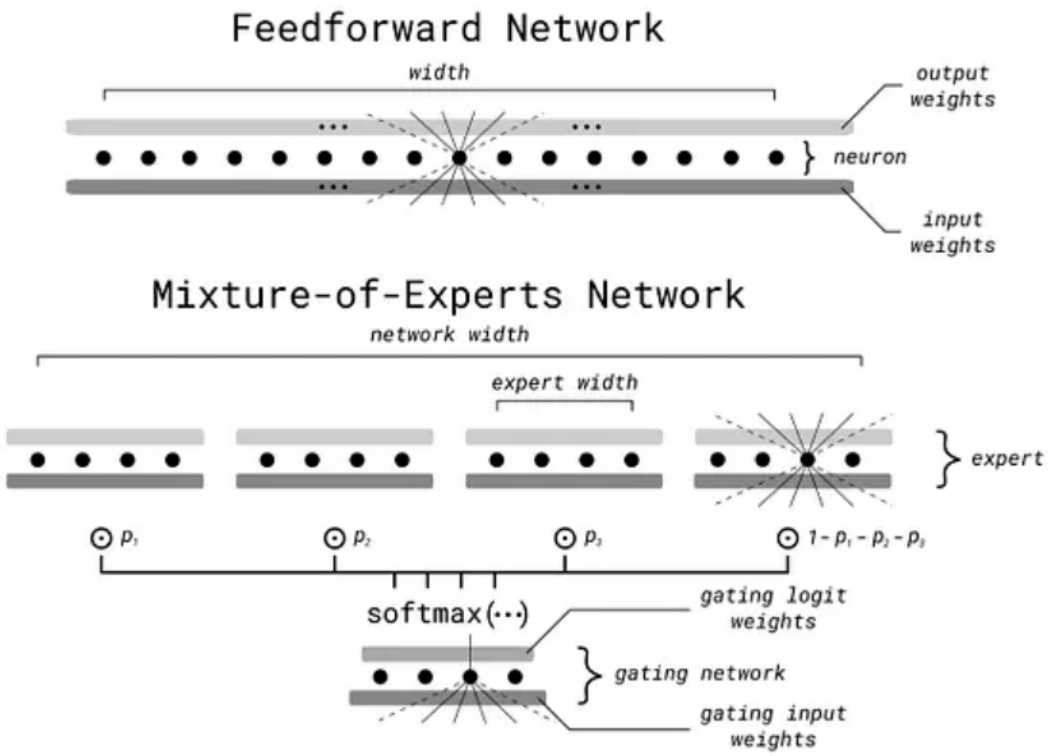
- 知识混合：由于专家数量有限，每个专家最终都要处理广泛的知识，这就产生了知识混合。这个问题的存在阻碍了专家们在特定领域的深入专业化。
- 知识冗余：当MOE模型中的不同专家学习相似的知识时，就会出现知识冗余。

Router会主动选择那些它预测会知道答案的专家。而且，由于从一开始就这样做，**不同的专家会专注于不同的主题。**

用更专业的术语来说，输入空间被“区域化”了。如果你想象一下某个 LLM 收到的“可能请求的完整空间”（对于一个大规模语言模型，这可能包括从文学到科技，从简单的问答到复杂的推理等各种任务），那么每位专家都会在各自的主题上变得更加精通。

你不必聘请一位“无所不知”的专家，而是组建一个拥有特定专业领域的团队。

因此，MoE 模型与其标准版本完全等同，但用 MoE 层替换了 FFN 层，如下所示：



> 在某些情况下，并非所有 FFN 层都被 MoE 取代，例如Jamba模型具有多个 FFN和MoE 层。

1. **输入的门控选择**：当输入数据进入模型时，首先会经过一个门控机制，该机制负责决定哪些专家将参与到当前的计算中。这个门控通常是一个softmax函数，它能够基于输入数据的特征，为每个专家计算一个概率分布，表明每个专家被激活的可能性。
2. **专家的概率分布**：以一个具有四位专家的MoE层为例，门控可能会输出如下的概率分布：[专家 1：25%，专家 2：14%，专家 3：50%，专家 4：11%]。这个分布反映了输入数据与各个专家相关性的大小，概率越高，表示该专家对于当前输入的预测任务越重要。
3. **专家的激活**：根据门控输出的概率分布，一部分专家将被选中并激活。在这个例子中，专家3和专家1因为具有较高的激活概率，将被选中参与到后续的计算中。这意味着，只有这两个专家的参数将被用于处理当前的输入数据。

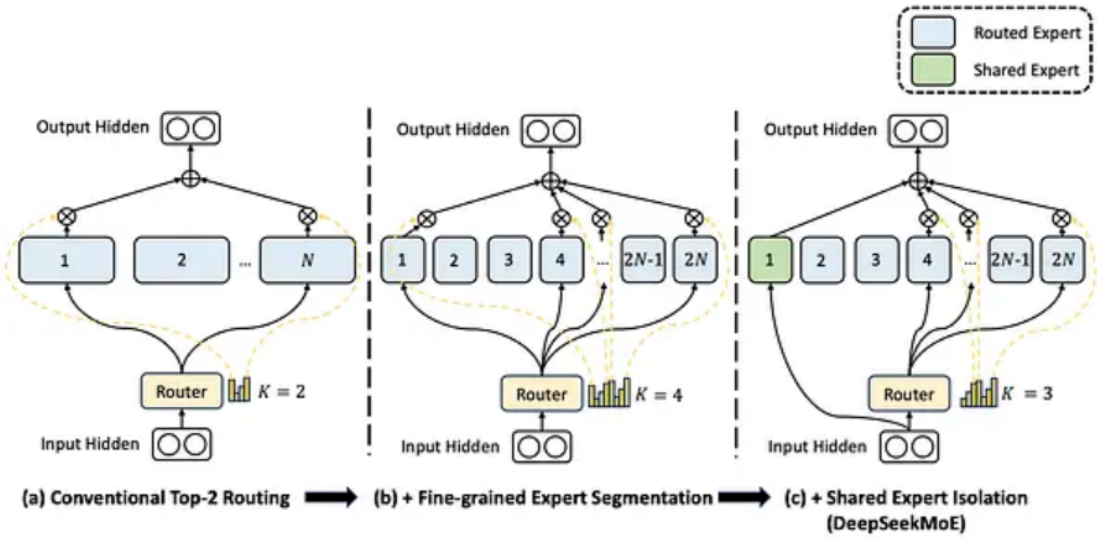
5 改进

如今，标准方法是是将模型划分为8个专家。在每次预测过程中，从这些专家中选择2个来进行计算，这样的设计旨在平衡模型的性能和计算成本。

最近一个有趣的趋势是添加“共享专家”的概念，例如DeepSeek 的DeepSeekMoE 家族。

DeepSeek-MoE 引入了“**细粒度/垂类专家**”和“**共享专家**”的概念。

1. “**细粒度/垂类专家**” 是通过细粒度专家切分 (Fine-Grained Expert Segmentation) 将一个 FFN 切分成 m 份。尽管每个专家的参数量小了，但是能够提高专家的专业水平。
2. “**共享专家**”是掌握更加泛化或公共知识的专家，从而减少每个细粒度专家中的知识冗余，共享专家的数量是固定的且总是处于被激活的状态。



这里，“k”位专家是固定的，也就是说他们始终会针对每个预测运行。这些“共享专家”或专家掌握广泛的知识，而超专业专家（对于此特定模型最多可达 64 位） 掌握更细粒度的知识。

这样，就可以“抵御”或至少最小化上述知识混合和知识冗余的问题，由于专家更加专业，知识冗余度降低，而“更广泛”的数据则由共享专家捕获。

尽管尚未完全大规模验证，但这些模型表现出非常有希望的结果，成为高于通常的 MoE 模型的标准，但现在下结论还为时过早。

6 代码

LLM-Dojo/llm_tricks/moe/make_moe_step_by_step.ipynb at main · mst272/LLM-Dojo · GitHub

MoE 一般实现：

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Expert(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Expert, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

class Gate(nn.Module):
    def __init__(self, input_size, num_experts):
        super(Gate, self).__init__()
        self.fc = nn.Linear(input_size, num_experts)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        weights = self.fc(x)
        weights = self.softmax(weights)
```

```
        return weights

class MoE(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_experts):
        super(MoE, self).__init__()
        self.experts = nn.ModuleList([Expert(input_size, hidden_size,
output_size) for _ in range(num_experts)])
        self.gate = Gate(input_size, num_experts)

    def forward(self, x):
        # 计算门控网络输出的权重
        gate_weights = self.gate(x)

        # 对每个专家进行前向传播，并根据门控网络的权重进行加权求和
        expert_outputs = [expert(x) for expert in self.experts]
        weighted_outputs = sum(gate_weight * output for gate_weight, output in
zip(gate_weights.t(), expert_outputs))

        return weighted_outputs

# 示例参数
input_size = 100
hidden_size = 200
output_size = 50
num_experts = 4

# 实例化MoE模型
model = MoE(input_size, hidden_size, output_size, num_experts)

# 假设有一个输入数据
input_data = torch.randn(1, input_size)

# 进行前向传播
output = model(input_data)
print(output)
```

对于上面这个实现，由于每个特征都以一定的权重参与了每个专家的计算，所以随着专家数量的增加，参数量和计算量都在增加。