

How to Train Really Large Models on Many GPUs?

原文地址: [How to Train Really Large Models on Many GPUs?](#)

关键概念: **数据并行 (Data Parallelism)** **模型并行 (Model Parallelism)** **流水线并行 (Pipeline Parallelism)** **张量并行 (Tensor Parallelism)** **混合专家模型 (Mixture-of-Experts, MoE)** **CPU 卸载 (CPU Offloading)** **激活重计算 (Activation Recomputation)** **混合精度训练 (Mixed Precision Training)** **内存高效的优化器 (Memory Efficient Optimizer)**

内容目录:

⌚ 训练并行性: 数据并行、 模型并行、 流水线并行、 张量并行

🧠 混合专家模型: 混合专家模型 Mixture-of-Experts (MoE)

💡 其他节省显存的设计: CPU 卸载、 激活重计算、 混合精度训练、 压缩策略、 内存高效优化器

近年来, 随着更大的预训练语言模型的出现, 我们在许多自然语言处理基准任务上看到了更好的结果。训练大型且深层的神经网络是一个具有挑战性的任务, 因为它需要大量的 GPU 显存和较长的训练时间。然而, 单个 GPU 工作者的显存是有限的, 许多大型模型的规模已经超出了单个 GPU 的容量。为了解决这个问题, 有若干并行训练的范式可以支持模型在多个 GPU 上的训练, 同时也有多种模型架构和内存节省设计, 使得训练非常大的神经网络成为可能。

训练并行性 (Training Parallelism)

训练超大规模神经网络模型的主要瓶颈是对 GPU 显存的大量需求, 这远远超过了单个 GPU 机器所能提供的内存。除了模型参数 (例如数十亿个浮点数) 外, 存储中间计算结果 (如梯度和优化器状态, 例如 Adam 中的动量和方差) 通常更为昂贵。

此外, 大模型训练往往配合大规模的训练语料, 因此单个进程可能需要非常长的时间才能完成。因此, 并行性是必要的。并行训练可以从多个维度进行, 包括数据、模型结构以及张量运算。

数据并行 (Data Parallelism)

最简单的**数据并行 (Data Parallelism)** 方法是将相同的模型参数复制到多个**计算节点 (worker)**, 每个节点处理部分数据, 多个节点并行执行计算。

当模型太大以至于无法放入单个 GPU 节点的显存时, 简单的数据并行方法就难以奏效。像 GeePS (Cui et al. 2016) 这样的技术会将暂时未使用的参数**卸载 (offload)** 回 CPU, 从而在 GPU 内存有限的情况下仍能训练大模型。这种数据交换应在后台完成, 不应干扰训练计算。

在每个 minibatch 结束时, 各个节点需要同步梯度或权重, 以避免模型参数的“陈旧”问题。主要有两种同步方式, 各有优劣:

- **同步并行 (BSP, Bulk Synchronous Parallel)**: 每个 minibatch 结束后, 所有节点同步梯度。这样可以防止权重陈旧, 具有良好的学习效率, 但每个节点都必须暂停并等待其他节点传递梯度。
- **异步并行 (ASP, Asynchronous Parallel)**: 每个 GPU 节点异步处理数据, 不等待也不阻塞。然而, 这种方式容易使用到过时的参数, 从而降低统计学习效率。即使计算时间更短, 训练收敛时间也未必更快。

介于两者之间的一种方式是每隔若干轮同步一次梯度 (例如每 K 轮)。在 Pytorch v1.5 (Li et al. 2021) 后, 这种方法被称为**分布式数据并行 Distribution Data Parallel (DDP)** 中的“梯度累积 (gradient accumulation)”。**分组梯度 (bucketing gradients)** 技术通过将多个梯度合并为一个 AllReduce 操作, 从而避免每个梯度立即执行 AllReduce, 提高了吞吐率。基于计算图 (computation graph), 还可以优化计算与通信的调度顺序, 其 Pytorch DDP 伪代码如下

Algorithm 1: DistributedDataParallel

```

Input: Process rank  $r$ , bucket size cap  $c$ , local model
        net
1 Function constructor(net):
2   if  $r=0$  then
3      broadcast  $net$  states to other processes
4   init buckets, allocate parameters to buckets in the
     reverse order of net.parameters()
5   for  $p$  in net.parameters() do
6      acc  $\leftarrow p$ .grad_accumulator
7      acc  $\rightarrow$  add_post_hook(autograd_hook)

8 Function forward(inp):
9   out = net(inp)
10   traverse autograd graph from out and mark unused
     parameters as ready
11   return out

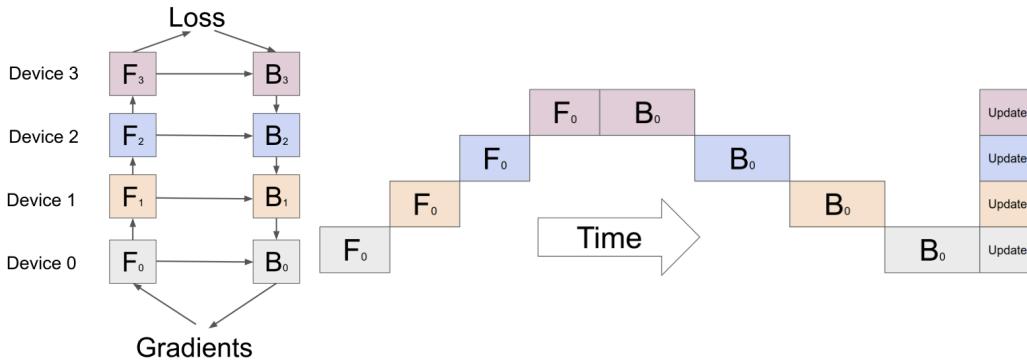
12 Function autograd_hook(param_index):
13   get bucket  $b_i$  and bucket offset using param_index
14   get parameter var using param_index
15   view  $\leftarrow b_i$ .narrow(offset, var.size())
16   view.copy_(var.grad)
17   if all grads in  $b_i$  are ready then
18      mark  $b_i$  as ready
19   launch AllReduce on ready buckets in order
20   if all buckets are ready then
21      block waiting for all AllReduce ops

```

模型并行 (Model Parallelism)

模型并行 (Model Parallelism) 旨在解决模型权重无法放入单个节点时的训练问题。其核心思想是将计算和模型参数划分到多个机器上进行分布式处理。与数据并行不同，数据并行中每个工作节点都保有整个模型的一份完整副本，而模型并行则只在每个工作节点上分配模型参数的一部分，从而减少了每个节点的显存占用和计算负担。

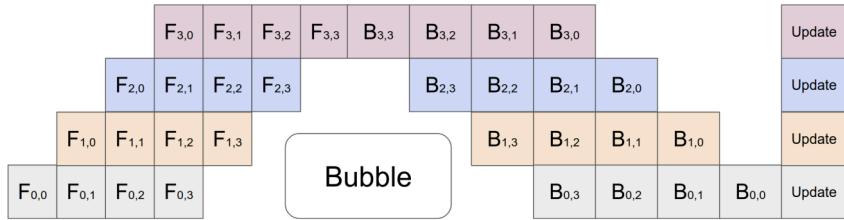
由于深度神经网络通常由一系列垂直堆叠的层组成，按层划分模型似乎是最直接的方式：将一小段连续的层划为一个分区，并放置在一个工作节点上执行。然而，如果将每个数据批次依次传递给多个有顺序依赖的工作节点，且实现方式较为朴素，那么这种串行依赖将导致大量的空泡 (bubbles) 等待时间，从而严重浪费计算资源，造成资源利用率低下。示意图 (Huang et al. 2019) 如下：



流水线并行 (Pipeline Parallelism)

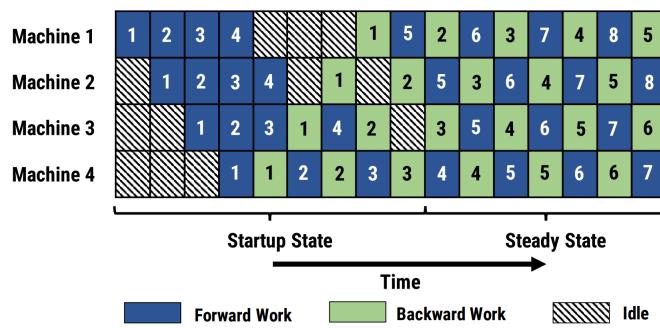
流水线并行 (Pipeline Parallelism) 结合了模型并行和数据并行的优点，旨在减少训练过程中的低效“空泡 (bubble)”时间。其主要思想是将一个小批量 (mini-batch) 划分为多个微批次 (micro-batch)，使每个阶段的工作节点可以同时处理一个微批次。需要注意的是，每个微批次都需经过一次前向传播和一次反向传播。节点间通信仅涉及激活值 (前向) 和梯度 (反向) 的传递。具体的调度策略以及梯度聚合方式在不同方法中存在差异。划分的节点数 (即工作节点数量) 通常被称为流水线深度 (pipeline depth)。

在 *GPipe* (Huang et al. 2019) 中，来自多个微批次的梯度会在每个批次结束时进行同步聚合并更新。该同步梯度下降策略可在不同工作节点数量下保持学习过程的一致性与效率。如图 3 所示，虽然仍存在空泡，但相比于模型串行执行已经显著减少。若将一个批次平均划分为 m 个微批次，并使用 d 个分区 (即工作节点)，且假设每个微批次的前向和反向传播均需 1 个时间单位，则空泡占比为： $(d - 1)/(m + d - 1)$ 。



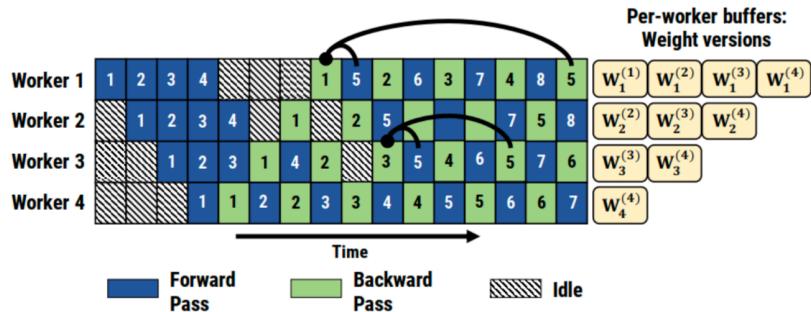
GPipe 论文指出，如果微批次数量大于分区数的 4 倍（且采用激活重计算activation recompute机制），空泡开销几乎可以忽略不计。GPipe 在设备数量增加时几乎可以实现线性吞吐提升，但前提是模型参数在各个工作节点间分布较均匀，否则可能无法达到理想加速比。

PipeDream (Narayanan et al. 2019) 提出了一种不同的调度方式：每个工作节点交替执行前向与反向传播操作（称为1F1B策略）。PipeDream将每个模型分区称为一个“阶段(stage)”，且每个阶段可以包含多个副本以实现数据并行。在任务分配过程中，PipeDream 使用确定性的轮询(round-robin)策略，在多个副本之间均衡负载，并确保同一个微批次的前向与反向传播在同一副本上执行。



由于 PipeDream 不进行整个批次结束后的全局梯度同步，若直接使用 1F1B 机制，很容易导致一个微批次的前向与反向传播使用不同版本的模型权重，从而降低学习效率。为了解决这一问题，PipeDream 设计了以下机制：

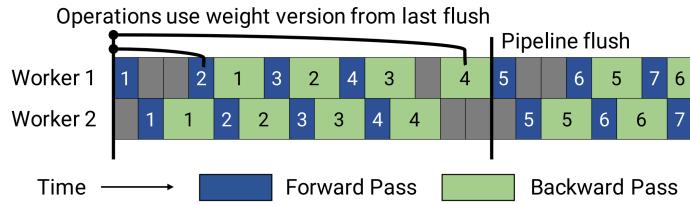
- **权重缓存 (Weight Stashing)**：每个工作节点保存多个版本的模型参数，并确保同一数据批次的前向和反向传播使用相同版本的权重。



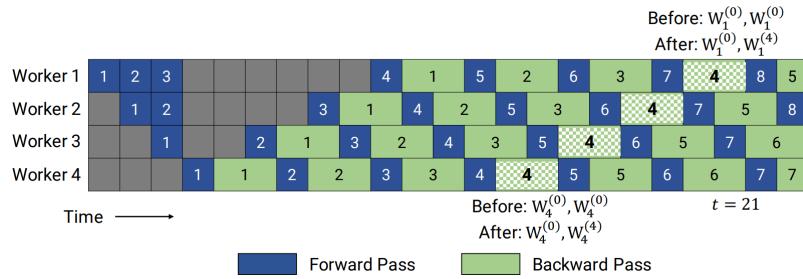
- **垂直同步 (Vertical Sync)**：模型权重的版本信息与激活值和梯度一起在阶段之间传递，后续计算采用从前一节点传播来的相应版本。这一过程保持了不同工作节点间的版本一致性，但它是异步的，与 GPipe 不同。

在训练开始前，PipeDream 首先对模型中每一层的显存使用与计算时间进行分析，然后通过动态规划求解模型层划分为阶段的最优方案。PipeDream 后续提出了两个变种(Narayanan et al. 2021)，以减少由缓存模型版本带来的内存开销。

PipeDream-flush 通过引入类似 GPipe 的全局同步流水线清空(pipeline flush)机制，在训练过程中周期性地执行同步操作。通过这一策略，只需维护一份模型权重副本，即可显著降低内存占用，但代价是略微牺牲了一些吞吐量。示意图如下：



PipeDream-2BW 仅保留两个版本的模型权重，“2BW”是“双缓冲权重 (double-buffered weights)”的缩写。该方法每处理 k 个微批次时生成一个新的模型版本，并要求 k 大于流水线深度 d ，即 $k > d$ 。由于有些尚未完成的反向传播仍依赖旧版本权重，因此新版本模型不能立刻替代旧版本。但整个过程中只需同时保存两个模型副本，从而大幅降低了内存开销。示意图如下：



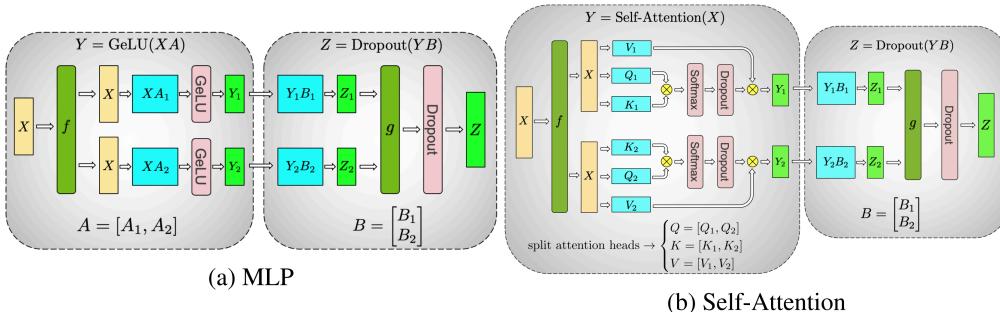
张量并行 (Tensor Parallelism)

模型并行 (Model Parallelism) 和流水线并行 (Pipeline Parallelism) 都将模型按垂直方向切分。另一方面，我们也可以将一次张量操作的计算按水平方向分布在多个设备上，这被称为 [张量并行 \(Tensor Parallelism, TP\)](#)。

以 Transformer 为例 (因其在 NLP 中的广泛应用)。Transformer 模型主要由多层的 MLP 和自注意力 (Self-Attention) 模块组成。[Megatron-LM \(Shoeybi et al. 2020\)](#) 提出了一种简洁的方法来并行化 MLP 和自注意力内部的计算。在 Transformer 中，一个 MLP 层包括一个 GEMM (通用矩阵乘法) 操作，随后是非线性的 GeLU 激活函数。我们可以将权重矩阵按列进行切分：

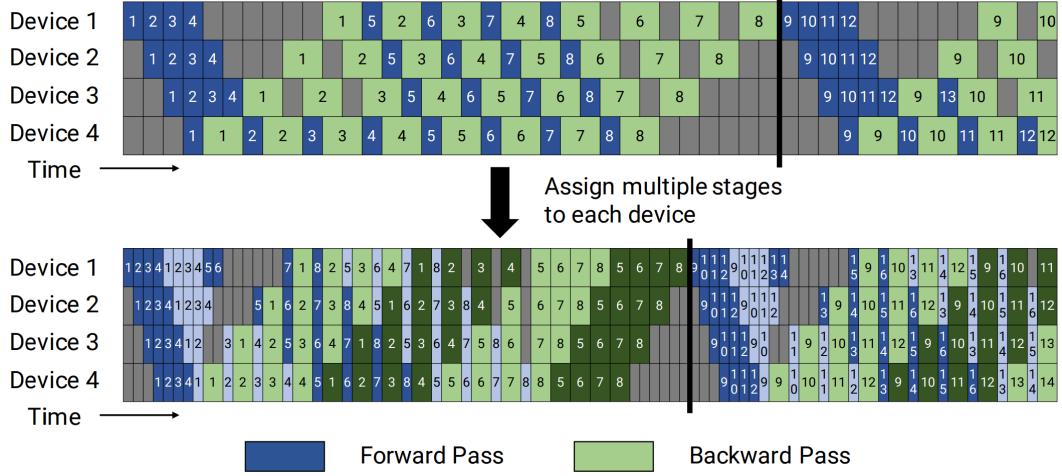
$$\begin{aligned} \text{Split } A &= [A_1, A_2], \\ Y &= \text{GeLU}(XA) \\ [Y_1, Y_2] &= [\text{GeLU}(XA_1), \text{GeLU}(XA_2)] \end{aligned}$$

在注意力模块中，对 query、key 和 value 的权重进行如上并行切分，并执行对应的 GEMM 操作，随后再通过另一个 GEMM 将它们组合为注意力头 (attention head) 输出：



[Narayanan et al. \(2021\)](#) 进一步将流水线并行、张量并行和数据并行结合起来，并提出了一种新的流水线调度策略，称为 [PTD-P \(Pipeline-Tensor-Data Parallelism\)](#)。

与传统方法将一段连续的层 (称为“模型块 model chunk”) 分配给一个设备不同，PTD-P 允许每个设备拥有多个较小的连续模型块。例如：device 1 has layers 1, 2, 9, 10; device 2 has layers 3, 4, 11, 12; each has two model chunks. 此外，为了保持调度一致性，一个 batch 中的 microbatches 数必须能被设备数整除。若每个设备拥有 v 个模型块，则相比 GPipe 的调度方式，流水线中的空泡时间 (pipeline bubble) 可以被最多减少 v 倍。以下展示一个 [交错式流水线策略 \(Interleaved Pipeline Schedule\)](#) 的例子 [Megatron-LM](#)。



混合专家模型 (Mixture-of-Experts, MoE)

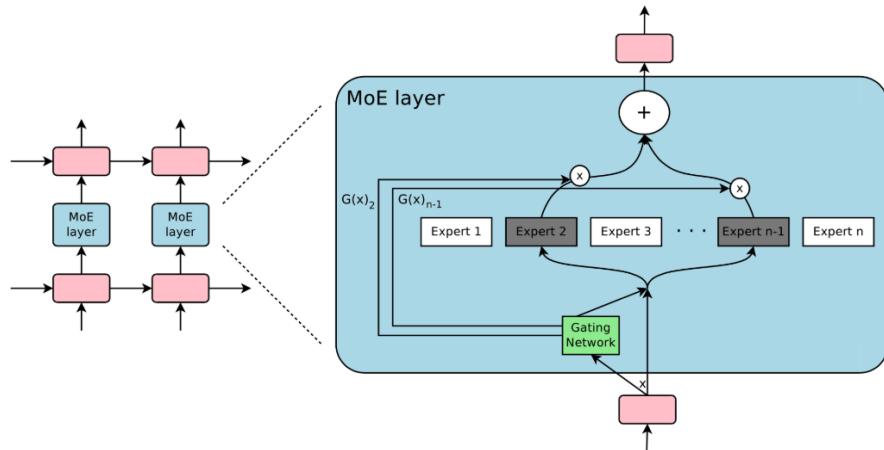
近年来，随着研究人员（主要来自 Google）尝试突破模型规模的极限，混合专家模型（MoE）引起了广泛关注。其核心思想源于集成学习（ensembling learning）：多个弱学习器的组合可以构成一个强学习器！

在一个深度神经网络内部，MoE 可以通过一个门控机制 (gating mechanism) 来实现专家集成(Shazeer et al., 2017)。门控机制控制网络中的哪一部分（即哪些专家）被激活参与输出。论文将这种结构命名为“稀疏门控混合专家层”(sparsely gated mixture-of-experts layer)。

一个 MoE 层通常包含：

- n 个前馈神经网络 (feed-forward networks) 作为专家 $\{E_i\}$;
 - 一个可训练的门控网络 (gating network) G , 用于学习一个对 n 个专家的概率分布, 从而将输入路由 (route) 到部分专家。

根据门控输出，并非所有专家都需要被评估。当专家数量太大时，可以采用二级层次化 MoE ([hierarchical MoE](#)) 以减少计算。以下提供一个简单的例子 [Shazeer et al., 2017](#):



门控函数 G 的一个简单实现方式是：将输入与一个可训练矩阵 W_g 相乘后进行 softmax，即：
 $G(x) = \text{softmax}(xW_g)$ 但这将产生一个稠密的门控向量，仍然会激活所有专家，无法节省计算资源。因此，MoE 只保留前 k 个最大的值，并引入可调高斯噪声来促进负载均衡，这种机制被称为 **noisy top- k gating**。

$$H^{(i)}(x) = (xW_g)^{(i)} + \epsilon \cdot \text{softplus}((xW_{noise})^{(i)}); \epsilon \sim \mathcal{N}(0, 1)$$

$$\text{topk}^{(i)}(v, k) = \begin{cases} v^{(i)} & \text{if } v^{(i)} \text{ is in the top } k \text{ elements of } v \\ -\infty & \text{otherwise} \end{cases}$$

为避免门控网络总是偏向少数“强专家”而忽视其他专家，Shazeer et al. (2017) 提出引入一个额外的基于重要性的损失函数（importance loss）作为软约束，鼓励所有专家获得类似的激活频率。该损失等价于每个专家在 batch 中的平均使用率的变异系数 coefficient of variation 平方：

$$L_{\text{aux}} = w_{\text{aux}} \cdot \text{CV}(\sum_{x \in X} G(x))^2$$

由于每个专家仅看到一部分训练样本（被称为“The shrinking batch problem”），因此在 MoE 中应尽可能使用较大的 batch size。但这会受到 GPU 显存限制的制约。可通过数据并行（DP）和模型并行（MP）来提升吞吐量。

GShard (Lepikhin et al., 2020) 通过 shard（分片）机制将 MoE Transformer 模型扩展到了 6000 亿参数的规模。其做法是将每隔一层的 FFN 替换为 MoE 层，并仅对这些 MoE 层在多机之间做分片，其余层则直接复制。

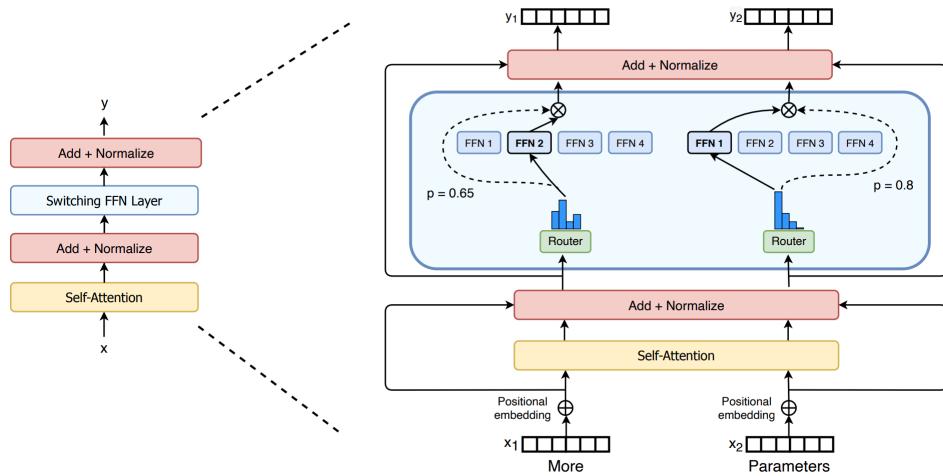
GShard 中对门控函数 G 进行了若干改进设计：

1. **专家容量 (Expert capacity)**：每个专家允许接收的 token 数不能超过设定阈值，超出部分会被标记为“溢出 (overflowed)”，对应的门控输出被设为零向量。
2. **局部部分组调度 (Local group dispatching)**：将 token 平均划分为多个本地组 (local groups)，专家容量在组级别进行控制。
3. **辅助损失 (Auxiliary loss)**：与原始 MoE 中的 loss 设计相似，GShard 也引入辅助损失，最小化各专家处理样本比例的均方差，以促进负载均衡。
4. **随机路由 (Random routing)**：第二好的专家按其权重概率随机选取，以增强训练的多样性与探索性。

Switch Transformer 将模型参数规模扩展到了数万亿级别 (trillions)，其关键做法是将原始的稠密前馈层 (FFN) 替换为稀疏的 Switch FFN 层 (sparse switch FFN layer)，其中每个输入只会被路由到一个专家网络 (top-1 routing)。Switch Transformer 同样引入了辅助损失函数以平衡专家负载。具体公式如下：

$$\text{loss}_{\text{aux}} = w_{\text{aux}} \sum_{i=1}^n f_i p_i,$$

其中专家数为 n , f_i 是路由到专家 i 的 token 比例, p_i 为路由到专家 i 的门控预测概率。结构图 Fedus et al. 2021 如下：



为了提升训练稳定性，Switch Transformer 引入了以下几个关键设计：

1. 选择性精度 (Selective precision)

将模型中只有路由器函数部分转为 FP32 精度进行计算，其余保持 FP16。这种做法在不显著增加通信开销的情况下提升了数值稳定性。

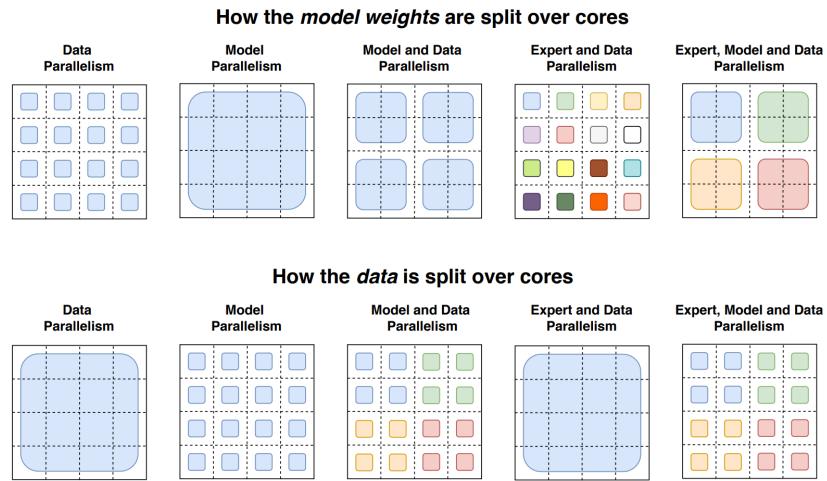
2. 更小的参数初始化 (Smaller initialization)

所有权重矩阵从截断正态分布中采样，均值为 0，标准差为 $\sigma = \sqrt{s/n}$ 。同时建议将 Transformer 初始化缩放参数从 $s = 1$ 降低到 $s = 0.1$ 。

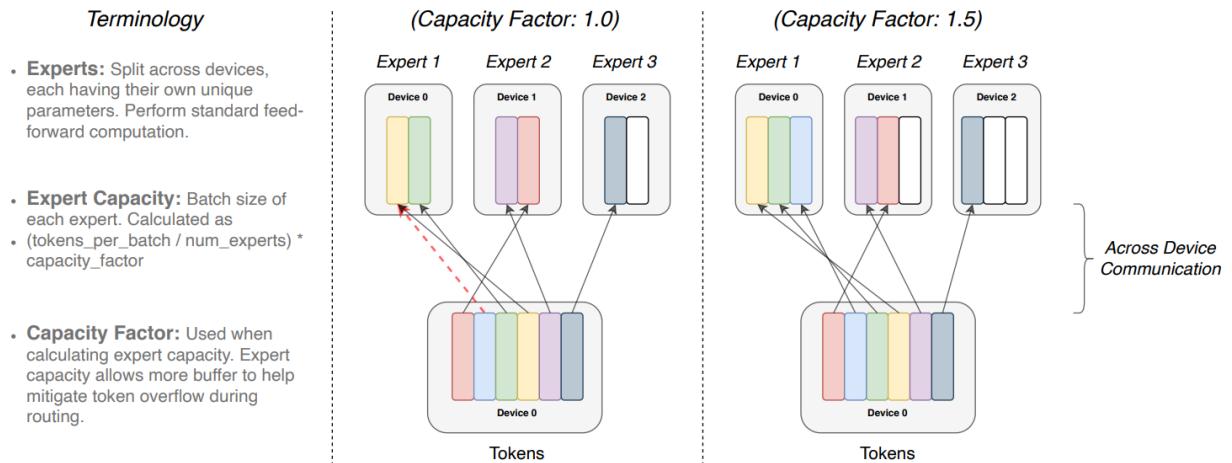
3. 更高的专家 dropout (Higher expert dropout)

在微调阶段通常数据较少，容易过拟合。作者建议仅在专家 FFN 中使用更高的 dropout (如 0.4)，而其他层则保持较低 (如 0.1)。他们发现对所有层同时增加 dropout 会导致性能下降。

Switch Transformer 的并行策略直观总结如下：



GShard 的 top-2 路由和 Switch Transformer 的 top-1 路由都依赖于 token 的选择，即每个 token 会选择最优的一到两个专家进行路由。这两种方法都引入了辅助损失函数，以鼓励更平衡的负载分配，但这并不能保证最佳的训练性能。此外，专家容量限制 (expert capacity limit) 可能导致 token 被浪费：如果某个专家已达到其容量上限，被路由到该专家的 token 会被丢弃，无法参与计算。expert capacity 的直观示意图如下：



因此，实际中会权衡利弊设置合适的capacity Factor，并且当一些专家真的分配超过超过容量Token的时候，那些 Token 将会被忽略，直接通过残差进入下一层。

Export Choice (EC) (Zhou et al. 2022) 与 GShard 的 top-2 或 Switch Transformer 的 top-1 路由机制不同，Expert Choice (EC) 则由专家来选择 token，即每个专家选择 top-k 个 token，从而确保每个专家负载固定，并允许每个 token 被多个专家处理。这种方法可以实现完美的负载均衡，并在训练中达到 2 倍收敛速度提升。

给定 e 个专家以及输入矩阵 $X \in \mathbb{R}^{n \times d}$, token-to-expert 亲和度分数计算为 $S = \text{softmax}(XW_g)$, 其中 $W_g \in \mathbb{R}^{d \times e}$, $S \in \mathbb{R}^{n \times e}$. token-to-expert 分配通过三个矩阵表示 $I, G \in \mathbb{R}^{e \times k \times n}$ 以及 $P \in \mathbb{R}^{e \times k \times n}$. $I(i, j)$ 标记 token 是第 i 个专家的第 j 个选择。 G 所选 token 的存储路由权重。 P 是 I 的 one-hot 版本，用于提供门控FFN层的输入矩阵 $PX \in \mathbb{R}^{e \times k \times d}$. 即 $G, I = \text{topk}(S^T, k)$, $P = \text{onehot}(I)$ 。

EC探索了一种正则，其通过限制每个token的最大专家数实现，形式化为：

$$\begin{aligned} & \max_A \langle S^T, A \rangle + \lambda H(A) \\ \text{s.t. } & \forall i : \sum_{j'} A[i, j'] = k, \forall j : \sum_{i'} A[i', j] \leq b, \forall i, j : 0 \leq A[i, j] \leq 1 \end{aligned}$$

其中 $A[i, j]$ 标记是否 token 是第 i 个专家的第 j 个选择。这个约束求解是一个非平凡问题。论文采用了 Dykstra 算法，通过迭代方式近似求解。尽管引入该约束略微降低了微调精度，但总体收敛与表现仍然优越。

参数 k 通过 $k = nc/e$ 确定，其中 n 是一个 batch 中的 token 的数量， c 是 capacity factor 即 token 平均要分配给多少个专家。作者主要使用 $c = 2$ ，即平均每个 token 由两个专家处理，但即使使用 $c = 1$ （即每 token 只由一个专家处理），其效果也仍然超过 top-1 gating。不过，EC 的一个主要缺点是它无法应用于 batch size 较小的情况，也不适用于自回归文本生成任务，因为 EC 需要看完整的 token batch 来决定如何分配专家。

其他节省显存的设计

CPU Offloading (CPU 卸载)

当 GPU 显存已满时，一种选择是将暂时不使用的数据卸载到 CPU 内存中，待后续需要时再读取回来 (Rhu et al. 2016)。CPU 卸载的思路非常直接，但由于它会显著拖慢训练速度，因此近年来逐渐不再流行。

激活重计算 (Activation Recomputation)

激活重计算 Activation Recomputation (also known as “activation checkpointing” or “gradient checkpointing”; Chen et al. 2016) 是一种用计算换内存的巧妙而简单的方法。它能将训练一个 ℓ 层神经网络的内存开销从 $O(\ell)$ 降低到 $O(\sqrt{\ell})$ ，代价是在每个 batch 中多进行一次前向传播。

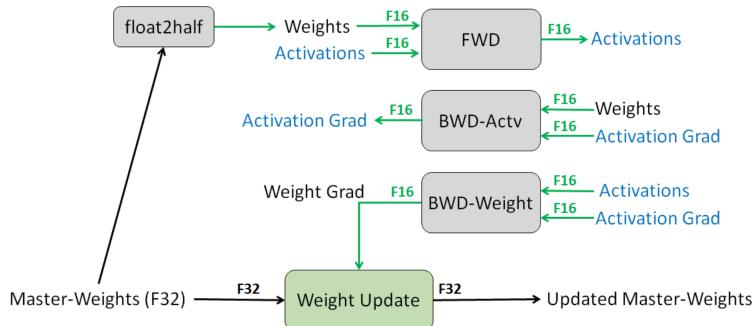
假设我们将一个 ℓ 层的网络平均分为 d 个分段。我们只保存每个分段边界处的激活，并在后向传播时重新计算中间层的激活（因为仍然需要它们来计算梯度）。在这种设置下，训练的内存成本 $M(\ell)$ 为：

$$M(\ell) = \max_{i=1, \dots, k} \underbrace{\text{cost-of-one-partition}(i)}_{\text{cost of back-propagation on the } i\text{-th partition}} + \underbrace{O(d)}_{\text{store intermediate outputs}} = O\left(\frac{\ell}{d}\right) + O(d)$$

在 $d = \sqrt{\ell}$ 时，达到最小开销 $O(\sqrt{\ell})$ 。激活重计算技术可以使内存开销相对于模型规模呈次线性增长。

混合精度训练 (Mixed Precision Training)

Narang & Micikevicius et al. (2018) 提出了一种使用半精度浮点数 (FP16) 进行模型训练的方法，在不损失模型精度的前提下降低显存消耗。直观示意图如下：



避免因 FP16 精度不足而损失关键信息，设计了三种关键技术：

- 保持全精度权重副本 (Full-precision master copy of weights):** 维护一份 FP32 的权重副本用于梯度累积，前向和反向传播使用转换后的 FP16 权重。这样可以避免梯度更新过小时在 FP16 表示范围内“变为 0”的问题。

2. **损失缩放 (Loss Scaling)**：将 loss 缩放到较大值以提升小梯度的数值稳定性。这可以将梯度值映射到 FP16 的更高精度区域，避免被截断或舍弃。
3. **运算精度控制 (Arithmetic precision)**: 对于常见的张量计算（如点积、求和），中间结果在 FP32 中累加，最终结果再转成 FP16 保存；对于逐点操作 (point-wise ops)，则可选择使用 FP16 或 FP32 执行。

文中的实验显示，loss scaling 并非总是必需。例如，对于图像分类、Faster R-CNN 等模型，loss scaling 不起明显作用；但对于 Multibox SSD、大型 LSTM 等模型则是必不可少的。

压缩策略 (Compression)

中间结果往往占用大量内存，尽管它们只在前向传播和反向传播中各被使用一次。这两次使用之间存在明显的时间间隔。因此，[Jain et al. \(2018\)](#) 提出了一种数据编码策略：在前向传播中首次使用之后，对中间结果进行压缩，在反向传播中再次使用前再解码还原。

他们提出的系统 [Gist](#) 包含：(1) 两种编码方案：层特定的无损压缩 (*Layer-specific lossless encoding*) 针对 ReLU-Pooling 层采用“二值化 (Binarize)”方法，针对 ReLU-Conv 层使用 “稀疏存储 Sparse storage + 稠密计算 dense computation” 策略；(2) 激进的有损压缩 (*Aggressive lossy encoding*)；(3) 引入延迟精度降低 (Delayed Precision Reduction, DPR)。实验表明，Gist 能够在五个主流图像分类网络上将内存占用平均减少约 1.8 倍（最高可达 2 倍），仅带来 4% 的训练开销。

内存高效的优化器 (Memory Efficient Optimizer)

优化器往往是内存的“重灾区”。以广泛使用的 Adam 优化器为例：它内部需要维护动量项和方差项，这两个变量的规模都与模型参数和梯度相同。这样一来，训练时的内存需求瞬间膨胀到模型权重的 4 倍。

为此，研究者提出了一些内存友好的优化器：[Adafactor \(Shazeer et al. 2018\)](#)：与 Adam 不同，Adafactor 不保存完整的动量和方差信息，而是仅追踪行和列方向上的移动平均，从而估计二阶矩；[SM3 \(Anil et al. 2019\)](#)：提出了一种新的自适应优化方法，也大幅降低了内存消耗。此外，零阶冗余优化器 [ZeRO \(Zero Redundancy Optimizer; Rajbhandari et al. 2019\)](#) 专门针对大模型训练中的以下两大内存瓶颈提出了系统化解决方案：

1. **模型状态 (model states) 占用**：包括优化器状态（如 Adam 的动量和方差）、梯度以及参数。混合精度训练会进一步扩大这一部分的内存需求，因为它通常需要同时保存 FP16 和 FP32 两个版本；
2. **残余状态 (residual states) 占用**：如激活值、临时缓冲区，以及由于内存碎片造成的空间浪费。

ZeRO 的设计包括两个核心组件：

- **ZeRO-DP** (数据并行优化)：将优化器状态、梯度和参数在不同的数据并行进程之间划分，采用动态通信调度策略最小化通信开销；
- **ZeRO-R** (残余内存优化)：通过划分式激活重计算、常量大小缓冲区、以及运行时内存碎片整理，来减少残余状态的内存占用。