ROSE: Reordered SparseGPT for More Accurate One-Shot Large Language Models Pruning

Mingluo Su, Huan Wang*
Westlake University
{sumingluo, wanghuan}@westlake.edu.cn
https://github.com/mingluo-su/ROSE

Pruning is widely recognized as an effective method for reducing the parameters of large language models (LLMs), enabling more efficient deployment and inference. One classic and prominent path of LLM one-shot pruning is to leverage second-order gradients (*i.e.*, Hessian), represented by the pioneering work SparseGPT [1]. However, the predefined left-to-right pruning order in SparseGPT leads to suboptimal performance when the weights exhibit *columnar* patterns. This paper studies the effect of pruning order under the SparseGPT framework. The analyses lead us to propose ROSE, a reordered SparseGPT method that prioritizes weights with larger potential pruning errors to be pruned earlier. ROSE first performs pre-pruning to identify candidate weights for removal, and estimates both column and block pruning loss. Subsequently, two-level reordering is performed: columns within each block are reordered in descending order of column loss, while blocks are reordered based on block loss. We introduce the relative range of block loss as a metric to identify *columnar* layers, enabling adaptive reordering across the entire model. Substantial empirical results on prevalent LLMs (LLaMA2-7B/13B/70B, LLaMA3-8B, Mistral-7B) demonstrate that ROSE surpasses the original SparseGPT and other counterpart pruning methods.

1. Introduction

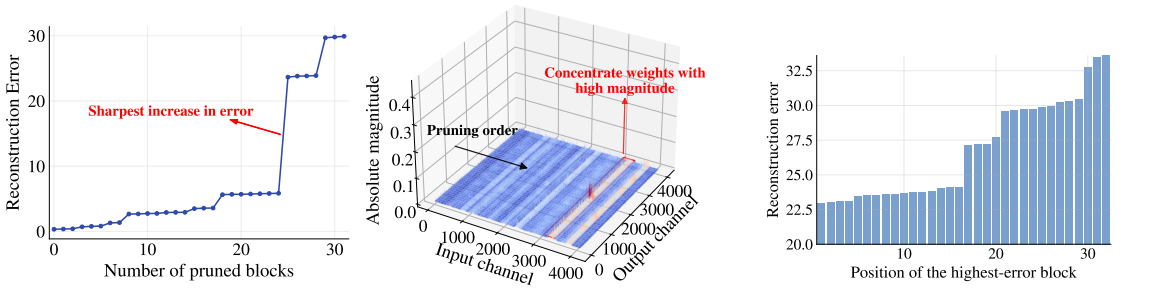
Large language models (LLMs) [2–5] have demonstrated remarkable capabilities in natural language understanding and generation attributed to the massive scale of their architectures and training data [6–9]. However, with hundreds of billions of parameters, these models require substantial memory and computational resources, posing significant challenges for deployment on resource-constrained devices [10–12].

Model pruning [13–18] is an effective way to enhance model inference and deployment efficiency by removing less critical weights while maintaining competitive performance. Traditional pruning methods typically determine which weights to prune in a single pass based on designed criteria [19–21], or iteratively select the weights with the smallest pruning error for removal, followed by retraining the remaining weights to recover performance [22–25].

Nevertheless, retraining-based approaches become prohibitively expensive and time-consuming for LLMs given the significant computational cost required for full-model optimization. On the contrary, pruning approaches for LLMs focus on post-training pruning (PTP) methods [1, 26–28]. Following the classic paradigms of traditional pruning, some current approaches directly prune weights in a single pass but without further adjusting the remaining parameters. Those methods focus on adopting more comprehensive pruning masks [27, 28]. Another paradigm adjusts the remaining weights by using closed-form second-order solutions [14, 29] instead of retraining, represented by the pioneering work SparseGPT [1]. SparseGPT implements a layer-wise approximate compensation strategy and enables few-hour unstructured pruning of hundred-billion-parameter models, safely pruning up to 60% of parameters without fine-tuning, highlighting its value for LLMs pruning.

*Corresponding author

Third Conference on Parsimony and Learning (CPAL 2026).



(a) Change in reconstruction error (b) Weight visualization (c) Reconstruction error after reordering

Figure 1: (a) Change reconstruction error of the "self_attn.o_proj" layer in the first Transformer Block of LLaMA2-7B during SparseGPT pruning as the number of pruned blocks increases. The sharpest increase in reconstruction error appears at a later stage. (b) Weight visualization of the corresponding layer. It exhibits a *columnar* pattern along the input channel, and there is a block with the most concentrated high-magnitude weights as illustrated. (c) Different reconstruction error after reordering the original block with the highest pruning error. The earlier the original block is pruned, the smaller the reconstruction error.

Revisiting the pruning process in SparseGPT, we find it adopts a fixed block[2] sparsity rate and block size during iterative blocking pruning, and differences in weight distribution across blocks lead to obvious variations in pruning errors. Figure 1(a) shows the change of layer-wise reconstruction error as the number of pruned blocks increases for the "self_attn.o_proj" layer in the first Transformer Block of LLaMA2-7B, where a sharp increase occurs at the late pruning stage, while error increases in other blocks remain relatively moderate. Figure 1(b) visualizes the weight magnitude of this layer. The weight distribution reveals a *columnar* pattern where weights with similar magnitude are concentrated within blocks. The block with the sharpest increase in pruning error in Figure 1(a) corresponds precisely to the block with the most concentrated high-magnitude weights in Figure 1(b).

Since SparseGPT performs approximate compensation using a common subset of remaining weights, earlier pruned weights have access to more available weights for error correction. The final reconstruction results are influenced by the pruning order. To mitigate this effect, we reorder the block with the highest pruning error from the earliest position to the last, while preserving the relative positions of other blocks. The resulting reconstruction errors under different modified pruning orders are shown in Figure 1(c). Interestingly, the earlier this block is pruned, the smaller the final reconstruction error and vice versa. This observation leads to the following question: *Can we achieve a better weight reconstruction in SparseGPT by proposing an optimized pruning order?*

In this paper, we introduce `ROSE`, a one-shot pruning order adjustment method based on SparseGPT. A pre-pruning step is performed to identify weights that are highly likely to be pruned, based on which both the column-wise losses and block-wise pruning losses are calculated. Columns within each block are reordered in descending order of their losses, while blocks are reordered according to their block losses. We identify layers exhibiting the *columnar* pattern based on the fluctuation range of block-wise loss and perform reordering for those layers. The experimental results demonstrate that `ROSE` can surpass the original SparseGPT and other existing unstructured pruning methods in prevalent LLMs. Our contributions are as follows:

- We find that a key factor in accurate one-shot pruning based on the SparseGPT framework is the pruning order, and propose `ROSE` to study the problem for the first time.
- We propose a more optimal pruning order for layers that exhibit a *columnar* pattern and an evaluation metric for detecting layers exhibiting such patterns.
- Extensive evaluations on prevalent models suggest our method performs favorably against prior SoTA counterparts.

[2]Throughout this paper, the lowercase term "block" refers to a sub-matrix along the input channel of a single layer.

2. Related Work

2.1. Network Pruning

Network pruning aims to reduce redundant parameters while maintaining model accuracy [13, 14]. In terms of workflow, one pruning paradigm is to determine all weights to be pruned based on a certain importance criterion at the initial stage and then retrain the remaining weights to recover performance [19–21]. The other is iterative pruning, which involves repeated cycles of pruning based on certain criteria, subsequent fine-tuning of the remaining weights, and re-assessment of the pruning criteria. Methods in this category generally adopt a greedy order, pruning weights in ascending order of pruning error [22, 23, 30]. As pruning progresses, the amount of remaining weight available for compensation gradually decreases. Consequently, in the later stage, when more significant weight is removed, the available parameters in the network become increasingly limited. To date, no work has explored the impact of pruning order on final model performance.

2.2. Unstructured Pruning for LLMs

Unstructured pruning aims to remove unimportant individual weights from the network [13, 15], which differs from structured pruning that aims to remove entire structures such as channels, attention heads, filters, and layers [21, 31–33]. Unstructured pruning preserves the original model structure and can be performed in a training-free manner [14, 29], an advantage that is particularly critical under conditions of constrained fine-tuning resources in the era of LLMs. Recent years have witnessed growing attention toward unstructured methods for LLMs. For instance, SparseGPT [1] pioneers one-shot unstructured pruning for LLMs via layer-wise Hessian-based reconstruction for compensation. Following a different yet simpler paradigm of pruning without weight updates, Wanda [27] combines weight magnitude and activation as pruning criteria, while DSnoT [28] improves upon it by dynamically adjusting the pruning mask. In contrast, OATS [34] approximates each weight matrix as the sum of a sparse matrix and a low-rank matrix, thereby explicitly maintaining the critical outlier features [35] of LLMs.

3. Prerequisites

Layer-Wise Pruning. Layer-wise pruning [29, 36]aims to remove less significant weights from each layer sequentially while maintaining overall model performance. The pruning process for a given layer l is formulated as a minimization problem of the ℓ_2-error between the original and pruned outputs, defined as follows:

$$\text{argmin}_{\hat{\mathbf{W}}_l} ||\mathbf{W}_l\mathbf{X}_l - \hat{\mathbf{W}}_l\mathbf{X}_l||_2^2, \tag{1}$$

where $\mathbf{W}_l$ represents the weight matrix before pruning, $\hat{\mathbf{W}}_l$ is the pruned weight matrix, and $\mathbf{X}_l$ denotes the input to the layer l.

Optimal Brain Surgeon (OBS) Framework. Optimal Brain Surgeon (OBS) is based on a Taylor expansion of the loss function. It removes the weight with minimal impact on the objective function and updates the remaining weights to minimize the change in loss. Given a weight matrix $\mathbf{W} \in \mathbb{R}^{M\times N}$ and the corresponding input data matrix $\mathbf{X} \in \mathbb{R}^{D\times N}$, let $\mathbf{H} = \mathbf{X}\mathbf{X}^\top$ denote the corresponding Hessian matrix, the increase in loss $\Delta\mathcal{L}$ caused by the removal of the weight w_q and the optimal updating of the remaining weights $\Delta\mathbf{w}$ are given by:

$$\Delta\mathcal{L} = \frac{w_q^2}{[\mathbf{H}^{-1}]_{qq}}, \quad \Delta\mathbf{w} = -\frac{w_q}{[\mathbf{H}^{-1}]_{qq}}\mathbf{H}^{-1}_{:,q}. \tag{2}$$

In the process of pruning, OBS employs an iterative update approach that involves multiple Hessian inverse operations, leading to high computational complexity for large-scale models. To address these problems, OBC [29] decomposes the objective of the layer-wise reconstruction into subproblems by row and proposes an efficient computational framework for matrix inversion through optimized Gaussian elimination. However, its direct application to LLMs remains computationally expensive.

Revisiting SparseGPT. SparseGPT [1] is a one-shot pruning method that adapts the OBS framework specifically for LLMs. It decouples pruning into mask selection and approximate sparse weight reconstruction. For mask selection, SparseGPT selects the pruning mask for B_S columns at a time and adaptively chooses the mask during the pruning process. For weight reconstruction, it employs a fixed left-to-right pruning order and leverages the common set of remaining pruned weights for multi-row parallel compensation to achieve pruning acceleration. The inverse Hessian information used for weight compensation during the whole pruning process can be stored in the lower triangular matrix $\mathbf{L}$ obtained from the Cholesky decomposition [37] of $\mathbf{H}^{-1}$ in advance:

$$[\mathbf{H}_{i:,i:}]^{-1} = \mathbf{L}_{i,i}\mathbf{L}^{\top}_{i:,i:}, \tag{3}$$

where $\mathbf{H}^{-1} = \mathbf{L}\mathbf{L}^{\top}$.

4. Methodology

In this section, we first analyze the dilemma of adjusting the pruning order within the SparseGPT framework and give our main solutions. Then, we propose ROSE: (1) By performing a pre-pruning step, candidate weights with a high probability of being pruned are selected out to estimate both column loss and block. (2) By performing two-level reordering, weights with greater potential errors are pruned earlier. (3)By calculating the relative range of block loss to identify *columnar* layers, an automatic reordering strategy is implemented for whole models.

4.1. Analyses

Our objective is to prioritize weights with larger pruning errors to be pruned earlier. To achieve this, we need to determine which weights will be removed first. In SparseGPT, once a block is pruned, the mask for the subsequent block is determined based on the updated weights, making it difficult to precisely predict which weights will be pruned.

Fortunately, we observe that most weights (>80%) deviate only marginally (<30%) from their original values, as shown in Figure 2. This suggests that the relative importance of most weights remains largely stable throughout the pruning process. Therefore, we can first estimate which weights are highly likely to be pruned based on the magnitudes of the initial weights, compute the corresponding pruning loss, and then reorder the weights accordingly. Additionally, since SparseGPT adopts a block-wise masking strategy, we reorder each block as a whole to ensure that the mask remains essentially unchanged.

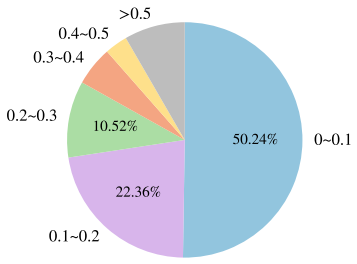


Figure 2: The distribution of relative change of weights before and after pruning. The majority of weights remain relatively stable.

4.2. Proposed ROSE

Pre-pruning. We estimate the potential pruning loss by performing a pre-pruning step based on the importance of the initial weights. For the importance score for each weight, we adopt the metric proposed in Wanda [27], which combines the weight magnitude and the corresponding input activation. Specifically, given the weight matrix $\mathbf{W} \in \mathbb{R}^{M\times N}$, where M donates the number of output channels and N donates the number of input channels, let $\mathbf{X} \in \mathbb{R}^{(B\times L)\times N}$ represent the input activation matrix, where B represents the batch size and L denotes the sequence length. For a single weight $\mathbf{W}_{ij}$, the importance score $\mathbf{S}_{ij}$ is defined as:

$$\mathbf{S}_{ij} = |\mathbf{W}_{ij}| \cdot \|\mathbf{X}_j\|_2, \tag{4}$$

where $\cdot$ represents the element-wise product and $\|\mathbf{X}_j\|_2$ denotes the ℓ_2 norm of the corresponding input activation.

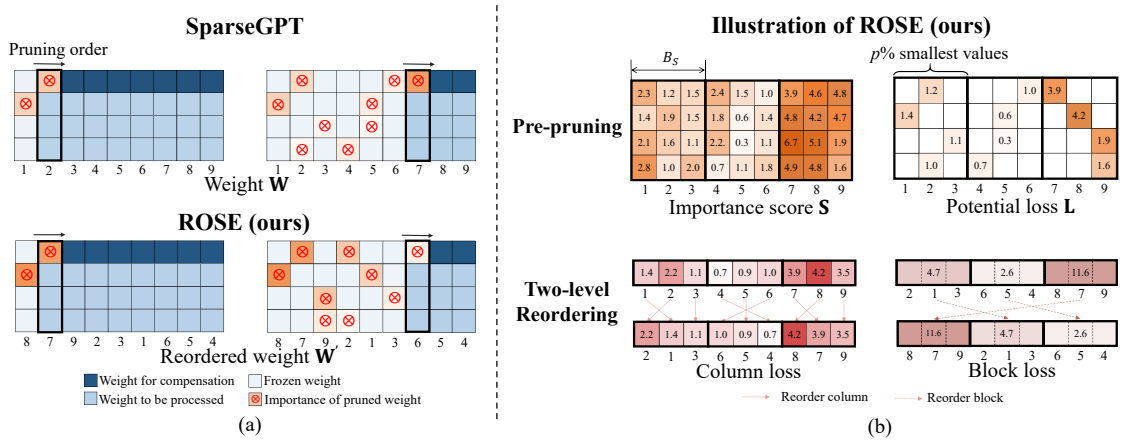


Figure 3: (a) Overview of difference between SparseGPT and ROSE. Orange color represents weight importance, and the darker the color, the greater the importance. In SparseGPT, the number of weights available for error compensation (shown in dark blue) decreases during pruning, limiting recovery if high-error weights are pruned late. ROSE reorders those with potentially large pruning errors to the front to be pruned earlier. In this way, more parameters remain available for larger error compensation. (b) Illustration of our ROSE for the *columnar* layer. Given the dense weight $\mathbf{W}$ and target sparsity rate $p\%$, we calculate the importance score $\mathbf{S}$ and split it into blocks based on B_S. The smallest $p\%$ of values from each block are selected as the loss matrix $\mathbf{L}$. Column loss and block loss are calculated based on the loss matrix. Columns within one block are reordered in descending order of column loss, and blocks are reordered in descending order of block loss.

We construct the potential loss matrix $\mathbf{L}$ by performing the pre-pruning process. Following the fixed iterative block pruning of SparseGPT, $\mathbf{W}$ is divided into K blocks where $K = \lceil N/B_s \rceil$ along the column dimension. The corresponding weight blocks and input activation blocks can be denoted as $\mathbf{W}^{(k)} = \mathbf{W}[:, i_1 : i_2]$ and $\mathbf{X}^{(k)} = \mathbf{X}[:, i_1 : i_2]$, respectively, where $i_1 = (k-1) \cdot B_s$ and $i_2 = \min(k \cdot B_s, N)$. Then we can get block score $\mathbf{S}^{(k)}$ by Equation 4. Let $p\%$ be the target sparsity rate. For each block $k = 1, 2, \ldots, K$, the smallest $p\%$ of elements in $\mathbf{S}^{(k)}$ are extracted to form candidate pruning loss of blocks $\mathbf{L}^{(k)}$.

Two-level Reordering. In order to prioritize pruning weights with larger errors, our reordering process includes two levels: column reordering and block reordering. Column reordering is performed within each block separately. Specifically, the pruning loss of each column in block k is calculated as $l_j^{(k)} = \sum_{i=1}^{M} \left[\mathbf{L}^{(k)}\right]_{ij}$. The columns in each $\mathbf{W}^{(k)}$ are sorted in descending order of column loss $l_j^{(k)}$:

$$\mathbf{W}^{(k)} \leftarrow \left[\mathbf{w}_{j_1}^{(k)}, \mathbf{w}_{j_2}^{(k)}, \ldots, \mathbf{w}_{j_B}^{(k)}\right] \quad \text{where} \quad l_{j_1}^{(k)} \geq l_{j_2}^{(k)} \geq \cdots \geq l_{j_B}^{(k)}. \tag{5}$$

For block reordering, the entire block is treated as a unit and reordered in descending order.The total block loss is calculated as $L^{(k)} = \sum_{i=1}^{M} \sum_{j=i_1}^{i_2} \left[\mathbf{C}^{(k)}\right]_{ij}$. All blocks are reordered in descending order based on $L^{(k)}$:

$$\mathbf{W} \leftarrow \left[\mathbf{W}^{(k_1)}, \mathbf{W}^{(k_2)}, \ldots, \mathbf{W}^{(k_K)}\right] \quad \text{where} \quad L^{(k_1)} \geq L^{(k_2)} \geq \cdots \geq L^{(k_K)}. \tag{6}$$

Through these two-levels reordering operations, weights with larger pruning errors are prioritized to be pruned earlier.

Columnar Layer Identification. Reordering is performed on those layers that presented the *columnar* pattern. In order to identify this structure, we use the differences in block losses for identification. To quantify this difference, we define the relative range of block loss as:

$$R_{\text{rel}} = \frac{\max_k L^{(k)} - \min_k L^{(k)}}{\text{mean } L^{(k)}}. \tag{7}$$

Consequently, if the relative range of block loss of a layer exceeds a predefined threshold, we classify it as *columnar* layer and apply reordering to it.

5. Experimental Results

5.1. Experiment Settings

Models and Datasets. We select the current public LLMs for evaluation, including the LLaMA2 series [38], LLaMA3 series [39], and Misrtal-7B [40]. These models range in size from 7 billion to 70 billion parameters. We primarily assess the performance of pruned large language models through perplexity, a widely adopted and stable metric for measuring LLM performance, and use WikiText-2-raw [41] datasets. To further evaluate the capabilities of pruned models, we also conduct experiments on seven standard common-sense benchmark tasks: BoolQ [42], WinoGrande [43], PIQA [44], OpenBookQA [45], HellaSwag [46], ARC-Easy and ARC-Challenge [47]. All zero-shot tasks are performed uniformly based on the lm-eval-harness framework [48].

Comparison Methods. We compare our method with counterpart methods, including: (1) Magnitude pruning [15] that removes weights based on the magnitude metric. (2) SparseGPT [1] that utilizes approximate second-order Hessian information to evaluate weight importance and perform weight reconstruction. (3) Wanda [27] that removes weights based on magnitudes multiplied by the corresponding input activation norms. (4) DSnoT [28] that performs training-free fine-tuning with dynamic masks after pruning. (5) OATS [34] that decomposes weight matrices into a sparse matrix and a low-rank matrix, with a designed strategy to preserve critical outlier features.

Implementation Details. ROSE is implemented with the PyTorch framework [49] and HuggingFace Transformers [50]. Consistent with previous works [1], the calibration data contains 128 samples randomly selected from the first shard of the C4 dataset [51]. Each sample contains sequences of 2048 tokens. All experiments are conducted on NVIDIA 48GB 4090 GPUs. The results of comparison methods are reproduced by their official code. DSnoT is combined with both Magnitude, Wanda, and SparseGPT in its paper. In our experiments, we run them both and take the best one to put in the results. For SparseGPT and ROSE, the blocksize is set to 128. For ROSE, the threshold for the *columnar* layer is set to 0.5, and the specific reasons are explained in Appendix E.

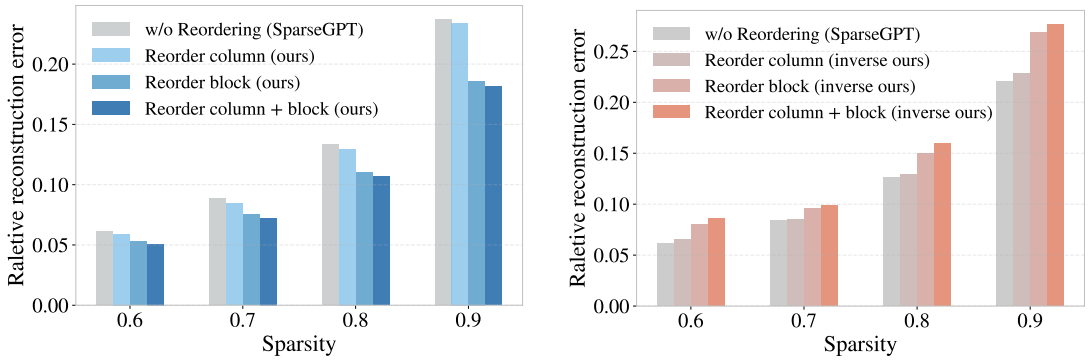
5.2. Reconstruction Error Analyses

In this section, we evaluate the reconstruction error of the single layer using different methods. First, we analyze the reconstruction error by all methods. The results are shown in Table 1. It can be observed that Wanda, DSnoT, and Magnitude, which prune weights directly, exhibit a significant increase in reconstruction error as sparsity increases. OATS, despite decomposing the weights into the sum of a sparse matrix and a low-rank matrix, also shows large reconstruction errors. In contrast, SparseGPT and ROSE consistently achieve lower reconstruction errors across different sparsity levels. This is because both methods are based on the OBS second-order effective compensation. Moreover, our method achieves a smaller reconstruction error than SparseGPT at all sparsity rates.

Figure 4 provides a detailed analysis demonstrating that ROSE achieves a lower reconstruction error than SparseGPT. As shown in Figure 4(a), both column reordering and block reordering individually contribute to reducing the reconstruction error, with block reordering yielding more pronounced improvements. Meanwhile, the reduction in reconstruction error becomes more pronounced as the sparsity level increases. Interestingly, Figure 4(b) shows the opposite trend when the order of our method is reversed, pruning the blocks and columns first with smaller errors. This contrast strongly indicates that the pruning order accounts for pruning error.

5.3. Main Benchmark Results

Unstructured Pruning Results. Table 2 shows the WikiText perplexity performance on LLaMA3-8B and Mistral-7B at different sparsity rates. Under high sparsity conditions, SparseGPT and ROSE clearly outperform other approaches since they adjust the unpruned weights for error compensation. Moreover, ROSE achieves lower perplexity compared with SparseGPT at most sparsity rates. For example, ROSE reduces perplexity from 203.45 to 172.14 at 80% sparsity rate on LLaMA3-8B.



(a) Descending order (b) Ascending order

Figure 4: Relative reconstruction error of the "self_attn.o_proj" layer in the second Transformer Block of LLaMA2-7B by ROSE and its variants at varying sparsity rates.

Table 1: Relative reconstruction error of the "self_attn.o_proj" layer in the second Transformer Block of LLaMA2-7B by different methods.

Sparsity	Magnitude	SparseGPT	Wanda	DSnoT	OATS	ROSE (ours)
60%	1.50e-1	6.12e-2	9.39e-2	9.39e-1	1.50e-1	**5.09e-2**
70%	2.20e-1	8.84e-2	1.40e-1	1.40e-1	2.00e-1	**7.22e-2**
80%	3.20e-1	1.33e-1	2.20e-1	2.20e-1	3.00e-1	**1.07e-1**
90%	5.00e-1	2.37e-1	3.60e-1	3.60e-1	4.80e-1	**1.81e-1**

Table 2: WikiText perplexity (↓) performance on LLaMA3-8B and Mistral-7B model at varying sparsity rates.

Method	**LLaMA-3 8B** (Dense: 6.14)				**Mistral-7B** (Dense: 5.32)			
	60%	**70%**	**80%**	**90%**	**60%**	**70%**	**80%**	**90%**
Magnitude	3.38e5	1.62e6	8.54e6	2.35e6	31.42	8.88e3	1.34e4	1.22e5
SparseGPT	**15.23**	40.48	203.45	1.10e3	9.37	21.48	**78.69**	286.54
Wanda	23.34	123.78	986.97	1.02e4	11.11	57.31	236.17	5.16e3
DSnoT	19.66	126.99	995.57	8.38e4	9.67	30.51	1.91e3	7.76e3
OATS	16.34	88.93	770.54	7.95e3	10.54	35.20	261.60	6.44e3
ROSE (ours)	15.50	**40.29**	**172.14**	**840.10**	**9.30**	**20.86**	78.96	**266.88**

Table 3 presents the WikiText perplexity performance and zero-shot task performance of unstructured pruning methods on LLaMA2 models at 70% sparsity rate. SparseGPT and ROSE outperform other pruning methods in most cases. Moreover, ROSE achieves lower perplexity results than the SparseGPT across all evaluated models. In terms of zero-shot evaluations, ROSE achieves better average accuracy across all evaluated models than SparseGPT. For task-specific evaluations, ROSE can achieve higher accuracy in the majority of tasks across the models of different sizes. Notably, at the 7B, our approach surpasses SparseGPT by over 1.5% in ARC-c and ARC-e tasks.

Semi-structured Pruning Results. ROSE can be extended to semi-structured pruning by changing the pre-pruning step according to the semi-structured sparsity pattern and adjusting the blocksize parameter accordingly. Specifically, the blocksize is set to 4 for the 2:4 pattern and to 8 for the 4:8 pattern. Table 4 and 5 show the perplexity performance of LLaMA models in two semi-structured patterns. The results show that our approach outperforms SparseGPT in both the 2:4 and 4:8 patterns. For example, under the 2:4 pattern on the LLaMA3-8B, our method reduces perplexity on WikiText by 0.5 compared to SparseGPT, demonstrating the effectiveness and superiority of ROSE in semi-structured pruning.

Table 3: WikiText perplexity (↓) and zero-shot task accuracy (↑) performance on LLaMA2 models at 70% sparsity rate for different unstructured pruning methods.

Model	Method	Perplexity	Zero-shot Accuracy							
			BoolQ	WinoG.	PIQA	OBQA	HellaS.	ARC-e	ARC-c	Avg.
LLaMA2-7B	Dense	5.47	77.68	69.14	79.05	44.20	76.01	74.54	46.33	66.71
	Magnitude	4.98e4	37.95	49.25	51.52	28.00	26.32	27.90	**26.96**	35.41
	SparseGPT	27.68	63.61	58.41	62.35	29.60	40.38	40.19	23.46	45.43
	Wanda	72.58	48.50	49.33	53.86	25.80	30.21	30.64	21.33	37.10
	DSnoT	60.44	62.14	55.25	**63.00**	30.40	39.24	**44.15**	25.94	45.73
	OATS	50.44	60.46	51.38	55.11	28.20	32.32	32.45	21.59	40.22
	ROSE (ours)	**26.38**	**64.04**	**59.19**	62.84	**30.60**	**41.35**	41.71	25.26	**46.43**
LLaMA2-13B	Dense	4.88	80.55	71.98	80.52	45.20	79.36	77.53	48.98	69.16
	Magnitude	2.14e2	38.65	49.49	53.10	26.60	29.51	32.11	24.49	36.28
	SparseGPT	19.78	**68.17**	61.88	67.85	32.20	46.70	48.11	**28.67**	50.51
	Wanda	46.22	62.08	50.75	57.34	28.20	31.62	35.56	21.25	40.97
	DSnoT	31.21	64.86	56.20	66.92	**33.60**	47.17	49.45	27.22	49.35
	OATS	40.80	62.42	56.04	60.39	29.20	35.27	38.01	22.61	43.42
	ROSE (ours)	**19.54**	65.90	**63.22**	**68.01**	33.00	**47.61**	**49.54**	27.99	**50.75**
LLaMA2-70B	Dense	3.32	83.76	77.98	82.70	48.80	83.81	81.06	57.25	73.62
	Magnitude	423.46	39.57	57.14	67.63	35.80	57.20	54.55	34.56	49.49
	SparseGPT	9.34	**80.58**	**75.30**	77.04	41.60	69.19	70.03	**43.86**	65.37
	Wanda	10.59	74.10	74.03	75.63	40.00	64.73	69.99	40.78	62.75
	DSnoT	**8.29**	79.02	73.95	**77.31**	**42.80**	**71.96**	69.53	42.75	65.33
	OATS	9.97	75.60	73.23	75.83	40.70	68.13	69.49	41.38	63.48
	ROSE (ours)	9.29	80.18	75.14	76.44	42.60	69.40	**70.79**	43.77	**65.47**

Table 4: WikiText perplexity (↓) on LLaMA models with 2:4 pattern.

Method	2-7B	2-13B	3-8B
SparseGPT	11.00	8.77	16.33
ROSE (ours)	**10.73**	**8.60**	**15.84**

Table 5: WikiText perplexity (↓) on LLaMA models with 4:8 pattern.

Method	2-7B	2-13B	3-8B
SparseGPT	8.46	7.00	12.20
ROSE (ours)	**8.30**	**6.96**	**12.00**

5.4. Ablation Study

Blocksize. ROSE involves reordering both blocks and the columns within one block, a process governed by the blocksize. This section investigates its performance with SparseGPT under identical blocksize conditions on LLaMA2-7B at 70% sparsity rate. As illustrated in the Figure 5(a), ROSE exhibits robustness similar to SparseGPT, with WikiText perplexity remaining stable over a wide range of blocksize values and ROSE consistently achieves a lower perplexity than SparseGPT.

Calibration Data. Since both SparseGPT and ROSE rely on the Hessian matrix computed from calibration data for weight compensation, we analyze the impact of calibration data number and sequence length on the LLaMA2-7B model at 70% sparsity rate. The results are shown in Figure 5(b) and 5(c). For the number of calibration data, ROSE consistently outperforms SparseGPT by achieving lower perplexity. Similarly, longer input sequences also lead to reduced perplexity, and ROSE maintains its performance advantage across all sequence lengths.

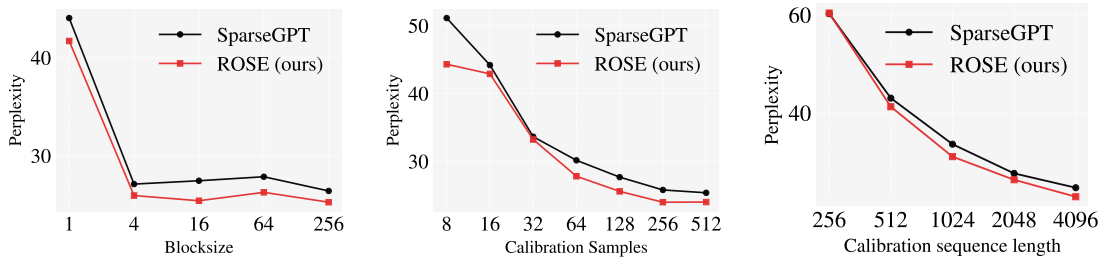


Figure 5: Ablation study of blocksize, calibration samples, and calibration sequence length in LLaMA2-7B at 70% sparsity rate.

Table 6: Pruning time (minutes) for pruning a whole model with a calibration set of size 128 at 70% sparsity rate.

Model	Magnitude	SparseGPT	Wanda	DSnoT	OATS	ROSE (ours)
LLaMA2-7B	-	4.76	1.75	1.95	572	5.15
LLaMA2-13B	-	8.45	1.85	2.35	925	9.34

5.5. Running Consumption Analyses

Pruning Time Consumption. Table 6 compares the pruning computations of ROSE and other counterpart pruning methods on LLaMA2-7B and 13B models. Wanda and DSnoT do not involve any weight updates, leading to extremely fast pruning speeds. However, they suffer from significant performance degradation at high sparsity rates, as demonstrated in Section 5.3. OATS exhibits the longest pruning time. For instance, it requires 572 minutes to prune the LLaMA2-7B model, which is hundreds of times that of ROSE. ROSE introduces two additional lightweight steps compared to SparseGPT: computing the pruning loss and performing reordering operation. The overall pruning time increases marginally relative to SparseGPT (from 4.76 minutes to 5.15 minutes on the LLaMA2-7B and from 8.45 minutes to 9.34 minutes on the LLaMA2-13B, respectively).

Inference Acceleration. 2:4 sparsity is a common semi-structured pattern, and NVIDIA's CUTLASS library provides optimized kernels for it. Results of end-to-end latency on LLaMA2-70B can be found in Table 7. The inference results of SparseGPT and ROSE are nearly identical. ROSE treats every group of four weights as a unit and reorders them when handling 2:4 sparsity, without altering the standard 2:4 sparsity pattern. The reorder operation is conducted during pruning (shown in Appendix C), and no extra reorder operations are needed during inference after pruning. Consequently, both versions achieve similar acceleration.

Table 7: End-to-end latency obtained by different methods on LLaMA2-70B.

Method	Latency (ms)	Speedup
Dense	1791	-
SparseGPT	1458	1.23×
ROSE (ours)	1450	1.24×

6. Conclusion

This paper introduces ROSE, a new one-shot layerwise pruning method based on the second-order pruning framework. We are motivated by an interesting observation that certain layers in existing LLMs exhibit a *columnar* pattern, and directly applying SparseGPT leads to suboptimal results. We propose ROSE to reorder these layers, allowing columns with higher pruning loss to be processed first, thereby preserving more adjustable parameters. ROSE employs a pre-pruning step to estimate both column-wise and block-wise pruning loss. It leverages the relative range of block loss to identify *columnar* layers and subsequently performs two-level reordering operations on them. Specifically, within each block, columns are reordered in descending order of individual column loss, while the blocks are reordered in descending order of block loss. Extensive results on representative LLMs show ROSE surpasses SparseGPT and other counterpart pruning methods.

References

[1] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *ICML*, 2023.

[2] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[5] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[6] Yu Zhang, Chen Liu, et al. Native sparse attention: Accelerating MT without sacrificing quality. In *ACL*, 2025.

[7] Li Wang, Jun Li, et al. MACP: Parameter-efficient fine-tuning for abstractive summarization. In *ICML*, 2025.

[8] Alex Brown, Amrita Singh, et al. EPO: Multi-round rl pushes an 8B model beyond GPT-4 on open-domain QA. In *NeurIPS*, 2025.

[9] Sicheng Feng, Gongfan Fang, Xinyin Ma, and Xinchao Wang. Efficient reasoning models: A survey. *arXiv preprint arXiv:2504.10903*, 2025.

[10] Benoit Steiner, Mostafa Elhoushi, Jacob Kahn, and James Hegarty. Model: memory optimizations for deep learning. In *ICML*, 2023.

[11] Jian-Hao Luo, Hao Zhang, Hong-Yu Zhou, Chen-Wei Xie, Jianxin Wu, and Weiyao Lin. Thinet: Pruning cnn filters for a thinner net. *TPAMI*, 41(10):2525–2538, 2018.

[12] Haolei Bai, Siyong Jian, Tuo Liang, Yu Yin, and Huan Wang. Ressvd: Residual compensated svd for large language model compression. *arXiv preprint arXiv:2505.20112*, 2025.

[13] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *NeurIPS*, 1990.

[14] B. Hassibi, D.G. Stork, and G.J. Wolff. Optimal brain surgeon and general network pruning. In *NeurIPS*, 1993.

[15] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2016.

[16] Junhan Zhu, Hesong Wang, Mingluo Su, Zefang Wang, and Huan Wang. Obs-diff: Accurate pruning for diffusion models in one-shot. *arXiv preprint arXiv:2510.06751*, 2025.

[17] Sicheng Feng, Keda Tao, and Huan Wang. Is oracle pruning the true oracle? *arXiv preprint arXiv:2412.00143*, 2024.

[18] Kaiwen Tuo and Huan Wang. Sparsessm: Efficient selective structured state space models can be pruned in one-shot. *arXiv preprint arXiv:2506.09613*, 2025.

[19] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *CVPR*, 2019.

[20] Yue Yang, Zi Wang, Xiaozhe Zeng, Zhengxue Li, and Xinbo Gao. Global vision transformer pruning with hessian-aware saliency. In *CVPR*, 2023.

[21] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *ICLR*, 2017.

[22] Zhuang Liu, Jianguo Li, Zhiqi Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *ICCV*, 2017.

[23] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NeurIPS*, 2015.

[24] Y. Zhang, X. Wang, Y. Liu, and J. Liu. Magnitude attention-based dynamic pruning. In *ICML*, 2023.

[25] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training. *JMLR*, 22(241):1–124, 2022.

[26] Gui Ling, Ziyang Wang, and Qingwen Liu. Slimgpt: Layer-wise structured pruning for large language models. In *NeurIPS*, 2024.

[27] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.

[28] Yuxin Zhang, Lirui Zhao, Mingbao Lin, Yunyun Sun, Yiwu Yao, Xingjia Han, Jared Tanner, Shiwei Liu, and Rongrong Ji. Dynamic sparse no training: Training-free fine-tuning for sparse llms. In *ICLR*, 2024.

[29] Elias Frantar and Dan Alistarh. Optimal brain compression: A framework for accurate post-training quantization and pruning. In *NeurIPS*, 2022.

[30] Sidak Pal Singh and Dan Alistarh. Woodfisher: Efficient second-order approximations for model compression. In *NeurIPS*, 2020.

[31] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? In *NeurIPS*, 2019.

[32] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. Depgraph: Towards any structural pruning. In *CVPR*, 2023.

[33] Minsoo Kang and Bohyung Han. Operation-aware soft channel pruning using differentiable masks. In *ICML*, 2020.

[34] Stephen Zhang and Vardan Papyan. Oats: Outlier-aware pruning through sparse and low rank decomposition. In *ICLR*, 2025.

[35] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. In *NeurIPS*, 2022.

[36] Sungbin Shin, Wonpyo Park, Jaeho Lee, and Namhoon Lee. Rethinking pruning large language models: Benefits and pitfalls of reconstruction error minimization. In *EMNLP*, 2024.

[37] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. In *ICLR*, 2022.

[38] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[39] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[40] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[41] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[42] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.

[43] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 2021.

[44] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *AAAI*, 2020.

[45] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *EMNLP*, 2018.

[46] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

[47] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[48] L Sutawika, L Gao, H Schoelkopf, S Biderman, J Tow, B Abbasi, B Fattori, C Lovering, J Phang, A Thite, et al. Eleutherai/lm-evaluation-harness: Major refactor, 2013.

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[50] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[51] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 21(140):1–67, 2020.

A. Acknowledge

This paper is supported by Young Scientists Fund of the National Natural Science Foundation of China (NSFC) (No. 62506305), Zhejiang Leading Innovative and Entrepreneur Team Introduction Program (No. 2024R01007), Key Research and Development Program of Zhejiang Province (No. 2025C01026), Scientific Research Project of Westlake University (No. WU2025WF003), Chinese Association for Artificial Intelligence (CAAI) & Ant Group Research Fund - AGI Track (No. 2025CAAI-ANT-13). It is also supported by the research funds of the National Talent Program and Hangzhou Municipal Talent Program.

B. Proof of Equation 3

Equation 3 employs Cholesky decomposition to pre-compute and store the inverse Hessian information required for pruning from the first column to the last column of the matrix, thereby eliminating the need for iterative inversion of the Hessian. The formal derivation is provided below.

First, we perform the Cholesky decomposition on $\mathbf{H}^{-1}$. It can be written as:

$$\mathbf{H}^{-1} = \mathbf{L}\mathbf{L}^{\top}, \tag{8}$$

where $\mathbf{L}$ is a lower triangular matrix. Then, the original Hessian matrix can be denoted as:

$$\mathbf{H} = \left[\mathbf{H}^{-1}\right]^{-1} = \left[\mathbf{L}\mathbf{L}^{\top}\right]^{-1} = \left[\mathbf{L}^{\top}\right]^{-1}\mathbf{L}^{-1}. \tag{9}$$

According to matrix block multiplication, the sub-matrix $\mathbf{H}$ starting from the i-th row and i-th column can be expressed as:

$$\mathbf{H}_{i:,i:} = \left[\mathbf{L}^{\top}\right]^{-1}_{i:,i:}\mathbf{L}^{-1}_{i:,i:}. \tag{10}$$

Based on this equation, we have:

$$\begin{aligned}\left[\mathbf{H}_{i:,i:}\right]^{-1} &= \left[\left[\mathbf{L}^{\top}\right]^{-1}_{i:,i:}\mathbf{L}^{-1}_{i:,i:}\right]^{-1} \\ &= \left[\mathbf{L}^{-1}_{i:,i:}\right]^{-1}\left[\left[\mathbf{L}^{\top}\right]^{-1}_{i:,i:}\right]^{-1}.\end{aligned} \tag{11}$$

We can write the $\mathbf{L}$ and $\mathbf{L}^{-1}$ as block matrices:

$$\underbrace{\begin{bmatrix}\mathbf{L}_{:i,:i} & O\\ \mathbf{L}^{*} & \mathbf{L}_{i:,i:}\end{bmatrix}}_{\mathbf{L}}\cdot\underbrace{\begin{bmatrix}\mathbf{L}^{-1}_{:i,:i} & O\\ \mathbf{L}^{*} & \mathbf{L}^{-1}_{i:,i:}\end{bmatrix}}_{\mathbf{L}^{-1}} = \mathbf{I}_n. \tag{12}$$

where $\mathbf{I}_n$ is unit matrix with dimension n. The inverse matrix on the diagonal of a block diagonal matrix can be written as:

$$\left[\mathbf{L}^{-1}_{i:,i:}\right]^{-1} = \mathbf{L}_{i:,i:}. \tag{13}$$

Similarly, for the upper triangular matrix $\mathbf{L}^{\top}$, we can obtain:

$$\left[\left[\mathbf{L}^{\top}\right]^{-1}_{i:,i:}\right]^{-1} = \mathbf{L}^{\top}_{i:,i:}. \tag{14}$$

Substituting Equation 13 and Equation 14 into Equation 11 yields Equation 3.

Overall, Equation 3 pre-stores all Hessian inverse update information during the pruning process, which means the pruning order must be fully determined before pruning is executed. This is why `ROSE`pre-determines the entire pruning order in advance rather than determining it progressively during the pruning process.

C. ROSE Algorithm

We present the full procedure of ROSE in Algorithm 1. First, the weight matrix is partitioned into blocks. Within each block, we compute weight importance scores and perform minimum-element selection based on a target sparsity to generate the pruning loss matrix. This loss matrix is then divided into blocks, and the sum of elements within each block is computed as the block-wise loss. Next, the relative range of block loss is calculated. If the value is below a given threshold, we directly apply SparseGPT pruning. Otherwise, the two-step reordering operation is performed: first, columns are reordered within each block according to their loss values; then, all blocks are globally reordered based on their total loss. The activation matrix is correspondingly transformed, and all indices are stored. Subsequently, SparseGPT is applied to prune the reordered weight matrix. Finally, the resulting sparse matrix is restored to its original order using the previously saved indices.

Algorithm 1 ROSE

Input: Weight matrix $\mathbf{W}$, input activation $\mathbf{X}$, block size B_s, target sparsity $p\%$, identification threshold η
Output: Sparse matrix $\mathbf{W}$
$K \leftarrow \lceil N/B_s \rceil$
for $k = 1$ to K **do**
    $i_1 \leftarrow (k-1) \cdot B_s$, $i_2 \leftarrow \min(k \cdot B_s, N)$
    $\mathbf{W}^{(k)} \leftarrow \mathbf{W}[:, i_1 : i_2]$
    $\mathbf{X}^{(k)} \leftarrow \mathbf{X}[:, i_1 : i_2]$
    $\mathbf{S}_{ij}^{(k)} \leftarrow |\mathbf{W}_{ij}^{(k)}| \cdot \|\mathbf{X}_j^{(k)}\|_2$
    $\mathbf{L}^{(k)} \leftarrow p\%$ smallest scores in $\mathbf{S}_{ij}^{(k)}$
    $L^{(k)} \leftarrow \text{sum}\left(\mathbf{L}^{(k)}\right)$
end for
Calculate relative range τ of L; ▷ Equation 7
if $\tau > \eta$ **then**
    for $k = 1$ to K **do**
        $i_1 \leftarrow (k-1) \cdot B_s$, $i_2 \leftarrow \min(k \cdot B_s, N)$
        $\mathbf{W}^{(k)} \leftarrow \mathbf{W}[:, i_1 : i_2]$
        $\mathbf{X}^{(k)} \leftarrow \mathbf{X}[:, i_1 : i_2]$
        $\mathbf{W}^{(k)} \leftarrow \text{Reorder column}\left(\mathbf{W}^{(k)}\right)$ ▷ Equation 5
        $\mathbf{X}^{(k)} \leftarrow \text{Reorder column}\left(\mathbf{X}^{(k)}\right)$
    end for
    $\mathbf{W} \leftarrow \text{Reorder block}\,(\mathbf{W})$ ▷ Equation 6
    $\mathbf{X} \leftarrow \text{Reorder block}\,(\mathbf{X})$
    $\mathbf{W} \leftarrow \text{SparseGPT}\,(\mathbf{W}, \mathbf{X}, B_S, p\%)$
    $\mathbf{W} \leftarrow \text{Reorder back}\,(\mathbf{W})$
else
    $\mathbf{W} \leftarrow \text{SparseGPT}\,(\mathbf{W}, \mathbf{X}, B_S, p\%)$
end if
return $\mathbf{W}$

D. More Experimental Results

D.1. Combining Quantization

We have conducted additional experiments on sparsification-quantization joint compression. Specifically, we modify the block loss to be the sum of the pre-pruning loss and the pre-quantization loss, while keeping all other components of our method unchanged. The results of the joint compression under 4-bit and 8-bit quantization across varying sparsity levels are shown in Table 8 and Table 9. It can be observed that when combined with 4-bit quantization, our method achieves obvious improvements over SparseGPT in zero-shot accuracy across different models and sparsity levels. Under 8-bit quantization, our method outperforms SparseGPT in the vast majority of cases. This demonstrates the correctness of the core idea of our method and its potential for extension to other compression domains.

Table 8: Performance of zero-shot task accuracy (↑) on different models under **4-bit** quantization at different sparsity rates.

Group	Sparsity	Method	BoolQ	WinoG.	PIQA	OBQA	HellaS.	ARC-e	ARC-c	Avg.
LLaMA2-7B	0%	SparseGPT	**76.42**	**68.19**	**78.51**	42.20	**74.60**	71.97	44.28	**65.14**
		ROSE (ours)	75.84	67.88	78.45	**42.40**	**74.60**	**72.22**	**44.62**	**65.14**
	25%	SparseGPT	**74.92**	**68.90**	77.42	40.60	75.15	69.36	42.66	64.14
		ROSE (ours)	73.73	68.51	**78.18**	**40.60**	**75.38**	**70.83**	**46.50**	**64.81**
	50%	SparseGPT	73.61	**69.06**	75.90	**40.60**	69.31	63.51	39.93	61.70
		ROSE (ours)	**75.99**	68.35	**77.48**	**40.60**	**69.78**	**65.45**	**41.04**	**62.67**
Mistral-7B	0%	SparseGPT	80.61	71.43	80.30	43.20	77.43	74.66	50.00	68.23
		ROSE (ours)	**80.89**	**72.77**	**81.34**	**43.20**	**78.78**	**76.01**	**50.17**	**69.02**
	25%	SparseGPT	**81.44**	69.93	79.71	43.20	77.35	73.61	47.35	67.51
		ROSE (ours)	80.64	**71.11**	**80.96**	**44.00**	**78.51**	**75.80**	**48.98**	**68.57**
	50%	SparseGPT	80.86	69.38	78.13	**39.60**	71.23	71.30	43.26	64.82
		ROSE (ours)	**81.96**	**69.93**	**78.18**	39.00	**72.88**	**71.93**	**43.94**	**65.40**

Table 9: Performance of zero-shot task accuracy (↑) on different models under **8-bit** quantization at different sparsity rates.

Group	Sparsity	Method	BoolQ	WinoG.	PIQA	OBQA	HellaS.	ARC-e	ARC-c	Avg.
LLaMA2-7B	0%	SparseGPT	77.34	**68.82**	79.00	**44.20**	76.00	74.33	**46.25**	66.56
		ROSE (ours)	**77.40**	68.75	**79.05**	**44.20**	**76.01**	**74.54**	46.16	**66.59**
	25%	SparseGPT	76.97	69.61	78.84	44.60	**76.26**	**73.06**	**46.50**	66.55
		ROSE (ours)	**77.03**	**70.24**	**79.16**	**45.00**	76.25	72.43	46.33	**66.63**
	50%	SparseGPT	**75.78**	**70.17**	**77.15**	41.80	**71.11**	**67.85**	41.47	**63.62**
		ROSE (ours)	75.54	69.30	76.88	**42.80**	70.85	66.54	**41.55**	63.35
Mistral-7B	0%	SparseGPT	**81.99**	**73.88**	81.77	**45.00**	**80.40**	**78.49**	**52.30**	**70.55**
		ROSE (ours)	81.96	73.48	**81.94**	44.80	80.36	78.45	52.30	70.47
	25%	SparseGPT	**82.63**	72.77	**81.99**	43.60	**80.12**	77.78	**51.96**	70.12
		ROSE (ours)	82.51	**73.56**	81.94	**43.80**	80.06	**77.95**	51.79	**70.23**
	50%	SparseGPT	82.17	70.64	**79.33**	**40.80**	**75.29**	**73.99**	44.54	66.68
		ROSE (ours)	**82.45**	**70.72**	**79.33**	40.60	75.25	73.91	**45.82**	**66.87**

D.2. Varying Sparsity Rates

We extend our evaluation to broader sparsity regimes in Table 10 and 11. While both methods yield competitive results under moderate sparsity, our method exhibits increasingly clear advantages over SparseGPT at higher sparsity rates.

Table 10: Performance of zero-shot task accuracy (↑) for different unstructured pruning methods on LLaMA3-8B at different sparsity rates.

Sparsity	Method	BoolQ	WinoG.	PIQA	OBQA	HellaS.	ARC-e	ARC-c	Avg.
0%	Dense	81.38	72.61	80.79	45.00	79.16	77.74	53.24	69.99
30%	SparseGPT	82.14	74.03	80.09	**44.20**	78.54	**76.60**	50.43	69.43
	ROSE (ours)	**82.23**	**74.51**	**80.58**	**44.20**	**78.56**	76.43	**50.51**	**69.57**
40%	SparseGPT	**82.42**	**73.24**	**79.43**	42.80	**76.71**	**74.87**	50.09	68.51
	ROSE (ours)	82.14	72.61	79.00	**44.60**	76.62	74.37	**50.34**	**68.53**
50%	SparseGPT	78.41	**72.85**	**77.64**	**41.60**	72.92	68.73	44.11	65.18
	ROSE (ours)	**78.87**	72.45	77.20	40.80	**73.19**	**70.92**	**45.65**	**65.58**
60%	SparseGPT	75.78	**68.43**	**72.52**	**37.40**	**61.79**	**59.72**	**34.64**	**58.61**
	ROSE (ours)	**76.39**	66.69	72.20	35.00	61.61	57.58	33.19	57.52
70%	SparseGPT	68.78	55.88	61.15	**29.60**	**41.12**	40.11	**25.34**	46.00
	ROSE (ours)	**68.65**	**57.14**	**61.70**	28.80	40.60	**40.74**	24.83	**46.07**
80%	SparseGPT	53.76	49.25	**53.48**	25.60	**28.37**	**30.30**	20.31	37.30
	ROSE (ours)	**56.67**	**50.59**	**53.48**	**27.00**	28.36	30.01	**21.76**	**38.27**
90%	SparseGPT	**38.07**	48.62	51.52	25.00	**27.07**	**28.49**	**23.29**	34.58
	ROSE (ours)	37.83	**51.30**	**52.94**	**27.20**	26.99	28.07	23.04	**35.34**

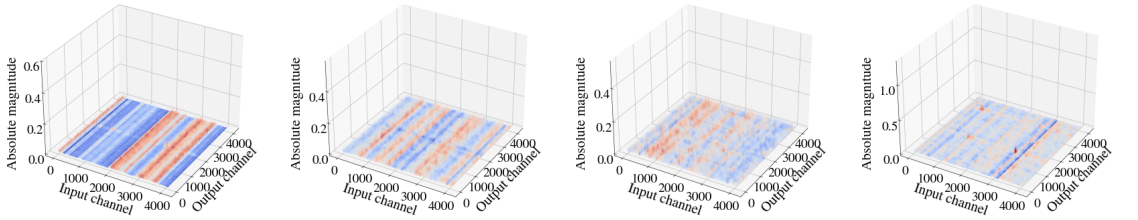
Table 11: Performance of zero-shot task accuracy (↑) for different unstructured pruning methods on Mistral-7B at different sparsity rates.

Sparsity	Method	BoolQ	WinoG.	PIQA	OBQA	HellaS.	ARC-e	ARC-c	Avg.
0%	Dense	82.14	73.80	82.26	44.20	80.42	78.20	52.30	70.47
30%	SparseGPT	82.45	**72.69**	**81.66**	43.00	**79.95**	**77.86**	**51.79**	**69.91**
	ROSE (ours)	**82.54**	72.45	**81.66**	**43.20**	**79.95**	77.53	51.54	69.84
40%	SparseGPT	82.75	72.45	81.34	42.60	78.49	**76.01**	48.55	68.88
	ROSE (ours)	**82.81**	**72.77**	**81.61**	**42.60**	**78.63**	75.88	**49.49**	**69.11**
50%	SparseGPT	**83.76**	**72.30**	**79.54**	**40.80**	**75.61**	74.37	44.97	**67.34**
	ROSE (ours)	82.39	71.67	79.43	40.60	75.61	**74.49**	**46.42**	67.23
60%	SparseGPT	**77.83**	**68.11**	**76.82**	**38.80**	67.25	**66.58**	**39.76**	**62.16**
	ROSE (ours)	76.27	67.32	76.39	38.20	**67.47**	66.37	37.63	61.38
70%	SparseGPT	**67.25**	58.25	65.56	**31.20**	46.64	46.80	27.22	48.99
	ROSE (ours)	65.14	**60.14**	**66.10**	30.00	**46.94**	**48.78**	**27.56**	**49.24**
80%	SparseGPT	53.91	**50.75**	**54.13**	**26.80**	29.56	29.63	**20.90**	37.95
	ROSE (ours)	**59.76**	50.20	53.86	26.60	**29.60**	**30.05**	20.82	**38.70**
90%	SparseGPT	37.83	**48.93**	**52.29**	25.80	**27.06**	**28.37**	**25.09**	**35.05**
	ROSE (ours)	**37.86**	47.67	51.80	**26.80**	**27.06**	**28.37**	24.74	34.90

E. Layer Analyses

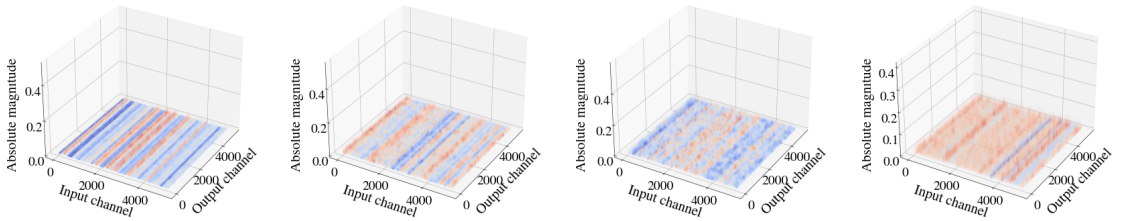
E.1. Columnar Layer Visualization

Figure 6-9 reveal layers with *columnar* pattern in current mainstream LLMs. Specifically, lots of columns with similar magnitudes tend to cluster together. Moreover, we also find that all matrices exhibiting this pattern are projection matrices of self-attention output.



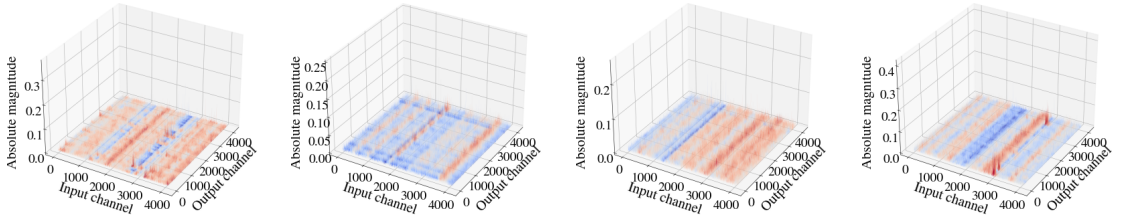
(a) Layer1 self_attn.o_proj (b) Layer11 self_attn.o_proj (c) Layer21 self_attn.o_proj (d) Layer31 self_attn.o_proj

Figure 6: Visualization of layers with columnar columnar distribution in **LLaMA2-7B**



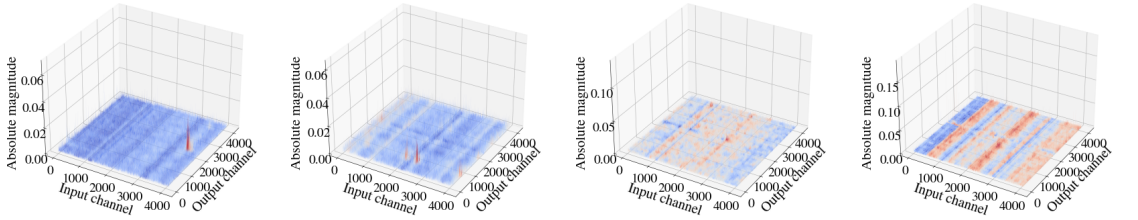
(a) Layer1 self_attn.o_proj (b) Layer11 self_attn.o_proj (c) Layer21 self_attn.o_proj (d) Layer31 self_attn.o_proj

Figure 7: Visualization of layers with tile columnar distribution in **LLaMA2-13B**



(a) Layer1 self_attn.o_proj (b) Layer11 self_attn.o_proj (c) Layer21 self_attn.o_proj (d) Layer31 self_attn.o_proj

Figure 8: Visualization of layers with the columnar distribution in **LLaMA3-8B**

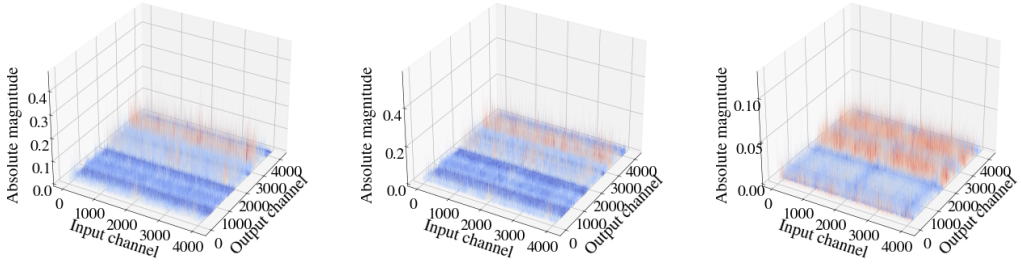


(a) Layer1 self_attn.o_proj (b) Layer11 self_attn.o_proj (c) Layer21 self_attn.o_proj (d) Layer31 self_attn.o_proj

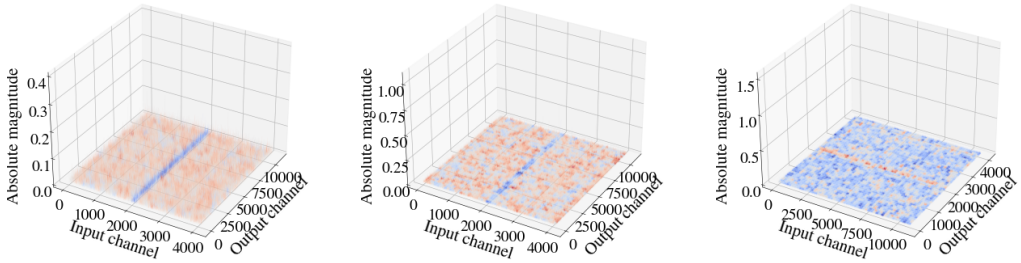
Figure 9: Visualization of layers with a columnar distribution in **Mistral-7B**

E.2. Non-columar Layer Visualization

Figure 10 and 11 present a visualization of the *non-columnar* layer weights. It can be observed that the weights of these layers are distributed relatively uniformly along the input channel dimension and no obvious columnar pattern is present.

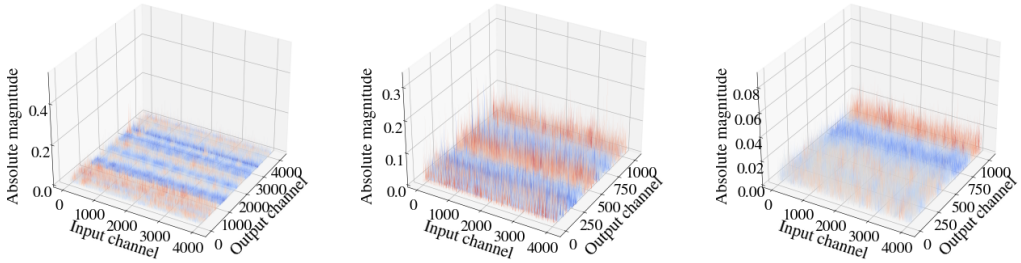


(a) self_attn.q_proj (b) self_attn.k_proj (c) self_attn.v_proj

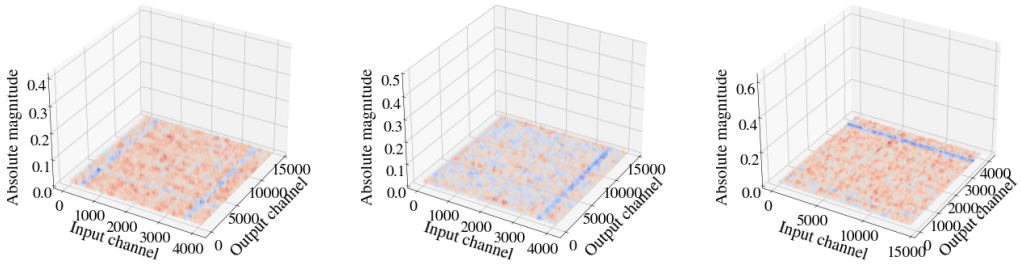


(d) mlp.up_proj (e) mlp.gate_proj (f) mlp.down_proj

Figure 10: Visualization of non-columar layers in **LLaMA2-7B**



(a) self_attn.q_proj (b) self_attn.k_proj (c) self_attn.v_proj



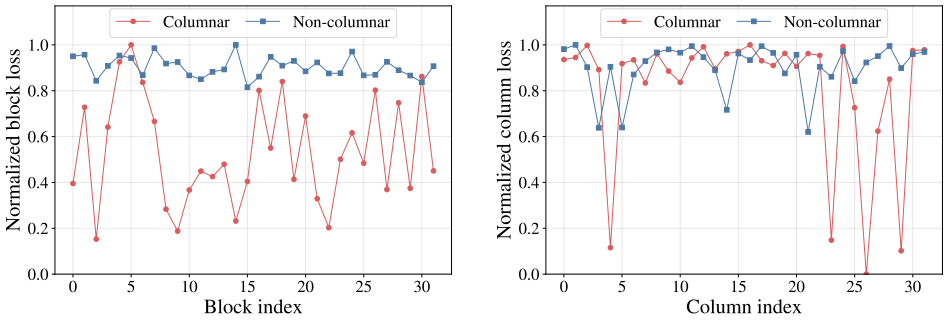
(d) mlp.up_proj (e) mlp.gate_proj (f) mlp.down_proj

Figure 11: Visualization of non-columar layers in **LLaMA3-8B**

E.3. Difference between Columar and Non-columnar Layer

We analyze distributions of column loss and block loss in *columnar* and *non-columnar* layers. Figure 12(a) shows that for *columnar* layers, there is a significant disparity in block loss. For instance, the block with the highest loss is ten times larger than the block with the lowest loss. In contrast, for *non-columnar* layers, due to the relatively uniform weight distribution, the differences in losses between blocks are minimal. Moreover, the variance between different columns within the same block in the columnar layer remains obvious, as shown in Figure 12(b).

Overall, for those *columnar* layers, both block loss and column loss exhibit significant fluctuations. This motivates us to adopt a two-stage reordering strategy: block reorder and column reorder, to let weights with potentially greater pruning errors be pruned earlier.



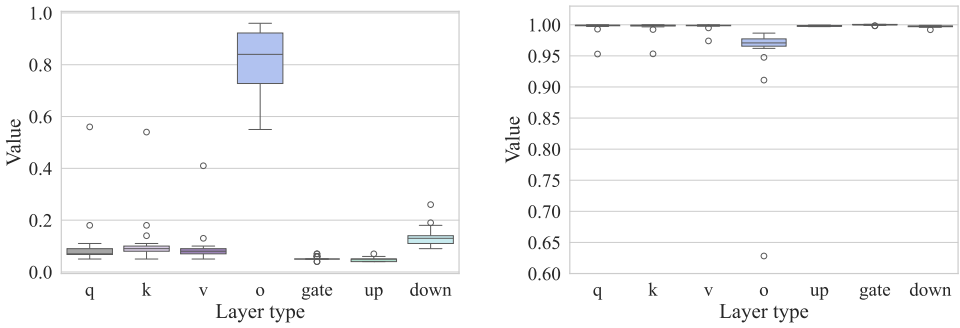
(a) Block loss (b) Column loss

Figure 12: Difference in block loss and column loss between the columnar and non-columnar layer. For the column loss, we selected 32 consecutive columns in one block for visualization.

E.4. Hyperparameter for Identifying Columnar Layer

We conduct a statistical analysis of the relative range of block loss across different types of layers in the whole model. Additionally, we compare the reconstruction error after reordering weight blocks with that of SparseGPT. The statistical results in LLaMA3-8B at 70% sparsity are shown in Figure 13. For all o_proj layers, the relative range of block loss exceeds 0.5. After applying the reordering strategy, the reconstruction error consistently is reduced. For other layer types, the relative block loss is generally much lower, mostly around 0.1, suggesting a more uniform block loss distribution. After reordering, their reconstruction errors either remain unchanged or exhibit slight reductions.

Based on the above observations, we set the threshold for identifying *columnar* layers to 0.5, ensuring the reconstruction error for identified layers after reordering is consistently reduced.



(a) Relative range of block loss (b) Relative reconstruction error

Figure 13: Relative range of block loss and the relative reconstruction error after reordering to that before reordering across layers in LLaMA3-8B at 70% sparsity.