Algorithm

- OP frequency
  - ONNX OP statistics
- Model Compression
  - Model Quantization
  - Neural Arch Search
  - Knowledge Distillation
  - Pruning
  - reference
- Data type
  - data type

OP frequency

ONNX OP statistics

alphabet order

```
   3 Abs
4325 Add
  20 And
   2 ArgMax
 101 AveragePool
3467 BatchNormalization
3152 Cast
   8 CategoryMapper
  27 Ceil
 407 Clip
   2 Compress
2706 Concat
7714 Constant
 788 ConstantOfShape
6448 Conv
   2 ConvInteger
   4 ConvTranspose
   1 CumSum
3227 DequantizeLinear
 503 Div
 100 Dropout
  82 DynamicQuantizeLinear
   5 DynamicQuantizeLSTM
  97 Equal
  24 Erf
 194 Exp
 132 Expand
  40 Flatten
  52 Floor
```

12 FusedMatMul
7681 Gather
251 Gemm
29 GlobalAveragePool
8 Greater
2 Hardmax
91 Identity
150 InstanceNormalization
228 LeakyRelu
52 Less
1 LessOrEqual
91 Log
30 Loop
90 LRN
5 LSTM
861 MatMul
96 MatMulInteger
2 Max
522 MaxPool
4 Min
3536 Mul
3 Neg
688 NonMaxSuppression
843 NonZero
41 Not
4 OneHot
160 Pad
234 Pow
100 PRelu
1 preprocess
276 QLinearAdd
5 QLinearAveragePool
67 QLinearConcat
1191 QLinearConv
4 QLinearGlobalAveragePool
72 QLinearLeakyRelu
30 QLinearMatMul
62 QLinearMul
10 QLinearSigmoid

1577 QuantizeLinear
7 Range
100 Reciprocal
6 ReduceMax
466 ReduceMean
65 ReduceMin
8 ReduceSum
4794 Relu
2397 Reshape
43 Resize
48 RoiAlign
2 Round
2 Scan
12 Scatter
36 ScatterElements
2229 Shape
90 Sigmoid
2276 Slice
230 Softmax
135 Split
274 Sqrt
1700 Squeeze
879 Sub
249 Sum
146 Tanh
47 Tile
60 TopK
1469 Transpose
7973 Unsqueeze
22 Upsample
15 Where

Frequency order

1 CumSum
1 LessOrEqual
1 preprocess
2 ArgMax
2 Compress

2 ConvInteger
2 Hardmax
2 Max
2 Round
2 Scan
3 Abs
3 Neg
4 ConvTranspose
4 Min
4 OneHot
4 QLinearGlobalAveragePool
5 DynamicQuantizeLSTM
5 LSTM
5 QLinearAveragePool
6 ReduceMax
7 Range
8 CategoryMapper
8 Greater
8 ReduceSum
10 QLinearSigmoid
12 FusedMatMul
12 Scatter
15 Where
20 And
22 Upsample
24 Erf
27 Ceil
29 GlobalAveragePool
30 Loop
30 QLinearMatMul
36 ScatterElements
40 Flatten
41 Not
43 Resize
47 Tile
48 RoiAlign
52 Floor
52 Less
60 TopK
62 QLinearMul
65 ReduceMin
67 QLinearConcat
72 QLinearLeakyRelu
82 DynamicQuantizeLinear
90 LRN
90 Sigmoid
91 Identity
91 Log
96 MatMulInteger
97 Equal
100 Dropout
100 PRelu
100 Reciprocal

101 AveragePool
132 Expand
135 Split
146 Tanh
150 InstanceNormalization
160 Pad
194 Exp
228 LeakyRelu
230 Softmax
234 Pow
249 Sum
251 Gemm
274 Sqrt
276 QLinearAdd
407 Clip
466 ReduceMean
503 Div
522 MaxPool
688 NonMaxSuppression
788 ConstantOfShape
843 NonZero
861 MatMul
879 Sub
1191 QLinearConv
1469 Transpose
1577 QuantizeLinear
1700 Squeeze
2229 Shape
2276 Slice
2397 Reshape
2706 Concat
3152 Cast
3227 DequantizeLinear
3467 BatchNormalization
3536 Mul
4325 Add
4794 Relu
6448 Conv
7681 Gather
7714 Constant
7973 Unsqueeze

Model Compression

Model compression techniques are mainly classified into quantization (Zafrir et al.,2019), pruning (Hoefler et al., 2021; Vadera and Ameen, 2022), Neural Architecture Search(NAS) (Sun et al., 2019), and knowledge distillation

Model Quantization

Quantization Granularity

Quantization is a magic spell to reduce the memory footprint of a model. But often quantization leads to a drop in the accuracy of the model. This is where the Granularity of quantization comes into the picture. Selecting the right granularity helps in maximizing the quantization without much drop in the accuracy performance

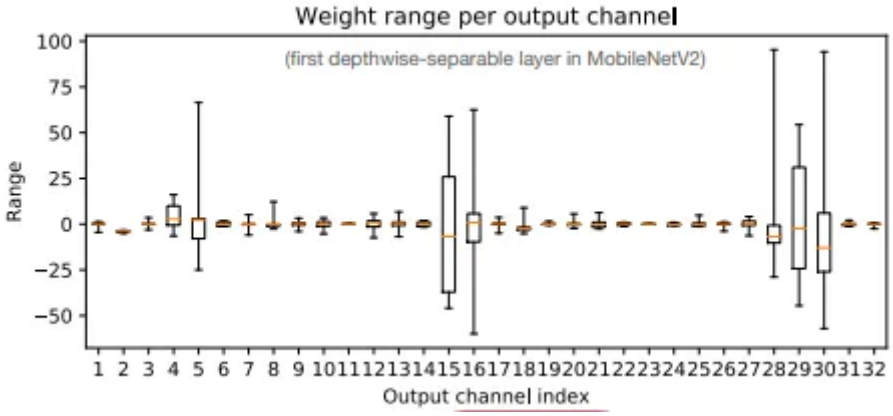
Per-Tensor Quantization

In per-tensor quantization, the same quantization parameters are applied to all elements within a tensor. Applying the same parameters across tensors will cause a drop in accuracy.

Per-Channel Quantization

In per-channel quantization, different quantization parameters are applied to each channel of a tensor independently. This often leads to a lower error while quantizing compared to per tensor quantization.

Per-channel quantization captures variations in different channels more accurately. This usually helps in CNN models where the range of weights varies over different channels.



Mix-precision Quantization

different layer using different precision.

Quantization Method

Quantization Method	Accuracy	Model Size Reduction	Inference Speed	Ease of Implementation
GGUF	High (adaptive quantization of parameter groups)	Significant (grouped quantization)	High (optimized through grouped quantization)	Moderate (requires parameter grouping)
AWQ	High (asymmetric quantization for accuracy)	Moderate	Moderate to High (asymmetric approach)	Moderate (asymmetric scaling)
GPTQ	Moderate (post-training fine-tuning)	Moderate	Moderate (benefits from post-quantization)	Easy (applied post-training)
GGML	High (mixed-precision within groups)	Significant (effective group-wise precision)	High (mixed-precision within groups)	Complex (group-wise mixed-precision)
PTQ	Moderate (quick and easy implementation)	High (straightforward implementation)	High (quick deployment)	Easy (quick application)
QAT	Very High (trained with quantization in mind)	Moderate to High (requires complex training)	High (optimized for quantization during training)	Complex (requires training with quantization)
Dynamic Quantization	Moderate (applied during inference)	High (applied at inference time)	Moderate to High (quick deployment)	Easy (inference-based)
Mixed-Precision	High (selectively applied different precisions)	Moderate to High (selectively applied)	High (selective precision boosts performance)	Complex (selective precision assignment)

Neural Arch Search

Knowledge Distillation

Pruning

Model Compression

reference

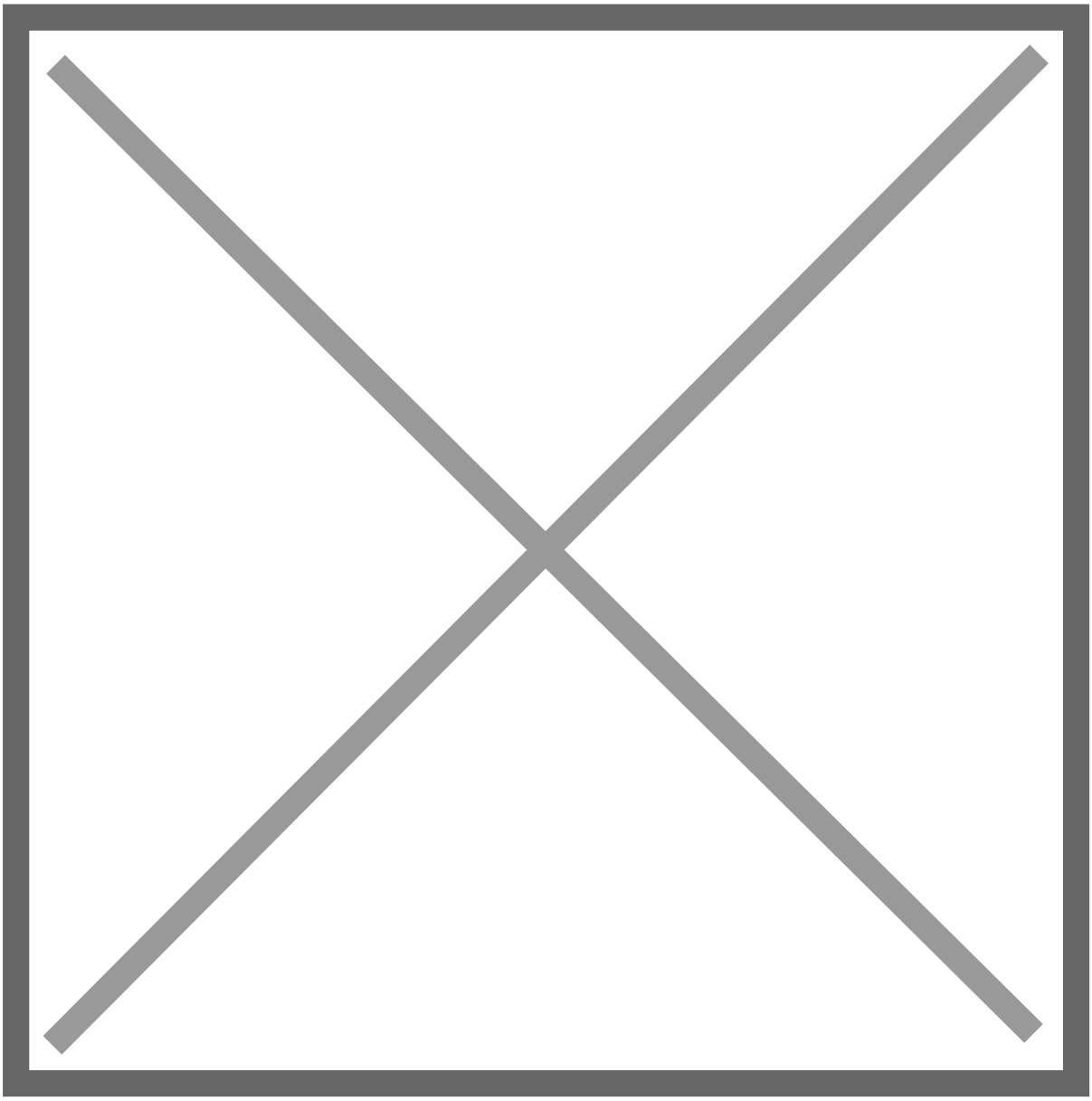
量化技术背景

从 CNN量化说起

在传统 CNN网络中，为了加速网络的推理速度，一种非常有效的方法是 INT8量化，即将权重与激活 (feature map) 的浮点数值量化成 8-bit整型表示，这样做的好处：一是将原来 32-bit 数值用更低位数替代，从而减少计算前后的数据访问量，这对于目前数据访问时间远大于计算操作的微架构来说，可以节省可观的时间；二是这种情况下卷积操作或者矩阵操作 (GEMM) 可以使用整型计算，大部分情况下专门针对此设计的硬件单元整型计算速度是要大于浮点计算的，而且拥有更多整型计算单元，因此 INT8计算吞吐量更高。

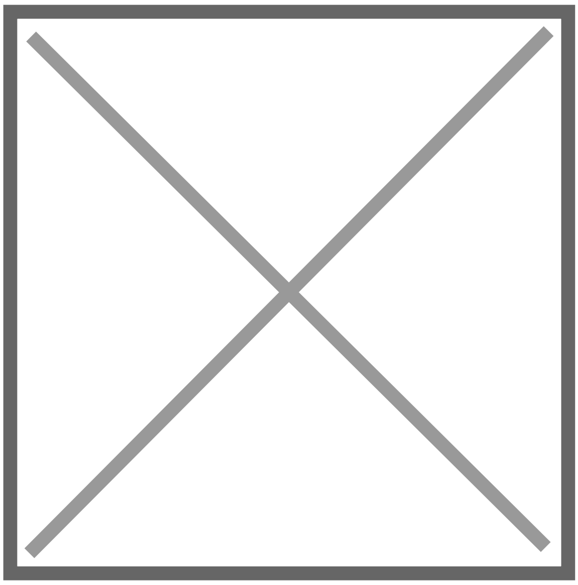
CNN依靠量化技术以及其他推理优化手段，使得像目标检测等 CV 领域模型可以在端侧设备满足实时推理要求。注意，由于大部分推理引擎只对卷积层或者矩阵乘法层进行量化推理，这使得量化层和非量化层之间是有反量化和量化开销，因此量化能取得加速的关键在于访存量的节省以及整型计算的加速远大于反量化这部分带来的开销。

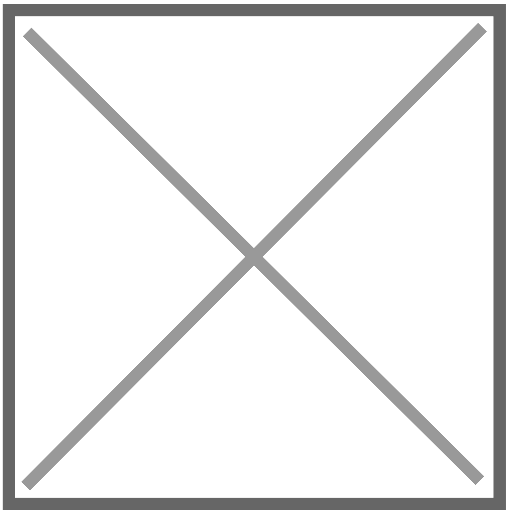
现在流行的大语言模型 (LLM)架构都是从 transformer 演变而来的解码器架构 (GPT)，其主要计算层是由矩阵乘法（GEMM）组成的注意力层，因此 LLM量化技术和 CNN量化的基础以及原理思想是有很大相通性的 (CNN的量化基础知识和算法可以参见《 white paper on Quantization。



量化公式

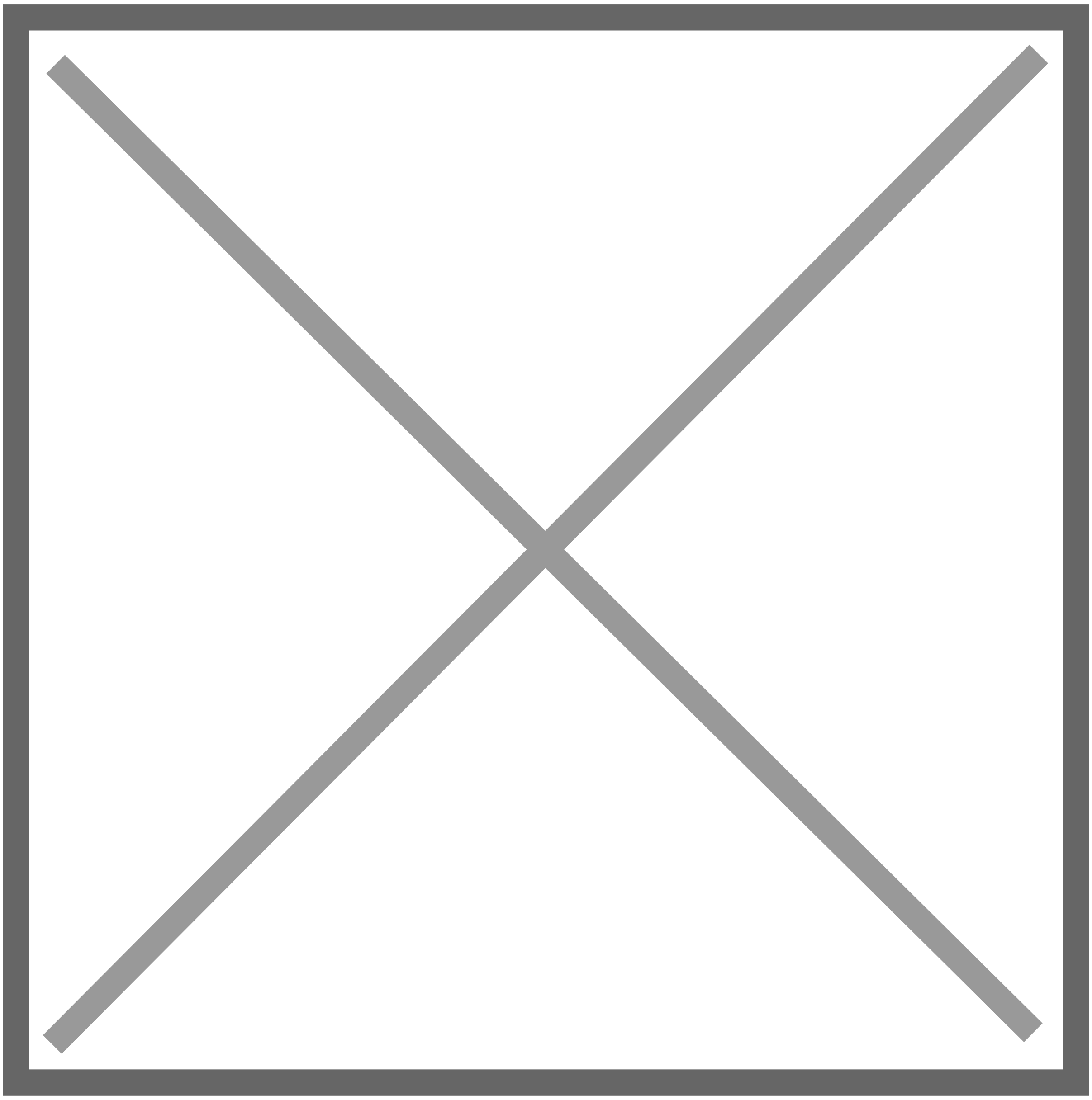
这里主要关注最常见的对称量化，简单来说，INT8 量化即将浮点数通过缩放因子映射到范围在 [-128, 127] 内的 8bit 表示，量化公式如下:





Round 表示四舍五入都整数，Clip 表示将离群值 (Outlier) 截断到 [-128, 127] 范围内。对于 scale 值，通常按如下方式计算得到：

下图展示的是浮点数进行量化的示意图，结合上述公式可以知道量化误差主要来自 Round和 Clip，量化误差不可避免，但是太高的量化误差会使得最终网络推理得到不准确或者错误的结果，这种情况再快的量化推理速度也是没有意义的，因此无论是 LLM还是 CNN在使用量化时，速度和精度 (效果) 是一对重要的 trade-off。

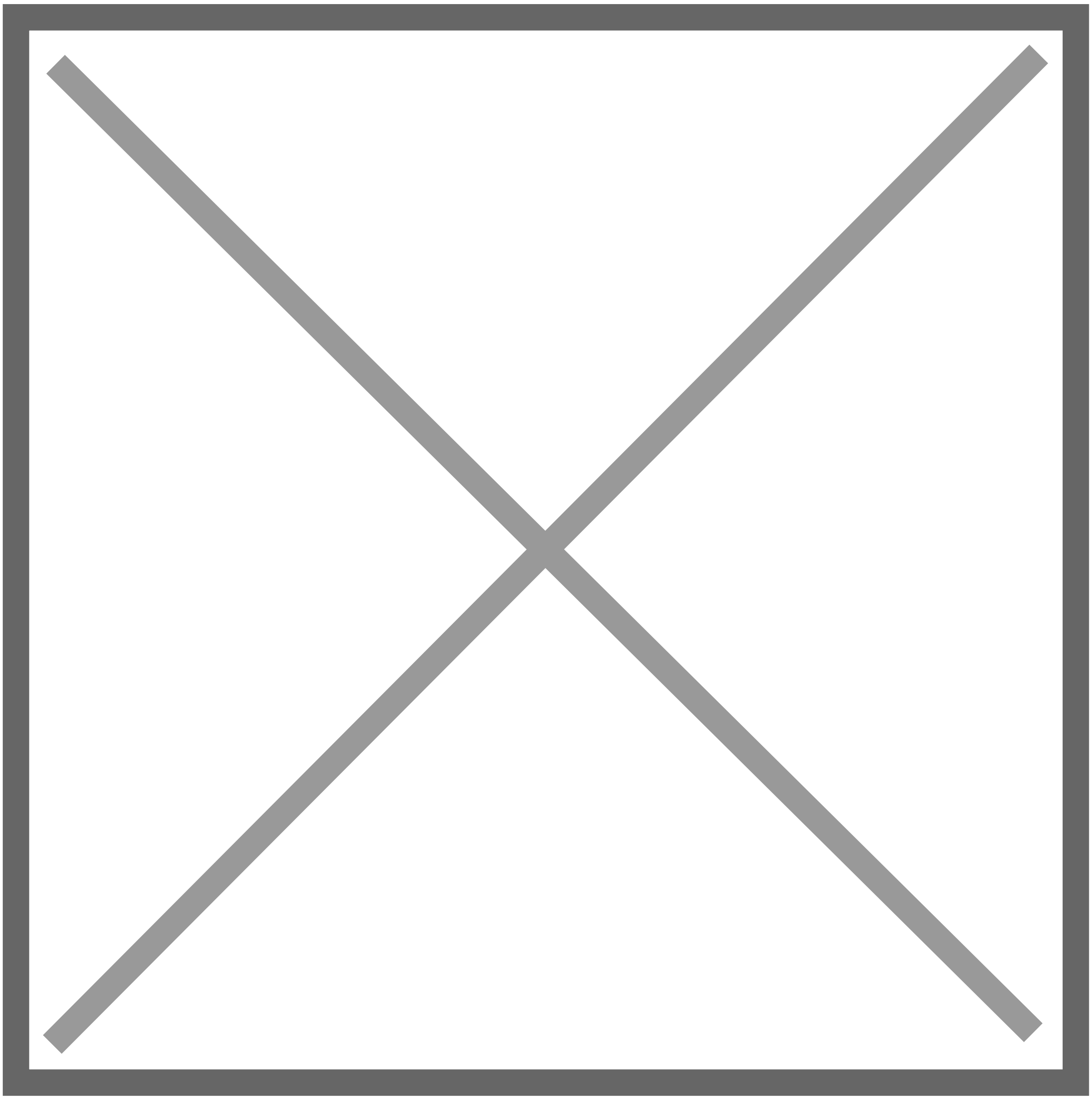


量化粒度

无论是量化激活还是权重，本质上都是对张量tensor矩阵的量化。从量化公式中，可以看出，量化是需要一个scale参数的，量化的粒度就在于是网络layer的整个大的tensor都共用一个scale呢？还是每个构成大tensor的小tensor分别使用一个scale呢？这种粒度的考虑往往很影响量化后最终效果。

在CNN中，针对权重和激活数值分布的特性，经常对激活tensor进行per-tensor这种粗粒度量化，而对于由多个卷积核tensor组成的权重（大tensor），则是采用每个卷积核对应一个scale这种更加细粒度的per-channel量化。

后面讲到LLM，我们会知道主流LLM量化采用的是更加细粒度的per-vector或者per-group（对vector进行分组）量化方式。

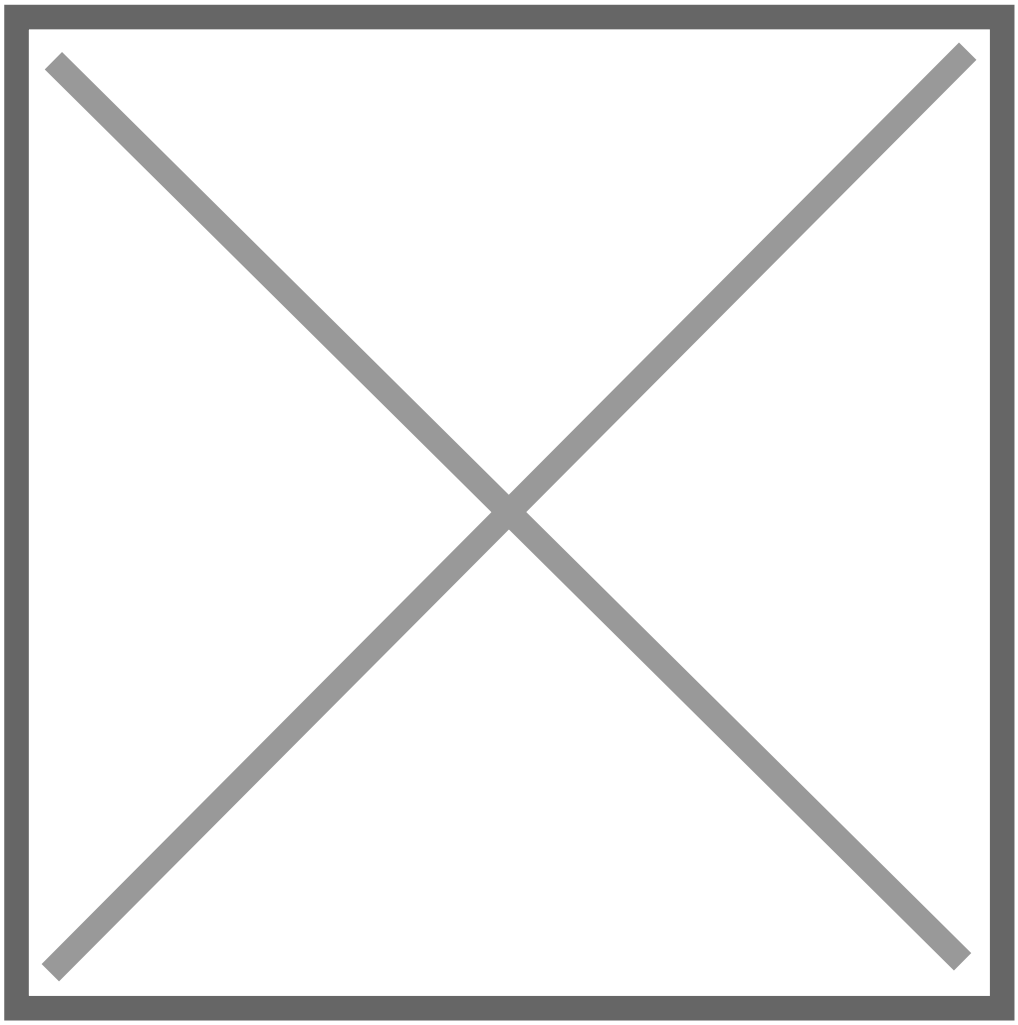


量化数值表示

不同数据类型会影响“精度”，当前主流的浮点类型表示为 float32、float16 或 bfloat16 之一，此外还出现了比 INT8更加激进的 INT4量化，后者实践证明是 LLM量化中一种性价比更高的方式。在这里对这些数值表示做简单介绍：

- 第一部分为 `sign` 符号位 s，占 1 bit，用来表示正负号；
- 第二部分为 `exponent` 指数偏移值 k，占 8 bits，用来表示其是 2 的多少次幂；
- 第三部分是 `fraction` 分数值（有效数字）M，占 23 bits，用来表示该浮点数的数值大小。

- **Float32 (FP32)**。标准的 IEEE 32 位浮点表示，指数 8 位，尾数 23 位，符号 1 位，可以表示大范围的浮点数。大部分硬件都支持 FP32 运算指令。



●**TensorFloat-32(TF32)** 。使用 19 位表示，结合了 BF16 的范围和 FP16 的精度，是计算数据类型而不是存储数据类型。目前使用范围较小。

●**Float16 (FP16)** 。指数 5 位，尾数 10 位，符号 1 位。 FP16 数字的数值范围远低于 FP32，存在上溢 (当用于表示非常大的数时) 和下溢 (当用于表示非常小的数时) 的风险，fp16混合精度训练中通过缩放损失 (loss scaling) 来缓解这个问题。

●**Bfloat16 (BF16)**。指数 8 位 (与 FP32 相同)，尾数 7 位，符号 1 位。这意味着 BF16 可以保留与 FP32 相同的动态范围。但是相对于 FP16，损失了 3 位精度。因此，在使用 BF16 精度时，大数值绝对没有问题，但是精度会比 FP16 差。

●**(Un)signed Char (INT8)**。定点数 7位，符号 1位。用来表示 [-128, 127] 之间的整数。根据量化方式不同，有时也会转而用 UINT8来表示 [0, 255]之间的整数。

●**Unsigned Char (INT4 * 2)**。一个字节的 Unsigned Char可以表示（装入）两个 INT4的值。定点数数 3 位，符号 1位。用来表示 [-8, 7]之间的整数。

CNN量化 vs. LLM量化

LLM量化与CNN量化的对比主要有几点：

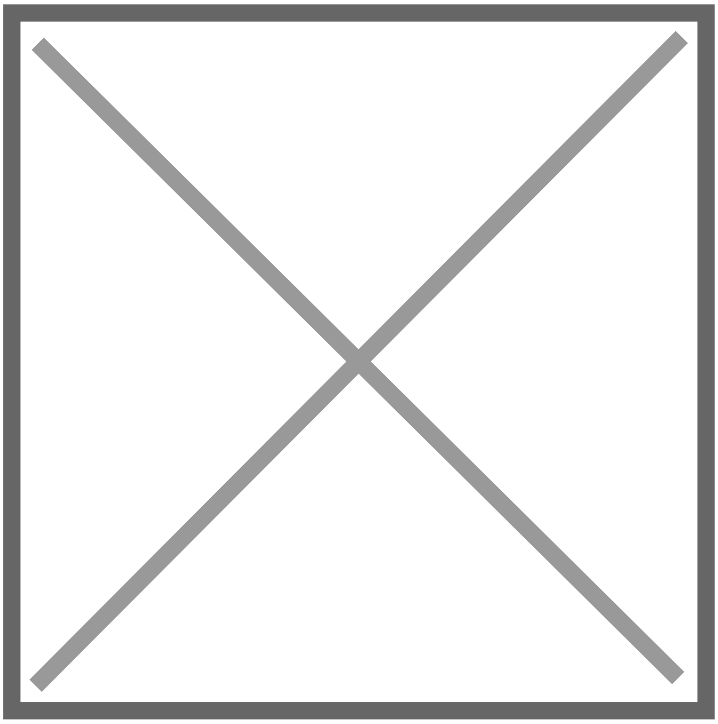
量化粒度：前面讲到CNN中我们常用per-channel或者per-layer的粒度来量化卷积kernel。但LLM中的主要运算是矩阵乘法，当进行矩阵乘法时，我们可以会采用vector粒度的量化(tensor的行或者列)，例如逐行或逐向量量化，来获取更精确的结果。对于矩阵乘法A*B=C，我们不会直接使用常规量化方式（per-tensor），而会找到A的每一行和B的每一列进行量化，然后进行整型INT计算，最后再将结果转回浮点结果（per-vector）。随着LLM的模型参数量越来越大，per-vector量化在LLM中的应用精度要求越来越高，逐行量化X和逐列量化W的方法已经没法满足误差要求，因此现在常见到的是将每行（每列）的FP16元素按照顺序每k个为一组（k一般是2的整数幂），k常见256和128为主，这也被称为per-group量化。

更低位量化***INT4***：CNN量化主流是INT8量化，基本上满足绝大多数小模型的推理性能和效果约束。LLM量化与CNN量化不同的另一个点是前者现在主流更加倾向使用更低精度量化，如INT4量化，这可以更进一步减少显存占用，做到在单卡可以跑13B模型。此外，更细粒度的量化需要更多scale参数，scale通常使用fp16或者fp32表示，因此这部分scale存储会占用越多空间，此时使用更低的INT4可以同步缩减内存占用。

数值分布特点：CNN的卷积层是由一个个卷积核组成，训练过程中由于梯度下降或者一些loss的约束，会使得每个卷积核的数值分布太不一样，这也是为什么使用per-channel方式分别对各个卷积核进行单独量化的原因，每个卷积核可以独享一个量化参数scale，可以减少量化误差。而在LLM训练过程中，由于普遍使用多头注意力机制的方法，因此训练过程每个head学到的表达不同，数值范围也不同，这种数值特点常常表现为激活tensor在hidden dim的列方向有相似性，而某些列之间数值差异反而非常大，比如有的列的数值范围在[50, 100]，有的列的数值范围在[0.5, 1.2]，插一句，最近有一种说法是LLM模型这种离群数值的特点本质原因是由于attention的softmax操作导致的（参考Attention Is Off By One），这种数值特点使得如果把矩阵tensor用per-tensor量化，会对效果产生很大影响，这也是许多LLM量化算法普遍采用更细粒度的量化方法的原因之一。

推理部署优化：CNN和LLM在量化后都需要配套的推理加速手段，比如使用cuda编写相应的INT卷积或者矩阵算子，将量化反量化融入计算算子中等等，硬件层面有些还会有专门的整型计算Core来提高计算吞吐。

要想量化误差低，取决与量化算法如何去挖掘数值分布的规律；要想量化后网络跑的快，基于微架构特性的工程优化（推理引擎优化）是必不可少的。



LLM量化工作

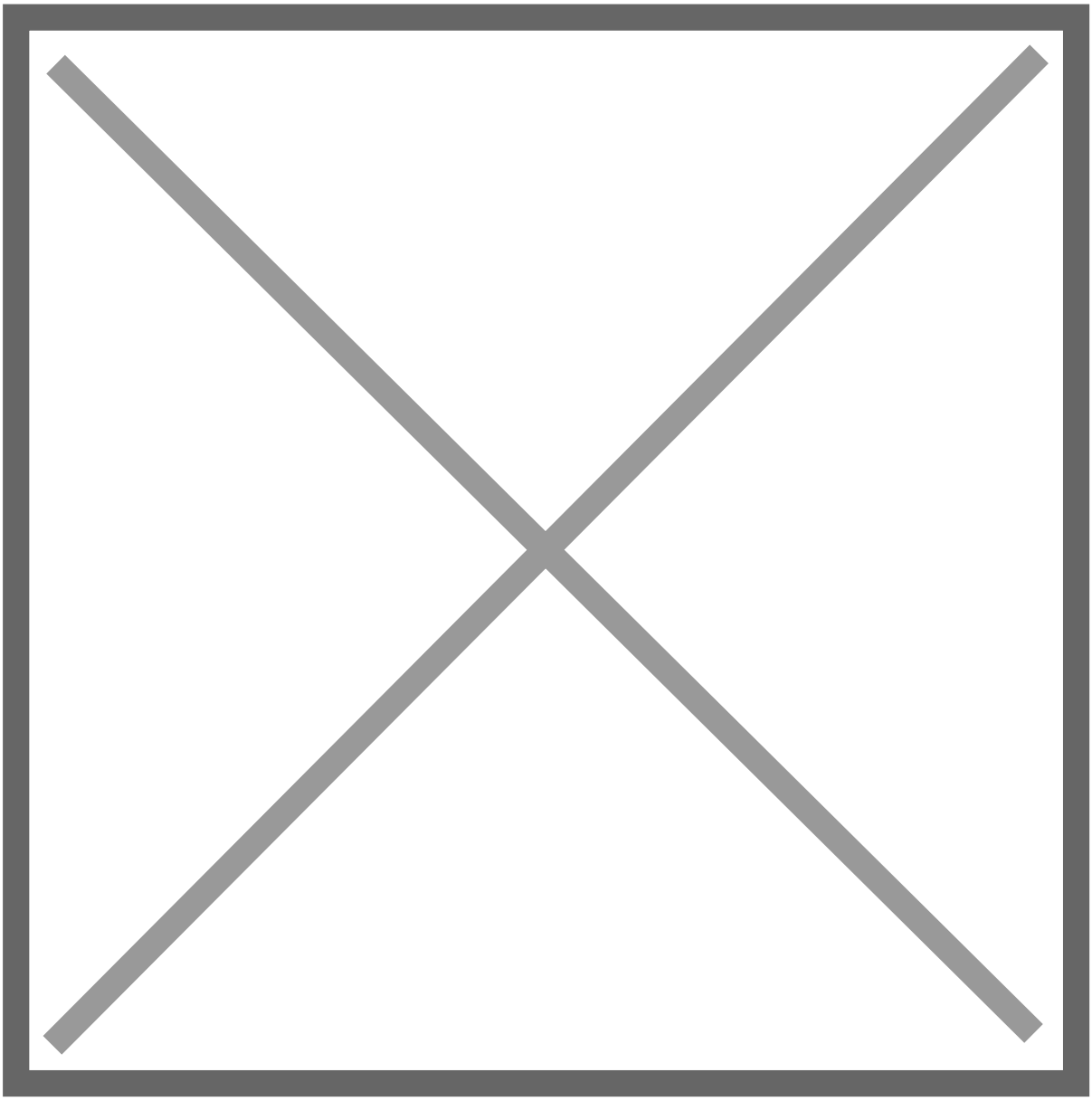
下面介绍当前几种比较常见的 LLM量化方案及其原理。

LLM.int8(): 混合精度量化

量化是将浮点范围映射到整型范围，而浮点异常值会对会使得量化后大量集中的浮点数被迫使用少量的 bit ，而异常值却生生浪费了大量的整型表示位，这也是 CNN一些 PTQ算法 （KL散度方式 ） 想找出合理的浮点数值截断范围的原因（本质是减少异常值的干扰，减少整型数位的浪费）。与 CNN 不同的是，如上述 LLM的 tensor 某些列有的表现非常大，实验发现这些异常值对最后的模型效果影响是比较大的，直接舍弃是不行的。于是实验中又统计了这些异常值的分布情况，发现了一些比较有意思的现象：

随着模型的增大（尤其是大于 6B之后），异常值的分布显著增加，几乎所有的 layer 里都有异常值，并且横跨了所有的 sequence length 。一个是异常值数量比较多并且比较大，和其他值进行传统的 quantization 方法，对精度损失影响是比较大的。另一个是它们分布的非常规整，基本是在某几个 hidden size 的维度出现，但是横跨多层和一个 sequence中所有的 token。

基于这些结论， paper提出了一个混合量化的方案：

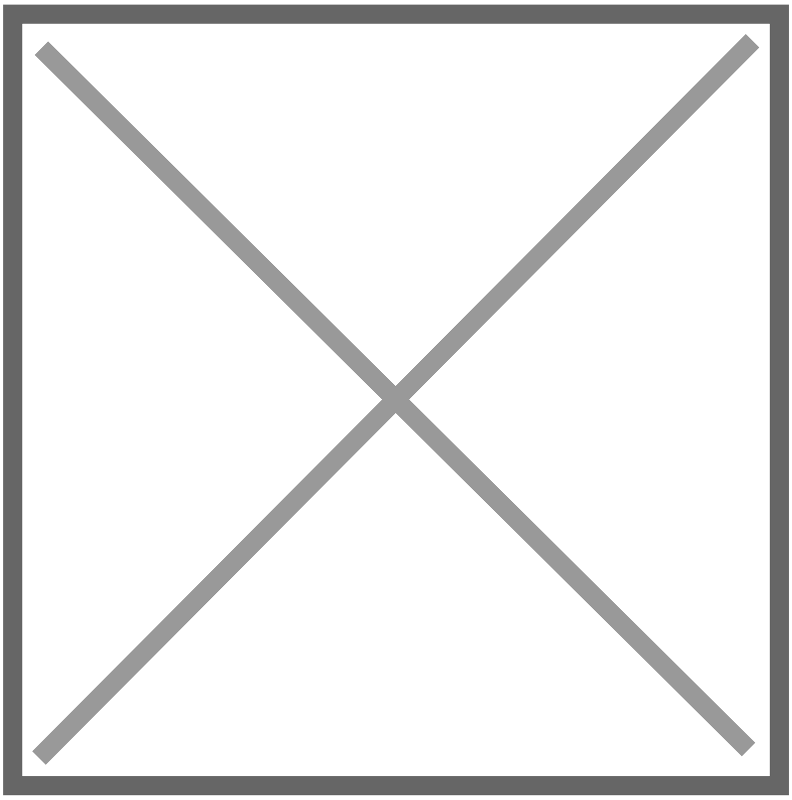


X activations sequence length hidden-size

X W int8 fp16 **(W token) element-wise concat**

LLM.INT8() gemm X W threshold 6 6

LLM.int8() LLM.int8() BLOOM-176B FP16 15% 23% (T5-3B T5-11B)

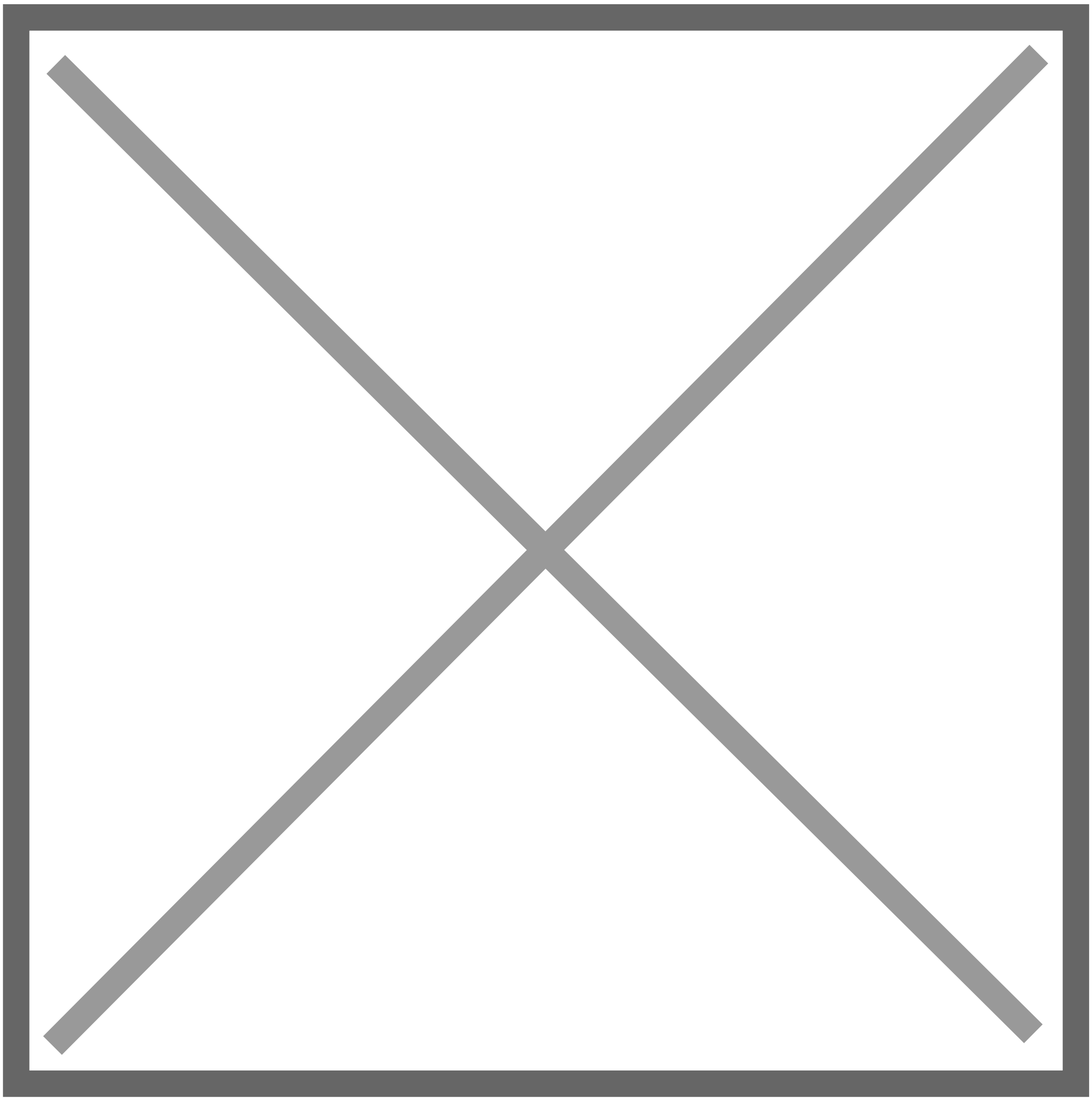


的降速幅度更大。这非常好理解，这种 LLM量化方法只是离线量化了权重，因此运行时当推到某一个 block 时候，再把这个 block的权重反量化成 fp16权重，与 fp16 的激活进行矩阵运算，对于参数量小的网络，反量化的开销无法被计算完全掩盖，因此反而推理速度是下降的。

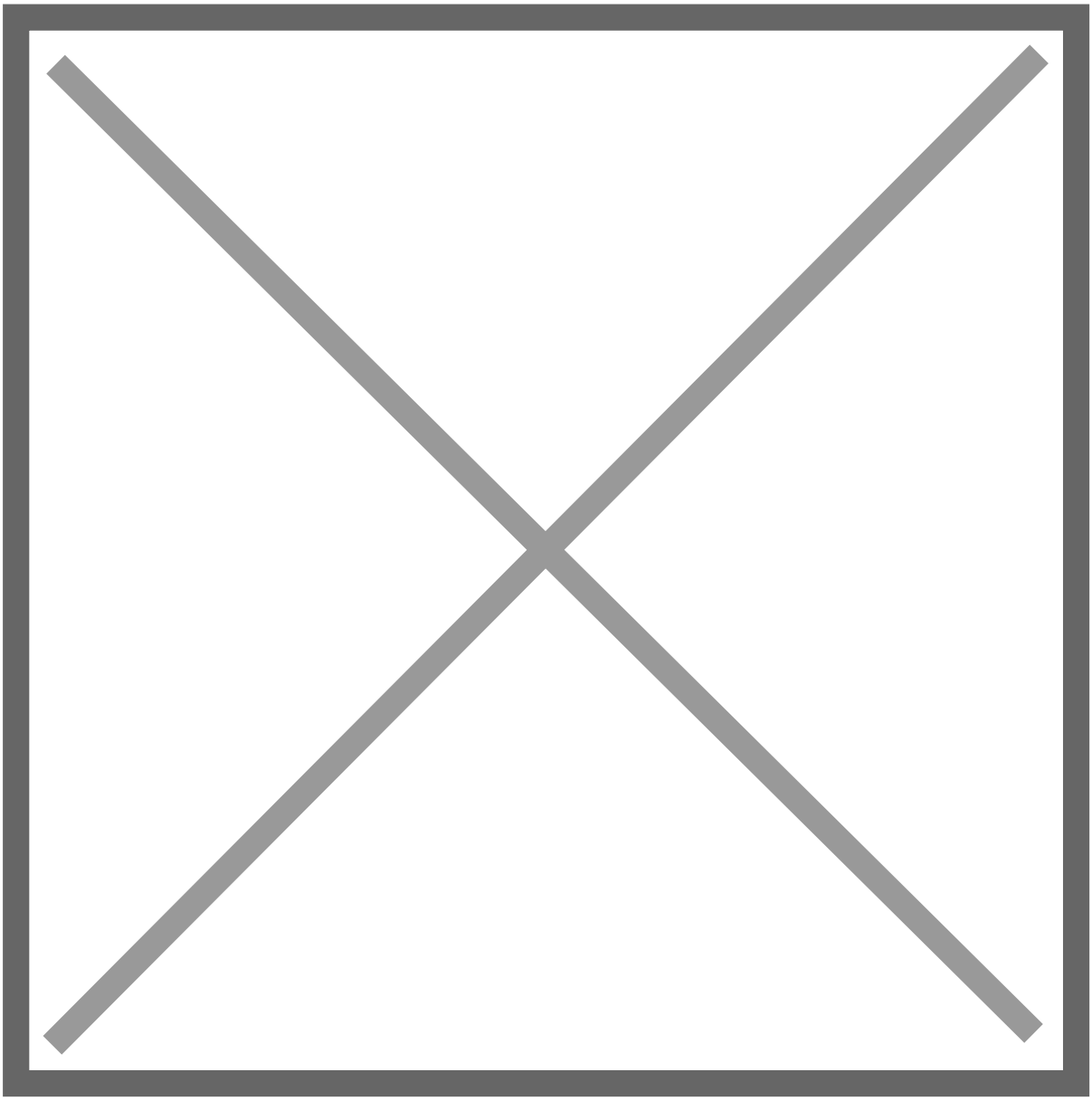
GPTQ：基于优化方法

GPTQ的原理来自于另一个量化方法 OBQ，GPTQ可以说是工程优化版本。OBQ实际上是对 OBS的魔改 (一种比较经典的剪枝方法，论文名字比较有趣，叫最优脑损伤手术）。在 OBS 中，作者希望找到一种模型剪枝方法，假设我们要抹去一个权重记为 W_q，使得其对整体的误差增加最少，并且同时计算出一个补偿 δ_q 应用于剩余的权重上，使得抹去的这个权重增加的误差被抵消：

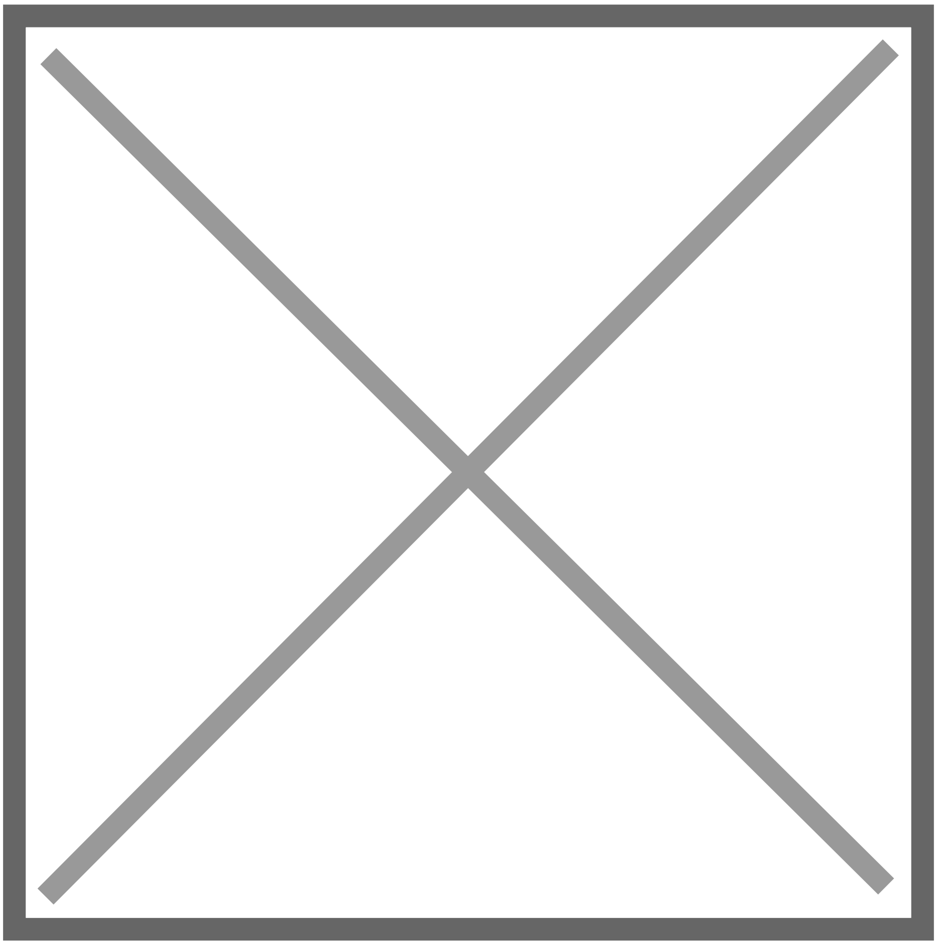
其中 H是一个代表 loss对权重的海森矩阵，OBQ把 OBS 的思想推广到量化中。其实很好理解，我们常用的量化则是把数值近似到一个接近的值，而剪枝实际上可以看做把数值直接近似成 0（某种意义上或许可以称作 1bit或 0bit 量化），可以理解为一种特殊的量化。OBQ里的公式是这样的：



其中 quant(q)是把 q近似到最近的 quant grid的点上，简单说就是四舍五入到指定位数上。可以看到假如 quant(q)永远返回零其实就是原版的 OBS。可以看到 OBQ的量化算法是从最优化的角度去操作的，理论基础足。 OBS 的计算过程如下图所示，整理来说，将权重分行计算，采用贪心算法（逐个找影响最小的 q来剪枝 /量化）：



GPTQ OBS



-相同顺序：在量化的过程中，以贪心算法来每次以最小的误差量化权重的方法表现良好，但相比固定顺序并没有明显提升，尤其是在大模型上可能固定顺序更好。所以使用固定顺序进行量化。

-并行计算：因为量化权重在行间没有影响，所以我们可以并行量化多行来加速计算。

-分批更新：如果我们每次量化一个参数就进行一次权重更新的话，更多的时间都被花到访存，不能充分利用GPU的算力，所以我们可以分批进行更新，批内完成后再更新全局的矩阵。

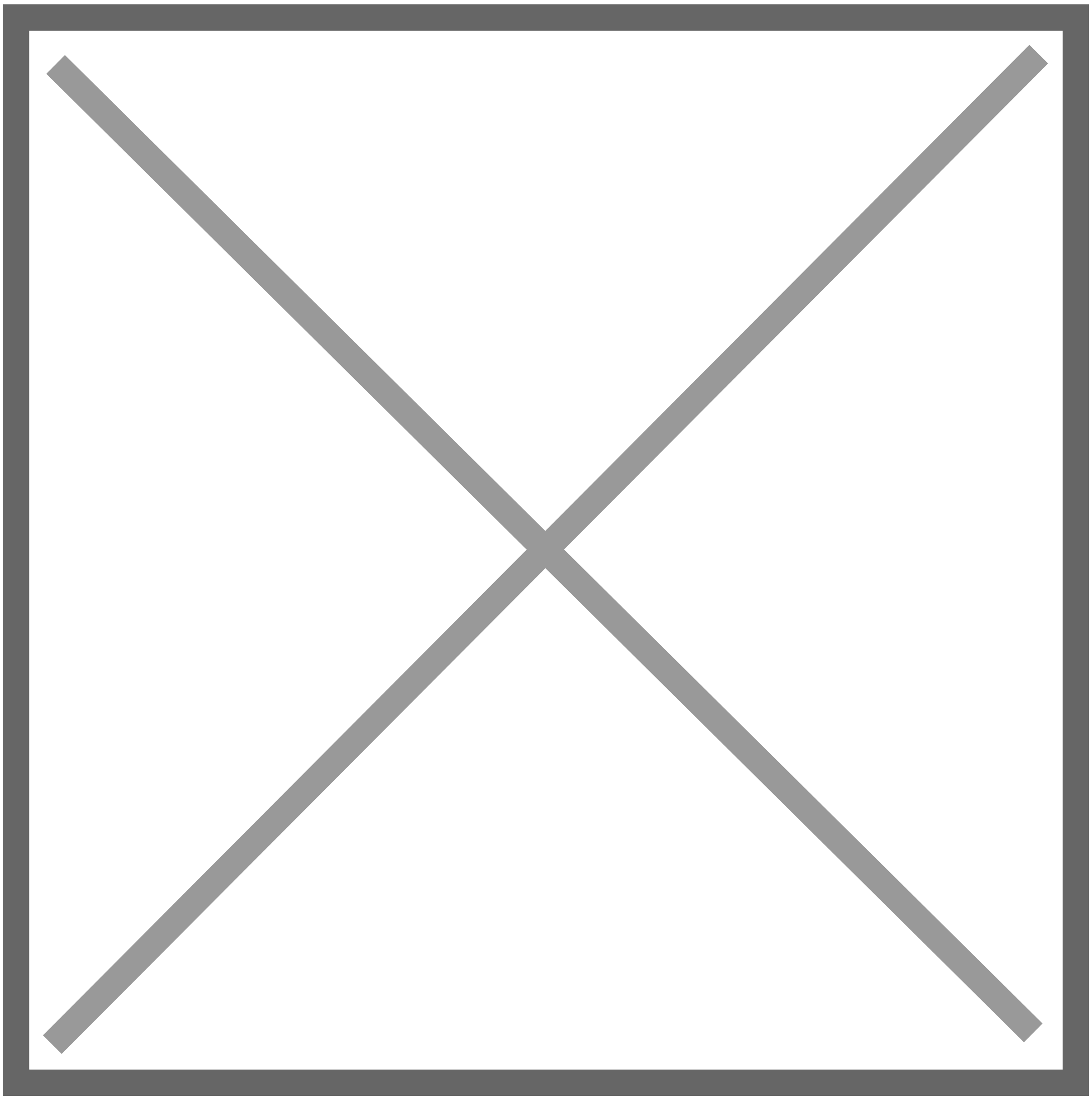
-数值稳定：通过求海森矩阵的逆来增强数值稳定性。

-分组量化：相对于采用相同的Scale，Zero对整个矩阵进行量化，选择一个小点的Group size在组内计算特定的量化参数可以应对全局的异常值，取得更好的量化效果。

从实际使用来看，GPTQ这种基于优化的量化方法的量化误差是要比其他LLM的量化算法的误差要低，是一种比较靠谱的量化方法。

SmoothQuant: W8A8与参数 scale

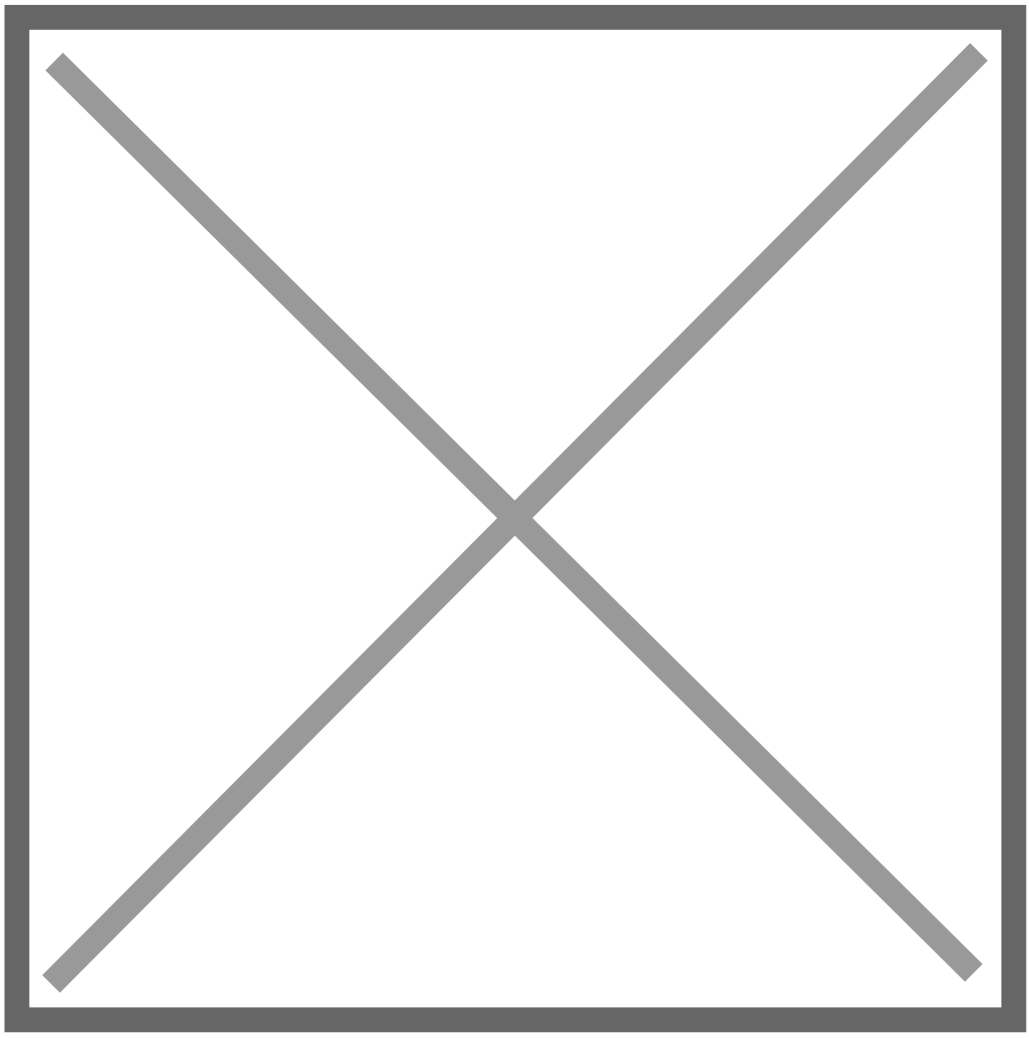
背景在上面也有说过，LLM的某些channel或某些维度的activation outlier值很多且很大(e.g. GLM-130B有30%)，导致量化的有效位变少，比如int8本来是-128到 127，没有outlier的时候，映射到-128到 127



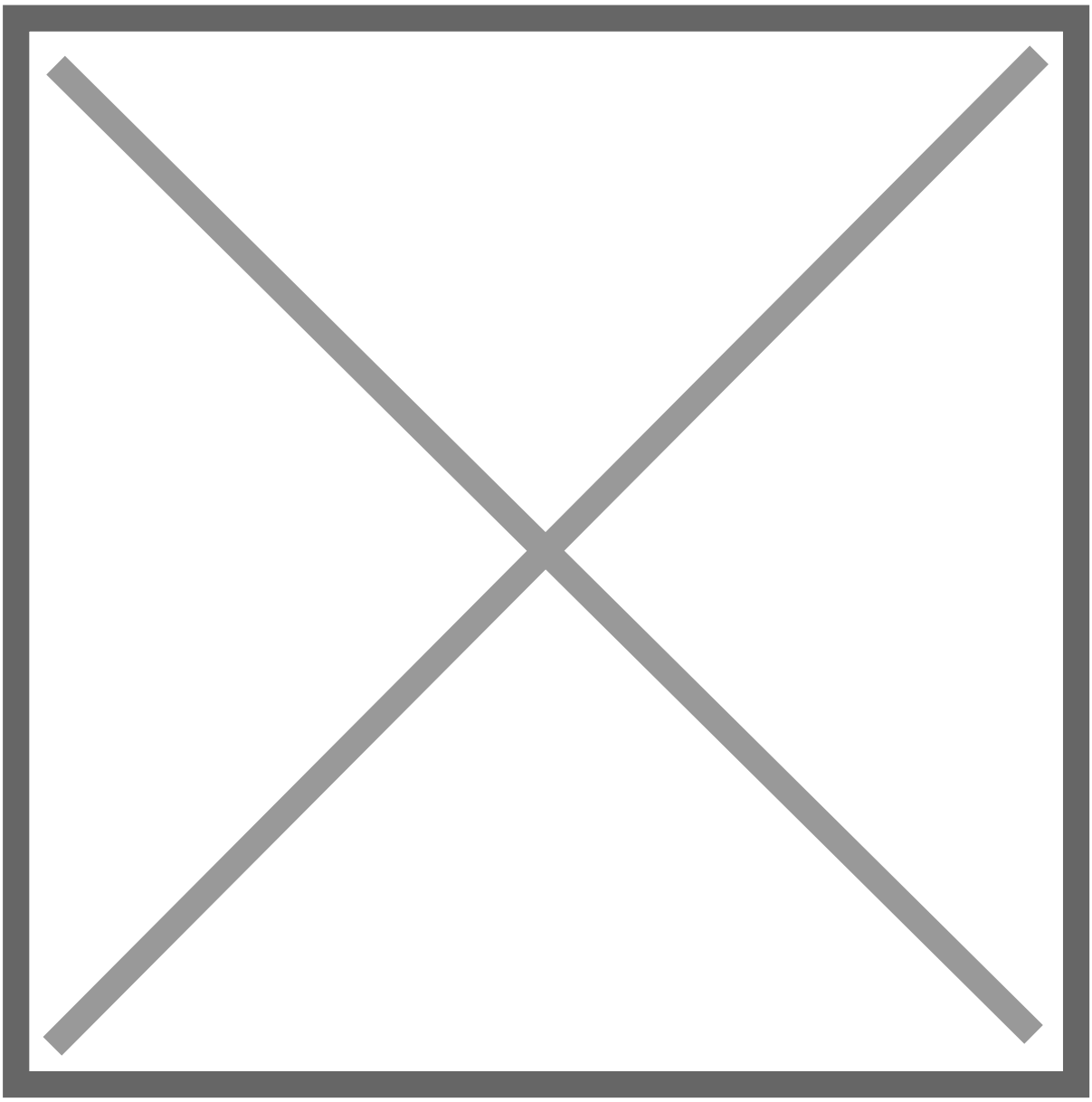
的数据分布均匀，占满了8bit位范围，精度损失很低，但是有了outlier之后，多数正常值的分布区间可能在[-20,20]或者[-10,10]，8bit位范围只利用到了5bit，甚至4bit，由此导致精度损失。

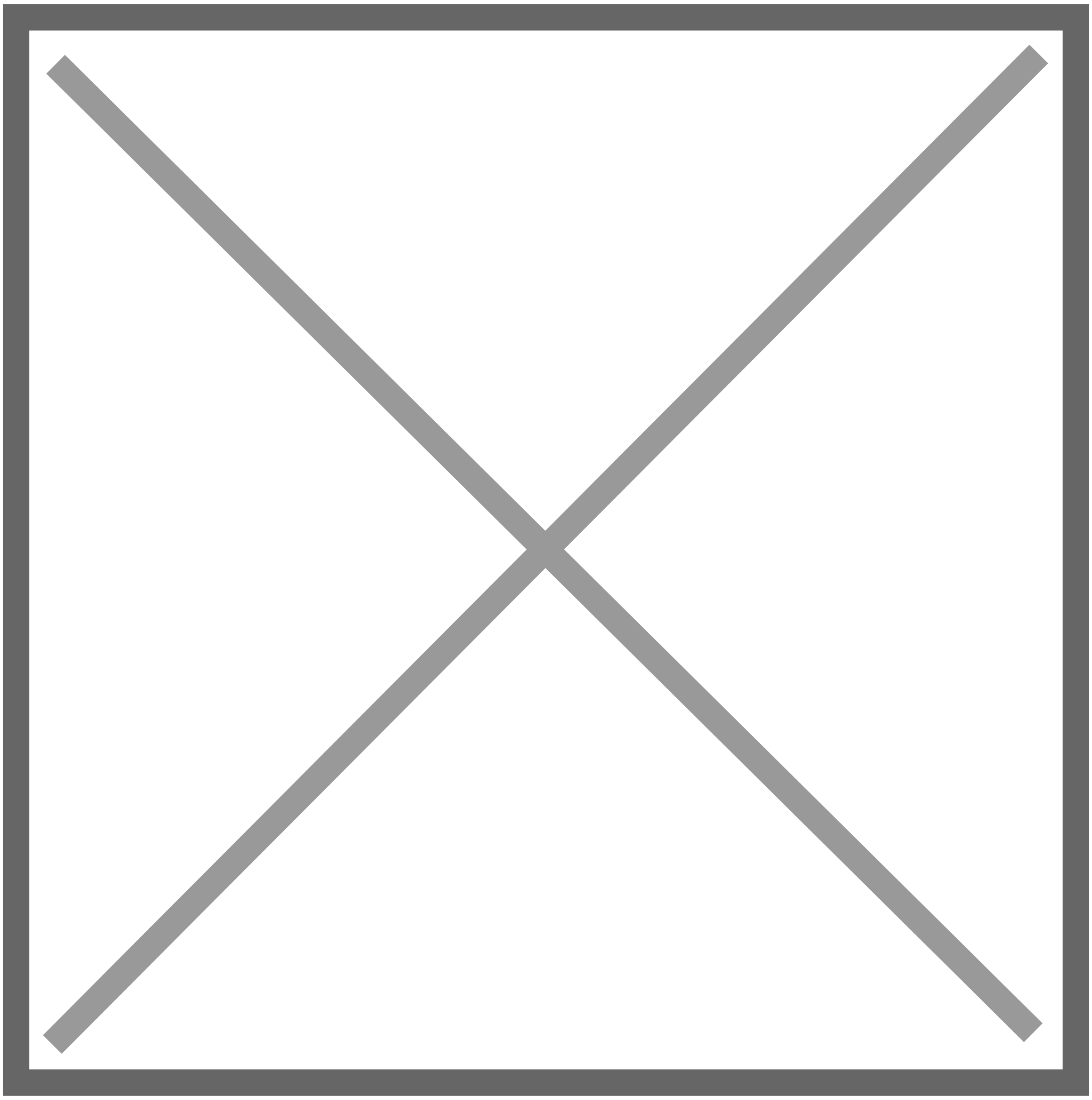
论文基于这样假设：激活值比权重更难量化。权重的分布相对更加均匀和平坦，之前的研究结果已经证明将大型语言模型的权重降低到INT8，甚至到INT4对准确率的影响都不大，但鲜有做的好的激活值量化方案。既然**weight**比**activation**更难量化，所以可以把**activation**量化难度迁移到**weight**上来。

SQ把activation里面不均匀的分布用weight中和一下，具体来讲，主要是在fp32阶段做的，保证一个等式的等价，X为输入act，W为weight，s为因子，通过s来中和。



中和之后的效果如下图所示：

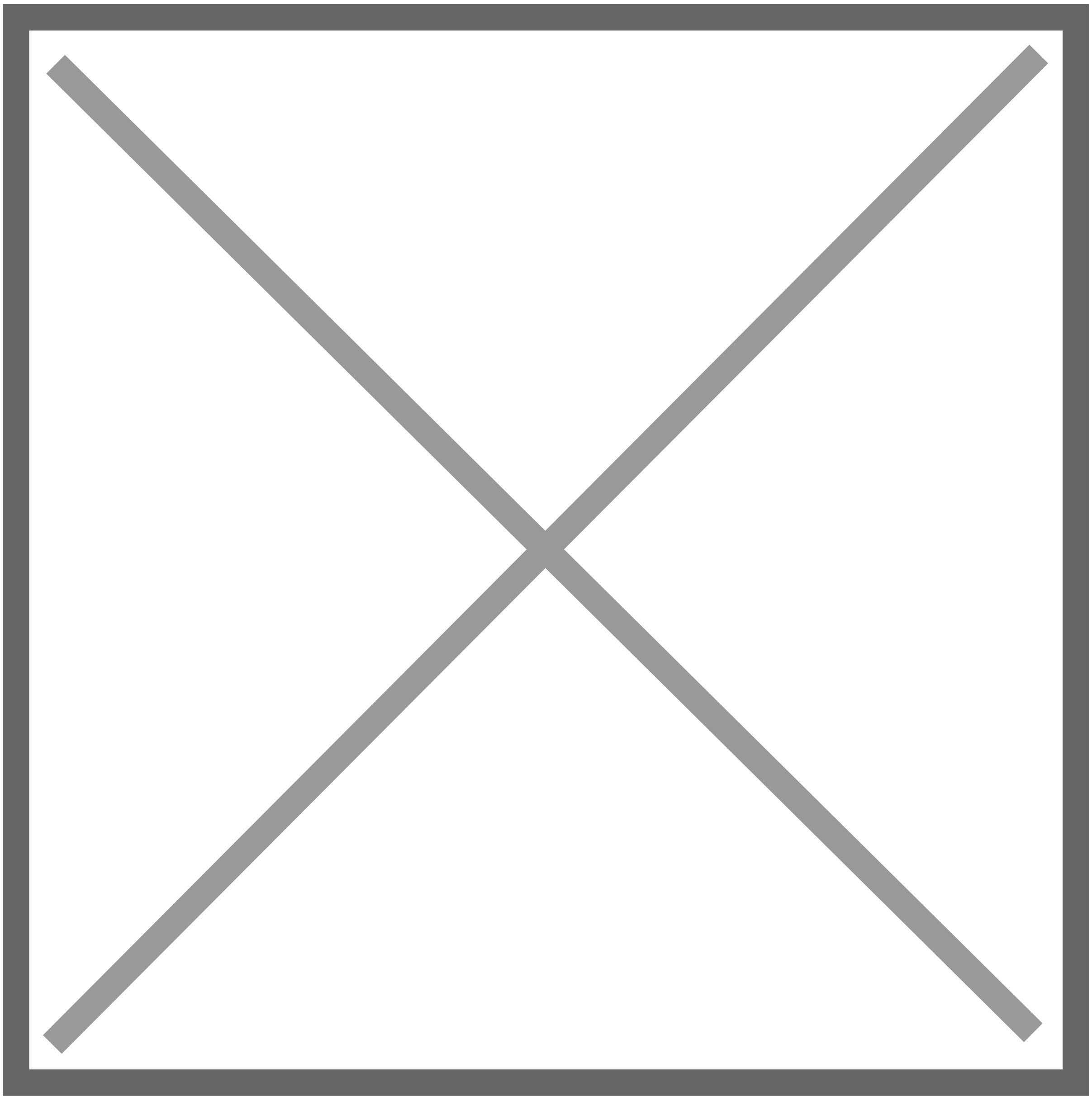




大部分LLM量化都是weight-only的实现，而SmoothQuant提出了一种对LLM的激活值和权重同时进行量化的方案，并且考虑了激活值中的离群值对量化精度的影响，巧妙的转移了这种离群值引起的范围波动。由于引入了激活值量化。SmoothQuant的方案只适用于W8A8，不能实现更低比特的量化，因为INT4时，量化bit的表示范围已经很粗粒度了，权重量化已经会造成一定的精度下降，再将激活的误差转移到权重会造成更严重精度下降。

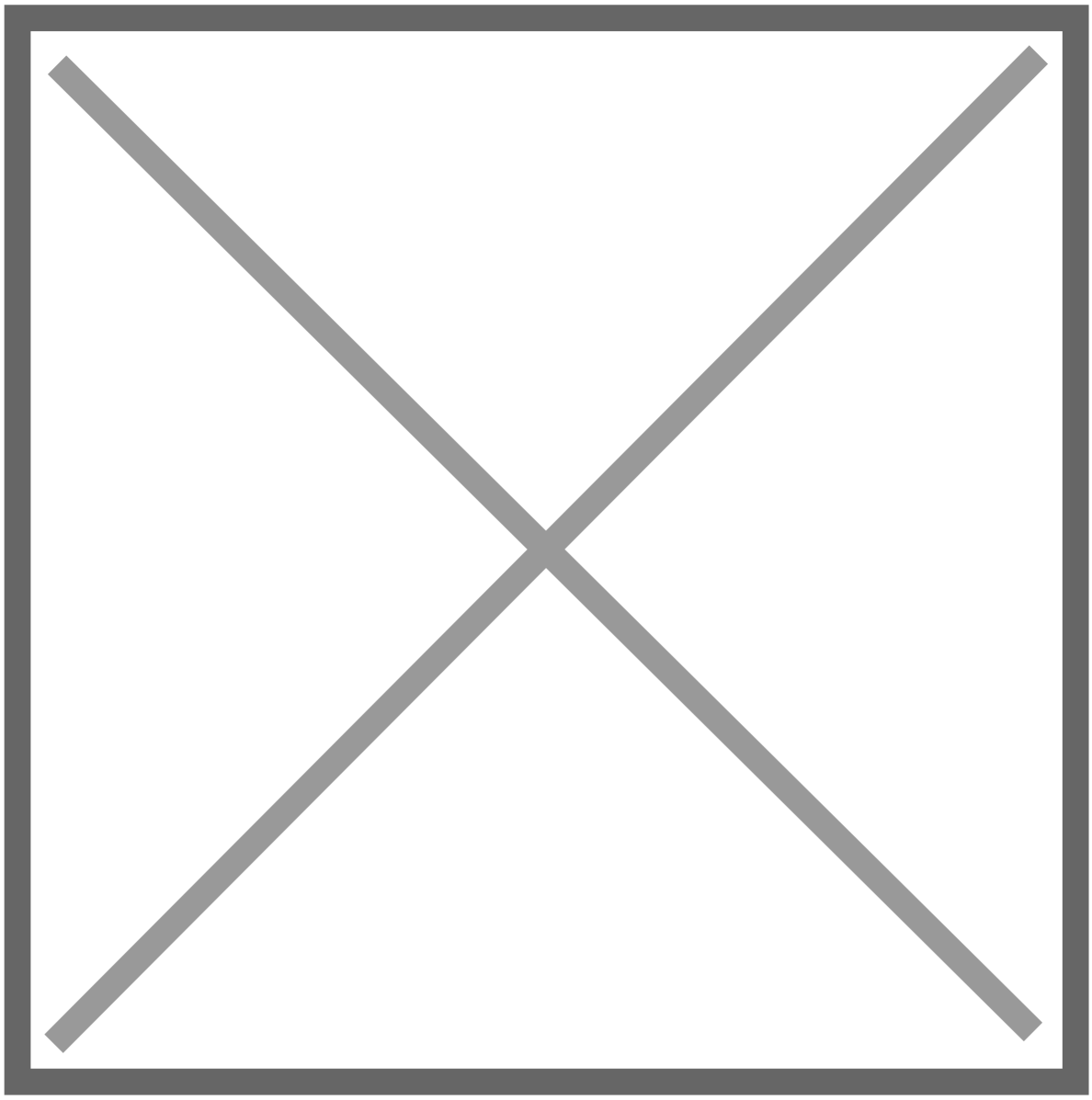
实际使用来看，SmoothQuant这种量化方法思想是非常直观的，也是**LLM**量化方案里面比较少见的既量化权重又量化激活的方案，但是目前对于这种方案并没有比weight-only量化方案表现出优势，因此SmoothQuant使用不算太多，据笔者所知，TensorRT-LLM（TensorRT的LLM推理解决方案）针对GPT2用的便是SmoothQuant。

AWQ: 基于激活感知的参数缩放量化

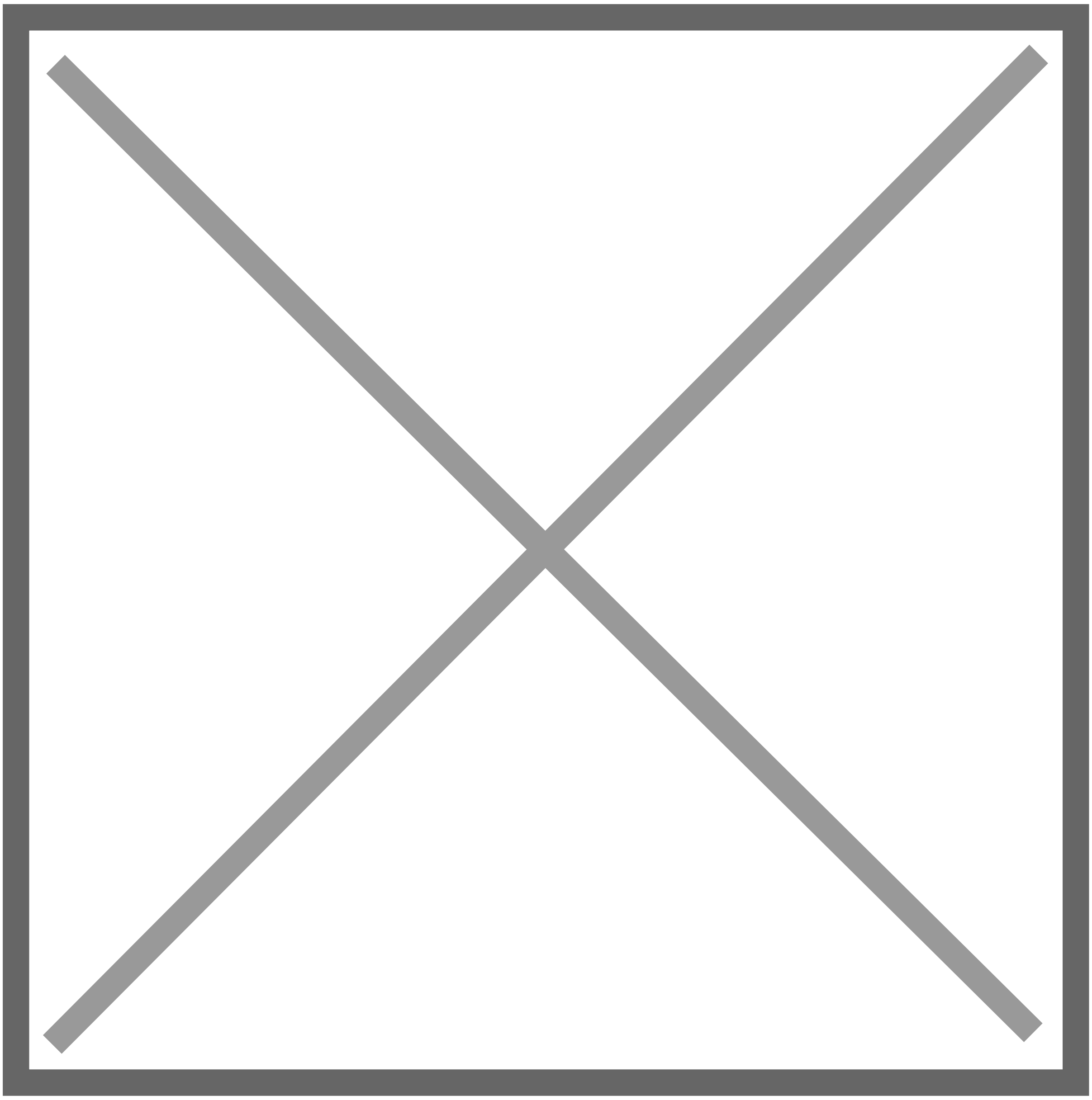


论文动机基于这样的一个现象：直接使用最简单的量化方式误差很大(a)，但是只需要找到大约1%左右的channel的参数，使它们保持fp16不量化，而量化剩下的参数，量化误差极大减少(b)，但是这种混合精度量化(int8+fp16)在实际推理中硬件不友好，感觉需要类似LLM.int8()那种 tensor 并行的执行方式，会使得推理速度大幅下降。因此论文提出基于激活感知的参数缩放量化方法(c)：通过分析激活分布来确定对应的那***1%***对量化误差影响大的参数 ***group***，对这些参数进行缩放来改变参数分布（和 ***SmoothQuant*** 的思想类似），使其更加量化友好，之后再对整体参数进行量化。

算法目标是对某一层寻找一个缩放比例 s，在参数量化之前乘以这个比例，计算时输入 X 除以这个比例，使得误差最小：



而论文中缩放系数 S，由下面的两部分决定:

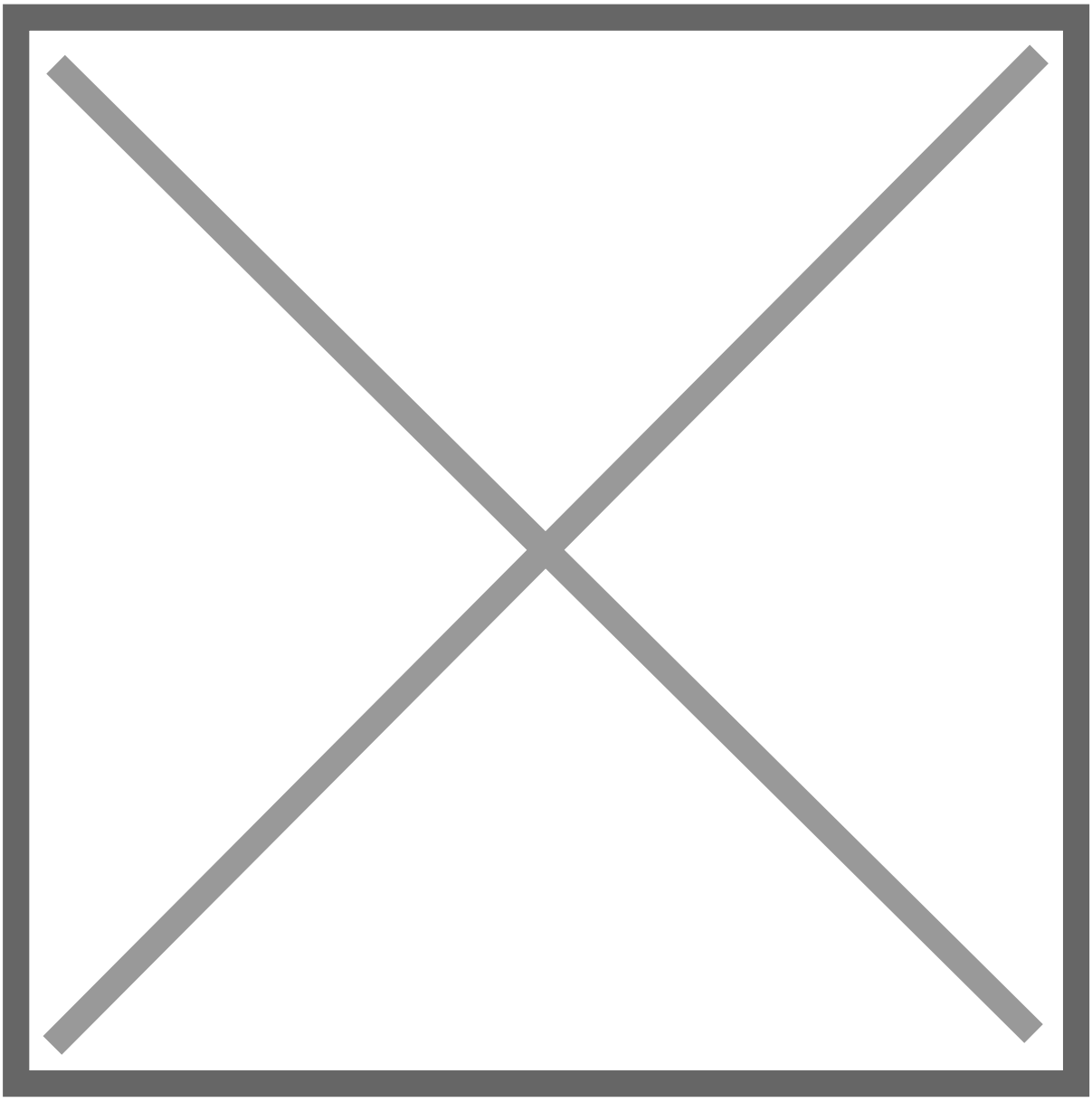


其中 $sX = meanc\_out|X|$，$sW = meanc\_out|W|$，可以看出 sX是对应激活 group 的平均值，以此作为衡量对应 weight参数 group的重要指示。 α 和 β是在 [0, 1] 中之间通过网格搜索来确定最合适的值。实际实验发现 sW 对于最终结果影响并不大，主要的影响来自激活系数，这也侧面说明，激活值中的异常 group和 weight 作用后对最终量化结果影响巨大，只有保护好对应参数不被这部分激活影响，才能减少量化误差。

工程实现上， AWQ通过手写 kernel方式，利用好 矩阵乘 以及 连续访存 的的优势，实现对推理的优化。

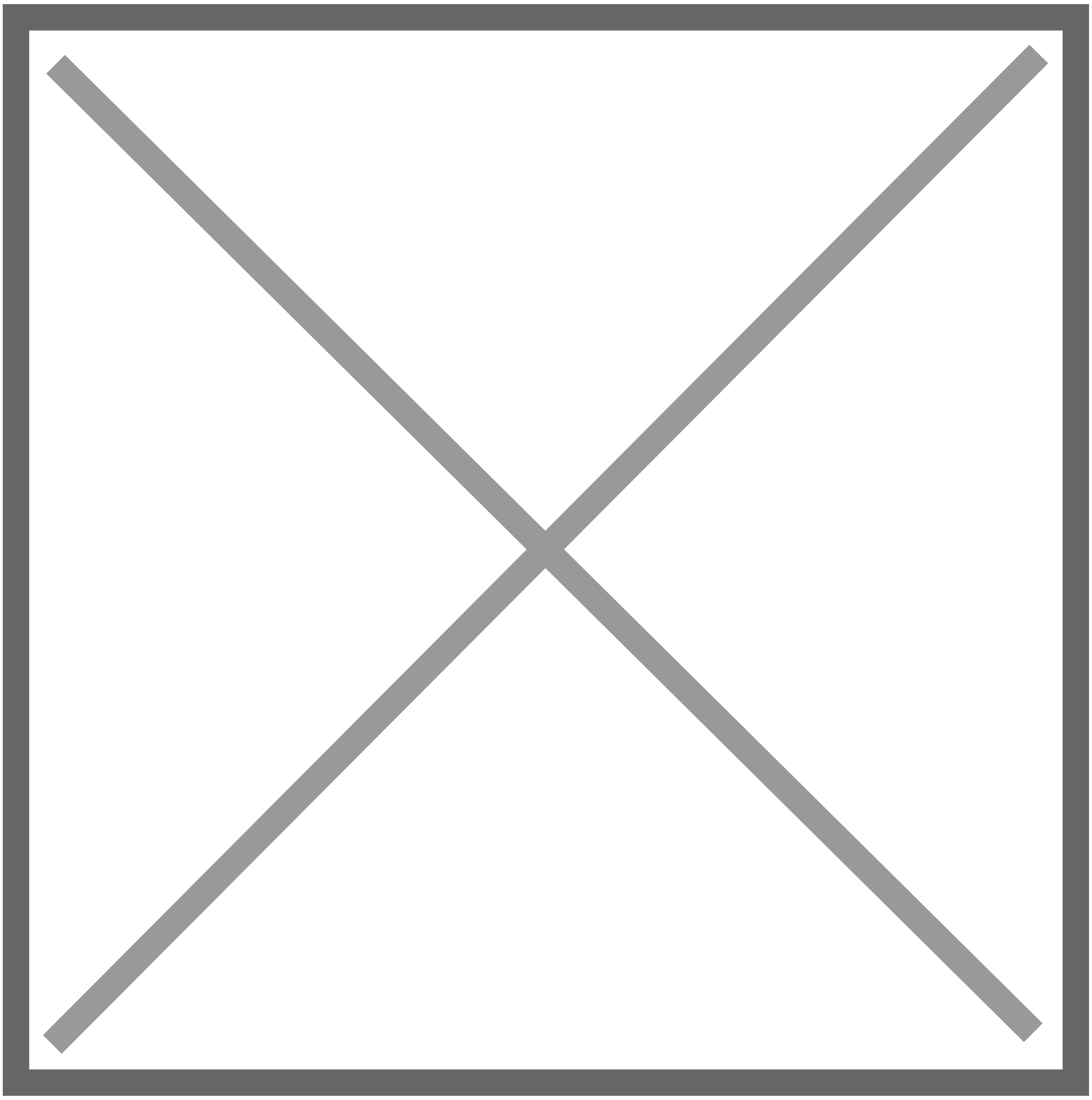
虽然在原理上和 smooth quant类似， AWQ本质上是 weight-only的量化算法，但是 SQ算法是 W8A8量化。从实测来看， AWQ的量化效果比 GPTQ的效果要好，推理也更快，但是推理更快大概率是因为更精细的手写 kernel技巧所造成的。

Llama. CPP: K-quant量化



LLaMA.cpp项目是开发者Georgi Gerganov基于Meta释出的LLaMA模型实现的纯C/C++版本（基于其另一个项目：tensor库GGML），用于模型推理。llama.cpp的初衷是让开发者在没有GPU的条件下也能运行Meta的LLaMA模型。后来在他本人和社区开发者的支持下，也支持了cuBLAS的加速。在6月15日的PR中，llama.cpp正式支持了CUDA库的加速，用大量的cuda kernel加速了llama在GPU的推理速度。

llama.cpp中量化部分主要采用了两种方案：INT4量化和K-quant量化。量化选项如下图所示，其中Q后面的第一个数字x表示了量化到x bit，下划线后的第二个数字y为0时表示对称量化，没有零点。y为1时表示非对称量化，每个scale还有一个zero point，当y为K时表示K-quant量化方法，K后面的字母表示量化模型的参数规模：Small, Meduim，Large。例如，Q4_0和Q8_0选项代表进行INT4、INT8量化。



当 y=K时，表示使用了K-quant方法。K-quant量化使用了16 x 8的“块”进行量化，每个“块”共有16个行。每8个权重为一组使用同一个量化参数scale，因此有16个一级量化参数。此外，为了进一步的降低资源消耗，还有1个fp16的二级量化参数K_2，用于量化16个一级量化参数，相当于“量化参数的量化”，这可以进一步减小模型size和显存消耗。当使用16*8的“块”K-quant时，每个一级量化参数K_1负责的group size = 8，但65B模型只需要37.5GB显存。可以看到，对于参数量越大的模型，使用粒度更细的量化，可以有效守住模型推理效果。

实践了**llama.cpp**后发现使用了**K-quant**量化后进行模型推理，比**fp16**原模型进行推理快了**3~4**倍（对比**token/s)**，这个不太符合前面说到的结论，权重量化的方案会比fp16推理慢，因为有反量化的开销。研究后发现，llama.cpp使用c++重新写了许多加速的cuda算子，其中包括将反量化+gemm实现成复合算子，减少两步操作的全局显存数据搬移开销。另外由于现在GPT推理普遍使用KV Cache优化，因此在推理计算吐出token的过程中，实际上是vector-matrix运算（GEVM?），而不是matrix-matrix的GEMM操作，因此llama.cpp也针对向量矩阵乘做了相应加速的实现。

因此本质上来看，llama.cpp实际是一个llama模型的推理引擎实现**+K-quant**量化算法，这回到最开始讲的，要想量化后还能加速，本质上就是访存和计算的加速可以掩盖掉反量化操作带来的

开销，因此 llama.cpp给出的方案是实现一个基础的推理引擎，并且底层自己实现一些针对性的复合算子 Kernel来进行加速。从实验来看，目前 llama.cpp量化的推理速度还是十分快的。

QLora：量化有效参数微调

Lora和 Qlora实际上是有效参数训练的方法，因此并不属于量化推理的范畴，但是 QLora使用了量化技术来支持更大模型的有效参数微调。关于 lora的机制已经有许多介绍的文章了，简而言之可以这样考虑，微调实际上是得到一个在原来基础参数上的

，形状大小和原参数是一样的，但是假如我们可以把它分解成

，A矩阵和 B矩阵的参数量可以原小于 W。LoRA的用意就是在冻结原模型的情况下训练 AB两个矩阵而不是整个 W，这样所需的计算量会大大减小，而在推理时候是将 AB融入到原始权重中，因此推理时候实际是没有增加额外参数的。

QLora使用的量化技术如下:

Block-wise Quantization 一批一批的量化，每一批（block)使用独立的 scale。在本文中 block的大小被设为 64,这种类似 K-quant量化。

Quantile Quantization 所有浮点数字由小到大排列，再分成 十六等分，最小的一块映射到量化后的第一个数，第二块映射到量化后的第二个数，以此类推。这样做原始数据在量化后的数字上分布就是均匀的。这样的做法就是 Quantile Quantization. 这里的 Quantile意思是分位，我们平常说的中位数，四分之一位数，都是一种分位。

双重量化 在 block-wise quantization当中我们提到，每一个 block有自己的一个常量 scale，这其实是个额外的负担，假如使用 float32, block大小为 64，那么摊到每个参数上的额外开销就是 32/64 = 0.5bit。对于 4bit量化来说额外的 0.5bit相当于多 12.5%的显存耗用，还是比较可观的。因此提出了双重量化，就是对 scale再进行一次量化，考虑到一般出现 outliner的概率较小，使用 256作为 block大小再量化一次得到 8bit的 c改进过后每个参数的额外消耗为 8/64 + 32/256 = 0.127 bit。

QLora 主要是训练前对权重进行量化，训练时每推到推到一层的时候，将这一层权重反量化回去进行正常推理计算，通过这种方式可以在单卡上对更大参数量的 LLM进行有效参数微调。

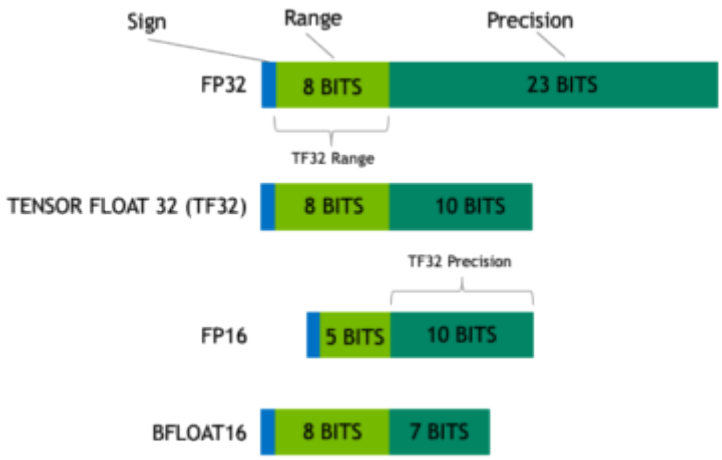
总结

- 当前主流的 LLM量化方案主要是只量化权重（weight-only）。
- LLM的量化粒度会比 CNN更加细（per-vector级别 +分组），并且会使用双重量化进一步量化一批次 scale值。

- 只量化权重在运行时需要反量化操作，会比fp16推理慢，但是可以通过优化算子加速来覆盖这部分开销，从而享受到INT量化带来的大幅显存下降同时又有加速效果（来自Llama.cpp的启发）。
- LLM中，INT4量化+更加细粒度量化，要比INT8量化有更高的性价比，模型参数越大，INT4量化优势越大（体现在省显存和推理速度上）。

Data type

data type


Sign
Range
Precision
FP32
8 BITS
23 BITS
TF32 Range
TENSOR FLOAT 32 (TF32)
8 BITS
10 BITS
TF32 Precision
FP16
5 BITS
10 BITS
BFLOAT16
8 BITS
7 BITS