大模型计算 HW5

December 2025

1 Introduction

稀疏化是当前大模型加速中的关键技术路线之一。在不改变模型架构的前提下，通过移除冗余参数，使模型满足硬件稀疏 kernel 的要求，从而在实际推理中实现加速。主要的剪枝方法按照剪枝粒度可以分为：**非结构化 (unstructured)**、**半结构化 (semi-structured)** 以及**结构化 (structured) 剪枝**。本次作业你需要在 LLaMA 模型上实现**非结构化 (unstructured)** 和**半结构化 (semi-structured)** 的剪枝方法。你将通过补充修改提供的代码实现权重稀疏化，并测试不同剪枝策略在**模型精度**与 **GPU 推理加速**（prefill / decode）中的影响。最终，你需要提交一份 PDF 报告以及代码，总结你的实现、实验过程、精度变化以及加速效果。

2 Preliminaries

稀疏化指将神经网络部分参数置零，以减少有效计算量。根据粒度不同，可分为三类。

2.0.1 非结构化剪枝（Unstructured Pruning）

对单个权重进行逐点剪枝：粒度最细，可实现较高稀疏率；不满足现有 GPU kernel 要求；理论 FLOPs 降低不等于实际加速。

2.0.2 半结构化剪枝（Semi-structured Pruning, 2:4 等）

在 Nvidia Ampere/Hopper GPU 中，2:4 稀疏模式最常用：在每 4 个连续权重中，仅保留 2 个非零值。其特点是硬件原生支持（2:4 TensorCore）；

固定 50% 稀疏率；能带来稳定的 $1.3\times - 1.6\times$ 加速。

2.0.3 结构化剪枝（Structured Pruning）

以更大粒度直接移除结构单元，例如：移除完整的 Attention heads；剪掉 MLP hidden channels；移除 FFN blocks 的部分行列。例如对 MLP：若删除中间维度的某些 channel，需要同时删除 W_1 的列与 W_2 的行。特点：(1) 对内存、吞吐、KV cache 有显著影响；(2) 可与现有 dense kernel 完全对齐；(3) 精度影响通常更大。

2.1 不同粒度的剪枝方法实际加速影响

剪枝方式	需特殊 kernel	可否 fuse 到 matmul	典型加速
非结构化	是（且困难）	否	几乎无
半结构化 2:4	**已有硬件支持**	易	30–60%
结构化	否	完全可 fuse	与降维一致

表 1: 不同剪枝方式对系统加速的影响

3 Tasks

3.1 代码部分

本次实验的代码部分旨在让你以一个完整的稀疏化框架为主线，逐步实现当前数种较为常见的剪枝方法，并在统一的模型与推理流程下比较它们在不同稀疏度下的表现。你需要从最基本的 magnitude 剪枝开始，逐步扩展到 Wanda、RIA 以及 OWL+X 等方法。所有方法都应在保持模型原有结构的前提下，对权重矩阵施加稀疏化约束，并保证剪枝后的模型能够正常完成前向推理。代码链接

3.1.1 Magnitude Pruning

在最简单的设定中，剪枝依据权重的绝对值大小进行。给定某层权重矩阵 $W \in \mathbb{R}^{m\times n}$，其稀疏化形式可写为

$$W'_{i,j} = W_{i,j} \cdot \mathbf{1}(|W_{i,j}| > \tau)\,,$$

其中阈值 τ 由目标稀疏率决定。你需要完成 lib.prune_all.py 当中 prune_magnitude 函数的代码，实现非结构化与半结构化稀疏的相关代码。

3.1.2 Wanda

Wanda 的思想基于权重元素对模型输出的相对贡献，引入激活统计来衡量某一输入通道的重要性。对权重矩阵 W 的元素 $W_{i,j}$，其 Wanda 得分为

$$s_{i,j} = |W_{i,j}| \cdot \|X_j\|_2,$$

其中 X_j 为输入激活矩阵中与列 j 对应的激活向量。稀疏化时保留得分高的元素，将其余置零。你需要完成 lib.prune_all.py 当中 prune_wanda 函数的代码，完成 Wanda 的标准实现。

3.1.3 Relative Importance Activation (RIA)

进一步地，我们观察到每一个元素的重要性需要与其在同一行或者同一列的其他元素进行比较。在这个部分中你需要实现的是 RIA（Relative Importance and Activations）打分指标。

我们首先定义相对重要性(Relative Importance, RI)指标。对权重 $W_{i,j}$，

$$\mathrm{RI}_{i,j} = \frac{|W_{i,j}|}{\sum_{i'} |W_{i',j}|} + \frac{|W_{i,j}|}{\sum_{j'} |W_{i,j'}|}.$$

第一项在每个输入通道（列）内归一化，第二项在每个输出通道（行）内归一化，避免整列或整行被完全剪空。

在此基础上，引入激活信息，形成 RIA（Relative Importance and Activations)。设输出通道 i 的激活为 X_i，则

$$\mathrm{RIA}_{i,j} = \mathrm{RI}_{i,j} \cdot \|X_i\|_2^a = \left(\frac{|W_{i,j}|}{\sum_{i'} |W_{i',j}|} + \frac{|W_{i,j}|}{\sum_{j'} |W_{i,j'}|} \right) \cdot \|X_i\|_2^a,$$

其中 a 是超参数（原文推荐 $a = 0.5$)。在实施时，你需要计算所有 $\mathrm{RIA}_{i,j}$ 并在给定稀疏率下选择得分最高的元素，同时支持非结构化与 N:M 半结构化模式。你需要完成 lib.prune_all.py 当中 prune_ria 函数的代码，实现非结构化与半结构化稀疏的相关代码, 并搜索最佳的超参 a。

3.1.4 OWL+Wanda：Outlier Weighed Layerwise Sparsity

在前面的剪枝方法中，通常默认对所有层采用统一的稀疏率，而 OWL（Outlier Weighed Layerwise Sparsity）试图利用模型中 outlier 特征的分布，为每一层分配非均匀的 layerwise 稀疏度。与 Wanda 一样，我们对 $W_{i,j}$ 定义 outlier 得分

$$A_{i,j} = \|X_j\|_2 \cdot |W_{i,j}|,$$

其中 X_j 是输入激活中与列 j 对应的向量。对第 l 层，设该层权重元素数为 N_l，其得分矩阵为 $A^{(l)}$，均值为 $\bar{A}^{(l)}$。设定阈值 $M > 1$，则第 l 层的 outlier 比例（LOD）定义为

$$D_l = \frac{1}{N_l} \sum_{i,j} \mathbf{1}\Big(A_{i,j}^{(l)} > M \cdot \bar{A}^{(l)}\Big).$$

将所有层的 outlier 比例组成向量

$$\mathrm{LOD} = [D_1, D_2, \ldots, D_L].$$

OWL 基本原则是：outlier 比例越高的层应当保留更多参数。因此给每一层分配的稀疏率满足

$$S_l \propto (1 - D_l),$$

并做归一化以满足全局稀疏率约束

$$\sum_{l=1}^{L} S_l N_l = S_{\text{global}} \sum_{l=1}^{L} N_l.$$

如果不同层之间的 sparsity 差异过大，会造成类似 “全局剪枝” 的破坏性效果，使稀疏 LLM 出现性能崩塌。为避免这种情况，OWL 引入一个至关重要的超参数 λ 来控制稀疏率的最大偏移范围。

OWL 因此将每层 sparsity 限制在

$$S_l \in [\, S_{\text{global}} - \lambda,\ S_{\text{global}} + \lambda \,],$$

同时仍需满足平均稀疏率为 S_{global}。你需要完成 lib.prune_all.py 当中 prune_wanda_outlier 函数的代码，实现 outliner 数量的计算并给出将其转化为不同层的 sparsity ratio 的代码。

3.2 性能分析

3.2.1 模型性能与稀疏度的关系

在完成上述稀疏化方法后，你需要系统化地评估不同稀疏度对模型性能的影响。建议在相同的验证环境下，对 Magnitude，Wanda 和 RIA 在多个稀疏率（例如 30%、50%、70%、90%）下的 Zero-shot 准确率或困惑度进行测试，并绘制折线图展示稀疏度与性能之间的关系。

3.2.2 测试半结构化稀疏模型的实际加速

进一步地，利用你在上一部分得到的 2:4 稀疏模型使用 TensorRT-LLM 并测试实际推理加速。在构建支持 2:4 稀疏的 TensorRT 引擎时，可参考官方文档：

```
https://nvidia.github.io/TensorRT-LLM/latest/commands/
trtllm-build.html
```

首先需要将 HF 格式的模型转为 Tensorrt 的格式：

```
python examples/llama/convert_checkpoint.py \
    --model_dir /path/to/hf_model_or_repo \
    --output_dir /root/autodl-tmp/trtllm_checkpoints/llama_7b \
    --dtype float16
```

接着构造 TensorRT-LLM engine。

```
trtllm-build \
  --checkpoint_dir /root/autodl-tmp/trtllm_checkpoints/llama_7b \
  --output_dir /root/autodl-tmp/trt_engines/llama_7b_fp16 \
  --dtype float16 \
```

最后可以手动利用 tensorrt_llm.LLM 测试 decode 与 Prefill 加速。或者可以使用 benchmark 脚本：

```
python TensorRT-LLM/benchmarks/python/benchmark.py ...
```

本任务需要你比较稀疏模型相对于 dense 模型的实际加速倍率。你可以控制输入输出长度以及 batchsize。实验中应特别注意推理 prefill 以及 decode 两个阶段呈现出的不同加速趋势，并在报告中做出合理解释。

3.3 拓展（二选一，多做可以加分）

3.3.1 不同结构对于剪枝的敏感性分析

本小节旨在探讨 Transformer 结构中的不同子模块在剪枝过程中的敏感性差异。你需要分析并比较 Attention 与 MLP 在稀疏化后性能下降的程度，判断究竟哪一部分更适合进行高比例剪枝，并给出理论或经验上的解释。分析视角可包括但不限于以下几个方向：算子在推理中的 FLOPs 占比、参数规模差异带来的结构性冗余、各自对稀疏性模式（如非结构化与 $N:M$ 半结构化）的适配性，以及它们在不同任务中的重要性差异等。希望你结合实验结果给出一个较为完整的 sensitivity 讨论。

3.3.2 剪枝对于 Calibration Data 的敏感性分析

在前面的实验中，我们计算激活统计量（如 $\|X_j\|_2$）时默认使用的是通用的 C4 数据集。本拓展任务希望你进一步分析剪枝过程对校准数据（Calibration Data）的依赖程度。具体而言，你可以尝试将 C4 替换为更具领域特征的校准数据集，例如 MaTH，并在相同稀疏率下评估模型的困惑度（ppl）以及其在数学任务评测集上的表现。通过比较不同类型的 Calibration Data 所带来的剪枝效果差异，思考模型的结构化知识是否影响稀疏化的稳定性，以及域内校准是否能够提升剪枝模型在特定任务上的适应能力。

4 DDL

本次作业 1.9 截止. 请提交一份 PDF 报告以及代码文件。