

解码与部署

Fangyue Zhu

Wuhan University

November 28, 2024



当完成训练后，我们就可以将大语言模型部署到真实场景中进行使用。大语言模型是通过文本生成的方式进行工作的。在自回归架构中，模型针对输入内容（prompt）逐个单词生成输出内容的文本。这个过程一般被称为 **解码**。

大语言模型的解码本质上是一个概率采样过程，需要合适的解码策略来生成合适的输出内容。我们将首先介绍常见的解码策略以及相应的优化加速算法，然后介绍对大语言模型大小进行压缩以适应低资源场景部署。

① 解码策略

- 背景知识
- 两种解码策略及其改进
- 使用效果示例

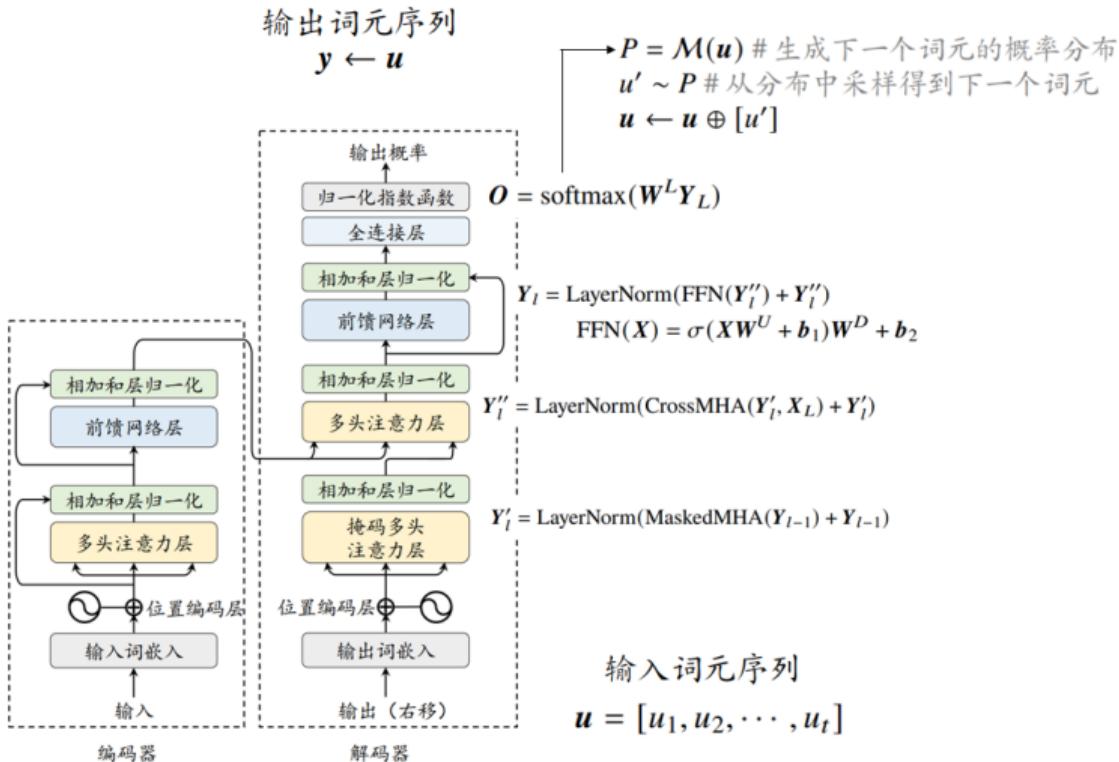
② 解码加速算法

- 解码效率分析
- 系统层面优化
- 解码策略优化

③ 低资源部署策略

- 模型量化
- 模型蒸馏
- 模型剪枝

语言模型解码的背景知识



① 解码策略

- 背景知识
- 两种解码策略及其改进
- 使用效果示例

② 解码加速算法

- 解码效率分析
- 系统层面优化
- 解码策略优化

③ 低资源部署策略

- 模型量化
- 模型蒸馏
- 模型剪枝

目前常见的大语言模型主要是通过语言建模任务进行预训练的，在训练过程中，模型通常根据以下的似然函数进行优化：

$$\mathcal{L}_{LM}(\mathbf{u}) = \sum_{t=1}^T \log P(u_t | \mathbf{u}_{<t}) \quad (1)$$

基于这种训练方式，一种直观的解码策略是**贪心搜索**，即在每个生成步骤中都选择概率最高的词元，其可以描述为以下形式：

$$u_i = \operatorname{argmax}_u P(u | \mathbf{u}_{<i}) \quad (2)$$

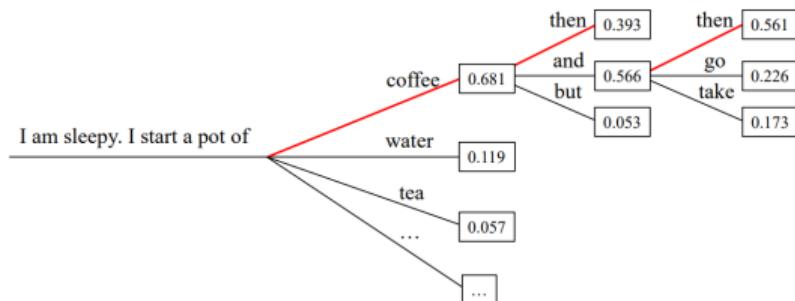


图 9.1 贪心搜索示意图

除了贪心搜索外，另一种可选的解码策略是**概率采样**，该方法根据模型建模的概率分布采样得到下一个词元，旨在增强生成过程中的随机性和结果的多样性：

$$u_i \sim P(u|\mathbf{u}_{*}) \quad (3)*$$

在图中展示了下一个词元的概率分布，虽然单词“coffee”被选中的概率较高，但基于采样的策略同时也为选择其他单词（如“water”、“tea”等）留有一定的可能性，从而增加了生成文本的多样性和随机性。

I am sleepy. I start a pot of _____					
coffee	0.681	strong	0.008	soup	0.005
water	0.119	black	0.008
tea	0.057	hot	0.007	happy	4.3e-6
rice	0.017	oat	0.006	Boh	4.3e-6
chai	0.012	beans	0.006

图 9.2 “I am sleepy. I start a pot of” 语境中下一个词元的降序排列概率分布

由于贪心搜索所采取的是确定性策略，它的效果在不同类型的任务中具有一定的差异。在机器翻译和文本摘要等任务中，任务输出高度依赖于输入内容，贪心搜索通常能够获得不错的结果，但是在开放式生成任务（如故事生成和对话系统）中，贪心搜索有时会因为过于关注局部最优，而生成不自然、重复的句子。为了解决这个问题，可以进一步采用以下的改进策略：

由于贪心搜索所采取的是确定性策略，它的效果在不同类型的任务中具有一定的差异。在机器翻译和文本摘要等任务中，任务输出高度依赖于输入内容，贪心搜索通常能够获得不错的结果，但是在开放式生成任务（如故事生成和对话系统）中，贪心搜索有时会因为过于关注局部最优，而生成不自然、重复的句子。为了解决这个问题，可以进一步采用以下的改进策略：

- ▶ **束搜索：**在解码过程中，束搜索会保留前 n 个具有最高概率的句子，并最终选取整体概率最高的生成回复。这里的 n 被称为束大小。当 $n = 1$ ，束搜索就退化为贪心搜索。在实践中，束的数量通常设定在 3 到 6 的范围内，设置过大的束会显著增加运算开销，并可能会导致性能下降。

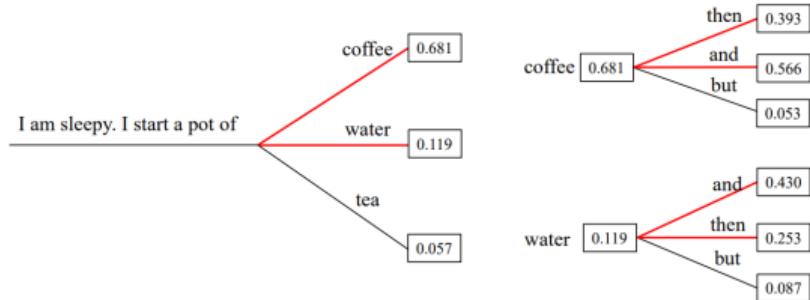


图 9.3 束搜索示意图 ($n=2$)

- ▶ **长度惩罚：**由于束搜索中需要比较不同长度候选句子的概率，往往需要引入长度惩罚（亦称为**长度归一化**）技术。否则束搜索会倾向于生成较短的句子。通过将句子概率除以其长度的指数幂 α ，对于句子概率进行归一化处理，从而鼓励模型生成更长的句子。在实践中， α 通常设置为0.6到0.7之间的数值。

- ▶ **长度惩罚：**由于束搜索中需要比较不同长度候选句子的概率，往往需要引入长度惩罚（亦称为**长度归一化**）技术。否则束搜索会倾向于生成较短的句子。通过将句子概率除以其长度的指数幂 α ，对于句子概率进行归一化处理，从而鼓励模型生成更长的句子。在实践中， α 通常设置为 0.6 到 0.7 之间的数值。
- ▶ **重复惩罚：**为了缓解贪心搜索重复生成的问题，可以使用 n -元惩罚来强制避免生成重复的连续 n 个词元，实践中 n 通常设置为 3 到 5 之间的整数。进一步地，研究人员还提出了相对“温和”的惩罚机制来降低生成重复词元的概率，而不是“一刀切”地完全避免某些短语的生成，如出现惩罚和频率惩罚。具体地，出现惩罚在生成过程中会将已经生成词元的 logits ($\mathbf{O} = \text{softmax}(\mathbf{W}^L \mathbf{Y}_L)$ 中的 $\mathbf{W}^L \mathbf{Y}_L$) 减去惩罚项 α 来降低该词元之后生成的概率。频率惩罚相较于出现惩罚，会记录每个词元生成的数目，然后减去出现次数乘以惩罚项 α ，因此如果一个词元生成得越多，惩罚也就越大。在实践中， α 的取值范围通常在 0.1 到 1 之间。这些重复惩罚方法不止适用于贪心搜索，对于随机采样也均适用。

基于概率采样的方法会在整个词表中选择词元，这可能会导致生成不相干的词元。为了进一步提高生成质量，可以进一步使用一些改进的采样策略，减少具有极低概率词汇对于生成结果的影响：

基于概率采样的方法会在整个词表中选择词元，这可能会导致生成不相干的词元。为了进一步提高生成质量，可以进一步使用一些改进的采样策略，减少具有极低概率词汇对于生成结果的影响：

- ▶ **温度采样**：为了调节采样过程中的随机性，一种有效的方法是调整 softmax 函数中的温度系数。具体来说， $\mathbf{l} = \mathbf{W}^L \mathbf{y}_L$ 称为 logits，调整温度系数后的 softmax 计算式如下：

$$P(u_j | \mathbf{u}_{<j}) = \frac{\exp(l_j/t)}{\sum_{j'} \exp(l_{j'}/t)} \quad (4)$$

当温度系数 t 设置为 1 时，该公式退化为标准的随机采样方法；而当 t 趋近于 0 时，实际上等同于贪心搜索，即总是选择概率最高的词。此外，当 t 趋近于无穷大时，温度采样会退化为均匀采样。

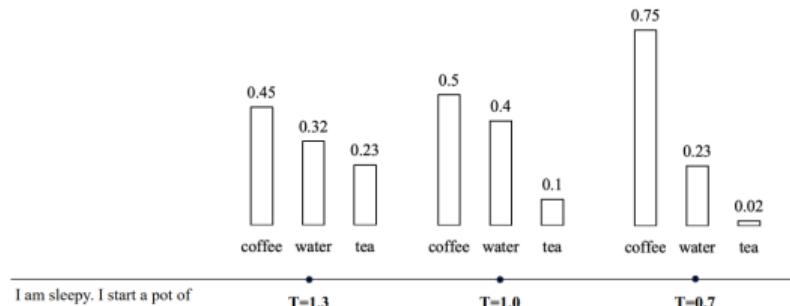
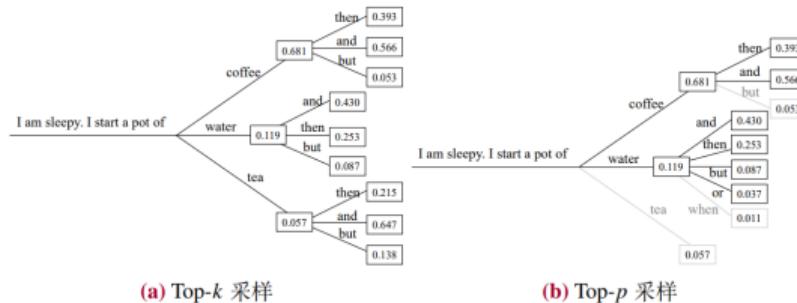


图 9.4 温度设置为 1.3、1.0 和 0.7 时的下一个词的概率分布变化

- ▶ **Top-k 采样**: 与温度采样不同, Top-k 采样策略是直接剔除概率较低的词元, 限制模型从概率最高的前 k 个词元中进行采样;
- ▶ **Top-p 采样**: 由于 Top-k 采样策略并不考虑整体概率分布, 因此固定的常数 k 可能无法适应不同的上下文语境, Top-p 采样是对这种情况的改进。从一个符合特定概率条件的最小词元集合中进行采样, 要求其中包含的所有词元累积概率大于或等于预设阈值 p 。在具体的实现过程中, 首先会按照生成概率从高到低的顺序对词元进行排序, 然后不断将词元添加到一个临时的集合中, 直到集合的累积概率首次超过阈值 p 。

图 9.5 Top- k 采样和 Top- p 采样示意图

- ▶ **对比解码：**由于大模型比小模型具有更强的生成能力，因而在预测下一个词元时，大语言模型相较于小模型更倾向于为重要词元分配更高的概率。基于这个想法，对比解码通过计算一个较大的语言模型（例如 GPT-2 XL）和一个较小的语言模型（例如 GPT-2 small）之间的对数概率分布差值，然后基于归一化的差值分布采样下一个词元，从而有效地提升重要词元在生成过程中的影响力。例如在预测一个给定片段“李时珍是湖北人，他出生于 ()”的下一个词时，GPT-2 XL 有 15% 的概率生成“湖北”、10% 的概率生成“明朝”，而 GPT-2 small 有 10% 的概率生成“湖北”、0.1% 的概率生成“明朝”，可以看到虽然 GPT-2 XL 生成“湖北”的概率仍然最高，但是其生成“明朝”的概率大幅增长，对比解码可以有效利用这一现象，在解码过程中提升重要词汇的影响力。

① 解码策略

- 背景知识
- 两种解码策略及其改进
- 使用效果示例

② 解码加速算法

- 解码效率分析
- 系统层面优化
- 解码策略优化

③ 低资源部署策略

- 模型量化
- 模型蒸馏
- 模型剪枝

用户输入：人类的正常体温大约是多少摄氏度？

贪心搜索的回复

模型输出：人类正常体表温度大约在 37 摄氏度（98.6 华氏度）左右。然而，体温可能因个体差异、环境条件以及测量方式的不同而有所变化。正常情况下，人体内部的核心温度保持相对稳定，而体表温度则会因外部环境和身体活动等因素而有所波动。因此，37 摄氏度通常被视为人类正常体温的参考值，但实际体温可能会有轻微的变化。

贪心搜索和频率惩罚 ($\alpha = 1$) 的回复

模型输出：人类正常体表温度大约在 37 摄氏度（98.6 华氏度）左右。然而，体温可能因个体差异、环境条件以及测量方式的不同而有所变化。由于人类体内有调节体温的机制，因此正常情况下人体内部温度保持相对稳定，不会出现剧烈波动。

Top- p 采样 ($p = 0.95$) 的回复

模型输出：人类正常体表温度大约在 36.5 到 37.5 摄氏度之间。然而，人体内部正常腋下体温通常在 36.5 到 37.2 摄氏度之间，而直肠温度则通常比腋下体温略高 0.2 到 0.4 摄氏度。健康成人的正常体表温度通常是指皮肤表面的温度，而正常腋下温度则反映了体内部分的温度变化。在医学检查中，腋下体温通常被认为比口腔和直肠体温更准确，因为体温测量方式不同。

① 解码策略

- 背景知识
- 两种解码策略及其改进
- 使用效果示例

② 解码加速算法

- 解码效率分析
- 系统层面优化
- 解码策略优化

③ 低资源部署策略

- 模型量化
- 模型蒸馏
- 模型剪枝

原始的自回归解码过程中，循环内相邻的两次前向传播过程中，需要进行概率分布的两次计算

$$\mathbf{P} = \mathcal{M}(\mathbf{u}); \quad \mathbf{P}' = \mathcal{M}(\mathbf{u} \oplus [u']) \quad (5)$$

这两次前向传播过程的具体计算操作

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \quad \mathbf{Q}' = [\mathbf{X} \oplus [u']] \mathbf{W}^{\mathbf{Q}} = \mathbf{X}\mathbf{W}^{\mathbf{Q}} \oplus [u']\mathbf{W}^{\mathbf{Q}} = \mathbf{Q} \oplus (\mathbf{u}'\mathbf{W}^{\mathbf{Q}}) \quad (6)$$

只需要额外计算一次新生成词元 \mathbf{u}' 相关的状态即可。 \mathbf{K} 、 \mathbf{V} 矩阵和 $\mathbf{FFN} = \mathbf{\sigma}(\mathbf{X}\mathbf{W}^{\mathbf{U}} + \mathbf{b}_1)\mathbf{W}^{\mathbf{D}} + \mathbf{b}_2$ 同理。

注意力计算的公式则有所不同

$$\text{Attention}(\mathbf{Q}', \mathbf{K}', \mathbf{V}') = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}'^T}{\sqrt{\mathbf{D}}}\right)\mathbf{V} \oplus \text{softmax}\left(\frac{(\mathbf{u}'\mathbf{W}^{\mathbf{Q}})\mathbf{K}'^T}{\sqrt{\mathbf{D}}}\right)\mathbf{V}' \quad (7)$$

其中 $\mathbf{K}' = \mathbf{K} \oplus (\mathbf{u}'\mathbf{W}^{\mathbf{K}})$, $\mathbf{V}' = \mathbf{V} \oplus (\mathbf{u}'\mathbf{W}^{\mathbf{V}})$ 。直观来说，对于新生成词元 \mathbf{u}' 而言，其需要作为查询与之前词元的键和值进行注意力计算。因此，可以通过缓存之前序列的键值状态，每次生成下一个词元时利用缓存的键值矩阵计算当前的多头注意力，这称为**键值缓存**（Key-Value Caching）优化。

总的来说，解码算法主要可以分为两个阶段：(1) 全量解码阶段，对于输入序列，一次性地计算其状态并缓存键值矩阵（算法第 1 至 3 行）；(2) 增量解码阶段，只计算上一步新生成词元的状态，并不断地以自回归方式生成新词元并对应更新键值缓存，直到生成结束（算法第 4-9 行）

算法 基于键值缓存优化的贪心解码

输入：模型 \mathcal{M} , 输入词元序列 u
输出：输出词元序列 y

- 1: $P, K_{past}, V_{past} = \mathcal{M}(u)$
- 2: $u' = \arg \max P$
- 3: $u \leftarrow u \oplus [u']$
- 4: **while** u' 不是结束词元且 u 的长度不超过预设长度 **do**
- 5: $P, K, V = \mathcal{M}(u', K_{past}, V_{past})$
- 6: $u' = \arg \max P$
- 7: $u \leftarrow u \oplus [u']$
- 8: $K_{past}, V_{past} \leftarrow K_{past} \oplus K, V_{past} \oplus V$
- 9: **end while**
- 10: $y \leftarrow u$

为了定量地进行解码效率的分析，我们首先引入 GPU 算力和 GPU 带宽这两个概念，以此来评估特定 GPU 的性能。

- ▶ **GPU 算力**: GPU 每秒能够进行的浮点运算次数，单位是 FLOP/s;
- ▶ **GPU 带宽**: 该显卡每秒能够进行的显存读写量，单位是 byte/s;
- ▶ **GPU 的计算强度上限 I_{max}** : 算力和带宽的比值，单位为 FLOP/byte.

以 A100GPU 为例，该显卡半精度浮点数的算力为 312 TFLOP/s，即每秒能进行 3.12×10^{14} 次半精度浮点数运算；同时，所对应的带宽为 2039 GB/s。通过按照上述定义进行计算，可以获得 A100 GPU 的计算强度上限约为 142.51 FLOP/byte。

同样地，模型在运行时也有相应的两个性能指标为：运算量和访存量：

- ▶ **运算量**：运行该模型需要的总浮点计算数，单位为 FLOP；
- ▶ **访存量**：运行该模型的过程中所需的显存读写量，单位为 byte；
- ▶ **模型的计算强度 I** ：运算量和访存量的比值被称为该模型的计算强度 I ，单位为 FLOP/byte.

同样地，模型在运行时也有相应的两个性能指标为：运算量和访存量：

- ▶ **运算量**：运行该模型需要的总浮点计算数，单位为 FLOP；
- ▶ **访存量**：运行该模型的过程中所需的显存读写量，单位为 byte；
- ▶ **模型的计算强度 I** ：运算量和访存量的比值被称为该模型的计算强度 I ，单位为 FLOP/byte.

比较 GPU 的计算强度上限 I_{max} 和模型的计算强度 I 可以得到模型的实际运行效率受 GPU 制约的情况：

- ▶ **带宽瓶颈**： $I < I_{max}$ ，说明 GPU 的理论最高显存读写速度低于实际运算所需速度，模型实际的运行效率将主要受到显存读写速度的影响；
- ▶ **计算瓶颈**： $I > I_{max}$ ，说明 GPU 的理论最高浮点运算速度低于实际运算所需速度，模型的运行效率将主要受到算力的影响。

以 LLaMA 模型为例，分析在全量解码和增量解码过程中的运算量和访存量并得到相应的计算强度。

矩阵 $\mathbf{A} \in \mathbb{R}^{n \times m}$ 和矩阵 $\mathbf{B} \in \mathbb{R}^{m \times p}$ 相乘

计算量: $(m + (m - 1)) \times n \times p \approx 2mnp$

读写量: 读入 \mathbf{A} 和 \mathbf{B} , 写入相乘结果 \mathbf{C} , $nm + mp + np$

全量解码运算量与训练阶段前向传播的运算量相同

表 9.1 全量解码的运算量、访存量和计算强度

计算公式	运算量	访存量	计算强度
① $\mathcal{Q}, \mathcal{K}, \mathcal{V} = XW^{\mathcal{Q}, \mathcal{K}, \mathcal{V}}$	$6BTH^2$	$O(BTH + H^2)$	$O\left(\frac{1}{\frac{1}{H} + \frac{1}{BT}}\right) \quad X \in \mathbb{R}^{B \times T \times H} \quad W^{\mathcal{Q}} \in \mathbb{R}^{H \times H} \quad H = ND$
② $\mathcal{Q}, \mathcal{K} = \text{RoPE}(\mathcal{Q}, \mathcal{K})$	$6BTH$	$O(BTH)$	$O(1) \quad \mathcal{Q}, \mathcal{K}, \mathcal{V} \in \mathbb{R}^{B \times T \times H}$
③ $\mathcal{O} = \text{Attn}(\mathcal{Q}, \mathcal{K}, \mathcal{V})$	$4BT^2ND + 4BT^2N$	$O(BT^2N + BTND)$	$O\left(\frac{1+\frac{1}{D}}{\frac{1}{D} + \frac{1}{T}}\right) \quad \mathcal{Q}, \mathcal{K}, \mathcal{V} \in \mathbb{R}^{B \times N \times T \times D} \quad \text{softmax}\left(\frac{\mathcal{Q}\mathcal{K}^\top}{\sqrt{D}}\right)\mathcal{V}$
④ $X = \mathcal{O}W^O$	$2BTH^2$	$O(BTH + H^2)$	$O\left(\frac{1}{\frac{1}{H} + \frac{1}{BT}}\right)$
⑤ $X = \text{Add\&Norm}(X)$	$5BTH$	$O(BTH + H)$	$O\left(\frac{1}{1 + \frac{1}{BT}}\right)$
⑥ $\mathcal{G}, \mathcal{U} = X[W^G, W^U]$	$4BTHH'$	$O(BTH + BTH' + HH')$	$O\left(\frac{1}{\frac{1}{B} + \frac{1}{H'} + \frac{1}{BT}}\right) \quad X \in \mathbb{R}^{B \times T \times H} \quad W^U \in \mathbb{R}^{H \times H'}$
⑦ $\mathcal{D} = \text{SiLU}(\mathcal{G}) \cdot \mathcal{U}$	$2BTH'$	$O(BTH')$	$O(1)$
⑧ $X = \mathcal{D}W^D$	$2BTHH'$	$O(BTH + BTH' + HH')$	$O\left(\frac{1}{\frac{1}{B} + \frac{1}{H'} + \frac{1}{BT}}\right)$
⑨ $X = \text{Add\&Norm}(X)$	$5BTH$	$O(BTH + H)$	$O\left(\frac{1}{1 + \frac{1}{BT}}\right)$

与全量解码相比，增量解码在大部分运算上只需要将输入词元长度视为 $T=1$ 即可，唯一不同的地方是 **多头注意力计算部分**（公式④）和额外的**键值缓存更新操作**（公式③）。因此除了这两个操作外，增量解码其他部分的运算量和访存量可以通过将全量解码公式中的 T 替换为 1 来获得。对于增量解码阶段的多头注意力计算，由于键和值依然是矩阵形式，所以其访存量变为 $O(BTN + BTND + BND)$ 。而键值缓存的更新操作，在不使用显存优化算法的情况下访存量为 $O(BTND)$ ，如果使用了例如 PagedAttention 等优化技术，其访存量可以降低为 $O(BND)$ 。

表 9.2 增量解码的运算量、访存量和计算强度

	运算量	访存量	计算强度
① $q, k, v = XW^{QKV}$	$6BH^2$	$O(BH + H^2)$	$O\left(\frac{1}{H+B}\right)$
② $q, k = \text{RoPE}(q, k)$	$6BH$	$O(BH)$	$O(1)$
③ $K, V = \text{Cache}(k, v)$	-	$O(BTND)$ 或 $O(BND)$	-
④ $o = \text{Attn}(q, K, V)$	$4BTND + 4BTN$	$O(BTN + BTND + BND)$	$O\left(\frac{1+\frac{1}{D}}{1+\frac{1}{D}+\frac{1}{r}}\right)$
⑤ $X = oW^O$	$2BH^2$	$O(BH + H^2)$	$O\left(\frac{1}{H+B}\right)$
⑥ $X = \text{Add\&Norm}(X)$	$5BH$	$O(BH + H)$	$O\left(\frac{1}{1+\frac{1}{B}}\right)$
⑦ $g, u = X[W^G, W^U]$	$4BHH'$	$O(BH + BH' + HH')$	$O\left(\frac{1}{H+H'+B}\right)$
⑧ $d = \text{SiLU}(g) \cdot u$	$2BH'$	$O(BH')$	$O(1)$
⑨ $X = dW^D$	$2BHH'$	$O(BH + BH' + HH')$	$O\left(\frac{1}{H+H'+B}\right)$
⑩ $X = \text{Add\&Norm}(X)$	$5BH$	$O(BH + H)$	$O\left(\frac{1}{1+\frac{1}{B}}\right)$

基于上述分析，可以获得全量解码和增量解码阶段的运算量和访存量，进而能够通过计算其比值得到每个操作的计算强度 I 和 GPU 的计算强度上限 I_{max} ，从而更好地发现解码过程中的瓶颈操作。

这里，将以 LLaMA (7B) 模型为例进行说明（其中 $N = 32, D = 128, H = 4096$ ）。在全量解码阶段，假设批次大小为 8，序列长度为 1024（即 $B = 8, T = 1024$ ），将这些具体数值代入到全量解码的计算强度的公式中可以计算得到，各个线性变换（公式①④⑥⑦）的计算强度大约为 2730.67，多头注意力（公式③）的计算强度大约为 114.67，而其余操作（公式②⑤⑧⑩）的计算强度都在 1 左右。在使用 A100 (80G) 的 GPU 时（计算强度上限 I_{max} 为 142.51），各个线性变换和多头注意力部分的计算强度都高于或接近这个值，考虑到这些操作占据了全量解码的绝大多数运算，可以说全量解码阶段是受限于 GPU 浮点数计算能力的（即计算瓶颈）。

类似地，将这些数值代入到增量解码的计算强度公式，可以得到在增量解码阶段各个线性变换和多头注意力的计算强度都不超过 8，远小于 A100 GPU 的计算强度上限 142.51，这表明增量解码阶段是受限于 GPU 显存读写速度的（即显存瓶颈），这种问题通常被称为内存墙（Memory Wall）问题。基于上述分析，可以看到解码阶段的低效问题主要出现在增量解码阶段，接下来将从系统优化和解码策略优化两个维度来介绍增量解码阶段的改进方法。

① 解码策略

- 背景知识
- 两种解码策略及其改进
- 使用效果示例

② 解码加速算法

- 解码效率分析
- 系统层面优化
- 解码策略优化

③ 低资源部署策略

- 模型量化
- 模型蒸馏
- 模型剪枝

针对“内存墙”问题，一个直观的解决方案是减少相关操作的访存量，从而达到提升计算强度的目的，即系统级优化方法。

FlashAttention 是一种针对原始注意力模块的优化方案，可以大幅减少注意力计算中的访存量，从而提升计算强度。它的核心思路是尽可能减少对于中间结果的保存，进而直接得到最终结果。根据注意力的计算方法 $\text{softmax}\left(\frac{QK^T}{\sqrt{D}}\right)V$ ，可以发现其中需要保留多个中间结果，如 QK^T 和 softmax 后的注意力分布矩阵。这些中间结果需要频繁写入显存，因此导致了大量的显存读写操作。而 FlashAttention 通过矩阵分块和算子融合等方法，将中间结果一直保留在缓存中，直到获得最终结果后再写回显存中，从而减少了显存读写量。

FlashAttention 有效地减少了访存量，同时也降低了峰值显存的占用量。例如，使用了 FlashAttention 的 LLaMA-2 (7B) 在序列长度为 2,048、批次大小为 8 的情况下，注意力操作的时间仅需传统注意力的十分之一。

PagedAttention 是针对键值缓存拼接和注意力计算的优化操作，能够有效降低这两个运算部分的访存量，从而提高计算效率。在键值缓存拼接操作中，传统的实现方法会在每次拼接时新分配一段显存空间，然后拷贝原始键值缓存和新生成词元的状态到新分配的显存中去，这容易导致反复的显存读写，并且产生较多的显存碎片。为此，PagedAttention 引入了操作系统中显存分页管理的方法，预先将显存划分成若干块给之后的键值缓存“预留空间”，从而显著减少了拼接时反复分配显存的操作。

此外，PagedAttention 还优化了注意力计算操作，提高了计算的并行度从而减少其访存量。具体来说，增量解码阶段是以当前词元作为查询向量，与之前序列的键值缓存进行注意力计算的过程。考虑到键值缓存通常会较长，PagedAttention 采用了上述的分页管理操作，并使用算子融合的方法将查询向量与多个分页的键值缓存并行地进行计算，从而提升了计算效率。

在传统的解码操作中，通常会等待一整个批次的所有实例都结束后再进行下一个批次的计算。然而，一个批次内的不同实例往往生成长度各异，因此经常会出现等待某一条实例（输出长度最长的实例）生成的情况。在计算批次大小 B 较小时，计算强度很低，因此解码效率低下。

为了解决这个问题，批次管理优化旨在通过增加计算中的批次大小来提高计算强度。一个代表性的方法是 vLLM 所提出的**连续批处理**（Continuous Batching）技术。该技术不同于传统确定顺序的定长批次处理方式，而是将每个输入实例视为一个请求，每个请求的处理过程可以分解为全量解码阶段和若干个单步增量解码阶段。在实现中，连续批处理技术会通过启发式算法来**选择部分请求进行全量解码操作**，或者**选择一些请求进行单步增量解码操作**。通过这样细粒度的拆分，连续批处理技术**在一步操作中能够容纳更多的请求**（相当于提高批次大小），从而提升了计算强度。进一步，DeepSpeed-MII 提出了动态分割技术，将全量解码部分进一步**拆分为多个子操作**，其可以在一次计算中选择一些请求同时进行全量解码和增量解码操作，进而获得更大的批次和更高的解码吞吐量。

通过批次管理优化技术，模型可以更好地适配线上大模型应用服务。例如，ChatGPT 的网页服务端通常会面临着大规模的用户请求，线上部署模型需要尽快地将生成结果返回给用户。传统的批次生成方法会带来很高的服务时延（新请求必须等待前一个完成），而批次管理技术可以细粒度地**分割不同请求的处理阶段**，使得不同请求的处理过程可以同时进行处理，从而实现更为高效的线上服务。

① 解码策略

- 背景知识
- 两种解码策略及其改进
- 使用效果示例

② 解码加速算法

- 解码效率分析
- 系统层面优化
- 解码策略优化

③ 低资源部署策略

- 模型量化
- 模型蒸馏
- 模型剪枝

推测解码提出首先使用**相对较小但解码更为高效的模型**（例如 n 元统计模型或者小型预训练模型）自回归地生成若干个词元，然后再由大模型对这个片段进行一次验证（大模型一次验证与一次解码的时间消耗相当），来判断是否每个词元都是当前步骤概率最高的输出，随后大小模型持续迭代此过程直到解码结束。推测解码不会降低大模型解码的质量，实验测试表明能够带来约两倍左右的解码加速，是目前使用较多的解码策略优化方案。

举例来说，假设输入为“我与父亲不相见已二年余了，我最不能”，大模型的输出为“忘记的是他的背影”，总共要生成 8 个单词（为了讲解方便以单词为基础单位），需要 8 次迭代的生成过程。如果使用推测解码，第一步先使用小模型生成 3 个单词，假设结果为“忘记他”，然后使用大模型进行验证，由于前两个词正确第三个词错误，则同时将其修正为“忘记的”；第二步，再使用小模型生成 3 个单词，假设结果为“日子是”，大模型验证第一个词便发现错误，同时将其修正为“他”；第三步，小模型生成“的背影”，大模型验证均正确，生成结束。回顾上述生成过程，小模型共计生成了 9 次，大模型验证了 3 次（验证同时可以修正错误），相较于大模型的 8 次生成过程可以有一定的效率提升。

与推测解码有类似的想法，级联解码考虑到不同请求的难易度不同，分别使用**不同规模的模型来处理请求**，从而实现最小化解码时间的效果。FrugalGPT引入了一系列模型（按照效率从高到低），然后将一个请求依次给这些模型进行生成，随后引入了一个专门训练的二分类模型来判断模型的生成结果是否符合任务要求。**如果结果可靠就不需要再给之后的模型进行生成**，以实现提升解码效率的目的。该策略也可以应用于不同的开源模型、商业 API 等，用户可以根据自己的需求设定分类器的判断阈值，从而平衡生成效率与生成质量。

现有解码策略普遍采用自回归的解码机制，即逐个词元进行生成，这是导致解码效率低下的重要原因。因此，机器翻译领域的研究人员提出了非自回归的解码机制，可以基于输入并行地一次性生成所有的词元。但是这种方法的生成质量往往与自回归方法有一定差距，因此有研究工作尝试结合这两种方法，进一步提出了**半自回归解码**，每一次生成一组词元（例如 3 至 10 个词元），再以这些词元作为输入继续生成下一组。然而，现有的大模型都是预测下一个词进行预训练的，无法直接进行非（半）自回归生成。为了解决这个问题，Medusa 在 Vicuna 模型的基础上，额外训练了两个预测头来分别预测第二个词和第三个词，因此可以达到一次生成三个词元的效果。但需注意的是，尽管这些非（半）自回归策略在效率上有所提升，但仍然不能达到自回归解码的效果。因此其**很少单独使用**，通常可以用于**推测解码中的候选片段生成**，进而加速大模型的解码流程。例如 Medusa 预测片段之后，需要原始 Vicuna 模型进行验证来保证生成质量。

有研究工作发现，在多层 Transformer 模型中，**可能不需要经过所有层的计算**，模型就可以较为可靠地预测下一个词的生成。基于这种想法，研究人员提出了基于早退机制的生成方式。在模型的解码过程中，可以通过设置相应的早退判断条件。当早退条件满足时结束网络层的前向传递，直接生成相关的词元，从而提升解码效率。早期一种常见的早退判断方法是对于 Transformer 每一层的输出都使用预测头得到在词表上的概率分布，然后计算该分布的熵。如果熵值小于预定义的阈值（即某一个词的概率显著较高），则可以判断为早退，不进行后续层的计算。在实现中，可以通过调整该阈值来平衡解码效率与生成质量。最近的研究工作提出了**混合深度方法**，来动态调整每一层的计算量。混合深度方法对于每一层的输入通过路由网络计算得分，如果该得分高于预先设定的阈值则进行该层的后续计算，否则直接跳过该层的计算。与传统早退机制直接跳过后续所有层计算相比，混合深度方法有选择性的跳过了部分层，因此可以更好地利用模型中不同层的特性。

① 解码策略

- 背景知识
- 两种解码策略及其改进
- 使用效果示例

② 解码加速算法

- 解码效率分析
- 系统层面优化
- 解码策略优化

③ 低资源部署策略

- 模型量化
- 模型蒸馏
- 模型剪枝

模型量化

由于大模型的参数量巨大，在解码阶段需要占用大量的显存资源，因而在实际应用中的部署代价非常高。一种常用的模型压缩方法是**模型量化**，能够减少大模型的显存占用，从而使得能够在资源有限的环境下使用大模型。

在神经网络压缩中，**量化**通常是指从浮点数到整数的映射过程，目前比较常用的是 8 比特整数量化，即 INT8 量化。针对神经网络模型，通常有两种类型的数据需要进行量化，分别为**权重量化**（也称为**模型参数量化**）和**激活（值）量化**，它们都以浮点数形式进行表示与存储。

量化的过程可以表示为一个函数，该函数将连续的输入映射到离散的输出集合。一般来说，这个过程涉及到四舍五入或截断等近似操作。下面介绍一个一般形式的量化函数：

$$x_q = R(x/S) - Z \quad (8)$$

通过上述数学变换，量化算法将浮点数向量 x 转化为量化值 x_q 。其中， S 表示缩放因子，用于确定裁剪范围， Z 表示零点因子，用于确定对称或非对称量化， $R(\cdot)$ 表示将缩放后的浮点值映射为近似整数的取整操作。一般来说，裁剪范围对于量化性能有很大影响，通常需要根据实际数据分布进行校准，可以通过静态（离线）或动态（运行时）方式。

作为上述变换的逆过程，反量化（Dequantization）对应地从量化值中恢复原始值，该过程首先加上零点因子，然后乘以缩放因子：

$$\tilde{x} = S \cdot (x_q + Z) \quad (9)$$

进一步，可以定义量化误差是原始值 x 和恢复值 \tilde{x} 之间的数值差异： $\Delta = ||x - \tilde{x}||_2^2$

基于上述量化函数的定义形式，下面介绍一些对于量化函数常见的分类与实现策略

基于上述量化函数的定义形式，下面介绍一些对于量化函数常见的分类与实现策略

- ▶ **均匀量化和非均匀量化：**根据映射函数的数值范围是否均匀，可以将量化分为两类：均匀量化和非均匀量化。均匀量化是指在量化过程中，量化函数产生的量化值之间的间距是均匀分布的。这种方法通常用于将连续的数据转换为离散的表示，以便在计算机中进行高效处理。与此不同，在非均匀量化方法中，它的量化值不一定均匀分布，可以根据输入数据的分布范围而进行调整。其中，**均匀量化**方法因其简单和高效的特点而在实际中被广泛使用。

基于上述量化函数的定义形式，下面介绍一些对于量化函数常见的分类与实现策略

- ▶ **均匀量化和非均匀量化：**根据映射函数的数值范围是否均匀，可以将量化分为两类：均匀量化和非均匀量化。均匀量化是指在量化过程中，量化函数产生的量化值之间的间距是均匀分布的。这种方法通常用于将连续的数据转换为离散的表示，以便在计算机中进行高效处理。与此不同，在非均匀量化方法中，它的量化值不一定均匀分布，可以根据输入数据的分布范围而进行调整。其中，**均匀量化**方法因其简单和高效的特点而在实际中被广泛使用。
- ▶ **量化粒度的选择：**量化算法通常针对一个批次的数据进行处理，其中**批次的规模大小**就反应了量化粒度，可以由算法设计人员进行选择。在神经网络模型中，输入数据和模型权重通常是以张量的形式进行表示与组织的。首先，如果每个张量只定义一组量化参数（即 S 和 Z ），这称为**按张量量化**。为了进一步提高量化的精度，可以针对每个张量定义多组量化参数，例如可以为权重矩阵的列维度（也称为“通道”）设置特定的量化参数，称为**按通道量化**。还有一些研究工作采用了更细粒度的量化方案，对一个通道的数值细分为多个组，即**按组的方式进行量化**。更小的量化粒度可以更好地保存原始模型的性能，但计算开销更大；按张量量化的粒度较粗，可能会引入较大的误差，但由于在硬件实现上更加简单，也是一种常见的量化粒度选择策略。实际粒度选择中需要平衡量化准确性以及额外计算开销。

► **对称量化和非对称量化**: 根据零点因子 Z ($x_q = R(x/S) - Z$) 是否为零, 均匀量化可以进一步分为两类: 对称量化 ($Z = 0$) 和非对称量化 ($Z \neq 0$)。对称量化与非对称量化的一个关键区别在于**整数区间零点的映射**, 对称量化需要确保原始输入数据中的零点 ($x = 0$) 在量化后仍然对应到整数区间的零点。而非对称量化则不同, 根据前面的公式 ($\tilde{x} = S \cdot (x_q + Z)$) 可以看出此时整数区间的零点对应到输入数值的 $S \cdot Z$ 。

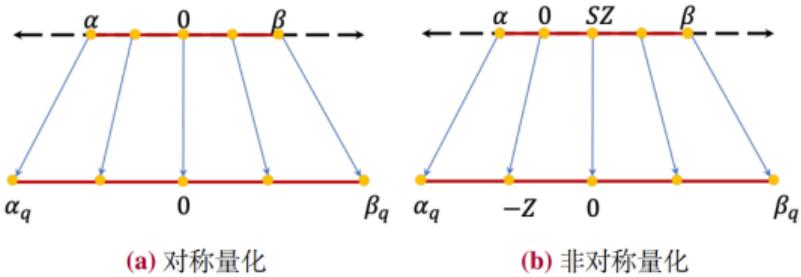


图 9.6 对称量化和非对称量化对比

以 INT8 为例, 将输入数值 x 通过一个映射公式转换为八比特整数的表示范围内, 有符号整数可以使用对称量化映射到 $[-128, 127]$, 适用于 x 的数值大致分布在零点两侧, 无符号整数可以使用非对称量化映射到 $[0, 255]$, 适用于 x 的数值集中在零点一侧。

以 INT8 为例，给定输入数据 $\mathbf{x} = [[1.2, 2.4, 3.6], [11.2, 12.4, 13.6]]$ ，希望将其量化到整数表示范围 $[-128, 127]$ 。

- ▶ **非对称量化示例：** 输入数据范围为 $\mathbf{x} \in [1.2, 13.6]$ ，求解二元一次方程组

$$1.2 = S \cdot (-128 + Z)$$

$$13.6 = S \cdot (127 + Z)$$

需要注意的是，这里缩放因子 S 是浮点数，零点因子 Z 是整数。求解得到 $S = 0.0486$, $Z = -152$ 。于是量化后的数值 $x_q = [[127, 103, 78], [78, 103, 127]]$ ，反量化后的数值 $\tilde{x} = [[1.2157, 2.3827, 3.5984], [11.1843, 12.4000, 13.5671]]$

- **对称量化示例：**取包含输入全部数据的对称区间 $\mathbf{x} \in [-13.6, 13.6]$ ，并且要求零点因子 $Z = 0$ ，得到缩放因子 $S = \frac{13.6}{(127+128)/2} = 0.1067$ 。量化后的数值 $x_q = [[11, 22, 34], [105, 116, 127]]$ ，反量化后的数值 $\tilde{x} = [[1.1733, 2.3467, 3.6267], [11.2000, 12.3733, 13.5467]]$

相较于非对称量化覆盖到的数据取值范围 $[1.2, 13.6]$ ，可以看出对称量化覆盖的范围更大。但是，由于对称量化的整型数据表示范围超出了实际数据的数值区间 $[1.2, 13.6]$ 。特别是在区间 $[13.6, 1.2)$ 范围内，没有实际数值存在，这会导致大量整型数值的浪费。此外，由于覆盖的范围更大，对称量化会引入更大的量化误差。可以看到，对称量化的数据反量化后的结果与原始结果的差异要大于非对称量化。

通常来说，模型量化方法可以分为两大类，即**量化感知训练**（Quantization-Aware Training, QAT）和**训练后量化**（Post-Training Quantization, PTQ）。其中，量化感知训练方法需要更新权重进而完成模型量化，而训练后量化方法则无需更新模型权重。

与小型语言模型相比，在为**大语言模型**设计或选择量化方法时需要侧重关注两个主要因素。首先，大语言模型由大规模神经网络参数组成，在设计量化算法时需要考虑到所需要花费的**计算开销**。一般来说，训练后量化方法需要更少的算力开销，实践中应用更为广泛。其次，大语言模型中具有**不同的数值分布特征**（如激活值中存在较大的数值），这使得对于大语言模型进行低比特量化变得非常困难，特别是对于激活值。

下面将简要介绍几种具有代表性的大语言模型的训练后量化方法。

主流的权重量化方法通常是基于逐层量化的方法进行设计的，旨在通过最小化逐层的重构损失来优化模型的量化权重，可以刻画为： $\text{argmin}_{\hat{\mathbf{W}}} \|\mathbf{X}\mathbf{W} - \hat{\mathbf{X}}\mathbf{W}\|_2^2$ ，其中 \mathbf{W} , $\hat{\mathbf{W}}$ 分别表示原始权重和量化后的权重， \mathbf{X} 为输入。

为了有效地优化该目标函数，GPTQ 的基本想法是在逐层量化的基础上，进一步将权重矩阵按照列维度分组（例如 128 个列为一组），对一个组内逐列进行量化，每列参数量化后，需要适当调整组内其他未量化的参数，以弥补当前量化造成的精度损失。在具体的实现中，GPTQ 还进一步采用了特殊设计的优化方法来加速整个过程，如延迟批次更新、Cholesky 重构等。GPTQ 可以在 3 比特或 4 比特精度下实现对于大语言模型的有效权重量化。

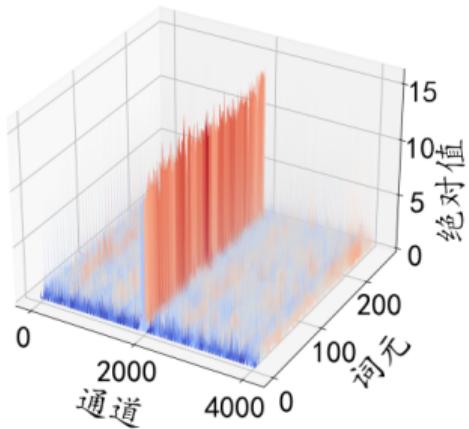
进一步，AWQ 发现在逐层和逐组权重量化过程中，对于模型性能重要的权重只占全部参数的一小部分（0.1% ~ 1%），并且应该更加关注那些与**较大激活值维度相对应的关键权重**，因为它们对应着更为重要的特征。为了加强对于这部分关键权重的关注，AWQ 方法提出引入针对权重的激活感知缩放策略。具体来说，AWQ 的优化目标将逐层的重构损失 $\|\mathbf{X}\mathbf{W} - \hat{\mathbf{X}}\mathbf{W}\|_2^2$ 修改为 $\|(diag(s)^{-1} \cdot \mathbf{X}) \cdot Q(\mathbf{W} \cdot diag(s)) - \mathbf{X}\mathbf{W}\|_2^2$ ，其中 Q 为量化函数。通过引入缩放因子 s ，AWQ 算法可以使得量化方法更为针对性地处理关键权重所对应的权重维度。

下面继续介绍一些可以同时针对权重与激活值进行量化的方法

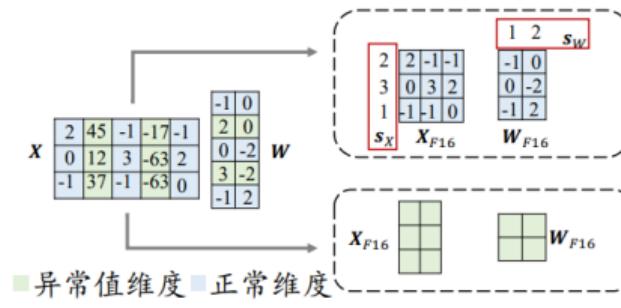
- ▶ **细粒度量化**: 对于 Transformer 模型来说, 权重与激活值通常以张量的形式表示。如之前所述, 可以使用粗粒度的方法量化, 对于每个张量使用一整套量化参数。然而, 这种粗粒度方法通常会导致不精确的数值重构, 可以使用更为细粒度的方法来减小量化误差。举例来说, 对**权重量化**, 可以从每个张量改为对每一个**通道**使用一套量化参数。对于**激活值量化**来说, 则是从每个张量改为对每个**词元**使用一套量化参数。ZeroQuant 采用了一种带有动态校准的词元级量化方法来压缩激活值, 而对于更容易量化的模型权重使用了基于分组的量化方法。在实际应用中, 常用的模型量化分组可以设置为 128。

下面继续介绍一些可以同时针对权重与激活值进行量化的方法

- ▶ **细粒度量化：**对于 Transformer 模型来说，权重与激活值通常以张量的形式表示。如之前所述，可以使用粗粒度的方法量化，对于每个张量使用一整套量化参数。然而，这种粗粒度方法通常会导致不精确的数值重构，可以使用更为细粒度的方法来减小量化误差。举例来说，对权重量化，可以从每个张量改为对每一个通道使用一套量化参数。对于激活值量化来说，则是从每个张量改为对每个词元使用一套量化参数。ZeroQuant 采用了一种带有动态校准的词元级量化方法来压缩激活值，而对于更容易量化的模型权重使用了基于分组的量化方法。在实际应用中，常用的模型量化分组可以设置为 128。
- ▶ **混合精度分解：**当模型参数规模超过一定阈值后（如 6.7B），神经网络中的激活值中会出现一些异常的超大数值，称为**异常值涌现**现象。这些异常值主要分布在 Transformer 层的某些特定激活值特征维度中。基于这一发现，在矩阵乘法中，可以将具有异常值的特征维度与其他正常维度分开计算。例如对于这两部分进行计算时分别使用 16-比特浮点数 (F_{16}) 和 8-比特整数 (I_8)，从而以较高精度地恢复这些异常值。



(a) 激活值异常值分布



(b) 混合精度分解方法

图 9.7 激活值异常值分布与混合精度分解

具体来说，对于**非异常值维度部分**，首先将激活值和权重分别量化到 8 比特整数，得到 \mathbf{X}^{I_8} 和 \mathbf{W}^{I_8} ，其量化过程如下所示：

$$127 \cdot \text{diag}(s_X)^{-1} \cdot \mathbf{X}^{F_{16}} = \mathbf{X}^{I_8}$$

$$127 \cdot \mathbf{W}^{F_{16}} \cdot \text{diag}(s_W)^{-1} = \mathbf{W}^{I_8}$$

其中， s_X 和 s_W 分别表示输入的激活值和权重中每一行/每一列中绝对值的最大值。

在量化数值 \mathbf{X}^{I_8} 和 \mathbf{W}^{I_8} 的基础上，可以进行 8 比特整数乘法操作，得到整数结果并用 32 位整数保存为 $\mathbf{O}^{I_{32}}$ ，再进行反量化，得到 16 比特浮点数结果 $\mathbf{O}^{F_{16}}$ ，具体的公式如下所示：

$$\mathbf{X}^{I_8} \mathbf{W}^{I_8} = \mathbf{O}^{I_{32}}$$

$$\frac{\mathbf{O}^{I_{32}} \odot (s_X^T s_W)}{127 \cdot 127} = \mathbf{O}^{F_{16}}$$

对于**异常值维度部分**，直接使用 16 比特浮点数乘法：

$$\mathbf{X}^{F_{16}} \mathbf{W}^{F_{16}} = \mathbf{O}^{F_{16}}$$

这两部分各自得到的结果进行相加即可得到最终的结果。

▶ **量化难度平衡**: 在模型的量化过程中, 由于激活值中的异常值问题比权重更加明显, 导致激活值往往比权重更加难量化。SmoothQuant 提出将量化难度从激活值转移到模型权重上。具体来说, 他们在线性层中引入了一个缩放变换来平衡权重和激活值之间的量化难度 $\mathbf{Y} = (\mathbf{X} \cdot \text{diag}(s)^{-1}) \cdot (\text{diag}(s) \cdot \mathbf{W})$ 。该公式通过数学上的等价变换, 引入缩放因子向量 s 来控制量化难度。为了设置 s , 该公式还引入了一个强度迁移参数 α 来平衡量化难度, 其中每个迁移系数的计算方法为 $s_j = \max(x_j)^\alpha / \max(w_j)^{(1-\alpha)}$ 。具体来说, 图 9.8 展示了这个变换的过程。这里以 $\alpha = 0.5$ 为例, 则 s 的计算为

$s_j = \sqrt{\max(x_j) / \max(w_j)}$ 。从图中看出, 变换后得到 $\hat{\mathbf{X}}$ 中异常值维度明显缓解, 而对应维度的权重 $\hat{\mathbf{W}}$ 也会受到一些影响, 但是相对于整体来说没有那么突出。

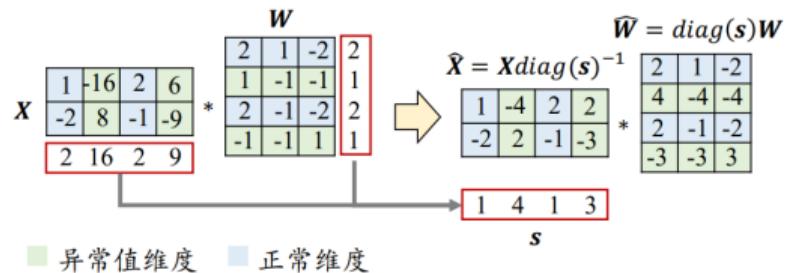


图 9.8 量化难度平衡方法

在上述内容中已经主要介绍了训练后量化方法。下面将介绍面向大模型的微调增强量化方法与量化感知训练方法。

- ▶ **高效微调增强量化：**对于大模型来说，直接进行低比特的量化（例如，INT4 量化）通常会导致性能的严重下降。为了克服这一挑战，QLoRA 在量化模型中进一步引入了额外的小型可调适配器，并且使用 16 比特精度进行训练与学习，用以补偿量化可能带来的损失。这一思路同时结合了 **LoRA 的轻量化优点与量化方法的模型压缩能力**。实验结果表明，QLoRA 可以通过 4 比特量化模型很好地保留原来 16 比特模型的微调性能。
- ▶ **面向大语言模型的量化感知训练：**由于量化感知训练方法需要引入额外的全参数学习，需要较大的计算开销，因此目前受到的研究关注还不多。最近一项研究工作测试了量化感知训练方法在大语言模型上的效果，通过教师模型生成的文本以及输出的预测分布来 **蒸馏小模型** 的方法，来压缩权重、激活值以及键值缓存。通过在开源模型 LLaMA 上进行实验，实验结果表明在权重和键值缓存上进行 4 比特量化可以获得不错的效果，但是在 4 比特激活值上的量化效果仍然需要进一步的提升。

针对大语言模型的模型量化研究受到了学术界的广泛关注，涌现了一批经验性的研究工作。下面针对学术界的研究结论进行一个简要汇总，使得读者更容易理解量化方法对于模型性能的影响以及不同量化方法的适用条件。

- ▶ **INT8 权重量化通常对于大语言模型性能的影响较小，更低精度权重量化的效果取决于具体的量化方法：**在大多数情况下，INT8 权重量化可以有效地减小显存占用而不显著影响模型性能。对于 INT4（或 INT3）权重量化，现有的方法通常使用不同策略来减少性能下降。例如，AWQ 方法采用了激活感知缩放。与小型语言模型不同的是，低比特权重量化对于大语言模型的影响通常较小。因此，在实际使用中，在相同显存开销的情况下，建议优先使用参数规模较大的语言模型，而不是表示精度较高的语言模型。给定同一个系列的模型（如 LLaMA 系列），量化精度为 4 比特的 60GB 的语言模型在性能上往往优于量化精度 8 比特的 30GB 的语言模型。此外，相关研究还表明，经过 4 比特权重量化后，大语言模型的上下文学习能力、复杂推理能力和指令跟随能力受到的影响都很少。

- ▶ 对于语言模型来说，激活值相对于模型权重更难量化：当 Transformer 语言模型的参数规模超过一个阈值后，激活值开始出现较大的异常值。数值较大的异常值对大语言模型激活量化带来了重要的挑战。为了克服这一问题，需要采用特定的处理方法，例如混合精度分解、细粒度量化和困难值迁移来减轻异常值的影响。由于小型语言模型中激活值的范围通常相对标准，激活值量化对于小模型的模型效果影响较小。尽管已经有一些研究工作表明，基于 INT8 的激活值量化可以获得较好的模型效果，但是这一任务仍然具有较高的研究挑战。此外，即使对于量化感知的训练方法，较低精度的激活值量化仍然需要更多探索。
- ▶ 轻量化微调方法可以用于补偿量化大语言模型的性能损失：大语言模型在超低比特权重量化时（如 2 比特量化），可能会出现预测精度的大幅下降，这种精度下降问题可以通过**轻量化微调（如 LoRA）** 的方式进行性能补偿。**LoRA** 方法的核心是维护两部分参数，包括不微调的模型权重与微调的适配器参数。基于 **LoRA** 的性能补偿方法的基本想法是，针对模型权重进行低比特量化，而对于适配器参数则使用 16 比特浮点数表示并使用 **LoRA** 算法进行微调。在推理时，量化部分的模型权重会先反量化为 16 比特浮点数，再与适配器权重相加进行融合使用。此外，**QLoRA** 更为针对性地设计了面向量化模型的性能补偿方法，在轻量化微调的同时还考虑了显存优化，主要提出了三种改进方法，包括引入新的数据类型 **NF4**（4-bit NormalFloat）来缓解量化误差、提出双重量化技术以减少平均显存占用，以及分页优化器来管理显存峰值。

① 解码策略

- 背景知识
- 两种解码策略及其改进
- 使用效果示例

② 解码加速算法

- 解码效率分析
- 系统层面优化
- 解码策略优化

③ 低资源部署策略

- 模型量化
- 模型蒸馏
- 模型剪枝

与模型量化不同，模型蒸馏和模型剪枝则通过精简模型的结构，进而减少参数的数量。

模型蒸馏（Model Distillation）的目标是将复杂模型（称为教师模型）包含的知识迁移到简单模型（称为学生模型）中，从而实现**复杂模型的压缩**。一般来说，通常会使用教师模型的输出来训练学生模型，以此来传递模型知识。

以分类问题为例，教师模型和学生模型在中间每一层会输出特征表示（特指神经网络模型），在最后一层会输出针对标签集合的概率分布。模型蒸馏的核心思想是，引入额外的损失函数（称为**蒸馏损失函数**），训练学生模型的输出尽可能接近教师模型的输出。在实际应用中，蒸馏损失函数通常与分类损失函数（交叉熵损失函数）联合用于训练学生模型。下面首先介绍传统的知识蒸馏方法，再介绍其在大语言模型中的应用。

根据蒸馏知识的不同，传统的模型蒸馏方法包括两种类型：基于反馈的知识蒸馏方法和基于特征的知识蒸馏方法。

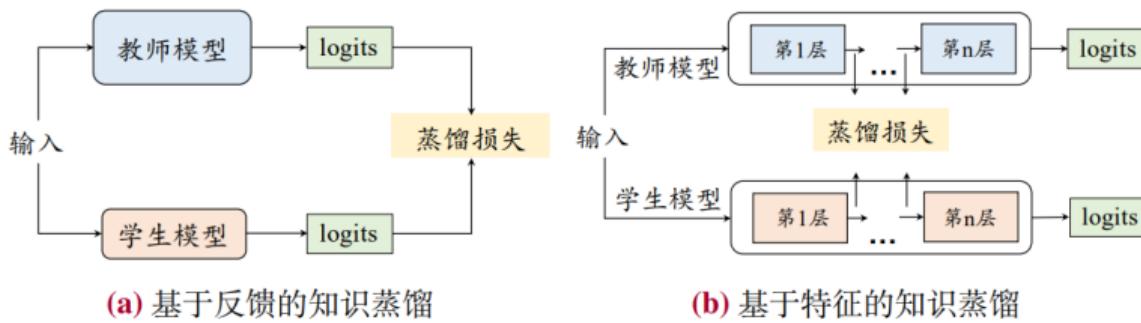


图 9.9 基于反馈的知识蒸馏和基于特征的知识蒸馏对比

- **基于反馈的知识蒸馏**: 这种方法主要关注教师模型最后一层输出的 **logits**, 这些 logits 经过 softmax 变换后, 可以用作学生模型的“软标签”来进行学习, 蒸馏损失函数可以形式化表示为:

$$\mathcal{L}(l_t, l_s) = \mathcal{L}_D(p_t(\cdot), p_s(\cdot))$$

其中 $p_t(\cdot)$, $p_s(\cdot)$ 分别表示教师模型和学生模型的 logits 经过 softmax 函数所获得的预测标签概率分布。

- **基于特征的知识蒸馏**: 这种方法关注于教师模型的中间层输出的激活值, 并使用这些激活值作为监督信息训练学生模型。例如, 在基于多层 Transformer 架构的大语言模型中, 每一层输出的特征表示都可以用作知识。相应的蒸馏损失函数可以表示为:

$$\mathcal{L}(f_t(x), f_s(x)) = \mathcal{L}_F(\Phi(f_t(x)), \Phi(f_s(x)))$$

其中 $f_t(x)$ 和 $f_s(x)$ 分别表示教师模型和学生模型的中间层输出特征, $\Phi(\cdot)$ 表示变换函数用于处理输出形状不匹配的情况。

相较于最终的预测分布, 中间层特征提供了更为丰富的模型信息, 有助于在模型蒸馏过程中实现更为有效的知识迁移。然而, 这种方法也存在一些技术难点, 如消除架构不一致的影响、目标层自动化选择等。

面向大语言模型的知识蒸馏旨在将大语言模型（教师模型）包含的知识迁移到小模型（学生模型）中。根据大语言模型权重是否可以获得，可以分别使用白盒模型蒸馏方法和黑盒模型蒸馏方法。

目前，比较典型的工作主要关注于**蒸馏大模型的关键能力**，如上下文学习能力、思维链推理能力以及指令遵循能力。下面以思维链推理能力的蒸馏为例进行介绍：

问题: A gentleman is carrying equipment for golf, what is he likely to have?

选项: (a) club (b) assembly hall (c) meditation center (d) meeting (e) church

思维链: The answer must be something that is used for golf. Of the above choice, only clubs are used for golf. So the answer is (a) club

例 9.2 思维链能力的知识蒸馏输入示例

其中输入为“问题”和“选项”，然后**使用大模型生成对应的解释“思维链”**。在这个输入的基础上，可以让学生模型同时学习预测标签，以及学习大模型生成的对应的推理过程解释。上述思路可以形式化表示为下面的损失函数：

$$\mathcal{L} = \mathcal{L}_{label} + \lambda \mathcal{L}_{cot}$$

其中， \mathcal{L}_{label} 表示对于标签的预测损失， \mathcal{L}_{cot} 表示生成思维链文本所带来的损失， λ 为结合系数。

① 解码策略

- 背景知识
- 两种解码策略及其改进
- 使用效果示例

② 解码加速算法

- 解码效率分析
- 系统层面优化
- 解码策略优化

③ 低资源部署策略

- 模型量化
- 模型蒸馏
- 模型剪枝

传统模型剪枝方法一般可以被分为两类，包括结构化剪枝和非结构化剪枝。下面针对这两种剪枝方法进行介绍：

- ▶ **结构化剪枝**：结构化剪枝（Structured Pruning）旨在去除对于性能影响较小的模型组件，可以删除神经元、通道甚至中间层。结构化剪枝的核心思想是，在尽量保持模型预测精度的条件下，**去除那些对于结果影响不大的结构单元**，如注意力机制中的注意力头、前馈层权重中的特定维度等。这里以前馈层中的维度剪枝为例进行介绍。具体来说，通过计算每一维列向量权重的数值大小来作为判断其重要性的标准，然后再去掉那些重要性较低的维度从而实现剪枝。
- ▶ **非结构化剪枝**：非结构化剪枝（Unstructured Pruning）主要关注去除模型权重矩阵中不重要的数值。与结构化剪枝不同，非结构化剪枝**并不修改模型的结构**。一般来说，模型权重都是以矩阵形式进行表示与存储的，非结构化剪枝通过创建一个包含0/1的掩码矩阵，并将这一矩阵与原始的权重相乘，其中0所在位置的权重则不会在模型的计算中产生作用。当剪枝完成后，那些被剪枝掉的位置只会存储数值0，从而节省了存储空间。

总体来说，在实际应用中，非结构化剪枝一般可以实现更高的剪枝比率，但是不会显著加速模型的计算过程，因为被掩码的部分可能仍然需要参与计算。要想实现加速，则需要特定软件和硬件的支持。而结构化剪枝通过去除结构单元，可以显著减少所需的矩阵乘法次数，实现模型的压缩和加速。

面向大语言模型的剪枝目的是在尽可能少地影响模型性能的情况下减少其计算资源的需求。与传统的模型剪枝类似，主要分为结构化和非结构化剪枝两类。

其中，**非结构化剪枝**一般容易获得更高的压缩率，典型的工作包括 SparseGPT，其只需要使用 1 张 A100 (80G) 显卡就可以完成对于 175B 参数规模大语言模型（如 OPT 模型）的剪枝，实现 60% 模型参数的剪枝，并较好地保持了困惑度不升。

作为另一类方法，面向大模型的**结构化剪枝**研究也取得了较好的模型压缩效果。例如 LLM-prune 在 LLaMA (7B) 上实现了剪枝 20% 参数但是依然保持原始模型 93.6% 的预测精度。Sheared LLaMA 则将 LLaMA-2 (7B) 剪枝得到 2.7B 参数规模，保持原始模型 87.8% 的预测精度。