

How to Train Really Large Models on Many GPUs?

原文地址：[How to Train Really Large Models on Many GPUs?](#) 本文为其中文（修改）翻译版本

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 GPU 显存和较长的训练时间。然而，单个 GPU 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需要大 GPU 显存，这远远超过了单个 GPU 机器所能提供，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、模型以及张量运算几个层面。

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 显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 完成后，为了保持多个工作节点间的模型一致性，**各节点需同步梯度或权重参数**，以避免出现所谓的“模型参数陈旧”问题 (stale weights)。根据同步的时机和策略，主要有以下两种方式：

- **同步并行 (BSP, Bulk Synchronous Parallel)**：每个 minibatch 结束后，所有节点同步梯度。这种方式可以最大限度地避免参数陈旧，具备良好的学习效率。但它的缺点是：所有节点都需等待最慢节点完成，容易造成阻塞。
- **异步并行 (ASP, Asynchronous Parallel)**：每个节点独立计算并更新模型，不等待其他节点，通信异步进行。这种方式提高了设备利用率，但由于可能使用“过时”的参数，统计效率下降、训练收敛变慢是常见代价。

此外，PyTorch (Li et al. 2021) 自 v1.5 起提供的 **分布式数据并行 (DDP, Distributed Data Parallel)** 支持一种更灵活的策略：**梯度累积 (Gradient Accumulation)**。也就是说，梯度不会在每个 iteration 后立刻同步，而是累计若干轮后再执行 AllReduce 操作。这样可以在保持训练稳定性的同时，减少通信开销，提高整体吞吐率。

为进一步优化性能，DDP 还引入了 **分组梯度 (Bucketing Gradients)** 技术。该方法将多个梯度合并为一个大张量进行 AllReduce，避免频繁通信，从而提升训练效率。下图展示了 PyTorch DDP 的伪代码逻辑：

Algorithm 1: DistributedDataParallel

Input: Process rank r , bucket size cap c , local model net

```

1 Function constructor( $net$ ):
2   if  $r=0$  then
3      $\downarrow$  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  $out$ 

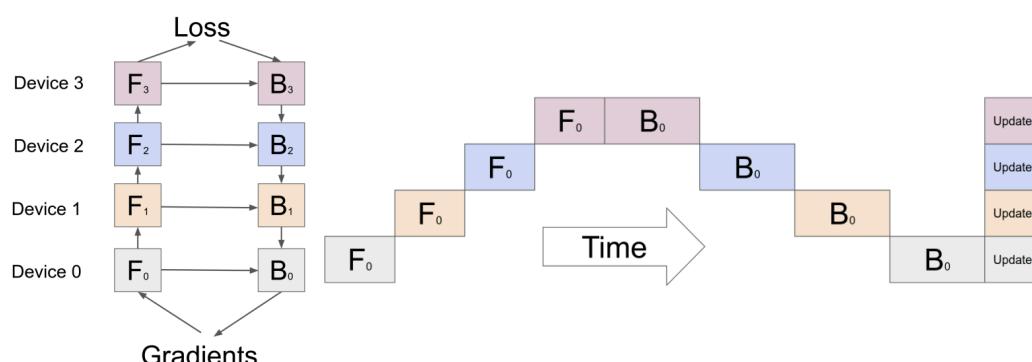
12 Function autograd_hook( $param\_index$ ):
13   get bucket  $b_i$  and bucket  $offset$  using  $param\_index$ 
14   get parameter  $var$  using  $param\_index$ 
15    $view \leftarrow b_i.narrow(offset, var.size())$ 
16    $view.copy_(var.grad)$ 
17   if all grads in  $b_i$  are ready then
18    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

```

模型并行 (Model Parallelism)

模型并行 (Model Parallelism) 主要用于解决模型权重无法完整加载到单个 GPU 节点显存中的问题。其核心思路是将模型参数和计算任务切分后分配至多个设备，协同完成训练过程。与数据并行不同，数据并行要求每个节点都保存模型的完整副本；而在模型并行中，每个节点只负责模型的一部分，从而有效降低了单个设备的显存压力和计算负载。

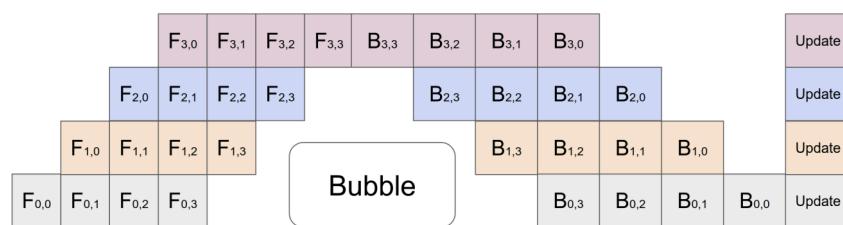
由于深度神经网络往往是由多层结构按序堆叠构成，最自然的划分方式是按层分区：将若干连续的层划分为一个阶段 (stage)，分别分配到不同的工作节点上。然而，如果采用顺序执行的朴素实现方式——即每个数据批次依次通过各个阶段——将会引发明显的空泡 (bubble) 问题：部分设备处于等待状态，无法并行执行，从而造成资源利用率的严重下降。以下是相关示意图 (Huang et al. 2019)：在朴素模型并行策略下，前向传播 (Forward) 与反向传播 (Backward) 在多设备上的执行过程。每个设备仅负责模型的一部分（如 F_0 、 F_1 、 F_2 、 F_3 表示前向的不同层）；每个 mini-batch 的数据必须顺序地经过每个设备执行前向，再依次反向；因为存在设备间的强依赖关系，导致**大量空闲等待**（如图中空白部分），称为 **bubble**（气泡）。



流水线并行 (Pipeline Parallelism)

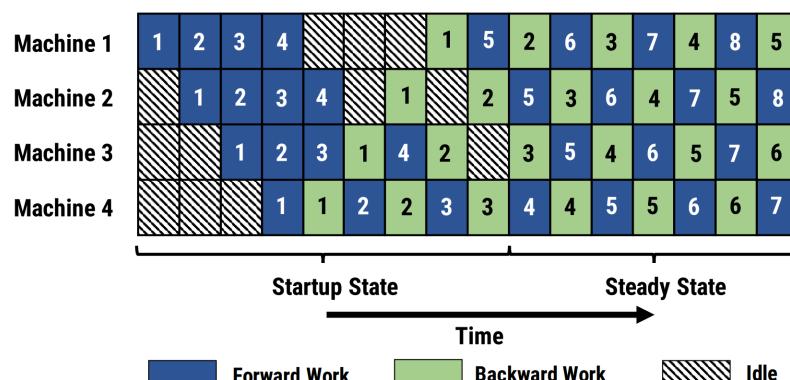
流水线并行 (Pipeline Parallelism) 结合了模型并行和数据并行的优势，目标是减少模型并行中因串行依赖所造成的资源空闲 (bubble) 时间，从而提高训练吞吐量。其核心思想是将一个 mini-batch 划分为多个更小的微批次 (micro-batch)，并以流水线方式同时在多个模型分区上并发处理这些微批次。每个微批次都需依次完成一次前向传播和一次反向传播，设备之间仅交换激活 (forward) 和梯度 (backward)。在这种机制下，每个工作节点可以接收前一阶段激活、执行计算、再将结果传给后一阶段，从而尽可能填满流水线。划分的节点数通常也被称为流水线深度 (pipeline depth)。不同流水线框架在调度策略、同步机制与梯度聚合方式上存在显著差异。

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。如下图所示，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： $(d - 1)/(m + d - 1)$ 。



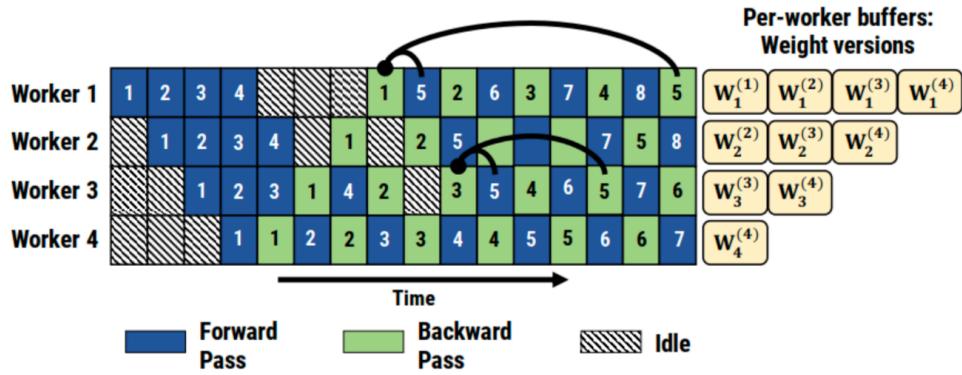
GPipe 论文指出，如果微批次数量大于分区数的 4 倍且启用了激活重计算 (activation recomputation) 机制时，空泡开销几乎可以忽略不计。此条件下，设备数量增加也能接近线性地提升训练吞吐量。不过，为达到良好的加速比，前提是模型参数在各个分区之间划分均匀。

PipeDream (Narayanan et al. 2019) 则采用一种**异步调度机制**，称为 **1F1B (One Forward, One Backward)** 策略：每个工作节点在训练过程中交替执行一个前向和一个反向操作，形成持续性的计算-通信流。PipeDream 还将每个模型分区视作一个阶段 (stage)，每个阶段可部署多个副本以实现数据并行。任务调度采用轮询 (round-robin) 分发，并确保每个微批次的前向和反向操作始终运行在同一副本上，提升缓存命中率和负载均衡。



由于 PipeDream 不会等待整个 mini-batch 结束再统一同步梯度，因此会出现前向和反向传播使用了**不同版本的模型权重**的问题，进而影响收敛性。为此，它引入两项机制：

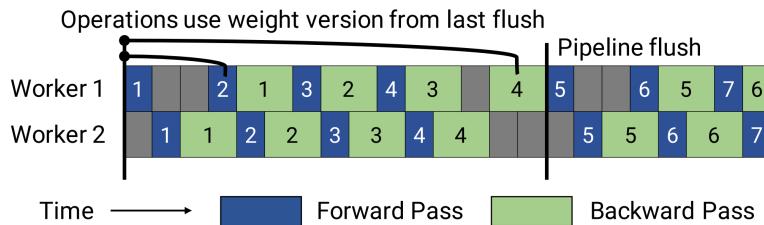
- **权重缓存 (Weight Stashing)**：每个工作节点保存多个版本的模型参数，并确保同一数据批次的前向和反向传播使用相同版本的权重。



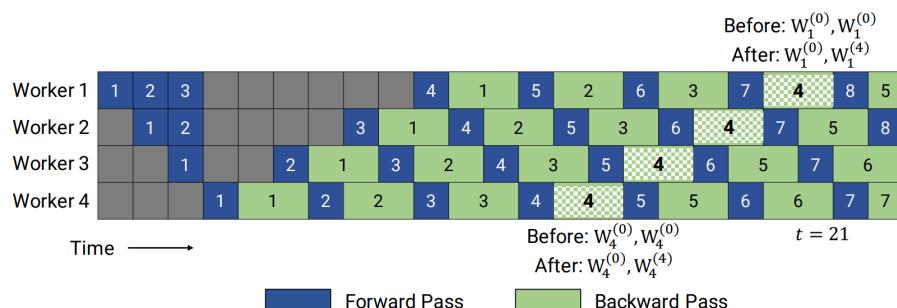
- **垂直同步 (Vertical Sync)**: 每次传播时，模型参数的版本信息也会随激活值和梯度一同传递到下一个阶段，从而确保各阶段对同一数据使用一致版本的权重。尽管保持了同步一致性，但整个训练仍为异步机制，不同于 GPipe 的全局同步模式。

PipeDream 在训练前还会对模型结构进行显存和计算负载分析，并使用动态规划求解出最优分区方案，以最大限度提升并行效率。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 该方法采用 GPipe 式的 **流水线清空机制 (flush)**，在训练过程中周期性同步权重，只需保存一个模型副本，显著降低显存占用，代价是略微牺牲吞吐率。



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 $k > d$ 。虽然仍需缓存旧权重以完成未返的梯度计算，但整个过程仅保留两个副本，内存占用大幅下降。



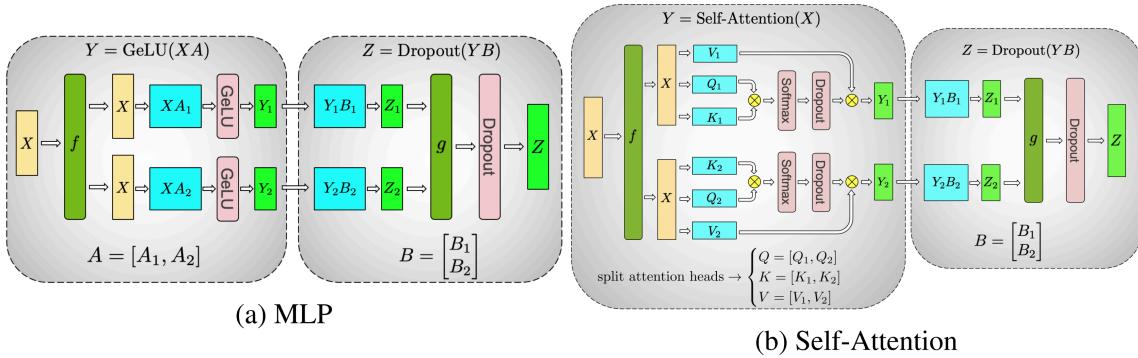
张量并行 (Tensor Parallelism)

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)**。

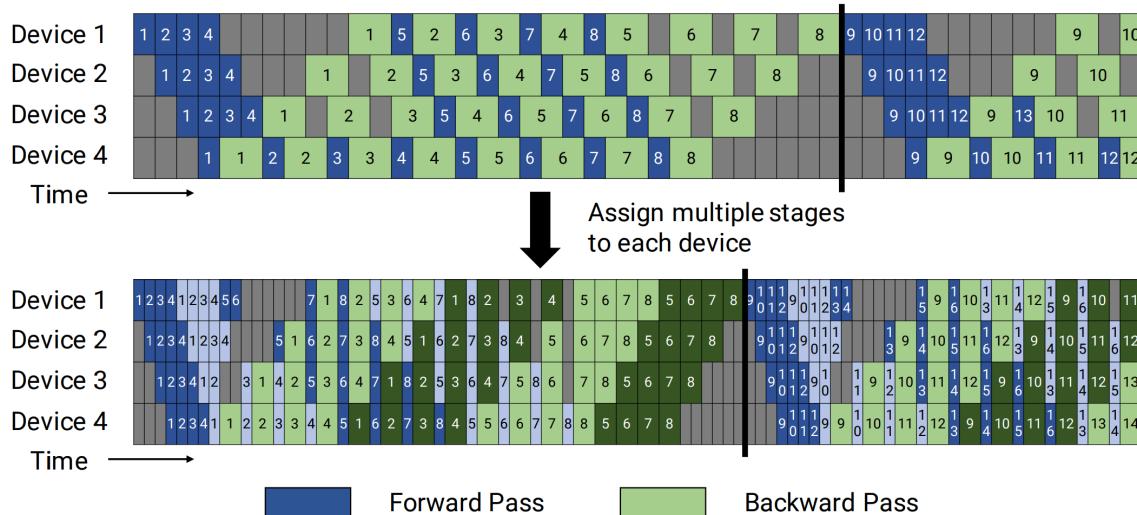
以 Transformer 为例 (因其在 NLP 中的广泛应用)。Transformer 模型主要由多层的 MLP 和自注意力 (Self-Attention) 模块组成。**Megatron-LM** (Shoeybi et al. 2020) 提出了一种简洁的方法来并行化 MLP 和自注意力内部的计算。在 Transformer 中，一个 MLP 层包括一个 GEMM (通用矩阵乘法) 操作，随后是非线性的 GeLU 激活函数。我们可以将权重矩阵按列进行切分：

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

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 包含 layers 1, 2, 9, 10；Device 2 包含 layers 3, 4, 11, 12；每个设备中拥有两个小模型块 $v = 2$ 。为了确保训练调度的正确性，要求每个 batch 中的 microbatches 数量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，随着模型参数规模的快速扩展，在 Google 等研究团队的推动下，

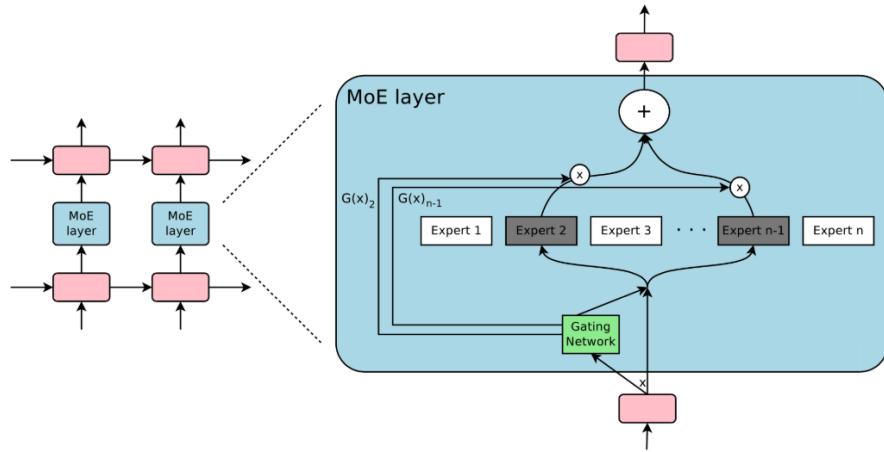
混合专家模型 (Mixture-of-Experts, MoE) 因其参数稀疏激活、计算高效等特性受到广泛关注。MoE 的核心思想来自于传统的**集成学习 (Ensemble Learning)**: 多个“弱”专家共同协作, 可以构建出更强大的学习器。

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）。

在神经网络中，一个典型的 MoE 层由以下两部分组成：

- 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。

$$G(x) = \text{softmax}(\text{topk}(H(x), k))$$

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

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

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：

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

由于每个专家仅处理部分样本，MoE 常出现“批次缩小问题 (shrinking batch problem)”。为保证训练稳定性，MoE 通常需要较大的 batch size，同时辅以数据并行 (DP) 和模型并行 (MP) 以维持吞吐。

GShard (Lepikhin et al., 2020) 通过分片机制将 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。

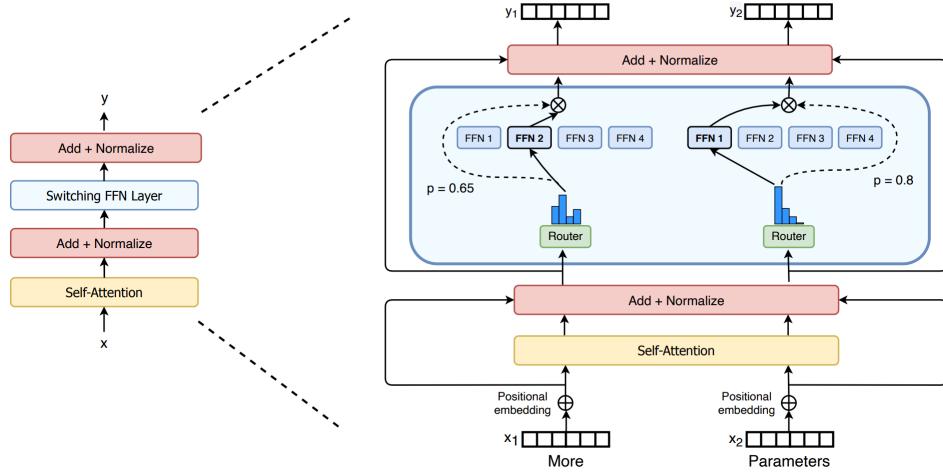
GShard 中对门控函数 G 进行了若干改进设计：

1. **专家容量 (Expert Capacity)**：控制每个专家最多接收的 token 数，防止过载；
2. **局部分组调度 (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)**：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:



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。

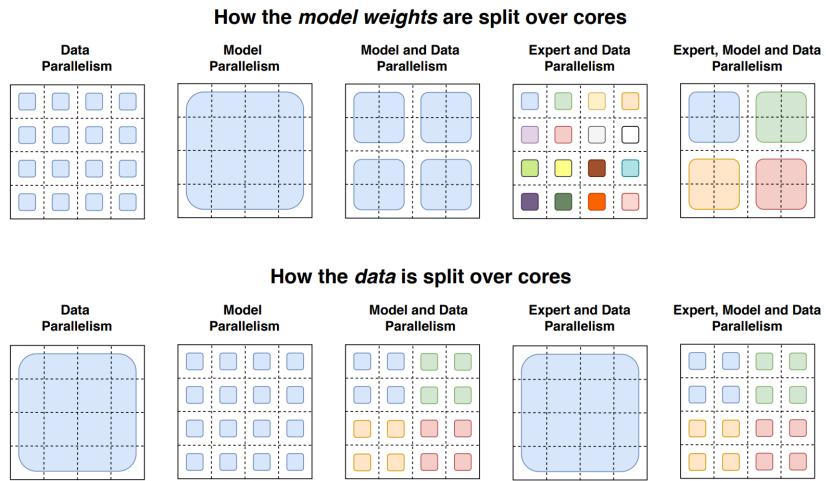
2. 小权重初始化 (Smaller Initialization)

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. 专家层高 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。

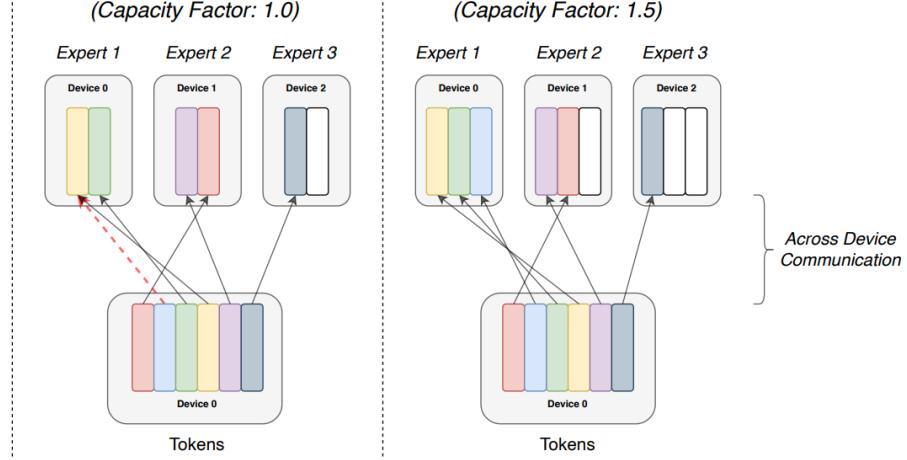
Switch Transformer 的并行策略直观总结如下:



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:

Terminology

- Experts:** Split across devices, each having their own unique parameters. Perform standard feed-forward computation.
- Expert Capacity:** Batch size of each expert. Calculated as $(\text{tokens_per_batch} / \text{num_experts}) * \text{capacity_factor}$
- Capacity Factor:** Used when calculating expert capacity. Expert capacity allows more buffer to help mitigate token overflow during routing.



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。

传统的 MoE 路由方式（如 GShard 的 top-2 或 Switch Transformer 的 top-1）是由路由器选择专家——每个 token 决定该被送往哪些专家。**Export Choice (EC)** (Zhou et al. 2022) 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。

形式化来讲，给定 e 个专家以及 n 个输入 token，记输入矩阵为 $X \in \mathbb{R}^{n \times d}$ 。token 与专家之间的匹配程度建模为专家选择分数 $S = \text{softmax}(XW_g) \in \mathbb{R}^{n \times e}$, 其中 $W_g \in \mathbb{R}^{d \times e}$ 。EC 的目标是构造一个 分配矩阵 $A \in \mathbb{R}^{e \times n}$ ，其中 $A[i, j]$ 表示第 i 个专家是否选择第 j 个 token，对应下面的优化问题：

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

直观上来讲， $\langle S^T, A \rangle$ 是“选择偏好”的得分总和：专家挑选它喜欢的 token； $H(A)$ 是熵项，鼓励 token 分配地更加多样、不集中于某些专家；约束条件防止专家过载或 token 被“滥用”。

论文使用了 **Dykstra 投影算法** 对这个优化问题进行迭代近似求解。参数 k 通过 $k = nc/e$ 确定， 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 $c = 1$ （即每个 token 只由一个专家处理），其效果也仍然超过 top-1 策略。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 需要看到完整的 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)

CPU 卸载 (CPU Offloading)：当 GPU 显存不足时，一种直接的办法是将暂不需要的数据卸载到主机内存 (**CPU RAM**)，待用时再加载回 GPU (Rhu et al., 2016)。这一策略实现简单，节省了宝贵的显存资源。

⚠ 但由于 PCIe 通信带宽较低，该方法会显著拖慢训练速度，因此在高效训练场景中逐渐被弃用。

激活重计算 (Activation Recomputation/ Checkpointing)

激活重计算 (Activation Recomputation) (Chen et al. 2016) 是一种以计算换内存的经典技巧。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。

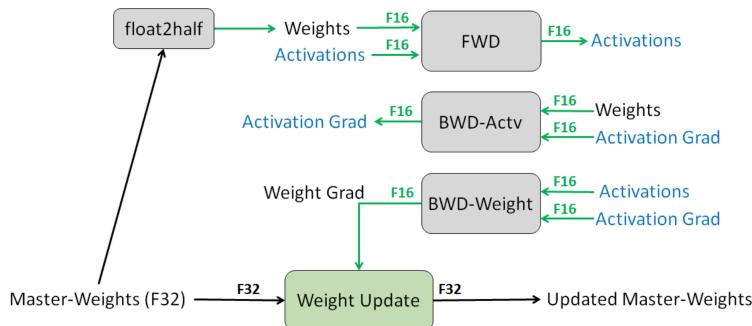
假设我们将一个 ℓ 层的网络平均分为 d 个分段。我们只保存每个分段边界处的激活，并在后向传播时重新计算中间层的激活（因为仍然需要它们来计算梯度）。在这种设置下，训练的内存成本 $M(\ell)$ 为：

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

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。该策略被广泛用于深层网络（如 BERT、GPT）训练，尤其适合与流水线并行结合使用。

混合精度训练 (Mixed Precision Training)

Narang & Micikevicius et al. (2018) 提出使用 **半精度浮点数 FP16** 替代传统的 **全精度 FP32**，以降低显存开销与加速计算。**混合精度训练 (Mixed Precision Training)** 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：

- 权重副本 (Master 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。
- 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。
- 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。

⌚ 实践中是否启用 loss scaling 与模型类型相关。例如 LSTM/SSD 模型受益明显，而图像分类模型变化不大。

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：

- 层特定的无损压缩 (Layer-specific Lossless Encoding)** 针对不同类型层使用不同编码策略：
 - ReLU + Pooling 层：** 使用二值化 (Binarization) 方法；
 - ReLU + Conv 层：** 采用“稀疏存储 + 稠密计算”(Sparse storage + Dense compute) 策略。

2. 激进的有损压缩 (Aggressive Lossy Encoding)

进一步压缩激活值，在可接受精度范围内牺牲一部分信息以节省内存。

3. 延迟精度降低 (Delayed Precision Reduction, DPR)

推迟激活精度的降低操作，直到再次使用前才进行精度还原，以提升训练稳定性。

 Gist 可将主流 CNN 模型内存占用减少约 1.8 倍，仅带来 4% 的额外训练时间。

内存高效的优化器 (Memory Efficient Optimizer)

以 Adam 为代表的优化器需要维护梯度一阶与二阶矩估计，显存开销是模型权重的 2~4 倍。为降低此成本，研究者提出如下替代方案：

- **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，也大幅降低了显存消耗。

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显存需求，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，来减少残余状态的显存占用。

 ZeRO 在 DeepSpeed 框架中已广泛应用于数百亿/万亿参数模型的训练。

References

- [1] Li et al. "PyTorch Distributed: Experiences on Accelerating Data Parallel Training" VLDB 2020.
- [2] Cui et al. "GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server" EuroSys 2016
- [3] Shoeybi et al. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." arXiv preprint arXiv:1909.08053 (2019).
- [4] Narayanan et al. "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM." arXiv preprint arXiv:2104.04473 (2021).
- [5] Huang et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism." arXiv preprint arXiv:1811.06965 (2018).
- [6] Narayanan et al. "PipeDream: Generalized Pipeline Parallelism for DNN Training." SOSP 2019.
- [7] Narayanan et al. "Memory-Efficient Pipeline-Parallel DNN Training." ICML 2021.
- [8] Shazeer et al. "The Sparsely-Gated Mixture-of-Experts Layer Noam." arXiv preprint arXiv:1701.06538 (2017).

- [9] Lepikhin et al. "GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding." arXiv preprint arXiv:2006.16668 (2020).
- [10] Fedus et al. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity." arXiv preprint arXiv:2101.03961 (2021).
- [11] Narang & Micikevicius, et al. "Mixed precision training." ICLR 2018.
- [12] Chen et al. 2016 "Training Deep Nets with Sublinear Memory Cost." arXiv preprint arXiv:1604.06174 (2016).
- [13] Jain et al. "Gist: Efficient data encoding for deep neural network training." ISCA 2018.
- [14] Shazeer & Stern. "Adafactor: Adaptive learning rates with sublinear memory cost." arXiv preprint arXiv:1804.04235 (2018).
- [15] Anil et al. "Memory-Efficient Adaptive Optimization." arXiv preprint arXiv:1901.11150 (2019).
- [16] Rajbhandari et al. "ZeRO: Memory Optimization Towards Training A Trillion Parameter Models Samyam." arXiv preprint arXiv:1910.02054 (2019).
- [17] Zhou et al. "Mixture-of-Experts with Expert Choice Routing" arXiv preprint arXiv:2202.09368 (2022).