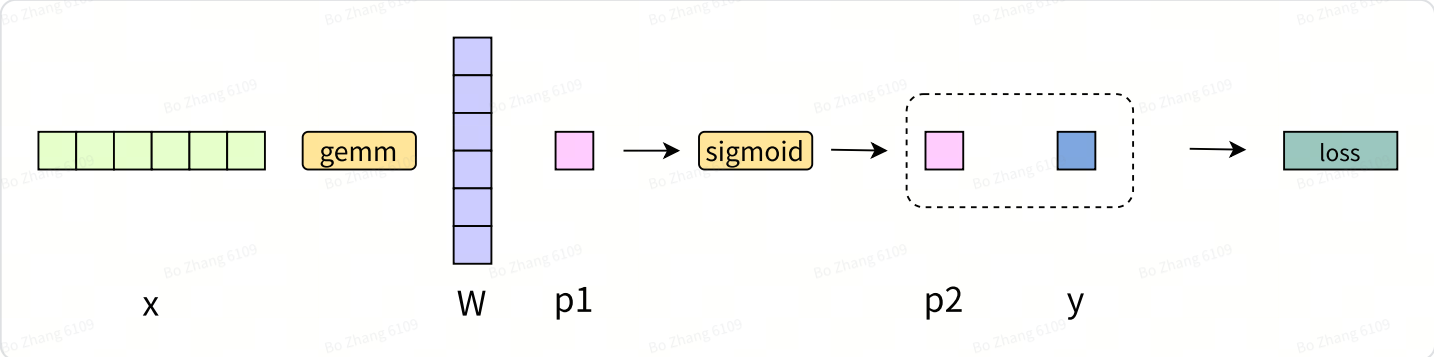
机器学习工程科普

这个科普希望达到的目的：

- 给定一个神经网络，会算他的理论FLOPS
- 给定一个神经网络，会算他训练和推理过程中的显存占用
- 知道什么是计算瓶颈，什么是访存瓶颈
- 了解基本的分布式计算优化技巧

最简单的MLP网络

单层MLP



$$p_1 = x \cdot W$$

$$p_2 = sigmoid(p_1)$$

$$loss = f(y, p_2) = (y - p_2)^2$$

然后，我们的工作，就是最小化这个loss。最简单的方法，是用梯度下降法。这里先不展开。

x：输入的特征矩阵（1 x 6）

W：单层MLP的权重矩阵（6 x 1）

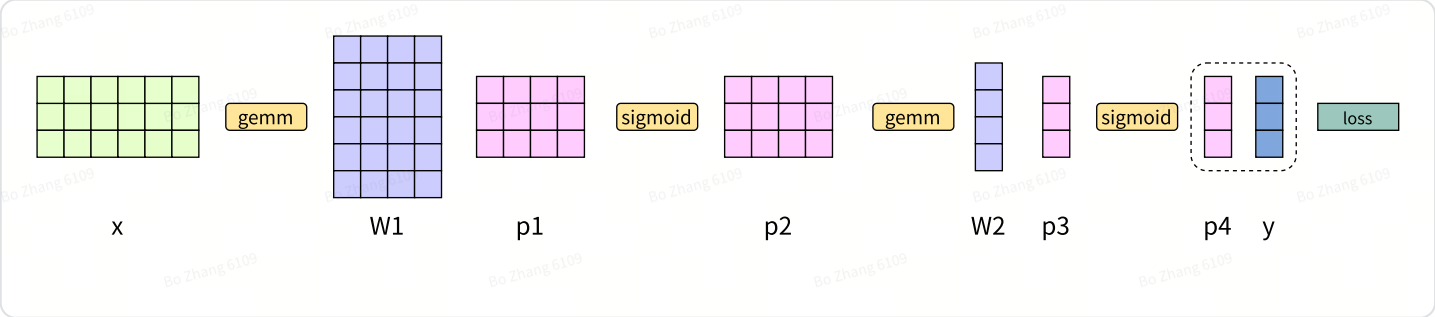
p1：gemm(x, W) 的输出，是个 1 x 1的矩阵，但其实就是一个数字

p2: sigmoid(p1)的输出，把p1做了非线性激活之后的输出，还是一个数字

y：标签，label，也是一个数字

loss：描述p2和y的差距，理论上这个差距越小，就表示神经网络的预估越准

两层MLP



x：输入特征 [3, 6]，这里的3 一般是batch size，就是一次输入了3个样本的特征

W1: 第一层MLP的权重矩阵 [6, 4]

p1: gemm(x, W1) [3, 4]

p2: sigmoid(p1) [3, 4]

W2: 第二层MLP的权重矩阵 [4, 1]

p3: gemm(p2, W2) [3, 1]

p4: sigmoid(p3) [3, 1]

Y: label [3, 1]

这个网络的理论推理FLOPS是多少呢?

p1 = gemm(x, W1) 是一个 3 x 6 的 矩阵 乘以 6 x 4 的矩阵得到 3 x 4 的矩阵，需要 3 x 6 x 4 = 72次乘法和 72 次加法，也就是 144 FLOPS

p2=sigmoid(p1) 这个是对3x4的矩阵每个元素做sigmoid。一般激活层因为计算量小，我们不算入FLOPS里面。

p3 = gemm(p2, W2) 是一个 3x4 的矩阵 乘以 4x1 的矩阵，得到 3x1的矩阵，需要 12次乘法和12次加法，24FLOPS

后面还有激活层，一般我们就不算到理论FLOPS里面了。

所以，这个网络的理论推理FLOPS 是144+24 = 168

如何训练

从上面的流程里，我们可以知道，如果给定了x，如何得到最终的预测结果 p。但是，这里面的各种权重矩阵 W 是怎么来的呢？这个就涉及到训练了，训练的核心算法叫做反向传播（BP）：

- 随机初始化W
- 不断的输入样本x，利用各种带着W的操作，得到预估值p，和实际标签y做对比，得到损失loss
- 基于loss，利用BP算法，更新W

BP的基本原理

神经网络最终就是要优化下面的这个损失 loss (L)

$$L = \sum_i (y_i - f_n(f_{n-1}(\cdots f_1(x_i, \theta_1), \theta_{n-1}), \theta_n))^2$$

$$x_0 \xrightarrow{f_0} x_1 \xrightarrow{f_1} x_2 \cdots x_{n-1} \xrightarrow{f_{n-1}} x_n \longrightarrow L(x_n, y)$$

这个loss的意思，就是 输入 x 经过一番 f 的操作，得到一个预估值，这个值 如果和 y 比较接近，就是我们的目标。

如何衡量预估值和 标签 y 是接近的，上面用了 均方误差的损失函数 ，就是 $(y-p)^2$ 。这个公式里面的 θ 就是每一个操作的参数，比如，MLP里面，gemm的操作的参数就是权重矩阵W，sigmoid操作就没有参数。

我们最终的目标，是在所有样本上，找到合适的 θ ，让loss最小。

我们考虑 神经网络中的一次操作。一次操作就是把 x_i 通过 带着 参数 θ_i 的 操作 f 变成了 x_{i+1}

$$x_{i+1} = f_i(x_i, \theta_i)$$

如果我们要通过梯度法优化参数 θ_i ，那么就要计算如下的导数：

$$\frac{dL}{d\theta_i} = \frac{dL}{dx_{i+1}} \frac{dx_{i+1}}{d\theta_i} = \frac{dL}{dx_{i+1}} \frac{df_i}{d\theta_i}$$

但这个公式里面的 $\frac{dL}{dx_{i+1}}$ 是怎么来的呢？因为：

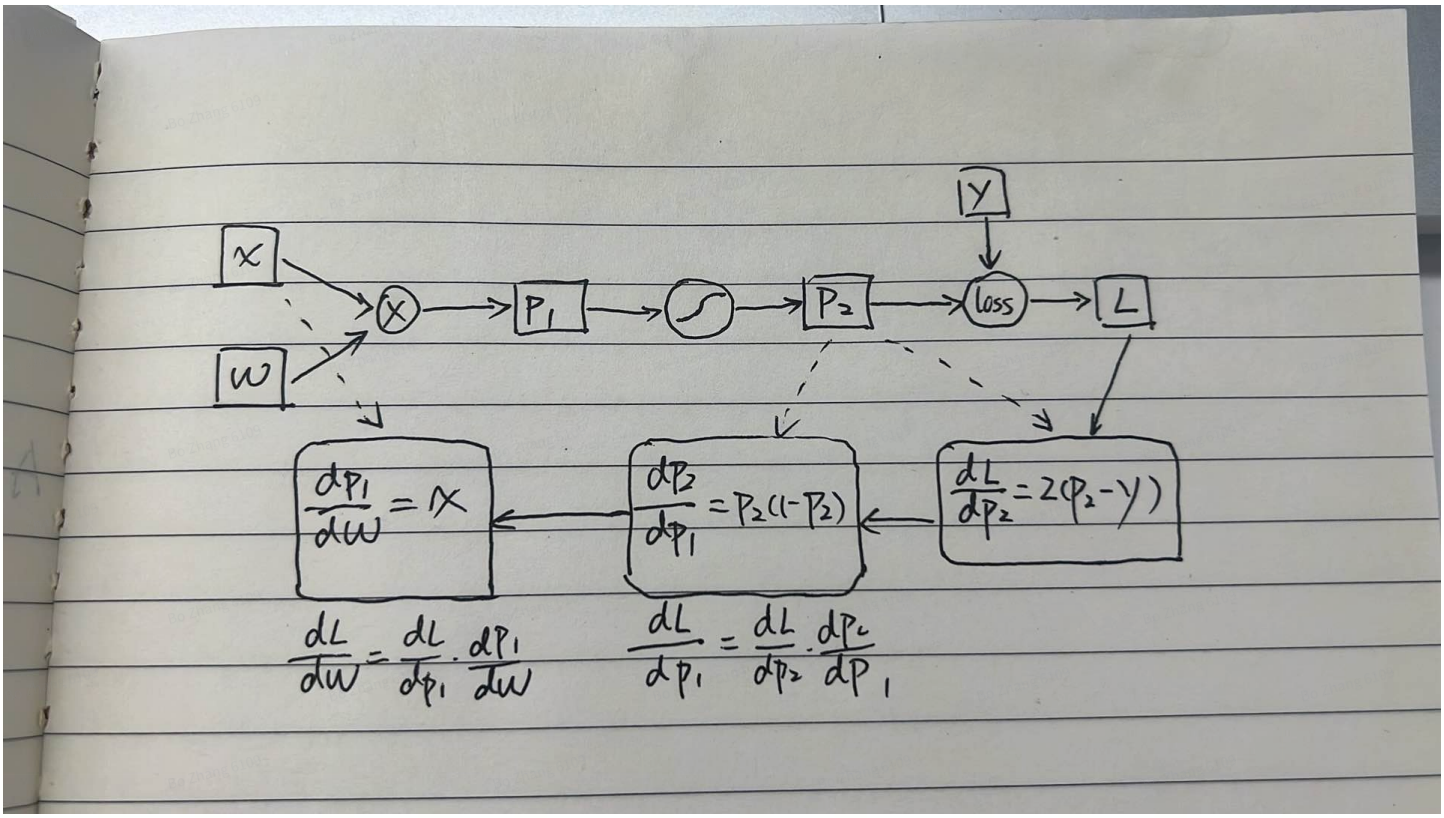
$$x_{i+2} = f_{i+1}(x_{i+1}, \theta_{i+1})$$

$$\frac{dL}{dx_{i+1}} = \frac{dL}{dx_{i+2}} \frac{dx_{i+2}}{dx_{i+1}} = \frac{dL}{dx_{i+2}} \frac{df_{i+1}}{dx_{i+1}}$$

所以，知道 $\frac{dL}{dx_{i+2}}$ 就能知道 $\frac{dL}{dx_{i+1}}$

这里面 $\frac{df_i}{d\theta_i}$ 和 $\frac{df_i}{dx_i}$ 一般被称为 f_i 这层的梯度

显存中需要存什么



我们以单层神经网络为例，上图中方框内的东西都是要存的，但存储的时间周期不一样。

1. 输入x：一直要存到最后，因为最后一步算 dp1/dW 需要用到x
2. 权重矩阵W：一直要存着，模型参数是被更新的对象，要一直存着
3. 中间结果p1: 算完p2就可以扔掉了
4. 中间结果p2: 得算完 dp2/dp1 才能扔掉
5. 标签 y：算完loss就可以扔掉了
6. 中间变量的梯度 dL/dp2, dL/dp1: 这个在反向传播的时候，用完可以随时释放
7. 对参数的梯度 dL/dW：这个在更新完参数可以释放，理论上，算出这个之后，就可以更新参数了，但也有某些特殊情况不会立即更新：（当然，单层MLP里就一个参数W，但多层MLP就有很多W了）
    a. 比如，有时要做全局的grad norm，可以理解为要把对所有参数的导数都算完之后，对所有参数的梯度做一次归一化
    b. Pytorch 默认最后算完了才统一更新

参数是如何更新的?

这个就涉及到优化算法了，比如最简单的是SGD（随机梯度下降法）。

在单层MLP的例子里，我们有参数 W，和 W 的梯度 dL/dW，那么SGD会这么更新W:

$$W^{(t)} = W^{(t-1)} - \alpha\, g^{(t)},\ g^{(t)} = \frac{dL}{dW^{(t)}}$$

但其实还有很多其他优化算法，比如机器学习里用的最多的Adam优化器

Adam 优化器

2014年, Kingma and Ba提出了Adam 优化器来代替随机梯度优化法. Adam引入了两个额外变量: $m^{(t)}$ 和 $v^{(t)}$ 来存储历史梯度的滑动平均 和 历史梯度平方的滑动平均, 然后使用这两个滑动平均对学习率进行自适应调整. Adam优化器在机器学习界得到了广泛使用. 使用Adam优化器进行训练的流程如下

1. 随机初始化权重矩阵 $W^{(0)}$. 将滑动平均 $m^{(0)}$ 和 $v^{(0)}$ 置为0. 设置两个衰减参数 β_1，β_2
2. 对t进行循环
    a. 计算loss对第t-1步权重矩阵 $W^{(t-1)}$ 的梯度 称为 $g^{(t)}$
    b. 更新 历史梯度的滑动平均 $m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1) g^{(t)}$
    c. 更新 历史梯度平方的滑动平均 $v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2) (g^{(t)})^2$
    d. 对 $m^{(t)}$ 进行纠偏 $\hat{m}^{(t)} = m^{(t)}/(1-\beta_1^t)$
    e. 对 $v^{(t)}$ 进行纠偏 $\hat{v}^{(t)} = v^{(t)}/(1-\beta_2^t)$
    f. $W^{(t)} = W^{(T-1)} - \alpha \hat{m}^{(t)} / (\sqrt{\hat{v}^{(t)}} + \epsilon)$, 这里 ϵ 是一个小正数防止分母为0

在使用Adam优化器时, 我们需要在显存中额外存储m和v, 而他们的维度与模型权重的维度相同

所以，如果用Adam优化器，原来如果用 XGB 的存储存了参数，那么在使用Adam的训练里，需要用 4X GB 的空间，如下图所示：

不过，这里讨论的，都是参数用同样的数值精度存储（比如float32）时的情况，机器学习里，还经常使用低精度，比如fp16，int8来进行加速和存储空间的节省。

fp16

我们正常会用float32（fp32）来计算。但是，为了计算速度变快，如果我们的数据变成了 fp16，那么理论上就能快一倍。而且存储也会降低一半。

不过，训练和推理时，对fp16的处理不太一样。以单层MLP为例，推理的时候，x, W, p1, p2 都是fp16，那么相对fp32，存储变成一半，速度提升1倍。

训练时就不一样了，比如 参数和梯度，是用fp32存储的，但是在计算前，会把他们转成fp16，做完计算后再转回fp32。这样做的主要原因是因为用来防止溢出，在fp16的grad上有一个scale的参数。

Train With Mixed Precision 这里介绍了这个过程的细节：

1. Maintain a primary copy of weights in FP32.
2. For each iteration:
   a. Make an FP16 copy of the weights.
   b. Forward propagation (FP16 weights and activations).
   c. Multiply the resulting loss with the scaling factor *S*.
   d. Backward propagation (FP16 weights, activations, and their gradients).
   e. Multiply the weight gradient with 1/*S*. （这一步的时候，就需要把fp16的梯度变成fp32了，否则乘以 1/S 有可能溢出）
   f. Complete the weight update (including gradient clipping, etc.).

一般情况下，S是一个大于1的数，因为往往在网络收敛的后期，梯度太小了，如果用fp16，可能就会被认为梯度为0。

因此，在fp16的情况下，训练的内存情况如下

参数 FP16
参数 FP32
参数梯度FP16
参数梯度FP32
历史梯度滑动平均FP32
历史梯度平方和FP32

假设FP16的参数显存占用X GB，那么总的显存占用就到了10X GB

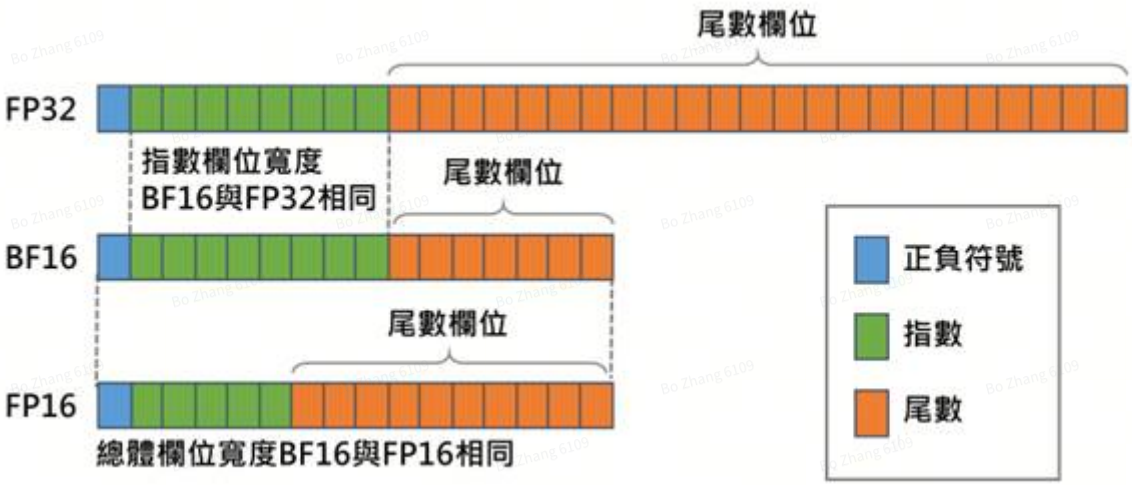
一个参数平均占用20bytes = 10 * 2 bytes。

那么，比如1.3B的GPT3模型，他在训练时的显存占用就大概是 26GB。如果他用来推理，可能2.6GB就够了，所以，训练和推理的显存占用是差距巨大的。

微软的DeepSpeed框架里的Zero优化，就是针对了这个特点，对于大模型的显存占用，做了优化。后面再说是如何优化的。

bf16

bf16和fp16有所区别，他的指数部分和fp32是一样的。



所以，在用bf16训练时，一般不需要scale梯度。但是，bf16存在的问题是，一个很大的数 a + 一个很小的数 b = a。也就是将梯度 dW 累加到 权重矩阵 W 上面时，如果dW的一些项太小，那么W对应的位置不会被更新。

所以，用bf16训练时，一般需要保留一份fp32的W，用来做累加。（需要注意的是，DeepSpeed当前master里没有实现这个特性）

一个复杂一点的网络的例子

Transformer

下面是transformer的伪代码 ，原始代码可以参考

- https://github.com/karpathy/nanoGPT/blob/master/model.py#L60
- https://github.com/karpathy/nanoGPT/blob/master/model.py#L101

```
# x : [B, T, C]
# B : batch_size
# T : seq_len
# C : dimension

```

```
x = layernorm(x)
q, k, v = qkv_proj(x).split()
# [B, T, C] x [C, 3C] -> [B, T, 3C]: 6BTC^2 FLOPS
attn = q @ k.T
# [B, T, C] x [B, C, T] = [B, T, T] : 2BT^2C FLOPS
attn = softmax(attn)
y = attn @ v
# [B, T, T] x [B, T, C] -> [B,T, C] : 2BT^2C FLOPS
y = proj(y)
# [B, T, C] x [C, C] -> [B, T, C] : 2BTC^2
y = layernorm(y)
y = fc1(y)
# [B, T, C] x [C, 4C] -> [B, T, 4C] : 8BTC^2
y = gelu(y)
y = fc2(y)
# [B, T, 4C] x [4C, C] -> [B, T, C] : 8BTC^2
```

所以，一层Transformer需要 24BTC^2 + 4BCT^2 FLOPS

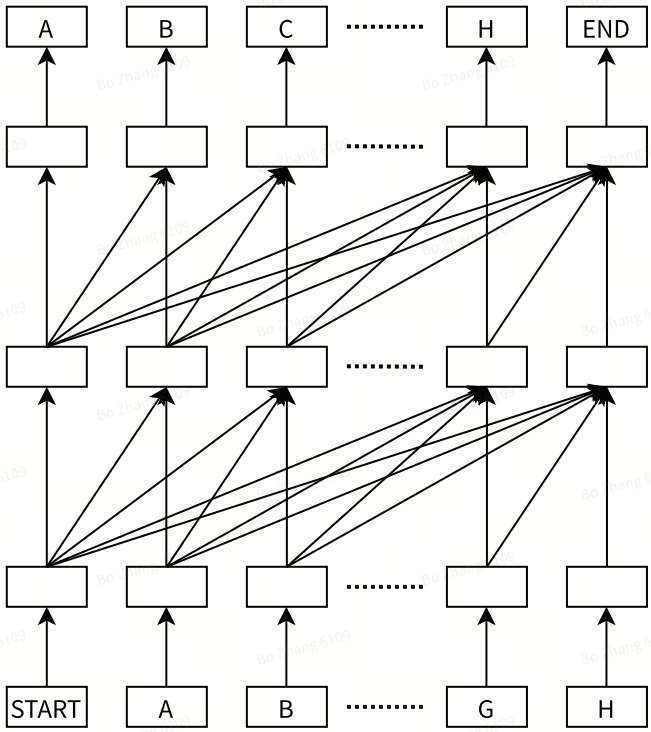
然后，attn的计算占比是

$$\frac{4BCT^2}{24BTC^2 + 4BCT^2} = \frac{T}{6C + T}$$

在一些模型配置中，如果C > T （比如GPT3-175B C = 12288, T = 8192），那么attn的占比其实是很低的。

以上推导虽然是针对 双向Transformer的，但是在单向Transformer的训练时，也适用。唯一区别是单项transformer需要把attn 矩阵变成三角阵，一半抹成0。

GPT



GPT在训练时和推理时的模式不太一样：

- 推理时，以上面的例子为例，比如，我通过A，预测了下一个Token是B，这个时候才能知道下一次预估的输入是B的Token，然后才能去预测C。所以，推理在这个意义是串行的，只能一个个token生成
- 训练时，我们会直接从语料中拿到 [START]AB---GH，然后可以直接过整个网络，得到最后一层的输出，再和 AB--GH[END]的label做对比，计算loss。

这个区别，导致了GPT训练时较容易打满GPU，但推理时比较难。

假设，我们已经预测出H了，那么输入H，得到下一个Token的预估值，这个过程，每层Transformer是什么计算复杂度呢？

```
# qkv_cache : [B, T-1, 3C]
# x : [B, 1, C]
# B : batch_size
# T : seq_len
# C : dimension

x = layernorm(x)
qkv = qkv_proj(x)
# [B, 1, C] x [C, 3C] -> [B, 1, 3C]: 6BC^2 FLOPS
qkv = concat(qkv, qkv_cache)
# [B, 1, 3C], [B, T-1, 3C] -> [B, T, 3C]
q, k, v = qkv.split()
attn = q[:, -1, :] @ k.T
```

```
# [B, 1, C] x [B, C, T] = [B, 1, T] : 2BTC FLOPS
attn = softmax(attn)
y = attn @ v
# [B, 1, T] x [B, T, C] -> [B,1, C] : 2BTC FLOPS
y = proj(y)
# [B, 1, C] x [C, C] -> [B, 1, C] : 2BC^2
y = layernorm(y)
y = fc1(y)
# [B, 1, C] x [C, 4C] -> [B, 1, 4C] : 8BC^2
y = gelu(y)
y = fc2(y)
# [B, 1, 4C] x [4C, C] -> [B, 1, C] : 8BC^2
```

所以总复杂度是 24BC^2 + 4BTC

访存

上面讨论的，都是GPU的计算。但是，实际程序运行时，还需要访存。GPU的存储如果从科普角度考虑，可以认为分2级：

- Global Memory：比如A100 80G，用的HBM2存储，2TB/s带宽
- Shared Memory：比如A100，192KB per SM，108个SM

然后Global Memory慢，Shared Memory快。如果从Global Memory中读东西，带宽是 2TB/s。

我们来算算推理时，计算一个token时的访存带宽情况。

> 以矩阵乘为例，输入[M, K] x [K, N] -> [M,N] 计算时间为2MKN/TFLOPS，访存时间为：(MN+MK+KN)/memory bandwidth。需要注意的是，估算的时候，不需要考虑实现，仅做最理想的分析。比如矩阵乘的访存，输入矩阵不可能只读一次，但是估算理论峰值不考虑。恰好这样估算是可以评估你实现的好坏。

	N layers	Dim	Head	Dim per Head
1.3B	24	2048	16	128
13B	40	5120	40	128
175B	96	12288	96	128

我们以1.3B/13B为例，看一下推理时的主要OP的计算延时和访存延时，按照batch_size=1，按照seq_len = 4096算一下

	M	K	N	FLOPS	访存	计算延时 us	访存
1.3B							
attn = q[:, -1, :] @ k.T	1	2048	4096	16777216	8394752	0.0538	
y = attn @ v	1	4096	2048	16777216	8394752	0.0538	
y=fc1(y)	1	2048	8192	33554432	16787456	0.1075	
y=fc2(y)	1	8192	2048	33554432	16787456	0.1075	
13B							
attn = q[:, -1, :] @ k.T	1	5120	4096	41943040	20980736	0.1344	
y = attn @ v	1	4096	5120	41943040	20980736	0.1344	
y=fc1(y)	1	5120	20480	209715200	104883200	0.6722	
y=fc2(y)	1	20480	5120	209715200	104883200	0.6722	

但训练时就不一样了

	M	K	N	FLOPS	访存	计算延时 us	访
1.3B							
attn = q @ k.T	4096	2048	4096	68719476736	33554432	220.2547	
y = attn @ v	4096	4096	2048	68719476736	33554432	220.2547	
y=fc1(y)	4096	2048	8192	137438953472	58720256	440.5095	
y=fc2(y)	4096	8192	2048	137438953472	58720256	440.5095	
13B							
attn = q @ k.T	4096	5120	4096	171798691840	58720256	550.6368	
y = attn @ v	4096	4096	5120	171798691840	58720256	550.6368	
y=fc1(y)	4096	5120	20480	858993459200	209715200	2753.1842	
y=fc2(y)	4096	20480	5120	858993459200	209715200	2753.1842	

所以可以看到，GPT训练时，是计算瓶颈，推理时，是访存瓶颈。

这个也解释了为什么推理用A100 ROI高。因为H100 vs A100的带宽和fp16算力对比如下：

	A100 SXM	H100 SXM	
访存带宽	2.039TB/s	3.35TB/s	1.64X
FP16	624T	1979T	3.17X

算力3倍，带宽只有1.64倍，做推理ROI就低了。

分布式计算

一般，分布式计算就得考虑通信，比如给定一个模型，用纸算算通信是不是瓶颈是基本功。

在GPT3-175B之前，一般只有训练才考虑分布式计算，推理时单卡就够了，但GPT3-175B之后，推理也需要用到分布式计算了。不过目前单机8卡可以搞定，所以暂时还不需要考虑在推理时的网络通信，不过未来如果模型更大，推理时还是有可能要用到多机的。

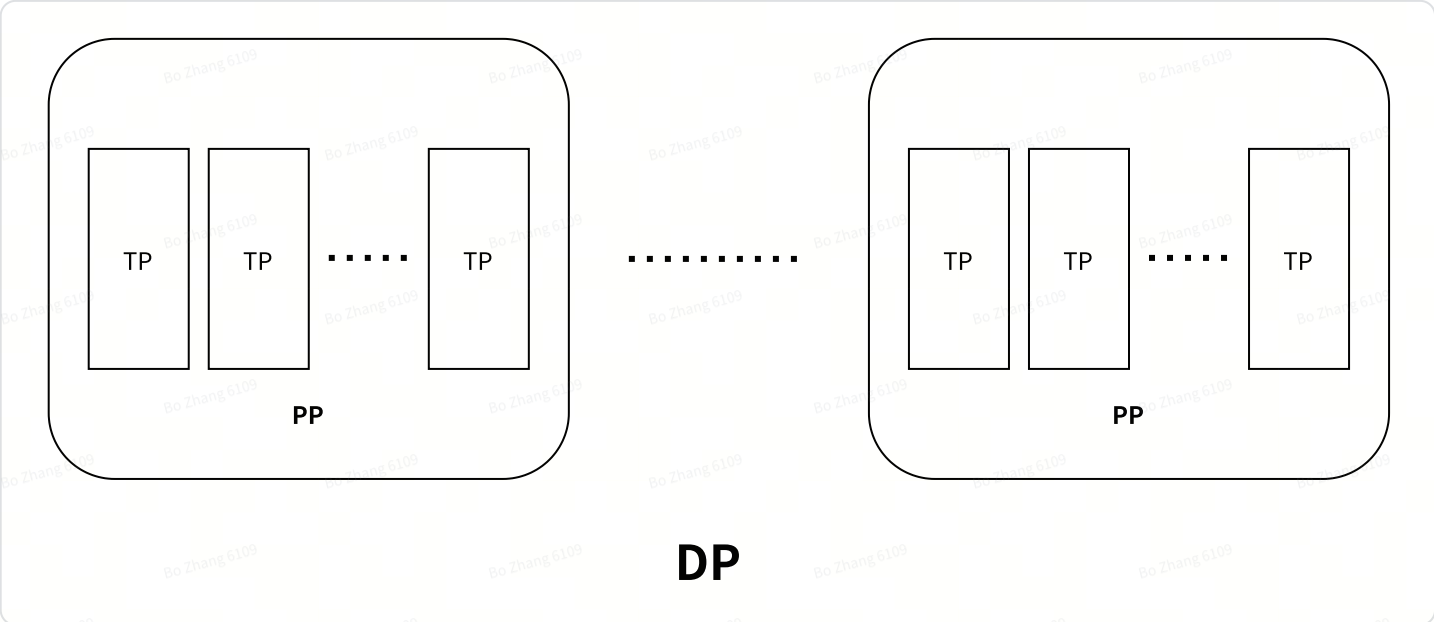
分布式训练

终极状态

分布式计算里，有3种并行算法

- 数据并行(DP)
- 流水线并行(PP)
- Tensor并行(TP)

如果训练规模非常大，需要把3种模式全部用起来，大概是下面这个架构：



也就是说

- 把机器分成N组，组之间用DP
- 一组机器有M台机器，不同台之间用PP
- 一台机器有8张卡，不同卡之间用TP

下面，我们再介绍DP，PP，TP分别怎么搞的。

DP 数据并行

我们还是从一个简单的例子开始，就是

$$Y = XW,\ L = f(Y)$$

X是输入，shape是[M, K], W 是权重矩阵，shape是 [K, N]， Y是输出，shape是 [M, N]。L是loss，是一个标量。

如果是数据并行，那么就是切X，比如切成P份，每份就是 [M/P, K]，每个计算节点上计算 1/P 的数据。W 矩阵每个计算节点都存。

但是，dL/dW 就是分散在 P 个节点上的，需要通过一次all reduce操作加到每张卡的W上去。

PP 流水线并行

假设网络后好几层，做如下的操作：

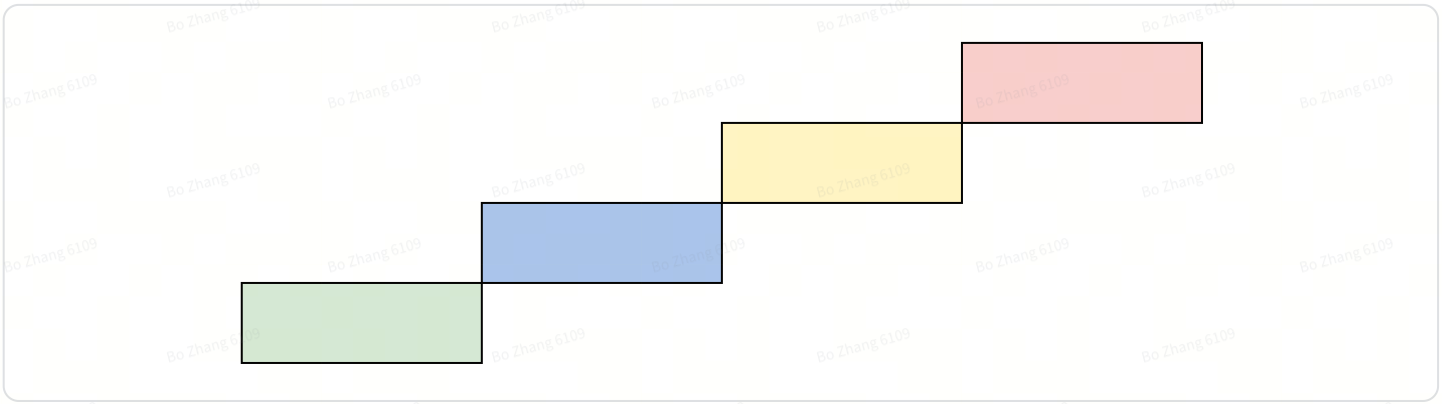
X1 = X0 W0

X2 = X1 W1

X3 = X2 W2

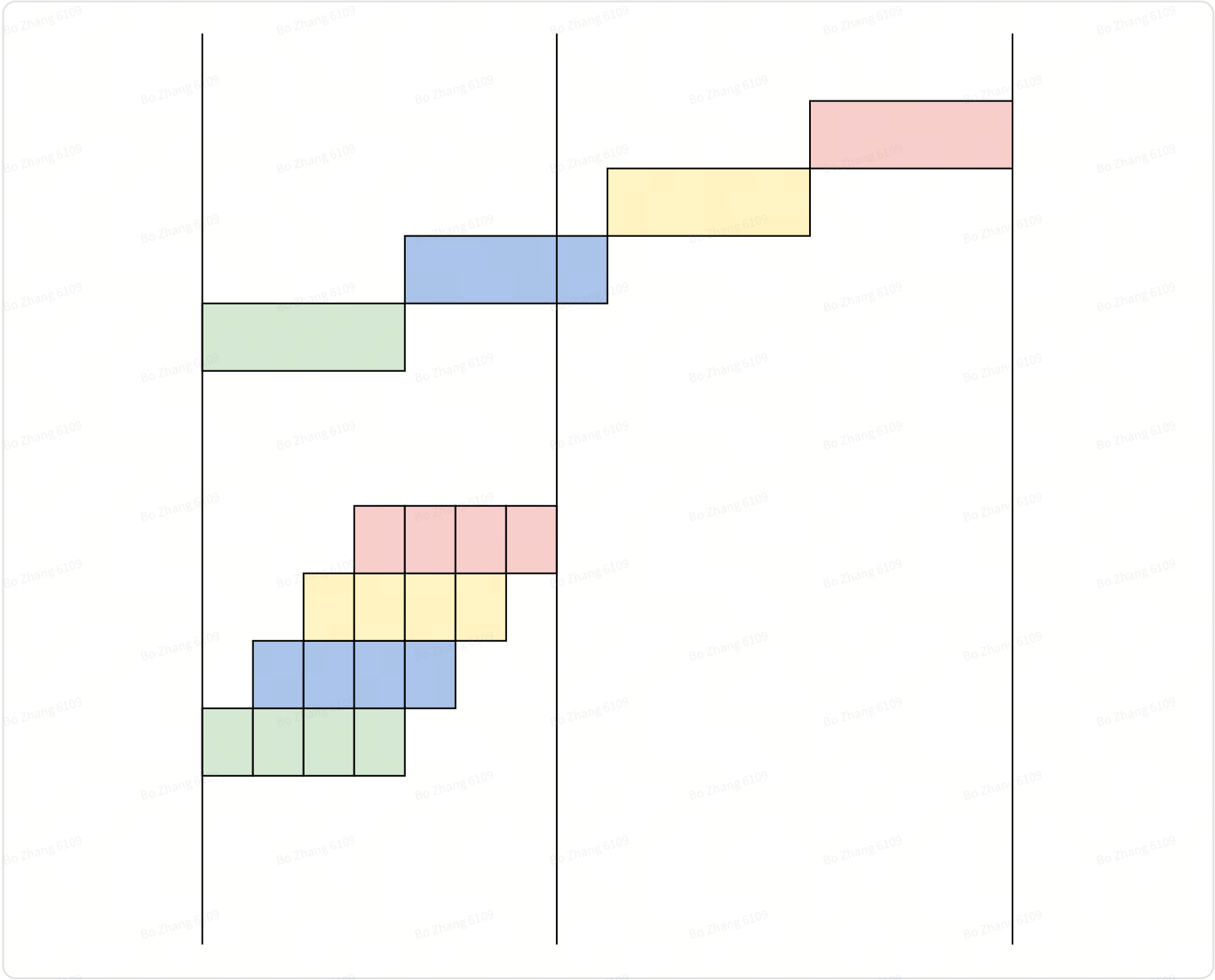
X4 = X3 W3

假设X0～X3的shape都是 [M, N], W0~W3的shape都是[N, N]。然后，每一层都在不同的卡上。那么，就需要一层算完，才能开始算下一层，计算的逻辑大概如下图



但是，我们可以把X切开，比如每个切成4份，每个X的shape就是 [M/4, N]。

那么，当计算完 X1,0 = X0,0 W0的时候，第二张卡就可以开始计算 X2,0 = X1,0 W1了，不需要等待第一张卡计算 X1,1 = X0,1 W0。于是计算过程变成下图：



可以看到，下面PP之后，4张卡的总耗时只有一开始的7/16

TP Tensor并行

还是以 Y = X W 为例，在DP和PP里，都是切的X，把X 从 [M, K] 切成了 P 个 [M/P, K]。而TP的思路是切W，把W从 [K, N] 切成 Q 个 [K, N/Q]。假设有Q个计算节点，每个节点上放一份W。那么，就需要把X拷贝到每个计算节点上。

Y = [Y1, Y2, ..., YQ] = X [W1, W2, ..., WQ]

最后需要一次all gather操作，把每个计算节点的Y copy到所有节点。

所以，经过DP,PP,TP后，无论是过大的X，还是过大的W，理论上都可以被切了。

一些其他优化策略

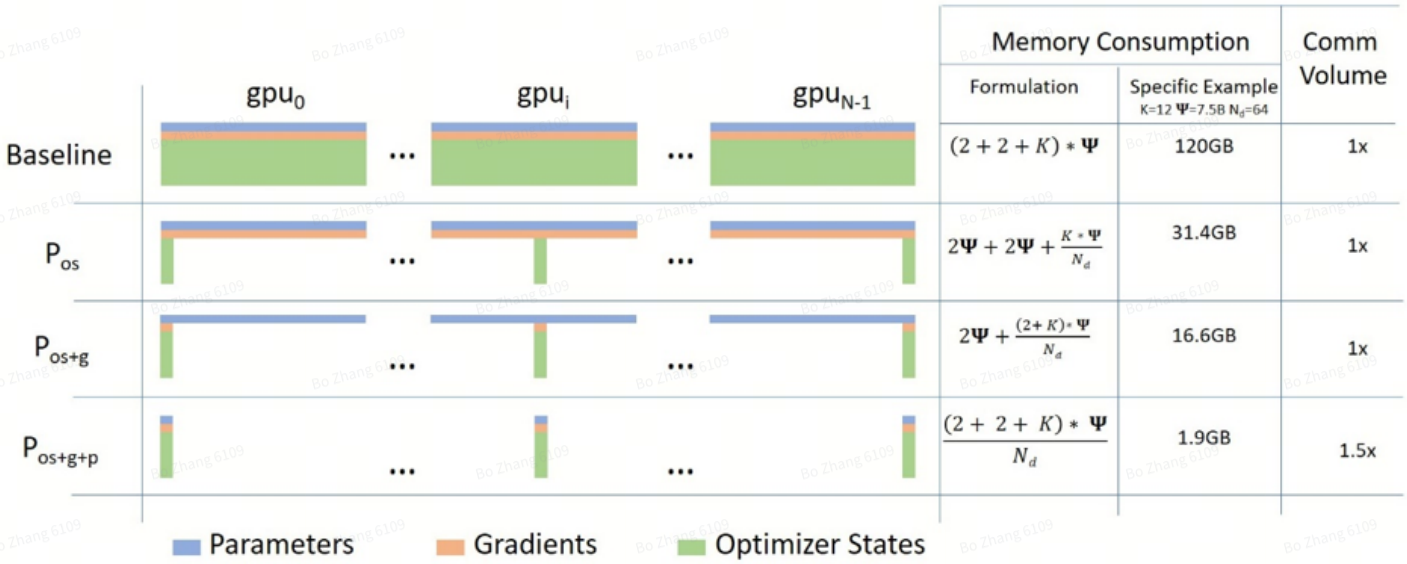
以上关于DP，PP，TP的描述是大面上的，实际上还有一些别的优化细节。

DeepSpeed Zero

DeepSpeed Zero本质是一个DP的框架。他的优化主要针对了前面提到的这张图：

<table>
<tr><td>参数 FP16</td></tr>
<tr><td>参数 FP32</td></tr>
<tr><td>参数梯度FP16</td></tr>
<tr><td>参数梯度FP32</td></tr>
<tr><td>历史梯度滑动平均FP32</td></tr>
<tr><td>历史梯度平方和FP32</td></tr>
</table>

它主要优化了显存的存储方式，把显存切分存储到不同的计算节点上。根据优化力度的不同，又分为Zero1/2/3



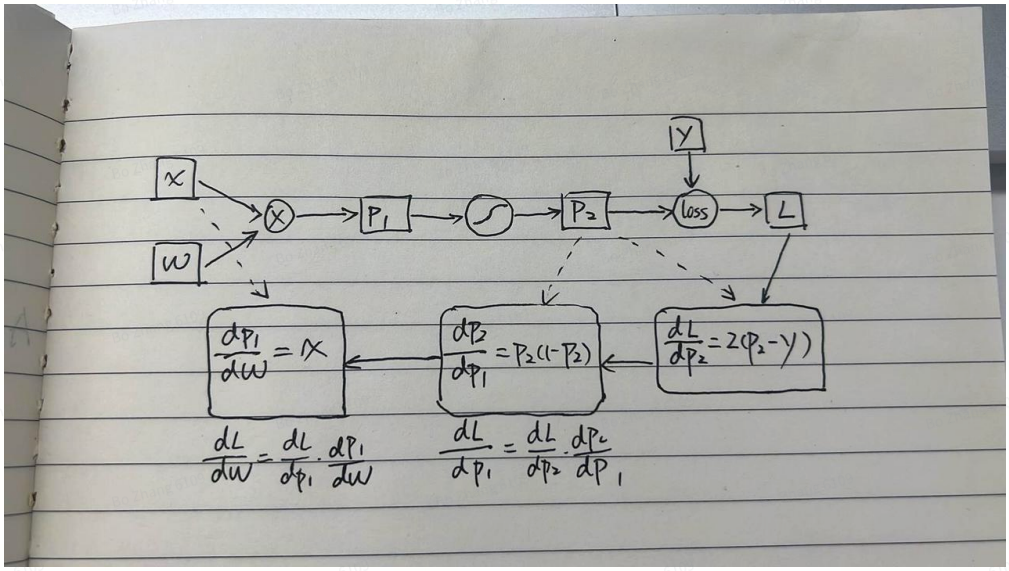
这里面Zero1就是只把优化器的参数切到不同节点上，Zero2 把梯度和优化器参数切到不同节点上，Zero3把参数，梯度和优化器参数，都切到的不同节点上。

这里Zero3把参数也切了，为啥还说他是一个DP（数据并行）框架，而不是模型并行的框架。是因为，区分数据并行和模型并行的本质，不是模型参数有没有切开存储。而是：

- 输入数据有没有切分开发送到不同的计算节点，如果是数据并行，输入就要切开，如果是模型并行，输入就不需要切开。（当然，如果是混合了数据并行和模型并行的，那就比较复杂了，比如我一开始说的那个终极状态的例子）

Zero3在训练时，会把输入X 切分开到不同计算节点，但会在这个节点上all gather其他节点的模型参数做计算，只是算完后，会把gather的参数空间释放掉。

重计算



还是这个单层MLP的图，我们可以看到，在forward的过程中，一些变量，比如 x，p2，都得存下来，供backward的时候使用。如果有多层的MLP，那就有很多变量在forward的过程中不能用完就删，而是要存下来。

那么，如果我们要节省这些存储呢？重计算的idea是用时间换空间，比如，我可以在forward后不存，但是在backward的时候重新forward一下。不过，forward也需要有一个起始点，比如每次都从头forward，那就太费了。所以，重计算里还是设置了一些存储点，比如一共N个MLP，我们每M个，存一个中间结果。