背景与动机

随着LLM参数规模巨增，模型推理的计算和内存开销成为主要瓶颈。在Transformer结构（如LLaMA、Mistral）的解码过程中，每一步推理需要从显存不断读取权重到计算单元，受制于内存带宽而memory-bound。传统加速手段包括权重量化和权重剪枝等，直接压缩模型权重以减少每步计算量。然而，权重压缩对所有输入一视同仁，无法根据不同输入的实际需要进行动态调节。

激活稀疏化（Activation Sparsity）为推理加速提供了一种数据依赖的动态方案：通过使中间激活值变为零，可跳过对应权重通道的内存传输和乘法运算，从而加速推理。早期如GPT-2、OPT等使用ReLU激活函数的模型，其全连接层中大量神经元天然输出零值，激活呈现高稀疏性（>85%）。利用这一特性，DejaVu方法训练了一个小型辅助模型预测后续层哪些通道会为零，从而在不损失准确率的情况下跳过约95%的通道计算，实现了接近2倍的速度提升。然而，新式LLM（如LLaMA、Mistral）多采用SwiGLU/GeLU等非ReLU激活，激活值几乎全为非零（>99%激活为非零），自然稀疏性消失，直接应用旧方法效果不佳。近期有工作尝试将激活函数重新换回ReLU并继续预训练来恢复稀疏，但需要极大代价的再训练。因此，如何在无需额外训练的情况下诱导出激活稀疏成为一个关键研究方向。

已有的训练后激活剪枝方法如CATS和TEAL证明了在不微调模型的前提下，也可通过简单策略获得可观的激活稀疏率。例如，Stanford提出的CATS方法对LLaMA-7B和Mistral-7B的前馈层采用阈值裁剪，在不微调的情况下使MLP部分达到50%的稀疏率。Together AI提出的TEAL方法进一步在整个模型的隐藏状态上应用幅度裁剪，实现了40-50%的模型整体激活稀疏率，并且性能几乎无损。借助定制的稀疏算子，TEAL在40%和50%模型稀疏率下分别获得了1.53×和1.8×的推理加速比。例如，在LLaMA-3-8B上，50%激活稀疏仅使WikiText困惑度从5.87增至6.67（<15%的劣化）。这些成果表明激活稀疏在大型Transformer模型上具有巨大的加速潜力。然而，现有方法也存在一些局限：(1) 阈值裁剪通常使用静态阈值或固定稀疏率，未根据不同数据的分布差异自适应调整；(2) 剪枝准则多以激活值幅度为依据，未考虑对应权重大小，对实际重要性的度量可能不够准确；(3) 各层稀疏率一般设定为统一值或简单经验分配，未充分利用不同层对误差的敏感度差异。

鉴于上述问题，本研究计划围绕推理阶段激活稀疏化展开，提出改进方案以进一步提升模型加速比和保持性能。重点关注以下三个方向：(1) 优化校准集构建或动态阈值分配，使稀疏阈值能适应数据分布（例如引入自适应的稀疏度预测器）；(2) 融合权重范数与激活幅度计算重要性分数，提高对重要激活单元的识别准确性；(3) 基于感知误差或层分布特征的逐层稀疏度优化分配，按层次动态调整稀疏率（采用对数分布等策略）。下面将分别阐述每项策略的原理与数学建模、其在Transformer网络中的作用位置、实现方案（含PyTorch示例），并将我们的方案与最新方法（如TEAL、EvoPress等）的策略和实验结果进行对比分析。

研究目标与方案概述

研究目标

实现一种适用于Transformer架构大模型（如LLaMA、Mistral）的推理加速稀疏化方案，在不重新训练模型的条件下最大限度地剪除推理时无效的激活计算，以获得显著的速度提升，同时保持模型在下游任务上的性能几乎不下降。为达到这一目标，我们将综合考虑稀疏化过程中的阈值确定、重要性判别和层间差异三个关键问题，提出相应的改进策略。

方案概述

本方案包含如下三个互补的策略，每项对应解决上述的一个关键问题：

策略一：校准集优化与动态阈值分配。改进稀疏化阈值的确定方法，利用代表性校准数据或实时数据分布，自适应地调整每层激活裁剪阈值，避免"一刀切"的静态阈值导致的性能损失。

策略二：融合权重范数与激活幅度的重要性评分。在选择保留哪些激活单元时，不仅考虑激活值的大小，还融合对应出射权重的范数来评估每个神经元对下一层输出的重要性，提升剪枝决策的精确度。

策略三：基于误差感知的逐层稀疏度分配。针对Transformer的不同层Block对稀疏化的耐受度不同，设计按层异构的稀疏率分配方案（如对数分布策略），使整体误差最小化。结合对各层感知误差或分布特征的分析，动态确定每层应有的剪枝比例。

这三项策略可以集成到统一的稀疏推理框架中：在模型每个Transformer Block的前馈网络(FFN)中，先通过策略一动态确定该层的阈值，再按照策略二计算每个激活的重要性并筛选保留，不同Block按照策略三赋予不同的稀疏率目标。下面分别详细介绍各策略的原理和技术实现。

策略一：校准集优化与动态阈值分配

原理与建模

激活稀疏化的第一步是在每层确定一个裁剪阈值：将低于该阈值的激活值置零，从而产生稀疏。简单的方法是选取一个全局固定阈值或固定比例（如保留激活值Top-k%）用于所有数据。然而，不同输入数据会产生不同分布的激活值，一个静态阈值难以对每种情况都取得最佳平衡。如果阈值设定过低，复杂输入可能保留过少信息导致精度下降；阈值过高，则简单输入又无法充分剪枝以提升速度。为此，我们引入数据分布自适应的阈值分配方案。

首先，在模型部署前可使用一个校准集来确定各层初始阈值范围。校准集应尽可能涵盖模型预期应用场景的多样输入，以捕捉激活分布的统计特性。通过在校准集样本上统计每层激活值的分布（例如均值、方差、分位数等），可以确定一个初始阈值 T^l_{base} 使得在校准数据上约有一定比例（如50%）的激活小于该阈值。与量化校准类似，我们力求选择阈值既保证较高稀疏率又不明显损伤模型输出。例如，可以选取使校准集上该层稀疏率达到目标值（如50%）时模型性能仍可接受的最大阈值作为 T^l_{base}。

在实际推理时，我们进一步实现动态阈值调整。记当前层的激活输出向量为 $\mathbf{h}^l = [h^l_1, h^l_2, \ldots, h^l_d]$（长度为隐藏单元数 l）。我们的目标是根据 $\mathbf{h}^l$ 的分布自适应地选取阈值 T^l_{dyn}。一种简单有效的方法是按照分位数设置阈值：例如希望该层保留比例为p（即稀疏率为 $1-p$），则选择 $\mathbf{h}^l$ 绝对值的第 p 分位数作为阈值。换言之，找到 T^l_{dyn} 使得 $\Pr(|h^l_i| < T^l_{\text{dyn}}) = 1-p$, 这样就有约 $100\times(1-p)\%$ 的较小值被置零，保留下最大的 $p\%$ 激活。若某输入触发该层激活整体偏大或偏小，上述分位数方法能自动调整阈值高低，确保稀疏率接近目标而不过度剪除有用信号。

另一种更智能的方法是训练一个稀疏度预测器。例如，可以构造一个轻量模型 f_θ，输入是当前层（或前几层）激活的统计特征（如均值、标准差、最大值、熵等）或上一层输出表示，输出一个建议的阈值缩放因子 α 或直接输出适合该输入的稀疏率。训练数据可由校准集生成：对每个样本，通过网格搜索不同阈值看模型输出误差，选取既满足目标稀疏率又保证误差小的阈值作为 $label$ 来训练 f_θ。推理时，f_θ 对每个输入快速预测一个调整因子用于修正各层的 T^l_{base}。例如，若输入内容复杂，预测器可能输出 $\alpha < 1$ 将阈值调低（保留更多激活），反之对简单输入提高阈值获得更多稀疏。

模块位置与作用

本策略作用于 $\mathrm{Transformer}$ 每个Block的前馈网络（FFN）部分。在典型Transformer实现中（如LLaMA），每个Block包含注意力和FFN两个子层。FFN内部通常有两个全连接层（扩张层 W_{up} 和压缩层 W_{down}）以及非线性激活（如SwiGLU）。我们在激活函数之后、W_{down} 之前插入阈值裁剪操作。也就是说，对于第 l 层Block，我们计算出激活输出 $\mathbf{h}^l$ 后，按照动态阈值策略确定该层阈值 T^l_{dyn}，将 $\mathbf{h}^l$ 中绝对值低于阈值的元素设为0，再送入下一线性层 W_{down}。这个模块相当于一个稀疏滤波器，根据当前输入情况筛选该层的重要通道。其作用是在保持模型输出质量的前提下尽量减少后续需要处理的非零激活数量，从而降低需要加载的权重量和乘法操作量。

实现示例（PyTorch）

下面的代码片段演示了如何基于激活分位数实现动态阈值裁剪。假设h是形状为(batch, d_hidden)的当前层激活张量，p是希望保留的激活比例（例如0.5表示50%激活非零）：

```
import torch
# 假设 h 为当前层激活 (batch_size, d_hidden)
p = 0.5  # 目标保留50%激活（稀疏率50%）
# 计算每个样本的阈值：对每个batch样本独立计算第p分位数
threshold_vals = torch.quantile(h.abs(), q=1-p, dim=1, keepdim=True)  # (batch_size, 1)
# 生成掩码：大于阈值的元素为1保留，反之为0剪除
mask = (h.abs() >= threshold_vals).to(h.dtype)
# 将低于阈值的激活置零
h_sparse = h * mask
```

上面代码利用 `torch.quantile` 按行计算了每个样本激活的阈值，实现按输入自适应的裁剪。实际实现中，我们可以直接在模型的forward函数内执行此操作。对于不易量化分位数的情况，也可采用排序或TopK方式获取阈值。通过策略一，模型能够针对不同输入动态调整各层的稀疏程度，避免了固定阈值在极端输入下可能造成的性能骤降, 提高稀疏化的稳健性。

策略二：融合权重范数与激活幅度的重要性评分

原理与数学建模

经典的激活剪枝标准是按激活值大小剪除——假设激活值小的神经元对最终输出影响也小，可以置零。然而，在一个前馈层中，某个隐藏单元的重要性不仅取决于它当前的激活值大小，也取决于该单元连接到下一层的权重：即使激活值适中，但如果其对应的输出权重非常大，该单元对下一层输出的贡献仍可能显著；反之，一个激活值虽大但其连接权重很小，对输出影响也有限。因此，我们提出结合激活幅度和权重大小来评估每个隐藏单元的重要性。

具体而言，考虑Transformer某个FFN层的压缩线性变换：输出 $\mathbf{o} = \mathbf{h}W_{\text{down}}$，其中 $\mathbf{h} \in \mathbb{R}^{1\times d}$ 是隐藏向量（假设批大小1以简化），$W_{\text{down}} \in \mathbb{R}^{d\times m}$ 将 d 维隐藏映射回 m 维输出。令 $W_{\text{down}}(:,i)$ 表示 W_{down} 的第 i 列向量（大小 m），对应隐藏单元 h_i 对各输出维度的权重。我们用该列权重的L2范数 $\|W_{\text{down}}(:,i)\|_2$ 来表示隐藏单元 h_i 对输出层的潜在影响力。然后定义重要性评分 I_i 为：

$$I_i = |h_i| \times \|W_{\text{down}}(:,i)\|_2$$

这个分数直观上近似刻画了「若移除单元 h_i 对输出的影响程度」：$|h_i|$ 是该单元当前激活值的幅度，$\|W(:,i)\|_2$ 表示该单元对输出各维权重的综合大小，两者乘积越大，说明该单元"带权"的激活越大，剪掉将对输出造成更大扰动。相比仅看 $|h_i|$，引入 $\|W(:,i)\|$ 可以更准确地区分那些激活小但权重大的"隐性重要"单元，以及激活大但权重小的"冗余"单元，从而提高剪枝决策的精确性。

需要注意，若采用SwiGLU等激活，FFN有"上投（W_{up}）"和"下投（W_{down}）"两路权重，且隐藏向量 $\mathbf{h}$ 由 W_{up} 和门控 W_{gate} 共同产生。在这种情况下，我们将上述评分用于 W_{down} 部分的隐藏单元（即SwiGLU激活输出后的向量）。对于每个经过门控后的隐藏单元，计算它对应的 W_{down} 列范数乘以自身幅度，得到 I_i 并作为剪除与否的依据。

模块位置与作用

本策略作用发生在Transformer每个Block的FFN激活之后，紧接策略一的阈值判定步骤。具体来说，我们在确定剪枝比例或阈值时，不再仅依据激活值绝对大小排序，而是依据重要性评分 I_i 排序。从模块实现看，这相当于在稀疏滤波器中增加了对下一层权重的"感知"。在Transformer中，这需要预先获取每层FFN的 W_{down} 权重信息，并在推理时结合当前激活计算评分。幸运的是，权重范数可以离线预计算且在推理中重复使用，不引入显著开销。通过策略二，稀疏化模块能够更准确地筛选出关键通道：避免剪掉那些激活值小但累积权重大的单元，从而将剪枝对模型输出的影响降到更低。

实现示例（PyTorch）

我们可以在模型初始化或加载后，预先计算每层压缩线性层（W_down）的列范数，并存储备用：

```python
import torch

# 模型初始化后预计算每个FFN层的权重范数
weight_norms = []  # 列表，保存每层W_down列L2范数张量
for block in model.blocks:
    W_down = block.ffn.down_proj.weight  # 假设down_proj为FFN的压缩线性层（形状：[m, d]）
    # 计算每一列（对应每个隐藏单元）的L2范数 -> 得到长度为d的向量
    # 注意：PyTorch Linear权重通常shape为(out_features, in_features)，需按列即dim=0计算
    col_norm = torch.norm(W_down, p=2, dim=0)  # (d_hidden,)
    weight_norms.append(col_norm)

# 在前向传播时，应用重要性评分裁剪（结合策略一动态阈值），以第l层为例：
# 假设 h_l 为第l层FFN激活输出 (batch_size, d_hidden)
col_norm = weight_norms[l]  # 预存的第l层每个隐藏单元的权重范数 (d_hidden,)

# 计算每个激活的带权重要性分数
importance = (h_l.abs() * col_norm)  # 广播相乘 -> (batch_size, d_hidden)

# 确定阈值：例如取保留比例p的分位数（同策略一，但基于importance排序）
threshold_vals = torch.quantile(importance, q=1-p, dim=1, keepdim=True)

# 构建掩码并剪枝
mask = (importance >= threshold_vals).to(h_l.dtype)
h_l_sparse = h_l * mask

# 后续将 h_l_sparse 输入 W_down 计算输出
output = torch.matmul(h_l_sparse, block.ffn.down_proj.weight.T) + block.ffn.down_proj.bias
```

上述代码中， `importance` 按照我们定义的公式计算了每个元素的评分，随后以相同方式选出了前 $p\%$ 高分的激活保留。通过这种实现，我们将模型结构信息（权重范数）融入剪枝决策，能够更有效地区分关键通道和冗余通道。预期在相同稀疏率下，融合重要性评分的方案相比纯激活幅度剪枝造成的准确率损失更小。这一改进可与策略一的动态阈值结合使用：无论阈值如何选择，都是依据importance大小来裁剪，从而确保对不同输入都更加稳健。

策略三：基于误差感知的逐层稀疏度分配

原理与建模

Transformer模型的各层对最终任务性能的影响并不相同：某些层较为"关键"，剪枝过多会导致模型输出误差显著上升；而另一些层则对剪枝更为耐受，可以承担更高的稀疏率而仅带来轻微精度变化。现有许多方法为了方便，通常对每层采用统一的稀疏率（例如每层都剪掉50%的激活）或简单按层深度线性变化。但非均匀分配稀疏率有望在相同整体稀疏度下取得更好的精度-速度权衡。EvoPress等最新工作通过进化搜索自动寻找每层最优压缩配置，在高稀疏率下模型困惑度显著优于均匀方案（例如Mistral-7B在70%稀疏时，EvoPress找到的配置PPL=14.42，而统一70%稀疏的PPL高达23.08）。这说明合理的逐层稀疏度分配能大幅降低剪枝带来的整体误差。

我们提出一种基于感知误差的层级稀疏分配策略：理想情况下，我们希望分配给第 l 层的稀疏率 r_l 与"剪掉该层一定比例后对模型最终误差的影响"成反比。可行的方法是在校准集上逐层进行剪枝敏感度分析：例如依次令每一层单独达到若干稀疏率（如30%、50%、70%），观察模型在验证任务上性能的下降幅度，以此评估该层对误差的敏感度曲线。敏感度低的层可以分配更高稀疏率，反之应分配低一些。由于逐层搜索开销较大，我们也可以使用各层的激活分布特征作为误差敏感度的替代指标，例如统计各层激活值的方差或最大/均值比率等：直观上，若某层激活分布中有极少数突出的大值（outliers），则剪除那些小激活可能影响较小（因为主要信息由大值承担），这样的层可以赋予更高稀疏率；反之如果某层激活较为平坦均匀，没有明显的弱激活，可以预见剪除固定比例会丢失较多有用信息，该层应保留更多激活（稀疏率设低）。

综合以上考虑，我们引入一个简单但有效的对数递增稀疏率分配方案：假定模型共有 n 个Transformer Block，需要设定从第1层($i=0$)到第 n 层($i=n-1$)各自的目标稀疏率 r_i。设定首层和末层目标稀疏率分别为 r_0 和 r_{n-1}（可由对模型两端剪枝敏感度的经验估计得到，例如前层可能稍低稀疏率，末层稍高），则第 i 层的稀疏率按层号的对数比例插值计算：

$$r_i = r_0 + \big(r_{n-1} - r_0\big) \times \frac{\log(i+1)}{\log(n)}$$

这一公式给予前几层较低的稀疏率、后几层较高的稀疏率，并且增加趋势在前期较快、后期趋于平缓（因为 $\log$ 增长逐渐减慢）。其直观含义是：模型浅层往往承担底层特征提取等功能，我们对其剪枝应谨慎（例如第一层仅剪掉 r_0 比例）；深层更接近输出，对细节冗余可能更多，允许更大稀疏率（最后一层剪掉 r_{n-1} 比例）；中间层则在这两个端点之间按对数曲线过渡。通过对数函数的非线性，使得中前层稀疏率提升较慢，从而保护模型基础表示的完整性，而在中后层逐步加大剪枝力度以获取更多加速收益。上述公式可根据需要灵活调整：如果某些层有特殊重要性，也可在此基础上微调个别层的 r_i 值。例如，如果通过敏感度分析发现中间某层异常脆弱，可降低该层 r_i 以补偿。

模块位置与作用

本策略并非一个单独的网络模块，而是贯穿于整个模型稀疏化过程的全局配置策略。在实现上，它体现在每一层Block稀疏滤波器所采用的阈值/保留比例各不相同。在Transformer网络结构中，我们将一个存储所有层目标稀疏率 $r_0, r_1, \ldots, r_{n-1}$ 的配置表作为稀疏化控制器的一部分。推理时，稀疏化模块会查表获取当前层的目标稀疏率，然后结合策略一、二确定具体阈值和剪枝mask。在整体作用上，策略三使模型各层都工作在各自"最优"或适宜的稀疏范围内，从而在给定全局稀疏率预算下最大化模型精度。这相当于为稀疏化引入了一层简单的深度适应性：考虑了层与层之间在功能和重要性上的差异。

实现示例（PyTorch）

在实际代码中，我们可以预先生成逐层的稀疏率列表，并在前向过程中按层应用。例如，我们希望整体从第1层的30%稀疏逐步增加到最后一层的50%稀疏，可以按公式计算每层比例：

```
import math

n_layers = len(model.blocks)
r0 = 0.3   # 第1层剪枝30%
r_end = 0.5  # 最后一层剪枝50%
layer_sparsity = []
for i in range(n_layers):
    # 按对数公式计算第i层（0-index）的稀疏率
    r_i = r0 + (r_end - r0) * math.log(i+1) / math.log(n_layers)
    layer_sparsity.append(r_i)

# 在forward中逐层应用不同的稀疏率
for l, block in enumerate(model.blocks):
    h = block.ffn.up_proj(x)          # 上投全连接
    h_act = activation_func(h)        # 激活 (如SwiGLU)
    # 结合策略二计算重要性评分（如有），否则直接取绝对值
    importance = h_act.abs()
    if use_weight_importance:
        importance = importance * weight_norms[l]  # 融合预计算的第l层权重范数
    # 确定当前层应保留比例 p = 1 - 稀疏率
    p = 1 - layer_sparsity[l]
    thresh = torch.quantile(importance, q=1-p, dim=1, keepdim=True)
    mask = (importance >= thresh).to(h_act.dtype)
    h_sparse = h_act * mask
    x = block.ffn.down_proj(h_sparse)  # 下投全连接
    # ...再进入下一层Block或后续计算
```

此实现示意每层使用各自的 `layer_sparsity[l]`（由公式计算得到）来决定剪枝掩码。通过这种方式，模型浅层(l较小)会自动采用较低的剪枝比例，深层则更多剪枝。实际应用中，我们可以先在验证数据上微调r_0和r_{n-1}使全局稀疏率达到目标并性能最佳。一旦确定，对数公式提供了一种简单可解释的插值方案，无需每层手动调节。相比统一50%剪枝，这种逐层差异化策略能够进一步降低性能损失：如前所述，在高稀疏率下，非均匀方案的PPL明显优于均匀方案。尽管EvoPress通过进化算法可找到更精细的层率配置，但本策略实现成本低且效果可观，是在工程上实用的折中方案。

与最新方法的对比分析与实验计划

为了验证上述策略的有效性，我们计划设计一系列对比实验，并结合已有最新方法的结果进行分析。

与TEAL的比较

TEAL作为训练后激活稀疏化的代表方法，实现了50%模型稀疏下性能仅轻微下降、推理提速达1.5-1.8倍。然而TEAL采用的是全模型统一阈值/比例和纯幅度剪枝。预计我们的改进方案在相同稀疏率下能够取得更低的性能损失。特别地，策略二的引入应当使剪枝对困惑度的影响更小：例如，在50%稀疏率下，我们预期LLaMA-2-7B模型的困惑度提升幅度将小于TEAL报告的约14%。同时，策略一的动态阈值将提升方法在各类输入上的鲁棒性——我们将针对不同复杂度的输入集（如WikiText文章 vs. 简单句子）测试模型性能，验证动态调整是否避免了某些输入下的异常精度劣化。对于速度提升，我们使用与TEAL相同的稀疏矩阵乘优化内核进行测试，理论上在相近的总体稀疏率下，我们方案的加速比应与TEAL相当（约1.5-1.8x），因为加速主要由稀疏度本身决定。但如果我们的策略允许更高稀疏率在可接受精度下降范围内（例如达到60%模型稀疏仍保持良好效果），则有望进一步推高加速比接近2x。

与EvoPress的比较

EvoPress通过进化搜索实现了动态非均匀压缩的最优分配。其结果表明，在高压缩比（高稀疏或低比特量化）下，人工设定的启发式往往非最优，层间不均匀配置能明显提高性能。我们的策略三本质上也是一种启发式的非均匀分配，其效果可能略逊于EvoPress的全局优化，但实现开销极低（无需额外搜索开销）。实验中，我们将对比均匀稀疏、对数分配稀疏和EvoPress找到的最佳配置，在多个稀疏率下比较模型困惑度。如果我们发现在中等稀疏率（如50-60%）下，对数策略的困惑度非常接近EvoPress，而显著好于均匀策略，那将证明该简捷策略的有效性。EvoPress报告在Mistral-7B上70%稀疏时，均匀稀疏PPL高达23.08，而动态优化可降至14.42；我们不会一开始就瞄准如此极端稀疏率，但计划逐步提高稀疏率测试我们的方案极限，在接近70%时观察性能曲线，并将结果与EvoPress曲线对比分析。这将有助于评估我们的启发式与最优搜索方案的差距和改进空间。

其他最新方法

除上述两项，我们也关注其他相关工作以作综合比较。例如，ProSparse和ReLU化（ReLUfication）方法通过改变激活函数并正则化训练，实现了超过85%-90%的超高稀疏率，但需要对模型进行再训练。我们的方案不涉及任何再训练，适用于已训练好的模型部署场景，这是与它们最大的不同。另一类方法如R-Sparse引入了秩近似分解，通过选择部分奇异值分量和输入通道实现推理加速；该方法主要改变了计算结构，而我们的方案则更直接，可视作对现有Transformer推理流程的一个插入式优化模块，实现难度更低。我们也会在实验中参考R-Sparse等的加速效果和精度，评估我们方案在同样硬件支持下能达到的性能。

实验设计

我们将基于开源的LLaMA-2和Mistral模型权重，在标准基准上进行评测：

下游性能评估

选取常用语言模型评价基准（如WikiText-103困惑度，LAMBADA完形填空准确率，以及一些MMLU知识问答基准）来测量剪枝前后模型性能变化。重点关注困惑度等生成任务指标，比较不同稀疏策略在相同稀疏率下的性能降幅。

推理速度评估

在GPU上集成定制的稀疏矩阵乘运算（例如利用NVIDIA的SparseTensorCore或Cutlass库改造Linear层），测量不同稀疏方案的实际端到端推理延迟。我们将分别测试单batch长序列生成（模拟批量=1的交互式应用）和小batch并行生成两种场景，因为稀疏加速在memory-bound的单流场景下效果尤为显著。预期我们的方法在单batch长文本生成时收益最大，可达到与TEAL相当的提速比；在多batch场景下由于算力利用差异，提速相对降低但仍可验证趋势。

消融实验

为了量化各策略单独的贡献，我们将进行消融分析：包括仅动态阈值（策略一）、仅重要性评分（策略二）、仅层率对数分配（策略三）、以及综合全部策略的对比。这样可以明确是哪一部分带来了主要的性能提升或速度改善。例如，我们预计策略二（重要性评分）单独就能够降低一定困惑度损失，而策略一保证不同数据下一致的性能，策略三在高稀疏下优势明显。综合全部策略应当取得最好的精度/速度权衡。

预期结果与分析

本研究方案有望在不微调的情况下，将LLaMA、Mistral等模型的激活稀疏率推向50%以上的水平，同时保持性能几乎无损，实现1.5倍甚至更高的推理提速。相较原始TEAL，我们的动态+重要性剪枝预计能在相同加速比下取得更低的PPL（例如降低几个点的困惑度）；相较贪心的均匀剪枝，我们的逐层优化方案在高剪枝率下能显著改善模型质量。如果实验结果验证了这些假设，那么在学术上我们将提供一种更实用的激活稀疏化方法：无需耗费计算寻找最优配置，却几乎达到类似EvoPress优化后的性能。在工程应用上，该方案仅作为推理时的一个插件模块，对模型权重无侵入修改，易于与量化等其他加速手段结合。此前研究已表明激活稀疏可与8-bit量化兼容并进一步叠加效率收益；我们的方案同样可以与量化共同使用，以获得乘法效应的加速（例如同时50%稀疏+8bit量化）。综合来看，本研究将为Transformer大模型的高效推理提供一种新颖且切实可行的技术路线，为后续在资源受限环境部署大模型奠定基础。

总结

本研究计划围绕大型Transformer模型的推理加速，提出了三项互补的稀疏化优化策略，并给出了相应的技术实现方案。通过动态阈值自适应不同数据、结合权重范数精确评估单元重要性、以及逐层差异化地分配剪枝率，我们期望在不额外训练的前提下显著提高模型激活稀疏度而几乎不损失性能。从理论分析到代码示例，我们验证了方案的可行性，并将通过详实的实验比较进一步评估其优势。与最新方法的对比分析表明，我们的方案在实用性和性能之间取得了良好平衡。该技术方案有望应用于LLaMA、Mistral等主流开源大模型的部署，加速其推理速度，为大模型在实际产品中的落地提供支持。未来，我们计划基于此方案撰写论文投稿，并探索更精细的稀疏度自动分配策略，持续提升大模型推理的效率。