(Dense Connections)。这种设计通过跨层连接使信息能够在模型的深层中高效传播，从而保留了细粒度的隐藏状态信息，显著提升了模型性能，尤其是在处理需要深度表征的任务时表现尤为突出。BlackMamba[479] 和 MoE-Mamba[480] 则将专家混合（MoE）技术引入到 SSM 模型中，进一步增强了 Mamba 系列的能力。BlackMamba 专注于利用专家模块的灵活性，动态分配计算资源，根据任务需求选择性地激活不同的专家，从而在保持高性能的同时优化了资源使用效率。而 MoE-Mamba 则进一步改进了专家混合技术，使其更适合状态空间模型的特性，通过更高效的专家选择机制在训练和推理过程中显著降低计算成本，同时保持甚至提升模型性能。

10.2.2 模型量化

量化（Quantization）是一种广泛应用的技术，将大语言模型的权重和激活值从高比特宽度转换为低比特宽度表示，从而显著降低计算成本和内存开销。具体来说，许多量化方法通过将 FP16 浮点张量转化为低比特整数张量来实现，其表示形式如下：

$$X_{\text{INT}} = \left\lfloor \frac{X_{\text{FP16}} - Z}{S} \right\rceil \tag{10.7}$$

$$S = \frac{\max(X_{\text{FP16}}) - \min(X_{\text{FP16}})}{2^N - 1} \tag{10.8}$$

其中，X_{FP16} 表示 16 比特浮点（FP16）值，X_{INT} 表示低精度整数值，N 表示比特数，S 和 Z 分别表示缩放因子和零点。

如上节所述，大语言模型的推理过程通常分为两个阶段：预填充阶段和解码阶段。在预填充阶段，模型需要处理较长的 Token 序列，其核心操作是通用矩阵乘法（General Matrix Multiplication，GEMM）。预填充阶段的延迟主要受到高精度 CUDA 核心执行计算的限制。为了解决这一问题，现有方法采用对权重和激活值同时进行量化的策略，以便利用低精度张量核心加速计算。如图10.6所示，在每次 GEMM 操作之前，激活值会被在线量化，从而允许使用低精度张量核心（例如 INT8）进行计算。这种量化方法称为**权重-激活量化**（Weight-Activation Quantization），它通过将权重和激活值同时转换为低精度表示，大幅提升了计算效率和硬件利用率。

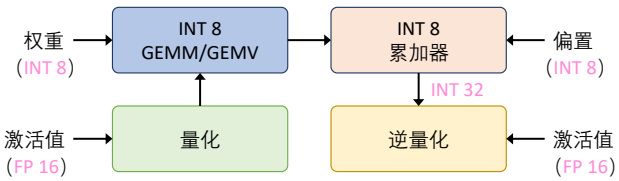


图 10.6 权重-激活量化流程[471]

在解码阶段，大语言模型在每个生成步骤中仅处理一个词元，其核心操作为通用矩阵-向量乘法（General Matrix-Vector Multiplication，GEMV）。解码阶段的延迟主要受到加载大规模权重张量的限制。为了解决这一问题，现有方法集中于对权重进行量化，以加速内存访问和减少带宽需求。这种方法称为**仅权重量化**（Weight-Only Quantization），其流程包括对权重进行离线量化，将其转换为低精度表示，并在计算时将低精度权重反量化为 FP16 格式进行运算。如图 10.7 所示，这种方法有效降低了解码阶段的内存开销，同时提升了推理效率。

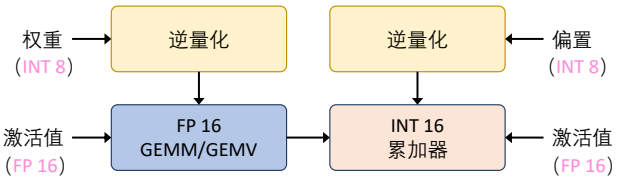


图 10.7　仅权重量化流程[471]

模型量化方法又可以根据在模型训练完成后，还是在模型训练过程中进一步细分为：训练后量化和量化感知训练。本节将分别介绍上述模型量化方法。

1. 训练后量化

训练后量化（Post-Training Quantization，PTQ）是一种对已完成训练的模型进行量化的方法，无需重新训练原有模型，从而避免了高昂的计算成本。尽管 PTQ 方法在较小规模模型上已经广泛的研究，但直接将现有的模型量化技术应用于大语言模型仍然面临诸多挑战。这主要是因为，与较小模型相比，大语言模型的权重和激活值通常具有更多的异常值，且分布范围更加广泛，使得量化过程变得更加复杂。

许多研究致力于开发高效的量化算法，以压缩大语言模型并提升其运行效率。在量化张量的类型方面，一些研究（如 [481]、[482]、[483]、[484]）专注于仅对权重进行量化，而另一些研究（如 [204]、[205]、[207]）则同时对权重和激活值进行量化。值得注意的是，KV 缓存作为大语言模型中的独特组件，对内存占用和访问效率有着显著影响。因此，一些研究（如 [485]、[486]、[487]）提出了针对 KV 缓存的量化方案，以进一步优化内存使用和访问性能。在数据格式方面，大多数量化算法选择统一的数据格式，以便于硬件实现和优化。在确定量化参数（如缩放因子和零点）时，大多数研究通过分析权重或激活值的统计特性来推断这些参数。然而，也有一些研究（如 [483]、[488]）通过最小化重构损失来搜索最优量化参数。此外，一些研究（如 [481]、[483]、[489]）在量化过程中提出了更新未量化权重（即“量化值更新”）的策略，以进一步提升模型的性能和表现。这些方法为量化领域提供了新的优化方案和实践方向。

在仅权重量化方面，Optimal Brain Quantization（OBQ）[490] 将经典的 Optimal Brain Surgeon（OBS）[491] 二阶权重剪枝框架推广应用于量化。OBQ 的核心思想是通过迭代的方式逐步将神经网

络的权重量化到目标精度，同时尽量减少量化带来的误差。具体来说，OBQ 采用了一种贪心策略，逐个量化权重，并在每次迭代中动态更新未量化的权重，以补偿量化误差。其目标是找到最优的量化参数，以在缩小模型规模的同时尽可能保留其性能。然而，OBQ 的计算复杂度较高，其与权重数量呈立方关系，因而需要极大的计算资源支持。为找到最佳量化参数，OBQ 通常需要多次迭代，而每次迭代都需更新整个模型的权重并重新计算相关参数。随着迭代次数的增加，计算成本也随之显著上升。

GPTQ（GPT Quantization）[481] 在 OBQ 算法的基础上进行了简化和改进，使量化过程更加高效。采用了一次性量化的方法，即在单次迭代中将整个模型的权重量化到目标精度。这种方式与 OBQ 的逐步迭代量化不同，大大降低了计算复杂性。通过对每一行权重采用统一的从左到右量化顺序，GPTQ 避免了频繁更新海森矩阵的高昂计算成本。仅在量化某一行时计算海森矩阵，并将其结果用于后续行的量化操作，从而显著减少计算开销并加速整体量化过程。此外，GPTQ 引入了批量更新操作，允许多个权重同时进行量化，从而提高了 GPU 的计算效率。为了进一步优化内存使用，GPTQ 采用了一种“Lazy Batch-Updates”策略，将模型划分为多个块并逐块压缩。这种分块处理方法使得即使在 GPU 内存较小的情况下，也能够高效完成模型量化，而无需一次性加载整个模型。

LUT-GEMM（Look-Up Table - General Matrix Multiplication）[482] 则是将矩阵乘法与查找表（LUT）结合，旨在通过减少反量化开销来加速量化后的大语言模型的推理过程。在量化模型中，由于权重和激活值被量化为低比特（如 8-bit、4-bit 或更低），数值取值范围有限，所有可能的乘法结果可以预先计算并存储在查找表中。运行时通过查表快速获得乘积结果，无需实际执行乘法运算，从而降低计算复杂度。查表操作还支持分组方式，例如 4-bit 权重与 4-bit 激活值的组合可形成 256 种结果，查表后再执行累加即可完成矩阵乘法。此外，LUT-GEMM 与通用矩阵乘法（GEMM）相结合，保留了高效的矩阵运算结构，进一步减少计算密度，同时适配硬件加速器（如 GPU 和 TPU），在低比特量化场景下显著降低延迟和能耗。

在权重和激活的量化方面，ZeroQuant[492] 提出了更精细的量化方法。它通过核融合技术有效减少量化过程中的内存访问成本，并利用逐层知识蒸馏来恢复模型性能。ZeroQuant 结合了组内量化（group-wise quantization）对模型权重进行压缩，以及按 Token 量化（token-wise quantization）对激活值进行处理，从而实现了高效的量化方案。ZeroQuantV2[493] 在此基础上引入了低秩补偿（Low-Rank Compensation, LoRC）技术，通过低秩矩阵来缓解量化误差的问题，从而进一步提升了量化的表现。ZeroQuant-FP[494] 探索了将权重和激活值量化为 FP4 和 FP8 浮点格式的可行性。研究表明，与整数格式相比，将激活值量化为浮点类型（FP4 和 FP8）能够显著提高模型性能，展现出更优的量化效果。

在此基础上，许多研究从不同角度对上述算法进行了改进，进一步提升了其性能和适用性。AWQ[483] 注意到权重通道对模型性能的贡献并不均等，尤其是那些与激活值中出现异常值的输入通道对齐的权重通道更为重要。为更好地保留这些关键权重通道，AWQ 引入了一种重参数化方法。

该方法通过网格搜索确定重参数化系数，从而有效最小化重构误差，增强了对关键权重的保留能力。OWQ[495] 针对与激活异常值相关的权重难以量化的问题，提出了一种混合精度量化策略。该方法通过识别权重矩阵中的“弱列”，为这些关键权重分配更高的精度，同时对其余权重以较低精度进行量化，从而在性能和效率之间达成平衡。SpQR[496] 专注于在量化过程中识别权重异常值，并为这些异常值分配更高的精度，而将其余权重量化为 3 比特。这种选择性高精度处理的方法减少了关键权重的量化误差，有效提升了模型性能。QuantEase[497] 在每一层的量化过程中，QuantEase 提出了一种基于坐标下降的优化方法，以更精确地补偿未量化的权重。此外，QuantEase 可以利用 GPTQ 生成的量化权重作为初始化点，并在此基础上进一步优化补偿过程，提高了量化的效果。AffineQuant[498] 则首次将等效仿射变换引入量化过程，扩展了优化的搜索空间。这种方法能够更全面地拟合权重分布，从而显著降低量化误差，为模型量化提供了新的视角。SqueezeLLM[484] 提议将异常值存储在全精度稀疏矩阵中，并对其余权重应用非均匀量化。非均匀量化的值根据量化敏感度确定，这有助于提高量化模型的性能。

2. 量化感知训练

量化感知训练（Quantization-Aware Training，QAT）通过在模型训练过程中整合模拟量化效应的层，使权重适应量化引起的误差，从而提高任务性能。然而，训练 LLMs 通常需要大量的训练数据和计算资源，这可能成为 QAT 实施的瓶颈。因此，当前的研究重点是减少数据需求或减轻 QAT 实施的计算负担。

为了减少对数据的需求，LLM-QAT[499] 提出了一种无需数据（Data-free）的量化训练方法。该方法通过原始的 FP16 大语言模型（LLM）生成训练数据。具体而言，LLM-QAT 使用词汇表中的每个词元作为起始词元生成句子。基于这些生成的训练数据，LLM-QAT 应用基于知识蒸馏的流程，对量化后的模型进行训练，使其输出分布接近原始 FP16 模型的输出分布。Norm Tweaking[500] 进一步改进了这一方法，通过限制起始词元的选择，仅选择那些属于顶级语言列表中语言类别的词元。该策略能够显著提升量化模型在各种任务上的泛化能力。同时建议在量化后训练 LayerNorm 层，并使用知识蒸馏来匹配量化模型的输出分布与 FP16 模型的输出分布，从而实现与 LLM-QAT 类似的效果，同时避免高昂的训练成本。

为降低计算成本，许多研究采用参数高效调优策略来加速量化感知训练（QAT）。QLoRA[235] 提出将大语言模型的权重量化为 4 位，并使用 BF16 对每个 4 位权重矩阵进行 LoRA[501] 微调。QLoRA 使得在单个 GPU 上仅使用 30GB 的内存即可对 65B 参数的大模型进行高效微调。QA-LoRA[502] 则在 QLoRA 的基础上引入了组内量化。作者指出，QLoRA 中的量化参数数量远少于 LoRA 参数数量，导致量化和低秩适应之间的不平衡问题。为解决这一问题，QA-LoRA 提议增加量化操作的参数数量，使用组内量化操作，并将 LoRA 项合并到相应的量化权重矩阵中，以提升性能。LoftQ[503] 则发现 QLoRA 中使用零初始化的 LoRA 矩阵对下游任务效率较低。为此，LoftQ 提出了一种改进方法，即利用原始 FP16 权重和量化权重之间的差异进行奇异值分解（SVD），以初始化 LoRA 矩

阵。通过迭代应用量化和 SVD，LoftQ 实现了对原始权重更准确的近似，从而进一步提升模型的性能和适配能力。

10.2.3 模型稀疏化

稀疏化（Sparsification）是一种模型压缩技术，其目标是通过增加模型参数或激活中零值元素的比例，降低计算复杂度和内存使用。稀疏化利用计算过程中对零元素的高效忽略，实现了资源的节约和性能的优化。在大语言模型中，稀疏化通常应用于权重参数和注意力激活。稀疏化的主要策略包括权重剪枝和稀疏注意力机制。稀疏注意力机制已在本书第1 节进行了详细讨论，本节将重点探讨权重剪枝机制。

权重剪枝（Weight Pruning）是一种系统地从模型中移除不那么关键的权重和结构的方法，旨在在预填充阶段和解码阶段减少计算和内存成本，同时不显著牺牲性能。权重剪枝方法根据剪枝过程的粒度可以分为两类：无结构剪枝和结构化剪枝，如图10.8所示。

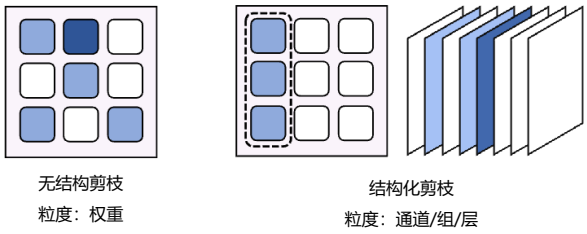


图 10.8　无结构剪枝和结构化剪枝示意图[471]

1. 无结构剪枝

无结构剪枝（Unstructured pruning）通过细粒度方式移除单个权重值，目标是在尽量减少对模型预测影响的情况下实现更高的稀疏度。无结构剪枝的研究重点通常集中在剪枝准则上，包括权重的重要性评估和剪枝率的设定。鉴于大语言模型的参数规模极其庞大，提高剪枝效率显得尤为重要。其中一种常用的剪枝准则是通过最小化模型的重构损失来选择需要剪枝的权重，从而尽可能减少对模型性能的影响。

SparseGPT[504] 是最小化重构损失策略的典型代表，通过一次性操作移除冗余参数，大幅减少模型规模，无需反复训练。其核心思想基于 Optimal Brain Surgeon（OBS）[491]，通过分析剪枝对网络重构损失的影响，生成剪枝掩码并调整未剪枝权重以补偿误差。SparseGPT 采用局部层级剪枝的方式，使剪枝过程高度并行化，同时通过近似二次损失避免直接计算海森矩阵的高计算成本。此外，它引入优化的排序和迭代策略以及自适应掩码选择技术，有效克服了 OBS 的效率瓶颈，在显著提升剪枝效率的同时保持模型性能。Prune and Tune[505] 对 SparseGPT 进行了改进，在剪枝过程中以最少的训练步骤对大型语言模型进行微调。ISC[506] 通过结合 OBS[491] 和 OBD（Optimal Brain

Damage）[507] 中的显著性准则，设计了一种新颖的剪枝准则。它还根据海森矩阵信息为每一层分配非均匀的剪枝率。

另一种常见的剪枝准则是基于幅度的方法（magnitude-based）。Wanda[508] 提出了一种剪枝策略，利用权重幅度与输入激活范数的逐元素乘积作为剪枝依据。RIA[509] 则引入了相对重要性和激活度（Relative Importance and Activations）这一指标，将权重与激活结合考虑，通过分析所有权重的连接关系来评估每个权重的重要性。此外，RIA 还将非结构化稀疏模式转换为结构化的 N:M 稀疏模式，从而在 NVIDIA GPU 上实现实际的加速。最近的研究 Pruner-Zero[510] 提出了为大语言模型（LLMs）自动确定最优剪枝准则的方法，超越了传统的手工设计标准。研究表明，对于 LLaMA 和 LLaMA-2，最优的剪枝度量是 $\mathbf{W} \odot \mathbf{W} \odot \sigma(\mathbf{G})$，其中 $\mathbf{W}$ 和 $\mathbf{G}$ 分别表示权重和梯度，而 $\sigma(\cdot)$ 是一种缩放函数，将张量的最小值和最大值归一化到 $[0, 1]$ 区间。

无结构剪枝以精细粒度的方式移除单个权重值。相比于结构化剪枝，它通常能够在对模型预测影响最小的情况下实现更高的稀疏度。然而，由于无结构剪枝产生的稀疏模式缺乏规律性，导致内存访问和计算模式变得不规则。这种不规则性显著限制了硬件的加速潜力，因为现代计算架构通常针对密集且规则的数据模式进行优化。因此，尽管无结构剪枝可以实现更高的稀疏度，但在硬件效率和计算加速方面的实际收益可能较为有限。

2. 结构化剪枝

结构化剪枝（Structured pruning）针对模型中较大的结构单元进行剪枝，例如整个通道或层，与非结构化剪枝相比，其粒度更粗。由于这些方法与传统硬件平台优化处理的密集、规则数据模式相契合，因此能直接加快在这些平台上的推理速度。然而，结构化剪枝的粗粒度往往会对模型性能产生更为显著的影响。这类工作的剪枝标准还会强化结构化剪枝模式。

LLM-Pruner[511] 提出了一种任务无关的结构化剪枝算法。该方法首先根据神经元之间的连接依赖关系，识别大语言模型中的成对结构，这些相互依赖的结构需要同时移除，以确保剪枝后的结构保持正确性。例如，在 LLaMA 模型中，存在 MLP 内部的耦合、MHA 内部的耦合以及整个网络中的维度耦合等层级依赖关系。通过特定公式将这些耦合关系整合为一个依赖图，并利用递归搜索快速定位耦合结构。在完成耦合结构分组后，该方法评估每个组对模型整体性能的贡献，并根据预设的剪枝比例对各组的重要性进行排序，剪除重要性较低的组。剪枝完成后，为了恢复模型性能，LLM-Pruner 引入 LoRA 进行参数高效的训练。

LoRAPrune[512] 为带有 LoRA 模块的大语言模型提出了一个结构化剪枝框架,以实现基于 LoRA 模型的快速推理。它设计了一种由 LoRA 引导的剪枝标准，不使用预训练权重的梯度，而是利用 LoRA 的权重和梯度进行重要性估计，避免了计算预训练权重梯度带来的巨大内存开销。将 LoRA 引导的修剪标准整合到迭代修剪过程中，能够有效地去除模型中冗余的通道和头部，实现模型的结构化剪枝，在减少模型规模的同时保持较好的性能。LoRAShear[513] 同样为基于 LoRA 的大语言模型设计了一种剪枝方法，通过分析大语言模型参数与 LoRA 模块的关系，创建原始大语言模型和 LoRA 模块的依赖图，以发现最少需要移除的结构，并分析知识分布。基于依赖图对 LoRA 适

配器进行渐进式结构化剪枝，使模型的固有知识得以转移，更好地保留冗余结构中的信息。同时引入结构稀疏优化算法，利用 LoRA 模块的信息来更新权重，提高知识保存率。

混合专家（MoE）技术在大语言模型领域备受关注。近期一些研究开始探索针对基于 MoE 的大语言模型的专家剪枝方法。ExpertSparsity[514] 是一种专家稀疏化方法，用于 MoE 中的前馈神经网络专家的稀疏化。它通过计算原始输出和稀疏化后输出之间的 Frobenius 范数来量化被稀疏化的专家的损失。对 MoE 模型中的专家进行分层评估和剪枝，根据专家对模型整体性能的贡献程度，去除那些对性能影响较小的专家，以达到压缩模型和提高计算效率的目的。采用渐进式剪枝（Progressive Pruning）方法，逐步地对专家进行剪枝操作，在每次剪枝后评估模型性能，确保剪枝过程不会导致模型性能大幅下降，通过这种渐进的方式找到最优的剪枝策略。在推理过程中，采用了动态跳过（Dynamic Skipping）方法，根据输入数据的特点动态地决定是否跳过某些专家的计算，对于那些对当前输入不太重要的专家，可以直接跳过，从而减少不必要的计算量，提高推理速度。

10.2.4 知识蒸馏

知识蒸馏（Knowledge Distillation, KD）是一种广泛应用的模型压缩技术，其核心思想是将大型模型（称为教师模型，Teacher Model）的知识迁移到较小的模型（称为学生模型，Student Model）中。现有研究主要关注如何高效地将大语言模型的各种能力传递到学生模型中。根据是否可以访问大模型的内部结构（如参数、梯度），知识蒸馏可以分为两大类：白盒知识蒸馏和黑盒知识蒸馏，如图 10.9 所示。

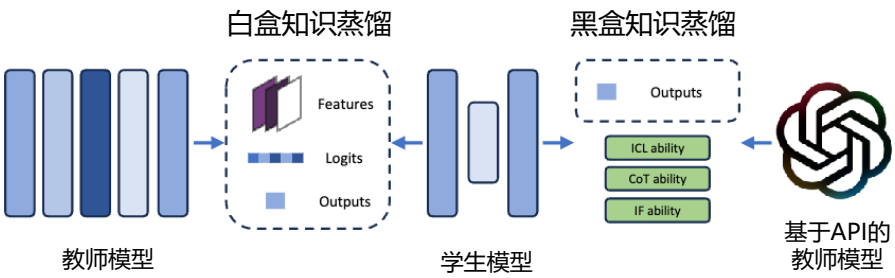


图 10.9 白盒知识蒸馏（左）和黑盒知识蒸馏（右）示意图[471]

白盒知识蒸馏（White-box KD）指的是利用对教师模型结构和参数的访问权限来进行蒸馏的方法。这种方式使得知识蒸馏能够有效地利用教师模型的中间特征和输出分布，以提升学生模型的性能。**黑盒知识蒸馏**（Black-box KD）指的是在教师模型的结构和参数不可用的情况下进行知

识蒸馏的方法。通常，黑盒知识蒸馏仅使用教师模型获得的最终结果来提炼学生模型。

1. 白盒知识蒸馏

白盒知识蒸馏能够获取教师模型的细节信息，因而可以采用多种策略来提高学生模型的性能。给定教师分布 $p_T(y|x)$ 以及由参数 θ 确定的学生分布 $p_\theta^S(y|x)$，标准的知识蒸馏目标（包括针对序列级模型的几种变体）[515, 516] 本质上是最小化教师分布和学生分布之间近似的正向 Kullback-Leibler divergence （KLD），记为 $KL[p_T||p_\theta^S]$，这会迫使 p_θ^S 覆盖 p_T 的所有高概率区域（mode，也成模态）。对于文本分类任务，这种方法表现良好，因为输出空间通常由有限的类别组成，使得 $p_T(y|x)$ 和 $p_\theta^S(y|x)$ 的高概率区域都很少。然而，对于开放式文本生成任务（大语言模型应用通常属于这种情况），输出空间要复杂得多，并且由于模型容量有限，$p_T(y|x)$ 所包含的高概率区域数量可能远远超过 $p_\theta^S(y|x)$ 所能表达的数量。最小化正向 KLD 会导致 p_θ^S 对 p_T 的空白区域（void region）赋予不合理的高概率[517]，在自由运行的生成过程中，这种现象可能会导致学生模型生成在教师分布 p_T 下几乎不可能出现的样本[518]。

针对该问题，MiniLLM[519] 采用标准的白盒知识蒸馏方法，但将正向 KLD 替换为反向 KLD，即 $KL[p_\theta^S||p_T]$。与最小化 $KL[p_T||p_\theta^S]$ 相比，最小化 $KL[p_\theta^S||p_T]$ 会能够引导学生分布 p_θ^S 关注教师分布 p 的主要高概率区域，同时对 p 的空白区域赋予较低的概率[520]。在大语言模型的文本生成任务中，这意味着学生模型可以避免学习教师分布中过多的长尾变体，而是更专注于生成内容的准确性。这在需要真实性和可靠性的实际场景中至关重要。为了优化 $\min_\theta KL[p_\theta^S||p_T]$，MiniLLM 使用策略梯度法（Policy Gradient）[521] 推导目标函数的梯度，并通过以下改进措施进一步稳定和加速训练：单步分解以降低方差，教师混合采样以缓解奖励操纵问题，以及长度归一化以消除长度偏差。

文献 [522] 则将自回归序列模型的知识蒸馏问题转换为一个带有交互式专家的模仿学习问题。将同策略模仿扩展到知识蒸馏，文献 [522] 提出了 on-policy KD。在知识蒸馏过程中使用同策略数据时，学生模型会根据教师模型的输出分布，针对其自生成输出序列中的错误词元获得词元特定的反馈。这形成了一种类似于在强化学习中看到的反馈循环，有助于最小化训练-推理分布不匹配的问题。此外，随着学生模型在训练过程中不断改进，其生成的数据质量也会提高。给定输入 x，学生模型生成输出序列 y，并在中间状态 $y_{<n}$ 上模仿教师模型的词元级分布 $p_T(y_n|x)$。具体而言，同策略损失 L_{OD} 由下式给出：

$$L_{OD}(\theta) = \mathbb{E}_{x\sim X}\left[\mathbb{E}_{y\sim p_S(\cdot|x)}\left[D_{KL}\left(p_T\|p_\theta^S\right)(y|x)\right]\right] \tag{10.9}$$

类似于同策略模仿，on-policy KD 不会通过学生模型的采样分布 $p_S(\cdot|x)$ 进行反向传播。这种不依赖采样的方式使得训练更加稳定，同时计算效率更高。在 on-policy KD 中，训练是在学生模型可能生成的输出序列上进行的。训练过程中，通过设置温度参数 $\gamma = 1$ 来鼓励学生生成具有多样性的序列。此外，针对无标签的输入提示，由于学生模型的规模通常小于教师模型，使用学生

模型生成序列的计算成本显著低于教师模型。

在此基础上，进一步结合有监督方法与同策略方法，文献 [522] 提出了一种更通用的方案，Generalized KD（GKD）。GKD 允许灵活选择优化的散度形式和用于训练的输出序列来源。具体而言，可以优化教师模型和学生模型之间的任意词元级概率分布散度。在训练数据上，GKD 结合了固定数据集（包括教师生成的序列或带标签的真实数据）与学生模型同策略生成的序列，从而形成混合训练数据。GKD 通过最小化以下形式的目标函数实现统一：

$$L_{GKD}(\theta) = (1-\lambda)\mathbb{E}_{(x,y)\sim(X,Y)}\left[D(p_T \| p_\theta^S)(y|x)\right] + \lambda\mathbb{E}_{x\sim X}\left[\mathbb{E}_{y\sim p_S(\cdot|x)}\left[D(p_T \| p_\theta^S)(y|x)\right]\right] \tag{10.10}$$

其中 $D(p_T, p_S)(y|x)$ 是教师模型和学生模型分布之间的散度 $\lambda \in [0,1]$ 是一个超参数，用于控制学生模型生成数据的比例，即同策略学生模型生成输出的比例。与 on-policy KD 类似，不会通过学生模型的采样过程进行梯度反向传播。on-policy KD 和有监督知识蒸馏是广义知识蒸馏的特殊情况，分别对应于散度 D 设为正向 KL 散度，且学生模型生成数据比例 λ 分别为 1 和 0 的情况。也就是说，广义知识蒸馏允许对比例 λ 和散度进行其他选择。

此外，TED[523] 提出了一种任务感知的逐层知识蒸馏方法。该方法在教师模型和学生模型的每一层后添加过滤器，首先训练这些特定于任务的过滤器，然后在训练学生模型的过滤器时冻结教师模型的过滤器，以使学生模型的输出特征能够与对应的教师过滤器输出特征对齐。MiniMoE[524] 则通过采用专家混合（Mixture-of-Experts, MoE）模型作为学生模型，来缩小学生模型与教师模型之间的能力差距。KPTD[525] 提出了一种通过知识蒸馏将实体定义中的知识转移到大语言模型参数中的方法。该方法基于实体定义生成一个转移集，并利用这些定义对学生模型进行蒸馏，使学生模型的输出分布与教师模型相匹配。

2. 黑盒知识蒸馏

黑盒蒸馏的核心目标是在无法访问大模型内部参数的情况下，通过其输出（如分类概率或生成的文本）来指导学生模型的学习。具体而言，学生模型可以通过模仿大模型的输出分布（如分类概率分布）来接近其行为，从而实现性能的压缩与迁移。此外，学生模型还可以在大模型的指导下学习特定任务能力或者大语言模型的泛化能力，包括上下文学习（ICL）能力[526]、思维链（CoT）推理能力[395] 以及指令跟随（IF）能力[24] 等。

TAPIR[527]（Task-Aware Curriculum Planning for Instruction Refinement）框架通过多任务课程规划，蒸馏黑盒大语言模型的指令回答能力。它利用教师大模型挑选学生模型难以遵循的指令，进行难度重采样，从而提升学生模型的学习效果。同时，为了平衡学生模型的多任务技能，TAPIR 对训练集中的任务配比进行调整，重新分配任务多样性分布，并根据多任务特点自动优化教师模型的回答风格。此外，通过引入课程规划机制，TAPIR 框架系统地提高任务难度级别，逐步增强学生大语言模型的能力。TAPIR 框架整体结构如图10.10所示。

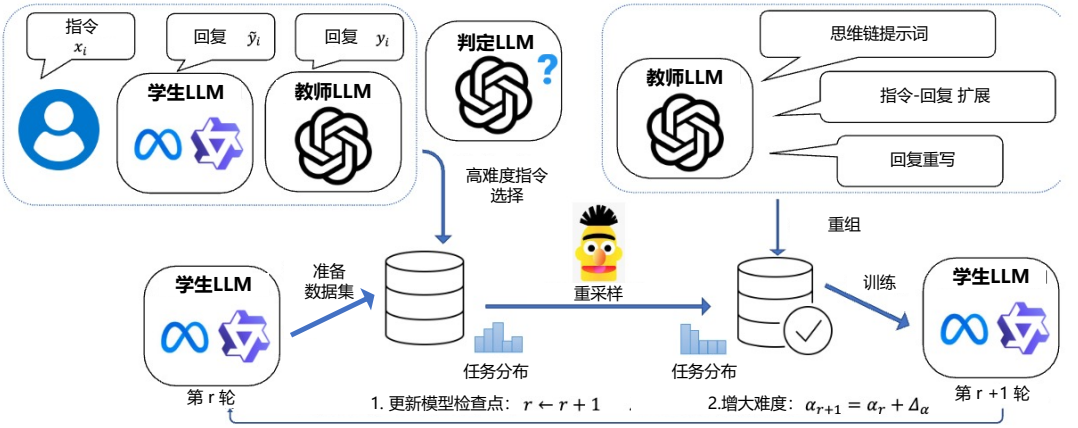


图 10.10 TAPIR 框架整体结构图[527]

整个流程从初始化一个预训练的学生模型开始，依次通过以下步骤进行：（1）利用一个开源指令数据集（如 Alpaca 数据集）作为基础，通过计算模型拟合难度（Model Fitting Difficulty，MFD）分数筛选出对学生模型较为困难的指令对，生成种子数据集；（2）采用多任务规划指令蒸馏方法，根据设定的任务类型配比，利用教师模型（如 ChatGPT）扩展种子数据集，生成更多具有相似难度水平的指令-响应对，并提升推理类任务的采样概率，以缓解能力冲突问题；（3）在多任务回答风格增强阶段，通过特定提示重写教师模型的响应，使其提供更精细、更详细或特定格式的回答（如思维链、代码注释），帮助学生模型更好地理解和学习复杂任务；（4）通过多轮优化迭代，利用裁判模型对学生模型的回答质量进行反馈评分，生成新的蒸馏种子数据集，并逐步增加其中挑战性指令的比例，实现从易到难的泛化学习，逐步提升学生模型的能力。

模型拟合难度（MFD）指标可以用于挑选出大语言模型难以拟合的指令在数据集 D 上对学生大语言模型 S 进行微调，从而得到具有基本指令跟随能力的初始模型 S_0。接下来，使用 S_0 为数据集中的每个 x_i 生成回复，即 $\tilde{y}_i = S_0(x_i)$。这一步评估了学生大语言模型拟合 $\{(x_i, y_i)\}$ 的能力。因此，每个指令 x_i 的 MFD 分数按如下方式确定：

$$\mathrm{MFD}(x_i) = f_J(x_i, \tilde{y}_i) - f_J(x_i, y_i) \tag{10.11}$$

其中，评判大语言模型 J 评估针对 x_i 由教师生成的回复 y_i 与由学生生成的回复 $\tilde{y}_i$ 之间的质量差异。评判模型 J 的任务是对学生模型回复 $\tilde{y}_i$（即 $f_J(x_i, \tilde{y}_i)$）和教师回复 y_i（即 $f_J(x_i, y_i)$）的有用性、相关性、准确性和细节程度进行评估，并以 1 到 10 分的分数作为输出。为了构造种子数据集，设定一个阈值 δ, 只有那些 MFD 分数超过 δ 的样本对才会被纳入。

文献 [528] 提出了一种名为 Distilling Step-by-Step 的方法，该方法包括两个主要步骤：（1）给定一个教师大语言模型和一个无标签数据集，利用教师模型生成输出标签，并同时生成用于证明

标签合理性的推理依据。推理依据以自然语言解释的形式呈现，用于支持模型预测的标签。(2) 在训练较小的学生模型时，不仅使用任务标签，还借助这些推理依据进行学习。推理依据提供了更加丰富和详细的信息，解释了输入为何被映射到特定的输出标签，同时也包含了仅通过原始输入可能难以推断出的相关任务知识。

10.3 低精度训练

大语言模型的训练通常需要海量的计算资源，包括大量的 GPU 或 TPU，以及庞大的存储和内存空间。DeepSeek-V3 模型 [40]，即使采用了多种训练优化策略，训练一次仍然需要耗费 266.4 万 H800 GPU 小时。在如此巨大的计算开销下，如何在有限资源内提升模型训练和推理效率已成为研究的核心热点。

降低训练精度被广泛认为是减少训练成本最具潜力的方向之一，它可以提供更高的速度、更小的内存占用以及更低的通信开销。目前主流训练框架（例如 Megatron-LM、MetaSeq 和 Colossal-AI）仍然采用 FP32 全精度或混合精度的 FP16/BF16 策略。随着 Nvidia H100 GPU 的推出，FP8 正逐渐成为下一代低精度数据表示的主流格式。相较于现有的 16 位和 32 位混合精度方案，FP8 不仅能够将训练速度提升一倍，还能实现 50% 到 75% 的内存和通信开销优化，这一突破性进展为构建下一代大规模基础模型开辟了广阔前景。

在本节中将首先介绍 FP8 编码方式，并在此基础上介绍基于 FP8 的大模型训练方法。

10.3.1 FP8 编码

FP8 是一种低精度浮点数格式，专为提高计算效率和降低存储需求而设计，广泛应用于深度学习模型的训练和推理中。FP8 编码采用 IEEE 浮点表示的变体，包括符号位（S，sign）、指数位（E，exponent）和尾数位（M，mantissa）。指数位决定了动态范围，而尾数位决定了表示精度。其关键特征是通过减少位数来降低计算复杂度和内存占用。FP8 的常见表示方式有以下几种：E5M2（5 位指数和 2 位尾数）、E4M3（4 位指数和 3 位尾数）、E3M4（3 位指数和 4 位尾数）以及 E2M5（2 位指数和 5 位尾数）。通过调整指数位的数量，FP8 可以适应不同动态范围的计算需求。由于 E3M4 和以及 E2M5 的动态范围过小，因此大语言模型中通常采用 E4M3 和 E5M2 两种表示方法[529]。

E4M3 和 E5M2 的详细信息如表 10.1 所示。其中，NVIDIA GPU 上的 E5M2 遵循 IEEE 754 标准，因此其动态范围与 IEEE 754 的 E5M2 编码保持一致。而 E4M3 则有所不同，不符合 IEEE 754 标准。E4M3 取消了无穷大，仅保留一个 NaN，从而能够额外表示 (256, 288, 320, 352, 384, 416, 448) 这些数字。这一优化将其动态范围从 240 扩展到 448，在深度学习领域尤其实用。总体而言，E4M3 更注重精度，在 [1, 2] 区间内，其最小间隔为 1/8，而 E5M2 的最小间隔为 1/4。但是，E5M2 的动态范围更大，E4M3 的表示范围为 [-448, 448]，而 E5M2 的表示范围为 [-57344, 57344]。在大语言模型的训练中，通常建议将权重和激活张量使用 E4M3 表示，而梯度张量使用 E5M2 表示[529]。但具体选择也需视模型特性而定，有些模型可能仅适合 E4M3 或 E5M2。

表 10.1 FP8 二进制形式[529]

指标名	E4M3	E5M2
指数偏置（Exponent bias）	7	15
无穷大（Infinity）	N/A	$S.11111.00_2$
非数值（NaN）	$S.1111.111_2$	$S.11111.\{01, 10, 11\}_2$
负零（Negative Zero）	$S.1000.000_2$	$S.10000.00_2$
最大正规数（Max normal）	$S.1111.110_2 = 1.75 \times 2^8 = 448$	$S.11110.11_2 = 1.75 \times 2^{15} = 57{,}344$
最小正规数（Min normal）	$S.0001.000_2 = 2^6$	$S.00001.00_2 = 2^{14}$
最大次正规数（Max subnorm）	$S.0000.111_2 = 0.875 \times 2^6$	$S.00000.11_2 = 0.75 \times 2^{14}$
最小次正规数（Min subnorm）	$S.0000.001_2 = 2^9$	$S.00000.012 = 2^{16}$

随着浮点数位数的降低，舍入误差（如“大数吃小数”）变得更加显著。例如，对于 FP16，其在不同区间的精度是不同的。在区间 [1024, 2048] 内，FP16 的最小间隔为 1。这意味着如果将 1024.0 加上 1.5，结果会被舍入为 1025.0。以下是一个简单示例来说明这种行为：

在 FP16 表示中，数值 1024.6 会被舍入为 1025.0。当用 FP16 精度计算 1025.0 加上 FP16 表示的 0.4 时，结果仍然是 1025.0，因为 0.4 太小，不足以引起值的变化。而在计算 FP16 表示的 1025.0 加上 100.6 时，结果是 1126.0，因为 100.6 足够大，能够影响计算结果。这种舍入误差在低精度浮点数中非常常见，特别是在数值范围较大的情况下，这可能会对模型训练和推理的数值稳定性产生显著影响，这也是低精度训练最需要解决的难点之一。

10.3.2 FP8 大模型训练

Nvidia Transformer Engine 在版本 1.1.0 版本中支持 GEMM 计算中应用了 FP8。然而，它仍然采用高精度格式（如 FP16 或 FP32）来存储主权重和梯度，因此在端到端的速度提升、内存节省和通信成本优化方面效果有限，未能充分挖掘 FP8 的潜力。为了解决这一问题，Microsoft Azure 和 Microsoft Research 的研究人员开源了 FP8-LM 框架 [530]。该框架提出了一种高度优化的 FP8 混合精度训练方法，专为大语言模型设计。其核心思想是将 FP8 的计算、存储和通信贯穿于大型模型训练的全过程，使前向传播和反向传播全程基于低精度 FP8，从而显著降低系统工作负载，并实现更高效的训练过程。2025 年 1 月，文献 [531] 提出的方法，更是将精度进一步降低到 FP4。

使用 FP8 进行大语言模型的训练并非易事，主要面临数据下溢或上溢问题，以及因 FP8 数据格式动态范围较窄和精度较低而引发的量化误差，这些问题可能导致数值不稳定性，甚至在训练过程中出现不可逆的发散现象。为了解决这些挑战，文献 [530] 指出，在大语言模型的训练中，大部分变量（如梯度、优化器状态）可以采用低精度数据格式，而不会影响模型的准确性，也无需调整超参数。具体而言，FP8-LM 提出了三个优化级别，通过逐步引入 FP8 通信、FP8 优化器以及 FP8 分布式并行训练，简化混合精度和分布式训练流程。这三个优化级别逐步扩大 FP8 在大语言模型训练中的应用比例，优化级别越高，训练过程中对 FP8 的依赖越强。此外，FP8-LM 框架还支

持 FP8 的低位并行化，包括张量并行、流水线并行和序列并行。

1. FP8 梯度和 AllReduce 通信

现有的混合精度训练方法通常采用 16 位或 32 位数据类型来计算和存储梯度[532]，这导致整个训练过程中集体通信对带宽的需求非常高。然而，直接将 FP8 应用于梯度会引发精度下降的问题，主要原因在于低精度全局归约（Low-bit All-Reduce）操作中容易出现下溢和上溢问题。

具体而言，在全局归约过程中，跨 GPU 聚合梯度通常有两种标准方法：预缩放（Pre-scaling）和后缩放（Post-scaling）。预缩放方法是在求和之前，将第 i 个 GPU 计算出的梯度 g_i 除以 GPU 总数 N，其公式如下：

$$g = g_1/N + g_2/N + + g_N/N \tag{10.12}$$

当 N 较大时，这种除法可能导致数据下溢，尤其是在使用 FP8 低精度表示梯度时。后缩放方法则先对梯度进行求和，然后在梯度收集的过程中进行除法缩放，公式为：

$$g = (g_1 + g_2 + + g_N)/N \tag{10.13}$$

后缩放方法使梯度值接近 FP8 数据类型的最大值，有效缓解了下溢问题。但与此同时，这种方法在梯度聚合时容易引发上溢问题。

针对上述问题，FP8-LM[530] 提出了一种自动缩放（Automatic Scaling）技术，以同时解决预缩放和后缩放方法中的下溢和上溢问题。该方法通过引入一个动态变化的自动缩放因子 μ，在训练过程中对梯度值进行适应性调整，从而减少梯度中上溢和下溢的情况。其核心公式为：

$$g_i' = \mu \cdot g_i \tag{10.14}$$

对 g_i' 的梯度值进行统计分析，旨在量化在 FP8 表示范围内达到最大可行值的数值比例。如果该比例超过指定阈值（例如 0.001%），则在后续训练步骤中将缩放因子 μ 减半（设置为 $\mu/2$），以降低上溢的风险。相反，如果该比例始终低于阈值，则在 1000 个训练步骤的时间跨度内逐步将 μ 按指数规律增加到原值的 2，从而有效降低下溢风险。这种动态调整机制能够根据实际梯度分布灵活调整 μ，在缓解上溢和下溢问题的同时，保证 FP8 精度下的数值稳定性。

FP8 集合通信（collective communication）的另一个关键挑战在于设计一种高效策略来管理与每个梯度张量相关的张量级缩放因子。然而，目前的 NCCL 实现尚不支持在全规约操作中引入额外的张量级缩放因子。同时，实现这一功能的效率也面临巨大挑战，特别是考虑到 NCCL 对梯度的求和操作是在子张量级别完成的。当需要纳入张量级缩放因子的更新时，操作的复杂性会显著增加。

为了解决这一问题，FP8-LM 提出了一种方法，采用单个共享标量对跨 GPU 的 FP8 梯度进行统一缩放。具体来说，设 (g_i', s_i') 为一个缩放张量，其中 g_i' 是第 i 个 GPU 上存储的 FP8 权重梯度

张量，s'_i 是对应的缩放因子。实际的权重梯度可以表示为 g'_i/s'_i。

在执行梯度张量的全局归约操作之前，需要先收集所有 GPU 上每个梯度张量的缩放因子 s'_i，并计算出一个全局最小缩放因子 s'_g。其计算公式为：

$$s'_g = min(s'_1, s'_2, ..., s'_N) \tag{10.15}$$

全局最小缩放因子 s'_g 在所有 GPU 间共享。利用该共享缩放因子 s'_g 对跨 GPU 的梯度张量进行统一重新缩放。通过这种方式，与同一权重相关的所有梯度张量在所有 GPU 上都使用相同的共享缩放因子，将张量量化为 FP8 格式：

$$g''_i = \text{FP8}[s'_g(g'_i/s'_i)] \tag{10.16}$$

这种方法通过仅传输单个标量 s'_g 来显著降低通信开销，从而使额外的同步步骤变得非常高效。由于所有输入张量共享相同的缩放因子，无需并行处理缩放因子的全规约操作，可以直接执行标准的 NCCL 全局规约操作。最终收集到的梯度通过以下方式获得：

$$g = g''_1 + g''_2 + + g''_N \tag{10.17}$$

$$s = N \cdot s'_g \tag{10.18}$$

其中，g 表示最终聚合的梯度，s 是对应的缩放因子。从理论上讲，对聚合后的梯度 g 进行缩放等价于将 g 除以 N。通过实施上述分布式与自动缩放相结合的策略，可以在保持模型精度的同时，实现 FP8 低位梯度通信的有效性。此外，该方法通过以 FP8 格式存储梯度并进行通信，大幅降低了 GPU 内存使用量和通信带宽消耗。

2. FP8 优化器

在大语言模型的训练中，Adam[533] 及其变体是最常用的优化算法。这些方法会存储模型权重、梯度，以及一阶和二阶梯度矩的副本，用于更新模型参数。在混合精度训练中[532]，使用 Adam 优化器时通常以 32 位浮点格式存储主权重、梯度和梯度矩，以确保数值稳定性。因此，在训练过程中，Adam 优化器的每个参数需要消耗 16 字节的内存：

$$\underbrace{4}_{\text{主权重}} + \underbrace{4}_{\text{梯度}} + \underbrace{4+4}_{\text{Adam 状态}} = 16\text{字节} \tag{10.19}$$

当模型规模较大时，Adam 优化器中内存消耗会成为一个瓶颈。先前的研究[534] 表明，在训练规模达到数十亿参数的模型时，将优化器变量的精度降低到 16 位可能会导致模型精度下降。因此，需要评估优化器中的哪些变量需要保留高精度存储，以及哪些变量可以使用低精度存储。

FP8-LLM 的研究对优化器中变量的精度需求进行了深入分析，探讨了哪些变量可以分配较低

的精度。研究提出了一个指导原则：梯度统计量可以使用较低的精度，而主权重则需要分配较高的精度。具体而言，一阶梯度矩能够容忍较大的量化误差，因此可以使用低精度的 FP8 格式，而二阶梯度矩则需要更高的精度。这是因为在 Adam 的模型更新过程中，梯度的方向比其大小更为关键。尽管带有张量缩放的 FP8 格式在一定程度上会引入精度损失，但它能够有效保持一阶矩的分布，与高精度张量几乎一致。此外，由于梯度值通常较小，在计算二阶梯度矩时对梯度进行平方运算可能导致数据下溢。为了避免数值不稳定性并保持精度，二阶梯度矩需要分配 16 位的较高精度存储。

另一方面，FP8-LM 的研究团队发现保持主权重使用高精度存储至关重要。其主要原因在于训练过程中，权重更新的幅度可能会变得极小或极大，为主权重分配更高的精度能够有效防止信息丢失，从而确保训练的稳定性和准确性。在实现中，主权重有两种可行的存储方案：使用 FP32 全精度或带有张量缩放的 FP16。相比之下，带有张量缩放的 FP16 在不显著降低精度的同时，可以显著节省内存。因此，FM8-LM 默认选择在优化器中使用带有张量缩放的 FP16 来存储主权重。通过这一设计，FM8-LM 的 FP8 混合精度优化器在训练过程中，每个参数仅消耗 6 字节的内存：

$$\underbrace{2}_{\text{主权重}} + \underbrace{1}_{\text{梯度}} + \underbrace{1+2}_{\text{Adam 状态}} = 6\text{字节} \tag{10.20}$$

3. FP8 分布式并行训练

训练大语言模型需要分布式学习策略，以实现跨多 GPU 的并行化。常用的策略包括数据并行（Data Parallelism）、张量并行（Tensor Parallelism）、流水线并行（Pipeline Parallelism）以及序列并行（Sequence Parallelism）。每种并行策略都有其优点，并在现有系统中以互补的方式使用。对于这些策略的 FP8 支持而言，数据并行和流水线并行无需进行任何特定的修改，因为在将数据批次或模型层拆分到不同设备时，这两种策略并不涉及额外的 FP8 计算和通信。

张量并行将模型的单个层划分到多个设备上，使得权重、梯度和激活张量的分片分布在不同的 GPU 上，而不是集中在单个 GPU 上。为了在张量并行中支持 FP8，FP8-LM 将分片的权重和激活张量转换为 FP8 格式，用于线性层的计算，从而使前向计算和反向梯度的集合通信都可以使用 FP8 格式。

另一方面，序列并行通过将输入序列拆分为多个子序列，并将这些子序列分配到不同的设备上，从而有效节省激活内存，如图 10.11 所示，其中橙色部分突出显示了 FP8 低精度操作。序列并行和张量并行针对 Transformer 模型的不同部分同时执行，以最大化内存利用率并提高训练效率。在序列并行区域与张量并行区域之间，有一个转换器 g，用于在前向传播中执行全收集（All-Gather）序列分区，或在反向传播中执行规约-散播（Reduce-Scatter）张量分片。为进一步降低通信成本，在 g 之前添加了 FP8 数据类型转换，使全收集（或规约-散播）操作能够利用 FP8 低精度激活值，从而显著减少跨 GPU 的通信开销。

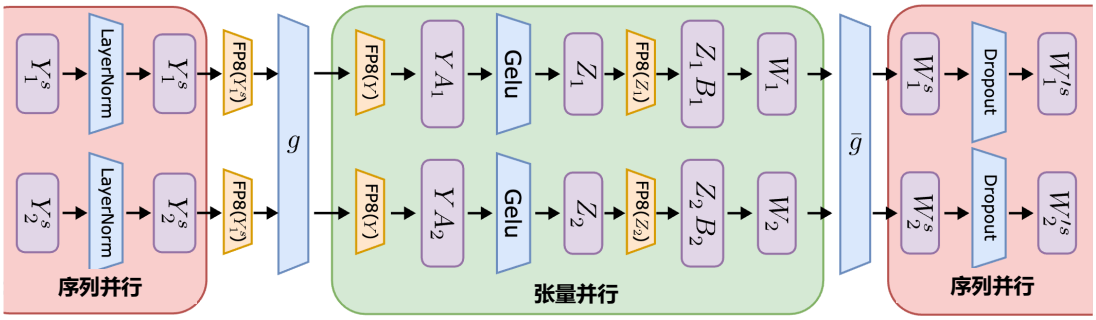


图 10.11 采用 FP8 张量和序列并行的 Transformer 层[530]

零冗余优化器（Zero Redundancy Data Parallelism，ZeRO）[173–175] 也是大模型训练中的另一种常用分布式学习技术。ZeRO 的核心思想是将模型状态分片到各设备，使每个设备仅保存训练步骤所需数据（如主权重、梯度和优化器状态）的一部分。为了减少内存消耗，ZeRO 方法通常将单个张量分割为多个子张量，并将其分布到不同的设备上。

直接将 FP8 应用于 ZeRO 也是不可行的，因为难以处理与 FP8 分片相关的缩放因子。每个张量的缩放因子需要与 FP8 分片一起分发。为了解决这一问题，FP8-LM 实现了一种新的 FP8 分布方案，该方案将整个张量分布到设备上，而不是像 ZeRO 那样将张量分割为多个子张量进行分布。FP8 张量的分布采用贪婪策略，具体过程如算法 1 所述。具体来说，我们的方法首先根据张量状态的大小对其进行排序，然后根据每个 GPU 的剩余内存大小将张量分发到不同的 GPU。分布遵循一个原则：剩余内存较大的 GPU 优先接收新的分布张量。

通过这种方式，可以将张量的缩放因子与张量一并顺利分发，同时降低通信和计算复杂性。图10.12展示了在包含和不包含缩放因子的情况下，ZeRO 张量分片方式的差异。ZeRO 张量分区方式可以分为两种：有缩放因子和无缩放因子。左图展示了原始的高精度 ZeRO 方法，其中一个张量被分割成多个分区后分配到不同的设备上。右图展示了提出的 FP8 ZeRO 方法，该方法将每个张量的完整副本分配到设备上，同时保留并考虑张量的缩放因子。

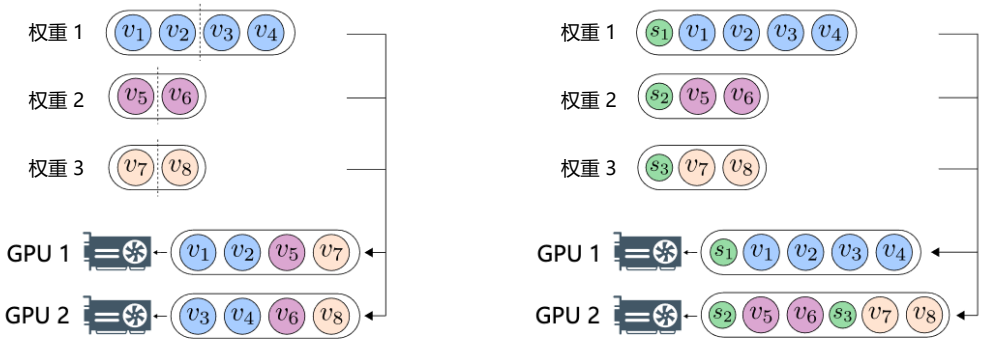


图 10.12　ZeRO 张量分片方式的差异示意图[530]

10.4 高效推理

高效的推理技术主要致力于降低大语言模型在推理过程中的计算成本和资源消耗，从而提高推理的速度和效率。这些技术可以大致分为算法级别和系统级别两个方面。

算法级别高效推理常涉及优化模型本身的结构或推理方法，以减少计算复杂度。例如，推测解码，通过生成多个候选结果并快速筛选以减少推理时间。另一个关键技术是 KV-cache 优化，通过高效存储和重用注意力机制中的键值对，显著降低计算开销。

系统级别的高效推理则关注优化推理的硬件和软件环境，以更高效地执行模型的计算任务。例如，模型的分布式推理可以将计算任务分配到多个 GPU 或 TPU 上，以并行化执行，需要结合硬件资源（GPU、CPU 和磁盘）以及对内存和计算的优化。

通过结合算法级别和系统级别的优化方法，可以在保持模型性能的同时，大幅降低推理成本，从而使大语言模型在实际应用中更加高效和实用。本节将分别介绍算法级别和系统级别推理优化方法。

10.4.1 算法级别推理优化

算法级别的推理效率优化主要通过改进算法机制来提升推理性能，主要集中在推测解码和 KV-缓存优化两个方面。推测解码通过在生成过程中引入预测机制，利用小模型或轻量化计算模块快速生成候选结果，从而提升推理速度。KV-缓存优化则针对注意力机制中存储和访问键值缓存的效率问题。本节将分别介绍上述两类算法级别推理优化方法。

1. 推测解码

推测解码（Speculative Decoding），也称**投机采样**，是一种专为自回归大语言模型设计的解码技术，旨在不降低生成质量的前提下显著提升解码效率。其核心思想是引入一个较小的模型，称为草稿模型（Draft Model），快速预测多个候选词元（Draft Tokens），然后由目标大语言模型对这

些预测结果进行并行验证，从而实现效率提升。通过这种方法，大语言模型能够在单次推理的时间内生成多个词元，如图10.13所示。

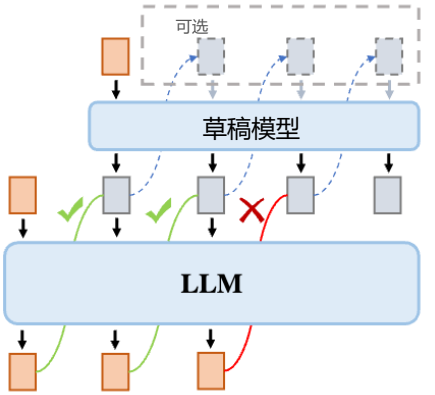


图 10.13 推测解码示意图[471]

具体而言，推测解码包含两个主要步骤：（1）草稿生成：利用草稿模型以并行或自回归的方式高效生成一批候选词元，即草稿词元；（2）草稿验证：目标模型在单次推理步骤中计算所有草稿词元的条件概率，并按顺序验证每个词元是否符合分布要求，确定其是否被接受。推测解码的性能通常通过接受率来衡量，即每次推理步骤中被接受的草稿词元的平均数量。接受率越高，推测解码的效率提升越显著。这一方法有效利用了草稿模型的预测能力和目标模型的验证能力，实现了生成速度与输出质量的良好平衡。

推测解码目标是能够确保生成的输出与标准自回归解码方法保持等效性。传统的解码技术通常采用两种主要的采样策略：贪婪采样（Greedy Sampling）和核采样（Nucleus Sampling）。贪婪采样在每个解码步骤中选择概率最高的词元，从而生成确定性的输出序列。Blockwise Parallel Decoding是该方向的早期代表性工作之一[535]，其目标是确保草稿词元与通过贪婪采样生成的词元完全一致，从而严格保持输出的等效性。相比之下，核采样则从概率分布中随机采样词元，每次运行可能产生不同的词元序列。这种随机性为生成结果带来了更大的多样性，因此被广泛应用于需要丰富输出的场景中。

为了在推测解码框架中适配核采样，文献 [536, 537] 提出了推测采样（Speculative Sampling）技术。推测采样在保持输出分布等效性的同时，与核采样的概率特性一致，从而能够生成多样化的词元序列。形式上，假设给定的词元序列为 $x_1, x_2, \ldots, x_n$ ，草稿模型生成的草稿词元序列为 $\hat{x}_{n+1}, \hat{x}_{n+2}, \ldots, \hat{x}_{n+k}$，推测采样的策略是根据以下概率接受第 i 个草稿词元：

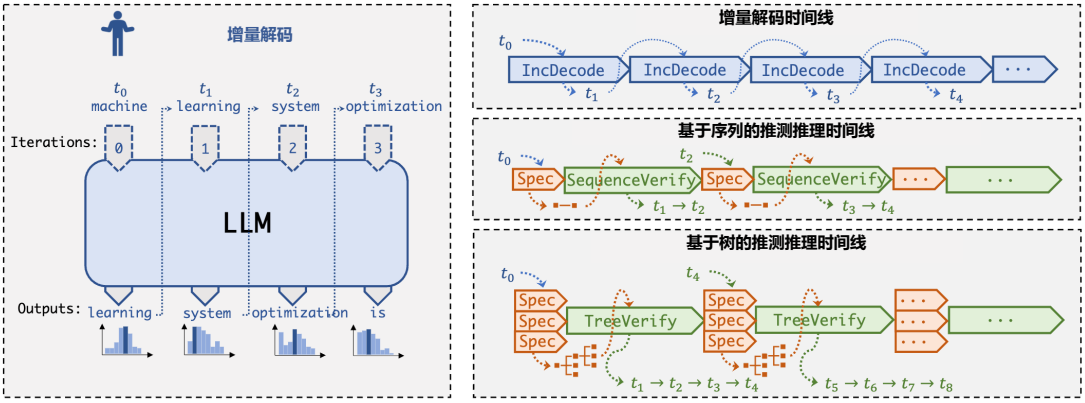
$$\min\left(1, \frac{p(\hat{x}_i|x_1, x_2, \ldots, x_{i-1})}{q(\hat{x}_i|x_1, x_2, \ldots, x_{i-1})}\right) \tag{10.21}$$

其中，$p(\cdot|\cdot)$ 和 $q(\cdot|\cdot)$ 分别表示目标大语言模型和草稿模型的条件概率。如果第 i 个草稿词元被接受，则将其设置为 $x_i \leftarrow \hat{x}_i$。如果未被接受，则停止验证后续草稿词元，并从以下分布中重新采样 x_i：

$$\text{norm}\left(\max\left(0, p(\cdot|x_1, x_2, \ldots, x_{i-1}) - q(\cdot|x_1, x_2, \ldots, x_{i-1})\right)\right) \tag{10.22}$$

基于推测采样，衍生出了多种变体方法[538, 539]，这些方法的目标是验证多个草稿词元序列。

SpecInfer[538] 提出了基于树的推测解码和验证（Tree-based Speculative Inference and Verification）框架。增量解码、基于序列的推测推理以及基于树的推测推理之间的对比如图10.14所示。



(a) 增量解码　　(b) 时间线对比

图 10.14　增量解码、基于序列的推测推理以及基于树的推测推理示意图[471]

SpecInfer 算法的核心在于利用小模型预测目标大语言模型的输出，并将这些预测组织为词元树结构。词元树的每个节点表示一个候选词元序列，通过基于树的并行解码机制，同时验证所有候选词元序列的正确性。为最大化推测性能，需要探索极其庞大的候选词元序列搜索空间。目前的大语言模型通常涉及非常大的词汇表，例如，Qwen 2.5 的词汇表大小达到了 15.16 万[136]，而 SpecInfer 平均能够正确预测接下来的 4 个词元。因此，需要处理一个包含 $151643^4 \approx 5.29 \times 10^{20}$ 个可能词元组合的搜索空间。

为了解决上述问题，首先需要使用大语言模型现有的提炼、量化和/或剪枝变体，构造小推测模型（Small Speculative model，SSM），来指导推测。使用 SSM 进行推测推理的一个关键挑战在于，由于 SSM 通常比大语言模型小 100 - 1000 倍，SSM 与大语言模型之间的一致性本质上受到模型能力差距的限制。SpecInfer 通过同时考虑针对给定输入提示以树结构组织的各种词元序列，来最大化推测性能。分别通过利用单个 SSM 内部以及多个 SSM 之间的多样性，引入了基于扩展和基于合并的两种机制来构建词元树。

基于扩展的词元树构建方法通过在单次解码步骤中从小型推测模型（SSM）生成多个词元来

构建词元树。这一方法的核心在于观察到，当 SSM 与大语言模型）出现不一致时（即两者选择的 Top-1 词元不同），LLM 选择的词元通常出现在 SSM 的 Top-K 词元中，且 K 值较小。但是，如果直接在每一步都选择 Top-K 词元会导致潜在词元序列数量呈指数增长，显著增加推理延迟和内存开销。因此，SpecInfer 采用了一种静态扩展策略，以预设的扩展配置表示为向量 $< k_1, k_2, ..., k_m >$，其中 m 为最大推测解码步数，k_i 表示第 i 步每个词元的扩展数量。例如，扩展配置 $< 2, 2, 1 >$ 会生成 4 个词元序列。

基于合并的词元树构建通过整合多个 SSM 来协同预测大语言模型的输出。SpecInfer 采用无监督方法，通过自适应提升（Adaptive Boosting）对多个 SSM 进行联合优化，使它们的输出与 LLM 的结果更为一致。在此过程中，SpecInfer 利用通用文本数据集（如 OpenWebText 语料库），将文本数据转换为一系列提示样本，并通过 LLM 生成相应的词元序列。具体而言，SpecInfer 逐一对每个 SSM 进行全面微调，词元那些提示样本中 SSM 与 LLM 生成词元完全一致的部分；接着，过滤已词元的提示样本，利用剩下的样本对下一个 SSM 进行微调。通过重复这一流程，SpecInfer 生成了一组多样化的 SSM，它们的联合输出在训练数据上能够与 LLM 的结果实现高度一致性。

SpecInfer 使用基于树的并行解码来计算其树注意力，为了能够在词元树上进行并行化的验证，SpecInfer 提出了一种树形注意力（Tree Attention）计算方法，通过构造的掩码矩阵和基于深度优先的 KV-cache 更新机制，验证器可以在不增加额外存储的同时，尽可能并行化树中每一条路径的解码过程。相比于朴素的逐序列或逐词元的解码方式，树形解码可以同时在内存开销和计算效率上达到最优。对于给定的推测词元树 $\mathcal{N}$，SpecInfer 使用基于树的并行解码来计算其树注意力，并生成一个输出张量 $\mathcal{O}$，该张量为 $\mathcal{N}$ 中的每个节点 u 都包含一个词元。SpecInfer 的词元树验证器对照大语言模型检查推测词元的正确性 SpecInfer 同时支持贪心解码和随机采样。

一些大语言模型使用贪心解码生成词元，即在每个解码步骤中贪心选择可能性最高的词元。针对此类模型，SpecInfer 从 $\mathcal{N}$ 的根节点开始，迭代地对照大语言模型的原始输出检查节点的推测结果。对于 $\mathcal{N}$ 中的节点 u，如果 u 包含一个子节点 v（即 $p_v = u$），且其词元与大语言模型的输出匹配（即 $t_v = \mathcal{O}(u)$），那么 SpecInfer 就成功推测出其下一个词元。在这种情况下，SpecInfer 完成对节点 u 的验证，并继续检查其子节点 v。当节点 u 不包含与大语言模型输出匹配的子节点时，SpecInfer 将 $\mathcal{O}(u)$ 作为已验证节点添加到 $\mathcal{N}$ 中，并终止验证过程。最后，所有已验证节点追加到当前生成的词元序列 $\mathcal{V}$ 中。词元树验证使 SpecInfer 能够机会性地解码多个词元，同时保持与增量解码相同的生成性能。

为了提高生成词元的多样性，许多大语言模型采用随机解码，即从概率分布 $P(u_i|U; \Theta_{LLM})$ 中采样一个词元，其中 $U = u_0, \ldots, u_{i-1}$ 是此前生成的词元，u_i 是要生成的下一个词元，Θ_{LLM} 表示参数化的大语言模型。为了使用随机解码验证推测词元树，SpecInfer 引入了一种多步推测采样（Multi-step Speculative Sampling，MSS）算法来进行验证。对于词元树 $\mathcal{N}$ 中的非叶子节点，对比多个 SSM 输出与大语言模型输出的概率，$\frac{P(x_s|u,\Theta_{LLM})}{P(x_s|u,\Theta_{SSM_s})}$，在一定范围之内就可以通过验证。

在推测解码方法中，草稿词元的接受率在很大程度上取决于草稿模型与目标大语言模型输出

分布的对齐程度。因此，为了提升推测解码的效率和准确性，许多研究集中在改进草稿模型的设计上。DistillSpec[540] 提出了一种直接从目标大语言模型中提炼草稿模型的方法，通过蒸馏技术生成一个更小、更高效的草稿模型，以提高推测解码的计算效率。与此类似，SSD[541] 提供了一种自动化的解决方案，它从目标模型的层结构中识别一个子模型（即部分层的子集）作为草稿模型，而无需对草稿模型进行单独训练，从而简化了模型设计流程。在动态优化方面，OSD[542] 针对在线大语言模型服务，提出了一种在线提炼方法。通过监控大语言模型拒绝的草稿词元，OSD 能够动态调整草稿模型的输出分布，使其更贴合用户查询分布，从而提升推测解码的性能。此外，PaSS[543] 提议直接使用目标大语言模型本身作为草稿模型，通过在输入序列中添加可训练的前瞻词元（Lookahead Tokens），使模型能够在生成后续词元的同时优化草稿生成，从而降低复杂度。另一种创新性方法是 REST[544]，它引入了基于检索的推测解码机制，使用非参数化的检索数据存储作为草稿模型，使解码过程更加灵活且高效。Kangaroo[545] 提出了以轻量化为目标的设计思路。该方法固定目标模型的一个浅层子网作为草稿模型，并在子网之上训练一个轻量级的适配器模块。这种方式避免了单独训练草稿模型的需求，同时保持了较高的推测解码性能。

2. KV-缓存优化

在推理过程中，大语言模型需要将过去生成的词元键值对（Key-Value，KV）存储到缓存中，以便生成未来的词元。随着生成词元长度的增加，所需的 KV 缓存大小会急剧增长，从而导致显著的内存消耗和较长的推理延迟。因此，减少 KV 缓存的大小是提升推理效率的关键。现有的 KV-缓存优化技术主要分为两类：缓存压缩和缓存清理。

KIVI[546] 是一种无需调优的 2bit KV 缓存压缩算法。通过对 KV 缓存的深入分析，KIVI 针对键缓存（Key Cache）和值缓存（Value Cache）的不同分布特性。键缓存中一些固定通道幅度非常大，每个通道内存在持续的异常值，逐通道量化可以将量化误差限制在每个通道内，不影响其他正常通道。值缓存没有明显的异常值，且由于注意力分数高度稀疏，输出是一些重要词元的值缓存组合，按词元量化可以将误差限制在每个单独的词元上，量化其他词元不会影响重要词元的准确性，相对误差更小。

根据上述分析提出了独特的量化策略：键缓存采用按通道（Per-channel）量化，以应对少数固定通道的大幅值问题；值缓存则基于按词元（Per-token）量化，以适应注意力计算中按词元混合的特性。将每 G 个词元的键缓存分组并分别进行量化。把当前键缓存中的词元分成分组部分和余留部分，分组部分可均匀分为组，只存储分组量化的结果，残差部分保持全精度。在解码过程中，新到达的键缓存添加到余留部分，到达一定数量（超参数余留长度 r）的词元后，将其量化并与之前量化的连接起来，然后重置余留部分为空张量。将值缓存也分为两部分，维护一个队列，新到达的值缓存推入队列，达到预定义的余留长度 r 时，弹出最久的值缓存，按词元进行量化后与先前量化的值缓存连接。实验结果表明，KIVI 在 Llama、Mistral 和 Falcon 等模型的主流生成任务中表现出色，可将 KV 缓存压缩至 2bit，带来高达 2.6 倍的峰值内存使用减少，同时几乎不影响生成

性能。

Heavy-Hitter Oracle（H_2O）[547] 提出了一种 KV 缓存清理策略，将缓存管理问题建模为动态次模优化问题（Dynamic Submodular Problem），通过动态保留近期生成的词元和性能关键的词元，从而显著提升大语言模型（LLM）推理的吞吐量。在次模性中，随着已选择元素数量的增加，添加新元素所带来的边际收益会递减。在 KV 缓存场景中，每个词元对模型性能的贡献可以看作一种收益。H_2O 动态评估每个词元的重要性，在保留近期生成词元（因其与当前生成任务关系密切）和关键性能词元之间找到平衡。其核心在于识别出那些频繁使用或对模型输出质量影响较大的“重命中”（heavy-hitters，H_2）词元，优先保留这些词元的 KV 对于缓存中，而将不重要的 KV 对逐出。通过这样的策略，H_2O 能够在有限的缓存空间内最大化利用率，提高推理效率和输出质量。这种动态平衡策略有效缓解了由于 KV 缓存过大而导致的性能瓶颈问题，使得 LLM 能够在单位时间内处理更多输入或生成更多输出，从而显著提升推理的吞吐量。

StreamingLLM[548] 发现大语言模型中存在“注意力吸槽”（Attention Sink）现象，模型在注意力机制中倾向于将大量的注意力分数集中于序列最初的几个词元，即便这些词元在语义上并不重要。StreamingLLM 提出了通过保留这些“注意力吸槽”词元的 KV 值来稳定注意力计算的方法。这些词元的 KV 值作为锚点，帮助注意力机制在后续计算中保持稳定性，从而避免因注意力分布的异常而导致的性能下降。为了进一步优化长文本处理的效率和内存使用，StreamingLLM 引入了滑动窗口机制。这种机制动态缓存最近一段时间生成的词元的 KV 状态，定期清理过往不再需要的 KV 值。这种策略不仅能够显著降低内存消耗，还能在处理长文本时保持解码速度的稳定性。为了增强生成响应的相关性和连贯性，StreamingLLM 没有完全依赖原始文本中的绝对位置，而是使用相对于缓存中位置的相对位置编码。这种设计使得模型能够更有效地捕捉上下文关系，减少长文本生成中位置偏移对注意力计算的负面影响。

10.4.2 系统级别推理优化

在经过语言模型预训练、指令微调及基于强化学习的类人对齐之后，以 ChatGPT 为代表的大语言模型能够与用户以对话的方式进行交互。用户输入提示词之后，模型迭代输出回复结果。虽然大语言模型通过这种人机交互方式可以解决翻译、问答、摘要、情感分析、创意写作和领域特定问答等各种任务，但这种人机交互方式对底层推理服务提出了非常高的要求。许多用户可能同时向大语言模型发送请求，并期望尽快获得响应。因此，低作业完成时间（Job Completion Time，JCT）对于交互式大语言模型应用至关重要。

随着深度神经网络大规模应用于各类任务，针对深度神经网络的推理服务系统也不断涌现，Google 公司在开放 TensorFlow 框架后不久也开放了其推理服务系统 TensorFlow Serving[549]。NVIDIA 公司也于 2019 年开放了 Triton Inference Server[550]。针对深度神经网络的推理服务系统也是近年来计算机体系结构和人工智能领域的研究热点，自 2021 年以来，包括 Clockwork[551]、Shepherd[552] 等在内的推理服务系统也陆续被推出。推理服务系统作为底层执行引擎，将深度学习模型推理阶段

进行了抽象，对深度学习模型来说是透明的，主要完成对作业进行排队、根据计算资源的可用情况分配作业、将结果返回给客户端等功能。由于像 GPU 这样的加速器具有大量的并行计算单元，推理服务系统通常会对作业进行批处理，以提高硬件利用率和系统吞吐量。启用批处理后，来自多个作业的输入会被合并在一起，并作为整体输入模型。但是此前推理服务系统主要针对确定性模型进行推理任务，它们依赖于准确的执行时间分析来进行调度决策，而这对于具有可变执行时间的大语言模型推理并不适用。此外，批处理与单个作业执行相比，内存开销更大。由于内存开销与模型大小成比例增长，因此大语言模型的尺寸限制了其推理的最大批处理数量。

目前，已经有一些深度神经网络推理服务系统针对生成式预训练大语言模型 GPT 的独特架构和迭代生成模式进行优化。

另一个研究方向是针对作业调度进行优化。传统的作业调度将作业按照批次执行，直到一个批次中的所有作业完成，才进行下一次调度。这会造成提前完成的作业无法返回给客户端，而新到达的作业则必须等待当前批次完成。针对大语言模型，Orca[553] 提出了迭代级（Iteration-level）调度策略。在每个批次上只运行单个迭代，即每个作业仅生成一个词元。每个迭代执行完后，完成的作业可以离开批次，新到达的作业可以加入批次。Orca 采用先到先服务（First-Come-First-Served，FCFS）策略来处理推理作业，即一旦某个作业被调度，它就会一直运行直到完成。批次大小受到 GPU 显存容量的限制，不能无限制地增加批次中的作业数量。这种完全运行处理（Run-to-completion）策略存在头部阻塞（Head-of-line blocking）问题[554]。对于大语言模型推理作业来说，这个问题尤为严重，这是因为，一方面大语言模型的计算量大，导致了较长的绝对执行时间；另一方面，一些输出长度较长的作业将会运行很长时间，很容易阻塞后续的短作业。这种问题非常影响交互式应用的低延迟要求的达成。

FastServe[472] 系统是由北京大学的研究人员开发的，针对大语言模型的分布式推理服务进行了设计和优化。整体系统设计目标包含以下三个方面。

（1）低作业完成时间：专注于交互式大语言模型应用，用户希望作业能够快速完成，系统应该在处理推理作业时实现低作业完成时间。

（2）高效的 GPU 显存管理：大语言模型的参数和键值缓存占用了大量的 GPU 显存，系统应该有效地管理 GPU 显存，以存储模型和中间状态。

（3）可扩展的分布式系统：大语言模型需要多块 GPU 以分布式方式进行推理，系统需要可扩展的分布式系统，以处理大语言模型的推理作业。

FastServe 的整体框架如图10.15 所示。用户将作业提交到作业池（Job Pool）中，跳跃连接多级反馈队列（Skip-join MLFQ）调度器使用作业分析器（Job Profiler）根据作业启动阶段的执行时间决定新到达作业的初始优先级。FastServe 作业调度采用迭代级抢占策略，并使用最小者（Least-attained）优先策略，以解决头部阻塞问题。一旦选择执行某个作业，调度器会将其发送到分布式执行引擎（Distributed Execution Engine），该引擎调度 GPU 集群为大语言模型提供服务，并与分布式键值缓存（Distributed Key-Value Cache）进行交互，在整个运行阶段检索和更新相应作业的键值张量。为了解决 GPU 显存