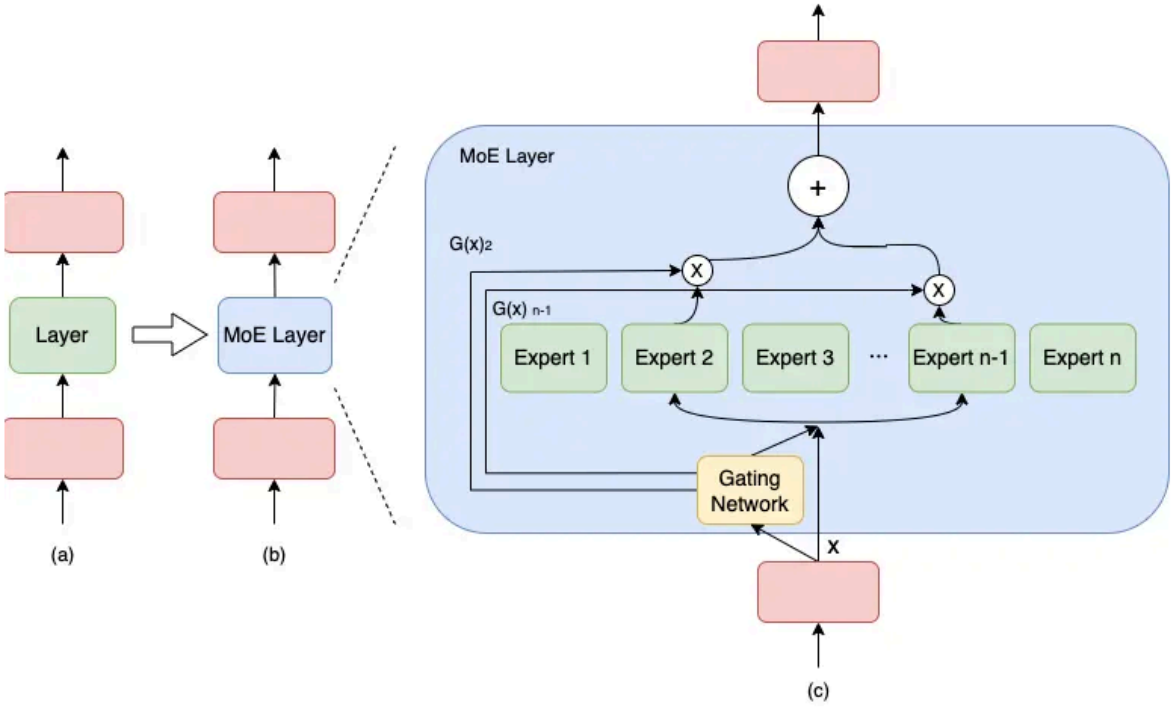
# 8.moe并行

## 1.MOE

通常来讲，模型规模的扩展会导致训练成本显著增加，计算资源的限制成为了大规模密集模型训练的瓶颈。为了解决这个问题，一种基于稀疏 MoE 层的深度学习模型架构被提出，即**将大模型拆分成多个小模型(专家，expert)， 每轮迭代根据样本决定激活一部分专家用于计算，达到了节省计算资源的效果；** 并引入可训练并确保稀疏性的门( `gate` )机制，以保证计算能力的优化。

与密集模型不同，MoE 将模型的某一层扩展为多个具有相同结构的专家网络( `expert` )，并由门( `gate` )网络决定激活哪些 `expert` 用于计算，从而实现超大规模稀疏模型的训练。

以下图为例，模型包含 3 个模型层，如(a)到(b)所示，将中间层扩展为具有 `n` 个 `expert` 的 MoE 结构，并引入 `Gating network` 和 `Top_k` 机制，MoE 细节如下图(c)所示。




公式如下：

$$\mathrm{MoE}(x) = \sum_{i=1}^{n} \left(G(x)_i \cdot E_i(x)\right)$$

$$G(x) = \mathrm{TopK}\left(\mathrm{softmax}(W_g(x) + \epsilon)\right)$$

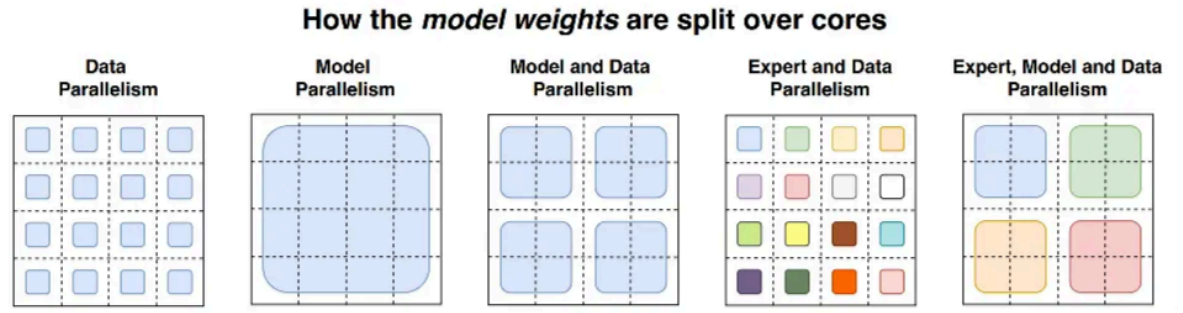
上述第一个公式表示了包含 $n$ 个专家的 $MoE$ 层的计算过程。

具体来讲：

- 首先对样本 $x$ 进行门控计算，$W$ 表示权重矩阵；
- 然后，由 $Softmax$ 处理后获得样本 $x$ 被分配到各个 $expert$ 的权重；
- 然后，只取前 $k$（通常取 $1$ 或者 $2$）个最大权重；
- 最终，整个 $MoE\ Layer$ 的计算结果就是选中的 $k$ 个专家网络输出的加权和。

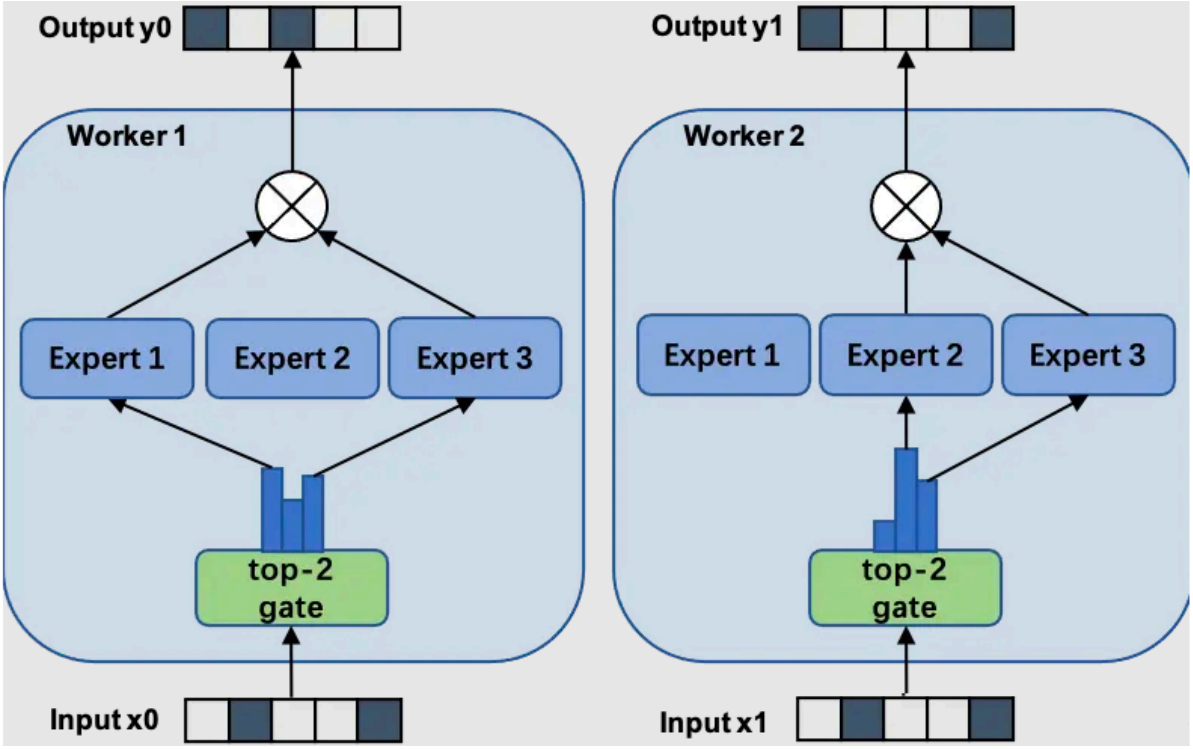
## 2.MOE分布式并行策略

上面讲述了 MOE 整体结构，下面来讲述含MOE架构的模型的分布式并行策略。




### 2.1 MOE + 数据并行

**该策略是在数据并行模式下包含MOE架构，门网络(gate)和专家网络都被复制地放置在各个运算单元上。下图展示了一个有三个专家的两路数据并行**MoE模型进行前向计算的方式。




优点：

- 实现简单，对现有代码改动小（侵入性低）
- 利用成熟的 DP 框架（如 PyTorch DDP）

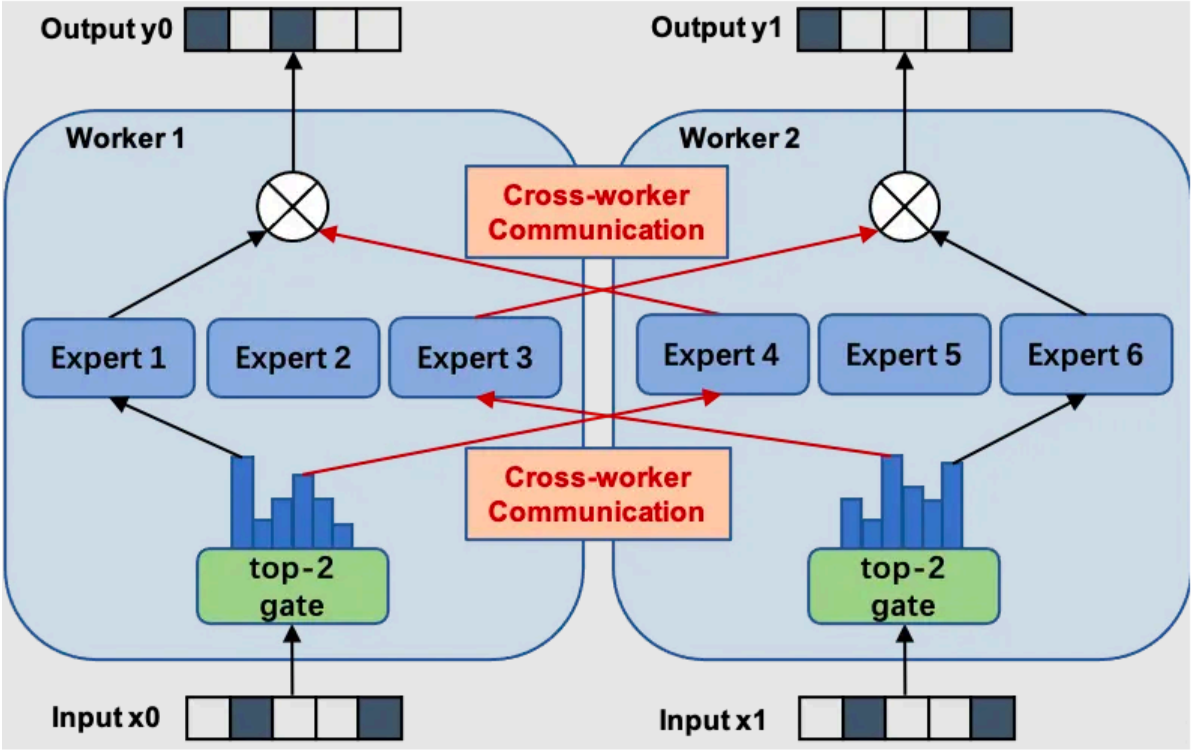
缺点：

- **内存瓶颈**：所有专家必须能放进单个 GPU 的显存中
- 专家数量受限于单卡显存，无法扩展太多专家

### 2.2 MOE + 模型并行

**该策略门网络依然是复制地被放置在每个计算单元上， 但是专家网络被独立地分别放置在各个计算单元上。**因此，需引入额外的通信操作，该策略可以允许更多的专家网络们同时被训练，而其数量限制与计算单元的数量(如：GPU数量)是正相关的。

下图展示了一个有六个专家网络的模型被两路专家并行地训练。注意：专家1-3被放置在第一个计算单元上，而专家4-6被放置在第二个计算单元上。




该模式针对不同的模型和设备拓扑需要专门的并行策略，同时会引入额外的通信，因此，相较于数据并行+MOE策略，侵入性更强。

- **数据并行**：每个设备都复制**完整模型 + 所有专家** → 显存受限
- **模型并行（针对 MoE）**：专家被**切分并分布到不同设备**，不再复制 → 可扩展更多专家，但需通信协调

除了上述两种MOE并行方案之外，还可以**MOE+数据并行+模型并行**、**MOE+ZeRO增强的数据并行**等。

**1.MoE + 数据并行 + 模型并行（Hybrid Parallelism）**

- 在多个节点内使用模型并行分布专家
- 多个节点之间使用数据并行复制非专家层或同步梯度
- 常见于大规模训练系统（如 DeepSpeed、Megatron-LM）

**2.MoE + ZeRO 增强的数据并行**

- 使用 ZeRO 技术（Zero Redundancy Optimizer）来减少数据并行中的内存冗余
- 即便专家仍复制在各卡上，但通过分片优化器状态、梯度等进一步节省显存
- 可以让更多专家"挤"进有限显存

# 3.业界大模型的MOE并行方案

## 3.1 GShard

GShard 是第一个将 MoE 的思想拓展到 Transformer 上的工作。具体的做法就是把 **Transformer 的 encoder 和 decoder 中每隔一个（every other）的FFN层，替换成 position-wise 的 MoE 层，使用的都是 Top-2 gating network。**

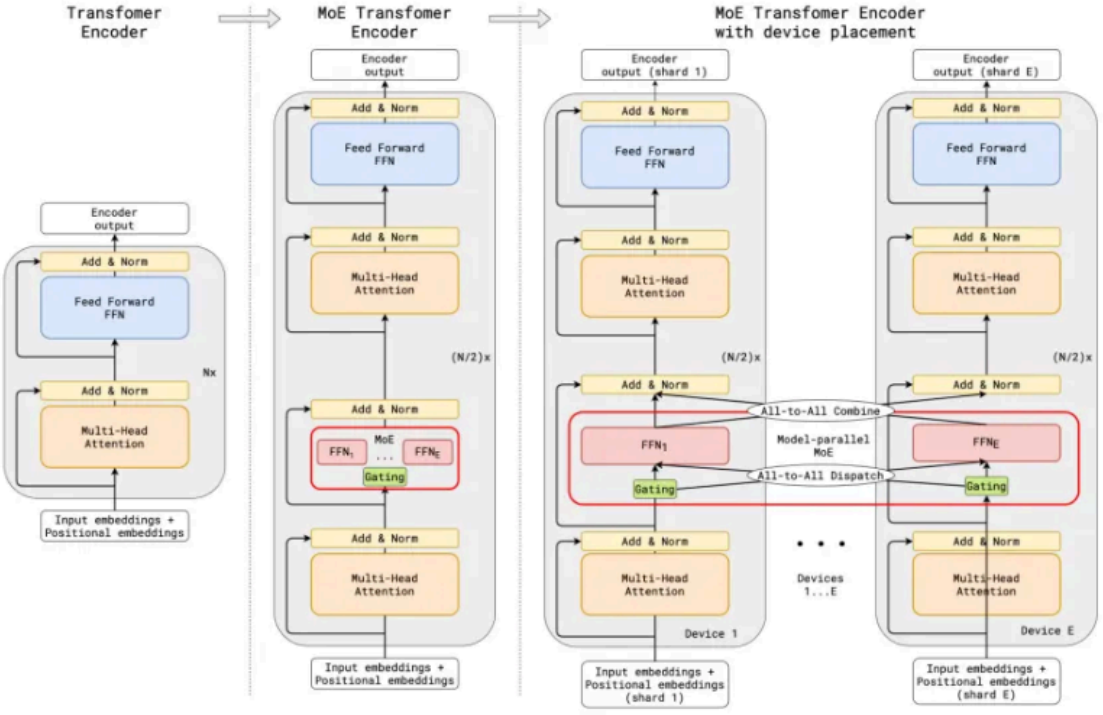



Figure 3: Illustration of scaling of Transformer Encoder with MoE Layers. The MoE layer replaces the every other Transformer feed-forward layer. Decoder modification is similar. (a) The encoder of a standard Transformer model is a stack of self-attention and feed forward layers interleaved with residual connections and layer normalization. (b) By replacing every other feed forward layer with a MoE layer, we get the model structure of the MoE Transformer Encoder. (c) When scaling to multiple devices, the MoE layer is sharded across devices, while all other layers are replicated.

此处之外，GShard还加入了很多其他设计：

- **Expert capacity balancing**：强制每个expert处理的tokens数量在一定范围内。
- **Local group dispatching**：通过把一个batch内所有的tokens分组，来实现并行化计算。
- **Auxiliary loss**：为了缓解"赢者通吃"问题，尽可能把token均分给各个专家。
- **Random routing**：在Top-2 gating的设计下，两个expert如何更高效地进行routing。

## 3.2 Switch-Transformer

Switch-Transformer 是在T5模型的基础上加入了 MoE 设计，并在C4数据集上预训练，得到了一个"又快又好"的预训练大模型。

```
# 1. 用一个小型门控网络（通常是线性层）计算每个 token 对各专家的得分
g = W_gate @ x                    # x: token 向量, W_gate: [d_model, num_experts]

# 2. 找出得分最高的那个专家（Top-1）
expert_index = argmax(g)          # shape: scalar 或 batch_size 个索引

# 3. 只将这个 token 发送给该专家进行计算
output = experts[expert_index](x)

# 4. 其他专家不参与此 token 的计算
```

虽然 Switch Transformer 采用 **Top-1 路由** 提升了计算效率，但也带来了 **专家负载不均** 的风险：

- 某些专家可能被频繁选中（过载）
- 某些专家长期闲置（欠载）

为保证训练稳定性和模型性能，Switch Transformer 引入了两个关键机制：

1. **辅助负载均衡损失（Auxiliary Load Balancing Loss）**

目标:让每个专家处理的 token 数量尽量均衡，避免“忙的忙死，闲的闲死”。

方法:在训练过程中添加一个额外的损失项，**仅用于反向传播**，以引导门控网络（gate）学会更公平地分配 token。

> 注意：该损失 **不影响前向传播**，只在训练时调节 gate 的学习方向。

数学形式（简化版）

$$\mathcal{L}_{\text{balance}} \propto \sum_{i=1}^{N} \left( \frac{\text{tokens routed to expert } i}{\text{total tokens}} \right) \times \left( \frac{\text{gates to expert } i}{\text{total gates}} \right)$$

其中：

- $N$：专家总数
- 第一项：专家 $i$ 实际处理的 token 占比（**负载**）
- 第二项：门控网络分配给专家 $i$ 的概率占比（**路由倾向**）

作用原理：

- 当某个专家同时具有高负载和高路由倾向时，损失增大
- 模型会通过梯度下降调整 gate 参数，抑制过度激活的专家
- 最终促使 routing 分布接近均匀，提升整体利用率

2. **专家容量控制（Expert Capacity）**

目标：防止某个专家被过多 token 拥挤，导致显存溢出或信息丢失。

定义容量：每个专家设置一个最大处理 token 数（即容量）：

$$\text{Capacity} = \alpha \times \frac{\text{Total tokens in batch}}{\text{Number of experts}}$$

- $\alpha$：可调超参数，通常取 `1.0 ~ 1.2`
- 容量过小 → 太多 token 被丢弃
- 容量过大 → 负载均衡效果减弱

超出容量的处理

- 超出容量的 token 将被 **丢弃** 或 **截断**
- 为减少信息损失，可通过 **重要性采样** 进行补偿（如优先保留高得分 token）

实现方式：在分布式训练中，通常结合 **All-to-All 通信** 实现 token 分发与容量限制。

> **Note**
>
> 以 2 个 GPU 为例All-to-All通信，假设：
>
> - GPU0 和 GPU1 两个设备
> - 专家分布：GPU0 负责 Expert 0 和 1，GPU1 负责 Expert 2 和 3
> - 每个 GPU 上都有一些 token（比如句子中的词）
>
> **步骤 1：路由决策（Routing）**
>
> 每个 token 经过“门控网络”判断该由哪个专家处理：

- token A → Expert 0 → 应在 GPU0
- token B → Expert 2 → 应在 GPU1
- token C → Expert 3 → 应在 GPU1
- token D → Expert 1 → 应在 GPU0

**步骤 2：本地打包**

每个 GPU 把自己手里的 token 按"目标设备"分组：

- GPU0 发现：有些 token 要去 GPU1（如 B、C）
- GPU1 发现：有些 token 要去 GPU0（如 D）

**步骤 3：All-to-All 通信开始**

- GPU0 把"给 GPU1"的数据发出去，同时接收来自 GPU1 的"给 GPU0"的数据
- GPU1 同样发送和接收

结果：

- GPU0 收到了所有应该由 Expert 0/1 处理的 token（包括原本在 GPU1 的 D）
- GPU1 收到了所有应该由 Expert 2/3 处理的 token（包括原本在 GPU0 的 B、C）

**步骤 4：本地计算**

每个 GPU 在自己的专家上处理收到的所有 token。

**步骤 5：反向还原（Backward）**

反向传播时再做一次 All-to-All，把梯度送回到这些 token 最初所在的设备。

Swith Transformer 简化了MoE的routing算法，从而大大提高了计算效率，具体如下图所示：

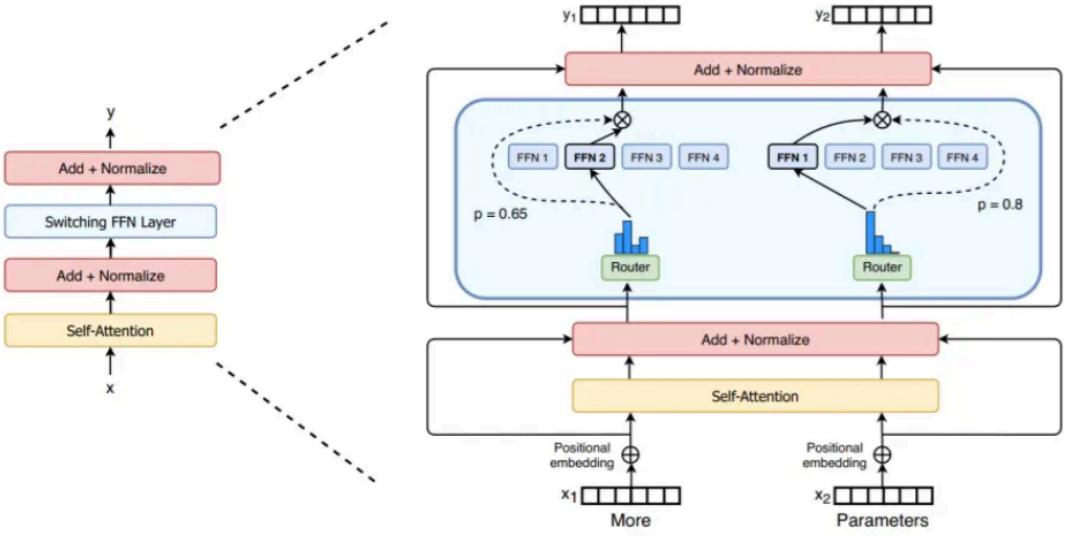



Figure 2: Illustration of a Switch Transformer encoder block. We replace the dense feed forward network (FFN) layer present in the Transformer with a sparse Switch FFN layer (light blue). The layer operates independently on the tokens in the sequence. We diagram two tokens ($x_1$ = "More" and $x_2$ = "Parameters" below) being routed (solid lines) across four FFN experts, where the router independently routes each token. The switch FFN layer returns the output of the selected FFN multiplied by the router gate value (dotted-line).

Swith Transformer 其设计的指导原则是以一种简单高效的实现方式**尽可能地把Transformer模型的参数量做大。**跟其他MoE模型的一个显著不同就是，**Switch Transformer 的 gating network 每次只 route 到 1 个 expert**，而其他的模型都是至少2个。这样就是最稀疏的MoE了，因此单单从MoE layer的计算效率上讲是最高的了。

## 3.3 GLaM

这是 Google 在2021年底推出的一个超大模型，完整的 GLaM 总共有 1.2T 参数，每个 MoE 包含 64 个专家，总共 32 个 MoE 层，但在推理期间，模型只会激活 97B 的参数，占总参数的 8%。

GLaM 的体系架构，**每个输入 token 都被动态路由到从 64 个专家网络中选择的两个专家网络中进行预测**，如下图所示。

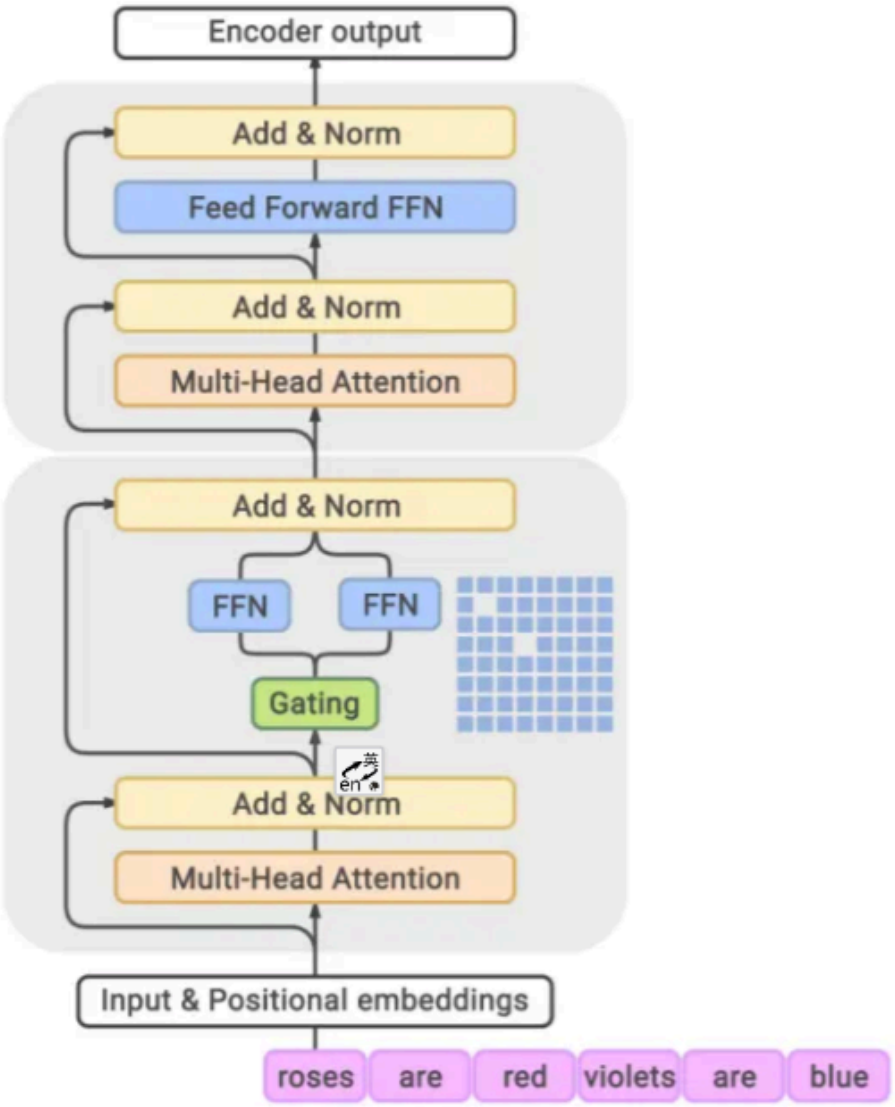



*Figure 2.* GLaM model architecture. Each MoE layer (the bottom block) is interleaved with a Transformer layer (the upper block). For each input token, *e.g.*, 'roses', the *Gating* module dynamically selects two most relevant experts out of 64, which is represented by the blue grid in the MoE layer. The weighted average of the outputs from these two experts will then be passed to the upper Transformer layer. For the next token in the input sequence, two different experts will be selected.

GLaM比GPT-3大7倍，但是由于使用了Sparse MoE的设计，训练成本却只有GPT-3的1/3，并且推理过程中的计算量减少了约一半；同时，在29个NLP任务上超越了GPT-3。

*Table 2.* A sample of related models (Devlin et al., 2019; Raffel et al., 2020; Brown et al., 2020; Lieber et al., 2021; Rae et al., 2021; Shoeybi et al., 2019; Lepikhin et al., 2021; Fedus et al., 2021) pre-trained on text corpora. $n_{\text{params}}$ is the total number of trainable model parameters, $n_{\text{act-params}}$ is the number of activated model parameters per input token.

| Model Name | Model Type | $n_{\text{params}}$ | $n_{\text{act-params}}$ |
|---|---|---|---|
| BERT | Dense Encoder-only | 340M | 340M |
| T5 | Dense Encoder-decoder | 13B | 13B |
| GPT-3 | Dense Decoder-only | 175B | 175B |
| Jurassic-1 | Dense Decoder-only | 178B | 178B |
| Gopher | Dense Decoder-only | 280B | 280B |
| Megatron-530B | Dense Decoder-only | 530B | 530B |
| GShard-M4 | MoE Encoder-decoder | 600B | 1.5B |
| Switch-C | MoE Encoder-decoder | 1.5T | 1.5B |
| GLaM (64B/64E) | MoE Decoder-only | 1.2T | 96.6B |

## 4.AI训练框架中的MOE并行训练

从 Google 发布的很多的论文和超大参数规模模型（千/万亿参数）可以看到，其基本都使用了 MOE 架构。除此之外，业界很多的AI训练框架中也继承了 MOE 并行，比如：PaddlePaddle、DeepSpeed、ColossalAI等。

### 4.1 PaddlePaddle 中的 MOE 并行

下面是一个在动态图模式下使用 PaddlePaddle 框架进行 MoE 架构的适配和训练示例。

实现流程：
Input Tokens
↓
[ Gate Network ] → Top-2 专家选择
↓
[ All-to-All ] → 跨设备分发 token
↓
[ 专家计算 ] → 各设备本地运行专家网络
↓
[ All-to-All ] → 结果返回原设备
↓
[ 加权融合 ] → 输出 MoE 结果
↓
继续前向传播...

```
# 导入需要的包
import paddle
from paddle.nn import Layer, LayerList, Linear, Dropout
from paddle.incubate.distributed.models.moe import MoELayer
```

```
from paddle.distributed.collective import Group
from paddle.distributed import fleet
import numpy as np

# 专家数
num_experts = 8

d_model = 512
d_hidden = 2048


# 封装专家层
class ExpertLayer(Layer):
    def __init__(self, d_model, d_hidden, name=None):
        super().__init__()
        self.htoh4 = Linear(d_model, d_hidden)
        self.h4toh = Linear(d_hidden, d_model)

    def forward(self, x):
        x = self.htoh4(x)
        x = self.h4toh(x)
        return x


# 初始化分布式环境，并构建 expert 通信组 moe_group
fleet.init(is_collective=True)
moe_group = paddle.distributed.new_group(list(range(fleet.worker_num())))


gate_config = {
    "type": "gshard",
    "top_k": 2,
}


experts_list = LayerList()
for expi in range(num_experts):
    exp_layer = ExpertLayer(d_model, d_hidden)
    experts_list.append(exp_layer)


# 调用 MoELayer API 封装并创建出 MoE 模型
class Model(Layer):
  def __init__(self, d_model, d_hidden, name=None):
      super().__init__()
      self.linear1 = Linear(d_model, d_model)
      self.moe_layer = MoELayer(d_model = d_model,
                                experts=experts_list,
                                gate=gate_config,
                                moe_group=moe_group,
                                recompute_interval=0)

      self.linear2 = Linear(d_model, d_model)
      self.dropout = Dropout(p=0.1)

  def forward(self, x):
```

```
        x = self.linear1(x)
        x = self.moe_layer(x)
        x = self.linear2(x)
        x = self.dropout(x)
        return x


model = Model(d_model, d_hidden)
optim = paddle.optimizer.SGD(parameters=model.parameters())

# 创建数据集，开始训练
for step in range(1, 100):
    x = paddle.rand([4, 256, d_model])

    y = model(x)
    loss = y.mean()
    loss.backward()
    optim.step()

    optim.clear_grad()

    print("=== step : {}, loss : {}".format(step, loss.numpy()))
```

### 4.2 DeepSpeed 中的 MOE 并行

DeepSpeed中也提供了对 MOE 并行的支持。目前，DeepSpeed MoE 支持五种不同的并行形式，可以同时利用GPU和CPU内存，具体如下表所示。

| 缩写 | 并行配置 | 又是 |
| --- | --- | --- |
| E | Expert | 通过增加专家数量扩展模型规模 |
| E + D | Expert + Data | 多数据并行组提升训练吞吐 |
| E + Z | Expert + ZeRO-powered data | 利用 ZeRO 分片非专家参数，支持更大基线模型 |
| E + D + M | Expert + Data + Model | 支持超大隐藏层和更大的基线模型 |
| E + D + Z | Expert + Data + ZeRO-powered data | 支持超大隐藏层和比 E+Z 更大的模型 |
| E + Z-Off + M | Expert + ZeRO-Offload + Model | 结合 GPU 和 CPU 内存，用于有限 GPU 下的大模型 |

- **E**：仅专家并行，通过增加专家数量扩展模型规模。
- **E + D**：专家并行结合数据并行，提升训练吞吐和批量处理能力。
- **E + Z**：专家并行结合 ZeRO 分片，支持更大的非专家部分（基线模型）。
- **E + D + M**：专家并行（E）、数据并行（D）与模型并行（M）三者结合，分别用于扩展专家数量、提升数据吞吐、支持超大层（如FFN）的切分。
- **E + D + Z**：专家、数据并行与 ZeRO 结合，实现高吞吐与高效显存利用。
- **E + Z-Off + M**：结合 CPU 内存卸载，在有限 GPU 下训练超大规模 MoE 模型。

下面是使用 ZeRO-Offload (stage 2) 和 DeepSpeed MOE组合的样例：

```
# MOE 模型架构
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        if args.moe:
            # MoE 层
            fc3 = nn.Linear(84, 84)
            self.moe_layer_list = []
            for n_e in args.num_experts:
                # 基于专家数创建 MOE 层
                self.moe_layer_list.append(
                    deepspeed.moe.layer.MoE(
                        hidden_size=84,
                        expert=fc3,
                        num_experts=n_e,
                        ep_size=args.ep_world_size,
                        use_residual=args.mlp_type == 'residual',
                        k=args.top_k,
                        min_capacity=args.min_capacity,
                        noisy_gate_policy=args.noisy_gate_policy))
            self.moe_layer_list = nn.ModuleList(self.moe_layer_list)
            self.fc4 = nn.Linear(84, 10)
        else:
            # 原始模型层
            self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        if args.moe:
            # 将原始 FFN 层替换成 MoE 层
            for layer in self.moe_layer_list:
                x, _, _ = layer(x)
            x = self.fc4(x)
        else:
            x = self.fc3(x)
        return x


net = Net()


# 组合 ZeRO-Offload (stage 2) 和 DeepSpeed MOE
def create_moe_param_groups(model):
```

```
    from deepspeed.moe.utils import
split_params_into_different_moe_groups_for_optimizer

    parameters = {
        'params': [p for p in model.parameters()],
        'name': 'parameters'
    }

    return split_params_into_different_moe_groups_for_optimizer(parameters)


parameters = filter(lambda p: p.requires_grad, net.parameters())
if args.moe_param_group:
    parameters = create_moe_param_groups(net)


ds_config = {
  "train_batch_size": 16,
  "steps_per_print": 2000,
  "optimizer": {
    "type": "Adam",
    "params": {
      "lr": 0.001,
      "betas": [
        0.8,
        0.999
      ],
      "eps": 1e-8,
      "weight_decay": 3e-7
    }
  },
  "scheduler": {
    "type": "WarmupLR",
    "params": {
      "warmup_min_lr": 0,
      "warmup_max_lr": 0.001,
      "warmup_num_steps": 1000
    }
  },
  "gradient_clipping": 1.0,
  "prescale_gradients": False,
  "bf16": {
      "enabled": args.dtype == "bf16"
  },
  "fp16": {
      "enabled": args.dtype == "fp16",
      "fp16_master_weights_and_grads": False,
      "loss_scale": 0,
      "loss_scale_window": 500,
      "hysteresis": 2,
      "min_loss_scale": 1,
      "initial_scale_power": 15
  },
  "wall_clock_breakdown": False,
  "zero_optimization": {
      "stage": args.stage,
```

```
        "allgather_partitions": True,
        "reduce_scatter": True,
        "allgather_bucket_size": 50000000,
        "reduce_bucket_size": 50000000,
        "overlap_comm": True,
        "contiguous_gradients": True,
        "cpu_offload": True
    }
}

# 初始化
model_engine, optimizer, trainloader, __ = deepspeed.initialize(
    args=args, model=net, model_parameters=parameters, training_data=trainset,
config=ds_config)
...
```

## 5.总结

本文简要介绍了目前业界的一些 MOE 并行方案。如果说Transformer结构使得模型突破到上亿参数量，那么稀疏 MoE 结构可以在不显著增加计算成本的情况下，使模型参数量进一步突破，达到上千亿、万亿规模。虽然，1990年左右 MOE 的概念就已经出现了；但是可以预见，MOE 将在通往AGI的道路上扮演越来越重要的角色。