

大模型训练方法

单 GPU 优化方法：

1. 常见的减少内存和加快训练速度的方法如下，图中括号表示不一定严格成立（即混合精度训练在通常情况下能节省显存，但并不是100%成立）

Method	Speed	Memory
Gradient accumulation	No	Yes
Gradient checkpointing	No	Yes
Mixed precision training	Yes	(No)
Batch size	Yes	Yes
Optimizer choice	Yes	Yes
DataLoader	Yes	No
DeepSpeed Zero	No	Yes

2. Transformer 模型中的操作可以分为3个 group:

- 张量缩并 (**Tensor Contractions**) : Linear 和 Multi-head Attention 都需要做批矩阵乘法 (Batched matrix-matrix multiplications)，以及卷积运算，这些操作是计算量最大的部分；使用包含 **TensorCore** 的GPU能极大加速运算速度 (关于 TensorCore 相关内容参照后面混合精度训练)；
 - 统计归一化 (**Statistical Normalizations**) (或称为 规约 reduction): 与张量收缩相比，**Softmax** 和 **LayerNorm** 计算强度较低，并且涉及一个或多个 reduction 操作，然后使用map 对结果映射；**这些操作应该以 FP32 的模型计算** (Large reductions (sums across elements of a vector) should be carried out in FP32)。在混合精度的实现中，这些规约操作 (例如 softmax、layernorm、batchnorm等) 从 memory 中读取 FP16 格式的 tensor，以 FP32 格式进行计算，然后将结果保存以 FP16 的形式保存 (Both of the layer types in our implementations still read and write FP16 tensors from memory, performing the arithmetic in FP32)；**使用 FP32 进行规约操作不影响训练速度**，因为这些操作受内存带宽限制 (memory-bandwidth limited)；
 - 逐元素操作 (**Element-wise Operators**) : 典型的操作包括 non-linearities (非线性激活) 和 element-wise matrix products (逐元素矩阵乘法)，还包括 dropout、residual connection。**这些是计算密度最低的操作**；这些逐元素操作受内存带宽限制 (memory-bandwidth limited)，算术精度 (FP16 或者 FP32) 不影响这些操作速度。
3. 模型训练过程中，以下组件会占用内存：1. 模型权重；2. 优化器状态；3. 梯度；4. 为梯度计算保存的前向激活 (forward activations saved for gradient computation)；5. 临时缓冲区 (temporary buffers)；6. 特定功能内存 (functionality-specific memory)；
- 使用 AdamW 优化器以及混合精度进行训练时，在训练阶段，每个参数需要 18 字节加上激活内存，在推理阶段，没有优化器状态和梯度，此时，每个模型参数需要 6 个字节加上激活内存。(A typical model trained in mixed precision with AdamW requires 18 bytes per model parameter plus activation memory. For inference there are no optimizer states and gradients, so we can subtract those. And thus we end up with 6 bytes per model parameter for mixed precision inference, plus activation memory.)
 - 模型权重 (**Model Weights:**) :
 - 对于 FP32 训练，每个参数占用 4 个字节 (4 bytes * number of parameters for fp32 training)
 - 对于混合进度训练，每个参数需要 6个字节，需要在显存中维护一个 FP32 的模型和一个 FP16的模型 (6 bytes * number of parameters for mixed precision training (maintains a model in fp32 and one in fp16 in memory))
 - 优化器状态 (**Optimizer States**) :
 - 对于一般的 AdamW 优化器，每个参数需要维持2个状态 (一阶矩和二阶矩)，需要 8 个字节 (8 bytes * number of parameters for normal AdamW (maintains 2 states)) .
 - 对于 8-bit 优化算法 (例如 bitsandbytes)，每个参数需要 2 个字节；
 - 对于 SGD with Momentum, 每个参数需要 4 个字节 (只保留动量这一个状态)

4. 梯度 (Gradients)

- 每个参数需要 4 个字节, 对于 FP32 和 mixed precision training 都一样 (梯度总是保存为 FP32 的形式, 在反向传播的时候使用FP16形式的梯度, 但是在进行 unscale 以及更新参数的时候, 需要FP32 的形式)

5. 前向激活 (Forward Activations)

- 前向激活所占用的内存空间由 序列长度 (sequence length)、隐藏状态维度 (hidden size) 以及 batch_size;
- 前向激活被保存用于计算梯度;

6. 临时内存 (Temporary Memory) :

- 在编写代码时应该考虑在适当的时候销毁一些临时变量;

7. 特定功能内存:

- 对于某些特定的任务, 可能需要特殊的内存, 例如使用 beam search 进行文本生成时, 需要保留多个输入和输出的副本;

4. 前向和后向执行速度对比:

- 对于卷积\线性层等结构, 后向过程 (backward) 中的计算量 (Flops, **Floating-point Operations**, 浮点运算次数) 可以认为是前向过程 (forward) 的2倍。因此耗时通常也是大约2倍 (有时候会更多);

5. Batch Size:

- batch size 和 输入/输出神经元个数建议为 2^N (通常为 8 的倍数) ;
- 对于 Transformer 模型, 确保词表的大小可以被 8 整除;
- Tensor Core 建议: 对于 FP16 数据类型, 乘数最好是 8 的倍数, 而对于 A100 GPU, 乘数建议为 64 的倍数;

6. 梯度累加 (Gradient Accumulation) :

- 梯度累加的做法是: 通过对小batch 数据进行前向和后向计算, 在此过程中累加梯度 (**累加循环**), 当足够的梯度累加之后, 进行模型的优化步骤; 通过这种方式, 可以实现较大的 batch size; (相比于在条件允许时直接使用较大的batch size, 此种方法速度会相对较慢, 因为包含多次 forward\backward 过程)

2. 为什么需要对每一个 micro-step 的 loss 除以 accumulation_steps 呢?

- 假设 $BatchSize = 100$ 时, 这100个样本总的 Loss 为 L_{all} , 则在更新梯度时使用的 $L_{ori} = \frac{L_{all}}{100}$
- 假设现在使用 $micro_BatchSize = 25$, 且 $Accumulation_Step = 4$ 的梯度累计, 则对于每一个 $micro_step$, 假设这 25 个样本得到的总的 Loss 为 L_{part} , 则每个 $micro_step$ 更新梯度时使用的 $L_{new_1} = \frac{L_{part_1}}{25}$
- 由于在梯度累加过程中, 模型参数没有更新, 因此, 100 个样本产生的总的 Loss 应该等于 4 个批次的 (每批次25个样本) 的 Loss 之和, 即有 $L_{part_1} + L_{part_2} + L_{part_3} + L_{part_4} = L_{all}$
- 若不对每个 $micro_step$ 的 loss 除以 $accumulation_steps$, 则相当于使用

$$L_{new_1} + L_{new_2} + L_{new_3} + L_{new_4} = \frac{L_{part_1} + L_{part_2} + L_{part_3} + L_{part_4}}{25} = \frac{L_{all}}{25} \quad (1)$$

更新梯度, 这比实际应该使用的梯度值扩大了 4 倍;

- 综上, 对每个 $micro_batch_size$ 的梯度除以 $accumulation_step$ (进行归一化) 是必须的。

3. 梯度累加的实现:

1. 低效实现:

```

1  optimizer = ...
2  NUM_ACCUMULATION_STEPS = ...
3  for epoch in range(...):
4      for idx, sample in enumerate(dataloader):
5          inputs, labels = sample
6          # Forward Pass
7          outputs = model(inputs)
8          # Compute Loss and Perform Back-propagation
9          loss = loss_fn(outputs, labels)
10         # Normalize the Gradients
11         loss = loss / NUM_ACCUMULATION_STEPS
12         loss.backward()
13         if ((idx + 1) % NUM_ACCUMULATION_STEPS == 0) or (idx + 1 == len(dataloader)):
14             optimizer.step()
15             optimizer.zero_grad()

```

- 由于 Pytorch 在 DDP训练时, 每个 $loss.backward()$ 会触发梯度同步 (**gradient synchronization, all_reduce 操作**), 但是在梯度累加的应用场景下, 我们只希望达到累加的 step 之后才进行梯度同步, 因此上面的实现是低效的, 因为包含了很多多余的通讯 (all_reduce 时间成本较高)。Pytorch 提供了 `no_sync` 上下文管理器解决这个问题。实现如下:

```

1  device = "cuda"
2  model.to(device)

```

```

3     gradient_accumulation_steps = 2
4     for index, batch in enumerate(training_dataloader):
5         inputs, targets = batch
6         inputs = inputs.to(device)
7         targets = targets.to(device)
8         if (index + 1) % gradient_accumulation_steps != 0 or index != len(training_dataloader) - 1:
9             with model.no_sync():
10                 outputs = model(inputs)
11                 loss = loss_function(outputs, targets)
12                 loss = loss / gradient_accumulation_steps
13                 loss.backward()
14         else:
15             outputs = model(inputs)
16             loss = loss_function(outputs, targets)
17             loss = loss / gradient_accumulation_steps
18             loss.backward()
19             optimizer.step()
20             scheduler.step()
21             optimizer.zero_grad()

```

3. 一个更优雅的实现方式如下(下面的实现中, `local_rank != -1` 表示是分布式训练):

```

1  from contextlib import nullcontext
2  # 如果python版本小于3.7, 则使用下面这个:
3  # from contextlib import suppress as nullcontext
4
5  if local_rank != -1:
6      model = DDP(model)
7
8  optimizer.zero_grad()
9  for epoch in range(epoches):
10      for i, data in enumerate(train_loader):
11          # 只在DDP模式下, 轮数不是accumulation_steps整数倍的时候使用no_sync
12          mcontext = model.no_sync if local_rank != -1 and accumulation_count % accumulation_steps != 0 else nullcontext
13          with mcontext():
14              loss = model(data)
15              loss = loss / accumulation_steps
16              loss.backward()
17          # 轮数为accumulation_steps整数倍的时候, 传播梯度, 并更新参数
18          if accumulation_count % accumulation_steps == 0:
19              optimizer.step()
20              if model_scheduler is not None:
21                  model_scheduler.step()
22              optimizer.zero_grad()
23          accumulation_count += 1

```

7. 激活重计算 Activation recomputation (或者称为 Gradient Checkpoint, activation checkpointing)

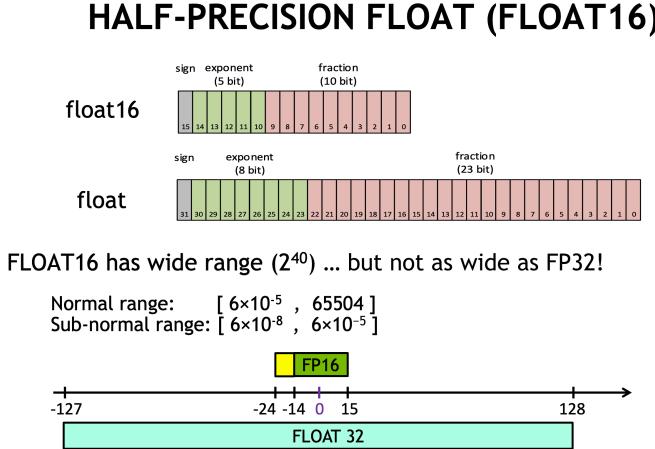
- 激活重计算旨在减少训练过程中激活 (activation) 带来的显存占用; 它的基本原理是以时间换空间。可以使得训练 ℓ 层深度网络的显存空间消耗 (memory cost) 优化为 $O(\sqrt{\ell})$, 因此是一种亚线性内存优化技术; 同时, 它最多带来了一次完整 forward 的计算成本;
- 其实现原理是: 将 L 层网络均匀划分为 d 个分片 (partition); 只有在分片边界处的激活被保存并且在 worker 之间进行通信。每个分片内部的中间激活在 backward 过程中重新计算, 以完成梯度计算 (Only activations at partition boundaries are saved and communicated between workers. Intermediate activations at intra-partition layers are still needed for computing gradients so they are recomputed during backward passes)。
- 通过使用激活重计算, 显存消耗 $M(L)$ 计算如下:

$$M(\ell) = \max_{i=1, \dots, k} \underbrace{\text{cost-of-one-partition}(i)}_{\text{cost of back-propagation on the } i\text{-th partition}} + \underbrace{O(d)}_{\text{store intermediate outputs}} = O\left(\frac{\ell}{d}\right) + O(d) \quad (2)$$

其中, d 表示分片的数量, 当 $d = \sqrt{\ell}$ 上面式子最小。

- 通常的经验表明: 使用 gradient checkpointing 通常会使得训练速度变慢 20% (A general rule of thumb is that gradient checkpointing slows down training by about 20%)
- 上面的方案被称为全量激活重计算 (full activation recomputation), nvidia 最新提出了选择激活重计算 (selective activation recomputation); 在计算量和显存使用之间权衡, 只对占用大量显存但是计算成本低的激活进行重计算 (详细参考 Megatron-LM3 论文);
- 浮点数类型

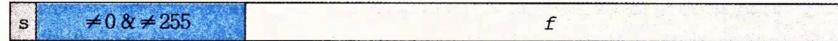
1. 知识点：浮点数在计算机内部的存储（以下内容来自《深入理解计算机系统（第三版）》）：



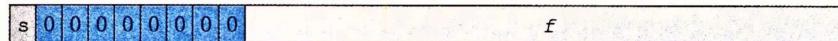
1. IEEE 浮点标准用 $V = (-1)^s \times 2^E \times M$ 表示一个浮点数，其中， s 为符号位 (sign)， E 表示阶码 (exponent)， M 表示尾数 (尾数通过小数字段 (fraction) 实现的)；对于 FP16 来说，符号位占 1 位 (bit)，阶码占 5 位，小数位占 10 位；

2. 根据阶码位 (exp) 的值，被编码的值可以分为如下图所示的 3 种情况：

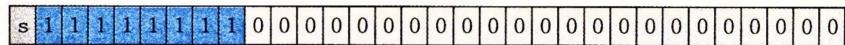
1. 规格化的



2. 非规格化的



3a. 无穷大



3b. NaN

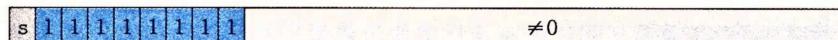


图 2-33 单精度浮点数值的分类(阶码的值决定了这个数是规格化的、非规格化的或特殊值)

1. 规格化的值 (Normalized values) :

1. 当 exp 的位不全为 0 且不全为 1 时，属于这种情况；

2. 在这种情况下，阶码位的值是 $E = e - Bias$ ，其中 e 为 exp 位表示的无符号数，而 $Bias$ 是一个等于 $2^{k-1} - 1$ 的值 (k 为阶码的位数)；

3. 小数字段 $frac$ 被解释为小数值 f ，其取值为 $0 \leq f < 1$ ，在规格化的情况下，尾数定义为 $M = 1 + f$ ，这种方式被称为隐含的以 1 开头 (implied leading 1) 的表示。使用这种表示方法是因为可以通过调整阶码 E ，使得尾数 M 在范围 $1 \leq M < 2$ 之中，这样可以获得一个额外精度位（第一位始终为 1，可以不显示表示）

4. 在 FP 16 的情况下，最大的规格化数的二进制表示为 $[0][11110][1111111111]$ ，根据上述计算方法：

$$1. Bias = 2^{5-1} - 1 = 15, e = 11110_2 = 30_{10}, 因此 E = e - Bias = 30 - 15 = 15$$

$$2. f = \frac{2^{10}-1}{2^{10}}, M = 1 + f = 2 - 2^{-10}$$

3. 因此 FP16 最大的规格化数为：

$$V = (-1)^s \times 2^E \times M = (-1)^0 \times 2^{15} \times (2 - 2^{-10}) = 2^{16} - 2^5 = 65504$$

5. 在 FP16 的情况下，最小的规格化数二进制表示为 $[0][00001][000000000]$ (这里的最小指的是最 小正数)，根据同样的计算方法：

$$1. Bias = 15, e = 00001_2 = 1_{10}, 因此 E = e - Bias = 1 - 15 = -14$$

$$2. f = 0, M = 1$$

3. 因此 FP16 的最小规格化数为 $V = (-1)^0 \times 2^{-14} \times 1 = 2^{-14} \approx 6.104 \times 10^{-5}$

6. 使用与 FP16 相同的计算方法，可以得出 FP32 的最大规格化数为： $2^{128} - 2^{104} \approx 3.40 \times 10^{38}$ ，FP32 最小的规格化数为 $2^{-126} \approx 1.18 \times 10^{-38}$

2. 非规格化的值 (Denormal values) :

1. 当阶码全为 0 时，表示的是 非规格化数。非规格化数的作用有 2 个：(1) 表示数值 0，因为规格化数要求 $M \geq 1$ ，因此不能表示 0；(2) 非规格化数的另一个功能是表示那些非常接近于 0.0 的数，它们的数值分布均匀地接近于 0.0；

2. 当非规格化的情况下，阶码值 $E = 1 - Bias$ ，而尾数的值 $M = f$ ，也就是非规格化的情况下，尾数的值等于小数字段的值，不包含隐含的开头的 1。

3. FP16 的最小非规格化数的二进制表示为 `[0][00000][0000000001]` (同样考虑的是正数)，根据非规格化数的计算：

$$1. Bias = 2^{5-1} - 1 = 15, E = 1 - Bias = -14$$

$$2. M = f = \frac{1}{2^{10}} = 2^{-10},$$

$$3. 因此 FP16 的最小非规格化数为: V = (-1)^0 \times 2^{-14} \times 2^{-10} = 2^{-24} \approx 5.96 \times 10^{-8}$$

4. 而 FP16 的最大非规格化数的二进制表示为 `[0][00000][1111111111]`，根据同样的计算方法：

$$1. Bias = 15, E = 1 - Bias = -14$$

$$2. M = f = \frac{2^{10}-1}{2^{10}} = 1 - 2^{-10}$$

$$3. 因此 FP16 的最大非规格化数为$$

$$V = (-1)^0 \times 2^{-14} \times (1 - 2^{-10}) = 2^{-14} - 2^{-24} \approx 6.098 \times 10^{-5}$$

5. 使用与 FP16 相同的计算方法，可以得出 FP32 的最小非规格化数为 $2^{-149} \approx 1.4 \times 10^{-45}$ ，FP32 最大的非规格化数为 $2^{-126} - 2^{-149} \approx 1.18 \times 10^{-38}$

3. 特殊值：

1. 当阶码全为 1 是表示特殊值。当小数位全为 0 时，表示无穷，当符号为 $s = 0$ 时，表示 $+\infty$ ，当 $s = 1$ 时，表示 $-\infty$ ；当小数位非 0 时，表示 `NaN`；

3. 如下图所示，表示在包含 3 个阶码位和 2 个小数位共 6 位时，所表示的值，数值范围为 $[-14, 14]$ ，可以看出，可表示的数并不是均匀分布的，越靠近原点越稠密：

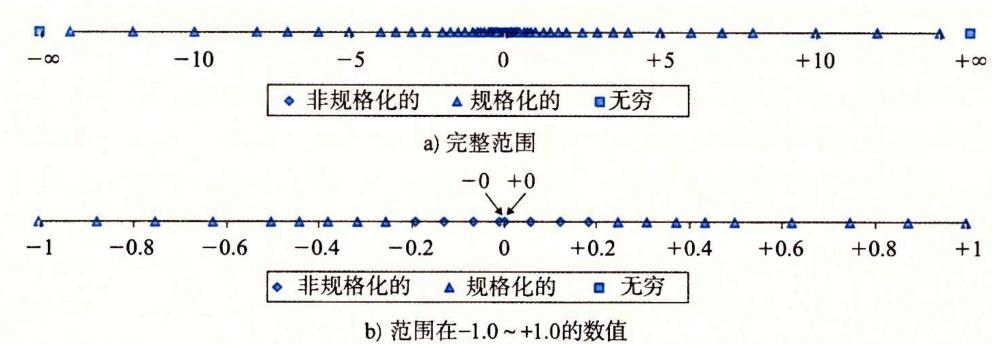


图 2-34 6 位浮点格式可表示的值($k=3$ 的阶码位和 $n=2$ 的尾数位。偏置量是 3)

4. 通过上述计算步骤，就可以理解如下范围了：

1. FP16 的取值范围为 $[-65504, 65504]$

2. 当只考虑大于 0 的情况时，如下图所示，FP16 规格化的值的范围为 $[2^{-14}, 2^{16} - 2^5] \approx [6 \times 10^{-5}, 65504]$ ，而非规格化数的范围为 $[2^{-24}, 2^{-14} - 2^{-24}] \approx [6 \times 10^{-8}, 6 \times 10^{-5}]$

3. FP16 能表示的最小精度为 2^{-24} ，小于此范围会被置 0；

4. 当只考虑大于 0 时，FP32 规格化数的值的范围为： $[2^{-126}, 2^{128} - 2^{104}] \approx [1.18 \times 10^{-38}, 3.40 \times 10^{38}]$ ，而非规格化数的值的范围为 $[2^{-149}, 2^{-126} - 2^{-149}] \approx [1.4 \times 10^{-45}, 1.18 \times 10^{-38}]$

5. FP16 的动态范围 (dynamic range) (包含非规格化数) 为： $[2^{-24}, 2^{16} - 2^5]$ ，大约是 2^{40} 这个量级；

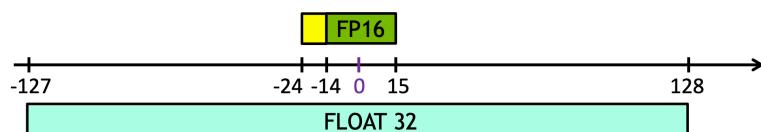
6. 由于 FP16 中小数字段占用 10 位，因此，最大舍入误差 (max rounding error) 是 2^{-10} 量级 ([机器精度 Machine epsilon](#))；而 FP32 中小数字段占用 23 位，因此最大舍入误差是 2^{-23} ；

7. 下面的这个表需要背诵：

	最小非规格数 (Minimum denormal)	最大规格化数 (Maximum normalized)	动态范围 (dynamic range)	最小精度 (precision)
FP16	$2^{-24} \approx 6 \times 10^{-8}$	$2^{16} - 2^5 = 65504$	2^{40}	2^{-24}
FP32	$2^{-149} \approx 1.4 \times 10^{-45}$	$2^{128} - 2^{104} \approx 3.4 \times 10^{38}$	2^{277}	2^{-149}

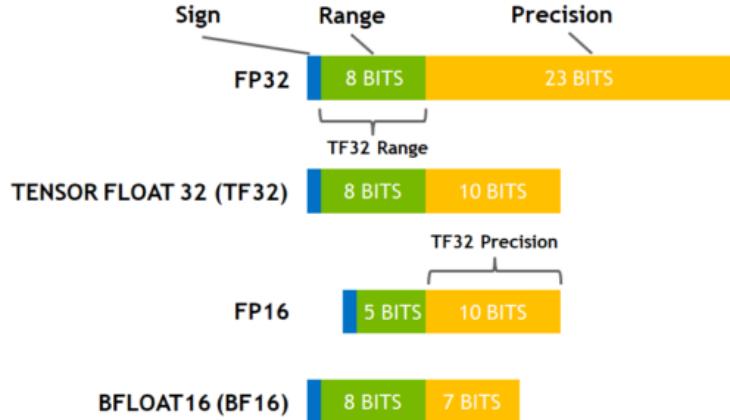
FLOAT16 has wide range (2^{40}) ... but not as wide as FP32!

Normal range: $[6 \times 10^{-5}, 65504]$
Sub-normal range: $[6 \times 10^{-8}, 6 \times 10^{-5}]$



2. 混合精度训练：

1. 混合精度训练的想法是：并非所有的变量都需要以完整的 32 位浮点数存储。如果可以降低精度，就可以加快计算。
2. 以下是影响内存使用和吞吐量的常用浮点数据类型 (both memory usage and throughput)：
 1. fp32 (float 32)
 2. fp16 (float16)
 3. bf16 (bfloating16)
 4. tf32(CUDA internal data type)



其中，bf16 和 tf32 仅支持 NVIDIA Ampere 架构的 GPU，例如 A100、A10 等；（不包括 V100, V100 为 Volta 架构）

3. FP16 计算优点：

1. 使用 FP16 可以使用更少的位存储权重、激活值、梯度 (Weights, activations, and gradients) 等，因此：1. 降低了显存占用，可以使用更大的模型或者更大的 batch size; 2. 减少了分布式训练时的通信带宽的压力，提升了吞吐；
2. 较新架构的 GPU (包含 Volta、Turing、Ampere 架构) 都支持专为 FP16 矩阵运算而设计的专用张量内核 (Recent generations of NVIDIA GPUs come loaded with special-purpose **tensor cores** specially designed for fast fp16 matrix operations.), 可以加速算术运算；(Pascal 架构不支持 Tensor Cores，因此混合精度提升不明显)

4. 使用 FP16 计算的缺点：

1. 由于 FP16 数值范围的限制，使用 FP16 训练会存在数据溢出和舍入误差的问题：
 1. 数据溢出 (主要是下溢出, Underflow) : FP16 的最小非规格化数是 $2^{-24} \approx 6 \times 10^{-8}$ ，小于这个数的值会被置为 0，在 Mixed Precision Training 的论文中，他们实验的模型中，有 5% 的权重梯度无法用 FP16 表示；因此影响模型参数更新；
 2. 舍入误差 (Rounding Error) : 由于 FP16 的表达中，有 10 位表示小数字段，因此最大舍入误差为 2^{-10} ，因此，当权重值和权重更新 (the ratio of the weight value to the weight update) 的比值大于 $2^{11} = 2048$ 时，在 FP16 的表示下，权重更新也会为 0；
2. 根据论文中实验的模型，完全使用 FP16 训练会导致 80% 的相对精度损失 (results in 80% relative accuracy loss)

5. 为了解决纯 FP16 计算的问题，混合精度训练使用如下 3 种方法解决：

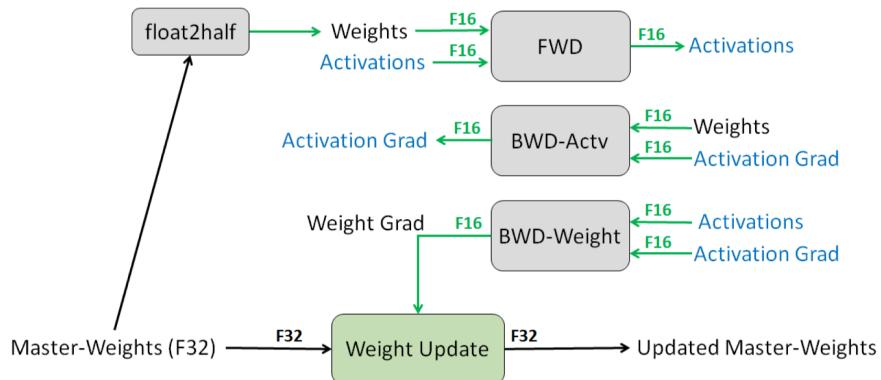


Figure 1: Mixed precision training iteration for a layer.

1. FP32 权重备份 (FP32 MASTER COPY OF WEIGHTS)

1. 权重备份主要用于解决舍入误差的问题。其主要思路是在训练过程中，使用 FP16 的格式存储权重、激活值、梯度（weights, activations and gradients），同时保存一份 FP32 的权重（weight）备份。FP32 多备份一次 weight 会增加显存占用，但是由于最占显存的是 activations 的值（特别是在 batch size 很大时），使用 FP16 存储 activations 会极大降低显存占用。因此混合精度训练整体也会降低显存占用；
2. 动态损失缩放（Loss Scaling）

1. 损失缩放主要解决 FP16 的下溢出（underflow）问题，论文中数据显示，在训练 SSD 过程中，67% 的激活函数梯度都小于 2^{-24} ，因此在 FP16 的格式下都为 0。论文实验表明，不进行缩放时，训练无法收敛；只需要对梯度放大 8 倍 ($\times 2^3$)，就可以得到与 FP32 匹配的精度。表明 $[2^{-27}, 2^{-24}]$ 范围内的梯度激活值很重要，而小于 2^{-27} 的梯度激活值与训练无关（此结论针对 SSD 网络）。

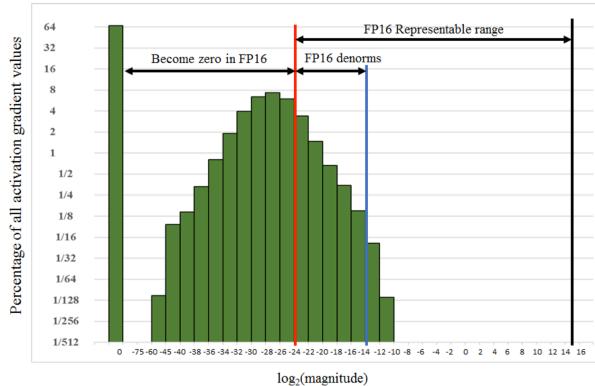


Figure 3: Histogram of activation gradient values during the training of Multibox SSD network. Note that the bins on the x-axis cover varying ranges and there's a separate bin for zeros. For example, 2% of the values are in the $[2^{-34}, 2^{-32}]$ range, 2% of values are in the $[2^{-24}, 2^{-23}]$ range, and 67% of values are zero.

2. 得到 FP32 的 loss 后，乘以 loss scale 放大并保存为 FP16 格式，进行反向传播，更新时转为 FP32 并且缩小回来。很多梯度值比较小，无法用 FP16 表示。因此在前向传播后对 loss 进行扩大（通常使用动态值），这样在反向传播时（由于链式法则）所有的值也都扩大了相同的倍数。在更新 FP32 的权重之前需要缩小（unscale）回去。
3. 动态损失缩放（Dynamic Loss Scaling）：

1. 为了充分利用 FP16 的动态范围，更好地缓解舍入误差，应该在没有溢出的情况下，尽量使用比较大的 scale 值。在 Pytorch 的实现中，`init_scale = 216 = 65536`，选择较大的初始 scale 是有好处的，因为 GradScaler 很容易检测到数据上溢出（overflow）问题（会出现 `inf`），也容易处理（跳过此 batch）。但是无法检测和组织下溢出问题（underflow），因为 0 总是合法的。因此如果设置 `init_scale` 太低，而 `growth_interval` 太高，会导致在 GradScaler 介入模型已经下溢并且发散；
2. 动态损失缩放算法，就是每当梯度溢出时候减少 scale 值，并且间歇性地尝试增加 scale 值，从而实现在不引起溢出的情况下使用最高的放大因子，更好地恢复精度。
3. 在 Pytorch 的实现中，`torch.cuda.amp.GradScaler(init_scale=65536.0, growth_factor=2.0, backoff_factor=0.5, growth_interval=2000, enabled=True)`；初始的 scale 值设置为 2^{16} ，随着训练的进行，梯度值更新幅度会逐渐减少，因此每隔 `growth_interval` 个 step，就将 scale 值扩大 `growth_factor` 倍，如果发现上溢出，就将缩小 `backoff_factor` 倍；

3. 精度累加（Precision Accumulated）

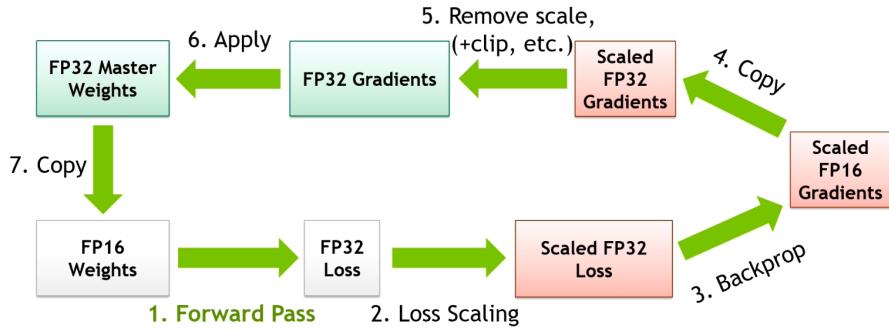
1. 在一些模型中，使用 FP16 进行矩阵乘法时，需要利用 FP32 进行矩阵乘法中间的累加，然后再将结果转化为 FP16 进行存储（some networks require that FP16 vector dot-product **accumulates the partial products into an FP32 value**, which is converted to FP16 before writing to memory），如果没有中间 FP32 的累加步骤，模型最终的结果会下降；
2. 在 Volta 及之后的架构（包括 Turing、Ampere）的 GPU 中，支持 TensorCore，专门针对 FP16 的矩阵运算进行加速；（**在没有 Tensor Core 的 GPU 上进行混合精度训练基本没有速度的提升**）
3. 知识点：什么是 TensorCore：

1. 尽管 mixed precision training 会节省显存以及 IO 等。但是如果没有任何特殊 GPU 硬件的支持，它也不会提供模型加速。
2. TensorCore 是一种新型的处理单元（processing unit），它针对一个非常具体的操作进行优化：将 2 个 4×4 的 FP16 矩阵相乘，并将结果加到第 3 个 FP16 或者 FP32 矩阵上。
3. 使用此运算作为基本构建块实现更大的 FP16 的矩阵乘法（Larger `fp16` matrix multiplication operations can be implemented using this operation as their basic building block.），神经网络中计算密集的所有矩阵操作均可以使用这种方式加速。

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} + \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

6. 混合精度算法流程总结：

MIXED PRECISION TRAINING



1. 保存一份 FP32 的模型权重 (Maintain a primary copy of weights in FP32)
2. 初始化一个很大的缩放因子 (Initialize scale factor S to a large value)
3. 在每个迭代步骤中 (For each iteration) :
 1. 复制一份 FP16 的模型权重 (Make an FP16 copy of the weights)
 2. 使用 FP16 进行前向过程 (FP16 weights and activations)
 3. 得到 FP32 的 Loss 之后乘以 scale factor S (损失函数的计算一定是 FP32 的格式)
 4. 使用 FP16 进行反向传播 (FP16 weights, activations, and their gradients)
 5. 如果梯度中出现了 `Inf` 或者 `Nan` :
 1. 减小 S 的值 (在 Pytorch 的实现中, 乘以 `backoff_factor`, 按默认值就是将 S 乘以 $1/2$)
 2. 跳过权重更新的步骤
 6. 将 FP16 的梯度转化为 FP32 格式, 乘以 $1/S$ 进行 unscale
 7. (进行与梯度相关的操作, 例如 gradient clipping 等) 完成权重更新;
 8. 如果连续 `growth_interval` 个 step 没有出现 `Inf` 或者 `Nan`, 则增大 S 的值(在 pytorch 实现中, 乘以 `growth_factor`, 默认值是乘以 2)
7. 在 Pytorch 的 `torch.cuda.amp.autocast()` 的上下文管理器中, 会自动进行数据类型的转化 ([参考文档](#)) :
 1. 被转化为 `float16` 的操作主要包括矩阵乘法和卷积:


```
matmul, addbmm, addmm, addmv, addr, baddbmm, bmm, chain_matmul, multi_dot, conv1d, conv2d, conv3d, conv_transpose1d, conv_transpose2d, conv_transpose3d, GRUCell, linear, LSTMCell, matmul, mm, mv, prelu, RNNCell
```
 2. 以 `float32` 运行的操作主要包括: 对数、指数、三角函数、常见的损失函数、batchnorm 等


```
pow, rdiv, rpow, rtruediv, acos, asin, binary_cross_entropy_with_logits, cosh, cosine_embedding_loss, cdist, cosine_similarity, cross_entropy, cumprod, cumsum, dist, erfinv, exp, expm1, group_norm, hinge_embedding_loss, kl_div, l1_loss, layer_norm, log, log_softmax, log10, log1p, log2, margin_ranking_loss, mse_loss, multilabel_margin_loss, multi_margin_loss, nll_loss, norm, normalize, pdist, poisson_nll_loss, pow, prod, reciprocal, rsqrt, sinh, smooth_l1_loss, soft_margin_loss, softmax, softmin, softplus, sum, renorm, tan, triplet_margin_loss
```
 3. 还有一些操作, 只需要保证输入的类型一致即可, 2 中精度都可以, 如果一个输入为 FP16, 另一个为 FP32, 则所有输入都使用 FP32 格式进行运算。

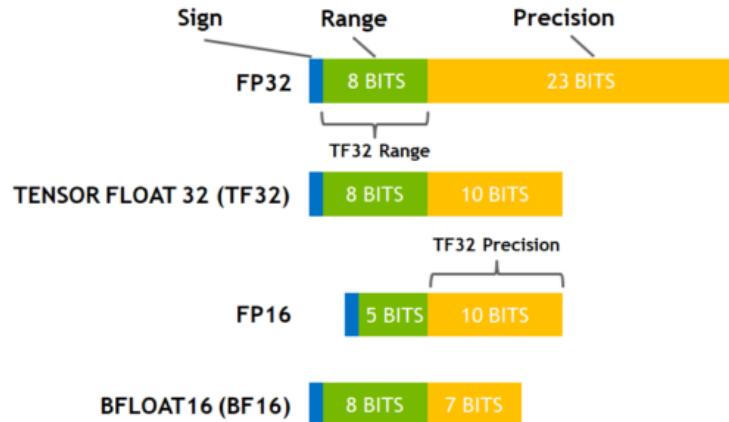

```
addcdiv, addcmul, atan2, bilinear, cross, dot, grid_sample, index_put, scatter_add, tensordot
```

8. 在 Pytorch 2.0 等较新版本中，可以通过设置 `torch.set_float32_matmul_precision(precision)` 来隐式的进行混合精度训练，通过设置 `precision` 参数，可以对 float32 的矩阵乘法采用不同的内部精度（Sets the internal precision of float32 matrix multiplications.），`precision` 参数支持如下3种设置：
1. `precision='highest'` 表示使用 float32 的数据类型完成 float32 的矩阵乘法（默认值）
 2. `precision='high'` 表示在内部使用 TensorFloat32 或者 bfloat16_3x (Performs float32 computations in 3 bfloat16 passes) 的数据类型进行运算（如果针对这些数值类型的告诉矩阵算法是可用的（也就是环境允许），否则退化为 `highest`）；
 3. `precision='medium'` 表示 float32 矩阵乘法在内部使用 bfloat16 数据类型进行（如果环境允许，否则退化为 `high`）。
 4. 以上三种设置都不影响输出矩阵的数据类型，只控制矩阵乘法内部计算；
 5. 当设置 `precision='high'` 时，相当于设置 `torch.backends.cuda.matmul.allow_tf32 = True`
 6. 下图中的对比来自于[pytorch lightning AMP blog](#)

	Training speed (3 epochs)	Predictive acc. (test set)	Memory allocated
Float32	21.75 min matmul precision: "highest"	89.92%	10.42 GB
	8.11 min matmul precision: "high"	92.50%	5.37 GB
	8.14 min matmul precision: "medium"	92.50%	5.37 GB
Float16-mixed	7.25 min matmul precision: "highest"	92.15%	4.31 GB
	7.10 min matmul precision: "high"	92.15%	4.31 GB
	7.07 min matmul precision: "medium"	92.15%	4.31 GB

9. BF16 (Brain Floating Point) 数据格式：

1. 如下图所示，bf16 与 float16 相比，增加了阶码位（因而增加了动态范围），降低了尾数位（因而降低了精度）
2. 由于 BF16 与 FP32 有相同的动态范围，因此，不需要使用 `gradscale` 进行缩放。



10. TF32 (Tensor Float32) 数据格式：

1. TensorFloat32 模式只存在于 CUDA 内部，Pytorch 中不存在这种数据类型；
2. TF32 共包含19位，与 FP32 有相同的动态范围，但是精度与 FP16 相同（都是10位尾数位），通过在代码中设置 `torch.backends.cuda.matmul.allow_tf32 = True`，最多可以获得 3 倍的吞吐提升（get up to 3x throughput improvement.）（参考上面的图片，`precision=high` 时，训练时间相比 fp32 降低了 62%）
3. 即使是使用 fp16 或者 bf16 进行混合精度计算，设置此参数也可以提升吞吐。

11. 如下代码块实现了，在混合精度 + 梯度裁剪 + 梯度累加（代码来源：[nanoGPT](#)）

```

1 torch.backends.cuda.matmul.allow_tf32 = True # allow tf32 on matmul
2 torch.backends.cudnn.allow_tf32 = True # allow tf32 on cudnn
3 device_type = 'cuda' if 'cuda' in device else 'cpu' # for later use in torch.autocast
4 # note: float16 data type will automatically use a GradScaler
5 ptdtype = {'float32': torch.float32, 'bfloating16': torch.bfloat16, 'float16':
6 torch.float16}[dtype]
7 ctx = nullcontext() if device_type == 'cpu' else
8 torch.amp.autocast(device_type=device_type, dtype=ptdtype)
9 # initialize a GradScaler. If enabled=False scaler is a no-op
10 scaler = torch.cuda.amp.GradScaler(enabled=(dtype == 'float16'))
11
12 while True:
13     # forward backward update, with optional gradient accumulation to simulate larger
14     batch size
15     # and using the GradScaler if data type is float16

```

```

13     for micro_step in range(gradient_accumulation_steps):
14         if ddp:
15             # in DDP training we only need to sync gradients at the last micro step.
16             # the official way to do this is with model.no_sync() context manager, but
17             # I really dislike that this bloats the code and forces us to repeat code
18             # looking at the source of that context manager, it just toggles this
19             variable
20             model.require_backward_grad_sync = (micro_step ==
21             gradient_accumulation_steps - 1)
22             with ctx:
23                 logits, loss = model(X, Y)
24                 loss = loss / gradient_accumulation_steps # scale the loss to account for
25                 gradient accumulation
26                 # immediately async prefetch next batch while model is doing the forward pass on
27                 the GPU
28                 X, Y = get_batch('train')
29                 # backward pass, with gradient scaling if training in fp16
30                 scaler.scale(loss).backward()
31                 # clip the gradient
32                 if grad_clip != 0.0:
33                     scaler.unscale_(optimizer)
34                     torch.nn.utils.clip_grad_norm_(model.parameters(), grad_clip)
35                 # step the optimizer and scaler if training in fp16
36                 scaler.step(optimizer)
37                 scaler.update()
38                 # flush the gradients as soon as we can, no need for this memory anymore
39                 optimizer.zero_grad(set_to_none=True)

```

1. 上述代码注意点：

1. 第 1、2 行使用 TF32 加快矩阵运算；
 2. 第 6 行设置精度转化时的目标精度类型
 3. 第 8 行设置缩放参数，注意，当 `dtype=bfloat16` 时，是不需要进行梯度放大的（grad scale）；
 4. 第 19 行设置在分布式训练时，只有最后一个 `micro_step` 才进行梯度同步，去掉不必要的通讯；
 5. 第 22 行中，需要在每一个 `micro_step` 中对 `loss` 除以 `accumulation_step`；
 6. 第 26 行中，`scaler.scale(loss).backward()` 是在放大的 `loss` 上计算梯度（Calls backward() on scaled loss to create scaled gradients.），注意，在使用梯度累加（Gradient accumulation）时，也是在放大之后的梯度上进行累加（Accumulates scaled gradients.）
 7. 第 29 行和 30 行是进行梯度裁剪，梯度裁剪必须在原始的梯度上进行（在放大的梯度上裁剪没有意义），因此首先将 `gradient unscale`（取消放大），然后进行裁剪；
 8. 第 32 行 `scaler.step(optimizer)` 首先将梯度 `unscale`（本例子中由于 29 行已经进行了 `unscale`，因此不需要再次 `unscale`）；然后判断梯度值中是否包含 `Nan` 或者 `Inf`，如果没有包含，则调用 `optimizer.step()` 更新模型参数，否则，`optimizer.step()` 这一步被省略（也就是不更新参数）
 9. 第 33 行更新 scale factor 的值。
12. 总结：在 V100 机器上实验，使用 BERT-base 模型，混合精度训练速度提升 50% ~ 60%，显存降低 ~ 10% ([参考链接](#))；

9. 内存高效的优化器（Memory Efficient Optimizer）

1. 常识知识点：

1. 1GB 对应着 10亿字节，一个参数为30亿的模型，每个参数占用 4 个字节（float32类型），占用的显存空间是 $\frac{30 \times 4}{10} = 12GB$ ；10亿（1 Billion）参数的模型需要4G 显存；1亿参数需要 372.5M 的 RAM 空间，BLOOM-176B 需要 640GB 空间（8块80G的A100存储）；
2. 对于标准的 Adam 或 AdamW（Adam with weight decay）优化器，优化器需要对每个参数保留 8 个字节的存储（需要存储一阶矩估计和二阶矩估计），因此仅优化器需要 24GB 的显存（1B 参数需要 8G 显存放置优化器参数）；
3. Adafactor 优化器只需要保存 $O(M + N)$ 的二阶矩状态量，并且没有动量项，因此优化器需要对每个参数使用少于 4 个字节的存储，因此仅优化器需要少于 12GB 显存（小于 Adam 优化器的 1/2）。
4. 8Bit Adam 优化器如果对所有的状态都进行量化，每个参数只使用 2 个字节的存储，则将仅使用 6GB 显存（是 Adam 优化器的 1/4）。

2. Adafactor ([Adaptive Learning Rates with Sublinear Memory Cost](#))：

1. Adafactor 优化器是在 Adam 基础上发展而来，主要优点在于：

1. (No momentum) 去掉了动量部分；

2. (Factored second moment estimation) 对 Adam 算法中的二阶矩进行矩阵低秩分解，将之间需要 $O(m \times n)$ 空间存储的梯度平方矩阵的指数移动平均修改为只需要 $O(m + n)$ 的空间存储，只跟踪梯度平方矩阵的行和列之和的指数移动平均 (only need to keep track of moving averages of row and column sums of the squared gradient)；
 3. (β_2 varies with time) 将二阶矩的衰减系数定义为训练步数的函数 ($\hat{\beta}_{2,t} = 1 - t^{-0.8}$)，使得二阶衰减率 β_2 从 0 逐渐增大到 1，因此不需要偏置修正；
 4. (Update Clipping) 使用更新裁剪防止模型发散；
 5. (Relative Step Size) 将学习率乘以参数矩阵的 RMS(root-mean-square)；
2. 从 Adafactor 优化点可以看出，它去掉了动量部分 (去掉动量之后，所占用空间与 SGD-Momentum 相同)；以及将二阶矩进行分解，因此，Adafactor 用的显存比 SGD-Momentum 少，但是比不包含动量的 SGD 多；

3. 8 bit-Adam

1. 背景：SGD-Momentum 以及 Adam 等包含梯度统计信息的优化算法需要占用额外的显存空间 (例如 Adam 对每个参数需要额外的8个字节 (2个fp32))，本文提出了使用 8bit 替换 32bit 的方案，同时保持了性能持平；
2. 如下图所示：对于 32bit 存储的状态，SGD-Momentum 对每个参数需要消耗 4 个字节，对于 10亿模型的参数，占用 4GB 显存；而 Adam 对每个参数需要消耗 8 个字节，对于 10亿的模型，占用 8GB 显存；

$$\text{Momentum}(\mathbf{g}_t, \mathbf{w}_{t-1}, \mathbf{m}_{t-1}) = \begin{cases} \mathbf{m}_0 = \mathbf{g}_0 & \text{Initialization} \\ \mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + \mathbf{g}_t & \text{State 1 update} \\ \mathbf{w}_t = \mathbf{w}_{t-1} - \alpha \cdot \mathbf{m}_t & \text{Weight update} \end{cases} \quad (1)$$

$$\text{Adam}(\mathbf{g}_t, \mathbf{w}_{t-1}, \mathbf{m}_{t-1}, \mathbf{r}_{t-1}) = \begin{cases} \mathbf{r}_0 = \mathbf{m}_0 = \mathbf{0} & \text{Initialization} \\ \mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t & \text{State 1 update} \\ \mathbf{r}_t = \beta_2 \mathbf{r}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2 & \text{State 2 update} \\ \mathbf{w}_t = \mathbf{w}_{t-1} - \alpha \cdot \frac{\mathbf{m}_t}{\sqrt{\mathbf{r}_t + \epsilon}} & \text{Weight update,} \end{cases} \quad (2)$$

where β_1 and β_2 are smoothing constants, ϵ is a small constant, and α is the learning rate.

3. 使用 8bit 优化器面临的主要挑战包含3部分：

1. 量化精度 (quantization accuracy):
 1. 为了保持精度 (maintain accuracy)，引入某种形式的非线性量化来减少常见小幅度值和罕见大值的误差至关重要 (non-linear quantization to reduce errors for both common small **magnitude values and rare large ones**)
 2. 计算效率 (computational efficiency):
 1. 为了保证方法的实用性，8bit 优化器需要足够快，以免影响训练速度 (特别对于非线性量化)；
 3. 大规模稳定性 (large-scale stability):
 1. 为了保持超过10亿参数的大模型训练稳定性，量化方法不仅需要良好的平均误差，而且需要出色的最坏情况性能 (not only have a good mean error but excellent worse case performance)，因为单个大的量化误差可能导致整个训练发散；
4. 为了解决以上问题，本文提出的 8bit 优化算法包含3个部分：
1. 逐块量化 (block-wise quantization) :
 1. 逐块量化的目的是：隔离异常值，并且将误差更均匀地分布在所有的位 (bit) 上 (**isolates outliers and distributes the error more equally over all bits**)
 2. block-wise 量化降低了计算归一化的成本：为了对某个 tensor 进行动态量化 (dynamically quantize)，需要将 tensor 中的值归一化到 $[-1, 1]$ 之间。这种归一化需要对整个 tensor 进行归约 (reduction)，这需要在 GPU cores 之间进行多次同步；block-wise 量化将输入的 tensor 分块为大小为 $B = 2048$ 的小 block，并在每个 GPU core 上独立对不同的 block 进行归一化；
 3. Block-wise 量化的好处：
 1. 每个 block 独立计算，GPU cores 之间没有同步，因此吞吐量更大；
 2. 对于输入 tensor 中的 outliers 更加鲁棒；例如：假设一个 tensor 中的值服从标准正态分布，则 tensor 中只有 1% 的值其绝对值在 $[3, +\infty)$ (正态分布 99.7% 的面积位于平均数左右3个标准差范围内)，假如 tensor 中包含 (幅度 magnitude) 值为 5 的 outlier，若使用普通的量化方式，则为了范围 $[3, 5]$ 而保留的量化桶 (quantization buckets) 将大部分没有使用 (因为只有 1% 的数值在此范围内)；而通过 block-wise 量化，异常值的影响将仅限于单个 block；因此，大部分的 bit 位能得到更有效的应用 (most bits are used effectively in other blocks)
 3. 由于 outlier 表示输入 tensor 中的绝对值最大值，因此 block-wise 量化对于 outlier 的量化没有任何误差，这保证了最大的优化器状态值将始终以全精度进行量化 (blockwise quantization approximates outlier values **without any error**)。This guarantees that the largest optimizer states, arguably the most important, will always be quantized with

full precision.)。这一特性使得逐块动态量化既稳健又精确 (both robust and precise) ;

2. 动态量化 (dynamic quantization) :

1. 动态量化的目的是：对小的数值和大的数值都进行高精度量化 (quantizes both small and large values with high precision)
2. 本文中使用的动态量化是使用作者之前提出的 Dynamic Tree Quantization 以及其扩展；
3. 动态树量化 (Dynamic Tree Quantization) :

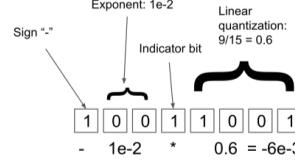
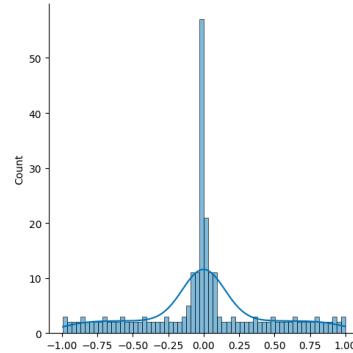


Figure 2: Dynamic tree quantization.

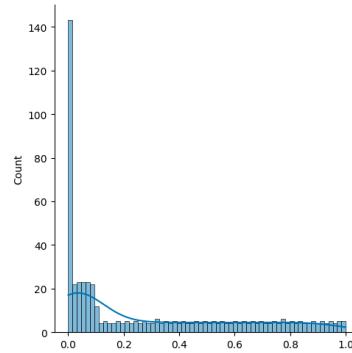
1. 如上图所示，动态树量化使用动态可变的阶码位 (Exponent) 和尾数位 (fraction)，包含 4 个部分组成：1. 第一位为符号位；2. 接下来的若干个 0 表示指数的大小 (如上图所示 2 个 0，表示 10^{-2})；3. 接下来为 1 的 bit 为指示位 (indicator bit)；指示位之后的所有 bit 用来使用线性量化的小数位 (第 4 部分)；因此，上图表示的小数为 $-10^{-2} \times 0.6 = 6 \times 10^{-3}$ ；
2. 通过移动 indicator bit 可以调整动态范围以及精度；如上图所示，8bit 的动态树量化，可以具有 10^{-7} 的大范围 (large exponent) 或者 $1/63$ 的精度 (precision)；
3. 与线性量化相比，动态树量化对于非均匀分布具有更好的绝对以及相对量化误差；
4. 动态树量化被严格定义为量化 $[-1.0, 1.0]$ 范围内的数字 (通过对 Tensor 使用绝对值最大值进行归一化)
5. 具体实现过程如下：
 1. 不断调整 indicator bit 的位置，在动态范围和精度之间调整；
 2. 8 bit 能够表示的最大数值个数为 $2^8 = 256$ ；
 3. 尾数部分表示范围在 $[0.1, 1]$ 之间的浮点数，此浮点数是首先确定尾数能表示的元素个数，然后通过区间二分得到的，参照下图中的 Boundaries 变量；(具体实现时设置变量 fraction_items 表示浮点数个数加一 ($k + 1$)，然后通过均匀插值，得到 k 个等分的区间，然后取每个区间的中点)，此过程构造二叉树，因此称为 Dynamic tree quantization；
 4. 下图中的 $i = 0$ 表示 8bit 为 1000 0001 的情况，即指数位(Exponent)所占bit数为 6，此时最后一个bit 表示 indicator bit，没有尾数位的空间，因此只能表示唯一的一个数（只能取 $[0.1, 1]$ 之间的中点）；因此，表示的2个值为 $[5.500000384017767e - 07, -5.500000384017767e - 07]$
 5. 下图中的 $i = 6$ 表示 8bit 为 1100 0000 的情况，指数位所占bit 为 0，尾数位占用6个bit，此时，尾数可以表示 $2^6 = 64$ 个数，因此对区间 $[0.1, 1]$ 二分得到共 64 个浮点数；

	Fraction_items (i*2 + 1)	Boundaries (torch.linspace(0.1, 1, fraction_items))	Means (Boundaries[:-1] + Boundaries[1:]) / 2	Data (10 ^ (-6+i)) * Means
i=0	2	[0.1000, 1.0000]	[0.5500]	Means * (10^-6)
i=1	3	[0.1000, 0.5500, 1.0000]	[0.3250, 0.7750]	Means * (10^-5)
i=2	5	[0.1000, 0.3250, 0.5500, 0.7750, 1.0000])	[0.2125, 0.4375, 0.6625, 0.8875]	Means * (10^-4)
i=3	9	[0.1000, 0.2125, 0.3250, 0.4375, 0.5500, 0.6625, 0.7750, 0.8875, 1.0000]	[0.1562, 0.2688, 0.3812, 0.4938, 0.6063, 0.7188, 0.8312, 0.9438]	Means * (10^-3)
i=4	17	[0.1000, 0.1562, 0.2125,..., 0.9438, 1.0000]	[0.1281, 0.1844, 0.2406,..., 0.9156, 0.9719]	Means * (10^-2)
i=5	33	[0.1000, 0.1281, 0.1562, 0.1844,..., 0.9438, 0.9719, 1.0000]	[0.1141, 0.1422, 0.1703, 0.1984,..., 0.9297, 0.9578, 0.9859]	Means * (10^-1)
i=6	65	[0.1000, 0.1141, 0.1281, 0.1422, 0.1562, 0.1703, 0.1844,..., 0.9156, 0.9297, 0.9438, 0.9578, 0.9719, 0.9859]	[0.1075, 0.1211, 0.1352, 0.1492, 0.1631, 0.1773, 0.1913,..., 0.9886, 0.9927, 0.9967, 0.9988, 0.9948, 0.9789, 0.9938]	Means * (10^0)

6. 动态树量化得到的 8bit 有符号数的分布如下图所示：



4. 由于 Adam 中的二阶矩估计都是平方项，严格为非负的，因此可以省略符号位。同时观察到对于大语言模型，adam 二阶矩在训练过程中变化了 $3 \sim 5$ 个数量级，因此对于二阶矩，对尾数使用固定位；8bit 无符号数的分布如下：



3. 稳定的嵌入层 (stable embedding layer):

1. stable embedding layer 的目的是：提高带有词嵌入的模型训练过程的稳定性 (improve stability during optimization for models with word embeddings)
2. 背景：
 1. 高度可变的梯度 (Highly variable gradients) 可能会导致不可预测的优化行为和不稳定性，表现为梯度发散或者爆炸；由于量化过程中引入的噪声，低精度优化器可能会放大梯度更新的方差；实验中发现，8bit 优化器对于卷积网络是稳定的，**word embedding** 是不稳定的主要来源；
 2. word embedding 不稳定的主要原因是：它是一个稀疏层，输入分布不均匀，可以产生比其他层大100倍的最大梯度幅度； (it is a sparse layer with non-uniform distribution of inputs which can produce maximum gradient magnitudes 100x larger than other layers.)

3. Stable Embedding Layer 使用如下方法：

1. 使用 Xavier uniform initialization；使用均匀分布初始化具有比正态分布更少的极值 (the uniform distribution initialization has less extreme values than a normal distribution)；Xavier Uniform initialization 是从均匀分布 $U(-a, a)$ 中采样得到的，如下图所示，其中 gain 默认为 0 :

$$a = \text{gain} \times \sqrt{\frac{6}{\text{fan_in} + \text{fan_out}}} \quad (3)$$

2. 在 add position embeddings 之前对 word embedding 使用 layer norm：使用 layer norm 之后，embedding 在 train 和 infer 时都保持大约为 1 的方差(maintains a variance of roughly one both at initialization and during training)（计算 layer norm 时，使用的 是有偏的方差，`torch.var(input, unbiased=False)`），word embedding 的维度为 `(batch_size, seq_len, embed_dim)`，layer norm 只对最后一维的 `embed_dim` 进行归一化；
3. 对 embedding layers 使用 32 bit 的优化器状态：这是唯一使用32bit 状态的优化器 (This is the only layer that uses 32-bit optimizer states)，但是对于 embedding layer 的 weight 和 gradient，依然使用通用的精度 (通常是 16-bit) (We still use the standard precision for weights and gradients for the embedding layers – usually 16-bit)

5. 总的流程：

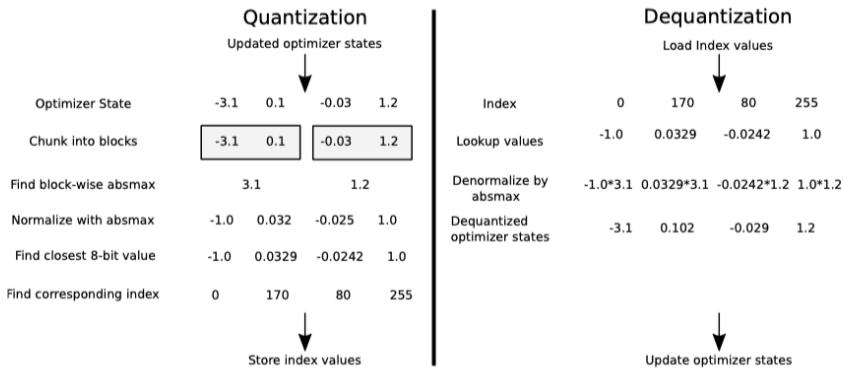


Figure 1: Schematic of 8-bit optimizers via block-wise dynamic quantization, see Section 2 for more details. After the optimizer update is performed in 32-bit, the state **tensor is chunked into blocks, normalized by the absolute maximum value of each block**. Then dynamic quantization is performed, and the **index is stored**. For dequantization, a lookup in the index is performed, with subsequent de-normalization by multiplication with the block-wise absolute maximum value. Outliers are confined to a **single block through block-wise quantization, and** their effect on normalization is limited.

如上图所示：

1. 量化过程：以 32 位执行优化器更新后，状态张量被分为不同的块（chunk into blocks），并通过每个块内的绝对值最大值进行归一化。然后进行动态量化，并存储索引；
2. 反量化过程：为了进行下一次更新，需要将 8bit 存储的优化器状态反量化为 32bit 的形式，首先通过查找表找出索引对应的浮点数，然后与 block 绝对值最大值进行相乘得到反归一化的结果（denormalize），以此 32bit 的形式进行更新；
4. Todo: optimizer compare

10. Dataloader

1. 设置 `pin_memory=True` 可以将数据加载到 锁页内存（page-locked memory）中，这样在训练过程中，将数据转移到GPU的速度会变快；
2. 默认的 `num_workers=0` 表示只使用主进程加载数据，将这个值设置为 4 或者 8 可以加速读取；
3. 将 `pin_memory=True` 和 `non_blocking=True` 搭配使用，`non_blocking=True` 表示将数据转移到GPU显存的操作是异步（asynchronously）进行的，如下面代码所示：第一行的转移（transfor）操作是非阻塞的，第二行代码在第一行代码启动之后立刻执行（不会等第一行代码的完成），直到后续的代码需要用到变量 `x`，才会等待第一行命令执行完成（如第4行代码所示）；

```

1 | x = x.pin_memory().to('cuda', non_blocking=True)
2 | pre_compute()
3 | ...
4 | y = model(x)

```

11. DeepSpeed Zero

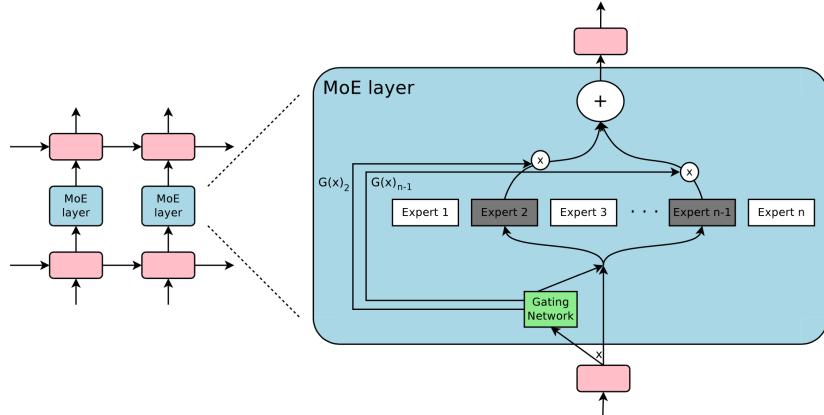
1. 快速判断是否使用 DeepSpeed：
 1. 单块 GPU 可以放置模型，并且有足够的空间可以适应小批量（small batch size）的训练，此时不需要使用 DeepSpeed；
 2. 模型无法放置在单个 GPU 上，或者无法容纳 small batch size，此时考虑使用 DeepSpeed + CPU Offload，对于更大的模型，使用 NVMe Offload；

12. Sparsity 稀疏模型

1. MOE (Mixture-Of-Experts) 概述：
 1. 很多论文报告称：通过将专家混合（MOE）集成到 Transformer 模型中，训练速度提升了 4-5 倍，同时推理速度更快；
 2. 由于更多参数量的模型可以带来更好的性能，因此 MOE 可以在不增加训练成本（training cost）的情况下将模型参数量提升一个数量级；
 3. MOE 最主要的缺点是它需要大量的 GPU 显存，几乎比同参数量的稠密模型大一个数量级（almost an order of magnitude larger than its dense equivalent）。
 4. 一个直接的权衡方案是：在一个 2-3 倍小的基础模型上使用几个专家，而不是在一个 5 倍小的模型上使用数十个或几百个专家。（you can use just a few experts with a 2-3x smaller base model instead of dozens or hundreds experts leading to a 5x smaller model）
2. Sparsely-Gated Mixture-of-Experts layer (MoE): (2017 年 1 月)
 1. 背景：
 1. 条件计算（Conditional computation）是指对于每个样本，模型中只有部分网络处于激活状态。条件计算在理论上被认为是一种大幅度增加模型容量的同时不需要成比例增加计算量的方法（Conditional computation, where parts of the network are active on a per-example basis, has been proposed in theory as a way of dramatically increasing model capacity without a proportional increase in computation）；

2. 对于通常 (typical) 的深度学习模型, 对于样本都要激活整个模型, 这导致了随着模型大小和训练样本数量的增加, 训练成本大致成二次方激增。

2. 方案:



1. 如上图所示: 本文实现条件计算的方法是: 引入一种新型的通用神经网络组件: 稀疏门控专家混合层 (Sparsely-Gated Mixture-of-Experts Layer (MoE)) , MoE 层由多个专家组成, 每个专家都是一个简单的前馈神经网络 (FFN), 以及一个可训练的门控网络 (trainable gating network) , 这个门控网络选择专家的稀疏组合来处理每个输入; 网络的所有部分都通过反向传播联合训练;
2. MoE 层包含 n 个专家网络 ("expert network") E_1, \dots, E_n 和一个门控网络 ("gating network") G , 门控网络的输出是一个稀疏的 n 维向量。每个 expert network 拥有独立的参数, 虽然原则上只要求每个专家的输入和输出维度相同。但在本论文中, 只考虑每个专家都是拥有相同结构 (但是参数不同享) 的前馈网络。

1. MoE 层:

1. 设对于输入 x , gating network 的输出为 $G(x)$, 第 i 个专家网络的输出为 $E_i(x)$, MoE 模块的输出为:

$$y = \sum_{i=1}^n G(x)_i E_i(x) \quad (4)$$

根据 $G(x)$ 输出稀疏性来节省计算; 当 $G(x)_i = 0$ 时, 就不需要计算 $E_i(x)$ 的值;

2. 如果专家的数量特别多的时候, 可以使用分层 (hierarchical) MoE; 在分层 MoE 中, 主门控网络 (primary gating network) 选择 “专家”的稀疏加权组合 (a sparse weighted combination of “experts”), 每个“专家”本身是具有自己门控网络的二级专家混合 (a secondary mixture-of-experts with its own gating network)。

2. 门控网络 ("Gating Network")

1. 一个简单的门控网络选择是 **Softmax Gating**: 使用一个可训练的权重矩阵 W_g 乘以输入变量 x , 然后使用 softmax 函数归一化: $G_\sigma(x) = \text{softmax}(xW_g)$ 。这种方法的缺点是: 这种方式的 Gating Network 不是稀疏 (non-sparse) 的, 因此并没有节省计算;

2. 论文提出的方案是 **Noisy Top-K Gating**:

$$G(x) = \text{Softmax}(\text{KeepTopK}(H(x), k)) \quad (5)$$

$$H(x)_i = (x \cdot W_g)_i + \text{StandardNormal()} \cdot \text{Softplus}((x \cdot W_{\text{noise}})_i) \quad (6)$$

$$\text{KeepTopK}(v, k)_i = \begin{cases} v_i & \text{if } v_i \text{ is in the top } k \text{ elements of } v \\ -\infty & \text{otherwise.} \end{cases} \quad (7)$$

$$\text{Softplus}(x) = \log(1 + e^x) \quad (8)$$

如上式所示, 在 Softmax Gating 的基础上增加了2个组件: 稀疏性和噪声 (sparsity and noise) ;

在进行 Softmax 函数之前, 加上可学习噪声, 然后只保留 top k 个值, 将余下的值设置为 $-\infty$ (经过softmax 之后为 0); 添加噪声有助于负载平衡 (improve load balancing) ; 这里的 $(x \cdot W_g)_i$ 中的下标 i 表示向量的第 i 个维度;

3. 应对训练挑战

1. batch size 的收缩问题 (the shrinking batch size) :

1. 问题: 在现代的 GPUs 上, 大的 batch size 对于提升计算效率是有必要的, 因为这样可以摊薄参数加载和更新的开销 (amortize the overhead of parameter loads and updates) 。对于 batch size 为 b 的训练样本, 如果从 n 个专家中选择选择 k 个计算, 则对于每个专家, 它的输入批次大小为 $\frac{b}{n} \times k \ll b$ 。这会导致当专家数量增加时, naive 的 MoE 实现非常低效; 论文提出使用如下技术增加批次大小;

2. 解决方案: 混合数据并行和模型并行 (Mixing Data Parallelism and Model Parallelism) :

1. 对于网络中的普通层和 gating network，使用数据并行，每个节点都拥有一份模型参数； (**data-parallel replicas (for the standard layers and the gating networks)**)
 2. 对于专家 expert，则使用模型并行，每个节点包含所有专家的一个子集 (**model-parallel shards (each hosting a subset of the experts)**)；
 3. 通过数据并行和模型并行的混合，对于包含 d 个节点的分布式环境，每个专家 (each expert) 得到的样本数为 $\frac{b}{n} \times k \times d$ ；
 4. 对于层级 MoE，主门控网络使用数据并行，二级MoE使用模型并行 (the primary gating network employs data parallelism, and the secondary MoEs employ model parallelism)
 5. 两种并行的结合，允许通过成比例增加节点数来增加专家数量；保持每个专家处理的样本数不变 (keeping the batch size per expert constant)，总的 batch size 增加；每个节点所需要的显存和带宽要求也保持不变；
2. 网络带宽问题 (Network Bandwidth)
1. 分布式计算的另一个问题是网络带宽问题。训练过程中，大多数通讯涉及通过网络发送专家的输入和输出 (most of the communication involves sending the inputs and outputs of the experts across the network)。
 2. 为了保持计算效率，专家的计算与其输入和输出大小的比率必须超过计算设备的计算能力与网络容量的比例； (the ratio of an expert's computation to the size of its input and output must exceed the ratio of computational to network capacity of the computing device)；对于 GPUs 来说，这个比值可能是千比一 (thousands to one)；
 3. 在本论文的实验中，使用的 expert 是包含数千个RELU-activated units 的单层网络 (we use experts with **one hidden layer** containing thousands of RELU-activated units)。由于 expert 中权重矩阵的大小为 $input_size \times hidden_size$ 和 $hidden_size \times output_size$ ，因此 expert 的计算相对于 input 和 output 的比例是 隐含层的大小 (the ratio of computation to input and output is equal to the size of the hidden layer)；因此，可以通过使用更大的隐含层维度或者更多的隐含层来提高计算效率；
4. 平衡专家的使用 (balancing expert utilization)
1. 我们观察到，Gating network 倾向与收敛到总是为少数专家产生较大权重 (always produces large weights for the same few experts) 的状态；这种不平衡是自我强化的 (self-reinforcing)，因为受青睐的专家训练更快，因此更容易被 gating network 选择；
 2. 本论文采用软约束的方法 (soft constraint approach)，定义一个额外的损失函数 $L_{importance}$ ，鼓励所有的 expert 有相同的重要性：
 1. 定义一个expert 对于一个batch 训练样本的重要性为该 expert 在此batch 样本上的门控值之和 (define the importance of an expert relative to a batch of training examples to be the batchwise sum of the gate values for that expert)；
 2. 定义 $L_{importance}$ 为重要性值集合的变异系数的平方乘以手动调整的缩放因子 (This loss is equal to the square of the **coefficient of variation** of the set of importance values, multiplied by a hand-tuned scaling factor $W_{importance}$)：
$$\text{Importance}(X) = \sum_{x \in X} G(x)$$

$$L_{importance}(X) = w_{importance} \cdot CV(\text{Importance}(X))^2 \quad (9)$$

上式中， CV 表示变异系数 (coefficient of variation)，其计算方式为标准差与平均值的比值 $(\frac{\sigma}{\mu})$ ； 变异系数是概率分布离散程度的一个归一化度量；上式中， $\text{Importance}(X)$ 为 1 维向量，其大小等于 expert 的数量；
 3. 虽然使用 $L_{importance}$ 可以使得 expert 拥有同等重要性 (can ensure equal importance)，但是 expert 可能仍然收到数量非常不同的训练样本；这会导致分布式训练过程中的内存和性能问题；因此，引入第二个损失函数 L_{load} 用以保证负载均衡 (load-balancing purposes)：
 1. 由于 expert 接收的样本数量是离散值，因此不能用于反向传播；
 2. 论文定义了一个平滑的估计函数 (smooth estimator) $Load(X)$ ，用以表示一个 batch 的 X 个样本分配给专家的分配情况。定义 $P(x, i)$ 表示 $G(x)_i$ 不为 0 的概率 (x 表示 batch 中的一个样本， i 表示 expert 的序号)；由于只有当 $H(x)_i$ 是 $H(x)$ 中前 k 大的元素时，才会有 $G(x)_i \neq 0$ ，因此：
$$P(x, i) = \Pr((x \cdot W_g)_i + \text{StandardNormal}() \cdot \text{Softplus}((x \cdot W_{noise})_i) > kth_excluding(H(x), k, i)) \quad (10)$$

上式中， $kth_excluding(v, k, i)$ 表示 v 中不包含 i 的第 k 大元素 (也就是排序时不考虑 i ，然后取排序后的第 k 个最大值，这样可以方便比较 i 属于或者不属于前 k 大)。上式中 \Pr 表示事件发生概率；上式可以转化为：

$$P(x, i) = \Pr\left(\frac{(x \cdot W_g)_i - k\text{th_excluding}(H(x), k, i)}{\text{Softplus}((x \cdot W_{\text{noise}})_i)} + \text{StandardNormal}() > 0\right)$$

进一步可以转化为:

$$P(x, i) = \Phi\left(\frac{(x \cdot W_g)_i - k\text{th_excluding}(H(x), k, i)}{\text{Softplus}((x \cdot W_{\text{noise}})_i)}\right) \quad (12)$$

其中, Φ 表示标准正态分布的累计分布函数。估计函数如下:

$$\text{Load}(X)_i = \sum_{x \in X} P(x, i) \quad (13)$$

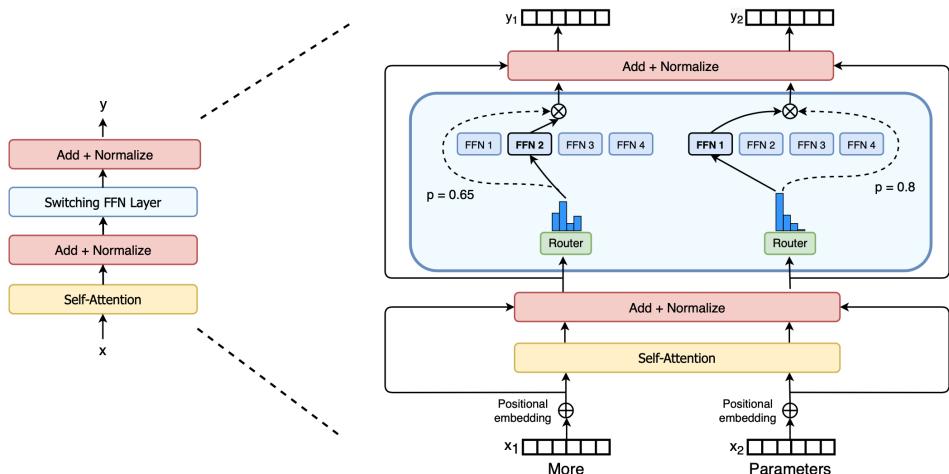
3. 由此可以定义 Load-Balancing loss 如下:

$$L_{\text{load}}(X) = w_{\text{load}} \cdot CV(\text{Load}(X))^2 \quad (14)$$

其中, CV 表示变异系数, 衡量专家负载的分布情况;

3. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

1. 背景: MoE 模型为每个传入的样本选择不同的参数, 得到了稀疏激活的模型 (sparsely-activated model) : 参数量惊人, 但是计算成本固定 (with an outrageous number of parameters—but a constant computational cost) ; 尽管 MoE 取得了一些显著成功, 但是复杂性、通讯成本和训练不稳定 (complexity, communication costs, and training instability) 等阻碍了 MoE 的更大范围使用;
2. 本论文提出了 Switch Transformers, 简化了 MoE 路由算法 (routing algorithm) , 设计了降低通讯量和计算成本的直观改进模型 (design intuitive improved models with reduced communication and computational costs.) ;
3. Switch Transformer 的结构如下图:



1. 总览:

1. 如图所示, 将 Transformer 中 dense FFN 层替换为稀疏的 Switch FFN layer; 这一层对序列中的每一个 token 独立操作; **Switch FFN** 层返回所选择的 FFN (专家) 乘以路由门控值 (The switch FFN layer returns the output of the selected FFN multiplied by the router gate value)

2. 简化的稀疏路由 (Simplifying Sparse Routing) :

1. 在本论文中, 每一个 token 只路由到一个 expert, 实验证明这种简化可以保持模型质量, 减少路由计算, 性能更好 (this simplification preserves model quality, reduces routing computation and performs better)

2. Switch layer (相比于通常的 MoE) 包含 3 方面的好处:

1. 减少了路由计算 (router computation is reduced) , 因为只需要将 token 路由到某一个专家;
2. 每个 expert 的 batch size 至少减半, 因为每个 token 只被路由到一个 expert (以前的 MoE 工作通常路由到至少2个expert)
3. 简化了路由实现并且降低了通讯成本;

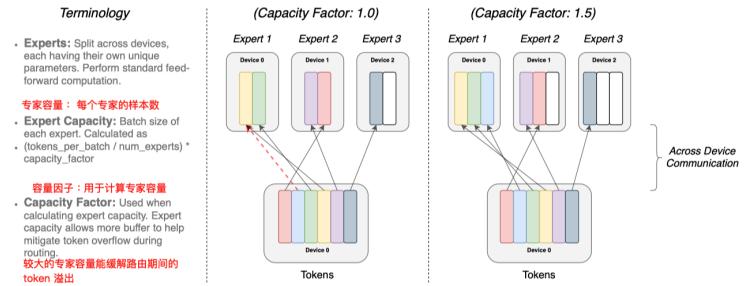
3. 高效稀疏路由 (Efficient Sparse Routing) (本论文来源于 Google, 因此使用 TensorFlow 和 TPU)

1. 分布式 Switch layer 实现:

1. 由于论文的代码基于 Mesh-Tensorflow, 张量的维度在编译时静态确定, 但由于训练和推理时的路由决策 (routing decisions) , 使得计算是动态的; 因此, 需要提前设置好专家容量 (expert capacity) ;
2. 专家容量 (expert capacity) 是指每个专家处理的 tokens 数量, 设置如下:

$$\text{expert capacity} = \left(\frac{\text{tokens per batch}}{\text{number of experts}} \right) \times \text{capacity factor} \quad (15)$$

其中， capacity factor 是容量因子，如下图所示， tokens per batch = 6, number of experts = 3;



1. 每个 expert 都处理固定数量 (fixed batch-size) 的 tokens, 此数量由 capacity factor 控制;
2. 每个 token 被路由到具有最高概率的 expert, 但是每个 expert 能处理的 token 数是固定的 (专家容量, expert capacity) , 因此, 如果 token 分配不均匀, 会导致某些 expert 溢出 (overflow) (如上图中间的红色虚线箭头所示, 有 3 个 token 被路由到了 Expert1, 超过了专家容量) ;
3. 如果出现溢出, 则这些超过专家容量的 token 不会被计算 (computation is skipped) , 这些 token 的 representation 直接通过残差链接到下一层 (这些 token 被称为 **dropped tokens**) ;
4. 通过将容量因此 Capacity Factor 设置为大于 1, 会创建额外的缓冲区, 以适应 token 分配的不均匀 (上图中右侧, Capacity Factor = 1.5)
5. 较大的容量因子可以缓解溢出, 但是会增加计算和通讯成本;
6. 实验发现, 较低的 token 丢弃率对于稀疏模型的扩展非常重要 (Empirically we find ensuring **lower rates of dropped tokens** are important for the scaling of sparse expert-models.)
7. 实验中, 没有发现 token 的丢弃率与专家数量存在依赖关系 (we didn't notice any dependency on the number of experts for the number of tokens dropped (typically < 1%)) ;

2. 负载平衡损失 (Load Balancing Loss)

1. MoE 中使用了 importance-weighting loss 以及 load-balancing loss 作为额外的损失, 鼓励专家之间的负载均衡; Switch Transformer 简化了此设计;
2. 每一层的负载损失如下:

$$\text{loss} = \alpha \cdot N \cdot \sum_{i=1}^N f_i \cdot P_i \quad (16)$$

其中, N 表示 expert 数量, f_i 是分配给专家 i 的 token 的比例, 其表达式如下, 其中, T 表示 token 的总数, \mathcal{B} 表示所有 token 的集合, x 表示单个 token:

$$f_i = \frac{1}{T} \sum_{x \in \mathcal{B}} \mathbb{1}\{\arg\max p(x) = i\} \quad (17)$$

而 P_i 表示分配给专家 i 的路由概率的比例 (the fraction of the router probability allocated for expert i) :

$$P_i = \frac{1}{T} \sum_{x \in \mathcal{B}} p_i(x) \quad (18)$$

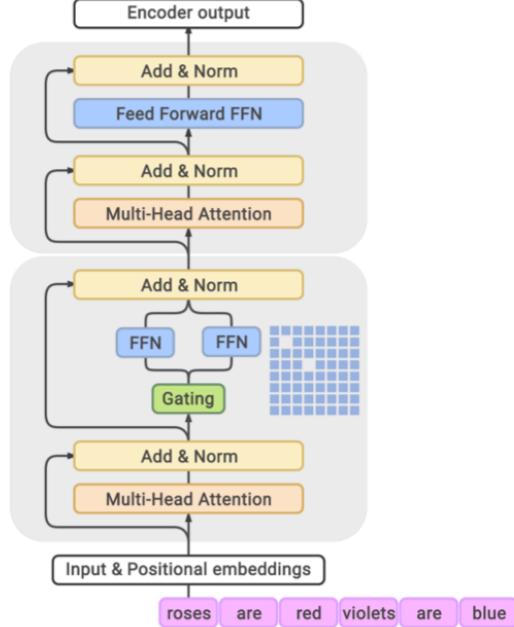
其中, $p_i(x)$ 表示将 token x 路由到专家 i 的概率;

3. 当均匀路由时, 式 (16) 的 loss 值最小, 此时 f_i 和 $p_i(x)$ 的值都为 $1/N$, 式 (16) 的损失函数中还乘以了专家数量 N , 这是为了保证当专家数量变化时, 负载平衡损失值保持不变, 因为 $\sum_{i=1}^N (f_i \cdot P_i) = \sum_{i=1}^N (\frac{1}{N} \cdot \frac{1}{N}) = \frac{1}{N}$;
4. 损失函数中还包括超参数 α , 它控制负载均衡项在总 loss 中的占比, 论文中使用 $\alpha = 10^{-2}$;
4. 提升训练和微调的技巧 (Improved Training and Fine-Tuning Techniques)
 1. 问题: 1. Switch-layer 中存在的硬交换 (路由) 决策 (hard-switching (routing) decision) , 可能引入不稳定性; 2. BF16 等低精度格式可能会加剧路由层中 softmax 计算中的问题;
 2. 解决方案:

1. 选择精度 (*Selective precision*) : 选择性地将模型的局部部分转化为 FP32 精度可以提高稳定性，同时避免 float32 的昂贵通讯成本；具体而言：**FP32 精度仅仅在路由函数内部使用** (the float32 precision is only used **within the body of the router function**—on computations local to that device)。路由函数的输入被转化为了 FP32 格式，并且在当前 expert 所在的 device 上完成计算，操作结果被转化为了 BF16 格式，因此不会有 float32 的 broadcast 操作；
2. 更小的初始化值 (Smaller parameter initialization) : Switch Transformer 的权重矩阵初始化是从均值为 $\mu = 0$ ，标准差 $\sigma = \sqrt{s/n}$ 的截断高斯分布中采样的（只采样平均值2个标准差内的值），其中， n 是权重的输入维度 (fan-in); s 是比例 (scale) 超参数；论文将比例参数 s 从默认值 1 调整为 0.1，实验验证，调整比例参数之后，既提高了模型质量 (**average model quality**, 使用负对数困惑度衡量) 又降低了训练的不稳定性 (每个实验用 3 个不同的seed 进行三次，统计负对数困惑度的均值和标准差)
3. 正则化稀疏模型 (Regularizing large sparse models) :
 1. Switch transformer 相比计算量匹配 (FLOP matched) 的 dense 模型具有更多参数，可能导致下游任务微调时出现更严重的过拟合；
 2. 论文采用的方法是对专家使用更高的 dropout rate (use higher expert dropout)。论文实验证明：**设置 expert dropout rate = 0.4, 同时其他层 (non-expert layer) 的 dropout rate 维持在 0.1 时的效果最好**；同时实验证明：对所有层同时使用较高的 dropout rate 会导致较差的效果；
5. 缩放属性 (Scaling Properties)
 1. 专家数量是扩展模型最有效的维度，同时，对于 Switch Transformer，增加专家数量可以使得计算成本大致固定，因为每个token 只选择一个 expert；
 2. 以训练步数为基础的 scale 结果 (Scaling Results on a Step-Basis) :
 1. 在计算预算相同的情况下 (FLOPS)，增加专家数会提升模型质量 (通过 perplexity 衡量)
 2. 增加专家数量会使得**样本的训练效率更高** (increasing the number of experts leads to more sample efficient models)，包含 64 个expert 的 Switch-Base 模型训练速度相比稠密的 T5-Base 模型提升了 7.5 倍 (这里的提升是以训练的步数 (step) 衡量的，Switch-Base 模型训练 6w 个step 达到了 T5-Base 训练 45w 个step 相同的困惑度)
 3. 包含更多专家模型的样本效率更高：对于固定数量可观测的 token，大模型学习的更快 (larger models are also more **sample efficient**—learning more quickly for a fixed number of observed tokens.)
 3. 以挂钟时间为基础的 scale 结果 (Scaling Results on a Time-Basis)
 1. 虽然 Switch Transformer 相比与 baseline 模型，FLOPs per token 大致相同，但是引入了跨设备的额外通讯成本以及路由机制的额外计算 (incurs **additional communication costs across devices** as well as the **extra computation of the routing mechanism.**)，因此，基于训练步数 (step) 观察到的样本效率的提高并不意味着相同训练时间 (wall-clock) 会得到更好的模型；
 2. 因此，这里存在一个问题：对于固定的训练时长 (training duration) 以及计算预算 (computational budget)，**应该训练稠密模型，还是稀疏模型呢？**
 3. 论文实验证明：对于固定的训练时间和计算预算，Switch Transformer 可以显著提高训练速度，包含 64 个专家的 Switch-base 得到与稠密模型相似的困惑度只需要 1/7 的时间；(7倍的加速)
 4. 与更大的稠密模型对比 (Scaling Versus a Larger Dense Model)
 1. 论文将 Switch-Base64e 与 T5-Large 对比，结论如下：尽管 T5-large 模型的 Flops per token 是 T5-base 的 3.5 倍 (T5-large 参数量是 base 的 3.5 倍)，但是 Switch-base 模型的样本效率还是稍微高一些 (以 train step 衡量)，同时，以训练时间衡量，Switch-base 64e 相比 T5-large 有 2.5 倍的加速 (相比于 T5-base 是 7 倍的加速)；
 6. 下游任务微调：
 1. 实验观察到推理和知识密集型任务都取得了显著的进步；验证了我们的架构，不仅是预训练良好的架构，而且可以通过微调在下游任务上体现出来；
 7. 蒸馏
 1. 由于 Switch Transformer 和对应的稠密模型 FLOP 匹配，因此非专家层包含相同的维度；**使用非专家层权重初始化 student 模型会带来改进**； (initializing the dense model with the non-expert weights yields a modest improvement.)
 2. 使用 0.25 的教师概率和 0.75 ground truth label 的混合会带来提升 (we observe a distillation improvement using a **mixture of 0.25 for the teacher probabilities and 0.75 for the ground truth label.**)
 3. 结合上面 2 个技术，**蒸馏之后的模型只有 1/20 的参数，但是保留了大型稀疏模型 30% 的质量增益**；质量增益是指：(蒸馏后的 student 模型相比 base-line 模型的提升) / (稀疏的 teacher 模型相比于 baseline 的提升) (we preserve $\approx 30\%$ of the quality gains from the larger sparse models with only $\approx 1/20$ of the parameters.)
 4. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts
 1. 摘要：

1. 论文提出并开发了一系列名为 GLaM (Generalist Language Model) (通才语言模型) 的语言模型，使用稀疏激活的 MoE 结构来扩展模型容量，同时与稠密模型相比，训练成本大大降低；
2. 最大的 GLaM 模型包含 1.2 万亿 (trillion) 的参数量，是 GPT-3 的 7 倍；但是训练所消耗的能量 (energy) 只有 GPT-3 的 $1/3$ ，同时推理 (inference) 时进行的浮点数运算只有 GPT-3 的一半 (requires half of the computation flops for inference)，同时在 29 个 NLP 任务的 zero、one、few-shot 能力更好；
2. 背景：
 1. GPT-3 等模型证明了上下文学习 (in-context learning) 对于少样本甚至零样本泛化的可行性 (few-shot or even zero-shot generalization)，意味着只需要很少的标注数据就可以实现在 NLP 任务上有良好的表现；虽然有效，但是进一步扩展变得非常昂贵并且消耗大量能源；
3. 本文贡献：
 1. 提出了一系列名为 GLaM 的通用语言模型，在密集计算和条件计算之间取得了平衡 (that strike a balance between dense and conditional computation)；
 2. GLaM 最大的版本包含 1.2 万亿 (T) 参数，每个 MOE 层包含 64 个 expert，输入数据中的每个 token 只激活 8% 的参数 (96.6B 参数，占 1.2T 的 8%)，但是相比于 GPT-3 (175B 参数)，GLaM 在 zero、one、few-shot 等方面优于 GPT-3；
 3. 论文实验表明：即使对于大模型，如果目标是生成高质量的自回归语言模型，也不应该为了数据数量而牺牲数据质量 (even for these large models, data quality should not be sacrificed for quantity if the goal is to produce a high-quality auto-regressive language model.)
 4. GLaM 首次证明了：稀疏的 decode-only 语言模型在 few-shot 上下文学习中比相同计算量的稠密模型具有更高的性能 (sparse decoder-only language models can be more performant than the dense architectures of similar compute FLOPs for the first time within the few-shot in-context learning setting at scale)；
 5. 本论文提出的 GLaM 与 Switch Transformer 都拥有万亿参数 (最大参数量分别为 1.2T 和 1.5T)，但是 Switch Transformer 是基于 T5 的 encoder-decoder 结构，而 GLaM 是基于 Decode-Only 结构；同时，Switch Transformer 论文主要探讨了在下游任务的 fine-tuning，而 GLaM 针对 few-shot 场景；
4. 模型结构：

1. GLaM 的模型结构如下：

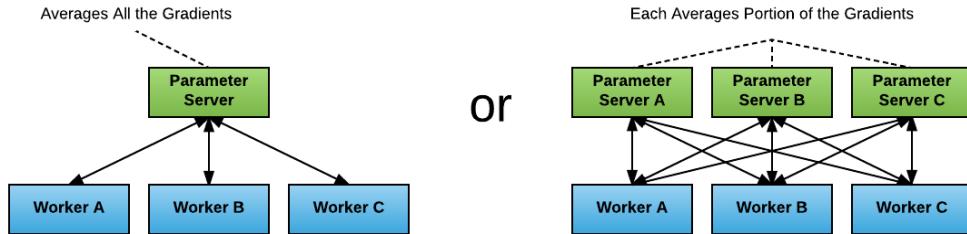


1. GLaM 将每隔一层的 FFN 层替换为 MoE 层 (replace the feed-forward component of every other Transformer layer with an MoE layer)
2. GLaM 中，每个 MoE 层激活的专家数量固定为 2，门控网络对输入序列中的每个 token 选择最好的 2 个专家；对于包含 E 个专家的 MoE 层，共有 $O(E^2)$ 个不同的专家组合；最终的输出是 2 个专家的输出加权和；
5. 训练万亿模型技巧：
 1. 首先使用小规模模型 (smaller-scale model) 验证收敛性，尽早暴露数据集和硬件问题；
 2. 如果梯度中出现了 $NaNs/Infs$ ，则跳过此 batch 的权重更新；
 3. 训练期间如果遇到罕见的大波动 (counteracting rare large fluctuations) 甚至是 $NaNs/Infs$ ，则从早期健康的 checkpoint 恢复训练，并且对输入数据进行随机 shuffle；
6. 实验结论：
 1. 比较 MoE 模型和稠密模型：

1. GLaM(64B/64E) ($n_{params} = 1.2T$, $n_{act_params} = 96.6B$) 与 GPT-3 (175B) 在 zero、one、few-shot 等方面效果更好 (competitive performance)，但是推理时的 FLOPs 只有 GPT3 的一半 (准确值为 55.2%) (GLaM (64B/64E) activates roughly 96.6B parameters per token during inference, which requires only half of the compute FLOPs needed by GPT-3 given the same input.)
2. GLaM 在开放域问答中达到了最佳效果，表明语言模型能够利用其容量来吸收知识 (the additional capacity of GLaM plays a crucial role in the performance gain)；尽管 GLaM (64B/64E) 的激活参数只有 GPT-3 的一半，但是 GLaM 的额外容量在性能增益中至关重要；
3. 数据质量的影响 (Effect of Data Quality)：
 1. 在过滤数据集 (filtered data) 上训练的模型在 NLG 和 NLU 任务上始终表现更好；
 2. 数据质量对 NLG 的任务影响更大； (the effect of filtering is bigger on NLG than that on NLU.)
 3. 预训练数据质量起着至关重要的作用，特别是对于下游任务 (the quality of the pretrained data also plays a critical role, specifically, in the performance of downstream tasks.)
4. 扩展性研究 (Scaling Studies)
 1. 对于稠密模型，每次推理时的 FLOPs 与模型参数 n_{params} 成正比，增大模型，也会提高推理成本；
 2. 对于稀疏模型，每次推理只有部分参数被激活， $n_{params} \gg n_{act_params}$ ；
 3. zero、one、few-shot 的性能与推理时的 FLOPs 计算量成正相关 (the average zero, one and few-shot performance across the generative tasks scales well with the effective FLOPs per prediction which is in turn determined by n_{act_params})
 4. 稠密模型和稀疏模型在小 scale 上表现相似，但是 MoE 模型在较大 scale 上表现更好；
 5. 对于固定计算预算的推理场景，添加更多专家会带来更好的预测性能 (for a fixed budget of computation per prediction, adding more experts generally leads to better predictive performance.)
5. GLaM 的效率
 1. 数据效率 (Data Efficiency)：
 1. 实验表明：GLaM MoE 模型达到与类似计算量 (FLOPs) 的 dense 模型相似的 zero、one、few-shots 性能所需的数据少得多 (注意这里指的 FLOPs 指的是 n_{act_params} ，并不是模型总参数量的对比)； (We first observe that GLaM MoE models require significantly less data than dense models of comparable FLOPs to achieve similar zero, one, and fewshot performance.)
 2. 也就是：当数据总量相同的情况下，MoE 模型表现更好，并且随着训练的 token 数量的增加，差距变大；GLaM 64B/64E (共包含 1.2T n_{params} 以及 96.6B n_{act_params}) 训练 280B 的效果好于训练 300B 的 GPT-3 (包含 175B)
 2. 计算效率和能源效率 (Computation Efficiency & Energy Consumption)
 1. 在下游任务上实现类似性能，训练稀疏激活的模型比训练密集模型需要的计算资源更少 (以 TPU year 衡量)； (We find that to achieve similar performance on downstream tasks, training sparsely activated models takes much less computational resources than training dense models.)
 2. 训练 600B token 的 GLM (64B/64E) 消耗的能源成本是 GPT-3 的 1/3，而得到与 GPT-3 (训练 300B) 性能相近的 280B token 的模型消耗能源是 GPT-3 的 1/6；

多 GPU 优化方法：

1. 数据并行 (Data Parallelism)
 1. 数据并行：是在多个设备上初始化相同的模型，不同设备并行处理不同的数据切片，各自完成前向和后向传播，然后通过 AllReduce 集合通讯 (collective communication) 计算所有设备 (也就是进程, process) (每个设备上包含 1 个进程) 上每个参数值的平均梯度，然后使用此平均梯度更新各个设备上的模型参数 (由于所有模型初始状态相同，并且每次迭代都使用相同的平均梯度，因此不同设备上模型参数保持同步) (data-distributed training works by initializing the same model on multiple different machines, **slicing the batch up** and backprogating on each machine simultaneously, collecting and averaging the resulting gradients, and then updating each local machine's local copy of the model prior to the next round of training.)
 2. 分布式训练架构：Parameter Server (PS) 、All-Reduce:

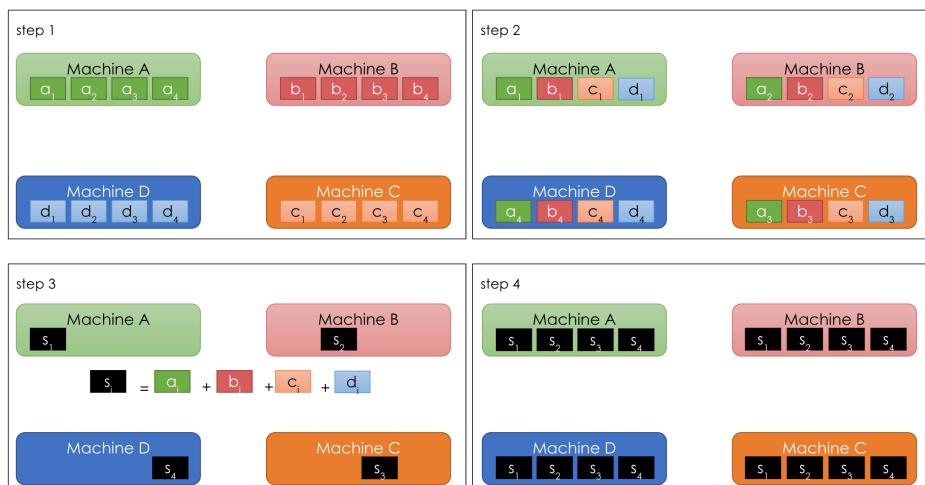
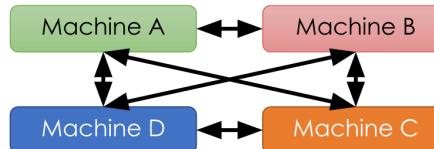


1. 参数服务器 (parameter server)

1. 深度学习中采用的第一个数据并行化技术是 Tensorflow 的 **参数服务器策略 (parameter server strategy)** (最早实现于 2012 年, Google 的 DistBelief 中)
2. 如上图所示, 在 PS 架构中, 包含 2 个部分, 分别是 Worker 和 (Parameter) Server。其工作过程如下:
 1. 每个 Worker 在属于它的数据分片 (batch slice) 上完成前向和后向传播, 然后将计算的梯度发送给 Parameter Server;
 2. Parameter Server 得到所有的 Worker 梯度后, 计算梯度均值 (PS 会等待直到收到所有 Worker 的梯度); 如果包含多个 PS (如上图右侧所示), 则每个 PS 只处理它负责的参数;
 3. 平均梯度被发送给每个 Worker, Worker 各自完成模型参数的更新;
 4. 当所有 Worker 的参数都更新之后, 进行下一个 batch 的训练;
3. PS 架构的主要缺点是:
 1. 每次同步, 都需要所有的 Server 与所有的 Worker 进行通讯, 通讯成本随着系统中的 Worker 数量线性增长, 随着 Worker 数量的扩展, 网络带宽会成为瓶颈;
 2. 如果采用同步更新 (Synchronous) 方式, 则系统中最慢的 Worker 影响整个系统的运行, 其他 Worker 处于空闲等待; 而异步更新方式 (asynchronous) 会导致由于参数不匹配引起的收敛缓慢和训练不稳定;

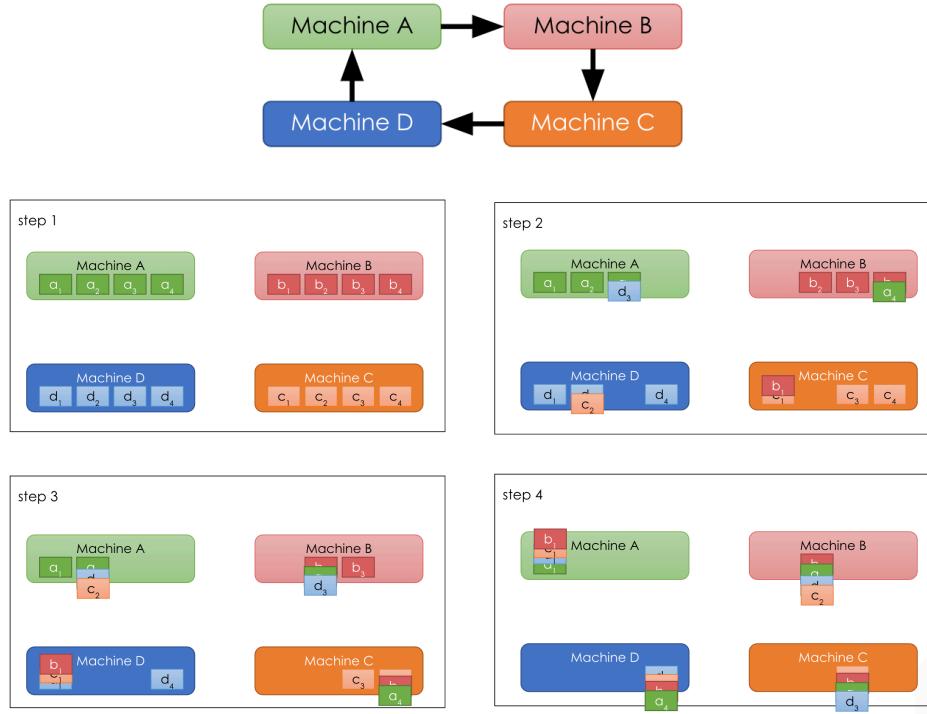
2. All Reduce

1. 在 All Reduce 中, 每个 process 都是 work process, 没有参数服务器;
2. All-Reduce 的原理如下图所示:

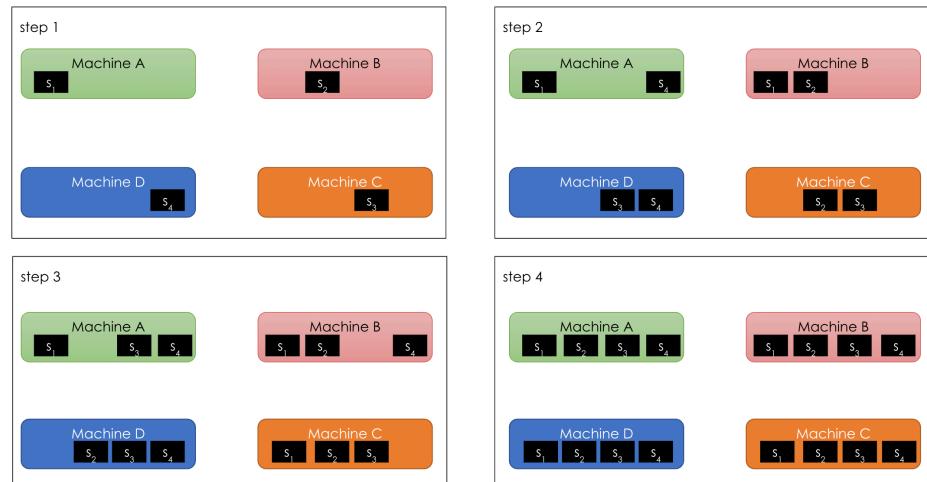


上图中, 假设模型中包含 4 个 chunk 的模型参数 (实际实现中, 将参数按照设备 (GPU) 数量 P 划分为 P 个 chunks), a_1, b_2 分别表示显卡 A (这里的机器指的是 GPU 卡, 而非节点) 上参数 1 和卡 B 上的参数 2; 如上图 Step2 所示, 每张卡将它拥有的模型参数 (梯度) 发送给其他所有卡 (each machine sends its version of all parameters to all other machines) (例如, 卡 A 将 a_2 发送给 B, 将 a_3 发送给 C, 将 a_4 发送给 D); 每张卡对不同的参数聚合; 然后将聚合后的结果发送给其他所有显卡;

3. 广泛使用的一个高效 All Reduce 实现是 Ring-All Reduce (Ring-all-reduce 并不是一类新的算法, 只不过是一个高效实现), 如下图所示:



与 All-Reduce 的主要区别在于，在 Ring-All Reduce 中，所有的设备（指 GPU 卡）设置为一个环状，每张卡将它拥有的模型参数 chunk 发送给下一台机器，如上图 Step2 所示，机器 B 将 b_1 发送给机器 C，C 将 c_2 发送给机器 D，D 将 d_3 发送给机器 A，A 将 a_4 发送给机器 B；经过类似的 Step3 和 Step4，每台机器都包含某个参数 chunk 的所有版本（上图中 Step4 中，Machine A 拥有所有 b_1, b_2, b_3, b_4 ），此过程称之为 Scatter-Reduce，对于 P 台机器的 Ring，需要进行 $P - 1$ 次迭代；进而可以求出每个参数的聚合版本，如下图中 Step1 所示：



与 All-Reduce 不同，在聚合梯度的广播（broadcast）阶段，每台机器也是将梯度发送到下一台机器；此过程称之为 Allgather，同样需要 $P - 1$ 次迭代；

4. 原始的 All-Reduce 和 Ring-All-Reduce 对比：

- 假设模型参数为 N ，设备（GPU）数量为 P ，则 All-Reduce 需要的带宽为 $\frac{N}{P} \times (P - 1) \times 2$ ；（ N/P 表示每台机器应该处理的参数 chunk，2 表示每个设备每次迭代需要发送和接收一个 chunk）（对于 Parameter Server，假设一台设备用作参数服务器，则需要的带宽为 $N \times (P - 1)$ ）
- Ring-All-Reduce 经过完整一轮需要的带宽为 N ，与机器数无关，因此更具扩展性（At each round, we have a constant bandwidth of N that does not depend on the total number of machines P , thus more scalable.）
- 但是 Ring-All-Reduce 的限制因素是每轮都需要 $P - 1$ 次通信，因此速度受到环中最慢连接的限制；
- 当每个 GPU 都正确设置了邻居，则 Ring-All-Reduce 是带宽最优的算法，同时在延迟成本可以忽略的情况下，也是最快的 AllReduce 算法（Given the right choice of neighbors for every GPU, this algorithm is bandwidth-optimal and is the fastest possible algorithm for doing an allreduce (assuming that latency cost is negligible compared to bandwidth)）

- 现有的 SOTA 实现中，All-Reduce 都是通过 two-step 实现的，第一个 step 是 reduce-scatter，第2个 step 是 all-gather；

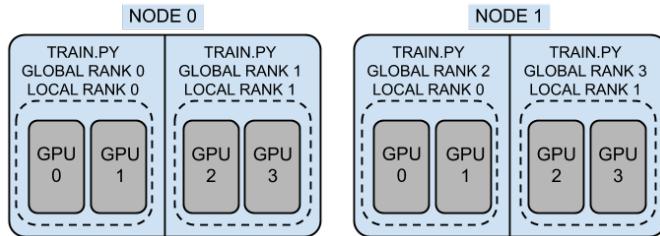
3. PS 和 All Reduce 参考链接:

1. <http://www.juyang.co/distributed-model-training-ii-parameter-server-and-allreduce/>
2. <https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/>

3. Pytorch 的 DDP 实现:

1. DDP 实现的第一步需要完成进程的初始化 (process initialization) , 在DDP训练时, 模型被复制到每个进程 (With DDP, the model is replicated on every process) ; 如下代码所示 (代码展示的是在 single-node-multi-gpus 的场景) :

1. `WORLD_SIZE` 所有进程数的总和;
2. `RANK` 表示当前进程在 `WORLD_SIZE` 个进程中的编号, 范围为 `[0, WORLD_SIZE-1]`; 而 `LOCAL_RANK` 为当前 process 在所在的节点上进程的编号, 范围为 `[0, LOCAL_WORLD_SIZE-1]` (`LOCAL_WORLD_SIZE` 表示 node 上的进程数量), 因此不同 node 上, `LOCAL_RANK` 的编号都是从 0 开始; 如下图所示, 包含 2 个 node, 每个 node 包含 4 张 GPU, 每个进程使用2块GPU, `GLOBAL RANK` 和 `LOCAL RANK` 如图所示:



3. `mp.spawn(fn, args=(), nprocs=1, join=True)` 创建 `nprocs` 个进程, 这些进程以 `fn` 为入口点 (entrypoint), `fn` 函数必须定义在模块顶层 (top level of a module) (多进程的要求); 在每个进程中, `fn` 被调用为 `fn(i, *args)`, 其中, `i` 表示进程索引 (process index), `args` 是传递的参数 tuple; 如下代码中, 第 12 行的 `train` 函数包含 3 个参数, 第一个参数就是 `rank`; `join=True` 表示主进程会等待 spawned process finish (Perform a blocking join on all processes) (来自 python 的 multiprocessing) ; `mp.spawn` 的 `args` 的参数一定会包含 `world_size` 作为参数;
4. 使用 `mp.spawn` 创建多个进程后, 每个进程都需要设置分布式环境 (setup the distributed environment) , 初始化进程组 (`dist.init_process_group`) , 然后才能执行训练函数;
5. `dist.init_process_group` 函数确保每个进程都能通过 `MASTER_ADDR:MASTER_PORT` 与主进程进行通信, 获取其他进程信息, 进而与其他进程建立链接 (all processes will be able to properly connect to the master, obtain information about the other processes, and finally handshake with them) , 在此过程中, 4 个环境变量是必须的 `MASTER_ADDR`, `MASTER_PORT`, `WORLD_SIZE`, `RANK` ; (所有机器都需要设置这 4 个环境变量)

```

1 # multi_init.py
2 import torch
3 import torch.distributed as dist
4 import torch.multiprocessing as mp
5
6 def init_process(rank, size, backend='gloo'):
7     """ Initialize the distributed environment. """
8     os.environ['MASTER_ADDR'] = '127.0.0.1'
9     os.environ['MASTER_PORT'] = '29500'
10    dist.init_process_group(backend, rank=rank, world_size=size)
11
12 def train(rank, num_epochs, world_size):
13    init_process(rank, world_size)
14    print(
15        f"Rank {rank + 1}/{world_size} process initialized.\n"
16    )
17    # rest of the training script goes here!
18
19 WORLD_SIZE = torch.cuda.device_count()
20 if __name__ == "__main__":
21    mp.spawn(
22        train, args=(NUM_EPOCHS, WORLD_SIZE),
23        nprocs=WORLD_SIZE, join=True
24    )

```

2. DDP 的第二步需要完成进程的同步 (process synchronization)

1. 在 DDP 模式下, 需要对代码做如下改动:

1. 任何执行文件 I/O 的方法都应该在主进程中完成 (例如下载数据、保存模型文件等)
2. 否则会导致多个进程同时写入一个文件, 浪费时间并且可能会面临文件系统死锁 (possibly risking a filesystem deadlock) ;

2. 正确写法如下代码所示，指定 master process 进行下载，并且使用 `dist.barrier()` 会阻塞其他进程，直到 master process 完成下载；

```

1 # import torch.distributed as dist
2 if rank == 0:
3     downloading_dataset()
4     downloading_model_weights()
5 dist.barrier()
6 print(
7     f"Rank {rank + 1}/{world_size} training process passed data
download barrier.\n"
8 )

```

2. 应该对 dataloader 使用 `DistributedSampler`，如下所示：

1. `DistributedSampler` 使用 `rank` 和 `world_size` 将跨进程的数据集划分为不重叠的批次 (partition the dataset across the processes into non-overlapping batches)

```

1 def get_dataloader(rank, world_size):
2     dataset = PascalVOCSegmentationDataset()
3     sampler = DistributedSampler(
4         dataset, rank=rank, num_replicas=world_size, shuffle=True
5     )
6     dataloader = DataLoader(
7         dataset, batch_size=8, sampler=sampler
8     )

```

3. Tensor 以及 Model 应该被放置在正确的 device 上，通过 `.cuda(rank)` 或者 `.to(rank)` 设置；

```

1 batch = batch.cuda(rank)
2 segmap = segmap.cuda(rank)
3 model = model.cuda(rank)

```

4. 需要使用 `DistributedDataParallel` 包裹 (wrapped) 模型：

1. 如下代码中第 7 行所示，DDP 包裹了低级别的分布式通信细节，只提供了一个简单的 API，好像 `ddp_model` 是 local model 一样；(DDP wraps lower-level distributed communication details and provides a clean API as if it were a local model)
2. 梯度同步通信发生在后向传递阶段，并且与后向计算重叠 (Gradient synchronization communications take place during the backward pass and overlap with the backward computation)
3. 如下面代码 14 所示，当 `backward()` 返回的时候，`param.grad` 已经包含同步的梯度张量 (When the `backward()` returns, `param.grad` already contains the synchronized gradient tensor)

```

1 def demo_basic(rank, world_size):
2     print(f"Running basic DDP example on rank {rank}.")
3     setup(rank, world_size)
4
5     # create model and move it to GPU with id rank
6     model = ToyModel().to(rank)
7     ddp_model = DDP(model, device_ids=[rank])
8
9     loss_fn = nn.MSELoss()
10    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)
11
12    optimizer.zero_grad()
13    outputs = ddp_model(torch.randn(20, 10))
14    labels = torch.randn(20, 5).to(rank)
15    loss_fn(outputs, labels).backward()
16    optimizer.step()
17
18    cleanup()

```

5. 使用如下代码使得模型可复现 (reproducible)

```

1 torch.manual_seed(0)
2 torch.backends.cudnn.deterministic = True
3 torch.backends.cudnn.benchmark = False
4 np.random.seed(0)

```

4. Launch Utility

1. `torch.multiprocessing.spawn()`

- 上面的实现中，代码中使用 `torch.multiprocessing.spawn()` 启动多个进程，Pytorch 的 imagenet 示例使用的是这种方法；
- 使用这种方法时，`mp.spawn(fn, args=(), nprocs)` 生成 `nprocs` 个进程时，每个进程的入口函数为 `fn(i, *args)`，其中 `i` 是模块自动生成的进程索引 (process index)；

2. `torch.distributed.launch` (此 API 将被弃用 deprecated)

- 可以通过 `torch.distributed.launch` 启动多个进程 (`torch.distributed.launch` is a module that spawns up multiple distributed training processes on each of the training nodes。);

```

1 python -m torch.distributed.launch --nproc_per_node=NUM_GPUS_YOU_HAVE
2 --nnodes=2 --node_rank=0 --master_addr="192.168.1.1"
3 --master_port=1234 YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3
4 and all other arguments of your training script)

```

- 使用 `torch.distributed.launch`，需要用户在代码中获取 `local_rank` 参数，有 2 种方法获取此参数：

- 在代码中解析命令行参数：`--local_rank=LOCAL_PROCESS_RANK`，此参数是由模块提供，但是需要用户在代码中解析：

```

1 import argparse
2 parser = argparse.ArgumentParser()
3 parser.add_argument("--local_rank", type=int)
4 args = parser.parse_args()
5
6 torch.cuda.set_device(args.local_rank) # before your code runs

```

- 通过环境变量 `LOCAL_RANK`，此时，需要在命令行中启动 `torch.distributed.launch` 时，指定参数 `--use-env=True`，然后在代码中，使用 `os.environ['LOCAL_RANK']` 替换方式一的 `args.local_rank`；

3. 使用 `torch.distributed.run` 或者 `torchrun` 简化初始化代码 (elastic launch) :

- `torch.run` 提供了 `torch.distributed.launch` 的一个超集 (superset)，包含如下新功能：

- 容错能力，如果某个 worker 出现故障，会自动重启所有 worker，因此代码中应该包含 `load_checkpoint(path)` 和 `save_checkpoint(path)` 逻辑，方便代码从最新的 checkpoint 恢复；
- Worker 的 `RANK` 和 `WORLD_SIZE` 是自动指定的，可以直接从环境变量中获取；
- 节点数量允许在最小值和最大值之间 (弹性) 变化 (Number of nodes is allowed to change between minimum and maximum sizes (elasticity))

2. 从 `torch.distributed.launch` 迁移到 `torch.run`:

- `torch.run` 支持除了 `--use-env` 之外的所有 `launch` 参数；因此如果在 `launch` 代码中，使用 `os.environ['LOCAL_RANK']`，则只需要去掉 `--use-env` 命令行参数
- 如果在代码中通过解析命令行参数获取 `--local_rank`，则需要将这些代码替换为直接从环境变量中获取；

5. Pytorch DDP 内部设计 (Internal Design)

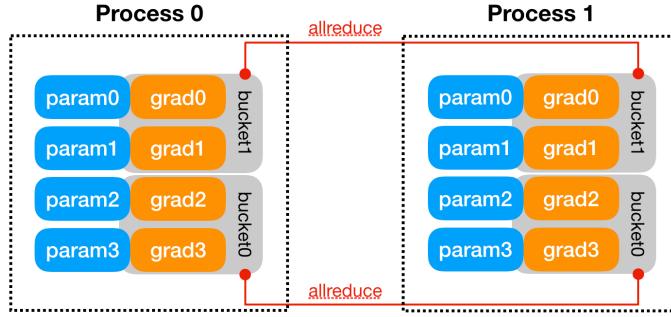
以下部分为 `torch.nn.parallel.DistributedDataParallel` 底层的工作原理：

- 先决条件 (Prerequisite): 在构造 DDP 模型之前，必须初始化进程组 (ProcessGroup)

2. 构建 (Construction) :

- DDP 构造函数引用本地模块 (takes a reference to the local module)，并将模型状态 `state_dict()` 从 master process 广播 (broadcast) 到 Group 中的所有其他进程中，以确保所有的模型副本都包含相同的初始状态；
- 每个 DDP 进程创建一个局部 (local) `Reducer`，用于处理 backward 期间的梯度同步；为了提高通讯效率，`Reducer` 将参数梯度进行分桶 (organizes parameter gradients into buckets)，每次对一个桶进行规约 (reduces one bucket at a time)；桶的大小 (Bucket size) 可以在 DDP 初始化时，通过参数 `bucket_cap_mb` 指定 (默认容量为 25MB)；

3. 参数梯度到桶的映射 (The mapping from parameter gradients to buckets) 是根据桶的容量 (**bucket size limit**) 和参数大小 (**parameter sizes**) 确定的; 模型参数按照 `model.parameters()` 的 (大致) 相反的顺序分配到桶中 (Model parameters are allocated into buckets in (roughly) the reverse order of `Model.parameters()` from the given model.) 使用相反顺序的原因是, **DDP 期望梯度在 backward 期间大约以该顺序准备好** (DDP expects gradients to become ready during the backward pass in approximately that order) ;
4. 如下图所示, `grad2` 和 `grad3` 位于 bucket0 中, 当 bucket0 中的梯度计算完成后, Reducer 就可以启动进程的通讯; 除了分桶以外, Reducer 还在构造过程中注册 (自动求导钩子) `autograd hook`, 每个参数一个 hook; 在 **backward** 过程中, 如果梯度准备就绪, 则触发 `hook`; (These hooks will be triggered during the backward pass when the gradient becomes ready.)



3. 前向过程 (Forward Pass)

1. DDP 获取输入, 并且传递给本地模型 (local model); 如果 `find_unused_parameters=True`, 此时允许在模型的 subgraph 上进行反向传播, DDP 从模型输出遍历 autograd 图, 将所有未使用的参数 (all unused parameters) 标记为已准备好规约 (ready for reduction), 这样可以防止 DDP 在 backward 过程中永远等待不存在的梯度 (prevent DDP from waiting for absent gradients forever during the backward pass.) ; (由于遍历 autograd 图会带来额外的开销, 因此只有在有必要的情况下设置 `find_unused_parameters` 为 `True`)

4. 后向过程 (Backward Pass)

1. `backward()` 函数直接在 loss `Tensor` 上调用, 此过程不受 DDP 控制, 当一个梯度 Ready 之后, 它对应参数的梯度累加器 (grad accumulator) 上的 autograd hook 被触发, 然后 DDP 将此参数梯度标记为 已准备好规约 (ready for reduction) (When one gradient becomes ready, its corresponding DDP hook on that grad accumulator will fire, and DDP will then mark that parameter gradient as ready for reduction.)
2. 当一个桶 (bucket) 中的所有梯度都 Ready 之后, Reducer 就会在该 bucket 上启动异步的 allreduce, 以计算所有进程的平均梯度; (When gradients in one bucket are all ready, the Reducer kicks off an asynchronous allreduce on that bucket to calculate mean of gradients across all processes.)
3. 当所有的 bucket 都 ready 后, Reducer 会阻塞等待所有的 allreduce 完成; 完成此操作后, 平均梯度将写入到所有参数的 `param.grad` 字段 (When all buckets are ready, the Reducer will block waiting for all allreduce operations to finish. When this is done, averaged gradients are written to the `param.grad` field of all parameters.) ; 因此, 经过 `backward()` 过程之后, 不同 DDP 进程中相同对应 (same corresponding parameter) 的参数会相等;

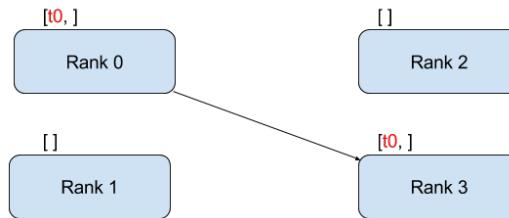
5. 优化步骤 (Optimizer Step)

1. 从优化器角度看, 它优化的是局部模型 (From the optimizer's perspective, it is optimizing a local model.) , 所有 DDP 进程上的模型副本都可以保持同步, 因为使用相同状态开始, 并且每次迭代中具有相同的平均梯度;

6. Pytorch 中的通讯

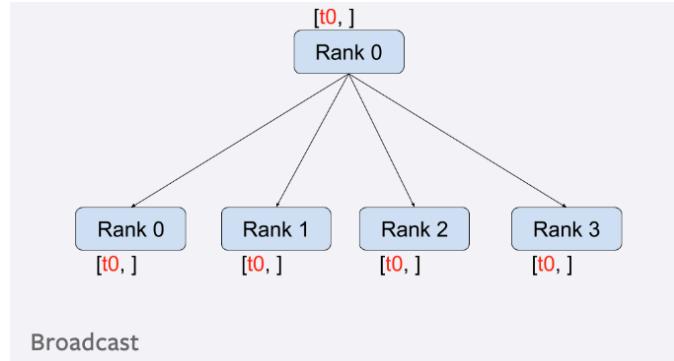
1. 点对点通讯 (Point-to-Point Communication)

1. 将数据从一个进程传输到其他进程称为点对点通讯 (A transfer of data from one process to another is called a point-to-point communication)
2. `torch.distributed` 中包含了 `send` 和 `recv` 函数, 以及他们的非阻塞版本 (**non-blocking**) `isend` 和 `irecv`

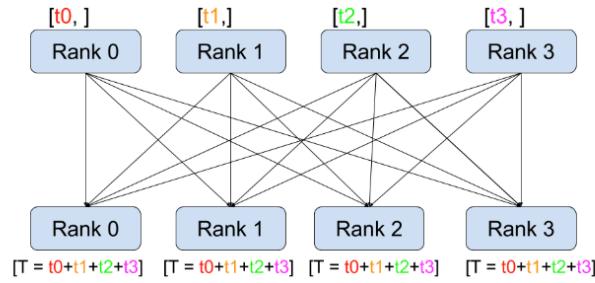


2. 集合通讯 (Collective Communication)

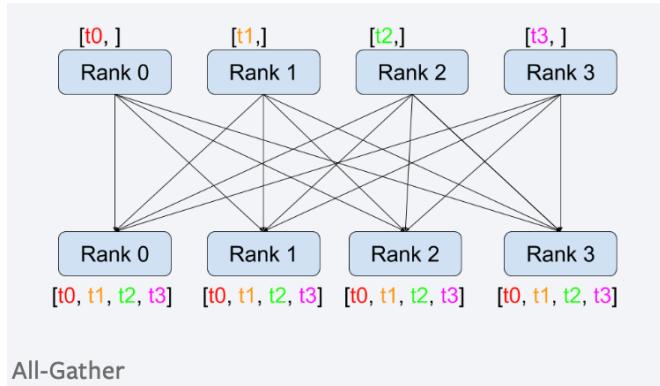
1. 集合通讯允许进程组中多个进程同时通讯，Pytorch 实现了 6 个集合通讯方式，分别为：`scatter`、`gather`、`reduce`、`all_reduce`、`broadcast`、`all_gather`；



2. 如下图所示，是分布式训练中最常用的 `broadcast` 和 `all_reduce`（下面图片中显示的是 `dist.ReduceOp.SUM`，pytorch 实现了 4 个操作：`dist.ReduceOp.SUM`、`dist.ReduceOp.PRODUCT`、`dist.ReduceOp.MAX`、`dist.ReduceOp.MIN`）：



All-Reduce



7. Pytorch DDP VS DP

1. DDP 使用多进程（multiprocessing），而 DP 使用多线程（multithreading），多线程的性能会受到 Python 语言的全局解释器锁（GIL）限制；
2. DDP 支持多机器训练（Supports scaling to multiple machines），而 DP 只支持单机器；
3. 在 DDP 中，模型只需要复制一次；而在 DP 中，每一个 forward 都需要对每个 device 复制模型一次；
4. 在 DDP 模式下，对于一个 batch 的数据，唯一的通信发生在梯度的 all-reduce 阶段；而 DP 需要进行 5 次通信（参考下图）；（The only communication DDP performs per batch is sending gradients, whereas DP does 5 different data exchanges per batch.）
5. 在 DP 模式下，GPU 0 比其他 GPU 执行更多工作，从而导致 GPU 利用率不足（resulting in under-utilization of gpus）；

8. 如下是 DDP 和 DP 的详细过程比较：

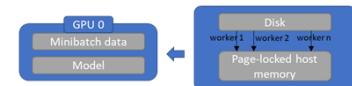
1. DP 详细的流程如下图所示：

Data Parallel

One GPU (0) acts as the master GPU and coordinates data transfer.

Implemented in PyTorch data_parallel module

1. Transfer minibatch data from page-locked memory to GPU 0 (master). Master GPU also holds the model. Other GPUs have a stale copy of the model



2. Scatter minibatch data across GPUs



3. Replicate model across GPUs



4. Run forward pass on each GPU, compute output. Pytorch implementation spins up separate threads to parallelize forward pass



5. Gather output on master GPU, compute loss



6. Scatter loss to GPUs and run backward pass to calculate parameter gradients



7. Reduce gradients on GPU 0



8. Update Model parameters



1. GPU 0 读取 batch 的数据, 然后将 min-batch 发送给其他 GPU;
2. 将 GPU 0 的模型复制 (replicates) 到其他 GPU;
3. 在每块 GPU 上运行 forward 过程, 并且将 output 发送给 GPU 0, 由 GPU 0 计算 loss;
4. 将 loss 从 GPU 0 发送 (scatters) 到所有 GPU, 运行 backward;
5. 将所有 GPU 计算的梯度发送给 GPU 0, GPU 0 计算梯度均值, 然后更新 GPU 0 上的模型参数;

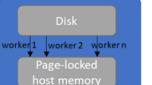
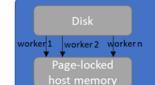
2. 而 DDP 详细流程如下:

Distributed Data Parallel

No master GPUs

Implemented in PyTorch DistributedDataParallel module

1. Load data from disk into page-locked memory on the host. Use multiple worker processes to parallelize data load.

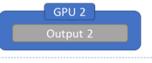
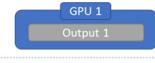


Distributed minibatch sampler ensures that each process loads non-overlapping data

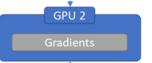
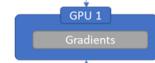
2. Transfer minibatch data from page-locked memory to each GPU concurrently. No data broadcast is needed. Each GPU has an identical copy of the model and no model broadcast is needed either



3. Run forward pass on each GPU, compute output



4. Compute loss, run backward pass to compute gradients. Perform gradient all-reduce in parallel with gradient computation

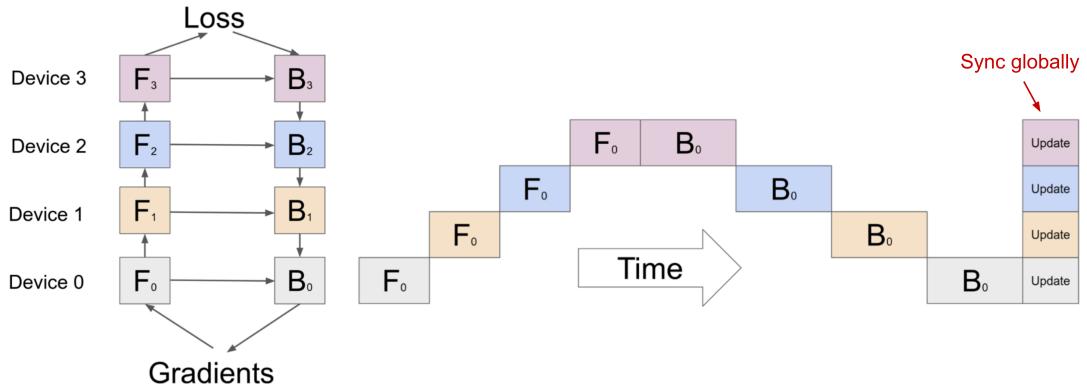


5. Update Model parameters. Because each GPU started with an identical copy of the model and gradients were all-reduced, weights updates on all GPUs are identical. Thus no model sync is required



2. 流水线并行 (Pipeline Parallelism)

1. 背景: 当模型太大无法放在单个设备上时, 需要采用模型并行 (**Model parallelism (MP)**) 的策略, 由于深度网络通常由若干层叠加形成, 因此将网络按层 (layer) 分割成多个部分, 每台设备负责一个部分 (若干个连续的层); 然而, 如果直接将一个 batch 的数据输入由多台机器组成的顺序依赖网络 (naive 实现), 会导致很长的等待时间和计算资源的严重不足 (However, a naive implementation for running every data batch through multiple such workers with sequential dependency leads to big bubbles of waiting time and severe under-utilization of computation resources.) , 如下图所示:



2. 介绍:

- 流水线并行将模型并行和数据并行结合，以减少低效的“bubbles”，模型的层可以跨越多个设备进行分片 (With pipeline parallelism, the layers of a model are sharded across multiple devices)；一个批次的数据被分为更小的微批次 (A batch is split into smaller microbatches)，执行时，不同阶段的 worker 可以对不同的微批次并行处理 (enable each stage worker to process one microbatch simultaneously)；
- 每个微批次都需要处理 2 次 (needs two passes, one forward and one backward.)；不同 worker 之间通信只是在前向过程中传输激活值以及反向传播过程中传输梯度 (Inter-worker communication only transfers activations (forward) and gradients (backward).)
- 有多种不同的方法规划 microbatch 的跨设备前向和后向过程 (scheduling forward and backward microbatches across devices;)，不同的方法在气泡大小、通讯、内存占用 之间进行权衡 (each approach offers different tradeoffs between pipeline bubble size, communication, and memory footprint)；

3. GPipe

- Gpipe 流程下图所示：



Figure 3: GPipe pipeline schedule with forward passes (blue) for all microbatches (represented by numbers) followed by backward passes (green). The gray area represents the pipeline bubble. For simplicity, we assume that the backward pass takes twice as long as the forward pass. The efficiency of the pipeline schedule does not depend on this factor. Each batch in this example consists of 8 microbatches, and the numbers in each blue or green box are unique identifiers given to the corresponding microbatch (in particular, the first batch consists of microbatches 1–8, the second batch consists of microbatches 9–16, and so on). The optimizer is stepped and weight parameters updated at the pipeline flush to ensure strict optimizer semantics, leading to idle devices and a pipeline bubble.

- 上图中，数字 1, 2, ..., 8 表示 8 个微批次 (microbatches)，蓝色 (blue) 方块表示前向过程 (forward pass)，绿色 (green) 方块表示反向过程 (backward pass)，没有数字标识的灰色方块表示气泡 (pipeline bubble)，上图中假设反向过程的耗时是前向的 2 倍 (上图中方块宽度)；上图中，第一批 (the first batch) 包含 1 – 8 个微批次；
- 从上图可以直观看到，GPipe 对一个 batch 中的所有微批次进行前向过程，然后执行所有微批次的后向过程 (GPipe proposes a schedule where the forward passes for all microbatches in a batch are first executed, followed by backward passes for all microbatches)；所有微批次的梯度被聚集，并且在最后同步 (gradients from multiple microbatches are aggregated and applied synchronously at the end)，如上图所示，在所有微批次的 backward 完成后，进行 pipeline flush 更新模型参数；
- 设 batch 中包含的 micro-batch 数量为 m ，流水线的 stage 数 (用于进行流水线并行的设备数) 为 p ；单个 micro-batch 完成 forward 的时间为 t_f ，完成 backward 的时间为 t_b ，因此，在理想情况下，处理完 m 个 micro-batch 的最优 (少) 时间为： $t_{ideal} = m \cdot (t_f + t_b)$ ，而上图 GPipe 中，空闲气泡所占的总时间为： $t_{pb} = (p - 1) \cdot (t_f + t_b)$ (如上图所示，在 forward 阶段，每个 device 都包含 3 个 ($p - 1$ 个) 空闲的 unit 时间，backward 同理)，因此，GPipe 流水线中气泡时间占比为： $\frac{t_{pb}}{t_{ideal}} = \frac{p-1}{m}$
- 为了使得气泡占比时间最小，需要 micro-batch 的数量远大于流水线并行数 ($m \gg p$)，GPipe 的论文表明，在使用激活重计算 (activation checkpoint) 时，当 $m \geq 4 \times p$ 时，气泡开销可以忽略；
- 对于较大的微批次 m ，如果不使用激活重计算，会导致很高的显存占用。因为 GPipe 的同步更新 (synchronous way) 方式需要一个 batch 中的所有 micro-batch 都完成 backward 之后更新参数，这就需要将 iteration 周期内所有 m 个微批次的中间激活值保留在显存中 (而使用激活重计算，只需要保留分片边界处的激活)；

4. PipeDream 系列

- PipeDream:

- PipeDream 规划每个 worker 交替执行 forward 和 backward (PipeDream schedules each worker to alternatively perform the forward and backward passes (1F1B for short))；

2. 在 PipeDream 模式下，设备首先进入启动状态 (**startup state**)，在此状态下执行不同数量的前向传递，当第一个微批次的前向过程完成后，**立刻开始此微批次的反向传播**，对于后面的微批次交替进行前向和反向传播 (In this schedule, the devices first enter a startup state where they perform a **different number of forward passes**. As soon as the output stage completes the forward pass for the first minibatch, it performs the backward pass for the same batch and then starts alternating between forward and backward passes in the following minibatches.)；在启动状态之后，进入稳定状态 (**Steady State**)，每个 worker 执行一次前向传递后执行一次反向传递 (简称 **1F1B**)，之后没有 GPU 处于空闲状态；

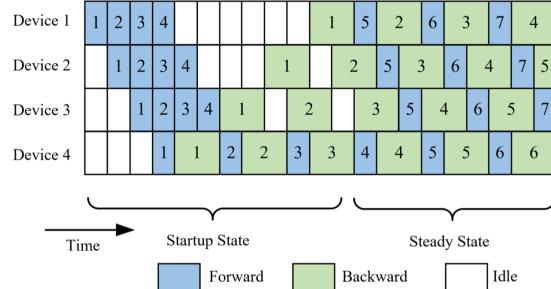


Fig. 4. Illustration of 1F1B microbatch scheduling in PipeDream.
Image based on: [48].

3. 由于 PipeDream 没有跨 worker 的 **end-of-minibatch 的全局梯度同步 (GPipe 包含此步骤)** (PipeDream does not have an **end-of-batch global gradient sync across all the workers**)，一个 Naive 的 1F1B 实现会导致一个 microbatch 的前向和后向使用不同的模型权重，导致学习效率的降低 (an native implementation of 1F1B can easily lead to the forward and backward passes of one microbatch using different versions of model weights, thus lowering the learning efficiency.)， PipeDream 使用如下方案解决此问题：

1. 权重存储 (**Weight stashing**)：每个 worker 保存多个版本的模型，确保同一个 micro-batch 的前向和后向使用的相同的版本的权重；最坏情况下，存储的权重版本和 pipeline 并行数 p 相同，对于大模型来说非常昂贵；
2. 垂直同步 (**Vertical sync (Optional)**)：将模型的权重与激活和梯度一起在不同的**work** 之间传播 (The version of model weights flows between stage workers together with activations and gradients)，注意，这种方式依然是异步的（与 GPipe 的同步方式不同）

2. PipeDream-Flush

1. PipeDream-Flush 在 PipeDream 的基础上，**增加了定期的全局同步方案 (globally synchronized pipeline flush periodically)**（与 GPipe 类似），通过这种方式，通过牺牲一些吞吐来极大减少内存占用 (PipeDream-Flush 只需要维护一个版本的模型权重) (it greatly reduces the memory footprint (i.e. only maintain a single version of model weights) by sacrificing a little throughput)
2. 如下图所示，相比于 GPipe，流水线的气泡数是一样的 (The time spent in the bubble is the same)，但是，活跃的 **input activation** 总数小于或者等于管道深度 p (也就是 pipeline 并行数) (the total number of in-flight “active” input activations is less than or equal to the pipeline depth)，而 GPipe 需要保持的 **input activation** 数量是微批次数量 m ，因此，当 $m \gg p$ 时，PipeDream-Flush 的内存占用比 GPipe 更少 (PipeDream-Flush is much more memory-efficient than GPipe.)
3. 上面 2 的说法也可以表达为：与 Gpipe 的方法相比，流水线气泡时间是相同的，但是处理中的微批次数量最多为流水线阶段数 p (pipeline stages)，因此，这种 schedule 只需要保留 p 个或者更少的中间激活结果。而 Gpipe 模式需要保留 m 个微批次的中间激活值。因此当 $m \gg p$ 时，**PipeDream-Flush 的内存效率比 GPipe 高得多** (The time spent in the bubble is the same for this new schedule, but the number of outstanding forward passes is at most the number of pipeline stages for the PipeDream-Flush schedule. As a result, **this schedule requires activations to be stashed for p or fewer microbatches** (compared to m microbatches for the GPipe schedule). Consequently, when $m \gg p$, PipeDream-Flush is much more memory-efficient than GPipe.)
4. PipeDream-Flush 由于有全局同步方案，因此可以确保 **使用最新权重版本计算的梯度进行权重更新 (weight updates can be performed with gradients computed using the latest weight version)**，也就是 $W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t)})$

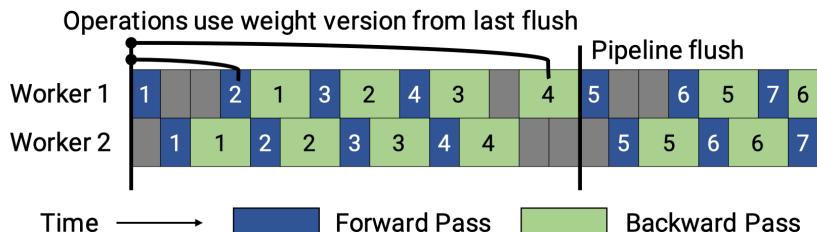
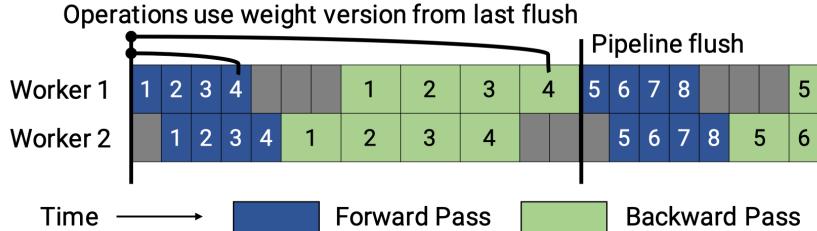


Figure 3. Timelines of GPipe and PipeDream-Flush for 2 stages. Both GPipe and PipeDream-Flush use pipeline flushes; PipeDream-Flush alternates between forward and backward passes in steady state to keeping memory footprint low compared to GPipe by limiting activation stashes to only in-flight microbatches.

3. PipeDream-2BW



Figure 2. Timeline showing PipeDream-2BW's double-buffered weight update (2BW) scheme with time along x -axis. Without loss of generality, backward passes are assumed to take twice as long as forward passes. PipeDream-2BW only stashes two weight versions at every worker, reducing the total memory footprint while no longer requiring expensive pipeline stalls. $W_i^{(v)}$ indicates weights on worker i with version v (contains weight gradient generated from input v). New weight versions are generated in checkered green boxes; $W_4^{(4)}$ is first used for input 9's forward pass.

1. PipeDream-Flush 和 PipeDream-2BW 出自同一篇论文 'Memory-Efficient Pipeline-Parallel DNN Training'
2. 2BW 指的是双缓冲权重 (double-buffered weight)， PipeDream-2BW 没有定期的全局同步方案 (异步并行方式)，但是需要每个 worker 保留 2 个版本的模型权重，这种模式每隔 k 个微批次生成一个新模型， k 应该比流水线深度 p 大 ($k > p$)，新更新的模型版本无法立即完全替换旧版本，因为一些剩余的后向传递仍然依赖旧版本；但是总共只需要保留 2 个版本；
3. PipeDream-2BW 不是在使用新权重之前进行 pipeline flush，而是对新允许进入 pipeline 的输入使用新的权重，同时对已经进行中的输入使用以前的权重 (instead of flushing the pipeline before using newly updated weights, 2BW uses the new weights for inputs newly admitted into the pipeline, while using the previous weight version, called the shadow version, for already in-flight inputs.)； PipeDream-2BW 模式比 GPipe 吞吐更高 (无 pipeline flush)，比 PipeDream 占用更少内存 (因为只有2个版本权重，而不是 PipeDream 中最多 p 个 (p 为流水线深度))；
4. PipeDream-2BW 引入了恒定权重延迟项 1，权重更新为： $W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t-1)})$ ，论文实验表明延迟项不影响模型收敛

5. Interleaved 1F1B Pipeline (Megatron的阶段交错流水线, Schedule with Interleaved Stages)

1. 为了减少气泡的数量，每个设备可以对网络层的多个子集执行计算 (称为模型块) (each device can perform computation for multiple subsets of layers (called a model chunk))，而不是单个连续网络层 (instead of a single contiguous set of layers)；例如，原来 device1 计算网络的 1 ~ 4 层，现在可以使 device1 计算网络的 1 ~ 2 以及 9 ~ 10 层，通过这种方案，流水线中的每个设备都被分配多个流水线阶段，每个流水线阶段的计算量相比之前变少了 (With this scheme, each device in the pipeline is assigned multiple pipeline stages (each pipeline stage has less computation compared to before.))

2. 如下图所示, 为了减少内存占用, Megatron 使用 PipeDream-Flush 的 1F1B 模式, 要求一个 batch 中的 microbatches 数量是流水线并行数 (流水线设备数) 的整数倍 (requires the **number of microbatches in a batch to be an integer multiple of the degree of pipeline parallelism** (number of devices in the pipeline))
3. 相比于 PipeDream-Flush 模型, Megatron 的阶段交错流水线模型在相同的 batch size 时, **发生 pipeline flush 的时间更早** (the pipeline flush for the same batch size happens sooner in the new schedule), 假设每一个 device 包含 v 个阶段 (stage) (也就是每个 device 包含 v 个模型块, model chunk) (下图中下方图中, 每个设备包含 2 个 model chunk), 则每个 stage 的前向和后向时间分别为 $\frac{t_f}{v}$ 和 $\frac{t_b}{v}$, 流水线中的气泡时间缩减为 $p_{pb}^{int.} = \frac{(p-1) \cdot (t_f + t_b)}{v}$, 此时, 流水线中气泡时间占比为 $\frac{t_{pb}^{int.}}{t_{id}} = \frac{1}{v} \cdot \frac{p-1}{m}$, 是 GPipe 的 $1/v$;
4. Megatron 的阶段交错流水线模式 (Schedule with Interleaved Stages) 通过对每台设备使用 v 的 model chunk, 将流水线中的气泡时间缩减为 GPipe 的 $1/v$, 然而, 这种模式将通讯量也增加了 v 倍;

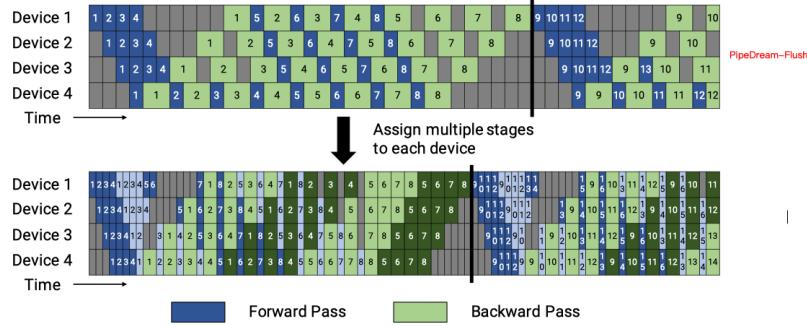


Figure 4: Default and interleaved 1F1B pipeline schedules. The top figure shows the default **non-interleaved 1F1B schedule**. The bottom figure shows the **interleaved 1F1B schedule**, where each device is assigned multiple chunks (in this case, 2). Dark colors show the first chunk and light colors show the second chunk. The size of the pipeline bubble is smaller (the pipeline flush happens sooner in the interleaved timeline).

3. Tensor 并行

- 张量 (Tensor) 并行是指将一个 tensor 分为多个 chunk (split up into multiple chunks), 每个 chunk 都分配给指定的 GPU, 在处理过程中, 每个 chunk 在不同的 GPU 上独立并行处理, 结果在计算步骤结束时同步;
- 一个 Transformer 包含一个 self-attention block 以及 followed 的 2 层 MLP:

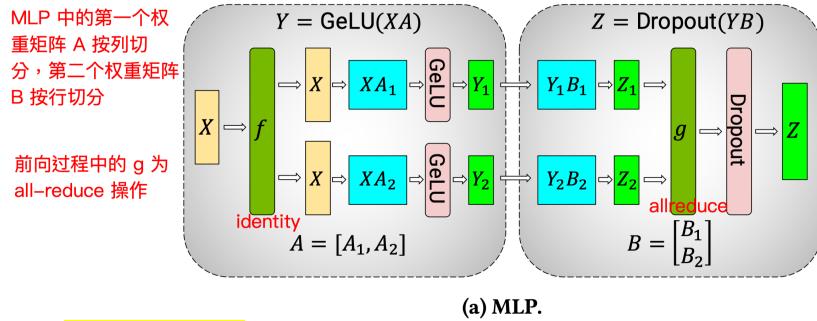
1. MLP 层的 Tensor 并行:

- MLP 层包含 2 个 GEMMs (通用矩阵乘法, General matrix multiply), 以及一个 GeLU 非线性:

$$\begin{aligned}
 Y &= \text{GeLU}(XA) \\
 \text{Split } A &= [A_1, A_2] \\
 [Y_1, Y_2] &= [\text{GeLU}(XA_1), \text{GeLU}(XA_2)] \\
 Z &= \text{Dropout}(YB) \\
 \text{Split } B &= \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \\
 Z &= \text{Dropout}([Y_1, Y_2] \begin{bmatrix} B_1 \\ B_2 \end{bmatrix})
 \end{aligned} \tag{19}$$

上式中, X 为输入, A 和 B 为第 1 个和第 2 个权重矩阵;

2. 如下图所示:



1. **矩阵 A 应该按列切分**, 这样使得非线性运算 GeLU 独立的对每一个中间结果作用, 避免了同步操作; (This partitioning allows the GeLU non-linearity to be independently applied to the output of each partitioned GEMM);

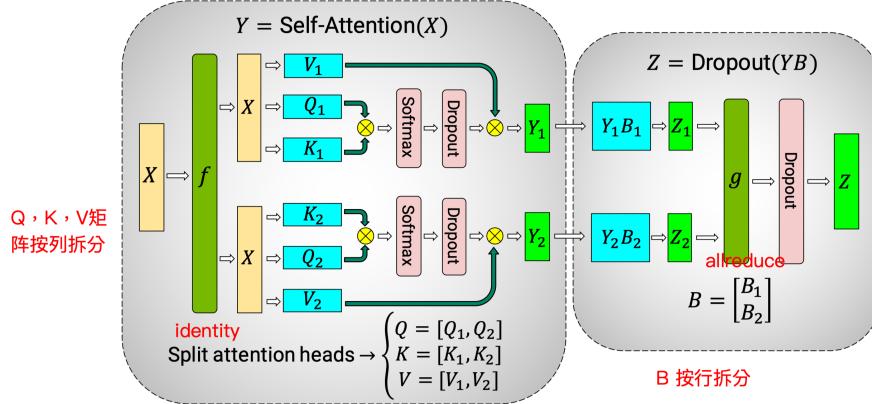
2. **矩阵 B 应该按行切分**, 这样可以直接对 GeLU 层的输出进行运算,

$Z = \text{Dropout} \left([Y_1, Y_2] \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \right)$, 这里需要注意的是, 在进行 Dropout 之前需要进行 allreduce 操作, 因为 $Y_1 B_1$ 和 $Y_2 B_2$ 在不同的设备上;

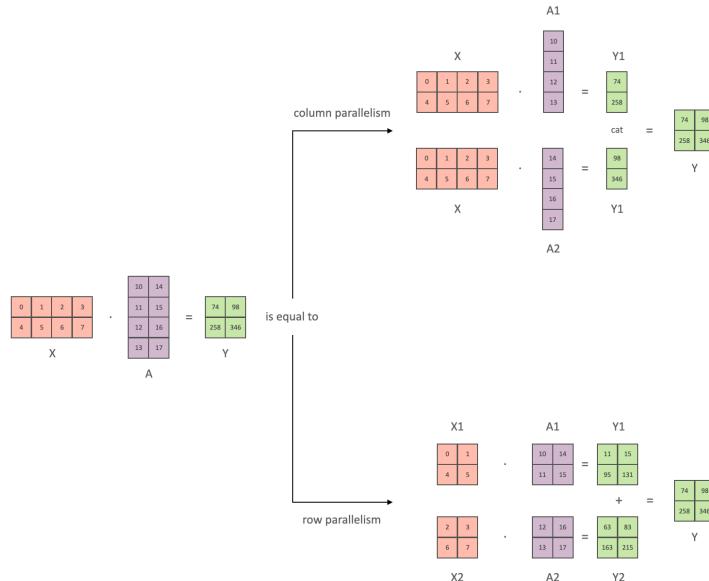
2. Self-Attention 层 Tensor 并行:

1. Self-Attention 中, Q 、 K 、 V 都是按列拆分, 而后续输出的线性层 (output linear layer) 是按行拆分;

2. 如下图所示:



3. 将 MLP 和 Self-Attention 中的 GEMM 操作切分到不同的 GPUs 上运行, 只需要在前向过程中引入 2 次 AllReduce 操作 (上图中 MLP 中的 g 以及 Self-Attention 中的 g) , 反向过程中引入 2 次 AllReduce 操作 (上面 2 张图中的 f 操作) ;
4. 上面 2 张图中的, f 和 g 是成对的 (conjugate) : f 在前向过程中是恒等操作, 在反向过程中是 all-reduce 操作 (与 g 刚好相反) ;
5. 沿着不同维度的矩阵划分结果如下:



4. 零冗余优化器 (Zero Redundancy Optimizer (ZeRO))

1. 大模型训练过程中的显存消耗可以分为 2 部分:

1. 模型状态 (model states) : 优化器状态 (optimizer states) 、模型梯度 (gradients) 、参数 (parameters)
2. 残留状态 (residual states) : 激活值 (activation) , 临时缓存 (temporary buffers) 、无法使用的碎片内存 (unusable fragmented memory)

2. 优化模型状态内存 (Optimizing Model State Memory)

1. 已有方法缺点:

1. 现有的 DP (数据并行) 模式会将完整的模型状态复制给所有的并行进程, 导致冗余内存消耗 (DP replicates the entire model states across all data parallel process resulting in redundant memory consumption;) ; **DP 具有良好的计算/通信效率, 但是内存效率较低** (DP has good compute/communication efficiency but poor memory efficiency)
2. 现有的 MP (模型并行) 模式对模型状态进行分片, 获得了高内存效率, 但是会导致过于细粒度的计算和昂贵的通信, 从而降低扩展效率 (MP partitions these states to obtain high memory efficiency, but often result in too fine-grained computation and expensive communication that is less scaling efficient) ; **MP 的计算/通信效率较差** (MP can have poor compute/communication efficiency) ;

3. 此外，现有方法都静态地维护在整个训练过程中所需要的所有模型状态，即使在训练期间并非始终需要这些模型状态；（all of these approaches maintain all the model states required over the entire training process statically, even though not all model states are required all the time during the training.）

2. Zero-DP:

1. Zero-DP 通过划分模型状态而不是复制模型状态来消除数据并行中的内存状态冗余（ZeRO-DP removes the memory state redundancies across data-parallel processes by partitioning the model states instead of replicating them），并在训练期间使用动态通信调度保留 DP 的计算粒度和通信量来保持计算/通信效率（it retains the compute/communication efficiency by retaining the computational granularity and communication volume of DP using a dynamic communication schedule during training）

2. 背景：混合精度训练（Mixed-Precision Training）（以下内容来自 Zero 论文）：

1. 以 Adam 优化器为例，对于包含 Ψ 个参数的模型，其显存占用如下：

分类	大小 (字节 byte)
Parameters (fp32)	4Ψ
Momentum (fp32)	4Ψ
Variance (fp32)	4Ψ
Parameters (fp16)	2Ψ
Gradients (fp16)	2Ψ
合计	$2\Psi + 2\Psi + 12\Psi$

2. Zero 论文中，将上表中前 3 个 fp32 类别的变量 Parameters、Momentum、Variance 合称为 optimizer states；使用变量 K 表示 optimizer states 占用的显存倍数，则 Adam 中 $K = 12$ ；由于 **1GB 内存可以包含 1Billion 字节** (**1Billion fp32 参数需要 4GB 存储**)，因此对于包含 **1.5Billion** 参数的 GPT2 模型，按照上表，其总的显存占用为 $16 \times 1.5 = 24Billion$ 字节，也就是需要 **24GB 显存**；而使用 fp16 格式存储的模型参数只需要 $1.5 \times 2 = 3Billion$ 字节，也就是 **3GB 显存**；

3. 当使用混合精度训练以及常规的 DDP 时，**通信量位 2Ψ** ，**通讯操作为 AllReduce**；
4. 注意：在 Zero 论文关于混合精度的分析中，将 Gradient 视为 fp16 的格式，占用 2Ψ 字节，**这是主要是为了分析通信量**，实际上 fp16 的 Gradients 在 unscale 之前需要转化为 fp32 的形式，然后使用此 fp32 的 unscale 梯度更新 fp32 的 master weight；

3. Zero-DP 包含 3 个主要优化步骤，分别对应于优化器状态、梯度和参数的划分（partitioning of optimizer states, gradients, and parameters）：

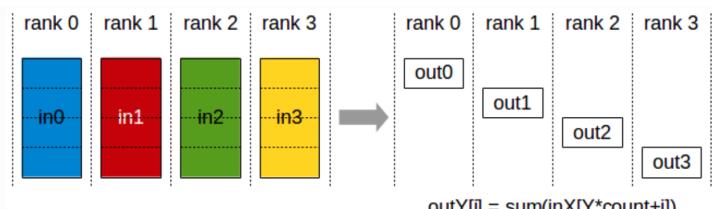
1. 优化器状态划分 (Optimizer State Partitioning) (P_{os})

1. 注意，这里的 Optimizer State 指的是上表中的 **fp32 格式的 Parameters、Momentum、Variance** 三部分；
2. 对于并行度为 N_d 的数据并行 (DP)，将 optimizer states 划分为 N_d 个相同分区，使得第 i 个数据并行进程 (data parallel process) 只更新第 i 个分区的优化器状态。因此，每个进程只需要存储和更新总优化器状态的 $\frac{1}{N_d}$ ，然后只更新 $\frac{1}{N_d}$ 的参数。在每个训练 step 结束之后，使用 all-gather 得到完整更新后的模型参数 (fully updated parameters)
3. 如下图所示，使用 P_{os} 之后，显存消耗从 $4\Psi + K\Psi$ 降低为 $4\Psi + \frac{K\Psi}{N_d}$ ；对于上表中的 Adam 优化器，当 DP (数据) 并行数 N_d 很大时，显存从 $4\Psi + 12\Psi = 16\Psi$ 降低为 $4\Psi + \frac{12\Psi}{N_d} \approx 4\Psi$ ，显存消耗为之前的 **1/4 (4x memory reduction)**

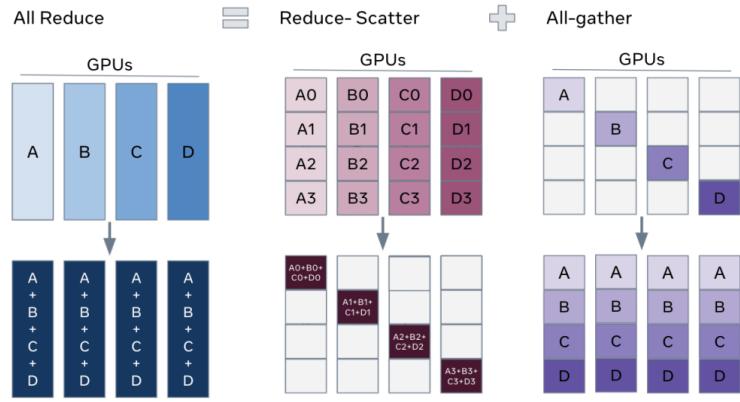
2. 在 (1) 的基础上增加 梯度划分 (Gradient Partitioning) (P_{os+g})

1. 由于步骤 (1) 中，每个数据并行进程只更新其相应的参数分区，因此每个进程也只需要对应分区参数的梯度信息，将 (fp16 的) 梯度占用的空间从 2Ψ 降低为 $\frac{2\Psi}{N_d}$ ；
2. 结合 (1) 和 (2) 的优化，显存从 $2\Psi + 2\Psi + 12\Psi = 16\Psi$ 降低为 $2\Psi + \frac{2\Psi}{N_d} + \frac{12\Psi}{N_d} = 2\Psi + \frac{14\Psi}{N_d} \approx 2\Psi$ ，显存消耗为之前的 **1/8 (8x memory reduction)**

3. 此处，梯度 Partition 的实现是通过 Reduce-Scatter 实现的：



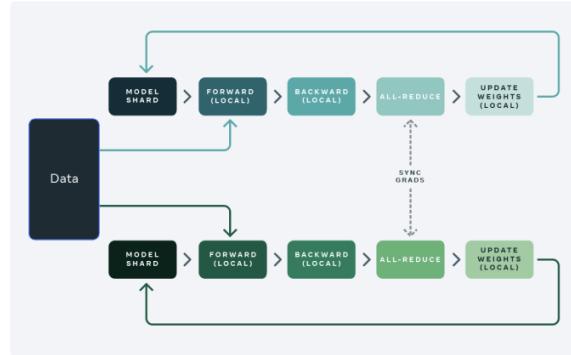
Reduce-Scatter operation: input values are reduced across ranks, with each rank receiving a subpart of the result.



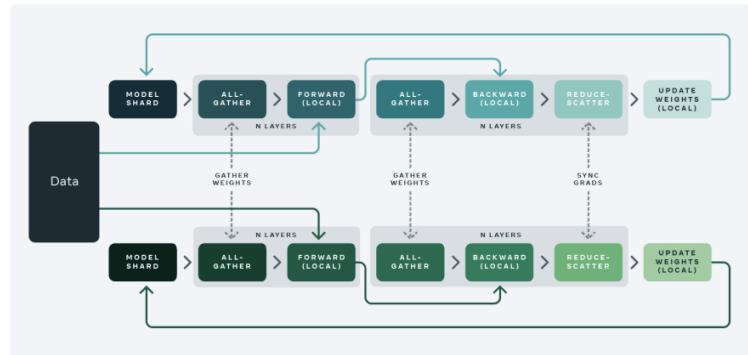
3. 在 (1) (2) 的基础上增加 **参数划分** (Parameter Partitioning) (P_{os+g+p})

- 这里的参数划分指的是 fp16 的参数, 因为 fp32 的 master 参数在 (1) 中已经划分过了;
- 每个进程只存储其分区对应的参数 (each process only stores the parameters corresponding to its partition), 当 forward 和 backward 需要其他分区的参数时, 使用 AllGather 从其他进程获取参数。这种方式会带来 1.5 倍的通信开销, 但是能够实现与并行数 N_d 成线性比例的显存缩减; (we show that this approach only increases the total communication volume of a baseline DP system to $1.5 \times$, while enabling memory reduction proportional to N_d)
- 结合 (1) (2) (3) 的优化, 显存从 16 降低到 $\frac{16\Psi}{N_d}$; (Memory reduction is linear with DP degree N_d)

Standard data parallel training



Fully sharded data parallel training



- 以上 3 个优化步骤如下所示:

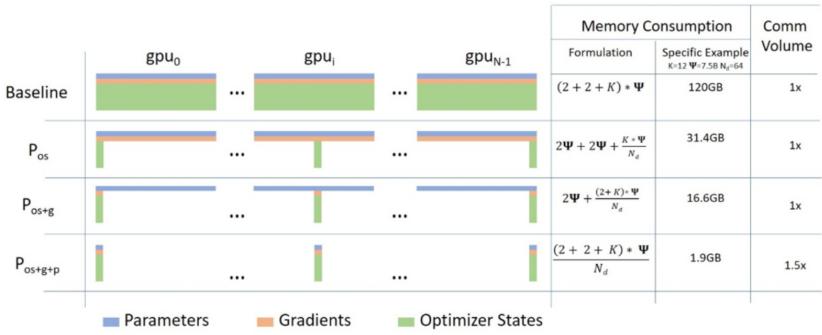


Figure 1: Memory savings and communication volume for the three stages of ZeRO compared with standard data parallel baseline. In the memory consumption formula, Ψ refers to the number of parameters in a model and K is the optimizer specific constant term. As a specific example, we show the memory consumption for a 7.5B parameter model using Adam optimizer where $K=12$ on 64 GPUs. We also show the communication volume of ZeRO relative to the baseline.

5. Zero-DP 通信分析:

1. 数据并行通信量 (Data Parallel Communication Volume) :

- 对于通常的数据并行 (DP)，需要在反向传播时计算所有进程的平均梯度信息，这是通过 All-Reduce 集合通信实现的 (Collective Communication)。下面的分析针对每个进程发送和接收的总通信量 (total communication volume send to and from each data parallel process)；
- SOTA 的 All-reduce 实现是通过 reduce-scatter 和 all-gather 这2个步骤；对于参数量为 Ψ 的模型，这2个 step 都需要 Ψ 的通信，因此 standard DP 每个训练 step 需要 2Ψ 的通信；

2. Zero Stage 1 通信量:

- 当使用 Zero Stage1 时，通信操作为 All Reduce + All Gather，通信量为 3Ψ ，其中 All Gather 通信量为 Ψ ；

3. Zero Stage 2 通信量 (Communication Volume with P_{os+g})

- 通过梯度分区，每个进程只存储用于更新其相应分区参数所需的部分梯度 (each process only stores the portion of the gradients)，因此，不需要使用 All-Reduce 操作，而只需要 reduce-scatter 操作即可。从而产生 Ψ 的通信；
- 当每个进程更新自己分区的模型参数后，需要使用 All-Gather 操作从其他进程获取其他分区更新后的模型参数，也需要 Ψ 的通信。
- 因此 P_{os+g} 的通信量与 Standard DP 的通信量相同 (exactly the same as the baseline DP)。但是显存消耗只有 $1/8$ ；

4. Zero Stage 3 通信量 (Communication Volume with P_{os+g+p})

- 参数分区后，每个数据并行进程只保留它负责更新的参数。因此，在 forward 过程中，需要通过 All-Gather 获取完整的模型参数，forward 完成后，立刻丢弃其他参数。这个过程需要 Ψ 的通信；在 backward 过程中，同样需要 All-Gather 获取完整模型参数，backward 计算完成后，立刻丢弃其他参数，通信量为 Ψ ；backward 计算得到当前分区数据的梯度后，需要进行一次 reduce-scatter 从其他进程获取当前进程所维护的参数梯度，这个步骤通信量为 Ψ ；
- 因此， P_{os+g+p} 的总的通信为 3Ψ ，是 Standard DP 的 1.5 倍；

5. Zero-R

- Zero-DP 对模型状态 (Model State, 包括 optimizer states、gradients、parameters) 进行了优化，Zero-R 是对残留状态 (Residual State, 包括 activations、temporary buffers、unusable memory fragments) 进行优化；

2. Zero-R 分别采用如下方法：

1. 激活值划分 (Partitioned Activation Checkpointing) :

- Zero-R 对 activations 进行切分，具体来说：一旦某一层的 forward 过程计算完成，激活值就会在所有的模型并行进程间切分 (input activations are partitioned across all the model parallel process)，当需要进行 backward 时，使用 all-gather 操作得到完整的 activation. (ZeRO removes the memory redundancies in MP by partitioning the activations checkpoints across GPUs, and uses allgather to reconstruct them on demand.)，这种方式记为 P_a ；
- 也可以将激活点划分 (activation partition) 与 CPU offloaded 结合，记为 P_{a+cpu} ；
- 激活值分区节省的显存与模型并行度成正比 (With partitioned activation checkpointing, ZeRO reduces the activation footprint by a factor proportional to the MP degree)

2. 固定大小的缓冲区 (Constant Size Buffers)

- Zero-R 定义了临时缓冲器的大小，以实现内存和计算效率的平衡；
- 模型训练过程中，对于某些操作，较大的 input size 可能会带来更高的带宽吞吐 (例如 all-reduce 操作)，因此，NVIDIA Apex 和 Megatron 库在进行这些操作之前，会使用 buffer 将一些参数融合 (以提升带宽使用)，然而，这些 buffer 的大小随着模型参数量增大而增大，会带来负面影响；因此 Zero-R 使用固定大小 Buffer Size，与模型参数量无关；

3. 内存碎片整理 (Memory Defragmentation)

1. 由于不同的 tensor 生命周期存在差异，所以训练过程中会遇到内存碎片 (fragmented memory)。即使有足够的可用内存，由于碎片导致缺乏连续内存也会导致内存分配失败 (Lack of contiguous memory due to fragmentation can cause memory allocation failure, even when enough free memory is available.) , Zero-R 根据 tensor 的不同生命周期主动管理内存，防止内存碎片；
2. 模型训练过程中的内存碎片是由于 activation checkpointing 和梯度计算导致的。1. 当使用 activation checkpointing 时，只有部分选择的激活值被存储，大部分激活值被丢弃（在反向传播期间重新计算），这会创建短期内存和长期内存的交错 (This creates an interleaving of short lived memory (discarded activations) and long lived memory (checkpointed activation))，从而导致内存碎片；2. 类似的，在反向传播期间，参数梯度保留的时间很长 (the **parameter gradients** are long lived)，而激活梯度 (**activation gradients**) 和其他计算梯度所需要的缓冲器寿命很短 (any other buffers required to compute the parameter gradients are **short lived**)，短期内存和长期内存的交错再次导致记忆碎片 (interleaving of short term and long term memory causes memory fragmentation.)
3. 当有足够的空闲内存时，有限的内存碎片通常不是问题。但是在有限内存下的大模型训练过程中，内存碎片会导致 2 个问题：1. 即使有足够的内存，也会由于缺乏连续内存 (contiguous memory) 而导致 OOM 问题；2. 由于内存分配器花费大量时间来搜索连续的内存块导致效率低下；
4. Zero 通过为 activation checkpoints 和 gradient 预先分配连续内存块，在它们生成时，将它们复制到预先分配的内存中，即时进行内存碎片整理 (ZeRO does memory defragmentation on-the-fly by pre-allocating contiguous memory chunks for activation checkpoints and gradients, and copying them over to the pre-allocated memory as they are produced.)；内存碎片整理使得 Zero 能够训练具有更大 batchsize 的更大的模型，而且提高了内存使用效率；

5. Zero-Offload

1. 之前所有的分布式并行训练技术，包括 pipeline 并行、tensor 并行、Zero 等都需要足够的 GPU 数量，以便聚合之后的 GPU 显存可以容纳训练所需要的 model states；
2. 论文提出的 Zero-Offload 技术是一种异构深度学习系统 (a heterogeneous DL training technology)，利用 CPU 内存和 CPU 计算进行 Offload (ZeRO-Offload exploits both CPU memory and compute for offloading)；
3. 为了保证效率，必须满足 3 个条件：
 1. CPU 的计算量应该比 GPU 少几个数量级 (it requires orders-of-magnitude fewer computation on CPU compared to GPU)，防止 CPU 计算成为瓶颈；
 2. 应该最大限度减少 CPU 和 GPU 之间的通信量，防止通信成为瓶颈；
 3. 应该最大限度节省 GPU 显存，同时实现最小的通信量；
4. 为了满足以上 3 个条件，必须将 **gradient16**、**optimizer states** 和 **optimizer compute** 卸载 (offload) 到 CPU 上，同时将 **parameters16** 参数、**forward** 以及 **backward** 计算保留在 GPU 上；

5. 如下图所示：

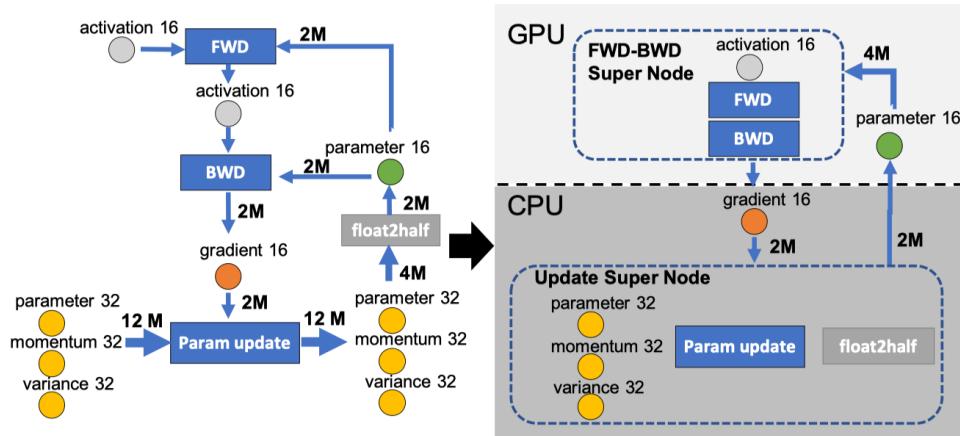
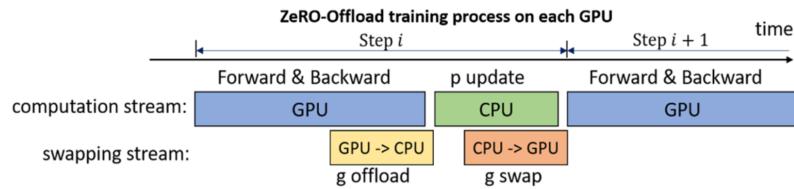


Figure 2: The dataflow of fully connected neural networks with M parameters. We use activation checkpoint to reduce activation memory to avoid activation migration between CPU and GPU.

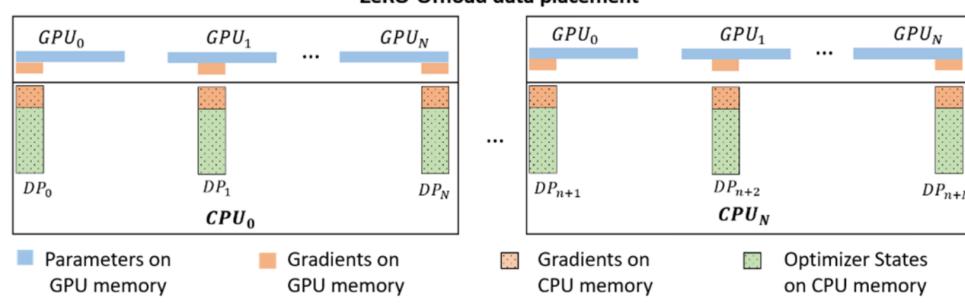
1. 在上图中：

1. 圆形节点表示 model states，包括 parameter16、gradient16、parameter32、momentum32、variance32；
2. 矩形方框表示计算操作 (compute)，包括前向计算 (forward)、后向计算 (backward) 和参数更新 (param update)；
3. 图中的箭头表示数据流 data-flow，边上的权重表示一个 training iteration 流经它的总数据量；

2. 上图展示的是混合精度训练，其中前向的 FWD 过程需要用到当前层的 FP16 参数以及上一层的 FP16 激活；BWD 过程也需要用到激活值和参数梯度；
3. 一个 CPU-GPU 之间的 Offload 策略就是将上图切分为 2 个部分；2 个部分之间的边的权重总和为 CPU 和 GPU 的通信数据量；
4. 上图中共有 4 个不同的计算操作：FWD、BWD、Param Update、float2half，其中，前 2 个操作的计算复杂度为 $O(MB)$ ，其中， M 是模型参数量， B 为 batch size；而后面 2 个操作的复杂度为 $O(M)$ ，与 batch size 无关；为了防止 CPU 计算成为瓶颈，FWD 和 BWD 必须在 GPU 上完成，后 2 个操作可以在 CPU 上完成；为了简化数据图，上面右图将 FWD 和 BWD 这 2 个计算节点融合为一个 FWD-BWD Super Node；上图需要注意的地方是，由于 FWD 和 BWD 都需要 FP16 的模型参数，因此融合之后的节点需要的通信为 $4M$ ；
5. 同时，为了达到最少的通信量，FP32 模型状态的生产者和消费者必须放在一起，也就是 FP32 的模型状态（包括 moment 32、variance 32、parameter 32）必须与 Param Update 和 float2half 放在一起；上图右图将这 3 个部分融合为 Update Super Node；
6. 如上图所示，融合之后的数据图只包含 4 个节点：FWD-BWD Super Node、parameter 16 data node、gradient 16 data node、Update Super Node；为了最大化通信，需要将 parameter 16 data node 放在 GPU 上（将 FP16 模型参数放在 GPU 上，只有 $2M$ 的通信，而放在 CPU 上，会带来 $4M$ 通信）；而为了最大化节省 GPU 显存，gradient 16 data node 需要放在 CPU 上；
6. Zero-Offload 在单 GPU 上的计算流程如下：



1. 训练过程中，由于 FP16 的模型参数在 GPU 上，因此不需要与 GPU 通信，可以直接完成 Forward 过程；在进行 Backward 计算时，可以在计算新的梯度同时，将已经完成的梯度 bucket 传输给 CPU，这样可以将很大部分的通信与 GPU 计算重叠；当 CPU 获取完整梯度之后，在 CPU 上进行参数以及优化器状态的更新，然后将更新之后的 FP32 参数转化为 FP16，然后复制给 GPU 上的 FP16 模型参数；
7. Zero-Offload 在多 GPU 下的扩展：



1. 在多卡场景下，Zero-Offload 利用了 Zero-2，Zero-2 是将模型状态和梯度进行分片，每张 GPU 保存 $\frac{1}{N}$ ，而 Zero-Offload 将每张卡负责的 $\frac{1}{N}$ 的模型状态和梯度都 offload 到 CPU 内存中（and each GPU offloads the partition it owns to the CPU memory and keeps it there for the entire training.），在 CPU 上进行计算；因为传输的是 FP16 的 gradient 和 fp16 的 parameters，因此总的传输量是固定的，与 GPU 的卡数 N 无关；由于可以利用多个 CPU 进行并行计算，因此总的 CPU 计算时间随着卡数增多反而减少了（As a result the total CPU update time decreases with increased data parallelism, since the CPU compute resources increase linearly with the increase in the number of compute nodes.）
2. 注意，在多卡场景下，利用 CPU 多核并行计算，每张 GPU 对应至少一个 CPU 进程，由这个进程负责局部参数的更新；
8. 优化 CPU 的计算：

1. 使用如下 2 个技术加快参数更新的 CPU 执行时间：

1. 为 CPU 优化的 Adam 优化器实现：

1. 在 DeepSpeed 框架下实现了一个 fast CPU adam 优化器，由于充分利用了 CPU 架构上的硬件并行性，因此比 SOTA 的 pytorch 实现更高效；

2. 延迟一步的参数更新（One-Step Delayed Parameter Update）：

1. 当 batch size 很小时，GPU 上计算会很快，导致 CPU 计算成为瓶颈，对于这种有限的情况，论文开发了延迟一步的参数更新策略（one-step delayed parameter update），通过延迟 one-step 的参数更新来重叠 CPU 和 GPU 的计算，从而隐藏 CPU 计算开销；

2. 其流程如下图所示：

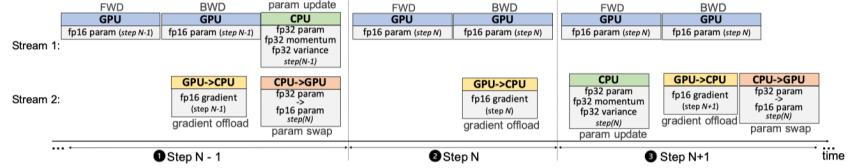


Figure 6: Delayed parameter update during the training process.

1. 在前 $N - 1$ 步，不进行延迟，避免早期训练不稳定（早期梯度变化快速）；
2. 在第 N 步，从 GPU 获取梯度后，跳过 CPU 优化步骤，不更新参数；
3. 在第 $N + 1$ 步，使用第 N 步计算的梯度更新模型参数，同时，使用第 $N - 1$ 更新得到的模型参数计算 forward 和 backward 过程，计算得到的梯度作为 $N + 1$ 步的梯度；从这一步开始，模型在第 $i + 1$ 步将会使用在 $i - 1$ 计算得到的梯度进行更新，以此将 CPU 和 GPU 计算重叠 (From this step onwards, the model at $(i + 1)^{th}$ step will be trained using the parameters updated with gradients from $(i - 1)^{th}$ step instead of parameters updated at i^{th} step)
4. 使用这种方法，用来更新参数的梯度并不是根据当前模型状态计算得到的，论文的实验结果表明暂未发现对收敛和效果产生影响。

6. Zero-Infinity

1. 现代 GPU 集群总的 CPU 内存是总 GPU 显存的 2-3 倍，而 NVMe 总存储则高出 50 倍。同时，现有的 NVMe 技术允许每个 DGX-2 节点实现超过 25GB/s 的读写速度，与 PCIe4.0 相当。因此，充分利用 CPU 内存和 NVMe 存储来 offload 由 Zero-3 分割的 model state 可以使得 Zero-Infinity 以前所未有的规模训练模型；

7. 总结及注意点：

1. 可扩展性策略选择

1. 单节点-多GPU情形 (Single Node / Multi-GPU)

1. 如果模型可以容纳 (fit) 在单个 GPU 上时：使用 DDP 或者 Zero-DP；

2. 如果模型无法 fit 在单个 GPU 上：

1. Pipeline 并行
2. Zero-DP
3. Tensor 并行

当节点的 GPU 之间由 NVLINK 或者 NVSwitch 链接时 (intra-node)，上面 3 种方法速度类似。当没有 GPUs 之间的高速链接时，Pipeline 并行的速度会比 Zero 和 Tensor 并行更快；同时 Tensor 并行的并行度也会有影响；Tensor 并行几乎总是在单节点内部使用 (TP is almost always used within a single node)，因此 TP size \leq gpus per node.

3. 如果模型中最大的层 (Largest Layer) 无法 fit 在单个 GPU 上：

1. 使用 Zero 的 MCT (Memory-Centric Tiling，内存为中心的切片) 功能，能够自动将任意大 layer 进行 split，并且按照顺序执行。MCT 能减少 GPU 上 alive 的参数数量，单不影响 activation memory；

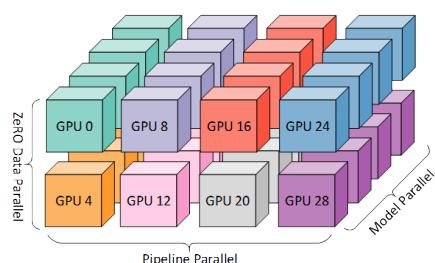
2. 如果不使用 Zero 的 MCT 的话，必须使用 Tensor 并行对 layer 的运算进行切分；

2. 多节点-多GPU情形 (Multi-Node / Multi-GPU)

1. 当具有高速的节点间 (inter-node) 连接时 (例如 IB 网络)：

1. Zero-DP，因为几乎不需要对模型修改；

2. Pipeline 并行 + Tensor 并行 + Zero-DP (称为 3D 并行)：



注意点：

1. 当 Zeor-DP 与 Pipeline 并行 (以及可选的 Tensor 并行) 时，通常只使用 Zero Stage 1，只对模型状态进行分片 (包括 fp32 格式的 Parameters、Momentum、Variance)。
2. 当使用 Zero Stage 2 与 Pipeline 结合使用时，每个 micro-batch 需要一个额外的 reduce-scatter 通讯来聚合梯度，然后才能对梯度分片，这会增加通信开销；(There would need to be an additional reduce-scatter collective for every micro-batch to aggregate the gradients before sharding, which adds a potentially significant communication overhead.)

3. 在使用 Pipeline 并行时，每个 GPU 上保存的模型层数已经比正常情况少，因此每个 GPU 上的梯度大小已经是 $\frac{1}{PP}$ ，因此再使用 gradient sharding 带来的显存节省不明显；
4. 处于同样的原因，Zero Stage3 也不是一个好的选择，需要更多的节点间通信；