机器学习系统 2026春

第八章 模型量化

张燕咏 讲席教授 yanyongz@ustc.edu.cn
张午阳 特任教授 wuyangz@ustc.edu.cn

中国科学技术大学
University of Science and Technology of China

第一节课

量化概览与代表方法

建立量化的操作直觉，理解四个 PTQ 名字背后的三个核心思想

本节课路线图: 量化概览与代表方法

主线问题: FP16 → INT4，模型为什么还能用？

本节课要做的

- 建立量化的**直觉**：为什么、是什么、怎么做
- 串起 PTQ 主线四个代表方法
- 理解 QAT 低比特路线 (BitNet b1.58)
- 看清硬件如何吃掉低精度算力

本节课不讲

- 详细数学推导 (放第三节课)
- 算法伪代码细节 (放第三节课)
- 在服务器上完整执行 (放第二节课)
- 选型决策树与误差拆解 (放第三节课)

L1 = 概念与直觉; L2 = 实战; L3 = 数学与细节。本节先建立量化的整体框架。

从剪枝到量化: 两种压缩范式

上一章压缩**哪些参数留下**，本章压缩**每个参数用几位**

剪枝：决定哪些参数留下

0	-0.31	0.23	0.28	-0.59	-0.39
0	0	0	-0.26	0.26	0.23
0	0.34	0	-0.26	0	-0.29
0.26	0	0	-0.20	0.37	0
0	0	0	0	0	0
0.64	0	0	-0.24	0	0.34

留下 17 / 36 个权重，每个仍 FP16 (16 bit)

量化：每个参数用更少 bit

8	4	10	12	0	3
8	6	8	4	10	10
8	12	10	4	8	4
10	8	6	4	12	6
6	6	10	8	10	10
15	6	6	4	10	12

保留全部 36 个权重，每个用 4 bit (16 个等级)

剪枝 (Pruning)

- 部分参数被置零 (50% 通用做法)
- 留下的参数仍是 FP16 / 16 bit
- 节省: 计算 FLOPs + 部分访存

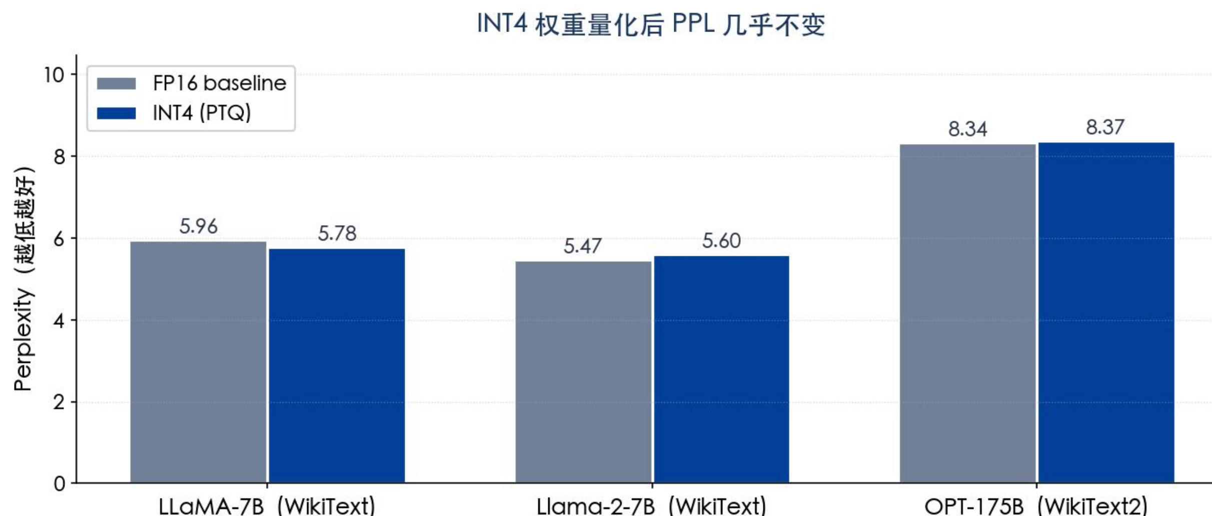
量化 (Quantization)

- 全部参数都保留
- 每个参数用更少 bit (INT8 / INT4)
- 节省: 显存 + 访存 + 低精度 Tensor Core 算力

两种压缩正交: 剪枝 + 量化可以叠加 (例如 GPTQ + 2:4 sparsity)。

驱动问题: FP16 → INT4，为什么模型还能用？

先看实证: 多篇论文报告 INT4 几乎不掉 PPL



来源：LLaMA / Llama-2: AWQ Lin et al. 2024, Table 4 | OPT-175B: GPTQ Frantar et al. 2023, Table 5

为什么 INT4 仍然够用 (本节给直觉，第三节课讲数学)

- LLM 权重分布**集中在 0 附近**，少量 bit 就能表达大部分信息
- 训练时的 SGD 噪声 + dropout 已让模型对**小幅扰动**鲁棒，量化噪声相当于**轻度正则**
- INT4 偶尔 PPL **略低于** FP16: 量化引入的噪声 + WikiText PPL 测量本身有 ±0.2 波动
- 关键: 处理好少数**离群权重 / 离群激活** (后面四个方法的核心)

本节课要回答: 上面这条曲线是如何实现的，对应 4 个代表方法各自的策略。

PART A

为什么要量化

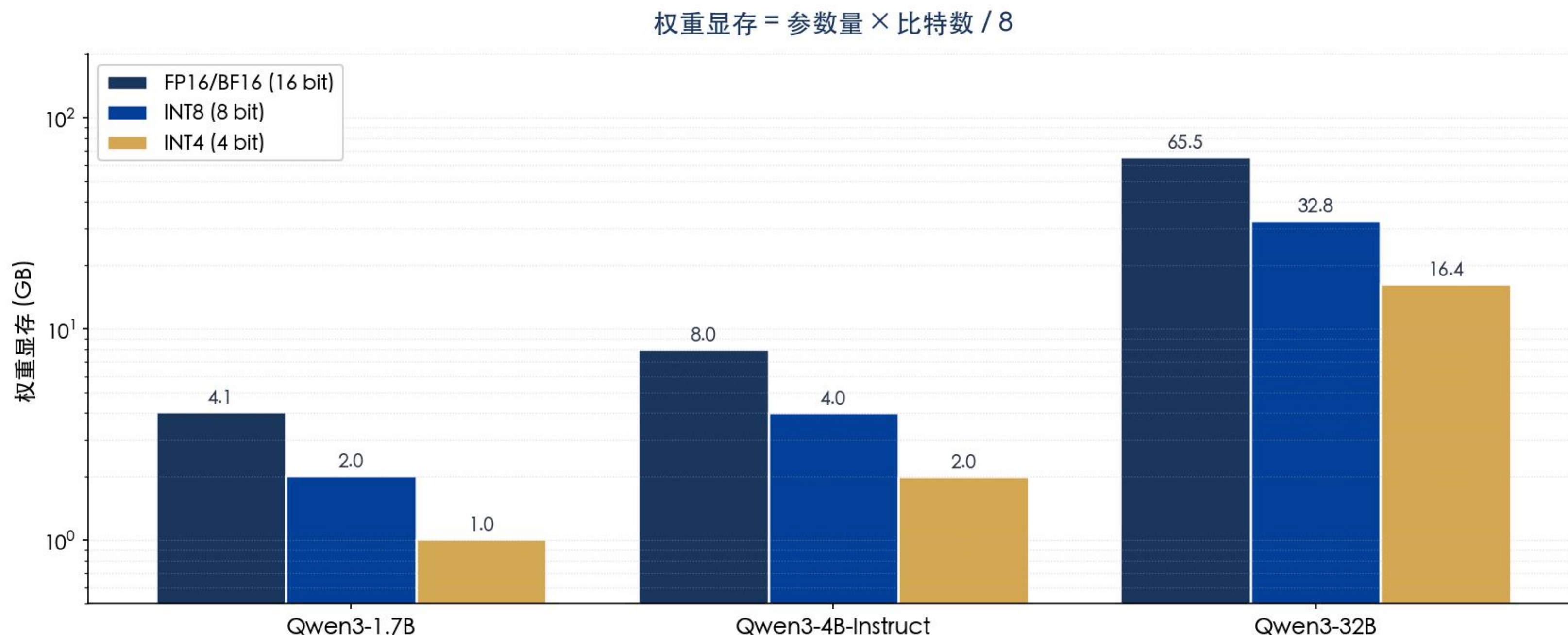
动机两条线: 显存计算 + 算力计算

本课进度

Part A	Part B	Part C	Part D	Part E
为什么要量化	量化是怎么工作的	三种核心思路	QAT 低比特	硬件视角

显存计算: 模型权重最低多少 GB

显存 (权重) = 参数量 × 比特数 / 8: 量化的最直接收益



实测数字 (Qwen3 系列)

- Qwen3-1.7B: **4.1 GB** → **1.0 GB** (FP16 → INT4)
- Qwen3-4B: **8.0 GB** → **2.0 GB**
- Qwen3-32B: **65.5 GB** → **16.4 GB** (单 24GB 卡可载)

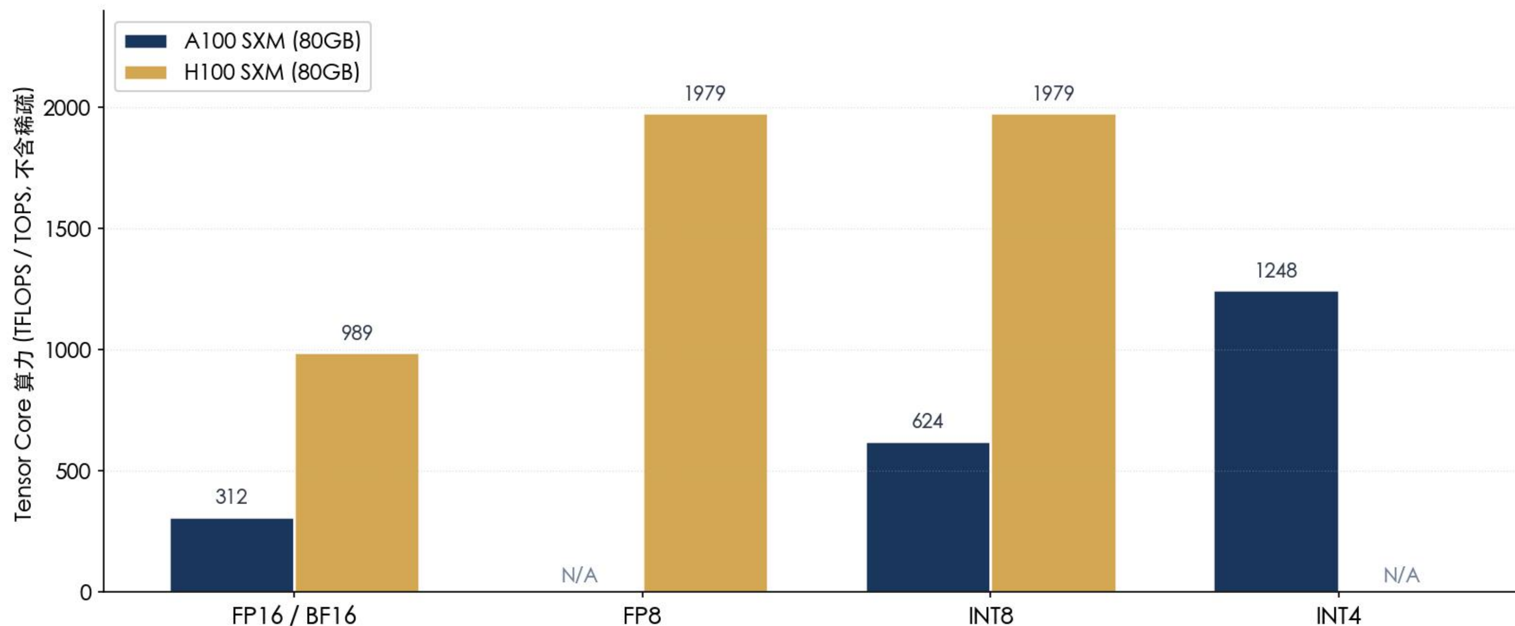
只是权重，还有大头没算

- KV cache 在长上下文下**比权重还大**
- 例: 540B PaLM, batch=512, ctx=2048 → KV ≈ 3 TB (KIVI, ICML 2024, arxiv:2402.02750, §1)

权重大小: /data/models/Qwen3- safetensors 实测; KV cache: KIVI 论文引用 Sheng et al. 2023*

算力计算: 低精度 = Tensor Core 翻倍

比特数减半 → 同一颗 GPU 的 Tensor Core 算力翻倍



读这张图的三个要点

- A100: FP16=312 → INT8=624 → INT4=1248 TFLOPS/TOPS (每降一半 bit, 算力翻倍)
- H100: 引入 **FP8** Tensor Core, 989 → 1979 TFLOPS
- 这是**硬件提供的免费加速**: 只要算子能用 INT8/FP8, 速度直接 2× / 6×

NVIDIA A100 / H100 Tensor Core GPU 官方 datasheet (公开白皮书)

PART B

量化是什么

数轴直觉 + 核心公式 + 量化对象

本课进度

√ Part A	Part B	Part C	Part D	Part E
为什么要量化	量化是怎么工作的	三种核心思路	QAT 低比特	硬件视角

量化的视觉直觉: 数轴上的分辨率

比特数 b 决定分辨率: 同一区间被切成 2^b 个等级

FP16 (≈ 65000 个等级)

几乎连续

INT8 (256 个等级)

粗一点点

INT4 (16 个等级)

明显粗

数值范围（同一区间，比特数下降 → 等级数指数级下降）

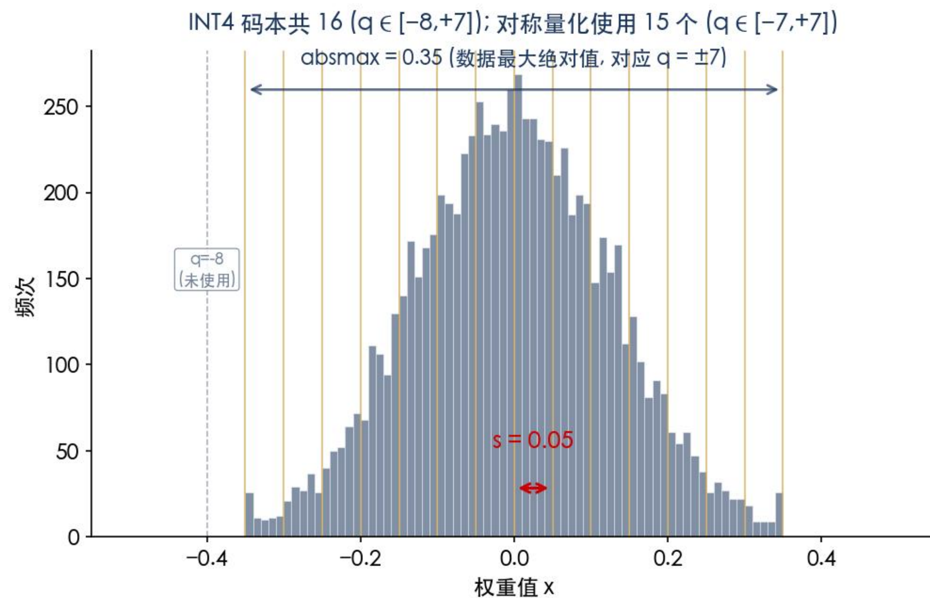
三种比特数的等级数对比

- **FP16**: 约 65000 个等级 (动态范围广，几乎连续)
- **INT8**: 256 个等级 (够大多数神经网络用)
- **INT4**: 16 个等级 (粗，但只要分布集中也能用)

量化的本质 = 把**实数轴**离散化到**有限的格点**上。

核心公式: q = round(x / s) + z

把数轴翻译成公式: 上一页的格点位置 = s · q + z, 反推得到 q



$$q = \mathrm{round}(x/s) + z$$

s 怎么定? 对称量化 (z=0):

$s = \text{absmax} / (2^{b-1} - 1)$

INT4 (b=4): s = 0.35 / 7 = 0.05

s : 步长 (每个等级有多大)　z : 零点偏移
x : 原始浮点　q : 量化后整数 ∈ [−7, +7]

示例 (s = 0.05, z = 0):
x = +0.32 → q = round(+6.4) = +6 → x' = +0.30
x = −0.27 → q = round(−5.4) = −5 → x' = −0.25

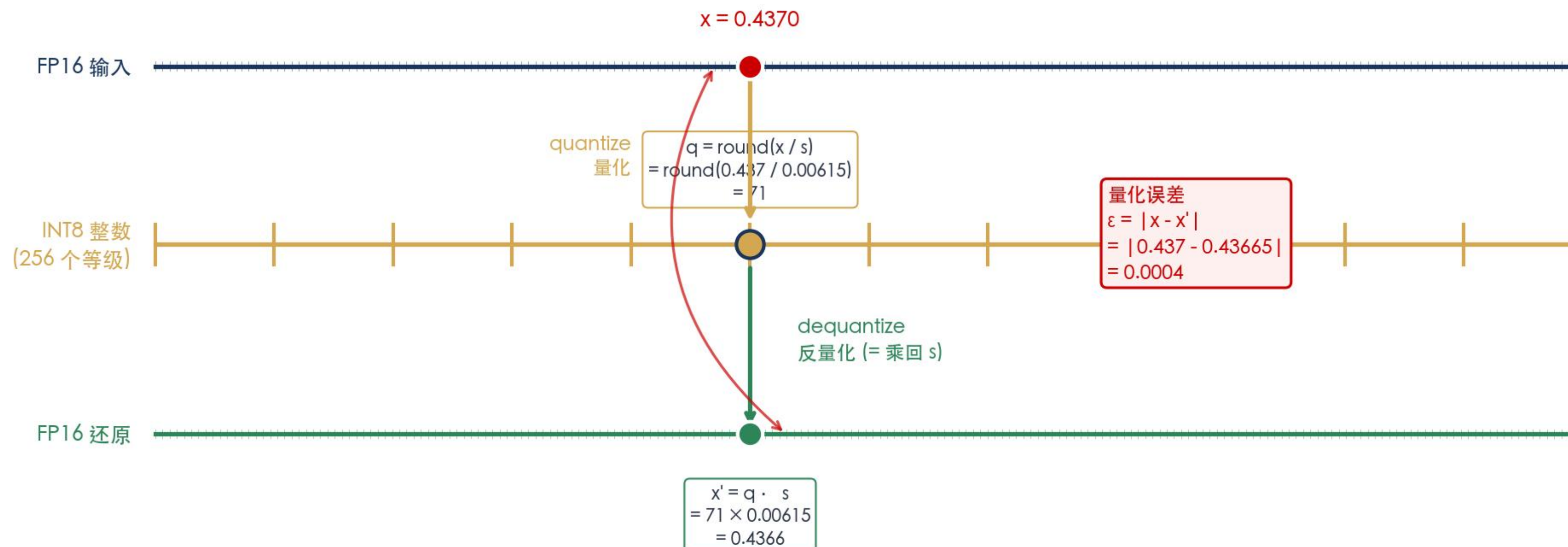
两个参数 s, z 决定一切; 实操中怎么定?

- **s (scale, 步长)** = 相邻格点间距; **z (zero-point)** = 整数 0 对应的浮点偏移
- 先从校准数据收集权重 / 激活的范围: **absmax = max(|x|)** 或 [α, β]
- **对称量化** (z=0, 适合权重): **s = absmax / (2^(b-1) − 1)**, INT4 → s = absmax/7
- **非对称量化** (z≠0, 适合激活): s = (β−α) / (2^b − 1), z = round(−α/s)
- 为什么不用满 2^b 个码? 对称量化弃用 −2^(b-1) 这一码, 让 ±absmax 精确还原 (PyTorch / GPTQ / AWQ 标准)

对称/非对称量化是工业界标准做法; 详细对比见第三节课 Part A

量化的三个基本概念: 量化 / 反量化 / 量化误差

核心循环: 浮点 x → 整数 q → 近似浮点 x'，误差 ε = |x - x'|



量化 = 把 FP16 浮点 "对齐" 到 INT 等级网格; 反量化 = 还原回浮点 (乘回 s); 误差 = 对齐时丢的精度

量化 (quantize)

- 浮点 → 整数: q = round(x/s) + z
- 离线一次完成 (PTQ)
- 推理时存盘 / 加载用整数

反量化 (dequantize)

- 整数 → 近似浮点: x' = (q-z) · s
- 推理时**实际计算前**做一次
- 或与 GEMM 融合 (低精度 kernel)

量化误差 (error)

- ε = |x - x'|, 由 round 引入
- 比特数越少，ε 越大
- 后面 4 个实例围绕"如何控制 ε"

实例 1: 8 个权重做 INT8 量化

先建立操作直觉: 拿真实尺度的 8 个权重值，对称量化全过程

INT8 对称量化的完整流程

Step 1 原始 FP16 权重 (8 个值)

Step 2 absmax = 0.781 → s = absmax / 127 = 0.006150

Step 3 q = round(x / s) (得到 INT8 整数)

Step 4 反量化 x' = q · s，误差 |x - x'|

平均误差 = 0.0013 最大误差 = 0.0023 (相对 absmax: 0.30%)

示例数据为占位 (synthetic)；后续将用 Qwen3-1.7B 实测权重替换

读完这张图你应该能回答

- 为什么 absmax 决定 scale？ 最大值刚好映射到 INT8 边界 (±127)
- 为什么 INT8 误差这么小？ 256 个等级，平均误差约 0.001 量级

实例 2: 同样 8 个值做 INT4，对比误差

比特数从 8 降到 4: scale 变粗，误差放大约一个量级

同样 8 个权重: INT8 vs INT4 误差对比

原始 x:

+0.437	-0.213	+0.053	+0.781	-0.554	+0.124	-0.346	+0.625

INT8 q (s=0.00615):

q=+71 x'=+0.437	q=-35 x'=-0.215	q=+9 x'=+0.055	q=+127 x'=+0.781	q=-90 x'=-0.553	q=+20 x'=+0.123	q=-56 x'=-0.344	q=+102 x'=+0.627

INT4 q (s=0.1116):

q=+4 x'=+0.446	q=-2 x'=-0.223	q=+0 x'=+0.000	q=+7 x'=+0.781	q=-5 x'=-0.558	q=+1 x'=+0.112	q=-3 x'=-0.335	q=+6 x'=+0.669

误差 |x-x'|:

INT8: 0.0004	INT8: 0.0022	INT8: 0.0023	INT8: 0.0000	INT8: 0.0005	INT8: 0.0010	INT8: 0.0016	INT8: 0.0023
INT4: 0.0093	INT4: 0.0101	INT4: 0.0530	INT4: 0.0000	INT4: 0.0039	INT4: 0.0124	INT4: 0.0113	INT4: 0.0444

INT8 平均误差 0.0013, 最大 0.0023 vs INT4 平均误差 0.0181, 最大 0.0530 → INT4 误差约 13.9× 大

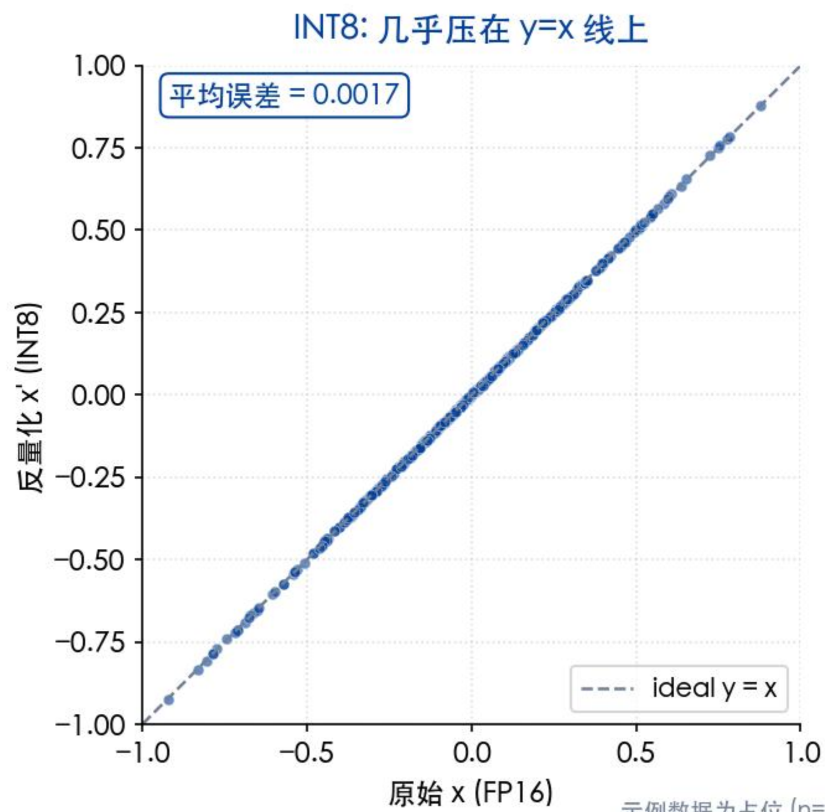
示例数据为占位 (synthetic); 后续将用 Qwen3-1.7B 实测权重替换

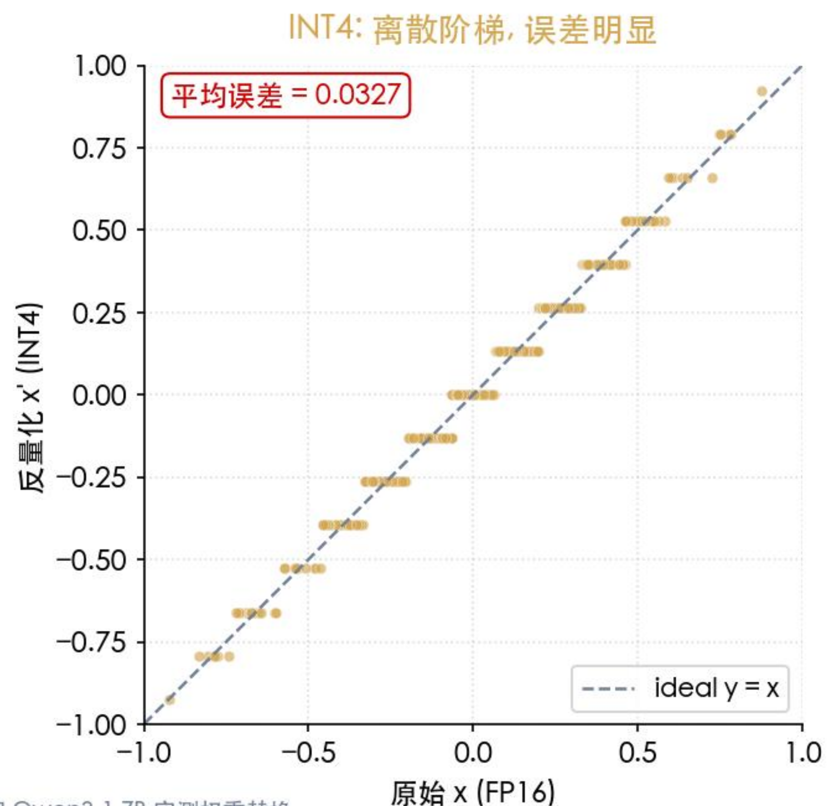
关键观察

- 第 3 个值 0.053 在 INT4 下被 round 到 0，**完全丢失**
- INT4 平均误差比 INT8 大约 **14 倍**：这是 8→4 bit 的代价

量化误差直观对比: 原始 vs 反量化散点

理想情况是 y = x, 量化后偏离这条线就是误差





示例数据为占位 (n=400, σ=0.3 synthetic); 后续将用 Qwen3-1.7B 实测权重替换

读图要点

- INT8: 散点紧贴 y=x 对角线，肉眼几乎看不出偏差
- INT4: 明显的**阶梯状离散**，每个权重对齐到 16 个等级中的一个
- 阶梯之间的水平距离 = scale s，决定了最坏单点误差

对称 vs 非对称量化

对称: z=0, 实现最简单 / **非对称**: z≠0, 范围利用率更高

对称量化 (适合权重)

- 假设分布**对零均值近似对称**
- z = 0, s = max(|α|, |β|) / (2^(b-1) - 1)
- 量化: q = round(x / s)
- 优点: 实现简单, kernel 优化容易

非对称量化 (适合激活)

- 处理**单侧分布** (例如 ReLU 后总 ⩾ 0)
- z = round(-α / s), s = (β - α) / (2^b - 1)
- 量化: q = round(x / s) + z
- 优点: 范围利用率 100%, 量化误差更小

工程上的常见组合

- **权重**: 通常用对称量化 (神经网络权重通常零均值)
- **激活**: 在 ReLU / SiLU 之后通常用非对称 (单侧, z 不为零)
- LLM 实际多用 **per-channel 对称权重 + per-token 非对称激活**
- 比特数同理: 权重多用 INT4, 激活更难压, 通常 INT8 起步

实例 3: per-tensor vs per-channel 在矩阵上的差异

当某列幅值偏大: per-tensor 一刀切让其他列陪跑, per-channel 各自缩放

per-tensor INT4 (一个 s)
平均误差 = 0.0515

+0.09 err 0.086	-0.11 err 0.090	+0.53 err 0.081	-0.73 err 0.089	+0.13 err 0.072	-0.03 err 0.029
-0.07 err 0.071	+0.21 err 0.009	-0.36 err 0.044	+0.52 err 0.096	-0.02 err 0.023	+0.35
+0.02 err 0.018	+0.18 err 0.025	-0.37 err 0.035	+0.70 err 0.086	+0.19 err 0.010	-0.13 err 0.078
-0.01 err 0.015	+0.15 err 0.058	-0.44 err 0.033	-1.43 err 0.000	-0.02 err 0.019	-0.31 err [illegible]
-0.14 err 0.062	-0.01 err [illegible]	-0.13 err 0.071	+0.81 err 0.002	+0.01 err [illegible]	+0.27 err 0.062
+0.32 err [illegible]	-0.08 err 0.079	-0.17 err 0.038	-0.06 err 0.058	+0.10 err 0.102	+0.05 err 0.050

s = 0.2040 (覆盖整个矩阵 absmax)

per-channel INT4 (每列一个 s)
平均误差 = 0.0169

+0.09 err 0.005	-0.11 err 0.008	+0.53 err 0.000	-0.73 err 0.089	+0.13 err 0.006	-0.03 err 0.021
-0.07 err 0.019	+0.21 err 0.000	-0.36 err 0.016	+0.52 err 0.096	-0.02 err 0.005	+0.35 err 0.000
+0.02 err 0.018	+0.18 err 0.004	-0.37 err 0.007	+0.70 err 0.086	+0.19 err 0.000	-0.13 err 0.024
-0.01 err 0.013	+0.15 err 0.006	-0.44 err 0.014	-1.43 err 0.000	-0.02 err [illegible]	-0.31 err 0.011
-0.14 err 0.007	-0.01 err [illegible]	-0.13 err 0.019	+0.81 err 0.002	+0.01 err [illegible]	+0.27 err 0.017
+0.32 err 0.000	-0.08 err 0.012	-0.17 err 0.014	-0.06 err 0.058	+0.10 err 0.009	+0.05 err 0.000
s=0.045	s=0.030	s=0.076	s=0.204	s=0.028	s=0.050

误差降低 3.04×, 列 outlier 不影响其他列

示例矩阵为占位 (6×6, 第 4 列 outlier σ=4× 其他列); 后续用 Qwen3 实测投影矩阵替换

一句话

- per-channel 误差比 per-tensor 降低 ~3×, 多花的存储微不足道 (每列多存一个 s)

量化什么: 权重 / 激活 / KV cache

三个对象，难度各不同

权重 (Weights)

- 静态: 训练完后**永不变**
- 分布平稳, 接近正态
- 离线一次性量化即可
- INT4 几乎无损 (PTQ)
- 收益: 显存 / 加载带宽
- 代表方法: **GPTQ, AWQ**

激活 (Activations)

- 动态: **每个输入都不同**
- 有少数 outlier 通道
- 在线量化或离线统计
- INT8 主流 (W8A8)
- 收益: Tensor Core 算力
- 代表方法: **SmoothQuant**

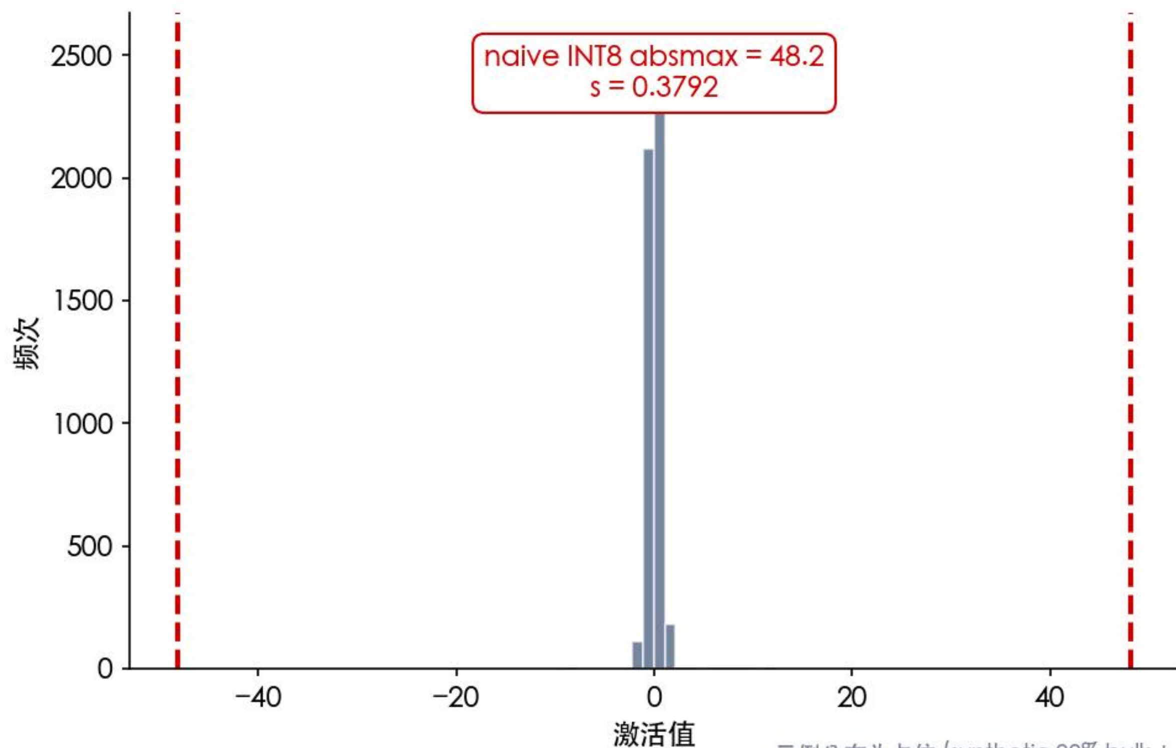
KV cache

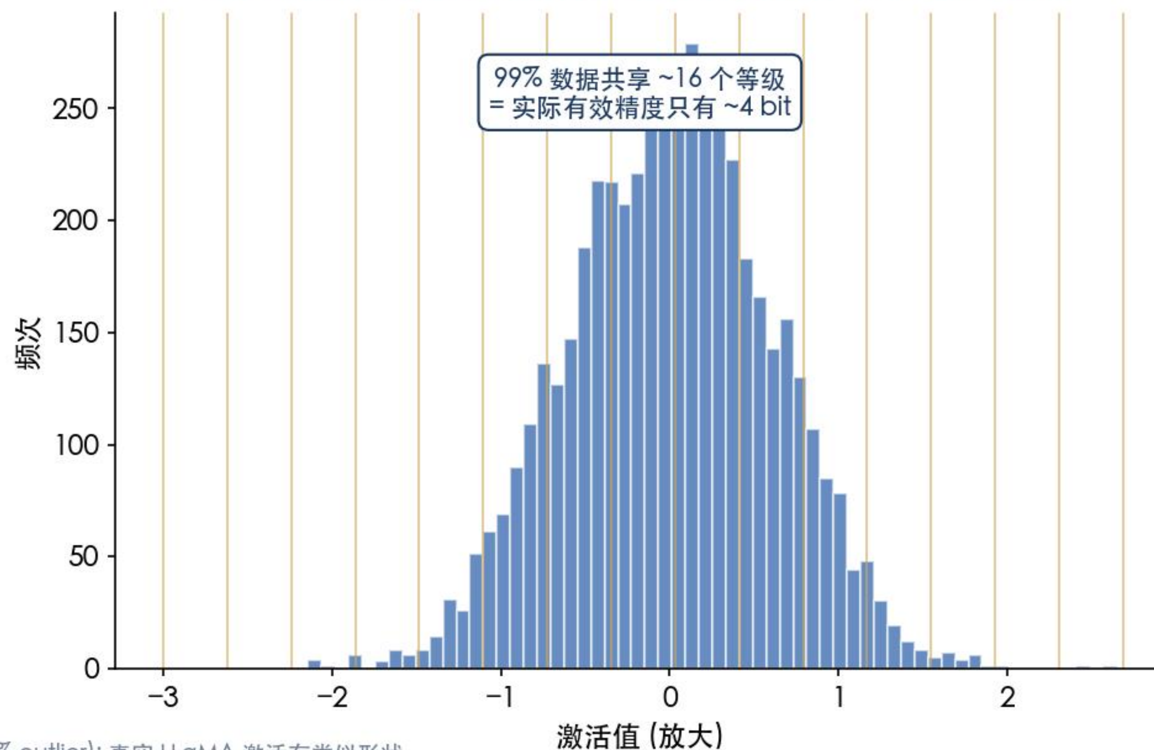
- 累积: **每生成一个 token 增长**
- 长上下文下显存超过权重
- 流式量化 (per-token / per-channel)
- INT4 KV 几乎无损
- 收益: 长上下文吞吐
- 代表方法: **KIVI**

三类对象需要**不同的量化策略**: 后面三节课会逐个展开

实例 4: 激活 outlier 突破 INT8 量化范围

一个 LLM 现象: 99% 激活在 [-2, 2] 之内, 1% 飙到 [-50, 50]





示例分布为占位 (synthetic 99% bulk + 1% outlier); 真实 LLaMA 激活有类似形状

为什么 W8A8 在 OPT-175B 上失效

- naive INT8 用 absmax = 50 算 scale → bulk 区域只用了不到 10 个等级
- 99% 数据**实际有效精度只剩 ~3 bit**, 任务精度急剧下降
- 这就是 LLM.int8() / SmoothQuant / AWQ 都要解决的核心问题

PTQ 整体流程: 校准 → 计算 → 量化 → 部署

PTQ = Post-Training Quantization: 不动训练, 一次性把模型压缩

时间成本

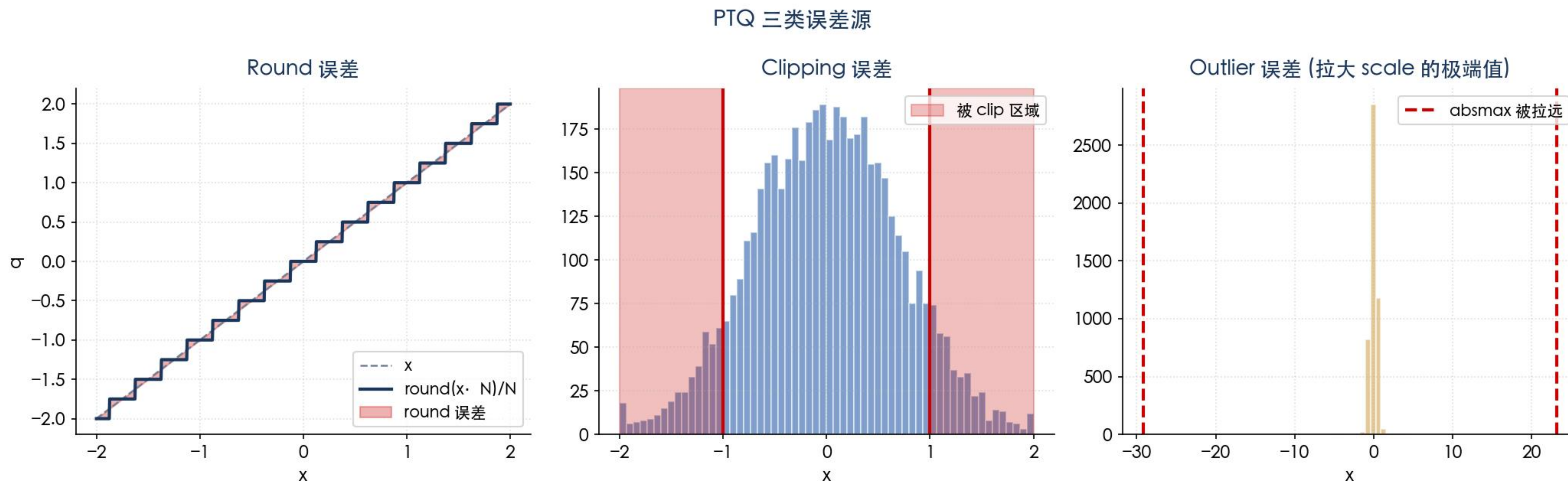
- 校准: 少量样本 (100~1000 条)
- 计算 (s,z) + 量化: 单卡几小时 (取决于方法)
- 不需要反向传播, 不需要 GPU 集群

对比: QAT (Quantization-Aware Training)

- 在训练 / 微调时插入伪量化 + STE 反传
- 精度更好, 但需要**完整训练成本**
- 极低比特 (1-2 bit) 才必须用 QAT

PTQ 误差来自哪里: round / clip / outlier

三类误差: 不同方法在攻击不同来源



后面要看到的攻击点

- **Round 误差**: 由 scale 决定，给定 bit 数已经下界 → 选**好的粒度** (per-channel/per-group)
- **Clip 误差**: 由 absmax 决定，可调 → 选**合适的 clip ratio** (LSQ / LWC 学习)
- **Outlier 误差**: 少数极大值使 scale 被撑大, 小值精度丢失 → **保护 / 迁移 / 旋转** outlier (LLM.int8 / SmoothQuant / QuaRot)

PART C

三种核心思路

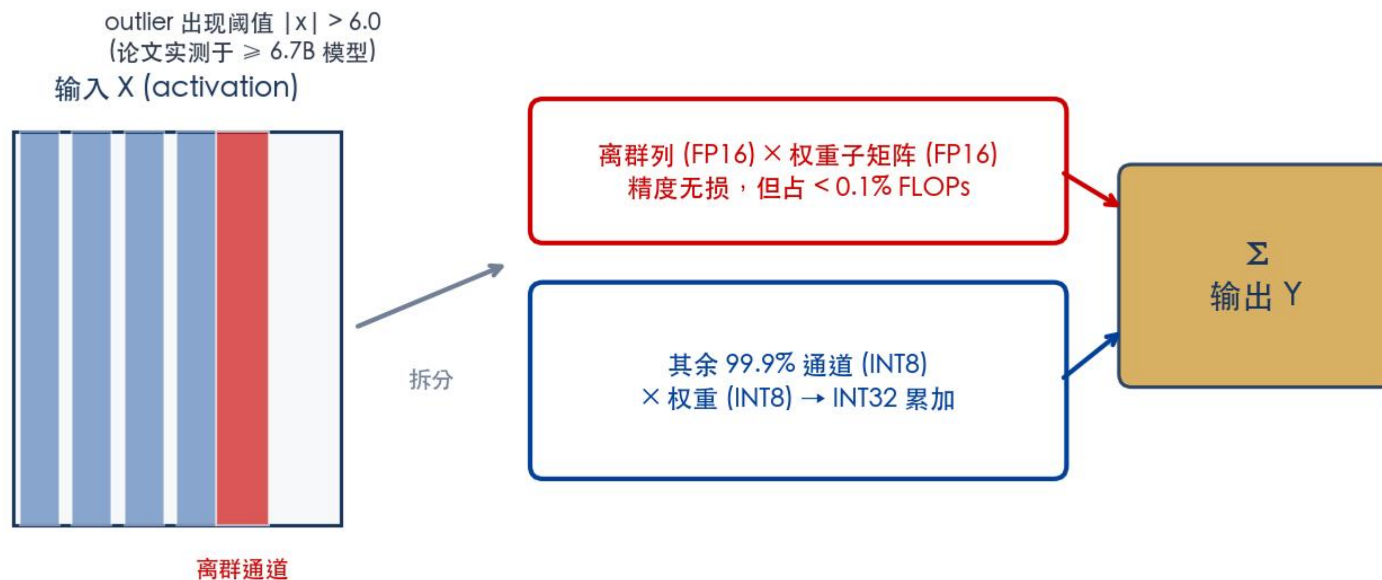
对应 4 个最常见的方法名字

本课进度

√ Part A	√ Part B	Part C	Part D	Part E
为什么要量化	量化是怎么工作的	三种核心思路	QAT 低比特	硬件视角

思想 1: 保护 outlier (LLM.int8 / AWQ)

核心: 找出少数最敏感的通道, 给它们留一条高精度通道



来源: Dettmers et al., LLM.int8(), NeurIPS 2022 (arxiv:2208.07339), §3 + Fig 1

LLM.int8() (NeurIPS 2022)

- 保护**激活 outlier 通道**: 走 FP16
- 其他 99.9% 通道走 INT8
- OPT-175B 量化后零退化 (Table 1)

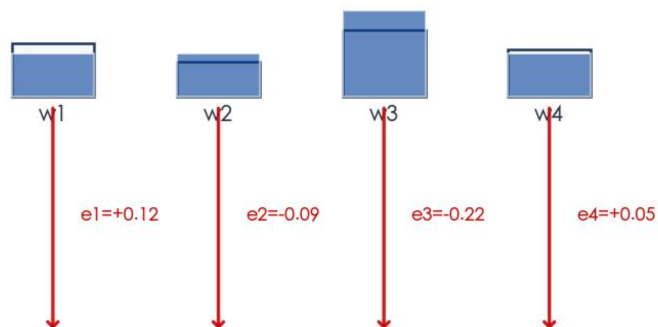
AWQ (MLSys 2024)

- 保护**权重 1% salient channel**: 放大 s 倍
- 激活除以 s 抵消, 数学等价
- LLaMA-7B INT4 PPL 5.78 (Table 4)

思想 2: 误差补偿 (GPTQ)

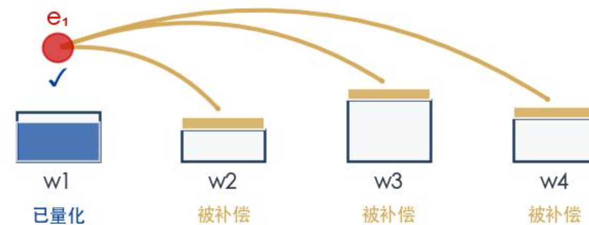
核心机制: 量化 w_i 产生误差 e_i 后, 立即调整后续未量化列 w_{i+1} …w_d 以吸收该误差

朴素 RTN: 各权重独立四舍五入



输出误差 ≈ Σ $e_i \cdot x_i$ (各列误差独立累积, 无补偿)

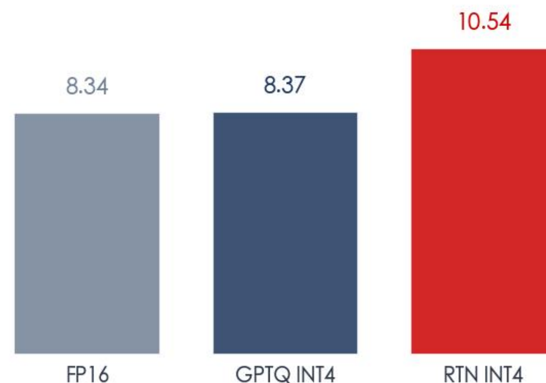
GPTQ: 量化 w_i 后, 将误差传递给 w_{i+1}…w_d



补偿公式: $w_j \leftarrow w_j - e_1 \cdot [H^{-1}]_{1j} / [H^{-1}]_{11}$ $(j > 1)$

Hessian $H = X^T X$ 决定如何分配 → 让 $Y = XW$ 几乎不变

效果: OPT-175B INT4



WikiText2 PPL (越低越好) · GPTQ ≈ FP16

来源: 概念图基于 Frantar et al., GPTQ, ICLR 2023 (arxiv:2210.17323), §3 + Table 5

GPTQ (ICLR 2023, arxiv:2210.17323)

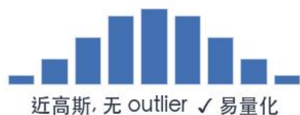
- **机制**: 量化第 i 列产生误差 e_i 后, 立即用 H^{-1} 加权分配给后续未量化列
- **方法演进**: OBS (1993, 二阶剪枝) → OBQ (2022, CV 量化) → **GPTQ (2023, LLM 量化)**; 三项优化使其可扩展到 175B: 固定列序 / lazy batched 更新 / Cholesky 数值稳定
- **结果**: OPT-175B INT4 PPL **8.37 vs FP16 8.34** (Table 5), 单卡 4 GPU · 小时 (Table 2)

思想 3: 等价变换 (SmoothQuant)

核心: W 易量化, X 难量化 → 用恒等变换将通道幅度从 X 重新分配到 W, 两者均可量化

① W 易量化 vs X 难量化

W (权重)



近高斯, 无 outlier ✓ 易量化

X (激活)

1 个固定通道 outlier ✗ 难量化

scale = max/127 被 outlier 撑大 → 小值压成 0

② 等价变换的数学依据

Y = X · W

= X · diag(s)⁻¹ · diag(s) · W

= (X · diag(s)⁻¹) · (diag(s) · W)

= (X / s) · (s · W)

↑ 第 2 行: 插入 diag(s)⁻¹· diag(s) = I ↑ 第 3 行: 结合律重组

插入对角矩阵恒等 → Y 严格不变

获得一个自由度: 通道 j 的幅度

在 X 和 W 之间重新分配

③ scale 的选取: per-channel s_j

s_j = max|X_j|^α / max|W_j|^(1-α)

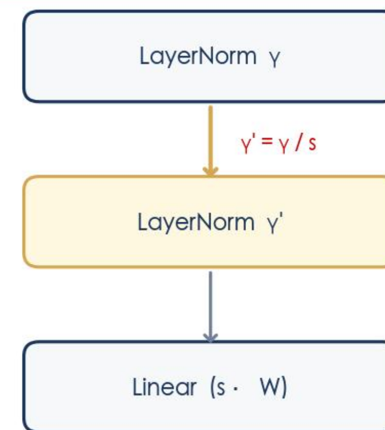
α=0 → s=1, 不变换

α=1 → 难度全部转移到 W

α=0.5 → X 与 W 平分量化难度 (默认)

通道 j 示例 (α=0.5, s ≈ 10):

④ 部署: 吸收进 LayerNorm



运行时零额外算子

s 离线计算一次, 全部折入权重

来源: Xiao et al., SmoothQuant, ICML 2023 (arxiv:2211.10438), Fig 2 + Eq 4

SmoothQuant (ICML 2023, arxiv:2211.10438)

- **变换的依据**: Y = X · W = (X/s) · (s · W), 对角矩阵插入恒等, Y 严格不变
- **变换的实施**: per-channel s_j = max|X_j|^α / max|W_j|^(1-α), α=0.5 默认令两侧承担相同的量化难度; s 离线计算后吸收进上游 LayerNorm γ, 推理无额外开销
- **结果**: OPT-175B W8A8 平均精度 **71.2% vs FP16 71.6%** (Table 4), **1.51× 加速** (Fig 8)

PART D

QAT 低比特

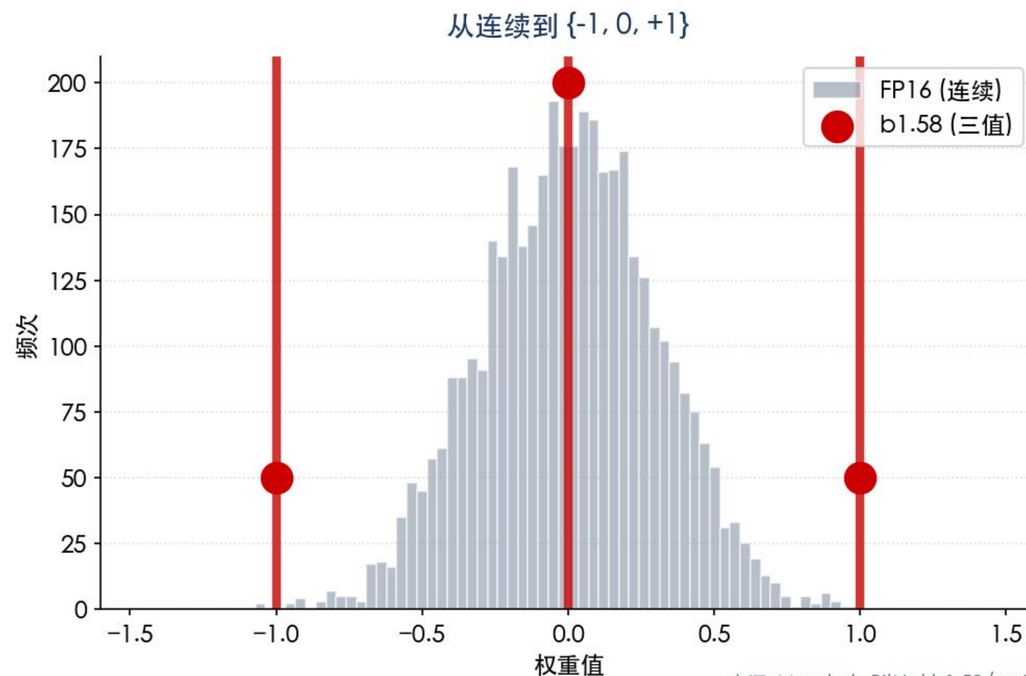
比特数压缩到 1.58 (三值)

本课进度

√ Part A	√ Part B	√ Part C	Part D	Part E
为什么要量化	量化是怎么工作的	三种核心思路	QAT 低比特	硬件视角

BitNet b1.58: 三值权重 {-1, 0, +1}

矩阵乘法退化为加法: 不是 PTQ, 必须从头训



矩阵乘法退化为加法

$$y_i = \sum_j w_{ij} \cdot x_j$$

(w 是 FP16 → 浮点乘法, 算力 + 功耗大)

$$y_i = \sum_{w=+1} x_j - \sum_{w=-1} x_j$$

(w ∈ {-1, 0, +1} → 加 / 减 / 跳过, 无乘法)

7nm 矩阵乘算术能耗: BitNet vs LLaMA = 71.4× 节省 (论文 Fig 3)

来源: Ma et al., BitNet b1.58 (arxiv:2402.17764), Eq 1-3 + Fig 3

方法

- 训练时直接用三值权重 + 8-bit 激活 (QAT)
- absmean 量化: W̃ = round(W / mean|W|), 限制在 {-1, 0, +1}
- 论文: Ma et al., 2024 (arxiv:2402.17764)

结果 (Table 1)

- 3B 起达到 FP16 LLaMA-alike 同等精度
- 7nm 矩阵乘**算术能耗节省 71.4×** (Fig 3)
- 但: 需要从头训, 不能转换已有 FP16 模型

PART E

硬件视角

memory-bound vs compute-bound

本课进度

√ Part A	√ Part B	√ Part C	√ Part D	Part E
为什么要量化	量化是怎么工作的	三种核心思路	QAT 低比特	硬件视角

量化为什么能加速: 两条路径

LLM 推理两个阶段: decoding 受访存限, prefill 受算力限

Decoding (单 token 生成)

瓶颈: HBM → SRAM 加载权重

INT4 加载 1/4 数据 → 4× 速度

受益主体: 权重量化 (GPTQ / AWQ)

Prefill (长 prompt 处理)

瓶颈: Tensor Core 算力

INT8/FP8 → 2× / 6× 算力

受益主体: W8A8 / FP8 (SmoothQuant + Hopper)

路径 1: 访存路径 (decoding 提速 ~4×)

- decoding 阶段每生成 1 token 都要把**整套权重**从 HBM 搬到 SM
- INT4 每权重 0.5 字节 vs FP16 2 字节，**搬运量 1/4**
- 算力没变，但搬运变快 → **端到端 ≈ 4×**
- 适用方法: **GPTQ / AWQ W4A16**

路径 2: 算力路径 (prefill 提速 2–6×)

- A100 同代: INT8 Tensor Core **624 TOPS** vs FP16 **312 TFLOPS** ≈ **2×**
- H100 跨代: FP8 **1979 TFLOPS** vs A100 FP16 **312 TFLOPS** ≈ **6×**
- 必须 W 与 A 都低精度 (W8A8 / FP8) 才能上 Tensor Core
- 适用方法: **SmoothQuant W8A8**, **FP8 (Hopper+)**

TOPS/TFLOPS 数据来自 NVIDIA A100 / H100 datasheet (dense, no sparsity)

本节小结: 方法谱图 + L2 实战预告

4 个名字, 3 个思想: 接下来在 A100 上把它们跑一遍

L1 要点回顾

- 量化 = 把实数轴离散到 2^b 个等级
- 三个对象: 权重 / 激活 / KV cache
- 三个思想: 保护 / 补偿 / 等价变换

L2 预告: 在 A100 上端到端复现

- bitsandbytes 一行加载 (NF4)
- AutoGPTQ 离线量化 Qwen3-1.7B
- AutoAWQ + vLLM 部署
- llama.cpp GGUF Q4_K_M (CPU 路径)

第二节课

实战: 在 A100 上量化 Qwen3

Qwen3-1.7B 跑通 4 条路径; Qwen3-32B 单 A100 部署故事

本节课路线图: 在 A100 上完整复现量化全流程

主线: 实验室 1 张 A100, 部署 Qwen3-32B; 看 4 条路径如何兑现承诺

本节课你将看到

- **真实 GPU**: A100-SXM4-80GB, GPU id 3-5
- **真实模型**: Qwen3-1.7B (hands-on), Qwen3-32B (部署案例)
- **真实工具链**: transformers / llmcompressor / llama.cpp
- **真实数据**: 5 个量化方案 × 4 项指标, 全部实测
- 全部命令、中间日志均可在课后复现

本节课重点不是

- 不是讲新方法的数学原理 (那是 L3 内容)
- 不是给标准答案: 不同方案各有得失
- 不是 fine-tune 调参表
- **重点**: 给同学一种**可复现、可观察**的实操脚手架
- 你跑完之后, 应能自己挑一个方法部署你想要的模型

案例场景: 实验室收到一个部署任务

任务: 用现有的 1 张 A100-SXM4-80GB 把 Qwen3-32B 部署成内部服务

约束条件

- 硬件: A100-SXM4-80GB × 1 (其他 GPU 被占满)
- 模型: Qwen3-32B BF16 (62 GB on disk)
- 服务对象: 实验室 ~10 名研究生, 偶发 batch=2-4
- 上下文需求: 最少 8K, 偶尔 16-32K (论文阅读场景)
- 时间预算: 数小时内完成, 排除需训练数日的方案
- 精度要求: 与原模型相比可接受 ≤5% 退化

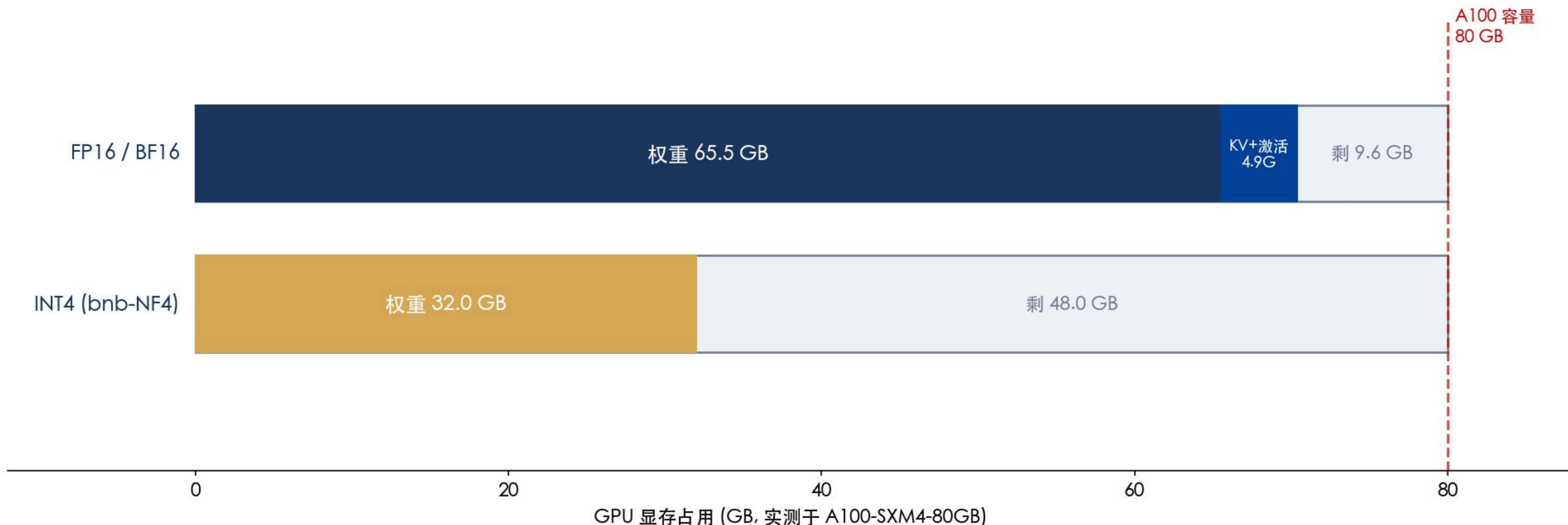
为什么这个任务有趣

- A100 显存上限 80 GB, 模型权重 62 GB, **看似够**
- 实际加载时还要算上: 激活、KV cache、CUDA 工作区
- batch=2 / context=16K 时 KV 占用快速上升
- 不做量化的方案要么 OOM, 要么吞吐过低
- 量化把权重压到 16-32 GB, 留出空间给 KV / 并发
- 接下来用真实命令一步步验证这个判断

本节课目标: 让模型从 "显存不足" 状态变为 "可加载且可推理" 状态。

诊断 1: Qwen3-32B FP16 在 A100 上还剩多少余量

实测: FP16 加载就吃掉 70.4 GB, 仅剩 9.6 GB; INT4 把权重压到 32 GB



数据说明

- weights: from_pretrained 之后 `torch.cuda.max_memory_allocated()` 第一次读数
- KV+激活: 跑一次 1024-token prefill 后峰值减去 weights
- 剩余: 80 GB 减去峰值; 这部分要**给 batch / 长 context 用**

实测于 GPU 4-5; 详见 results/qwen3_32b_fp16.json 与 qwen3_32b_bnb4.json

评价一个量化方案的四类指标

评估不是单点: 同一个方法在精度好但速度差, 或反过来

精度类

- WikiText-2 PPL: 通用语言建模的退化
- HF stride 协议 (max_len=2048, stride=512)
- llama.cpp 用自己的 PPL 协议, 不直接可比
- 阈值: 一般认为 PPL 涨幅 < 1.0 可接受

资源类

- disk_size_gb: 模型在磁盘上的大小
- weights_mem_gb: 加载完成后 GPU 占用
- peak_mem_gb: 跑完一次推理后的峰值
- 三者差异反映: 是否做权重压缩、是否预留 buffer

速度类

- decode tok/s: 单条 batch=1 自回归生成 (用户感知延迟)
- prefill tok/s: 1024-token 单次 forward (吞吐能力)
- 量化主要加速 decode (访存路径), 不一定加速 prefill

易用性

- 是否需要校准数据 (代价 5-30 min for 1.7B-32B)
- 是否需要额外文件格式转换
- 是否需要专门的推理 runtime

PART A

场景与评估

诊断 + 评估指标 + 4 条路径概览

本课进度

Part A	Part B	Part C	Part D	Part E	Part F
场景与评估	bitsandbytes	GPTQ 校准	AWQ 与 llama.cpp	对比与选型	回到部署案例

4 条路径概览: 简明区分

两类: 在线量化 (bnb) vs 离线量化 (GPTQ/AWQ/llama.cpp)

路径 1: bitsandbytes	路径 2: GPTQ	路径 3: AWQ	路径 4: llama.cpp
在线量化 (load 时)	离线量化 + 二阶补偿	离线量化 + 保护 salient	离线量化 + 自定义 kernel
· 不需校准	· 需校准 (~512 样本)	· 需校准	· 不需校准 (启发式)
· BitsAndBytesConfig	· llmcompressor	· llmcompressor	· GGUF 格式
· NF4 / int8	· W4A16	· W4A16	· Q4_K_M
· <1 min	· ~5 min/1.7B	· ~6 min/1.7B	· <1 min

读图要点

- bnb 不需校准, 加载即量化; GPTQ/AWQ 需校准 (5-10 min); llama.cpp 用启发式
- 后续每条路径花 4-5 张片子完整跑一遍, 重点是命令行 + 中间过程

实验环境: 软件栈 + 模型清单

复现这一节实验所需: 4 个 Python 包 + 1 个 C++ 项目

软件版本 (实测可用)

- **Python 3.10**, torch 2.10.0+cu128
- **transformers 5.6.2** (Qwen3 支持需要 ⩾4.51)
- **bitsandbytes 0.49.2** (CUDA 12.x)
- **llmcompressor 0.10.0** (内置 GPTQ + AWQ)
- **llama.cpp** commit 9d34231 (CUDA-only build)
- **datasets 4.8.4** (WikiText-2 加载 + PPL eval)

模型 + 数据

- /data/models/Qwen3-1.7B (BF16, 3.8 GB on disk)
- /data/models/Qwen3-32B (BF16, 62 GB on disk)
- 校准数据: WikiText-2 train, 512 samples × 2048 tokens
- 评估数据: WikiText-2 test split, 282 K tokens
- 工作目录: /workspace/zhangwuyang/quant_course/
- 结果输出: results/*.json + logs/*.log

环境一次配置完成后, 4 条路径每条只需 1-2 个命令即可执行。

PART B

路径 1: bitsandbytes

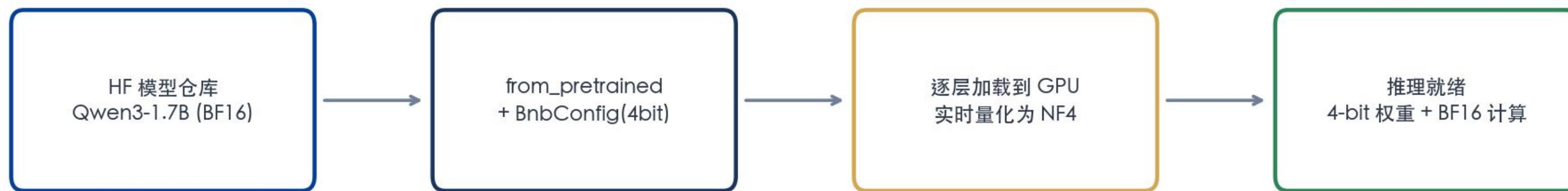
在线量化, 1 行代码; 适合原型与小模型

本课进度

√ Part A	Part B	Part C	Part D	Part E	Part F
场景与评估	bitsandbytes	GPTQ 校准	AWQ 与 llama.cpp	对比与选型	回到部署案例

bitsandbytes: 加载时即量化, 无中间文件

关键观察: 模型在硬盘上仍是 BF16, 量化发生在 GPU 加载阶段



无校准、无中间文件; 模型到达 GPU 时已是量化态

为什么它实现简单

- 不需要单独的量化脚本; 只在加载模型时多传一个 BitsAndBytesConfig
- 不需要校准数据; 量化用每张权重张量自身的 absmax 直接得 scale
- 配 NF4 (4-bit) 与 int8 两种模式, 都集成在 transformers 里
- **代价**: 在硬盘上仍是全精度, 不能省下载或部署带宽

实例 1: bitsandbytes 4-bit (NF4) 加载代码

目标: 用 NF4 (Normal Float 4) 量化 Qwen3-1.7B 并测 PPL/速度

bench.py (相关片段)

```
import torch
from transformers import AutoModelForCausalLM, BitsAndBytesConfig

cfg = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",          # NF4 vs fp4
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=True,     # 二次量化压 scale
)

model = AutoModelForCausalLM.from_pretrained(
    "/data/models/Qwen3-1.7B",
    quantization_config=cfg,
    device_map="auto",
)
```

实测结果 (results/qwen3_1.7b_bnb4.json)

- weights_mem = **2.28 GB** (FP16: 4.38 GB), PPL = **15.91** (FP16: 14.96)
- decode = **22.9 tok/s** (FP16: 45.4), prefill = 17 K tok/s
- load_time = 3.5 s; **量化与下载是同一步, 无独立的量化时间**

实例 2: bitsandbytes 8-bit 的实测 — 与预期相反的速度结果

与预期相反: INT8 显存确实降低, 但 decode 速度反而慢于 FP16

只改一行

```
cfg = BitsAndBytesConfig(
    load_in_8bit=True,
)

# 其他代码不变
```

实测对比 (FP16 vs bnb-8bit)

- weights_mem: 4.38 → **2.97** GB √
- PPL: 14.96 → **14.67** (基本不退化) √
- decode: 45.4 → **8.7** tok/s ✗
- prefill: 32 K → **7.6 K** tok/s ✗

原因: bnb int8 路径用了 LLM.int8() outlier 分离, kernel 较慢

- LLM.int8() 把激活分为 outlier (走 FP16) + bulk (走 INT8) 两路, 单独乘
- A100 上小模型 (1.7B) 权重已能完全驻留 HBM, 内存带宽不是瓶颈
- 把单一 GEMM 拆成两路, 反而引入了额外开销
- **教学点**: 量化方案的精度好不一定意味着推理快; runtime kernel 决定速度
- 大模型 (Qwen3-32B) 上访存压力大时, bnb-8bit 可能更优, 但需要单独验证

PART C

路径 2: GPTQ 校准

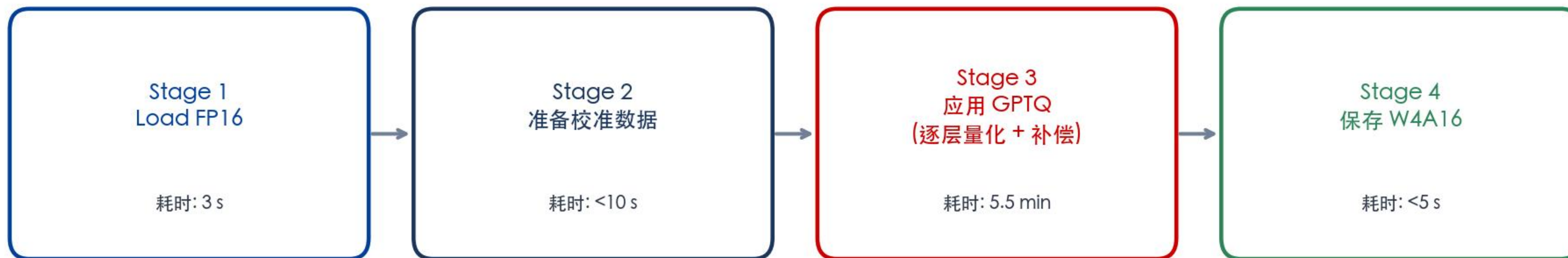
离线量化 + 二阶信息补偿; 精度通常最好

本课进度

√ Part A	√ Part B	Part C	Part D	Part E	Part F
场景与评估	bitsandbytes	GPTQ 校准	AWQ 与 llama.cpp	对比与选型	回到部署案例

GPTQ 流程: 4 个阶段

GPTQ 离线量化的特点: 量化前要看一批校准数据, 量化时逐层补偿误差



为什么时间几乎都花在 Stage 3

- 加载 / 数据准备 / 保存都是常数级开销
- Stage 3 要逐层做: 计算每层 Hessian + GPTQ 算法 + 误差传播
- 28 层 × 7 modules (q/k/v/o/gate/up/down) ≈ 196 个量化步骤
- 每步 0.7-2.4 秒 (与权重维度相关), 加上每层做一次校准前向
- A100 上 Qwen3-1.7B 总耗时 5.5 分钟; 32B 估算 ~30-60 分钟

GPTQ Stage 2: 准备校准数据

校准数据: 一小批让 GPTQ 看到模型典型激活分布的样本

methods/quant_gptq.py (Stage 2 片段)

```
from datasets import load_dataset

ds = load_dataset("wikitext", "wikitext-2-raw-v1", split="train")
ds = ds.filter(lambda x: len(x["text"]) > 200)  # 去掉短句
ds = ds.shuffle(seed=42).select(range(512))     # 取 512 条

def preprocess(example):
    return tokenizer(example["text"], truncation=True,
                     max_length=2048, padding=False,
                     add_special_tokens=True)

ds = ds.map(preprocess, remove_columns=ds.column_names)
```

实操要点

- **样本量**: GPTQ 论文用 128 样本; 实际 256-1024 都常见, 越多越稳
- **来源**: 通用任务用 WikiText / C4; 垂直领域应换成自己业务数据
- **长度**: max_length 太短 → 校准看不到长依赖; 太长 → 显存吃不消
- 这一段需要的代码量很少, 是整个流程里最容易出错的环节 (数据来源决定量化质量)

GPTQ Stage 3: 应用 recipe (4 行配置 + 1 行调用)

关键: scheme=W4A16 表示权重 4-bit, 激活留 16-bit (动态)

methods/quant_gptq.py (Stage 3 片段)

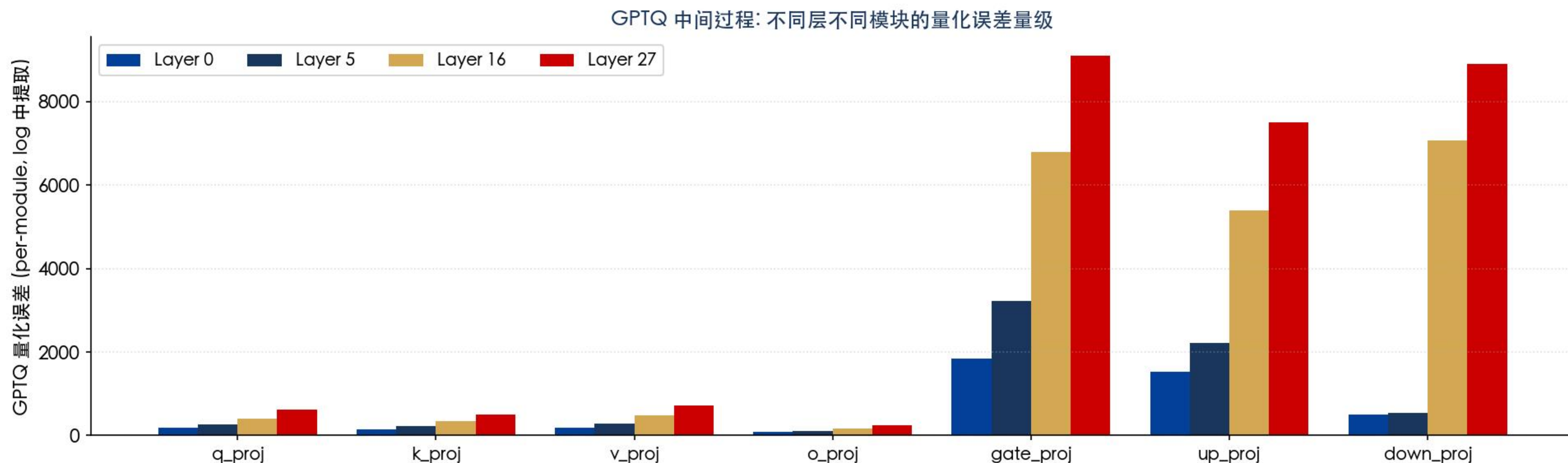
```
from llmcompressor.modifiers.quantization import GPTQModifier
from llmcompressor import oneshot

recipe = GPTQModifier(
    targets="Linear",       # 量化所有 nn.Linear
    scheme="W4A16",         # 权重 4-bit, 激活 fp16
    ignore=["lm_head"],     # 输出层保留高精度
)

oneshot(model=model, dataset=ds, recipe=recipe,
        max_seq_length=2048, num_calibration_samples=512)
```

recipe 的几个关键参数

- **scheme=W4A16**: 权重 4-bit, 激活保持 fp16 → 是所有 GPU 量化主流方案
- **targets="Linear"**: 只量化全连接; embedding/layernorm 通常不量化
- **ignore=["lm_head"]**: vocabulary 输出层敏感, 保留 fp16 是工程经验
- 其他常用: dampening_frac, sequential_targets, percdamp (与 Hessian 求逆稳定性有关)
- 同一份代码改 scheme 为 "W8A8" 即可做 SmoothQuant 风格量化

GPTQ 中间过程: 每层量化误差量级随深度上升

观察: 越深层、越宽的 module (mlp.gate/up/down) 误差越大



数据来自 logs/quant_gptq.log

- 每个 module 量化后, llmcompressor 打印一行 "METRIC - error <数字>"
- 这个数字是该层量化前后激活的 Frobenius 误差 (累计在校准 batch 上)
- 趋势: layer 0 → layer 27 误差量级增长 ≈ 5 倍, 因为后层的输入分布更复杂
- mlp 模块比 attention 模块误差大, 因为 mlp 维度更宽, 量化网格更稀

logs/quant_gptq.log: 完整日志见 quant_l2_logs/, 28 层 × 7 modules ≈ 196 行

GPTQ 实测: 1.35 GB / PPL 15.63 / decode 13.4 tok/s

5.5 分钟换来: 模型大小压到 1/3, PPL 升 0.7, 但 decode 比 FP16 慢

校准结束日志 + bench 结果

```
[Stage 3] √ GPTQ done in 5.5 min
[Stage 4] √ Saved (1.35 GB)

$ python bench.py \
   --model qwen3-1.7b-gptq-w4a16 \
   --tag qwen3_1.7b_gptq

[bench] load_time=2.2 s
[bench] weights_mem=1.35 GB
[bench] ppl=15.63 (vs FP16 14.96)
[bench] decode=13.4 tok/s
[bench] prefill=11707 tok/s
[bench] peak_mem=8.39 GB
```

对比 FP16 / bnb-4bit

- 大小: GPTQ **1.35** vs FP16 4.08 vs bnb 4.08 (硬盘)
- PPL: GPTQ **15.63** vs bnb-4bit 15.91 → GPTQ 略好
- decode: GPTQ **13.4** vs bnb-4bit 22.9 → bnb 反而快
- 原因: GPTQ 用 packed 4-bit + 反量化 kernel; transformers 实现不算很优
- 想要 GPTQ 同时取得最佳速度, 需配合 vLLM / SGLang 等专门 runtime
- 接下来 AWQ / llama.cpp 会进一步揭示这个 "算法 vs runtime" 的二维取舍

PART D

路径 3 + 4: AWQ 与 llama.cpp

AWQ: 保护 salient channel; llama.cpp: 自定义 K-quant kernel

本课进度

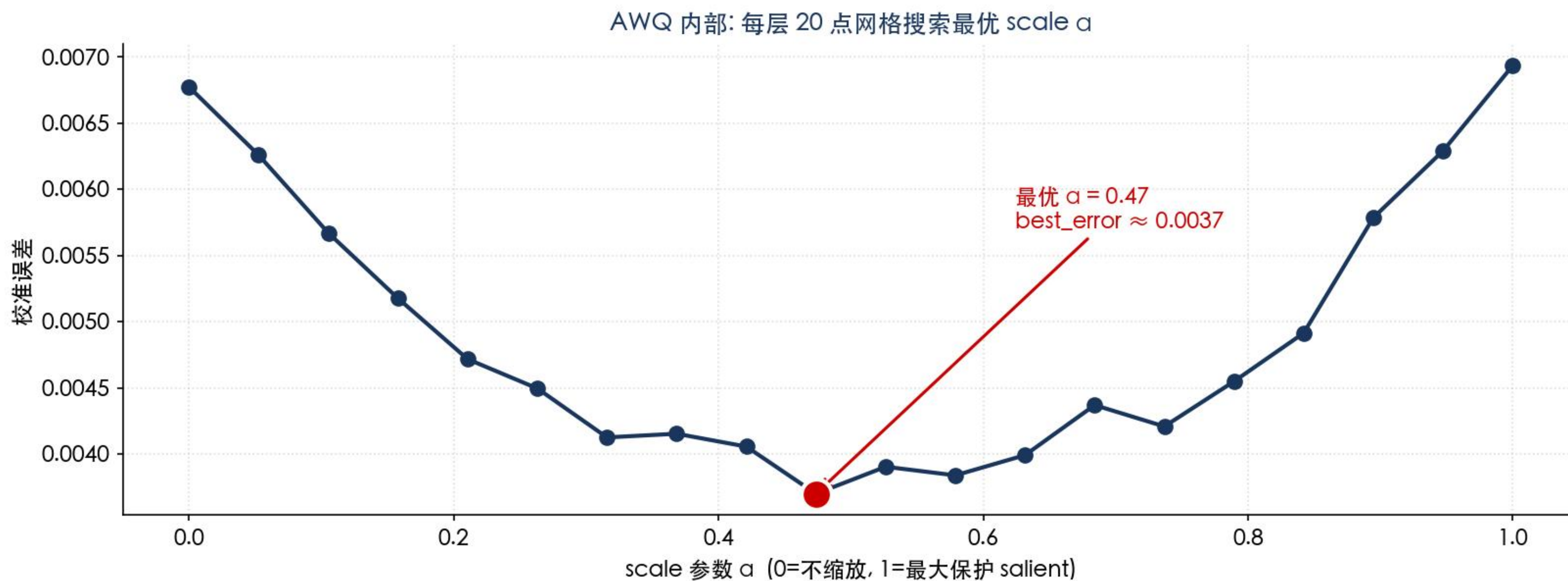
√ Part A	√ Part B	√ Part C	Part D	Part E	Part F
场景与评估	bitsandbytes	GPTQ 校准	AWQ 与 llama.cpp	对比与选型	回到部署案例

AWQ: 与 GPTQ 同一框架, 只换一个 modifier

llmcompressor 设计的好处: GPTQ 与 AWQ 共享 oneshot + dataset 接口

methods/quant_awq.py (与 GPTQ 仅差一行)

```
from llmcompressor.modifiers.awq import AWQModifier

recipe = AWQModifier(
   targets="Linear", scheme="W4A16", ignore=["lm_head"],
)
oneshot(model=model, dataset=ds, recipe=recipe,
      max_seq_length=2048, num_calibration_samples=512)
```



AWQ 内部对每层做 20 点网格搜索找最优 scale α; 这是 AWQ 比 GPTQ 慢的主要原因

AWQ 实测: 与 GPTQ 显存接近, PPL 略高

结论: 在 Qwen3-1.7B 上 GPTQ ⩾ AWQ; 但 AWQ 的强项在 W4A4 / 大模型场景

AWQ 实测 (results/qwen3_1.7b_awq.json)

- 校准时间: 5.9 min (略长于 GPTQ 5.5)
- 模型大小: 1.37 GB (同 GPTQ)
- PPL: 16.99 (GPTQ 15.63 / FP16 14.96)
- decode: 13.3 tok/s (与 GPTQ 几乎相同)

为什么 AWQ 在小模型上不如 GPTQ?

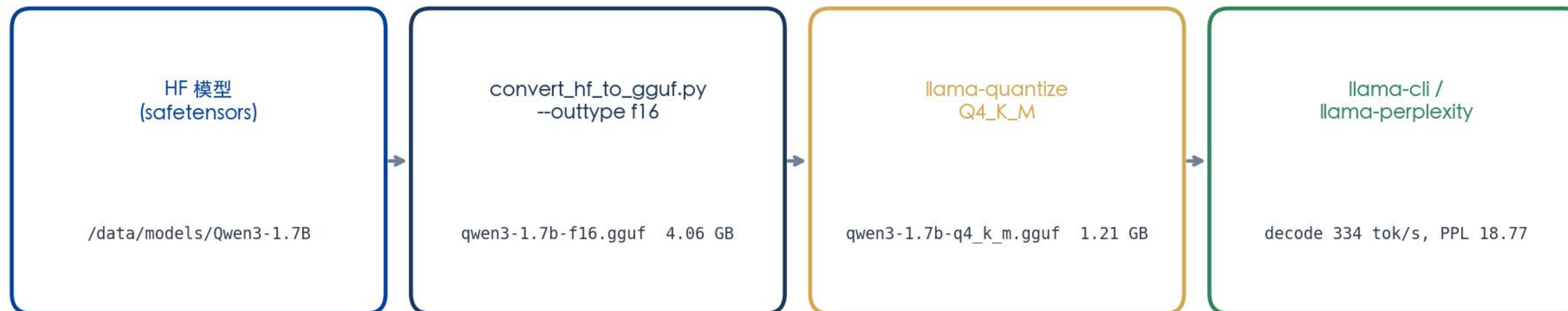
- AWQ 的优势在: 当激活有强 outlier 时, 通过缩放保护 salient channel
- Qwen3-1.7B 在 WikiText 上激活 outlier 不显著
- AWQ 的额外 grid search 反而引入 ~1 PPL 噪声
- 在 Qwen3-32B 或 Llama-2-70B 上 AWQ 通常达到与 GPTQ 持平或更优的精度

实操观察: 同一脚手架, 切换 modifier 只需 1 行

- llmcompressor 把 GPTQ / AWQ / SmoothQuant 抽象成 Modifier
- 对你的代码: 只换 import + 类名, 数据加载与保存逻辑共享
- 这个抽象很重要: 在生产里你会把多种方案放进同一 CI 跑对比
- 本课的两个 .py 文件 quant_gptq.py / quant_awq.py 高度相似

llama.cpp Q4_K_M: 三个命令完成量化与基准

llama.cpp 自带量化与推理 runtime: 不依赖 transformers / torch



GGUF 是 llama.cpp 的统一文件格式; Q4_K_M 是其 K-quant 中的中等档位

完整命令序列 (假设 llama.cpp 已 build)

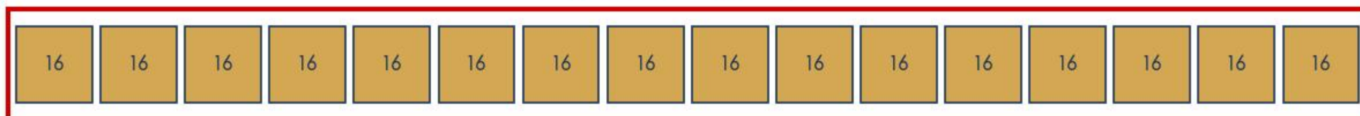
```
# 1) HF safetensors → GGUF F16
python convert_hf_to_gguf.py /data/models/Qwen3-1.7B \
    --outfile qwen3-1.7b-f16.gguf --outtype f16
# 2) F16 GGUF → Q4_K_M GGUF
llama-quantize qwen3-1.7b-f16.gguf qwen3-1.7b-q4_k_m.gguf Q4_K_M
# 3) Bench + PPL
llama-bench -m qwen3-1.7b-q4_k_m.gguf -p 1024 -n 64 -ngl 99
llama-perplexity -m qwen3-1.7b-q4_k_m.gguf -f wiki.test.raw -ngl 99
```

K-quants 是什么: 每 256 元素一个 super-block

Q4_K_M 不是单一 4-bit 量化, 而是 mixed: 部分子块 6-bit, 多数 4-bit

K-quants 数据结构: 256 元素的 super-block + 16 个子-scale

Super-block: 256 个权重



子块: 16 元素 (各自一个 6-bit 子-scale)

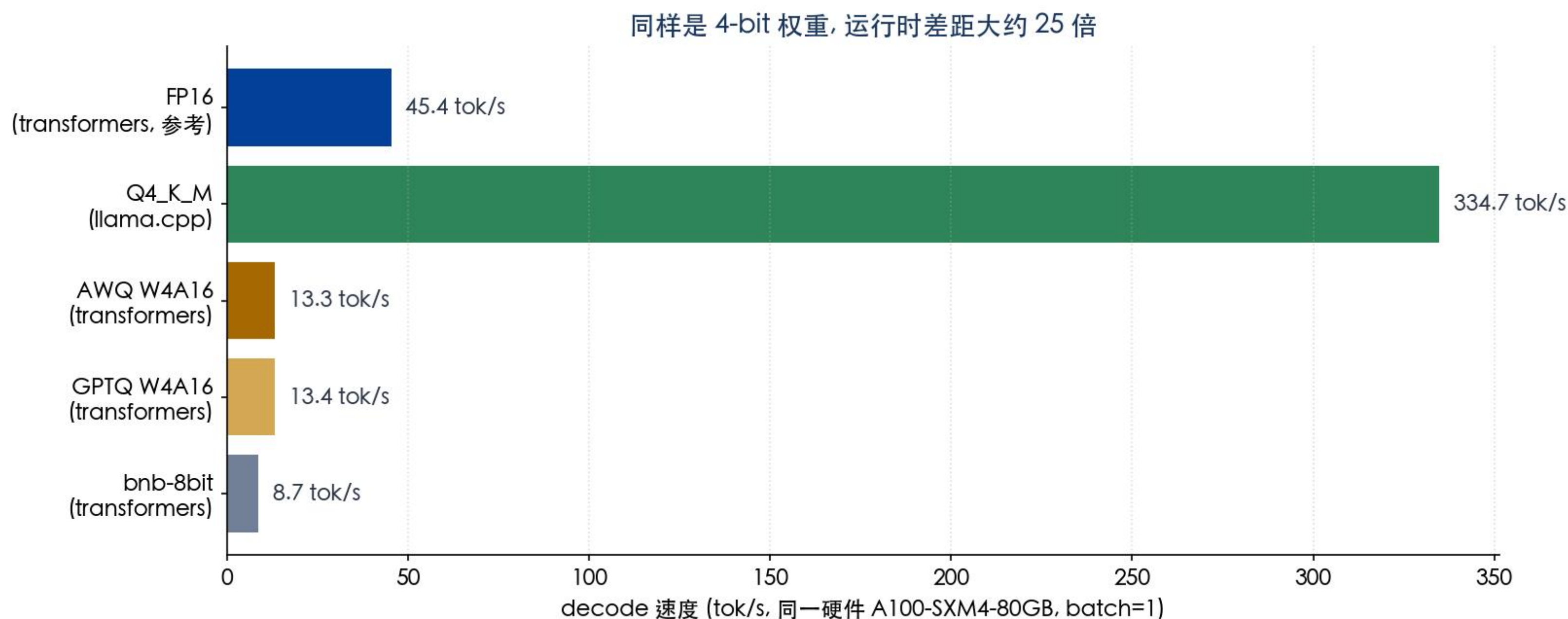
Q4_K_M = mixed:
attn_v / ffn_down 子块走 Q6_K (6-bit 高精度)
其余子块走 Q4_K (4-bit + 6-bit 子-scale)

为什么这种结构能让 decode 极快

- 子块的 scale 用 6-bit, 进一步压实, 但保留较好动态范围
- 256 元素一组对齐到 GPU warp / CPU cache 边界, 反量化访存友好
- llama.cpp 写了 hand-tuned CUDA kernel 直接 fused dequant + GEMV
- **教学点**: 同样是 4-bit, K-quants 的精度 + 速度都强于 simple group-wise int4

llama.cpp 实测: decode 比 transformers 路径快约 25 倍

同样的 4-bit 权重, 同样的 A100, 不同 runtime → 25× 差距



怎么看待这个差距

- llama.cpp Q4_K_M 在 1.7B 上 decode = 334 tok/s; transformers GPTQ/AWQ 仅 13 tok/s
- 差异主要来自: (1) 自定义 CUDA kernel; (2) 更紧的内存布局; (3) graph 复用
- 但 llama.cpp 不易嵌入 Python 业务代码; 适合独立服务或边缘部署
- 在 Python 服务里要拿到接近的速度, 应该用 vLLM / SGLang + GPTQ/AWQ 模型
- **结论**: 算法选好之后, runtime 决定上限

llama.cpp build 9d34231, sm_80 only; bench: pp1024 / tg64; 详见 results/

PART E

对比与选型

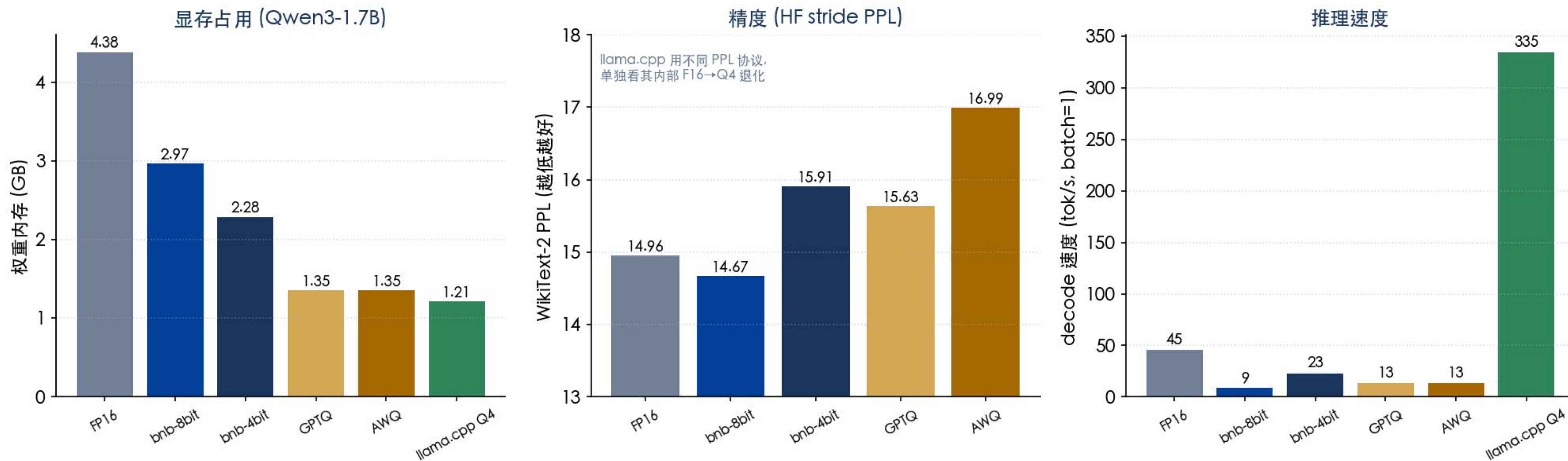
把四条路径放进同一张表, 看清取舍

本课进度

√ Part A	√ Part B	√ Part C	√ Part D	Part E	Part F
场景与评估	bitsandbytes	GPTQ 校准	AWQ 与 llama.cpp	对比与选型	回到部署案例

5 个方案 × 3 项指标 (Qwen3-1.7B 实测)

全部 PPL / 显存 / decode 数据来自实测, 详见 quant_l2_results/

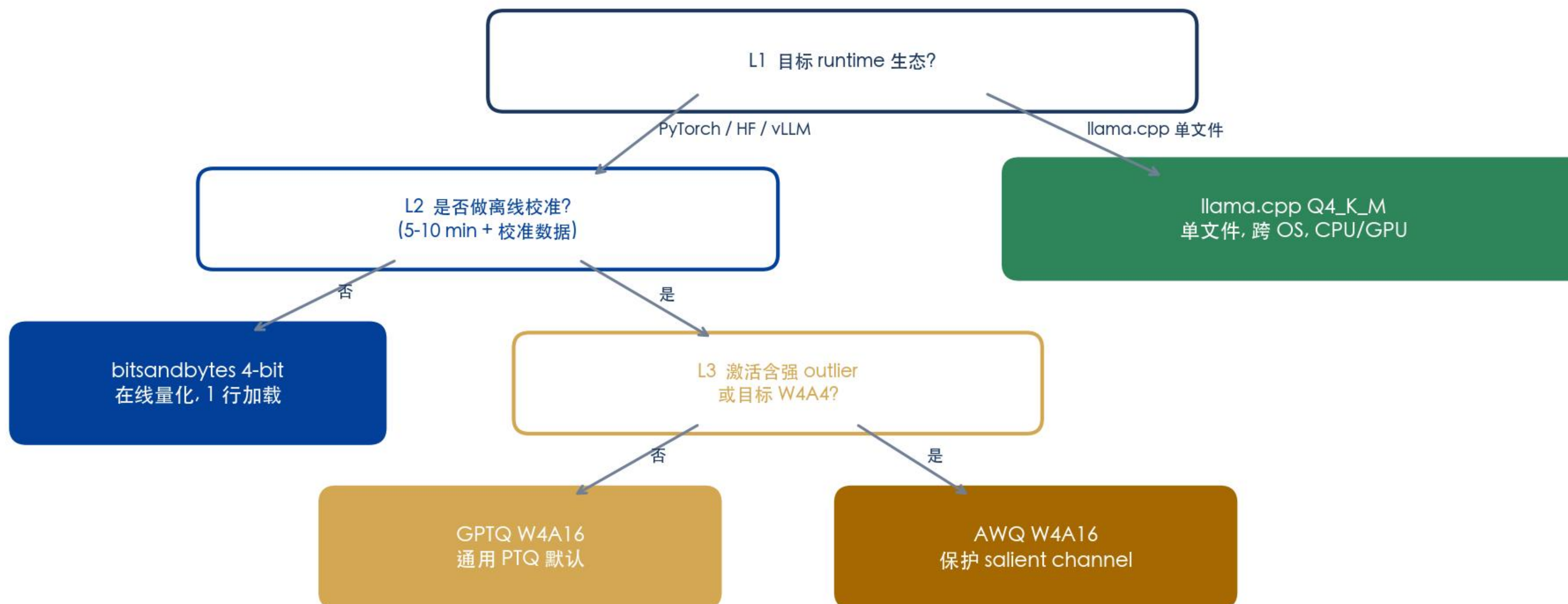


读图三个观察

- **显存**: GPTQ/AWQ/llama.cpp ≈ 1.3 GB; bnb 仍占 2-3 GB (硬盘没压)
- **PPL**: bnb-8bit 最好, GPTQ 次之, AWQ 偏差; llama.cpp PPL 协议不同, 单独看
- **decode**: 在 transformers runtime 下都不快; llama.cpp 一枝独秀

选型决策树: 三个独立维度

三个维度: runtime 生态 / 是否做离线校准 / 激活分布是否平稳



典型场景对照

- **PyTorch 服务器部署 7B – 13B**: GPTQ W4A16 + vLLM (默认通用配置)
- **大模型 30B+ 或 W4A4 目标**: AWQ W4A16 (保护 salient channel)
- **跨平台 / CPU / 端侧 / 单文件分发**: llama.cpp Q4_K_M (独立 runtime)
- **无校准数据 / 频繁切换模型 / QLoRA 微调**: bitsandbytes (在线 4-bit, 1 行加载)

PART F

回到部署案例

Qwen3-32B 单 A100 部署的最终方案

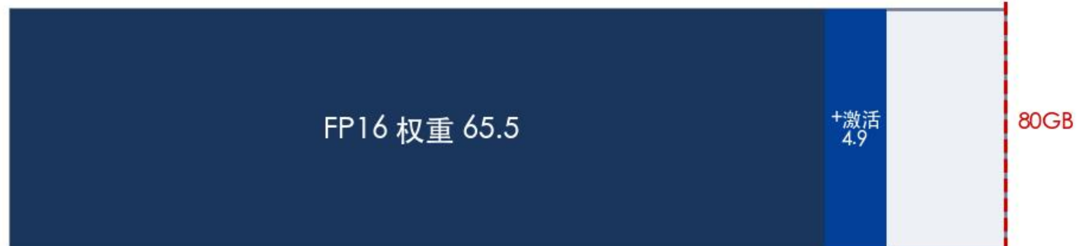
本课进度

√ Part A	√ Part B	√ Part C	√ Part D	√ Part E	Part F
场景与评估	bitsandbytes	GPTQ 校准	AWQ 与 llama.cpp	对比与选型	回到部署案例

回到部署案例: Qwen3-32B 单 A100 部署 — 量化前后对比

结论: bnb-NF4 使 Qwen3-32B 在单卡 A100 上从 "刚好可加载" 变为 "留有 48 GB 余量"

Qwen3-32B 单 A100 部署: FP16 vs INT4 实测对比



为什么本案例选 bnb-NF4 而非 GPTQ

- bnb-NF4 不需校准: 拿到模型即可上线, 节省时间
- 这是首次部署原型阶段, 之后可换 GPTQ + vLLM 提升 decode 吞吐
- 实际生产: 推荐 GPTQ-W4A16 + vLLM 组合, 同时取得精度与速度

Qwen3-32B FP16: 70.4/80 GB, decode 18.6 tok/s; bnb-NF4: 32/80 GB, decode 9.5 tok/s; PPL 7.61→7.94

本节小结 + L3 预告

两个核心要点, 课后练习, 以及 L3 知识细化预告

L2 两条要点

- **要点 1**: 量化方案的算法选择 ≠ 推理速度;
  runtime kernel 决定上限 (transformers GPTQ vs llama.cpp Q4_K_M ≈ 25×)
- **要点 2**: 4 条路径覆盖了不同场景的 sweet spot
  - bnb 适合原型, llama.cpp 适合端侧
  - GPTQ + vLLM 适合 GPU 生产
- 全部代码与日志: /workspace/zhangwuyang/quant_course/
- 数据 / 命令均可课后复现

L3 预告: 数学细节

- 误差到底从哪里来? round / clip / outlier 的数学结构
- GPTQ 二阶补偿: Hessian 与逐列误差传播为什么有效
- AWQ 的 salient channel 数学定义
- outlier 处理三策略: keep / scale / rotate (含 Hadamard)
- KV cache 量化的特殊性 (KIVI)
- 选型决策的完整画面

课后任务: 自己跑一遍 4 条路径中的 2 条, 记录命令与数据 (建议: bnb + GPTQ)

第三节课

量化知识点细化

数学推导 / GPTQ 算法 / outlier 主线 / QAT / KV / 选型

本节课的主线问题: 量化误差 ε 从哪来? 怎么控制?

衔接 L1 / L2: L1 建立概念框架, L2 给出实操命令, L3 解析命令背后方法的有效性来源

L3 的论证结构

- **不是罗列方法**, 而是回答 4 层递进式问题
- 每个 Part 都用上一个 Part 留下的问题驱动
- 主线: 量化误差 ε 怎么定义、分解、控制
- 全章末尾收成一棵选型决策树

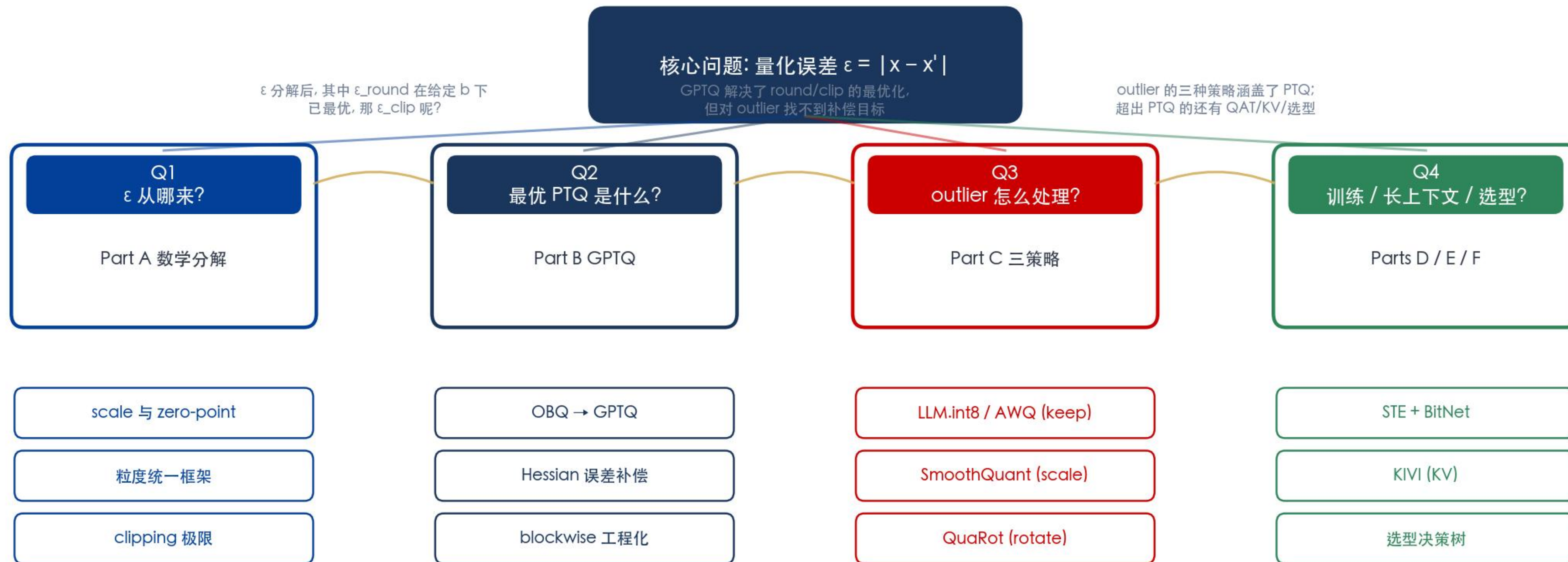
L3 不展开的内容

- 论文的实验细节 (放参考文献末尾)
- LSQ / LLM-QAT / OmniQuant 单独讲解 (合并提)
- 推理引擎工程 (vLLM / TensorRT-LLM 系统课讲)
- 命令级实操 (已在 L2 完成)

听完 L3 你应该能**自己推出**为什么 GPTQ → AWQ → QuaRot 是必然演化路径

L3 知识图谱: 4 层递进式问题

核心问题: 量化误差 ε; **递进**: 数学分解 → 最优算法 → 跳出框架 → 选型



请在脑中保留这张树: 后面每张 slide 都对应树上的一个节点

PART A

误差的数学结构

Q1: 量化误差 ε 由哪几部分组成? 各自如何控制?

本课进度

Part A
误差的数学结构

Part B
GPTQ 最优 PTQ

Part C
outlier 跳出框架

Part D
QAT 低比特

Part E
KV cache

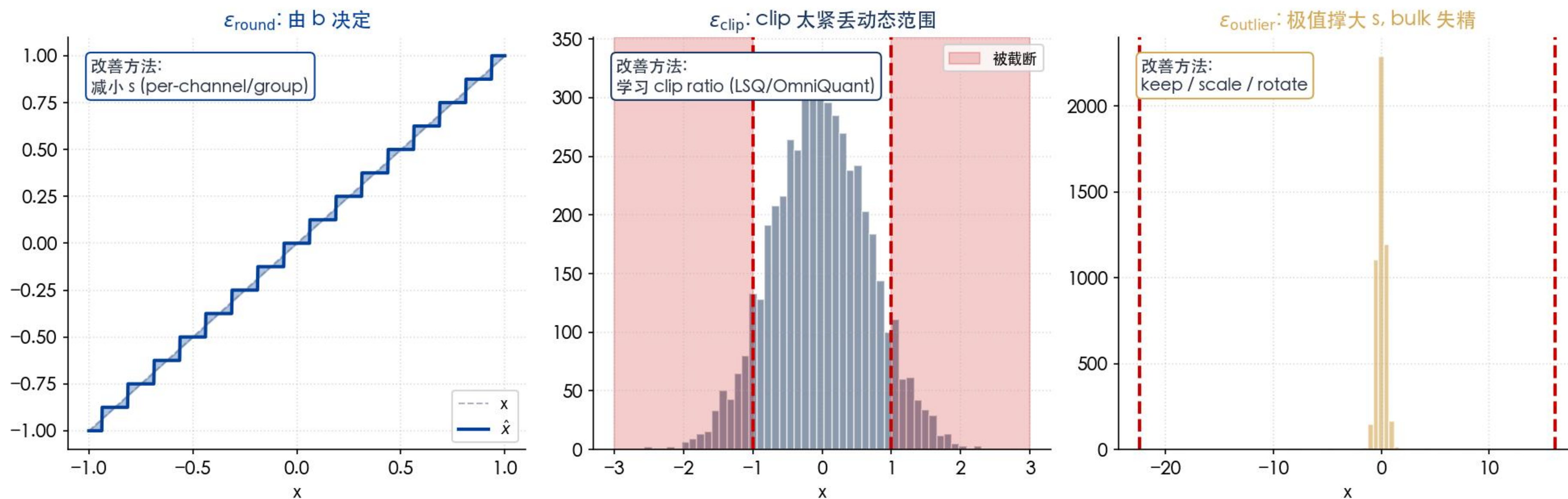
Part F
选型与总结

量化误差的分解: ε = ε_round + ε_clip + ε_outlier

统一框架: 三类误差对应三类攻击方法

$$\varepsilon = |x - \hat{x}| \le \varepsilon_{\text{round}} + \varepsilon_{\text{clip}} + \varepsilon_{\text{outlier}}$$

= 取整误差 (s/2) + 截断误差 (max(0, |x| - β)) + outlier 撑大 s 的代价



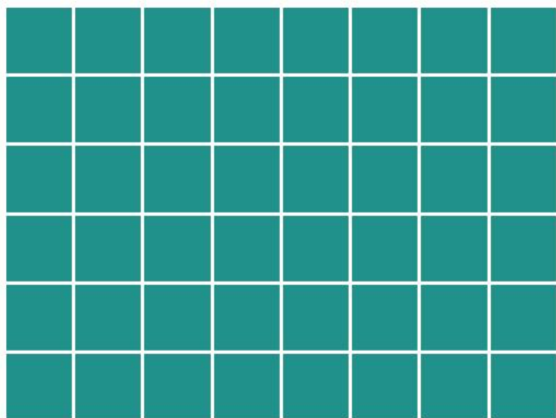
将本章方法归纳到三类误差源上

- ε_round ⩽ s/2，由 b 直接决定 → 缩小 s 的方法 = per-channel/group/token (本节 Part A)
- ε_clip = max(0, |x|-β)，由 clip ratio 决定 → 学习 clip 的方法 = LSQ / OmniQuant (本节 Part A)
- ε_outlier = 由极值撑大 s 引起 → 处理 outlier 的方法 = LLM.int8 / SmoothQuant / QuaRot (Part C)

粒度的统一框架: 减少 ε_round 的代价-收益

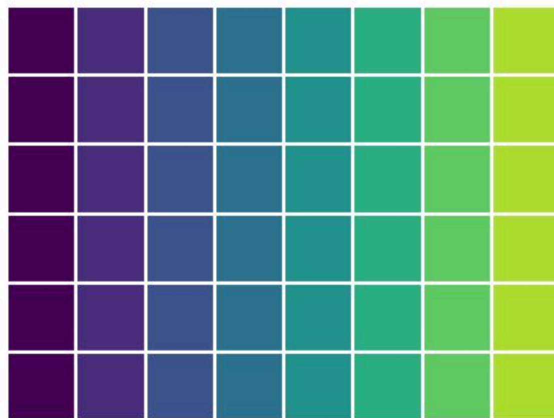
核心权衡: 粒度变细 → 每组 s 更贴近实际范围 → ε_round 更小; 但要存的 (s,z) 更多

per-tensor



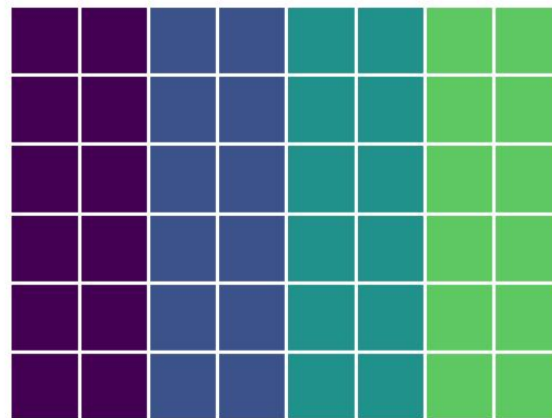
全矩阵 1 个 (s, z)

per-channel



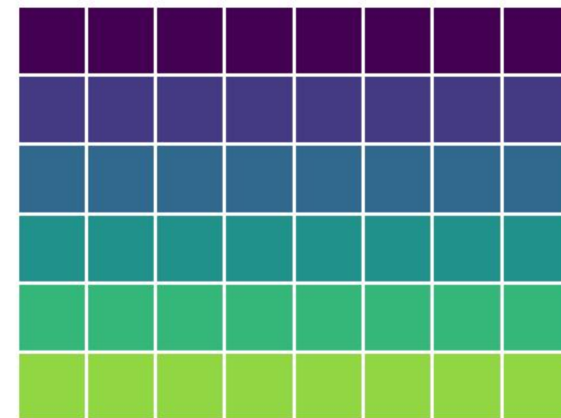
每行/列 1 个 (s, z)

per-group



每组 g=128 列 1 个

per-token



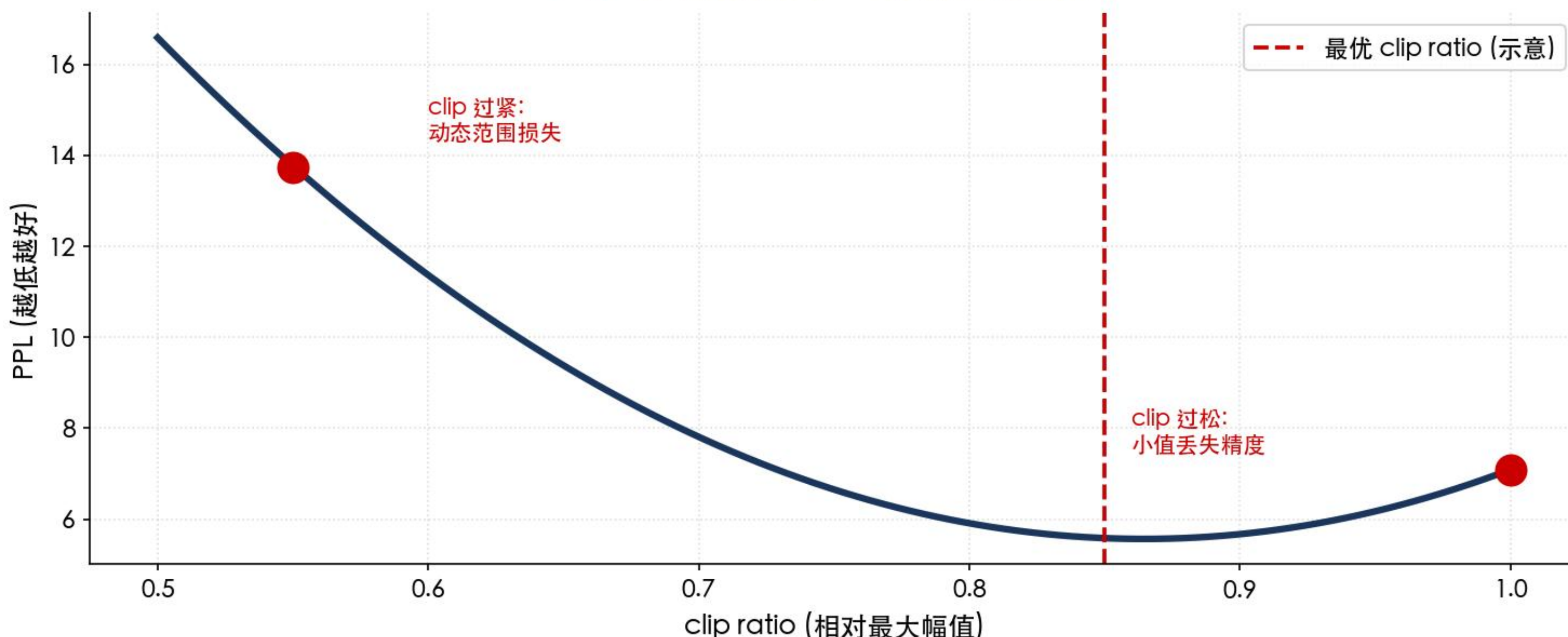
每行 (token) 1 个

四种粒度的统一公式 + 实现取舍

- **per-tensor**: 1 个 (s,z) 管整个矩阵; 元数据 ~0%; ε_round 大
- **per-channel**: 每行/列 1 个 (s,z); 元数据 0.05%; LLM 权重默认
- **per-group (g=128)**: 每 g 列 1 个; 元数据 0.4%; AWQ / GPTQ 实用粒度, ε_round 再降一档
- **per-token**: 每条样本一组 (s,z); 元数据小; 激活动态量化的常见选择
- 实践组合: **权重 per-group + 激活 per-token** 几乎是 LLM 量化的事实标准

clip 的上限: 为什么 ε_clip 不能无限优化

清晰的桥梁: clip 优化在 outlier 面前失效 → 本节末尾的两个出口



为什么 clip 不够

- 可学习的 clip (LSQ, ICLR 2020) 在 ResNet 上 W3A3 达到与 FP 持平精度
- 但 LLM 激活在**少数固定通道**上存在数十至数百倍的 outlier
- clip 紧 → bulk 数据丢动态范围
- clip 松 → bulk 数据有效精度只剩 ~3 bit
- 单 clip 参数无法两全

本节剩下的两个出口

- **出口 1 (Part B)**: 给定 ε_round + ε_clip 的最优 PTQ 算法是什么? → **GPTQ**
- **出口 2 (Part C)**: 当激活含 outlier, GPTQ 难以补偿时如何处理? → **三类策略**

PART B

GPTQ 最优 PTQ

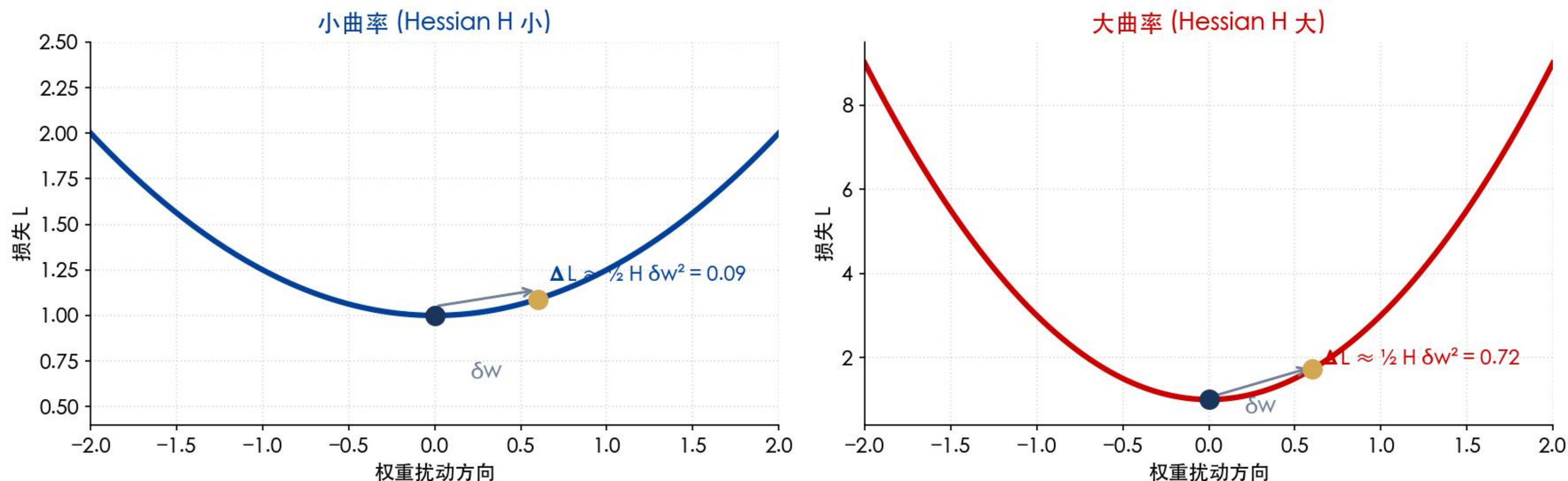
Q2: 给定 ε_round + ε_clip, 不重训练能找到最优解吗?

本课进度

√ Part A	Part B	Part C	Part D	Part E	Part F
误差的数学结构	GPTQ 最优 PTQ	outlier 跳出框架	QAT 低比特	KV cache	选型与总结

前置: Hessian 的定义及其与量化的关联

一句话: Hessian H = 二阶导矩阵, 告诉我们改一个权重对损失的**二阶影响有多大**



数学回顾: 损失的二阶展开

- 在最优点 w* 附近: L(w*+∂w) ≈ L(w*) + ½ ∂w^T H ∂w
- **H 大** → 改这个权重损失变化大 → 该权重重要
- **H 小** → 改这个权重损失变化小 → 该权重可压缩
- 一阶项 = 0 (在最优点梯度为 0), 二阶才是主项

为什么量化要用 Hessian?

- 量化误差 = 一种特殊的权重扰动 ∂w
- 想知道某个量化方案对损失的影响 → 看 ½ ∂w^T H ∂w
- **H^{-1}** 给出补偿方向: 量化掉的误差通过 H^{-1} 投射到其他权重
- 完整 H 是 d^2, 计算昂贵; 实际用 H ≈ 2 X^T X (按层近似)

记住三句话: H = 曲率; 大 H = 重要; H^{-1} = 补偿方向。下一页正式进入 OBQ。

从 OBS / OBQ 到 GPTQ: 二阶信息 + 误差补偿

核心机制: 量化一个权重后, 将误差按 Hessian 加权传递给后续未量化的列以抵消

来源: Frantar et al., GPTQ, ICLR 2023 (arxiv:2210.17323) + OBQ Frantar et al., NeurIPS 2022 (arxiv:2208.11580)

GPTQ 算法的数学形式 (单层 Linear 权重)

- **Step 1**: 由校准激活计算 Hessian $H = 2 \cdot X^T \cdot X$ (per-layer, 编码各权重对输出 Y 的局部影响)
- **Step 2**: 按列序量化第 q 列 w_q* = quantize(w_q)，得到误差 δ_q = w_q − w_q*
- **Step 3**: 用 H^{-1} 加权将 δ_q 分配到所有未量化列 w_p (p > q): w_p ← w_p − δ_q · H^{-1}_{qp} / H^{-1}_{qq}
- **几何含义**: 该更新是使 $\|Y - Y^*\|^2$ 在 w_p 子空间内最小化的闭式解 (OBS 的二阶最优补偿)
- **复杂度演化**: OBS (1993) $O(d^4)$, 仅理论 → OBQ (2022) $O(d \cdot d_col^2)$, CV 模型可行 → GPTQ (2023) 工程化后, OPT-175B 4 GPU · 小时完成

Frantar et al., GPTQ, ICLR 2023 (arxiv:2210.17323), §3 + OBQ NeurIPS 2022 (arxiv:2208.11580)

GPTQ 工程化: 算法与系统协同设计的代表性案例

三项优化将单步从带宽受限转为算力受限, 175B 量化时间从估算数天降至 4 小时

(1) 固定列序量化

- **OBQ 做法**: 每步用 H^{-1} 重新选最优下一列, 循环依赖高
- **GPTQ 做法**: 直接按列号顺序量化, 去掉选列开销
- **正确性**: 补偿后误差在剩余列上均匀分布, 顺序量化与贪心顺序精度等价
- **收益**: 主循环成为规整循环, 可批量化、可向量化

(2) Lazy batched updates

- **OBQ 做法**: 每量化 1 列立即更新一次 H^{-1}, 访存频繁
- **GPTQ 做法**: 累计 128 列后批量更新一次 H^{-1}
- **实现**: 一次更新 = 一次 GEMM, 走 Tensor Core
- **收益**: HBM 读写次数降至约 1/128, 单步从带宽受限转为算力受限

(3) Cholesky 数值稳定

- **问题**: H^{-1} 元素长链相乘易出现病态 (累积舍入误差)
- **方法**: 对 H 先做 Cholesky 分解 $H = LL^T$, 后续以 L 为基础做更新
- **效果**: 避免 H^{-1} 元素直接长链累乘, 保证 175B 量级模型的数值收敛
- **背景**: 数值线性代数中处理对称正定矩阵的标准做法

结果数据 (Frantar et al. 2023, Table 2 + Table 5)

- **OPT-175B / BLOOM-176B**: 单 A100 上 4 GPU · 小时 完成 4-bit 量化
- **OPT-175B WikiText-2 PPL**: FP16 = 8.34, GPTQ-4bit = 8.37 (差 0.03)
- **BLOOM-176B**: FP16 = 8.11, GPTQ-4bit = 8.21 (差 0.10)
- 在 Llama-3-8B / Qwen3-7B 这种现代尺寸上, AutoGPTQ 量化通常 30-60 分钟 (取决于校准集)

GPTQ 的局限: outlier 在 Hessian 框架下找不到补偿目标

清晰的桥梁: GPTQ 是数学框架内的最优解, 但激活 outlier 在框架外

GPTQ 能解决的

- ε_round (round 误差) 通过补偿把误差均摊到邻居
- ε_clip 在 absmax 给定后 对每行权重做最优分配
- 训练好的 175B 模型也能跑

GPTQ 解不了的

- 激活在少数固定通道存在显著 outlier 时 absmax 被这些通道撑大
- 权重 Hessian 不能补偿"激活的 scale"
- 因为 Hessian 框架假设激活已知
- 在 W4A4 下精度急剧下降

出路: 跳出权重 Hessian 框架, 直接处理 outlier 本身

- 思路 1: **保留** outlier 让它们走高精度 → LLM.int8 (NeurIPS 2022)
- 思路 2: **缩放** outlier 等价迁移 → SmoothQuant (ICML 2023) / AWQ (MLSys 2024)
- 思路 3: **旋转** 让 outlier 摊到所有维度 → QuaRot (NeurIPS 2024)
- 这三条思路构成 Part C 主线

PART C

outlier 跳出框架

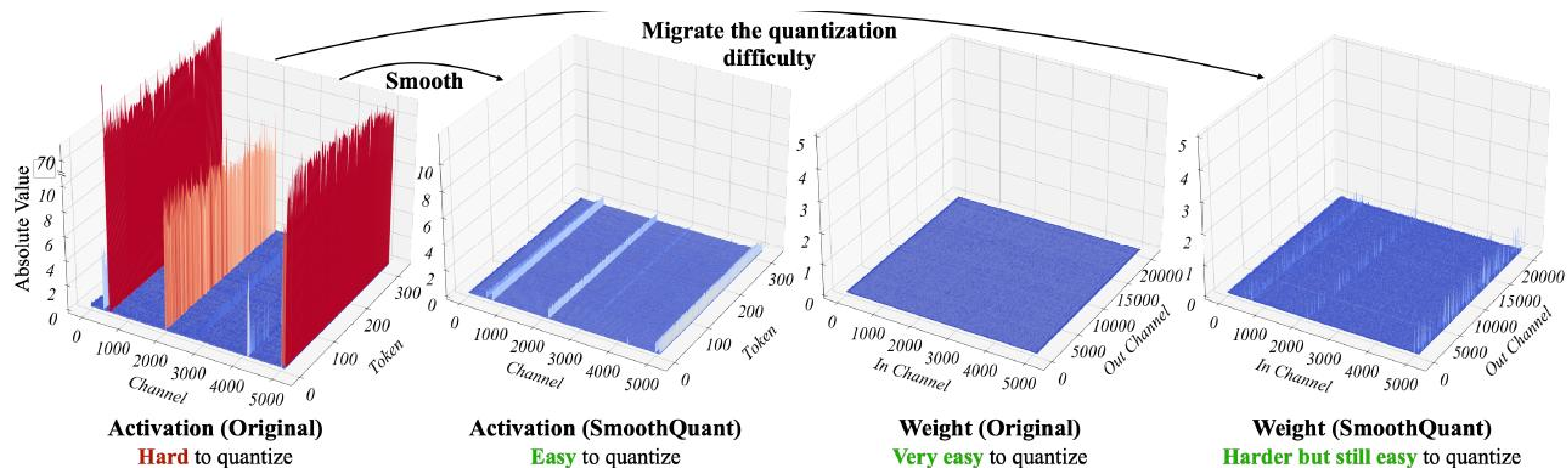
Q3: 当数学框架不够用, 该怎么处理 outlier?

本课进度

√ Part A	√ Part B	Part C	Part D	Part E	Part F
误差的数学结构	GPTQ 最优 PTQ	outlier 跳出框架	QAT 低比特	KV cache	选型与总结

outlier 现象: 在 Qwen3 / Llama-3 等现代模型上仍然普遍

关键事实: outlier 是 Transformer 训练涌现的固有现象, 而非旧模型 artifact



实证规律 (LLM.int8 §2 / SmoothQuant Fig 4)

- 模型规模 ≥ 6.7B 后, outlier **涌现成为固定现象**
- 集中在**少数固定通道**, 不随输入变化
- 幅值是 bulk 通道的 6-100 倍
- 在 OPT / BLOOM / Llama-2 / Llama-3 / Qwen 上均观察到

为什么这事影响很大

- 一个通道的 outlier 撑大整层的 absmax
- bulk 99% 数据有效精度从 8 bit 降到 ~3 bit
- W8A8 在 OPT-175B 上 naive 量化精度 32% (FP16 = 71.6%)
- 这就是 SmoothQuant 论文 Table 4 的现象

图: SmoothQuant Fig 4; 数字: SmoothQuant ICML 2023, Table 4; 在 Qwen3 上 L2 实战会再现

三种 outlier 策略: keep / scale / rotate

统一视角: 五个方法都属于这三种数学操作之一

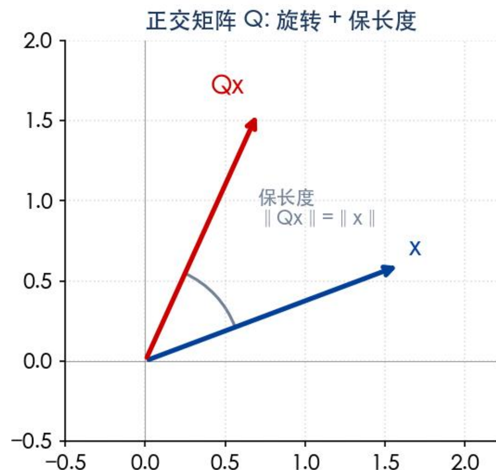
三种策略的数学本质

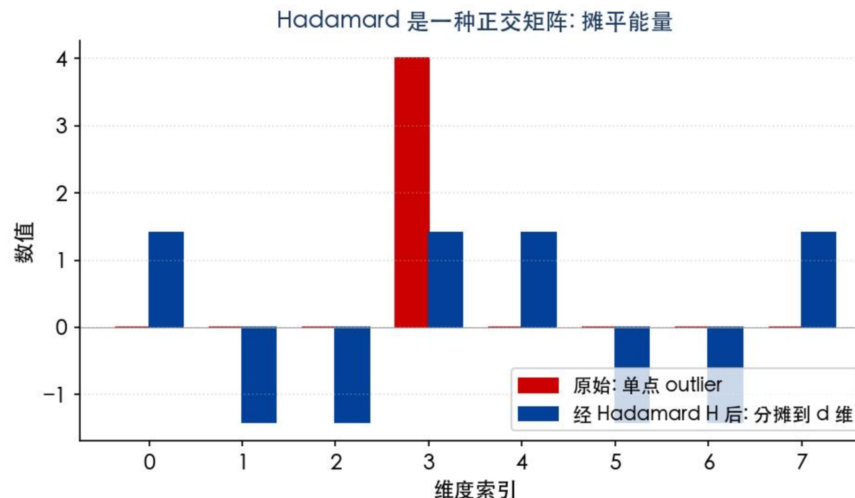
- **keep**: outlier 保留高精度, 工程上 mixed precision (LLM.int8: outlier 走 FP16; Atom: 走 INT8)
- **scale (diagonal)**: $Y = (X \cdot D^{-1}) \cdot (D \cdot W)$, 用对角矩阵在 X、W 间转移难度 (SmoothQuant / AWQ)
- **rotate (orthogonal)**: $Y = (X \cdot R) \cdot (R^T \cdot W)$, R 正交, 把 outlier 摊到所有维度 (QuaRot)
- 表达力递进: keep < diagonal < orthogonal, 但工程复杂度也递进

前置: 正交矩阵 Q 的三个性质

一句话: Q^T Q = I 的矩阵 Q, 几何上 = 旋转/反射, **不改变向量长度也不改变内积**

正交矩阵的两个性质: 保长度 (能量不变) + 内积不变 (输出不变)





三个核心性质

- **保长度**: ‖ Qx ‖ = ‖ x ‖ (能量不变)
- **保内积**: (Qx)^T (Qy) = x^T y (角度不变)
- **可逆且简单**: Q^{-1} = Q^T (转置即逆)
- 因此 Y = X W = (X Q)(Q^T W), Q 可插入矩阵两侧而不改变输出

Hadamard 矩阵 H: 一种特殊正交

- 元素全是 ±1/√d (同幅值, 无 outlier)
- 一个大值 x 经 Hx 后 = d 个 ±x/√d 之和
- 单点能量被**均摊**到 d 个新通道
- 量化看到的不再是 outlier, 而是 d 个普通值
- 计算便宜: 只需加减法, FFT 风格 O(d log d)

两个性质: 正交 = 输出不变; Hadamard = 将 outlier 分摊到所有维度。下一页介绍 QuaRot 如何同时利用这两条。

QuaRot: 用正交旋转把 outlier 分摊到 d 个维度

三步: ① 数学恒等保证 Y 不变 → ② 选 Hadamard 让单点能量摊到 d 维 → ③ 离线吸收 + 在线 Hadamard

① 数学恒等: 可插入正交 Q

$$Y = X \cdot W$$
$$= X \cdot (Q \cdot Q^T) \cdot W$$
$$= (X \cdot Q) \cdot (Q^T \cdot W)$$

↑ Q 正交 → $Q \cdot Q^T = I$

插入 $Q \cdot Q^T = I$ → Y 严格不变

获得一个自由度: 选取适当的正交 Q

可重排 X 的分布形态

② Hadamard Q 的特殊价值

原始 X (单点 outlier)

max = 4.0

×H →

X · H (摊到 d 维)

max ≈ 4/√d = 1.41

Hadamard H 元素均为 ±1/√d (近似各向同性)

单点幅值 v 经 H 后 → d 个 ±v/√d, max 缩小 √d 倍

d=4096 时 max 缩小 64×, 4-bit 即可量化

③ 部署: 离线 + 在线

X (含 outlier)

↓

× H (Hadamard) — 在线

↓

Linear ($H^T \cdot W$) — 离线

在线开销: O(d log d) Hadamard

远小于 Linear 的 O(d²)

来源: Ashkboos et al., QuaRot, NeurIPS 2024 (arxiv:2404.00456), §3.1-3.4 + SliceGPT (ICLR 2024)

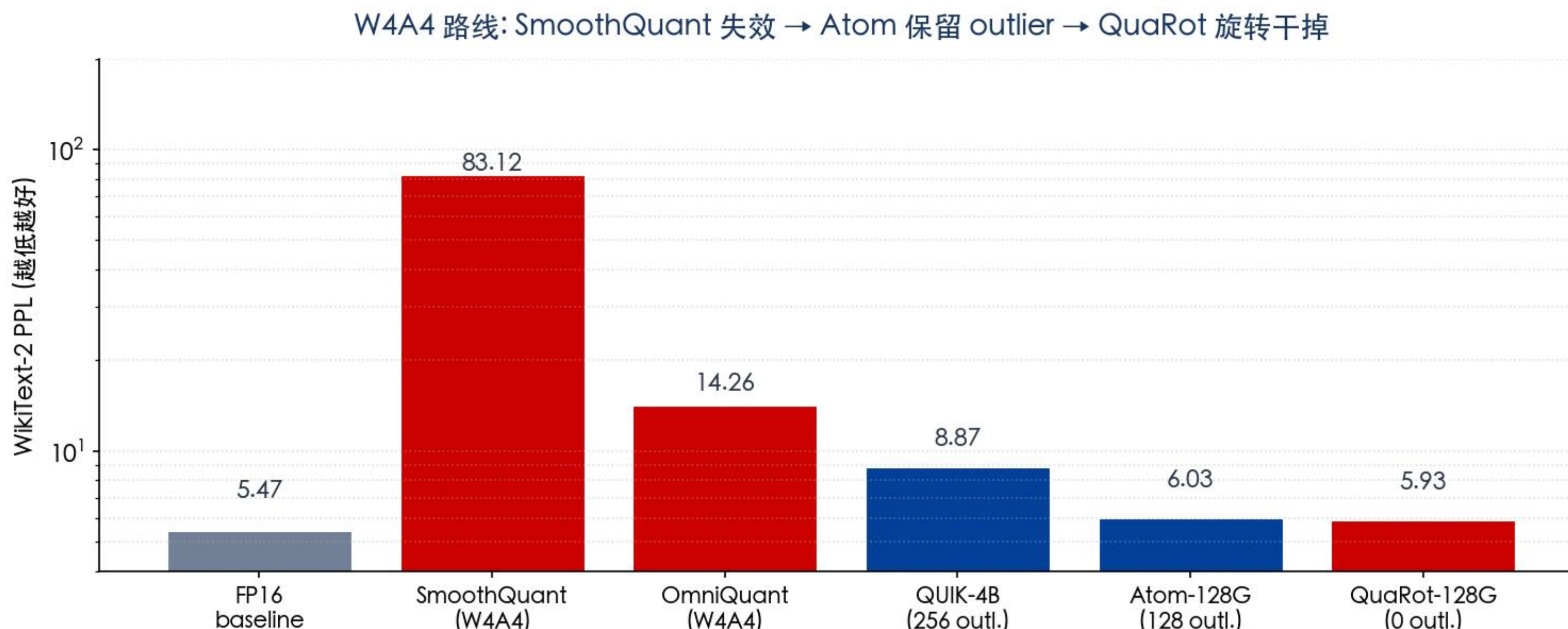
QuaRot (NeurIPS 2024, arxiv:2404.00456)

- **机制概要**: 正交 Q 满足 $Q \cdot Q^T = I$, 因此 $Y = X \cdot W = (X \cdot Q) \cdot (Q^T \cdot W)$; 选 Q = Hadamard 矩阵, 单点 outlier x 被分摊为 d 个 ±x/√d 项, max 缩小 √d 倍
- **Computational Invariance**: 在 RMSNorm pre-norm 结构中, Q 可吸收进相邻 LayerNorm γ 与权重 W (来自 SliceGPT, ICLR 2024); 推理时仅余在线 Hadamard, O(d log d) 远小于 Linear 的 O(d²)
- **结果**: Llama-2-7B W4A4 PPL 5.93, 接近 FP16 5.47 (Table 1); 是首个让 W4A4 在 LLM 上可用的方法

Ashkboos et al., QuaRot, NeurIPS 2024 (arxiv:2404.00456), §3.1-3.4 + Fig 1; SliceGPT, ICLR 2024

W4A4 演化: 五个方法的精度对比

演化路线: SmoothQuant 失效 → Atom 保留 outlier → QuaRot 旋转干掉



读图要点 + 趋势可推广

- SmoothQuant W4A4 = 83.12 (失效): diagonal scale 表达力不够
- Atom = 6.03 (keep + 8-bit outlier), QuaRot = 5.93 (orthogonal): 都接近 FP16 5.47
- 趋势 = QuaRot 在 Llama-3-8B / Qwen3-7B 上验证一致, 论文 + GitHub repo 给出对应数字

QuaRot Table 1 (Llama-2-7B); Llama-3 / Qwen3 的延伸数据见 QuaRot GitHub README

PART D

QAT 低比特

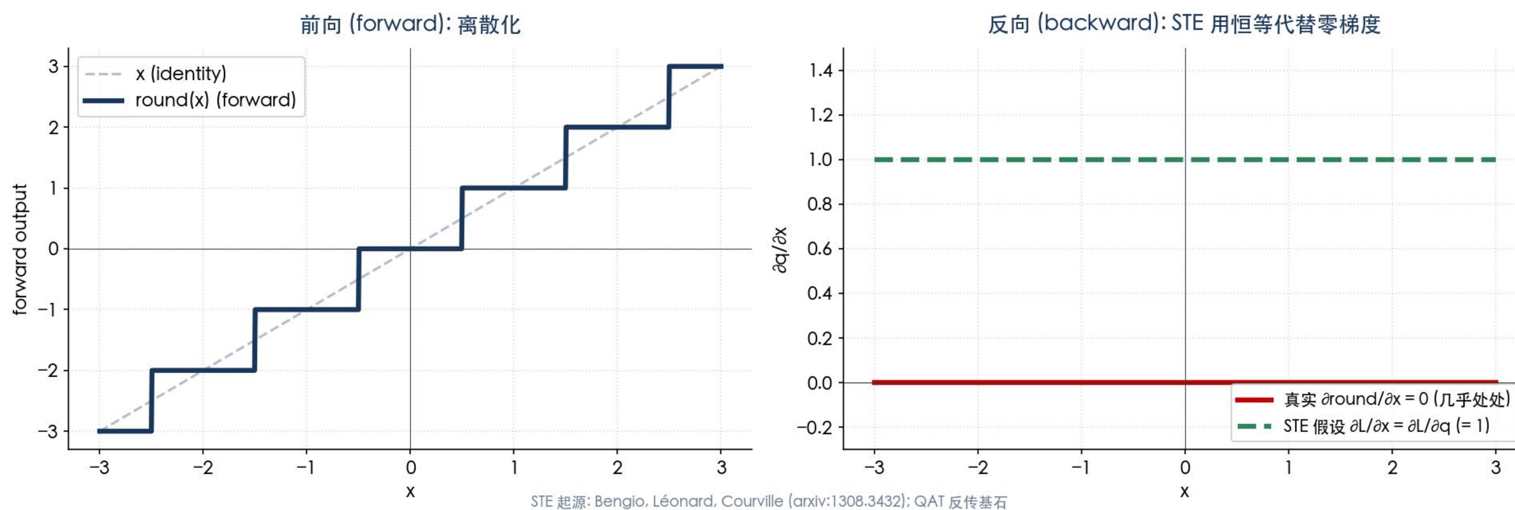
Q4a: PTQ 之外, 训练时量化能走到多低?

本课进度

√ Part A	√ Part B	√ Part C	Part D	Part E	Part F
误差的数学结构	GPTQ 最优 PTQ	outlier 跳出框架	QAT 低比特	KV cache	选型与总结

QAT 框架 + 低比特案例 (BitNet b1.58)

主线: 离散算子不可导 → STE 救场 → QAT 才能压到 1.58 bit



STE 是 QAT 反传基石

- 前向: q = round(x)
- 反向: ∂L/∂x ≈ ∂L/∂q (skip)
- 起源: Bengio 2013
- 所有 QAT 路线都依赖
- 偏置估计但实践有效
- LSQ / LLM-QAT / BitNet 都用

BitNet b1.58: QAT 走到三值 (Ma et al. 2024, arxiv:2402.17764)

- 训练时直接用三值 {-1, 0, +1} + 8-bit 激活
- absmean 量化: W̃ = round(W / mean|W|), clip 到 {-1, 0, +1}
- 矩阵乘退化为加减法; 7nm 算术能耗 71.4× 节省 (Fig 3)
- 局限: 必须从头训, 不可转换已有 FP16 权重; 这是 QAT 路线相对 PTQ 的根本代价

STE: Bengio 2013 (arxiv:1308.3432); BitNet b1.58: Ma et al. 2024; LSQ/LLM-QAT 见参考列表

PART E

KV cache 量化

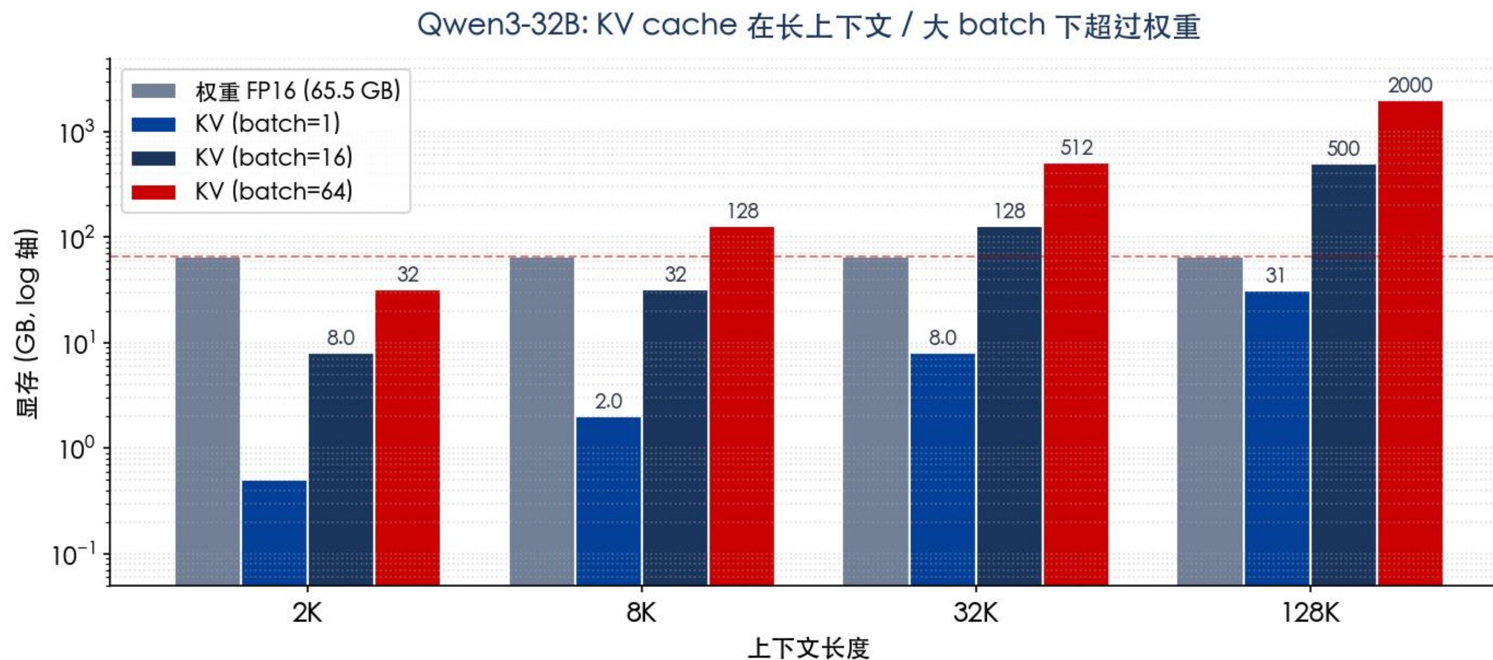
Q4b: 长上下文下, 权重量化解决后还剩什么访存瓶颈?

本课进度

√ Part A	√ Part B	√ Part C	√ Part D	Part E	Part F
误差的数学结构	GPTQ 最优 PTQ	outlier 跳出框架	QAT 低比特	KV cache	选型与总结

KV cache 显存计算 (Qwen3-32B + GQA)

关键事实: 长上下文下 KV cache 显存占用远超权重, 成为新的访存瓶颈



来源: Qwen3-32B 架构 (64L, 8 KV heads GQA, head_dim 128, 实测于 /data/models); KV per token = 2· L· H_kv· d· 2 bytes

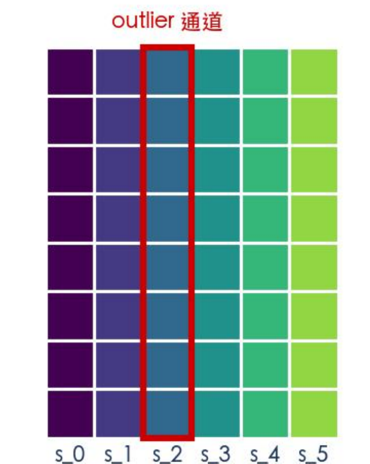
KV cache 大小公式 (per token)

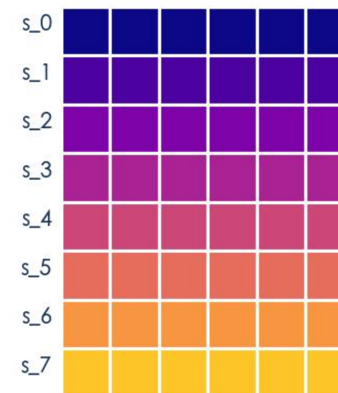
- KV per token = 2 (K+V) × L 层 × H_kv × d_head × 2 bytes (FP16)
- Qwen3-32B (实测于 /data/models): 64 layers, 8 KV heads (GQA), head_dim 128 → 0.25 MB / token
- 128k 上下文 + batch=64 → KV ≈ 2 TB ≫ 权重 65.5 GB; 这就是为什么 KV 量化重要

架构数字来自 /data/models/Qwen3-32B/config.json 实测

KIVI: 不同的切法对应不同的 outlier 模式

关键洞察: K 与 V 的 outlier 结构不同, 切法也应不同





来源: Liu et al., KIVI, ICML 2024 (arxiv:2402.02750), Fig 1 + §3

K cache: per-channel

- 存在固定 outlier 通道
- per-channel 把误差**锁定在那几列**
- 不污染其他通道
- 流式实现: 每 G=32 token 量化一组

V cache: per-token

- 没有固定 outlier 通道
- attention 是按 token 稀疏加权
- per-token 限制误差**到每个 token 内**
- 不污染重要 token 的输出

Liu et al., KIVI, ICML 2024 (arxiv:2402.02750), §3 + Table 1

PART F

选型与全章总结

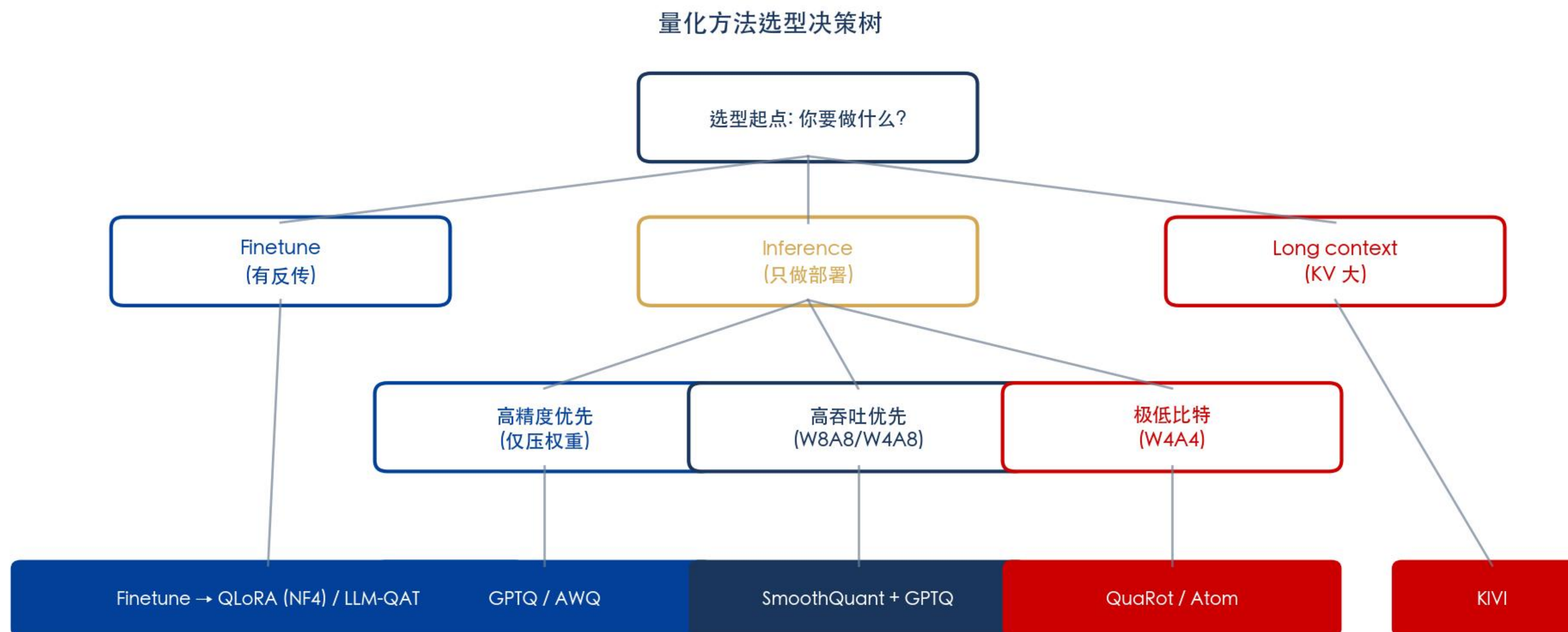
Q4c: 将本章所有方法对照该框架, 如何选型?

本课进度

√ Part A	√ Part B	√ Part C	√ Part D	√ Part E	Part F
误差的数学结构	GPTQ 最优 PTQ	outlier 跳出框架	QAT 低比特	KV cache	选型与总结

选型决策树: 从误差源到方法到应用场景

收回主线: 整章每个方法 = 处理一类误差 + 一个应用场景



实践建议 (从最稳到最激进)

- 顺序: 先用 **AWQ INT4 g=128** 评估精度 → 不达标改 **GPTQ + SmoothQuant** → 仍不足时使用 **QuaRot**
- 长上下文场景: 不论权重方案, 都该搭 **KIVI** 处理 KV (内存收益独立)
- finetune 场景: **QLoRA NF4** 是工业默认; 训练新模型考虑 BitNet b1.58

全章 Takeaway: 三个变量 + 一棵问题树

记住这张总结: 量化的工程师该带着这个心智模型走

变量 1: 比特数

- INT8 / FP8: 几乎无损
- INT4: 主流 (PTQ)
- W4A4: 需要 outlier 处理
- ⩽ 2 bit: 必须 QAT

变量 2: 粒度

- per-tensor: 已淘汰
- per-channel: 权重默认
- per-group=128: AWQ/GPTQ
- per-token: 激活动态量化

变量 3: 是否需训练

- PTQ: 不动训练, 几小时
- QAT: 反传, 数天-数周
- ⩽ 2 bit 才必须 QAT
- finetune 场景用 QLoRA

延伸阅读 (本课只点了名字, 论文细节自查)

- LSQ (ICLR 2020, arxiv:1902.08153): 学习 step size 的经典 QAT 方法
- LLM-QAT (2023, arxiv:2305.17888): data-free distillation 把 QAT 搬到 LLM
- OmniQuant (ICLR 2024, arxiv:2308.13137): 学习 clip + 学习 scale, W4A16 强 baseline
- SpinQuant (ICLR 2025, arxiv:2405.16406): QuaRot 的可学习旋转版本
- QLoRA (NeurIPS 2023, arxiv:2305.14314): NF4 + LoRA, 工业 finetune 默认

L1 + L2 + L3 = 概念 + 实战 + 数学, 这就是工程师的量化知识地图