EinSort: Sorting is All We Need for Tensorizing LLM

Toshiaki Koike-Akino [*1] **Jing Liu** [1] **Ye Wang** [1]

Abstract

Tensor networks provide efficient representations for compressing large neural networks. By carefully designing shapes and topologies, they can significantly reduce memory and computational costs. However, identifying implicit low-rank structures in large foundation models remains challenging due to their enormous scale and unstructured weight distributions. We propose an adaptive tensorization method that discovers inherent low-rank structure in a target tensor by index ordering. Experiments on weight and KV-cache compression demonstrate improved reconstruction quality compared to baselines.

1. Introduction

Large foundation models achieve remarkable performance across a variety of tasks (Katz et al., 2024). Nonetheless, their large parameter counts and memory requirements make deployment expensive (Schwartz et al., 2020). Numerous compression techniques have therefore been proposed (Xu & McAuley, 2023; Zhu et al., 2024; Bai et al., 2024a), including partial activation (Lin et al., 2024a), weight pruning (Bai et al., 2024b), quantization (Wang et al., 2024a), knowledge distillation (Hwang et al., 2024), and rank reduction (Yuan et al., 2023; Hwang et al., 2024; Saxena et al., 2024).

Among them, tensor decomposition methods (Lebedev et al., 2014; Sidiropoulos et al., 2017) provide a flexible framework for representing large neural networks with significantly fewer parameters (Novikov et al., 2015; Denil et al., 2013; Sainath et al., 2013; Jaderberg et al., 2014). Existing tensor network approaches (Roberts et al., 2019; Huggins et al., 2019) primarily focus on optimizing topology, contraction, and tensor ranks (Luo et al., 2024). However, one important degree of freedom has received relatively little attention: the ordering of tensor indices. In this work, we show that index ordering can dramatically alter the effective rank structure of tensors. In particular, simple sorting operations can expose latent low-rank structure that is otherwise hidden in pretrained LLM weights and KV caches. Motivated by this observation, we propose Einstein sorted sum (**EinSort**), a tensorization framework that augments Einstein summation with reversible permutation operators.

[1]Mitsubishi Electric Research Laboratories (MERL), 201 Broadway, Cambridge, MA 02139, USA.. Correspondence to: Toshiaki Koike-Akino <koike@merl.com>.

Accepted to the 43rd International Conference on Machine Learning (ICML) CoLoRAI Workshop, Seoul, South Korea, 2026.

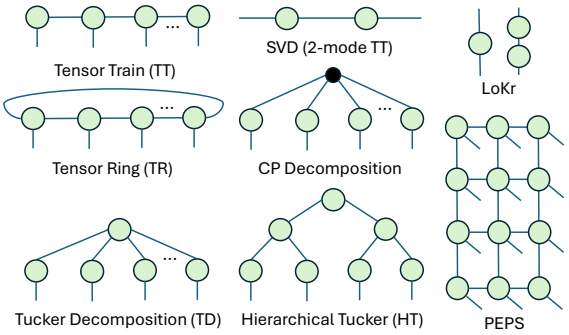


Figure 1. Various tensor networks in tensor diagrams.

Our contributions are summarized as follows: (1) We introduce EinSort, a tensor decomposition framework with adaptive index ordering for LLM compression. (2) We theoretically and empirically demonstrate that sorting operations can substantially reduce effective tensor rank. (3) We propose practical low-overhead permutation schemes for tensor compression. (4) We demonstrate improved compression quality on LLM weight and KV-cache compression tasks.

2. Tensorizing LLM for compression

2.1. Einstein summation as general tensor networks

Model decomposition is a widely used approach to compress large models, e.g., low-rank factorization via singular-value decomposition (SVD). More generally, tensorizing models can be viewed as a tensor network framework, including tensor train (TT), tensor ring (TR), canonical polyadic (CP), Tucker decomposition (TD), hierarchical Tucker (HT), and projected entangled-pair states (PEPS) as depicted in Fig 1. Almost all tensor networks can be represented by Einstein summation (**einsum**), which is a powerful format for linear algebra. Given tensor cores with proper shapes, a tensor network is factorized in the einsum form, e.g., for 4-mode TT: $X_{i,j,k,l} = \sum_{p,q,r} A_{i,p} B_{p,j,q} C_{q,k,r} D_{r,l}$, where $A_{i,p}$, $B_{p,j,q}$,

$C_{q,k,r}$, and $D_{r,l}$ are factorized cores. The set $\{i, j, k, l\}$ are physical site indices, and $\{p, q, r\}$ are bond indices whose maximum ranges are called as tensor ranks. If we reduce the tensor ranks, it can considerably reduce the memory footprint to represent a tensor X. For example, if X is of shape (d, d, d, d), the total number of parameters is a quartic order of d^4, while it can be reduced to $2d(R+1)R$ for the maximum rank of R. When $R = \rho d$ for $0 < \rho < 1$, it will be reduced to a cubic order of $2\rho d^2(\rho d + 1)$.

Several examples are listed in the pseudo code below:

```
TT = torch.einsum("ip,pjq,qkr,rl->ijkl", A,
     B, C, D)
TR = torch.einsum("mip,pjq,qkr,rlm->ijkl",
    A, B, C, D)
CP = torch.einsum("r,ri,rj,rk->ijk", A, B,
    C, D)
TD = torch.einsum("pqr,pi,qj,rk->ijk", A, B
    , C, D)
X = einops.rearrange("i p j q -> (i p) (j q
    )", TT) # unfolding 4-mode to 2-mode
```

Here we use `einops` for folding/unfolding. See Appendix B for more examples. As shown above, the einsum is a powerful tool to manipulate multi-linear operations. It gives an opportunity to optimize topologies for tensor networks by just designing the einsum equation (a string like `"ip,pj->ij"`) with properly shaped tensor cores.

In addition to the parameter reduction, the computational cost can be also reduced by designing the contraction order. For example with the 4-mode TT for a site dimension of $d = 32$ with a rank of $R = 20$, a naïve contraction from left to right requires 3.4×10^{10} FLOPs, whereas an optimized contraction order reduces it to 4.4×10^7, nearly 3 orders of magnitude when using `opt_einsum`.

2.2. Gauge fixing

Given a large tensor, we can numerically factorize into any tensor networks, while exact solutions generally do not exist, except in special cases such as matrix SVD. As einsum is a multi-linear operation, gradient optimization can be used to minimize the tensorization error. For TT, a recursive SVD per bond cutting offers a good initialization, while alternating least-squares (ALS) is often used to refine the estimate. We use `tensorly`.

Actually, the tensorization solution is not unique because they have **gauge** freedom (Evenbly, 2018). For example, consider factorizing X as $X \simeq AB$ for low-rank cores A and B. Optimal factors are given by truncated SVD, i.e., A is based on left-singular and B is on right-singular vectors of X. However, any full-rank junction matrix J has no impact when injecting it together with its inverse, i.e., $AB = A(JJ^{-1})B = (AJ)(J^{-1}B) = A'B'$. Hence, there are infinite solutions for A, B. By fixing the gauge freedom

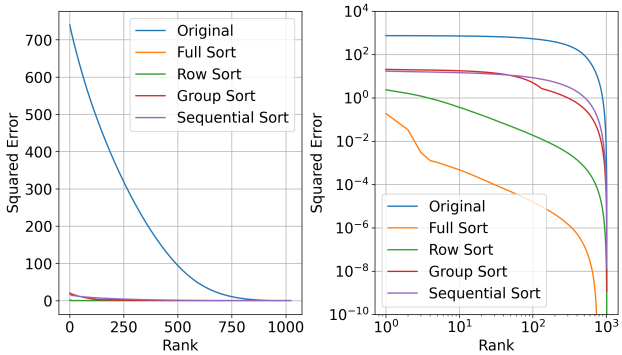


Figure 2. Tensorization error vs. rank for v_proj weights at the first layer of Qwen3-0.6B model. Left: linear; right: log-log plots.

such as canonicalization, the numerical stability (Phan et al., 2020) for tensor factorization can be improved.

More importantly, such gauge fixing can notably reduce the number of parameters when using a specific junction (Koike-Akino et al., 2026a). For instance, we can choose a junction to convert adjacent tensor cores into block identity form. In consequence, the number of parameters for 4-mode TT will be further reduced to $2d(R+1)R - 2R^2$ for example.

2.3. Index ordering

Using einsum, we can adaptively design tensor networks to factorize LLMs: determining einsum equation, tensor core shapes, folding/unfolding operations, numerical solvers, and gauge fixing for a target tensor to decompose (e.g., pre-trained weights). However, there are few papers on tensor network designs, which consider optimizing tensor indexing beyond a simple axis ordering like `"ijkl->kjli"`. In this paper, we show that index ordering plays a critical role in exposing low-rank structure in random tensors.

Let us first consider a toy example, assuming a 2×2 tensor W_0 and its re-ordered version W_π:

$$W_0 = \begin{array}{c} \xrightarrow{\quad j \quad} \\ i \downarrow \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \\ \text{swap} \end{array}, \qquad W_\pi = \begin{array}{c} \xrightarrow{\quad j \oplus i \quad} \\ i \downarrow \begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix} \end{array}. \tag{1}$$

The first matrix has singular values of $[3, 1]$, while the second one has $[\sqrt{10}, 0]$ and a reduced-rank of 1. Here W_π has an index re-ordering of $j \mapsto j \oplus i$ for indices (i, j). In fact, this index ordering is known as **entanglement** operations in quantum tensor networks and the above example is exactly same as controlled-NOT (CNOT) gates. Appendix L discusses their connections in more details.

We next show empirical results suggesting that the sorting operation can greatly reduce the required rank. Fig. 2 shows the tensorization loss in squared norm $\|W - \hat{W}\|^2$, where W is the original weights matrix and $\hat{W}$ is a reconstructed

version through a 2-mode TT. Specifically, we analyze the eigen spectrum for pretrained weights of an LLM. We also evaluate two sorting options, as given in the pseudo code:

```
S = torch.linalg.svdvals(W)
loss = S.square().flip(-1).cumsum(-1)
Ws, _ = torch.sort(W.view(-1)) # full-sort
S = torch.linalg.svdvals(Ws.reshape_as(W))
Wr, _ = torch.sort(W, dim=-1) # row-sort
```

As shown in Fig. 2, increasing the rank improves the accuracy, while the improvement is slow when original weights are decomposed by SVD without sorting. Notably, tensor sorting dramatically reduces reconstruction error, enabling near rank-1 reconstruction. From log-log plot, we can see that row sorting has relatively higher error than full sorting, whereas it is still more than 10 times lower than the one without sorting. Therefore, even with imperfect sorting, the index ordering has a great potential to improve the accuracy.

2.4. Theoretical analysis

We next provide a theoretical foundation supporting the reason why sorting can induce low-rank structure.

Lemma 2.1. *Let $X_0, \dots, X_{L-1}$ be i.i.d. random variables uniformly distributed on $[a, b]$, where $a < b$. Let $X_{(k)}$ for $k \in \mathbb{Z}_L$ denote their order statistics as $X_{(0)} \leq X_{(1)} \leq \cdots \leq X_{(L-1)}$, and reshape them into an $M \times N$ matrix X by $X_{i,j} = X_{(Ni+j)}$, for $i \in \mathbb{Z}_M$, $j \in \mathbb{Z}_N$, and $L = MN$. Then X is asymptotically approximated, in probability, by a matrix of rank at most 3.*

See proof in Appendix D, which uses a well-known order statistics (David & Nagaraja, 2004) to derive the mean and variance. As its mean is linear and the variance is bounded, we can find that a sorted matrix from uniformly distributed random variables is at most rank of 3 asymptotically, up to vanishing stochastic error. Although this does not guarantee that an arbitrary random matrix admits a low-rank factorization, we discuss more general cases in Appendix D.

2.5. EinSort: Index ordered tensor networks

Those results motivate us to consider sorting for tensor network designs. We then propose **EinSort** framework. Conceptually, we use a reversible permutation operation $\pi[\cdot]$ inside tensor decomposition. Let $\mathcal{T}_\theta[\cdot]$ denote the tensor decomposition with hyperparameters θ which determine einsum equation, shapes, topology, etc. The sorted tensor decomposition is written as $\mathcal{T}_\theta^\pi := \pi^{-1}[\mathcal{T}_\theta[\pi[X]]]$. It generalizes the einsum to find low-rank structure.

Although sorting may reduce the number of parameters for tensor cores with few rank, the memory overhead of storing permutations is not negligible. In fact, sorting L values requires at most $\lceil \log_2(L!) \rceil$ bits using factoradic/Lehmer code (Lehmer, 1960). For example, full sorting for a tensor of shape $(1024, 1024)$ requires roughly 18.6 bits per entry, which often exceeds the memory footprint of the original tensors in FP16. (Note that the row sorting requires 8.6 bits.) Therefore, we need to seek optimizing a tradeoff between sorting accuracy and tensor rank reduction.

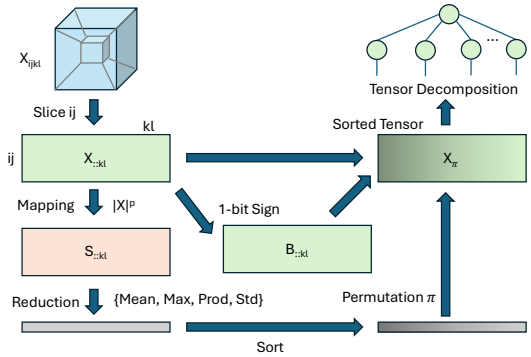


Figure 3. Sliced sorting with nonlinear mapping and reduction. Non-negative tensor decomposition keeps one-bit sign information.

Table 1. Reduction comparisons at the best power exponent.

Reduction	mean	max	min	median	std	prod
PPL	42.38	40.25	52.70	44.21	46.79	**33.17**
pow(p)	0.5	0.5	0.7	0.5	0.6	1.0

We introduce a simple sorting method reusing a shared permutation for multiple tensors. For a 4-mode tensor X with a site dimension of d, we may use a shared permutation across the first mode slice: $X_{:,j,k,l}$ to re-order the last three modes. It reduces the memory of permutation from $\lceil \log_2(d^4!) \rceil$ to $\lceil \log_2(d^3!) \rceil$ bits, i.e., more than d-fold memory reduction. Therefore, we can easily adjust the total memory. For example, with $d = 32$, we have $\lceil \log_2(d^3!) \rceil / d^4 \simeq 0.4$ bits per weight. For sorting metric, we consider some options for nonlinear mapping and reduction, including power scaling, and mean/max/min/median/std/prod operations. An example is given in the pseudo code below.

```
S = X.abs().pow(p) # nonlinear map score
S = S.std(dim=[0], keepdim=True).expand_as(
    X) # std reduction at the first mode
perm = torch.argsort(S.flatten(1), dim=-1)
X = X.flatten(1).gather(-1,perm).view_as(S)
```

We search for effective folding, slicing modes, power exponent and reduction options to increase the sorting accuracy. Besides the sorting metric, we introduce an approach to improve the accuracy by employing non-negative tensorization. Specifically, we project the original tensor into non-negative values before decomposition. To recover the negative values, we keep the sign bit of the original tensor, requiring only one additional bit. It is illustrated in Fig. 3. More details and example pseudo codes are found in Appendix E.

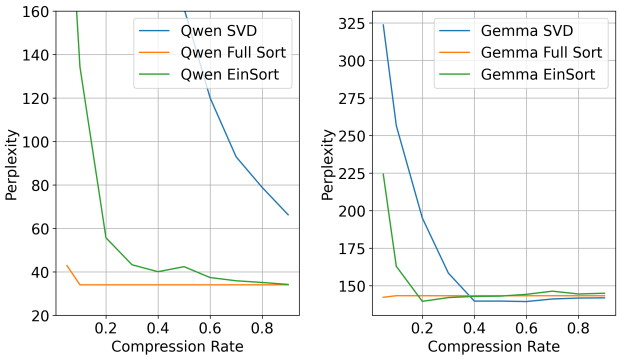


Figure 4. PPL for Qwen3-0.6B and Gemma3-4B models.

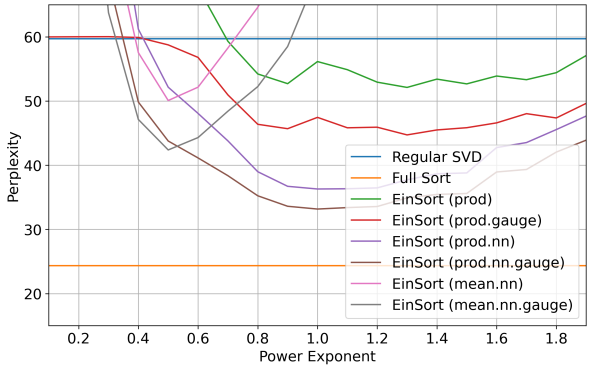


Figure 5. PPL vs. exponent for Qwen3-1.7B at 50% compression. Gauge fixing and non-negative tensorization improve the accuracy.

3. Experiments

We evaluate some LLMs, including Qwen3 (Yang et al., 2025) and Gemma3 (Gemma Team et al., 2025) on the witkitext-2 (WT2) (Merity et al., 2016) benchmark. See Appendix S for details of LLM models, and Appendices T/U for datasets and libraries we use. We focus on KV cache decomposition to reduce memory footprint for decoding. We partition the WT2 dataset into 64-token segments for prefilling, and then we decompose the KV cache for decoding up to 64 new tokens. For sorting slice, we use KV head axis for reduction. For Qwen3 models having 8 heads, the permutation memory can be at least 8-fold lower than the full sorting. For our case, the required amount of bits is just about 1.4 bit per weights to record the shared permutation.

Fig. 4 shows the perplexity (PPL) for Qwen3-0.6B and Gemma3-4B-it models when KV cache is compressed. We observe that full sorting achieves nearly un-compressed performance as expected. The regular SVD without sorting can rapidly degrade, whereas EinSort can achieve better PPL. We also observe that Gemma3 has higher tolerance while the base PPL is worse than Qwen3.

Fig. 5 shows the impact of nonlinear power scaling. We observe that the exponent near 1 is best for product reduction, implying that an absolute mapping is sufficient without exponent. However, we see that the mean reduction has a best exponent around 0.5. It is also confirmed that the gauge fixing can improve the accuracy because parameter redundancy can be eliminated. More importantly, non-negative tensorization is effective to further improve the performance at a cost of 1-bit additional memory. Table 1 compares different reduction methods at the same setting. It lists the PPL at the best power exponent for each reduction with gauge fixing and non-negative tensorization. We observe that product reduction is best across the reduction methods. Other reductions had lower best exponents around 0.5–0.7.

Table 2. Tensor network topology comparison (Qwen3-0.6B).

Compression	SVD	TT	TR	CP	TD
90%	47.38	27.04	36.94	**26.88**	27.04
80%	53.24	29.81	36.95	**27.99**	29.81

Table 3. Runtime analysis for Qwen3-0.6B decoding throughput relative to the original LLM without KV cache decomposition.

Device	SVD	TT	TR	CP	TD
CPU	78.7%	62.0%	62.7%	3.3%	32.5%
GPU	79.6%	61.9%	55.6%	3.7%	28.6%

Table 2 compares different tensor networks with square-root mapping, product reduction, non-negative factorization and gauge fixing. We fold KV cache into 3 mode tensor except for SVD. For this setting, CP decomposition is slightly better than other topologies, and TR is the worst one. Table 3 lists the runtime results for the same setting. KV compression inevitably decreases the LLM decoding throughput. For both Apple M1 CPU and NVIDIA A40 GPU devices, TT/TR had around 40% speed down, whereas TD had about 70% loss even though it has nearly the same performance as TT. More importantly, CP is way slower because it requires iterative Khatri–Rao product for all cores. Note that the current implementation prioritizes validating the compression principle rather than optimized inference kernels yet.

See Appendix R for more benchmark results, including different LLMs, GSM8K math reasoning (Cobbe et al., 2021), TextVQA visual reasoning (Singh et al., 2019) and LIBERO robot manipulation tasks (Liu et al., 2023a).

4. Conclusion

We proposed a new EinSort framework, which employs index ordering for tensor decomposition to discover implicit low-rank structure for LLM compression. We revealed that sorting is a powerful tool to reduce the tensor rank. We then introduced a simple sorting mechanism and demonstrated a potential advantage on some models.

Impact Statement

This paper's goal is to advance the field of machine learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted.

References

Abdin, M., Aneja, J., Behl, H., Bubeck, S., Eldan, R., Gunasekar, S., Harrison, M., Hewett, R. J., Javaheripi, M., Kauffmann, P., et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.

Abouelenin, A., Ashfaq, A., Atkinson, A., Awadalla, H., Bach, N., Bao, J., Benhaim, A., Cai, M., Chaudhary, V., Chen, C., et al. Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-LoRAs. *arXiv preprint arXiv:2503.01743*, 2025.

Abronin, V., Naumov, A., Mazur, D., Bystrov, D., Tsarova, K., Melnikov, A., Dolgov, S., Brasher, R., and Perelshein, M. TQCompressor: improving tensor decomposition methods in neural networks via permutations. In *2024 IEEE 7th International Conference on Multimedia Information Processing and Retrieval (MIPR)*, pp. 503–506. IEEE, 2024.

Aneja, J., Harrison, M., Joshi, N., LaBonte, T., Langford, J., and Salinas, E. Phi-4-reasoning-vision-15B technical report. *arXiv preprint arXiv:2603.03975*, 2026.

Arai, Y. and Ichikawa, Y. Quantization error propagation: Revisiting layer-wise post-training quantization. *arXiv preprint arXiv:2504.09629*, 2025.

Assran, M., Bardes, A., Fan, D., Garrido, Q., Howes, R., Muckley, M., Rizvi, A., Roberts, C., Sinha, K., Zholus, A., et al. V-JEPA 2: Self-supervised video models enable understanding, prediction and planning. *arXiv preprint arXiv:2506.09985*, 2025.

Bai, G., Chai, Z., Ling, C., Wang, S., Lu, J., Zhang, N., Shi, T., Yu, Z., Zhu, M., Zhang, Y., et al. Beyond efficiency: A systematic survey of resource-efficient large language models. *arXiv preprint arXiv:2401.00625*, 2024a.

Bai, G., Li, Y., Ling, C., Kim, K., and Zhao, L. SparseLLM: Towards global pruning for pre-trained language models. *arXiv preprint arXiv:2402.17946*, 2024b.

Bai, S., Cai, Y., Chen, R., Chen, K., Chen, X., Cheng, Z., Deng, L., Ding, W., Gao, C., Ge, C., et al. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Bershatsky, D., Cherniuk, D., Daulbaev, T., Mikhalev, A., and Oseledets, I. LoTR: Low tensor rank weight adaptation. *arXiv preprint arXiv:2402.01376*, 2024.

Beyer, L., Steiner, A., Pinto, A. S., Kolesnikov, A., Wang, X., Salz, D., Neumann, M., Alabdulmohsin, I., Tschannen, M., Bugliarello, E., et al. PaliGemma: A versatile 3B VLM for transfer. *arXiv preprint arXiv:2407.07726*, 2024.

Brayton, R. K., Hachtel, G. D., McMullen, C., and Sangiovanni-Vincentelli, A. *Logic minimization algorithms for VLSI synthesis*, volume 2. Springer Science & Business Media, 1984.

Cai, Z., Zhang, Y., Gao, B., Liu, Y., Li, Y., Liu, T., Lu, K., Xiong, W., Dong, Y., Hu, J., and Xiao, W. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=ayi7qezU87.

Chang, C.-C., Lin, W.-C., Lin, C.-Y., Chen, C.-Y., Hu, Y.-F., Wang, P.-S., Huang, N.-C., Ceze, L., Abdelfattah, M. S., and Wu, K.-C. Palu: Compressing KV-cache with low-rank projection. *arXiv preprint arXiv:2407.21118*, 2024.

Chavan, A., Liu, Z., Gupta, D., Xing, E., and Shen, Z. One-for-all: Generalized LoRA for parameter-efficient fine-tuning. *arXiv preprint arXiv:2306.07967*, 2023.

Chen, X., Liu, J., Wang, Y., Wang, P., Brand, M., Wang, G., and Koike-Akino, T. SuperLoRA: Parameter-efficient unified adaptation for large vision models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8050–8055, 2024a.

Chen, Z., Dangovski, R., Loh, C., Dugan, O., Luo, D., and Soljačić, M. QuanTA: Efficient high-rank fine-tuning of LLMs with quantum-informed tensor adaptation. *Advances in Neural Information Processing Systems*, 37: 92210–92245, 2024b.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *URL https://arxiv. org/abs/2110.14168*, 9, 2021.

David, H. A. and Nagaraja, H. N. *Order statistics*. John Wiley & Sons, 2004.

Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and De Freitas, N. Predicting parameters in deep learning. *Advances in neural information processing systems*, 26, 2013.

Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. Exploiting linear structure within convolutional networks for efficient evaluation. *Advances in neural information processing systems*, 27, 2014.

Ding, M., Xiao, B., Codella, N., Luo, P., Wang, J., and Yuan, L. DaViT: Dual attention vision transformers. In *European conference on computer vision*, pp. 74–92. Springer, 2022.

Dong, X., Chen, S., and Pan, S. Learning to prune deep neural networks via layer-wise optimal brain surgeon. *Advances in neural information processing systems*, 30, 2017.

Dong, Z., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. HAWQ: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 293–302, 2019.

Edalati, A., Tahaei, M., Kobyzev, I., Nia, V. P., Clark, J. J., and Rezagholizadeh, M. KronA: Parameter efficient tuning with kronecker adapter. *arXiv preprint arXiv:2212.10650*, 2022.

Evenbly, G. Gauge fixing, canonical forms, and optimal truncations in tensor networks with closed loops. *Physical Review B*, 98(8):085155, 2018.

Frantar, E. and Alistarh, D. SparseGPT: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pp. 10323–10337. PMLR, 2023.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Fu, D., Arora, S., Grogan, J., Johnson, I., Eyuboglu, E. S., Thomas, A., Spector, B., Poli, M., Rudra, A., and Ré, C. Monarch mixer: A simple sub-quadratic GEMM-based architecture. *Advances in Neural Information Processing Systems*, 36:77546–77603, 2023.

Gemma Team, Kamath, A., Ferret, J., Pathak, S., Vieillard, N., Merhej, R., Perrin, S., Matejovicova, T., Ramé, A., Rivière, M., et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

Grover, A., Wang, E., Zweig, A., and Ermon, S. Stochastic optimization of sorting networks via continuous relaxations. *arXiv preprint arXiv:1903.08850*, 2019.

Haider, E., Perez-Becker, D., Portet, T., Madan, P., Garg, A., Ashfaq, A., Majercak, D., Wen, W., Kim, D., Yang, Z., et al. Phi-3 safety post-training: Aligning language models with a "break-fix" cycle. *arXiv preprint arXiv:2407.13833*, 2024.

Hassibi, B., Stork, D., and Wolff, G. Optimal brain surgeon: Extensions and performance comparisons. *Advances in neural information processing systems*, 6, 1993.

Hayou, S., Ghosh, N., and Yu, B. LoRA+: Efficient low rank adaptation of large models. In *International Conference on Machine Learning*, 2024.

Hsieh, C.-Y., Li, C.-L., Yeh, C.-K., Nakhost, H., Fujii, Y., Ratner, A., Krishna, R., Lee, C.-Y., and Pfister, T. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*, 2023.

Hu, E. J., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2021.

Huggins, W., Patil, P., Mitchell, B., Whaley, K. B., and Stoudenmire, E. M. Towards quantum machine learning with tensor networks. *Quantum Science and technology*, 4(2):024001, 2019.

Hwang, I., Park, H., Lee, Y., Yang, J., and Maeng, S. PC-LoRA: Low-rank adaptation for progressive model compression with knowledge distillation. *arXiv preprint arXiv:2406.09117*, 2024.

Intelligence, P., Black, K., Brown, N., Darpinian, J., Dhabalia, K., Driess, D., Esmail, A., Equi, M., Finn, C., Fusai, N., et al. $\pi_{0.5}$: a vision-language-action model with open-world generalization. *arXiv preprint arXiv:2504.16054*, 2025.

Jaderberg, M., Vedaldi, A., and Zisserman, A. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

Katz, D. M., Bommarito, M. J., Gao, S., and Arredondo, P. GPT-4 passes the bar exam. *Philosophical Transactions of the Royal Society A*, 382(2270):20230254, 2024.

Koike-Akino, T., Liu, J., and Wang, Y. μ-moe: Test-time pruning as micro-grained mixture-of-experts. *arXiv preprint arXiv:2505.18451*, 2025a.

Koike-Akino, T., Tonin, F., Wu, Y., Wu, F. Z., Candogan, L. N., and Cevher, V. Quantum-PEFT: Ultra parameter-efficient fine-tuning. *arXiv preprint arXiv:2503.05431*, 2025b.

Koike-Akino, T., Chen, X., Liu, J., Wang, Y., Wang, P. P., and Brand, M. LatentLLM: Activation-aware transform to multi-head latent attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 22644–22652, 2026a.

Koike-Akino, T., Liu, J., and Wang, Y. TTQ: Activation-aware test-time quantization to accelerate LLM inference on the fly. *arXiv preprint arXiv:2603.19296*, 2026b.

Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I., and Lempitsky, V. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

LeCun, Y., Denker, J., and Solla, S. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

Lehmer, D. H. Teaching combinatorial tricks to a computer. In *Proceedings of Symposia in Applied Mathematics*, pp. 179–193. American Mathematical Society, 1960.

Li, M., Tromp, J., and Vitányi, P. Reversible simulation of irreversible computation. *Physica D: Nonlinear Phenomena*, 120(1-2):168–176, 1998.

Li, Y., Lee, D., Yin, R., and Panda, P. Optimal brain decomposition for accurate LLM low-rank approximation. *arXiv preprint arXiv:2604.00821*, 2026.

Lin, B., Tang, Z., Ye, Y., Cui, J., Zhu, B., Jin, P., Zhang, J., Ning, M., and Yuan, L. MoE-LlaVa: Mixture of experts for large vision-language models. *arXiv preprint arXiv:2401.15947*, 2024a.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. AWQ: Activation-aware weight quantization for LLM compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.-M., Wang, W.-C., Xiao, G., Dang, X., Gan, C., and Han, S. AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024b.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al. DeepSeek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024a.

Liu, B., Zhu, Y., Gao, C., Feng, Y., Liu, Q., Zhu, Y., and Stone, P. LIBERO: Benchmarking knowledge transfer for lifelong robot learning. *Advances in Neural Information Processing Systems*, 36:44776–44791, 2023a.

Liu, J., Koike-Akino, T., Wang, P., Brand, M., Wang, Y., and Parsons, K. LoDA: Low-dimensional adaptation of large language models. In *NeurIPS'23 Workshop on on Efficient Natural Language and Speech Processing*, 2023b.

Liu, J., Koike-Akino, T., Wang, Y., Mansour, H., and Brand, M. AWP: Activation-aware weight pruning and quantization with projected gradient descent. *arXiv preprint arXiv:2506.10205*, 2025.

Liu, Z., Yuan, J., Jin, H., Zhong, S., Xu, Z., Braverman, V., Chen, B., and Hu, X. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In Salakhutdinov, R., Kolter, Z., Heller, K., Weller, A., Oliver, N., Scarlett, J., and Berkenkamp, F. (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32332–32344. PMLR, 21–27 Jul 2024b. URL https://proceedings.mlr.press/v235/liu24bz.html.

Luo, S., Liu, M., Yu, Y., Ren, S., and Bai, Y. An adaptive tensor-train decomposition approach for efficient deep neural network compression. *arXiv preprint arXiv:2408.01534*, 2024.

McCluskey, E. J. Minimization of Boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956.

Mena, G., Belanger, D., Linderman, S., and Snoek, J. Learning latent permutations with Gumbel-Sinkhorn networks. *arXiv preprint arXiv:1802.08665*, 2018.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. P. Tensorizing neural networks. *Advances in neural information processing systems*, 28, 2015.

Orús, R. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of physics*, 349:117–158, 2014.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Peebles, W. and Xie, S. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 4195–4205, 2023.

Phan, A.-H., Sobolev, K., Sozykin, K., Ermilov, D., Gusak, J., Tichavskỳ, P., Glukhov, V., Oseledets, I., and Cichocki, A. Stable low-rank tensor decomposition for compression of convolutional neural network. In *European Conference on Computer Vision*, pp. 522–539. Springer, 2020.

Prillo, S. and Eisenschlos, J. SoftSort: A continuous relaxation for the argsort operator. In *International Conference on Machine Learning*, pp. 7793–7802. PMLR, 2020.

Roberts, C., Milsted, A., Ganahl, M., Zalcman, A., Fontaine, B., Zou, Y., Hidary, J., Vidal, G., and Leichenauer, S. TensorNetwork: A library for physics and machine learning. *arXiv preprint arXiv:1905.01330*, 2019.

Saha, R., Sagan, N., Srivastava, V., Goldsmith, A., and Pilanci, M. Compressing large language models using low rank and low precision decomposition. *Advances in Neural Information Processing Systems*, 37:88981–89018, 2024.

Sainath, T. N., Kingsbury, B., Sindhwani, V., Arisoy, E., and Ramabhadran, B. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6655–6659. IEEE, 2013.

Saxena, U., Saha, G., Choudhary, S., and Roy, K. Eigen attention: Attention in low-rank space for KV cache compression. *arXiv preprint arXiv:2408.05646*, 2024.

Schollwöck, U. The density-matrix renormalization group in the age of matrix product states. *Annals of physics*, 326(1):96–192, 2011.

Schwartz, R., Dodge, J., Smith, N. A., and Etzioni, O. Green AI. *Communications of the ACM*, 63(12):54–63, 2020.

Sidiropoulos, N. D., De Lathauwer, L., Fu, X., Huang, K., Papalexakis, E. E., and Faloutsos, C. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on signal processing*, 65(13):3551–3582, 2017.

Singh, A., Natarjan, V., Shah, M., Jiang, Y., Chen, X., Parikh, D., and Rohrbach, M. Towards VQA models that can read. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8317–8326, 2019.

Sinha, A. K. and Fleuret, F. AA-SVD: Anchored and adaptive SVD for large language model compression. *arXiv preprint arXiv:2604.02119*, 2026.

Sun, J., Zhang, W., Qi, Z., Ren, S., Liu, Z., Zhu, H., Sun, G., Jin, X., and Chen, Z. VLA-JEPA: Enhancing vision-language-action model with latent world model. *arXiv preprint arXiv:2602.10098*, 2026.

Sun, M., Liu, Z., Bair, A., and Kolter, J. Z. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.

Takeshita, O. Y. Permutation polynomial interleavers: An algebraic-geometric perspective. *IEEE Transactions on Information Theory*, 53(6):2116–2132, 2007.

Team, G., Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., Sifre, L., Rivière, M., Kale, M. S., Love, J., et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

Vidal, G. Class of quantum many-body states that can be efficiently simulated. *Physical review letters*, 101(11): 110501, 2008.

Wang, C., Wang, Z., Xu, X., Tang, Y., Zhou, J., and Lu, J. Q-VLM: Post-training quantization for large vision-language models. *arXiv preprint arXiv:2410.08119*, 2024a.

Wang, X., Zheng, Y., Wan, Z., and Zhang, M. SVD-LLM: Truncation-aware singular value decomposition for large language model compression. *arXiv preprint arXiv:2403.07378*, 2024b.

Williams, M. and Aletras, N. On the impact of calibration data in post-training quantization and pruning. *arXiv preprint arXiv:2311.09755*, 2023.

Wolf, T. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Xiao, B., Wu, H., Xu, W., Dai, X., Hu, H., Lu, Y., Zeng, M., Liu, C., and Yuan, L. Florence-2: Advancing a unified representation for a variety of vision tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4818–4829, 2024a.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=NG7sS51zVF.

Xu, C. and McAuley, J. A survey on model compression and acceleration for pretrained language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 10566–10575, 2023.

Yan, X., Li, Z., Zhang, T., Qin, H., Kong, L., Zhang, Y., and Yang, X. Recalkv: Low-rank kv cache compression via head reordering and offline calibration, 2025. URL https://arxiv.org/abs/2505.24357.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yeh, S.-Y., Hsieh, Y.-G., Gao, Z., Yang, B. B. W., Oh, G., and Gong, Y. Navigating text-to-image customization: From lyCORIS fine-tuning to model evaluation. In *International Conference on Learning Representations*, 2024.

Yuan, Z., Shang, Y., Song, Y., Wu, Q., Yan, Y., and Sun, G. ASVD: Activation-aware singular value decomposition for compressing large language models. *arXiv preprint arXiv:2312.05821*, 2023.

Yuan, Z., Shang, Y., Zhou, Y., Dong, Z., Zhou, Z., Xue, C., Wu, B., Li, Z., Gu, Q., Lee, Y. J., et al. LLM inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., and Smola, A. J. Deep sets. *Advances in neural information processing systems*, 30, 2017.

Zandieh, A., Daliri, M., and Han, I. QJL: 1-bit quantized JL transform for KV cache quantization with zero overhead, 2024. URL https://openreview.net/forum?id=xHPVGmLXjd.

Zandieh, A., Daliri, M., Hadian, M., and Mirrokni, V. Turboquant: Online vector quantization with near-optimal distortion rate. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=tO3ASKZlok.

Zhai, X., Mustafa, B., Kolesnikov, A., and Beyer, L. Sigmoid loss for language image pre-training. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 11975–11986, 2023.

Zhang, R., Wang, K., Liu, L., Wang, S., Cheng, H., Zhang, C., and yelong shen. LoRC: Low-rank compression for LLMs KV cache with a progressive compression strategy, 2025. URL https://openreview.net/forum?id=NI8AUSAc4i.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., Wang, Z. A., and Chen, B. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 34661–34710. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/6ceefa7b15572587b78ecfcebb2827f8-Paper-Conference.pdf.

Zheng, J., Li, J., Wang, Z., Liu, D., Kang, X., Feng, Y., Zheng, Y., Zou, J., Chen, Y., Zeng, J., et al. X-VLA: Soft-prompted transformer as scalable cross-embodiment vision-language-action model. *arXiv preprint arXiv:2510.10274*, 2025.

Zhu, X., Li, J., Liu, Y., Ma, C., and Wang, W. A survey on model compression for large language models. *Transactions of the Association for Computational Linguistics*, 12:1556–1577, 2024.

Zi, B., Qi, X., Wang, L., Wang, J., Wong, K.-F., and Zhang, L. Delta-LoRa: Fine-tuning high-rank parameters with the delta of low-rank matrices. *arXiv preprint arXiv:2309.02411*, 2023.

A. Related work

A.1. Model compression

The field of model compression for LLMs has aimed at mitigating the substantial computation and memory requirements (Zhu et al., 2024; Yuan et al., 2024). Such methods primarily fall into four categories: weight quantization (Lin et al., 2024b; Frantar et al., 2022; Wang et al., 2024a), network pruning (LeCun et al., 1989; Hassibi et al., 1993; Frantar & Alistarh, 2023; Bai et al., 2024b), knowledge distillation (Hsieh et al., 2023; Hwang et al., 2024), and rank reduction (Yuan et al., 2023; Liu et al., 2024a; Hwang et al., 2024; Saxena et al., 2024; Saha et al., 2024).

A.2. Low-rank compression

Low-rank compression for neural networks builds on the empirical observation that trained weight matrices and convolutional kernels are often highly redundant and admit accurate low-rank approximations. Early work by Denil et al. (2013) showed that a large fraction of network parameters can be predicted from a small subset, motivating structured factorization approaches. Sainath et al. (2013) and Jaderberg et al. (2014) popularized the use of SVD and related factorization techniques, achieving substantial speedups with limited accuracy loss. This line was extended by Denton et al. (2014), who applied low-rank approximations to convolutional filters, and Lebedev et al. (2014); Sidiropoulos et al. (2017), who introduced tensor decomposition methods such as Tucker and CP decompositions, though care is required to avoid instability or accuracy degradation (Phan et al., 2020).

A.3. KV cache reduction

KV-cache memory is a major bottleneck in long-context LLM inference, growing linearly with sequence length, batch size, number of layers, and KV hidden dimension. Existing approaches to reduce KV-cache can be broadly categorized into token eviction, quantization, and representation compression.

Token eviction methods reduce memory by selectively discarding cached tokens. StreamingLLM (Xiao et al., 2024b) preserves initial attention-sink tokens together with a sliding window of recent tokens for stable streaming decoding, while H2O (Zhang et al., 2023) retains a balance of recent tokens and heavy-hitter tokens that have significant contribution to attention scores. PyramidKV (Cai et al., 2025) further improves eviction-based compression by allocating nonuniform cache budgets across layers, retaining more KV entries in lower layers and fewer in higher layers. Although effective, eviction-based approaches may discard information needed for long-range reasoning.

Quantization methods compress KV tensors without removing tokens. KIVI (Liu et al., 2024b) proposes tuning-free asymmetric 2-bit KV-cache quantization, using per-channel quantization for keys and per-token quantization for values. QJL (Zandieh et al., 2024) applies a Johnson-Lindenstrauss transform followed by sign-bit quantization, eliminating the need to store scale and zero-point quantization metadata while providing an unbiased, low-distortion estimator for query-key inner products. TurboQuant (Zandieh et al., 2026) extends this direction with online vector quantization, combining an MSE-oriented quantizer with QJL-based residual correction to improve inner-product estimation.

Low-rank compression reduces KV-cache memory by exploiting redundancy in hidden dimensions or KV projections. LoRC (Zhang et al., 2025) applies low-rank approximation to KV weight matrices with layer-wise sensitivity and progressive compression. ReCalKV (Yan et al., 2025) introduces a post-training low-rank KV-cache compression method with tailored strategies for keys and values: Head-wise similarity-aware reordering for keys, which clusters structurally similar heads and applies grouped SVD to the key projection matrix, and Offline Calibration and Matrix Fusion for values, which calibrates the low-rank value projection matrix using a small calibration dataset.

A.4. Low-rank adaptation

LoRA (Hu et al., 2021) updates the pretrained weight matrix through the addition of a product of two low-rank matrices with widespread adoption (Zi et al., 2023; Chavan et al., 2023; Hayou et al., 2024). Many variants were introduced, e.g., CP decomposition with LoTR (Bershatsky et al., 2024) and Tucker decomposition with SuperLoRA (Chen et al., 2024a), and nonlinear low-rank mapping with LoDA (Liu et al., 2023b), Hadamard (Yeh et al., 2024) and Kronecker product (Edalati et al., 2022), and quantum tensor networks (Chen et al., 2024b; Koike-Akino et al., 2025b).

A.5. Activation-aware compression

Decomposition ASVD (Yuan et al., 2023) improves the low-rank decomposition by dealing with activation statistics. It was applied to SVD-LLM (Wang et al., 2024b) and Palu (Chang et al., 2024). AA-SVD (Sinha & Fleuret, 2026) incorporates error propagation factor motivated by QEP (Arai & Ichikawa, 2025). And further OBD-LLM (Li et al., 2026) uses gradient information as an approximated Hessian to improve ASVD.

Quantization HAWQ (Dong et al., 2019) uses layer-wise quantization based on optimal brain pruning (LeCun et al., 1989). Then, GPTQ (Frantar et al., 2022) extends it using zero-shot calibration, while activation-aware quantization (AWQ) (Lin et al., 2023) uses activation-dependent scaling. AWP (Liu et al., 2025) uses projected gradient descent for activation aware quantization and pruning based on compressed sensing framework. QEP (Arai & Ichikawa, 2025) mitigates error propagation across layers.

Pruning Activation-aware pruning methods (Williams & Aletras, 2023) include SparseGPT (Frantar & Alistarh, 2023), which uses layer-wise optimal brain surgeon (Dong et al., 2017; Hassibi et al., 1993; LeCun et al., 1989). SparseLLM (Bai et al., 2024b) extends with joint module compression. Wanda (Sun et al., 2023) greatly simplifies the pruning mechanism, and μ-MoE (Koike-Akino et al., 2025a) applied it for online pruning

A.6. Permutation/sorting networks

Permutation or sorting neural networks are architectures designed to process sets or sequences for ordering elements. Deep Sets by Zaheer et al. (2017) establishes that any permutation-invariant function over a set can be decomposed into a sum of elementwise embeddings followed by a global function, providing a theoretical foundation for set-based learning. In parallel, differentiable sorting and ranking networks have emerged to learn orderings explicitly, such as NeuralSort (Grover et al., 2019) and SoftSort (Prillo & Eisenschlos, 2020), which provide continuous relaxations of sorting operators to enable gradient-based optimization. A closely related approach is Gumbel–Sinkhorn network (Mena et al., 2018), which leverages entropic optimal transport and Gumbel noise to produce differentiable approximations of permutation matrices via Sinkhorn normalization.

A.7. Tensor network

Tensor network (Roberts et al., 2019) provides a way to represent/manipulate multi-dimensional arrays of data by factorizing into a network of lower-dimensional tensors. Many tensor decomposition methods are used for tensor networks, including matrix product state (MPS) and tree tensor network (TTN) (Huggins et al., 2019), based on tensor train (TT) decomposition and Hierarchical Tucker decomposition (HT), respectively. More sophisticated ones used in QML include multi-scale entanglement renormalization ansatz (MERA) (Vidal, 2008) and projected entangled-pair states (PEPS) (Orús, 2014). Tensorization provides efficient parameterization of DNN architecture (Novikov et al., 2015).

B. Einstein summation

The einsum is a powerful tool to represent diverse set of tensor networks. Several examples including TT/TR/CP/T-D/HT/PEPS as well as LoHa/LoKr (Yeh et al., 2024) are listed in the pseudo code below:

```
#TT: A(di,dp), B(dp,dj,dq), C(dq,dk,dr), D(dr,dl)
X = torch.einsum("ip,pjq,qkr,rl->ijkl", A, B, C, D)

#TR: A(dm,di,dp), B(dp,dj,dq), C(dq,dk,dr), D(dr,dl,dm)
X = torch.einsum("mip,pjq,qkr,rlm->ijkl", A, B, C, D)

#CP: A(dr), B(dr,di), C(dr,dj), D(dr,dk)
X = torch.einsum("r,ri,rj,rk->ijk", A, B, C, D)

#TD: A(dp,dq,dr), B(dp,di), C(dq,dj), D(dr,dk)
X = torch.einsum("pqr,pi,qj,rk->ijk", A, B, C, D)

#HT: A(di,dp), B(dj,dq), C(dk,dr), D(dl,ds), E(dp,dq,dt), F(dr,ds,du), G(dt,du)
X = torch.einsum("ip,jq,kr,ls,pqt,rsu,tu->ijkl", A, B, C, D, E, F, G)
```

```
#PEPS: A(di,dr,dp), B(dj,dp,ds,dq), C(dk,dq,dt), D(dl,dr,du), E(dm,du,ds,dv), F(dl,dv,dt)
X = torch.einsum("irp,jpsq,kqt,lru,musv,lvt->ijklmn", A, B, C, D, E, F)

#LoHa: A(di,dp), B(dp,dj), C(di,dq), D(dq,dj)
X = torch.einsum("ip,pj,iq,qj->ij", A, B, C, D)

#LoKr: A(di,dj), B(dp,dr), C(dr,dq)
X = torch.einsum("ij,pr,rq->ipjq", A, B, C)

X = einops.rearrange("i p j q -> (i p) (j q)", X) # unfolding 4-mode to 2-mode
```

Here we use `einops`[1] for folding/unfolding. It gives an opportunity to optimize topologies for tensor networks by just designing the einsum equation (a string like `"ip,pj->ij"`) with properly shaped tensor cores. Even a Monarch mixer (Fu et al., 2023) can be represented by a series of einsum operations without specifying permutation. The Monarch mixer uses dilated mixing with gradually expanded respective field:

$$W = P_{m+1}\mathrm{diag}[W_m]\cdots\mathrm{diag}[W_2]P_2\mathrm{diag}[W_1]P_1\mathrm{diag}[W_0]P_0, \tag{2}$$

where P_k's are permutations using Cooley–Tukey like butterfly architecture, and $\mathrm{diag}[W_k]$ are block diagonal weights. It generalizes many structured matrices like Fourier transform, Walsh–Hadamard transform, etc., while achieving low complexity with high parameter efficiency. It can be realized by the einsum chain, in a surprisingly elegant fashion as follows:

```
# 3-stage dilated Monarch mixer
# Assume input activation X (..., d, d, d), Mixer W (3, d, d, d, d)
X = torch.einsum("...ij,...i->...j", W[0], X) # the first local mixing
X = torch.einsum("...iaj,...ia->...ja", W[1], X) # the second dilated mixing
X = torch.einsum("iabj,...iab->...jab", W[2], X) # the last dilated mixing
```

As given in the above example, using einsum, we can construct any tensor networks by specifying einsum equation and shapes. The pseudo code for a tensor network module using arbitrary einsum equation and shapes can be listed as follows:

```
import opt_einsum as oe

class TensorNet(torch.nn.Module):
    def __init__(self, shapes, equation, device=None):
        super().__init__()
        self.shapes = shapes
        self.equation = equation
        self.device = device
        self.cores = torch.nn.ParameterList(
            [
                torch.nn.Parameter(torch.empty(*shape, device=self.device))
                for shape in self.shapes
            ]
        )

        # contraction order optimization
        self.expression = oe.contract_expression(self.equation, *self.shapes)

        self.reset_parameters() # initialize cores

    def contract(self, *args, **kwargs):
        # reconstruc a tensor from cores with optimized contraction expression
        return self.expression(*self.cores, *args, **kwargs)

    def fit_als(self, target, **kwargs):
        # fit cores to a target tensor with ALS
        with torch.inference_mode():
            cores = als(self.equation, self.cores, target, **kwargs)
            for k, core in enumerate(cores):
                self.cores[k].copy_(core)
```

[1] https://einops.rocks/

```
    def fit_grad(self, target, **kwargs):
        # gradient optimization
        lr = kwargs.get("lr", 1e-3)
        opt = torch.optim.AdamW(self.cores, lr=lr)
        for k in range(kwargs.get("steps", 1000)):
            opt.zero_grad()
            output = self.contract()
            loss = torch.nn.functional.mse_loss(output, target)
            loss.backward()
            opt.step()
```

Here, we use `opt_einsum`[2] to optimize the contraction order. We use `tensorly`[3] to initialize cores, and we can fine-tune them with gradient optimization.

In addition to the parameter reduction and flexible expressivity of tensor networks, the computational cost can be also reduced by designing the contraction order. For example with the 4-mode TT for a site dimension of $d = 32$ with a rank of $R = 20$, a naïve contraction from left to right requires 3.4×10^{10} FLOPs, whereas an optimized contraction order reduces it to 4.4×10^{7}, nearly 3 orders of magnitude when using `opt_einsum`. The pseudo code to demonstrate it is as follows:

```
# tensor reconstruction can be faster with contraction optimization
shape = [(32,20),(20,32,20),(20,32,20),(20,32)]
_, info = opt_einsum.contract_path("ip,pjq,qkr,rl->ijkl", *shape, shapes=True, optimize="
    optimal")
```

Furthermore, when the weight is decomposed in such a tensor network, we often do not need to explicitly reconstruct the weight before multiplying with the input activation. For example, if the input activation is batched as a shape of $(64, 1024)$, the $(1024, 1024)$ weight reconstruction and the weight-input multiplication require 2.7×10^{12} FLOPs in total. Whereas, it can be significantly reduced to 6.9×10^{6} FLOPs by optimizing the contraction order, achieving more than 5 orders of magnitude speedup, which is even faster than the weight materialization alone. The pseudo code to show its benefit is as follows:

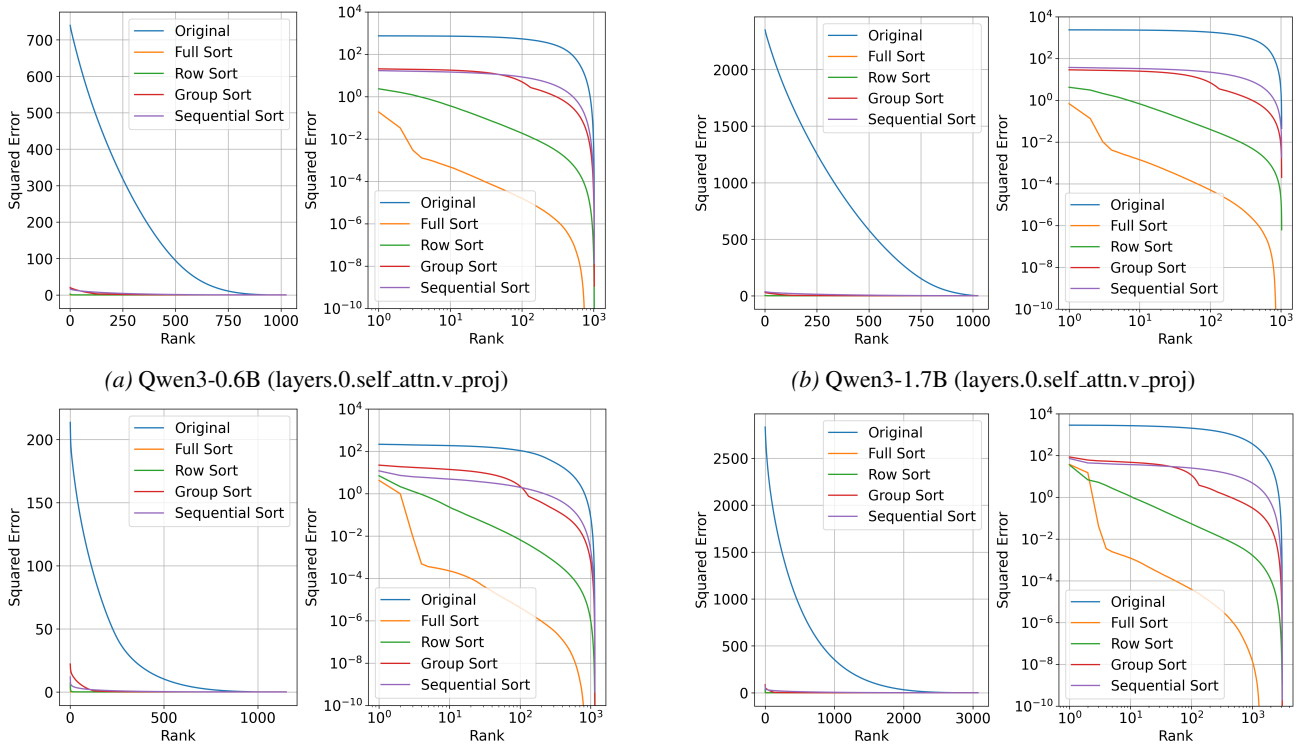
```
# weight-activation multiplication can be even faster with contraction optimization
# Weight: W (1024, 1024) -> (32, 32, 32, 32); Input X (64, 1024) -> (64, 32, 32)
shape = [(32,20),(20,32,20),(20,32,20),(20,32),(64,32,32)]
_, info = opt_einsum.contract_path("ip,pjq,qkr,rl,...ij->..kl", *shape, shapes=True,
    optimize="optimal")
```

C. Rank reduction with tensor sorting

We show more empirical results suggesting that the sorting operation can greatly reduce the required rank. Fig. 6 shows the tensorization loss in squared norm $\|W - \hat{W}\|^2$, where W is the original weights matrix and $\hat{W}$ is a reconstructed version through a 2-mode TT (i.e., SVD). Specifically, we analyze the eigen spectrum for pretrained weights of some LLM models, including Qwen3-0.6B, Qwen3-1.7B, Gemma3-4B, and Phi3.5-mini. We also evaluate several sorting options, as given in the pseudo code:

```
# Original eigen spectrum
S = torch.linalg.svdvals(W)
loss = S.square().flip(-1).cumsum(-1)

# Full sorting
Wf, _ = torch.sort(W.view(-1))
Wf = Wf.reshape_as(W)

# Row-wise sorting
Wr, _ = torch.sort(W, dim=-1)

# Row-wise group sorting
_, perm = torch.sort(W, dim=-1) # row-wise sort
```

[2] https://optimized-einsum.readthedocs.io/en/stable/
[3] https://tensorly.org/stable/index.html



(a) Qwen3-0.6B (layers.0.self_attn.v_proj)

(b) Qwen3-1.7B (layers.0.self_attn.v_proj)

(c) Gemma3-4B (layers.0.self_attn.v_proj)

(d) Phi3.5-mini (layers.0.self_attn.o_proj)

Figure 6. Tensorization error vs. rank for v_proj weights at the first layer of Qwen3-0.6B model. Sorting can significantly reduce the required rank. Left: linear; right: log-log plots.

```
perm, _ = torch.sort(perm.view(-1, groupsize), dim=-1) # revert permutation within group
Wg = torch.gather(W, dim=-1, index=perm.view_as(W))

# Row-wise sequential axis sorting
primes = sympy.factorint(W.shape[1]) # prime factorization
shape = [-1] + [p for p, e in primes.items() for _ in range(e)] # like [dout, 2,2,...,2]
Ws = W.view(*shape)
for axis in range(1, len(shape)):
    Ws, _ = torch.sort(Ws, dim=axis)
Ws = Ws.view_as(W)
```

The group sort means that we sort the row-wise elements but we do not care the ordering within each group of size g. For example, we sort 8 elements, but we do not care the ordering inside the first smallest 4 elements or the last smallest 4 elements. For this case, the required bits will be reduced from $\lceil \log_2(n!) \rceil / n$ to $\lceil \log_2(n!)/(\log_2(g!))^{n/g} \rceil / n$ bits per element. The sequential sorting does round-robin sorting across axis. For m-mode tensors with site dimension d, the required memory is $\lceil \log_2(d!) \rceil m/d$ bits.

As shown in Fig. 2, increasing the rank improves the accuracy, while the improvement is slow when original weights are decomposed by SVD without sorting. Notably, tensor sorting dramatically reduces reconstruction error, enabling near rank-1 reconstruction. From log-log plots, we can see that row-wise sorting, group sorting (with a groupsize of $g = 64$) and sequential axis sorting have relatively higher errors than full sorting, whereas they are still more than 10 times lower than the one without sorting. Note that the full sorting for Qwen3-0.6B requires $\lceil \log_2(2^{20}!) \rceil / 2^{20} \simeq 18.6$ bits per weight, the row sorting requires $\lceil \log_2(2^{10}!) \rceil / 2^{10} \simeq 8.6$ bits, the group sorting requires $\lceil \log_2(2^{10}!/(g!)^{2^{10}/g}) \rceil / 2^{10} \simeq 3.9$ bits, and the sequential sorting needs 5 bits per weight. Therefore, even with such an imperfect sorting to reduce the permutation memory, the index ordering has a great potential to improve the accuracy. Also we observe that the trend is similar to all LLM models we consider.

D. Theoretical analysis

D.1. Proof of Lemma 2.1

Proof. For $k \in \{0, \ldots, L-1\}$, the order statistic $X_{(k)}$ has the same distribution as

$$a + (b-a)Y_{(k)}, \tag{3}$$

where $Y_{(k)}$ is the $(k+1)$-st order statistic of L i.i.d. uniform random variables on $[0,1]$. It is well known (David & Nagaraja, 2004) that $Y_{(k)}$ follows the Beta distribution:

$$Y_{(k)} \sim \mathrm{Beta}(k+1, L-k). \tag{4}$$

Hence, we have

$$\mathbb{E}[X_{(k)}] = a + (b-a)\mathbb{E}[Y_{(k)}] = a + (b-a)\frac{k+1}{L+1}, \tag{5}$$

and

$$\mathrm{Var}[X_{(k)}] = (b-a)^2 \frac{(k+1)(L-k)}{(L+1)^2(L+2)}. \tag{6}$$

Since

$$(k+1)(L-k) \leq \frac{(L+1)^2}{4}, \tag{7}$$

we have

$$\mathrm{Var}[X_{(k)}] \leq \frac{(b-a)^2}{4(L+2)} = \mathcal{O}[L^{-1}]. \tag{8}$$

Therefore, by Chebyshev's inequality,

$$X_{(k)} = a + (b-a)\frac{k+1}{L+1} + \mathcal{O}_p[L^{-1/2}]. \tag{9}$$

Now setting $k = Ni + j$, we can write the matrix entry at (i,j) as

$$X_{i,j} = a + (b-a)\frac{(Ni+j)+1}{L+1} + \mathcal{O}_p[L^{-1/2}] \tag{10}$$

$$= N\frac{b-a}{L+1}i + \frac{b-a}{L+1}j + \left(a + \frac{b-a}{L+1}\right) + \mathcal{O}_p[L^{-1/2}]. \tag{11}$$

Thus, in matrix form, we can write

$$X = N\frac{b-a}{L+1}\ell_M 1_N^\top + \frac{b-a}{L+1}1_M \ell_N^\top + \left(a + \frac{b-a}{L+1}\right)1_M 1_N^\top + \mathcal{E}, \tag{12}$$

where every entry of $\mathcal{E}$ is $\mathcal{O}_p[L^{-1/2}]$.

The first three terms are rank-one matrices. Hence their sum has rank at most 3. Therefore X is asymptotically approximated by a rank-at-most-3 matrix, with entrywise stochastic error $\mathcal{O}_p[L^{-1/2}]$. □

Note that the bias factor in the third term is gone when $a = -b/L$ as $a + (b-a)/(L+1) = 0$.

D.2. Random projection to make uniformity

The preceding lemma can be extended heuristically beyond the uniform case. Indeed, many random vectors become approximately Gaussian after an orthogonal mixing transform due to the central limit effect. Since a Gaussian random variable can be converted into a uniform random variable through its cumulative distribution function, the sorted-matrix structure above approximately applies to a broader class of distributions after an appropriate unitary projection.

More specifically, let

$$x = [x_0, \dots, x_{L-1}]^\top \tag{13}$$

be a random vector with independent entries having finite variance, not necessarily uniformly distributed. Let $U \in \mathbb{R}^{L\times L}$ be a fixed unitary (orthogonal) matrix whose entries are sufficiently delocalized, such as the normalized Walsh–Hadamard matrix or discrete Fourier transform (DFT) matrix, and define

$$y = Ux. \tag{14}$$

Each component

$$y_i = \sum_{k=0}^{L-1} U_{ik} x_k \tag{15}$$

is then a weighted sum of many independent random variables. Under standard Lindeberg-type conditions, the central limit theorem implies that, as $L \to \infty$, each y_i converges in distribution toward a Gaussian random variable:

$$y_i \xrightarrow{d} \mathcal{N}(\mu_i, \sigma_i^2). \tag{16}$$

After normalization,

$$z_i = \Phi\left(\frac{y_i - \mu_i}{\sigma_i}\right), \tag{17}$$

where Φ denotes the standard Gaussian cumulative distribution function, we obtain approximately uniform random variables

$$z_i \approx \mathcal{U}(0, 1). \tag{18}$$

Consequently, after a sufficiently mixing unitary transform and marginal Gaussianization, a broad class of random matrices can be reduced approximately to the uniform setting analyzed in Lemma 2.1. Therefore, the asymptotic low-rank structure of sorted and folded matrices is expected to hold far beyond the exact uniform model.

This argument is heuristic because the transformed variables are generally not exactly independent, and finite-dimensional convergence from the central limit theorem does not directly imply joint uniformity. Nevertheless, for highly mixing transforms such as Walsh–Hadamard matrices, the approximation becomes accurate in high dimensions and is widely exploited in randomized numerical linear algebra and signal processing.

D.3. Row sorting

Even without assuming uniform distribution or random projection, we can derive the following theory for unknown random parameters for row-wise sorting.

Lemma D.1 (Asymptotic rank-one structure of row-wise sorted matrices)**.** *Let $X_{i,j}$ for $i \in \mathbb{Z}_M$ and $j \in \mathbb{Z}_N$ be i.i.d. random variables with continuous distribution function F. Assume: F has density f; there exists $c > 0$ such that $f(x) \geq c$ on the support of F; and the quantile function $Q(u) = F^{-1}(u)$ for $u \in (0, 1)$ satisfies as*

$$\int_0^1 Q(u)^2 \, \mathrm{d}u < \infty. \tag{19}$$

For each row i, let $X_{i,(1)} \leq \cdots \leq X_{i,(N)}$ denote the order statistics of $(X_{i,0}, \dots, X_{i,N-1})$, and define the row-wise sorted matrix $S \in \mathbb{R}^{M\times N}$ with $S_{i,j} = X_{i,(j+1)}$. Define the deterministic vector $q = (q_0, \dots, q_{N-1})^\top$ with $q_j = Q((j + 1)/(N + 1))$.

Then, the row-wise sorted matrix S is asymptotically rank one as $1_M q^\top$ for $N \gg 1$, i.e.,

$$\frac{\|S - 1_M q^\top\|_{\mathrm{F}}}{\|1_M q^\top\|_{\mathrm{F}}} = O_p(N^{-1/2}). \tag{20}$$

Proof. Define

$$U_{i,j} = F(X_{i,j}). \tag{21}$$

By the probability integral transform, the variables $U_{i,j}$ are *i.i.d.* uniform on $[0, 1]$. Since $Q = F^{-1}$,

$$X_{i,(j+1)} = Q(U_{i,(j+1)}), \tag{22}$$

where $U_{i,(j+1)}$ is the $(j + 1)$-st order statistic of N *i.i.d.* uniform random variables.

The uniform order statistics satisfy

$$\mathbb{E}[U_{i,(j+1)}] = \frac{j+1}{N+1}, \tag{23}$$

$$\mathrm{Var}(U_{i,(j+1)}) = \frac{(j+1)(N-j)}{(N+1)^2(N+2)} \leq \frac{1}{4(N+2)} = O(N^{-1}). \tag{24}$$

Since $f(x) \geq c > 0$, the quantile function $Q(u)$ is Lipschitz. Indeed,

$$Q'(u) = \frac{1}{f(Q(u))}, \tag{25}$$

hence

$$|Q'(u)| \leq \frac{1}{c}. \tag{26}$$

Therefore, for some constant $C > 0$,

$$|Q(u) - Q(v)| \leq C|u - v|. \tag{27}$$

Using this Lipschitz property,

$$\begin{aligned} \mathbb{E}\left[\left(X_{i,(j+1)} - q_j\right)^2\right] &= \mathbb{E}\left[\left(Q(U_{i,(j+1)}) - Q\left(\frac{j+1}{N+1}\right)\right)^2\right] \\ &\leq C^2 \,\mathrm{Var}(U_{i,(j+1)}) \\ &= O(N^{-1}). \end{aligned} \tag{28}$$

Summing over j,

$$\sum_{j=0}^{N-1} \mathbb{E}\left[\left(X_{i,(j+1)} - q_j\right)^2\right] = O(1).$$

Now define the error matrix

$$\mathcal{E}_{i,j} = X_{i,(j+1)} - q_j. \tag{29}$$

Then

$$\mathbb{E}\|\mathcal{E}\|_{\mathrm{F}}^2 = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} \mathbb{E}[\mathcal{E}_{i,j}^2] \tag{30}$$

$$= M \cdot O(1) \tag{31}$$

$$= O(M). \tag{32}$$

Hence,

$$\|\mathcal{E}\|_{\mathrm{F}} = O_p(\sqrt{M}). \tag{33}$$

Next,

$$\|1_M q^T\|_{\mathrm{F}}^2 = M\|q\|_2^2 = M\sum_{j=0}^{N-1} q_j^2. \tag{34}$$

Since $q_j = Q((j+1)/(N+1))$,

$$\frac{1}{N}\sum_{j=0}^{N-1} q_j^2 \to \int_0^1 Q(u)^2\,\mathrm{d}u. \tag{35}$$

Therefore,

$$\sum_{j=0}^{N-1} q_j^2 = \Theta(N), \tag{36}$$

which implies

$$\|1_M q^\top\|_{\mathrm{F}} = \Theta(\sqrt{MN}). \tag{37}$$

Finally,

$$\frac{\|\mathcal{E}\|_{\mathrm{F}}}{\|1_M q^\top\|_{\mathrm{F}}} = \frac{O_p(\sqrt{M})}{\Theta(\sqrt{MN})} = O_p(N^{-1/2}). \tag{38}$$

Since

$$S = 1_M q^\top + \mathcal{E}, \tag{39}$$

the matrix S converges to the rank-one matrix $1_M q^\top$ in relative Frobenius norm. □

D.4. Independent permutation

The row sorting may give a rank-one structure as in Lemma D.1. However, any independent permutation per tensor axis has no benefit.

Proposition D.2 (Invariance of singular values and rank under independent permutations)**.** *Let $X \in \mathbb{R}^{M\times N}$ be a matrix, and let $P_{\mathrm{r}} \in \mathbb{R}^{M\times M}$ and $P_{\mathrm{c}} \in \mathbb{R}^{N\times N}$ be arbitrary row and column permutation matrices, respectively. Then,*

$$\sigma(X) = \sigma(P_{\mathrm{r}} X P_{\mathrm{c}}), \tag{40}$$

where $\sigma(\cdot)$ denotes the set of singular values. Consequently,

$$\mathrm{rank}(X) = \mathrm{rank}(P_{\mathrm{r}} X P_{\mathrm{c}}). \tag{41}$$

Proof. Let the SVD of X be

$$X = U\Sigma V, \tag{42}$$

where $U \in \mathbb{R}^{M\times M}$ and $V \in \mathbb{R}^{N\times N}$ are orthogonal matrices, and $\Sigma \in \mathbb{R}^{M\times N}$ is diagonal rectangular with the singular values of X on its diagonal.

Since permutation matrices are orthogonal, we have

$$P_{\mathrm{r}}^{\top} P_{\mathrm{r}} = I_M, \qquad P_{\mathrm{c}} P_{\mathrm{c}}^{\top} = I_N. \tag{43}$$

Therefore, $P_{\mathrm{r}}U$ and VP_{c} are also orthogonal, because

$$(P_{\mathrm{r}}U)^{\top}(P_{\mathrm{r}}U) = U^{\top}P_{\mathrm{r}}^{\top}P_{\mathrm{r}}U = U^{\top}U = I_M, \tag{44}$$

$$(VP_{\mathrm{c}})(VP_{\mathrm{c}}^{\top} = VP_{\mathrm{c}}P_{\mathrm{c}}^{\top}V^{\top} = VV^{\top} = I_N. \tag{45}$$

Now,

$$P_{\mathrm{r}}XP_{\mathrm{c}} = P_{\mathrm{r}}U\Sigma VP_{\mathrm{c}} \tag{46}$$

$$= (P_{\mathrm{r}}U)\Sigma(VP_{\mathrm{c}}). \tag{47}$$

This is an SVD of $P_{\mathrm{r}}XP_{\mathrm{c}}$ with the same diagonal matrix Σ. Hence, $P_{\mathrm{r}}XP_{\mathrm{c}}$ and X have the same singular values.

Since the rank of a matrix is equal to the number of nonzero singular values, the rank is also invariant under independent row and column permutations. □

Comparing Lemma D.1, this trivial theory suggests that we should use axis-dependent permutation like CNOT entanglement, i.e., the permutation π for the column index j should be dependent on the row index i. From this perspective, TQCompress (Abronin et al., 2024) should have no benefit in the sense of singular values and tensor ranks as it uses row and column independent permutations.

D.5. Reduction impact

The sliced sorting with reduction can reduce the memory requirement. However, in return it can degrade the sorting accuracy. Under some assumptions, we can provide the potential impact when increasing the reduction size.

Proposition D.3 (Scaling law for mean and product reduction on uniform random variables)**.** *Let $X_{i,j}$ be uniform i.i.d. random variables uniformly distributed on $[-1, 1]$ for $i \in \mathbb{Z}_M$ and $j \in \mathbb{Z}_N$. Let the mean and product reductions after power scaling across column j be defined as $Y_i = \sum_{j=1}^{M} |X_{i,j}|^p/M$ and $Z_i = \prod_{j=1}^{M} |X_{i,j}|^p$, respectively, where $p > 0$ is power exponent. Then, the reduced random variables Y_i and Z_i conditioned on $X_{i,j}$ have moments:*

$$\mathbb{E}[Y_i \mid X_{i,j}] = \frac{X_{i,j}}{M} + \frac{M-1}{M(p+1)}, \tag{48}$$

$$\mathrm{Var}[Y_i \mid X_{i,j}] = \frac{M-1}{M^2}\Big(\frac{1}{2p+1} - \frac{1}{(p+1)^2}\Big), \tag{49}$$

$$\mathbb{E}[Z_i \mid X_{i,j}] = \frac{X_{i,j}}{(p+1)^{M-1}}, \tag{50}$$

$$\mathrm{Var}[Z_i \mid X_{i,j}] = X_{i,j}^2\Big(\frac{1}{(2p+1)^{M-1}} - \frac{1}{(p+1)^{2(M-1)}}\Big). \tag{51}$$

Hence, the error margin to detect $X_{i,j}$, i.e., target factor in mean over standard deviation, will decay as follows:

$$\frac{\mathbb{E}[Y_i \mid X_{i,j}] - (M-1)/M(p+1)}{\mathrm{Var}[Y_i \mid X_{i,j}]^{1/2}} = \frac{X_{i,j}}{\sqrt{M-1}}\Big(\frac{1}{2p+1} - \frac{1}{(p+1)^2}\Big)^{-1/2} \underset{M\gg 1}{\longrightarrow} X_{i,j}\mathcal{O}[1/M^{1/2}], \tag{52}$$

$$\frac{\mathbb{E}[Z_i \mid X_{i,j}]}{\mathrm{Var}[Z_i \mid X_{i,j}]^{1/2}} = \Big(\frac{(p+1)^{2(M-1)}}{(2p+1)^{M-1}} - 1\Big)^{-1/2} \underset{M\gg 1}{\longrightarrow} \mathcal{O}[((2p+1)/(p+1)^2)^{M/2}]. \tag{53}$$

Hence, the mean reduction decays the error margin slower than the product reduction when increasing the reduction size M.

Proof. Let X is the p-th absolute power of uniform random variable $U \sim \mathcal{U}(-1, 1)$, we have the CDF

$$F_X(x) = \mathbb{P}(|U|^p < x) = x^{1/p}, \qquad 0 \le x \le 1. \tag{54}$$

Its PDF is then

$$f_X(x) = \frac{1}{p} x^{1/p-1}, \tag{55}$$

which is Beta distribution $\mathrm{Beta}(1/p, 0)$. The moments are hence given as

$$\mathbb{E}[X^k] = \mathbb{E}[|U|^{kp}] = \frac{1}{pk+1}. \tag{56}$$

Let Y be the mean reduction of M *i.i.d.* Beta-distributed variables $X_1, \ldots, X_M$ conditioned on X_1, defined as

$$Y \mid X_1 = \frac{1}{M}(X_1 + \sum_{i=2}^{M} X_i). \tag{57}$$

Its mean is thus given as

$$\mathbb{E}[Y \mid X_1] = \frac{X_1}{M} + \frac{M-1}{M}\mathbb{E}[|U|^p] \tag{58}$$

$$= \frac{X_1}{M} + \frac{M-1}{M(p+1)}. \tag{59}$$

And, its variance is given as

$$\mathrm{Var}[Y|X_1] = \frac{M-1}{M}\mathrm{Var}[|U|^p] \tag{60}$$

$$= \frac{M-1}{M}\Big(\frac{1}{2p+1} - \Big(\frac{1}{p+1}\Big)^2\Big). \tag{61}$$

Let Z be the product reduction of $X_1, \ldots, X_M$ conditioned on X_1, defined as

$$Z \mid X_1 = X_1 \prod_{i=2}^{M} X_i. \tag{62}$$

Then, its mean is given as

$$\mathbb{E}[Z \mid X_1] = X_1\big(\mathbb{E}[|U|^p]\big)^{M-1} \tag{63}$$

$$= X_1\Big(\frac{1}{p+1}\Big)^{M-1}. \tag{64}$$

Also the second moment is

$$\mathbb{E}[Z^2 \mid X_1] = X^2\big(\mathbb{E}[|U|^{2p}]\big)^{M-1} \tag{65}$$

$$= X^2\Big(\frac{1}{2p+1}\Big)^{M-1} \tag{66}$$

Hence, the variance is given as

$$\mathrm{Var}[Z \mid X_1] = \mathrm{E}[Z^2 \mid X_1] - \big(\mathbb{E}[Z \mid X_1]\big)^2 \tag{67}$$

$$= X_1^2\Big(\Big(\frac{1}{2p+1}\Big)^{M-1} - \Big(\frac{1}{p+1}\Big)^{M-1}\Big). \tag{68}$$

□

For Gaussian random variables, it appears the similar scaling law. Note that the mean reduction has a target-dependent margin increasing with $X_{i,j}$, and thus the impact of the stochastic error is more severe at smaller value of $X_{i,j}$. Whereas, the product reduction has a constant margin across $X_{i,j}$, which may lead more reliable sliced sorting. For example, when we have $p = 1$ and $M = 8$, the error margin for the value $X_{i,j} = 0.1$ will be about 0.13 for the mean reduction, while it will be 0.41 for the product reduction.

E. EinSort

Those results motivate us to consider sorting for tensor network designs. We then propose **EinSort** framework based on the Einstein sorted sum. Conceptually, we use a reversible permutation operation $\pi[\cdot]$ inside tensor decomposition. Let $\mathcal{T}_\theta[\cdot]$ denote the tensor decomposition with hyperparameters θ which determine einsum equation, shapes, topology, etc. The sorted tensor decomposition is written as $\mathcal{T}_\theta^\pi := \pi^{-1}[\mathcal{T}_\theta[\pi[X]]]$. It generalizes the einsum to find low-rank structure.

Although sorting may reduce the number of parameters for tensor cores with few rank, the memory overhead of storing permutations is not negligible. Specifically, sorting L values requires at most $\lceil \log_2(L!) \rceil$ bits using factoradic/Lehmer code (Lehmer, 1960). For example, full sorting for a tensor of shape $(1024, 1024)$ requires roughly 19 bits per entry, which often exceeds the memory footprint of the original tensors in FP16. Therefore, we need to seek optimizing a tradeoff between sorting accuracy and tensor rank reduction.

We consider several simple sorting methods in this paper, while many other options could be explored. One of simplest approaches is to reuse a shared permutation for multiple tensors. To do so, we first need to determine what kind of sorting metrics is relevant for multi-tensor permutation. Considering a 4-mode tensor X of shape (d, d, d, d), we may use a shared permutation across the first mode slice: $X_{:,j,k,l}$ to re-order the last three modes for $\{j, k, l\}$. Then, we can reduce the memory of permutation from $\lceil \log_2(d^4!) \rceil$ to $\lceil \log_2(d^3!) \rceil$ bits, i.e., more than d-fold memory reduction. Therefore, adjusting the slicing dimension, we can easily reduce the total memory footprint, towards even lower than 1-bit per weight. For example, with $d = 32$, we have $\lceil \log_2(d^3!) \rceil / d^4 \simeq 0.4$ bits per weight. For sorting metric, we consider some options for nonlinear mapping and reduction, including power scaling, and mean/max/std/var/median/prod operations. An example using power scaling and std reduction is listed in the pseudo code as below.

```
# X (1024, 1024)
X = X.reshape([2] * 20) # folding to 20-mode tensor of shape (2,2,...,2)
S = X.abs().pow(p) # scoring with p-th power mapping
axis = [0,1,2] # slicing modes (the first 3 modes for simplicity)
S = S.std(dim=axis, keepdim=True).expand_as(X) # std reduction
perm = torch.argsort(S.view(2,2,2,-1), dim=-1) # shared permutation
X = X.view(2,2,2,-1).gather(-1, perm).view([2] * 20) # re-order
```

It has a flexibility to adjust the choices of slicing modes, power exponent, and reduction operation. We specifically consider six reduction operations:

- Mean reduction: `torch.mean()`
- Max reduction: `torch.max()`
- Min reduction: `torch.min()`
- Median reduction: `torch.median()`
- Standard deviation reduction `torch.std()`
- Product reduction: `torch.prod()`

In addition, we consider several nonlinear mapping before reduction:

- Linear: no mapping
- Power: `x.pow(p)`
- Exponential: `x.mul(p).clamp(-10,10).exp()`
- Additive logarithm: `x.add(p).clamp(4.4e-5,2.2e4).log()`
- Multiplicative logarithm: `x.mul(p).clamp(4.4e-5,2.2e4).log()`

We search for effective folding strategy and mode to increase the sorting accuracy, while keeping lower memory cost. Appendix F compares different mapping and reductions.

In addition to the sorting metric, we also introduce an approach to improve the accuracy by employing non-negative tensor decomposition. Specifically, we also project the original tensor into non-negative values before tensor decomposition. To recover the negative values, we keep the sign information of the original tensor, and hence it requires only one additional bit. It is illustrated in Fig. 3.

The pseudo code with non-negative tensorization is given as follows.

```
# X (1024, 1024)
X = X.reshape([2] * 20) # folding to 20-mode tensor of shape (2,2,...,2)
S = X.abs().pow(p) # scoring with p-th power mapping

if nn_flag: # non-negative tensorization flag
    X, B = S, X.sign() # X is now non-negative part, and B keeps its sign information

axis = [0,1,2] # slicing modes (the first 3 modes for simplicity)
S = S.std(dim=axis, keepdim=True).expand_as(X) # std reduction
perm = torch.argsort(S.view(2,2,2,-1), dim=-1) # shared permutation
X = X.view(2,2,2,-1).gather(-1, perm).view([2] * 20) # re-order
```

The sorted tensor now has non-negative elements for tensorization. After tensor reconstruction, the sign information is used to recover the original elements. Specifically, the reconstruction process is written as follows:

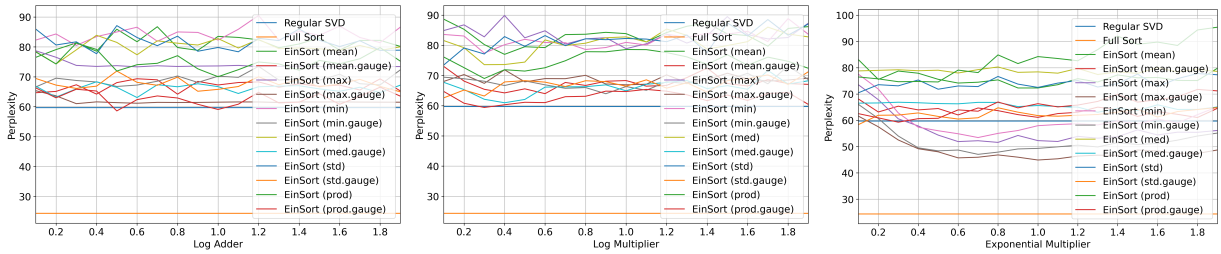
```
# tensor decomposition for sorted X(2,2,...,2)
factors = tensor_decomp(X, gauge=True) # gauge fixing to save memory

# tensor reconstruction: X is approx sorted non-negative tensor
X = tensor_reconst(factors, gauge=True)

# inverse permutation
perm_inv = torch.argsort(perm, dim=-1)
X = X.view(2,2,2,-1).gather(-1, perm_inv).view([2] * 20)

if nn_flag:
    X = X.clamp(0).pow(1/p) # approx. inverse nonlinear mapping
    X = X * B # inverse sign

# unfolding back to original tensor shape
X = X.reshape(1024, 1024)
```

Because the permutation is reversible, no information is discarded prior to tensor decomposition. Note that certain reversible permutations used in EinSort can be interpreted as classical analogues of entangling operations in tensor-network-based quantum circuits as discussed in Appendix L.

F. Nonlinear mapping and reduction

Besides power scaling in Fig. 5, we compare with different nonlinear mapping in Fig. 7. We observe that none of them achieves good performance except for some reductions of exponential mapping. Nevertheless, power scaling offers more accurate sorting than exponential mapping. Table 4 lists the PPL score at the best parameter with/without gauge fixing. For most cases, the product reduction was best except for exponential scaling, which prefers the max reduction. Nevertheless, the max and mean reductions with power scaling offer the third and fourth best performance of 40.25 and 42.38, respectively. Appendix D gives an insight for the difference of reduction operations, where we showed that product reduction is insensitive to the target value, while the mean reduction can be worse in low-magnitude regimes. All cases, the gauge fixing improves the accuracy.

Fig. 8 shows the case for Phi3-mini model at 50% and 80% compression rates. The trend is different from Qwen3 case in Fig. 5. Specifically, best power exponent is around 0.4 when non-negative tensorization is used, whereas larger than 2 may be best otherwise. Also, all reductions are comparable, achieving the best score at the similar exponent. Nonetheless, the gauge fixing and non-negative tensorizing are effective to compete with full sorting performance. For Phi3 case, the required memory for shared permutation is 0.35 bits per weight.



(a) Additive Logarithm *(b)* Multiplicative Logarithm *(c)* Exponentiation

Figure 7. Different nonlinear mapping for Qwen3-1.7B at 50% compression: additive logarithm; multiplicative logarithm; and exponentiation.

Table 4. PPL with different nonlinear mapping and reduction for Qwen3-1.7B at 50% compression. The four best scores are highlighted. The regular SVD without sorting has 60.0 PPL.

Reduction	mean	max	min	median	std	prod
linear w/o gauge fixing	77.13	73.55	78.12	79.07	73.82	**59.56**
linear w/ gauge fixing	62.79	61.49	66.04	66.34	61.19	**50.84**
pow w/o gauge fixing	77.46	73.54	78.08	79.05	72.46	**59.43**
pow w/ gauge fixing	63.60	61.44	66.03	66.30	57.42	**50.56**
abs.pow w/o gauge fixing	63.75	75.25	81.11	65.47	69.91	**52.12**
abs.pow w/ gauge fixing	55.26	63.25	67.52	56.22	60.78	**44.72**
pow.nn w/o gauge fixing	50.08	46.56	68.21	55.66	51.37	**36.29**
pow.nn w/ gauge fixing	42.38	40.25	52.70	44.21	46.79	**33.17**
exp w/o gauge fixing	72.31	**51.60**	53.45	77.44	70.48	74.60
exp w/ gauge fixing	61.07	**44.90**	47.06	58.37	63.07	59.29
add.log w/o gauge fixing	76.52	73.30	80.50	77.72	74.31	**70.04**
add.log w/ gauge fixing	64.20	61.03	66.62	64.99	63.07	**58.61**
mul.log w/o gauge fixing	76.99	78.88	77.42	73.48	73.58	**69.01**
mul.log w/ gauge fixing	63.98	66.91	63.96	62.60	60.97	**59.41**

G. Gauge fixing

Here, we describe the gauge fixing we use. Consider a 4-mode TT for a tensor X decomposed by four cores A, B, C, and D as depicted in Fig. 9. As there are gauge freedom, there are no unique tensor cores to represent X. Specifically, we can inject any full-rank junction matrices together with its inverse between the tensor cuts: A–B, B–C, and C–D.

One of typical methods for canonicalization is based on SVD or QR decomposition to make all cores orthogonal except for one orthogonality center. For example, we employ an r-truncated SVD for a product of AB as

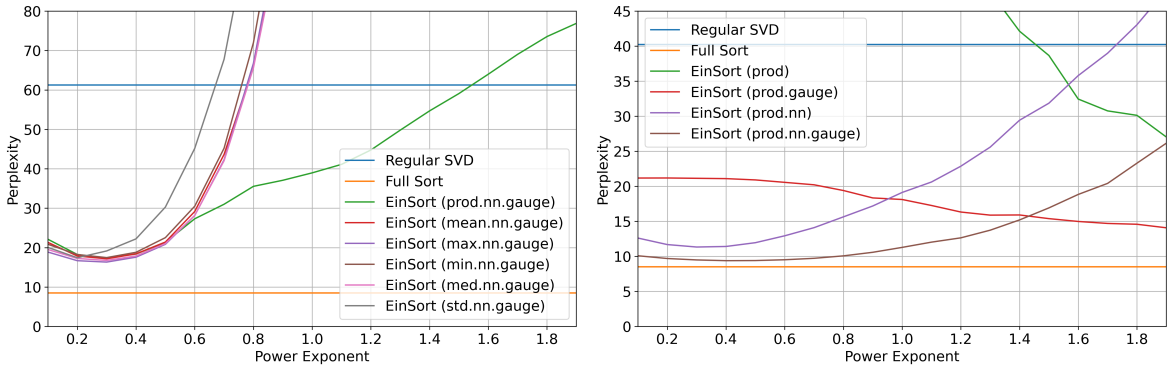
$$\mathsf{svd}_r[AB] = U_A S_B V_B, \tag{69}$$

where U_A and V_B are left and right singular vectors, S_B is diagonal singularvalues matrix, and r is the bond rank. We then assign as new tensor cores for A and B as follows:

$$A \leftarrow U_A, \tag{70}$$
$$B \leftarrow S_B V_B. \tag{71}$$

Then, we can repeat for a new product BC for $\mathsf{svd}_r[BC] = U_B S_C V_C$ to split as $U_B \to B$ and $S_B V_B \to C$. Similarly, the product $CD = U_C S_D V_D$ to split $U_C \to C$ and $V_D \to D$. The diagonal tensor S_D can be merged either into C or D, or left as a new tensor which plays as the orthogonality center. Doing so, all cores except the orthogonality center will be orthogonal matrices. This gauge fixing based on SVD canonicalization is illustrated in Fig. 10(a).

This canonicalization can fix the gauge freedom, and improve the numerical stability to solve the tensor decomposition. Besides the numerical stability, Koike-Akino et al. (2025b) discussed the potential to reduce the memory size for orthogonal



(a) 50% compression *(b)* 80% compression

Figure 8. Different reduction and nonlinear mapping for Phi3-mini at 50% and 80% compression.

matrices by parameterizing as the Stiefel manifold, either based on exponential map, Cayley transform, Householder reflections, Givens rotation, Taylor series, or Neumann series. The Stiefel manifold parameterization can reduce the required number of parameters from dr to $dr - r(r+1)/2$ for a tensor of shape $d \times r$ with $d \geq r$.

However, the regular canonicalization with Stiefel parameterization requires nonlinear mapping to construct orthogonal matrices. Koike-Akino et al. (2026a) proposed a simpler method to fix the gauge freedom by using block identity form. For example, decompose A into LU factorization form as follows:

$$A = P_A L_A U_A, \tag{72}$$

where $P_A \in \mathbb{R}^{d\times d}$ is an optional pivoting permutation, $L_A \in \mathbb{R}^{d\times r}$ is lower-triangular matrix and $U_A \in \mathbb{R}^{r\times r}$ is upper-triangular matrix. As permutation can be represented by index transform with $\lceil \log_2(d!) \rceil$ bits for factradic coding, it does not need $d \times d$ full matrix tensor. We split the lower triangular into upper square matrix and lower dense matrix:

$$L_A = \begin{bmatrix} \bar{L}_A \\ \underline{L}_A \end{bmatrix}, \qquad \bar{L}_A \in \mathbb{R}^{r\times r}, \qquad \underline{L}_A \in \mathbb{R}^{(d-r)\times r}. \tag{73}$$

Then L_A can be block identity as follows:

$$L_A = \underbrace{\begin{bmatrix} I_r \\ \underline{L}_A \bar{L}_A^{-1} \end{bmatrix}}_{L'_A \in \mathbb{R}^{d\times r}} \bar{L}_A. \tag{74}$$

Note that the permutation P_A is not necessary, while $\bar{L}_A$ can be singular without using a proper permutation. Now, we map the permuted block-identity as a new A and merge the rest to B as follows:

$$A \leftarrow P_A L'_A, \tag{75}$$
$$B \leftarrow \bar{L}_A U_A B. \tag{76}$$

Note that the block identity L'_A does not need to keep the upper identity part but only the lower part of $\underline{L}_A \bar{L}_A^{-1}$, and hence the number of parameters is reduced from dr to $dr - r^2$. Repeating the LU decomposition, the cores A, B, and C can become block identity. In consequence, the total number of parameters for TT will be reduced from $2dr(1+r)$ to $2dr(1+r) - 3r^2$. Fig. 10(b) shows LU-based gauge fixing.

The pseudo code is written as follows:

```
def gauge_fixing(A, B): # left core A and right core B with bond rank r
    d, r = A.shape # assuming d > r
    assert B.shape[0] == r # assuming B is in (r, ...)

    # LU decomposition: P @ L @ U = A
```

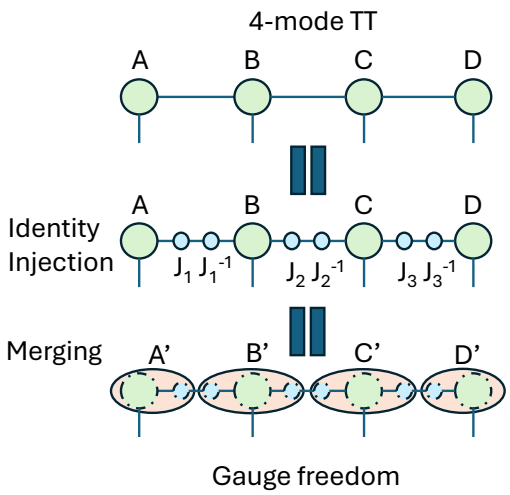


Figure 9. Gauge freedom: Tensor cores can be arbitrary up to any full-rank identity rotation injection at each cuts.

```
    P, L, U = torch.linalg.lu(A)

    # permutation index from matrix
    perm = P.argmax(-1)
    assert torch.allclose(A, (L @ U)[perm]) # A = (L @ U)[perm]

    # partition
    L0 = L[:r] # upper part, which is unitriangular
    L1 = L[r:] # lower part

    # block identity's lower part: A = L1 @ L0.inverse()
    A = torch.linalg.solve_triangular(L0, L1, upper=False, left=False, unitriangular=True)

    # New A tensor can be constructed from reduced-parameter A and perm if needed
    # A = torch.cat((torch.eye(r), A))[perm]

    # New B tensor merging: (L0 @ U) @ B
    B = torch.einsum("ij,j...->i..."), (L0 @ U), B)
    return A, perm, B
```

Alternative to LU decomposition, QR decomposition can be used. However, current implimentation of `torch.linalg.qr` can be numerically unstable if the first r columns are not independent, which requires a pivotting like the above LU decomposition.

H. Activation-aware tensorization

We note that EinSort can be integrated with activation-aware decomposition like ASVD (Yuan et al., 2023), AA-SVD (Sinha & Fleuret, 2026) and OBD-SVD (Li et al., 2026). The core idea of those methods is to use input and/or output preconditioning. ASVD uses the following objective function:

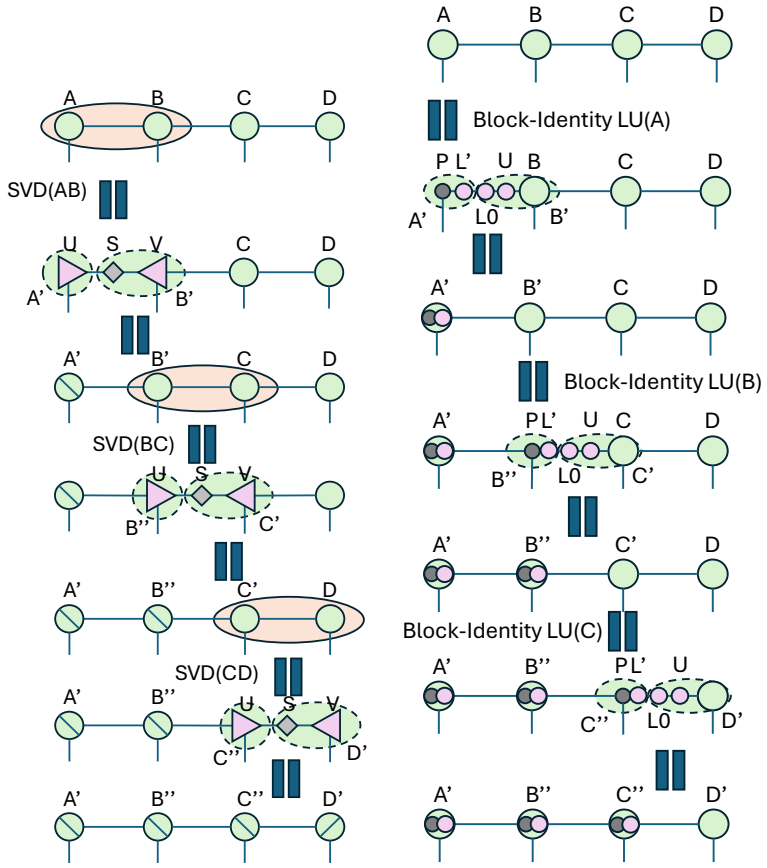
$$\hat{W} = \arg\min_{W'} \|WX - W'X\|^2 + \lambda\|W - W'\|^2, \tag{77}$$

where X is an input activation from calibration data and λ is a regularizer to consider activation-unaware loss. The solution is given as

$$\hat{W} = \mathsf{svd}_r[WC^{1/2}]C^{-1/2}, \tag{78}$$

where $C^{1/2}$ is a preconditioner, defines as $C = XX^\top + \lambda I$. AA-SVD extends it to deal with error propagation as follows:

$$\hat{W} = \arg\min_{W'} \alpha\|WX - W'X'\|^2 + \beta\|WX - W'X\|^2 + \lambda\|W - W'\|^2, \tag{79}$$



(a) SVD Canonicalization *(b)* LU Canonicalization

Figure 10. Gauge fixing with SVD and LU canonicalization. SVD: Tensor cores can be left- or right-unitary around the orthogonality center. Orthogonal tensors can be represented by lower number of parameters with Stiefel manifold (Koike-Akino et al., 2025b). LU: Tensor cores can be block identity. Block identity tensors can be represented with lower number of parameters (Koike-Akino et al., 2026a).

where X' is a modified input activation after decomposition of preceding layers, α and β are regularizers. The solution is given as

$$\hat{W} = \mathsf{svd}_r[WC_1C_2^{-1/2}]C_2^{-1/2}, \tag{80}$$

$$C_1 = \alpha XX'^\top + \beta XX^\top + \lambda I, \tag{81}$$

$$C_2 = \alpha X'X'^\top + \beta XX^\top + \lambda I. \tag{82}$$

The OBD-SVD extends ASVD to use output preconditioning to minimize:

$$\hat{W} = \arg\min_{W'} \|G^{1/2}(W - W')C^{1/2}\|^2, \tag{83}$$

where G represents the gradient information to approximate Hessian. The solution is given in the form:

$$\hat{W} = G^{-1/2}\mathsf{svd}_r[G^{1/2}WC^{1/2}]C^{-1/2}. \tag{84}$$

All cases including ASVD, AA-ASVD, and SVD-SVD can be expressed in the form with particular preconditioner G, C_A, and C_B:

$$\hat{W} = G^{-1/2}\mathsf{svd}_r[G^{1/2}WC_1C_2^{-1/2}]C_2^{-1/2}, \tag{85}$$

Naturally, sorted tensor decomposition can do the same way to deal with the activation and gradient statistics as follows:

$$\hat{W} = G^{-1/2}\mathcal{T}_\theta^\pi[G^{1/2}WC_1C_2^{-1/2}]C_2^{-1/2}. \tag{86}$$

I. Quantization-aware tensorization

The tensor cores can be also quantized or pruned to further save the memory for reduced-rank tensor networks. As einsum is differentiable, imposing quantization and pruning under gradient optimization is straightforward, e.g., based on straight-through estimation. Specifically, the tensor cores are quantized (or pruned) as $\hat{W} = \mathcal{Q}[W]$ where $\mathcal{Q}[\cdot]$ denotes the quantization operation (and/or pruning), which itself may be non differentiable. The quantized tensors are then pass through to the einsum contraction by superposing the non-quantized cores: $W' = \hat{W} + (W - W.\mathrm{detach})$ where $[\cdot].\mathrm{detach}$ denotes cutting the autograd path. The pseudo code is listed below.

```
def quantize_contract(self, target, *args, **kwargs):
    # quantize cores
    cores = self.quantize(self.cores, **kwargs)

    # straight-through estimator
    for k in range(len(cores)):
        cores[k] += self.cores[k] - self.cores[k].detach()

    # quantization-aware contraction to use gradient optimization
    return self.expression(*cores, *args, **kwargs)
```

J. Test-time adaptation

Tensor ranks can be optimized across layers and modules (Luo et al., 2024). However, most papers consider static rank selections, which do not adaptively change over the inference time. Inspired by test-time quantization (TTQ) (Koike-Akino et al., 2026b), we can consider test-time rank adaptation. TTQ provides a theoretical justification suggesting that test-time weight approximation adaptive to every input prompts can improve the accuracy, compared to offline weight static approximation.

Consider a simple toy example: the weights W are decomposed into 2-rank factors as $\hat{W} = a_1b_1^\top + a_2b_2^\top$ for vectors a_1, a_2, b_1, b_2, and the input token X is orthogonal to both b_1 and b_2. Then, $\hat{W}X = 0$, which implies that those two-rank factors are useless and we should increase the rank for such tokens. Whereas, if X is orthogonal to b_2 but b_1 like $X = b_1c^\top$, then we have $\hat{W}X = |b_1|^2a_1c^\top$, suggesting that the second factor is redundant to be omitted. This toy example gives a

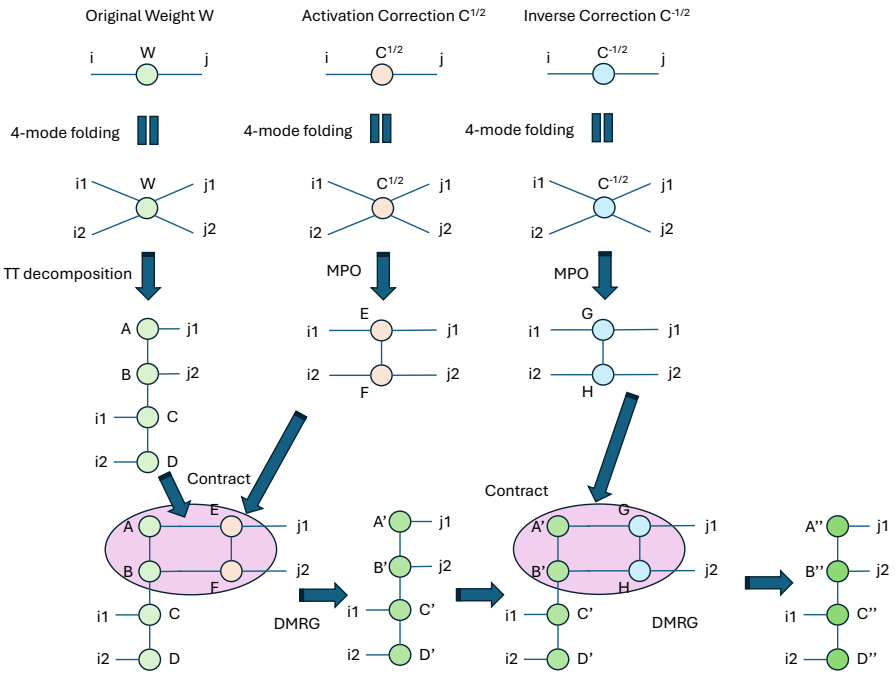


Figure 11. Test-time tensor train adaptation based on online activation preconditioner C.

key insight that the weight decomposition should be adaptive to the input tokens at inference time to be more efficient and effective.

We may use a simple scoring network to determine the bond ranks at test time. Specifically, a small neural network feeding online activation input X, and decide the ranks for tensor networks. The network can be trained through synthetic data or calibration data held out from test samples. This enables a test-time rank adaptation for tensor networks.

Another alternative is to feed an updated preconditioner at test time, and update the tensor cores in an online fashion. For example, W of shape (d^2, d^2) uses 4-mode TT of shape (d, d, d, d) with a moderately high rank of R, and the online token correlation $C = XX^\top + \lambda I_{d^2}$ is decomposed as another tensor network with 2-mode matrix product operator (MPO) of shape (d, d) with bond rank R' through density matrix renormalization group (DMRG) algorithm (Schollwöck, 2011). Then, we can locally and sequentially update the tensor cores of the 4-mode TT by contracting with MPO to find the dominant eigenspaces with limited bond ranks lower than R. Fig. 11 illustrates the test-time adaptation framework to update the TT depending on the online preconditioner C.

K. Activation compression

Besides compressing LLM weights or KV cache, we can also compress intermediate activation information, which are required for gradient updates. For example, consider a linear module taking a forward path: $Y = WX$. Then the weight update from gradient backpropagation requires:

$$W \leftarrow W - \eta \underbrace{\frac{\partial \mathcal{L}}{\partial Y}}_{G} X^\top, \tag{87}$$

where η is a learning rate, $\mathcal{L}$ is a loss function and G is gradient up to the output Y. This clearly shows the potential memory issue for LLM gradient updates, which require all intermediate activations X across layers. EinSort can reduce the memory for any tensors including such intermediate activations.

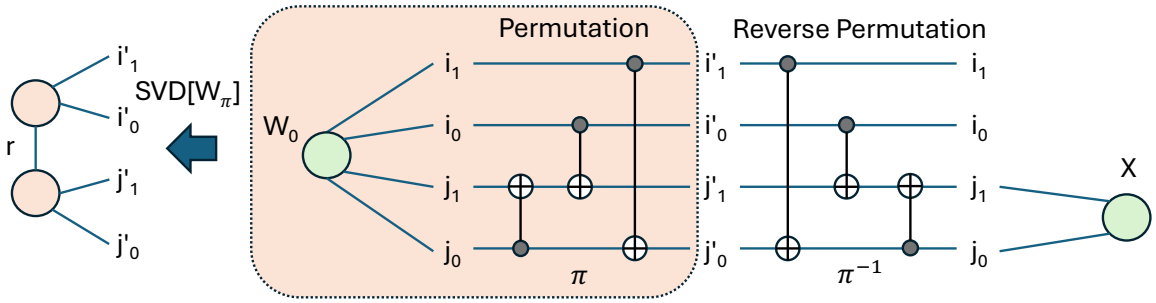


Figure 12. CNOT chain to realize the row-wise sorting π for W_0 in (89). The sorted matrix W_π can then be decomposed in low-rank tensors. The input X is contracted with the decomposed W_π through reverse permutation π^{-1} with conjugate CNOT chain.

L. Connection with quantum gates

Tensor index permutations in EinSort are closely related to entangling operations in quantum tensor networks. In particular, reversible Boolean permutations can be interpreted as classical analogues of quantum logic gates including controlled-NOT (CNOT) and Toffoli gates. For binary indices as in example of (1), the transformation $(i, j) \mapsto (i, i \oplus j)$ corresponds exactly to the action of a CNOT gate, where one index acts as a control bit and the other is conditionally flipped. Indeed, CNOT corresponds to a permutation matrix:

$$\mathsf{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{88}$$

As an another example, consider a 4×4 matrix W_0 to apply row-wise sorting:

$$W_0 = \begin{array}{c|c} & \xrightarrow{\quad j \quad} \\ i \downarrow & \begin{bmatrix} 4.0 & 1.0 & 2.0 & 3.0 \\ 2.1 & 3.1 & 4.1 & 1.1 \\ 1.2 & 3.2 & 4.2 & 2.2 \\ 3.3 & 2.3 & 1.3 & 4.3 \end{bmatrix} \end{array}, \qquad \pi = \begin{array}{c|c} & \xrightarrow{\quad j_1 \oplus j_0 \oplus i_0,\, j_0 \oplus i_1 \quad} \\ i \downarrow & \begin{bmatrix} 0 & 3 & 2 & 1 \\ 2 & 1 & 0 & 3 \\ 2 & 1 & 3 & 0 \\ 3 & 0 & 1 & 2 \end{bmatrix} \end{array}, \qquad W_\pi = \begin{bmatrix} 4.0 & 3.0 & 2.0 & 1.0 \\ 4.1 & 3.1 & 2.1 & 1.1 \\ 4.2 & 3.2 & 2.2 & 1.2 \\ 4.3 & 3.3 & 2.3 & 1.3 \end{bmatrix}, \tag{89}$$

where the row index $i \in \mathbb{Z}_4$ can be represented with two binary indices $i_1 \in \mathbb{Z}_2$ and $i_0 \in \mathbb{Z}_2$ as $i = 2i_1 + i_0$. Similarly the column index $j \in \mathbb{Z}_4$ has 2 binary indices as $j_1 \in \mathbb{Z}_2$ and $j_0 \in \mathbb{Z}_2$ such that $j = 2j_1 + j_0$. Then the row-wise permutation π can be expressed as: $j_1 \leftarrow j_1 + j_0 + i_0 \pmod 2$, $j_0 \leftarrow j_0 + i_1 \pmod 2$. The column index j is now dependent on row index i, which is the important factor to reduce the rank as discussed in Appendix D. This is entangling operator realized by CNOT chain as depicted in Fig. 12. The singular values for the unsorted matrix W_0 is about $[10.6, 4.1, 1.6, 0.6]$ and that for W_π is about $[11.5, 0.2, 0.0, 0.0]$, which is nearly a rank of 2. And, hence the sorted matrix can be well approximated by lower-rank tensor decomposition. Note that the CNOT chain for permutation is reversible, and we can use conjugate CNOT chain to contract with an activation X as in this figure.

However, CNOT plus NOT gates can only realize even affine permutation. More generally, arbitrary reversible permutation can be realized by CNOT plus Toffoli gates, e.g., $(i, j, k) \mapsto (i, j, k \oplus ij)$, where two control bits jointly determine the target transformation. From the tensor-network perspective, these reversible index transformations may convert highly entangled tensors into representations with lower effective bond dimensions, thereby exposing latent low-rank structure prior to tensor decomposition. This connection suggests broader links between tensor compression, reversible computation, and quantum-inspired representations, where index permutations play a role to basis transformations that simplify entanglement patterns.

Motivated by the interpretation of permutations as compositions of Toffoli-like reversible gates, we can further reduce the memory footprint required to represent sorting permutations. Specifically, after obtaining a target permutation vector, we convert the permutation into a sequence of Toffoli gate chains, which induces an equivalent Boolean function representation. To reduce the Toffoli gates, we can Using ancilla bits can further reduce the reversible circuits via Pebble game theory (Li

et al., 1998). The resulting Boolean function can then be compressed into a compact sum-of-products form using standard logic synthesis techniques such as the Quine–McCluskey algorithm (McCluskey, 1956) or Espresso (Brayton et al., 1984). Using PyEDA[4], we empirically confirmed that the number of permutation gates can be substantially reduced, particularly when introducing don't-care sets corresponding to low-score permutations that can be ignored with minimal impact on sorting quality. For example, the total number of gates was reduced from 1976 to 744 when allowing approximately 50% permutation cancellation through don't-care optimization. We also use RevKit[5] to reduce the reversible gates for permutations.

M. Algebraic polynomial permutation

CNOT gates work binary axis, while it can be extended to arbitrary dimension by modular additions like: $(i, j) \mapsto (i, j + i \mod d_j)$ for the second dimension of d_j. A related method to parameterize permutation is to use algebraic polynomial permutation. For example, quadratic polynomial permutation (QPP) (Takeshita, 2007) uses the following conversion:

$$\pi(i) = a + b\,i + c\,i^2 \mod n, \tag{90}$$

where $i \in \mathbb{Z}_n$ is an index of size n. The bijective conditions are $\gcd(b, n) = 1$ and $\mathrm{rad}(n) \mid c$ where $\gcd$ denotes the greatest common multiple and rad denotes the radical. We may optimize the coefficients of polynomials to improve the tensorization accuracy. However, polynomial over a finite ring does not cover all permutations unless n is a prime number. For arbitrary n, algebraic polynomial over a finite field such as Galois field can realize any reversible permutation instead. We may use Lagrange interpolation over Galois field as $\pi(x) = \sum_i y_i \prod_{j \neq i}(x - x_j)/(x_i - x_j)$ to pass through major sorted index conversion $x_i \mapsto y_i$. To design polynomial permutation on Galios field, we use `galios.laglange_poly`[6].

N. Learned permutation operations

In this paper, we focus on practical permutations based on sliced sorting with nonlinear mapping and reduction variety. Even though sorting has a strong theoretical justification as discussed in Appendix D, it is just one of heuristics we discovered. For more general cases, the permutations and nonlinear mappings can be learned to optimize them. For example, permutation operations can be optimized through gradient updates, e.g., using NeuralSort (Grover et al., 2019), SortSoft (Prillo & Eisenschlos, 2020), or Gumbel–Sinkhorn (Mena et al., 2018). Our objective could be formulated as follows:

$$\alpha \|W - \mathcal{T}_\theta^\pi[W]\|^2 + \lambda |[\theta, \pi]|, \tag{91}$$

where $|[\theta, \pi]|$ denotes the total memory cost for tensorization hyperparameters θ and permutation π. Note that the first term can be extended to deal with input/output preconditioners as discussed in Appendix K.

We can also include nonlinear mapping in θ not only for sorting but also tensorization. Specifically, we apply nonlinear mapping to W like exponentiation, and after decomposition, we use inverse nonlinear mapping to reconstruct. More specifically, we can write as

$$\mathcal{T}_\theta^\pi[W] = \phi^{-1}[\pi^{-1}[\mathcal{T}_\theta[\pi[\phi[W]]]]], \tag{92}$$

where $\phi[\cdot]$ denotes reversible nonlinear mappings such as exponentiation, and $\phi[\cdot]^{-1}$ is its inverse. Note that some function like absolute operation $|\cdot|$ can be reversible if we retain the sign information along with the absolute value, which we call non-negative tensorization. When some hyperparameters are not differentiable, we could employ reinforcement learning to design permutations and nonlinear mappings as well as tensor ranks, tensor shapes, and topologies.

O. Folding and ordering

Consider a mode-4 tensor X of shape $[d_1, d_2, d_3, d_4]$, e.g., the number of layers d_1, binary axis indicating key or value for d_2, number of heads for d_3, and the head dimension for d_4. There are a few steps to use different tensor ordering, folding, and unfolding. We can decompose it into many different ways. For example, we may use:

[4] https://github.com/cjdrake/pyeda
[5] https://github.com/msoeken/revkit
[6] https://github.com/mhostetter/galois

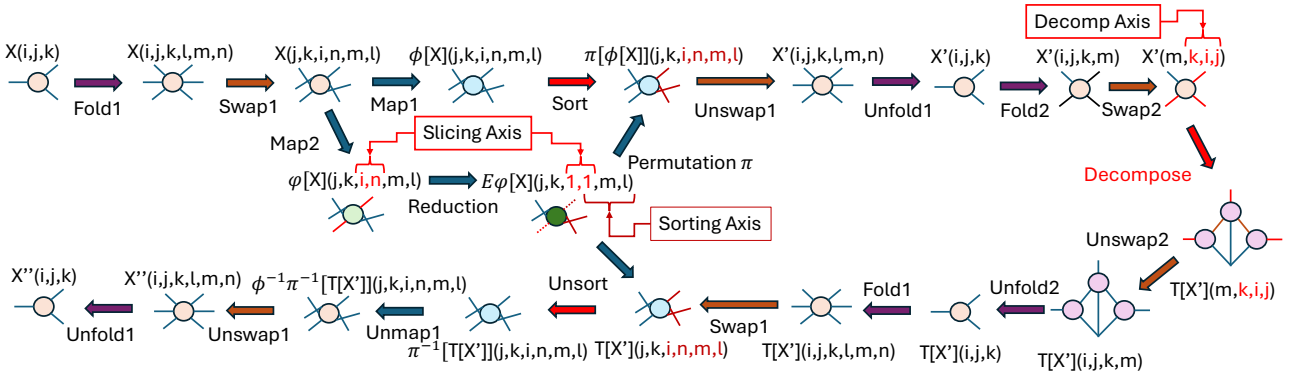


Figure 13. EinSort pipeline with generalized folding/unfolding, ordering, slicing, sorting, and mapping.

- 4-mode TT for X as is;
- 4-mode TT for axis-swapped version of X, like `X.permute(3,1,0,2)`;
- Use the second axis as a batch dimension, and apply 3-mode TT for the rest 3 axis;
- Unfold into $[d_1 \times d_2, d_3, d_4]$, and apply 3-mode TT: `X.reshape(-1, d3, d4)`;
- Fold X into 6-mode tensor like `X.reshape(a,b,c,d,e,f)`, and apply 5-mode TT for the last 5 axis.

As the tensor decomposition depends on the mode ordering, axis swapping and its folding shape can be adjusted so that the reconstruction is minimized. Similarly, before decomposition, we use sorting operations and we have flexibility to choose which axis to sort. Note that sorting axis and decomposition axis need not be identical or perfectly overlapped. In addition, we have additional degrees of freedom to choose the axis for slice reduction. Fig. 13 shows the EinSort framework using different folding/unfolding and axis ordering for sorting, reduction, and decomposition. We also use different nonlinear mappings for sorting ϕ and decomposition φ, in this figure.

In addition, the sorting axis can be sequential as discussed in Appendix C, e.g., axis $[3, 5]$ to sort first, and then $[2, 4]$ to sort later, rather than sorting all $[2, 3, 4, 5]$ axis. If the size of axis is all d, the whole sorting for the last 4 modes require $\lceil \log_2(d^4!) \rceil / d^4$ bits, whereas the sequential sorting requires $2\lceil \log_2(d^2!) \rceil / d^2$ bits. For example with $d = 32$, it will reduce from 18.6 to 17.1 bits. When doing 4 times in each axis sequentially, then it will be reduced to 14.7 bits.

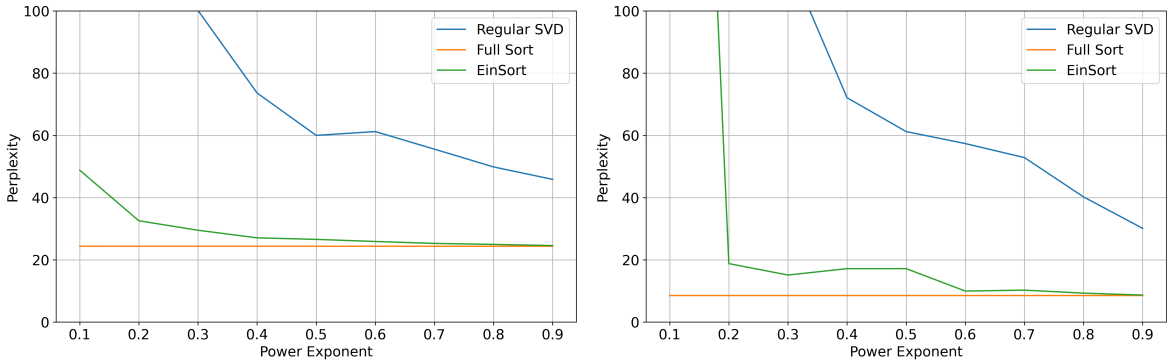
P. Benefits and limitations

EinSort has some benefits:

- There is a theoretical foundation suggesting that sorting operation has a potential benefit to reduce tensor ranks.
- Sorting operation has a good connection with quantum tensor networks, using entanglement operations.
- Einsum-based formulation gives a powerful tool to design almost arbitrary tensor network topology.
- EinSort provides a wide range of flexibility to adjust the hyperparameters, including slicing order, mode selection, folding combination, nonlinear mapping, and reduction operations.
- It can be used for any tensor compression: e.g., LLM weights decomposition; KV cache reduction; forward activation compression used for backward gradient calculation.

Nonetheless, it has some drawbacks and limitations.

- Finding best tensor topology, folding shapes, index ordering, etc. is not straightforward.



(a) Qwen3-1.7B model. *(b)* Phi3-mini (3.8B) model.

Figure 14. PPL over compression rate for Qwen3-1.7B and Phi3-mini (3.8B) models.

- Computing large tensor factorization is computationally challenging.
- Permutation memory is not easy to reduce.
- Compatible high-performance CUDA kernels which can optimize contraction and index reordering are not available.

Q. Experiments setup

We conduct experiments for LLM benchmarks to evaluate the effectiveness of our method. Our experiments are based on the same setting of SparseLLM (Bai et al., 2024b) and their code base[7]. Following existing work (Sun et al., 2023), we decompose all linear layers in LLM transformers. We implemented EinSort in PyTorch (Paszke et al., 2019) and used the HuggingFace Transformers library (Wolf, 2019) for handling models and datasets. All experiments are conducted on NVIDIA A40 or A100 GPUs.

For LLM experiments, we consider the Qwen3 (Yang et al., 2025), Gemma3 (Gemma Team et al., 2025), and Phi3 (Haider et al., 2024) models. We show results on different sizes of models to provide a broader picture for the performance of EinSort. We measure perplexity score for one of the most widely used benchmarks: raw-WikiText2 (WT2) (Merity et al., 2016). Details of LLMs and datasets we used are found in Appendices S and T.

We use the head axis for reduction and 64-stacked head dimension for sorting, and thus the sliced sorting requires $\lceil \log_2((64 \times d)!) \rceil / (64 \times d \times h)$ bits, where d and h are head dimension and the number of heads, respectively. For Qwen3-0.6B/1.7B, the sliced sorting requires $\lceil \log_2((64 \times 128)!) \rceil / (64 \times 128 \times 8) \simeq 1.44$ bits per KV cache parameter. For Gemma3-4B, the sorting requires $\lceil \log_2((64 \times 256)!) \rceil / (64 \times 256 \times 4) \simeq 3.14$ bits per cache parameter. For Phi3-mini, the permutation requires $\lceil \log_2((64 \times 96)!) \rceil / (64 \times 96 \times 32) \simeq 0.35$ bits per parameter. Note that we can freely adjust the slicing operation to control the required memory.

R. LLM benchmark

R.1. LLM model variants

Besides Qwen3-0.6B and Gemme3-4B models in Fig. 14, we add WT2 perplexity evaluations for various LLM models including Qwen3-1.7B and Phi-3.5-mini. Fig. 14a shows the PPL for Qwen3-1.7B model when KV cache is compressed by regular SVD, full-sort SVD, and EinSort. The trend is similar to Qwen3-0.6B model in Fig. 4.

Fig. 14b shows the PPL for Phi3-mini model when KV cache is compressed by regular SVD, full-sort SVD, and EinSort. We observe that EinSort can significantly improve the tensorization accuracy towards full-sort case (which requires $\lceil \log_2((64 \times 96)!) \rceil / (64 \times 96) \simeq 11.1$ bits for permutation memory) while the permutation memory is kept small as 0.35 bits.

[7] https://github.com/BaiTheBest/SparseLLM

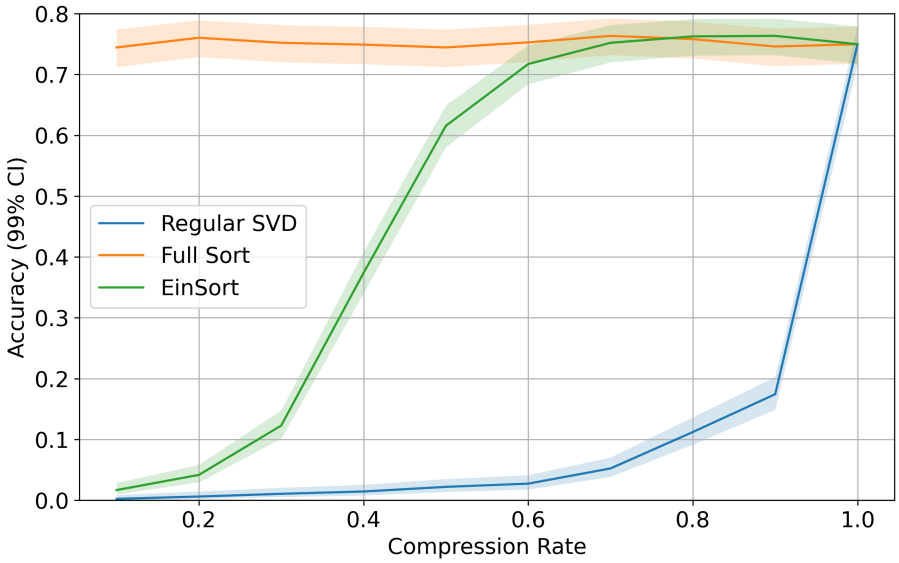
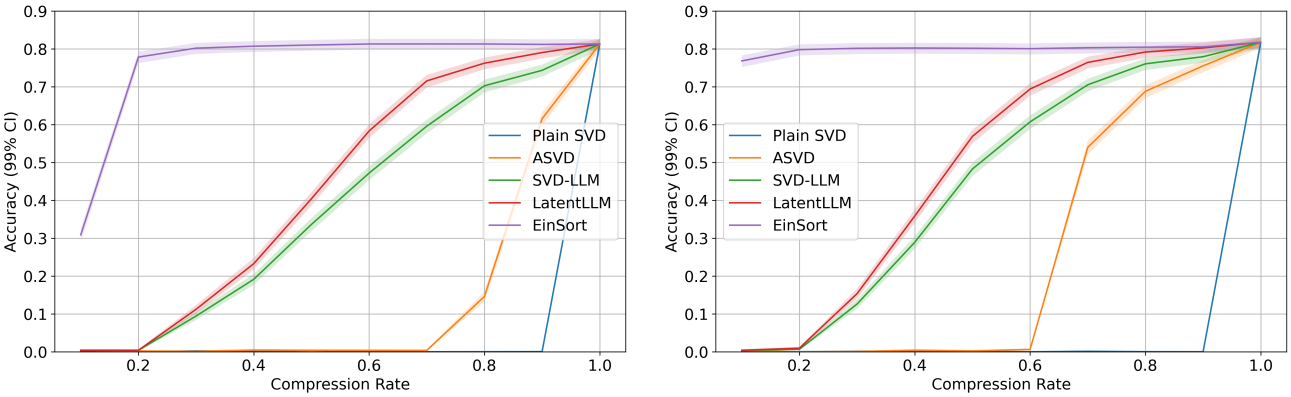


Figure 15. GSM8K accuracy for Phi-4-mini as a function of KV cache compression ratio. Shaded zone is Wilson's 99% confidence interval.



(a) Qwen3-VL-4B-Instruct *(b)* Qwen3-VL-8B-Instruct

Figure 16. Accuracy for compressed Qwen3-VL models on TextVQA benchmark.

R.2. LLM math reasoning

Besides perplexity evaluations, we add more practical LLM benchmark, specifically GSM8K (Cobbe et al., 2021) for mathematical reasoning. Fig. 15 shows the accuracy for Phi-4-mini when we employ KV cache compression. As well as the average accuracy, we plot a shaded band to represent the Wilson confidence interval at 99%. Similar to WT2 perplexity results, we observe that EinSort can significantly improve the accuracy over the regular SVD compression.

R.3. VLM visual reasoning

We next consider practical applications of VLMs on visual reasoning tasks. Specifically, we use Qwen3-VL (Bai et al., 2025) models on TextVQA (Singh et al., 2019) benchmark. Rather than KV cache reduction, we compress weights of linear modules for the LLM backbone in VLM models. We compare plain SVD, ASVD (Yuan et al., 2023), SVD-LLM (Wang et al., 2024b), and LatentLLM (Koike-Akino et al., 2026a) as baselines. All methods except plain SVD use activation aware preconditioning based on 64 calibration samples in TextVQA train splits. Fig. 16 shows performance for Qwen3-VL-4B and 8B models. We confirm that the proposed EinSort can keep high accuracy even at 20% compression, whereas the other baselines show steep degradation when we compress weights.

Table 5. Success rate (↑) of X-VLA model with different quantization methods on LIBERO robot manipulation benchmark at a compression rate of 0.1. **Bold** and underline denote the best and second best, respectively. Asterisk "*" indicates reaching competitive performance to the original un-compressed VLA.

Benchmark	Spatial	Object	Goal	Long	Avg	(95% CI)
Original	95.0%	98.5%	93.5%	87.5%	93.63%	(91.72–95.12%)
Plain SVD	7.5%	1.0%	0.0%	2.0%	2.63%	(1.72–3.98%)
ASVD	32.5%	21.0%	5.0%	13.0%	17.88%	(15.38–20.68%)
SVD-LLM	41.0%	26.0%	4.0%	31.0%	25.50%	(22.60–28.63%)
LatentLLM	43.0%	26.0%	3.5%	32.5%	26.25%	(23.32–29.41%)
EinSort	**93.5%**	**97.5%**	**90.5%**	***88.0%**	**92.38%**	(90.33–94.02%)

Table 6. Success rate (↑) of $\pi_{0.5}$ VLA model with different quantization methods on LIBERO robot manipulation benchmark at a compression rate of 0.4. **Bold** and underline denote the best and second best, respectively. Asterisk "*" indicates reaching competitive performance to the original un-compressed VLA.

Benchmark	Spatial	Object	Goal	Long	Avg	(95% CI)
Original	97.5%	100.0%	97.0%	96.5%	97.75%	(96.47–98.57%)
Plain SVD	27.0%	34.5%	8.5%	1.0%	17.75%	(15.26–20.55%)
ASVD	4.0%	5.5%	7.5%	0.0%	4.25%	(3.06–5.88%)
SVD-LLM	60.0%	79.0%	32.0%	27.0%	49.50%	(45.05–52.96%)
LatentLLM	66.5%	88.5%	37.0%	37.5%	57.38%	(53.92–60.76%)
EinSort	**96.0%**	**99.5%**	***97.5%**	**95.5%**	**97.13%**	(95.72–98.08%)

R.4. VLA robot manipulation

We further consider practical applications of LLMs on robot manipulation tasks using several VLAs. Specifically, we evaluate LIBERO (Liu et al., 2023a) robot manipulation benchmarks for X-VLA (Zheng et al., 2025), $\pi_{0.5}$ (Intelligence et al., 2025), and VLA-JEPA (Sun et al., 2026) foundation models, which use Florence-2 (Xiao et al., 2024a), PaliGemma (Beyer et al., 2024), and Qwen3-VL (Bai et al., 2025) as VLM backbone, respectively. Here, we focus on model weight compression via EinSort, rather than KV cache reduction. We compress all linear modules in attention and multi-layer perceptron (MLP) for the LLM in VLA.

We compare plain SVD, ASVD (Yuan et al., 2023), SVD-LLM (Wang et al., 2024b), and LatentLLM (Koike-Akino et al., 2026a) as baselines. Except plain SVD, we use 64 episodes in LIBERO-Spatial task suite for activation calibration to compute preconditioning. We evaluate success rates over 800 rollouts on four LIBERO benchmarks: Spatial, Object, Goal, and Long. Tables 5, 6, and 7 show the success rates for X-VLA, $\pi_{0.5}$, and VLA-JEPA models, respectively. We verify that our EinSort method significantly outperforms the state-of-the-art weight decomposition methods across all LIBERO benchmarks. It is interesting to see that EinSort can occasionally perform better than un-compressed VLA models. It is potentially because of regularization benefit, while we note that the gaps are within the Wilson confidence interval. Fig. 17 shows example video snapshots using the compressed VLA-JEPA model on LIBERO benchmarks.

S. Large foundation models

Qwen3 We use Qwen3 (Yang et al., 2025) dense models, which are decoder-only transformers spanning 270M to 30B parameters, built on a consistent architecture with RMSNorm, SwiGLU feed-forward layers, and rotary positional embeddings. All variants employ grouped-query attention (GQA) with a fixed small number of key–value heads while scaling the number of query heads with model width, reducing KV-cache cost. Importantly, the hidden size is decoupled from the attention projection width, providing additional flexibility. Parameters are listed in Table 8. It is under Apache-2.0 license.

Gemma3 Gemma3 models (Gemma Team et al., 2025) are decoder-only transformer architectures released across a wide range of scales, from 270M to 27B parameters, and include both text-only and multimodal variants. Similar to Qwen3,

Table 7. Success rate (↑) of VLA-JEPA model with different quantization methods on LIBERO robot manipulation benchmark at a compression rate of 0.8. **Bold** and underline denote the best and second best, respectively. Asterisk "*" indicates reaching competitive performance to the original un-compressed VLA.

Benchmark	Spatial	Object	Goal	Long	Avg	(95% CI)
Original	97.5%	100.0%	99.0%	93.5%	97.50%	(96.17–98.38%)
Plain SVD	0.0%	0.0%	0.0%	0.0%	0.00%	(0.00–0.48%)
ASVD	62.0%	75.5%	49.5%	21.0%	52.00%	(48.54–55.44%)
SVD-LLM	18.5%	16.0%	19.0%	0.5%	13.50%	(11.31–16.04%)
LatentLLM	87.0%	64.5%	59.0%	28.5%	59.75%	(56.31–63.09%)
EinSort	***98.0%**	***100.0%**	**98.0%**	***95.0%**	***97.75%**	(96.47–98.57%)

Table 8. Architecture parameters of Qwen3 dense models (Yang et al., 2025)

Model	layers	heads	KV heads	hidden size	head dim	MLP dim	Huggingface ID
0.6B	28	16	8	1024	128	3072	Qwen/Qwen3-0.6B
1.7B	28	16	8	2048	128	6144	Qwen/Qwen3-1.7B
4B	36	32	8	2560	128	9728	Qwen/Qwen3-4B
8B	36	32	8	4096	128	12288	Qwen/Qwen3-8B
14B	40	40	8	5120	128	17408	Qwen/Qwen3-14B
32B	64	64	8	5120	128	25600	Qwen/Qwen3-32B

all Gemma3 models adopt RMSNorm, SwiGLU feed-forward networks, and rotary positional embeddings, as well as grouped-query attention (GQA). The per-head dimension is fixed at 256 across model sizes. Parameters are listed in Table 9. It is licensed under the Gemma terms of use.

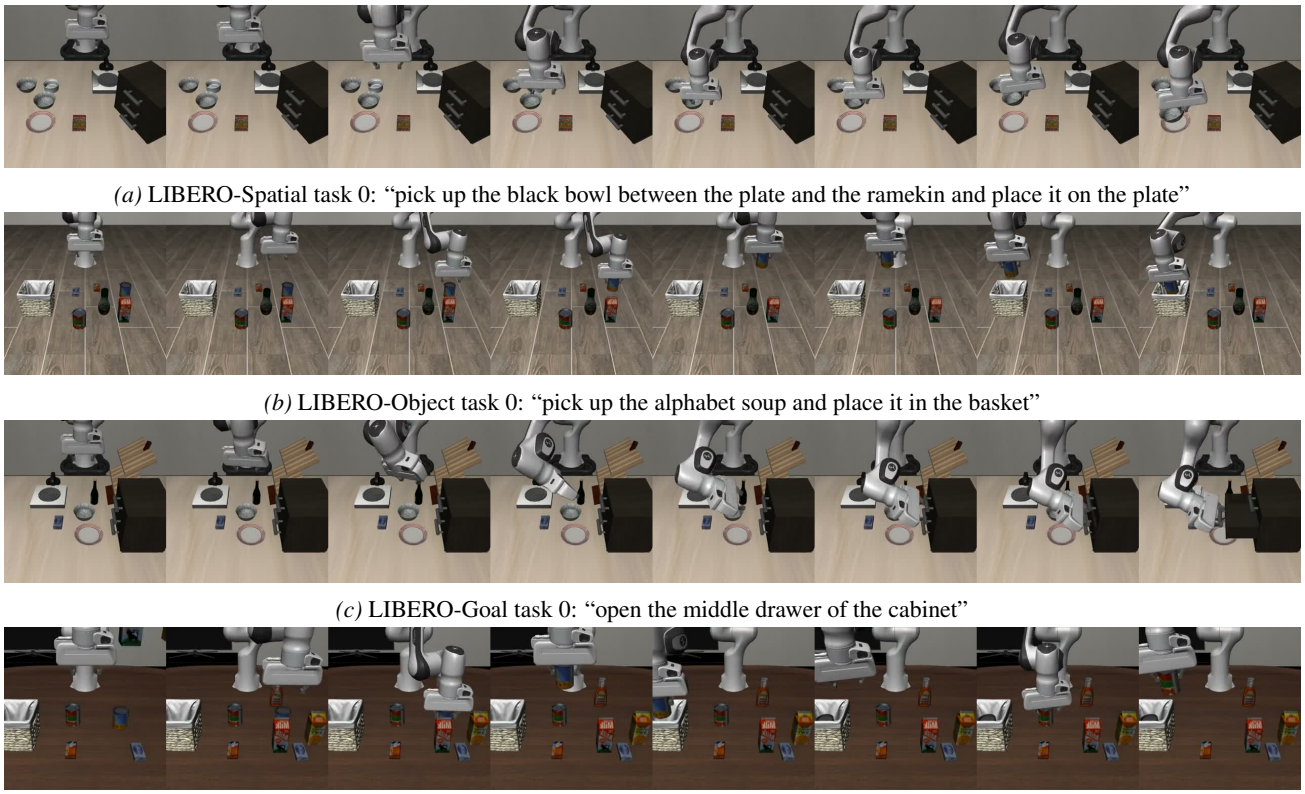
Phi-3 Phi-3 models (Haider et al., 2024) are language models developed for efficient on-device and edge AI. Phi-3 models achieve strong reasoning and coding performance through high-quality training data and optimized transformer architectures. Parameters are listed in Table 10. It is under MIT license.

Phi-4 Phi-4 models (Abdin et al., 2024) are a family of compact language and multimodal foundation models for efficient reasoning, coding, and on-device AI. The family includes text-only, reasoning-specialized (Aneja et al., 2026), and multimodal variants (Abouelenin et al., 2025) supporting language, vision, and audio inputs, while maintaining strong efficiency relative to model size. Parameters are listed in Table 11. It is released under the MIT license.

Qwen3-VL Qwen3-VL (Bai et al., 2025) is a family of multimodal LLMs that extend the Qwen3 transformer with vision–language capabilities. The models integrate a SigLIP-based vision encoder (Zhai et al., 2023) with the Qwen3 language backbone (Yang et al., 2025) through a projection module, enabling joint reasoning over images, videos, and text for tasks such as visual question answering, captioning, and document understanding. Parameters are listed in Table 12. It is licensed under Apache-2.0.

X-VLA X-VLA (Cross-modal Vision–Language–Action) (Zheng et al., 2025) is a unified framework that jointly models visual perception, language understanding, and action generation within a shared representation space. Leveraging multimodal pretraining, X-VLA captures compositional and temporal structure, allowing robust generalization across tasks and environments for embodied AI systems. It uses Florence-2 (Xiao et al., 2024a) as a VLM backbone, which uses DaViT-B vision encoder (Ding et al., 2022). The parameter is listed in Table 13. It is released under the Apache-2.0 license.

$\pi_{0.5}$ The $\pi_{0.5}$ model (Intelligence et al., 2025) is the state-of-the-art VLA transformer having 2.3B parameters, that maps visual observations and language instructions directly to continuous robot actions through flow-matching diffusion policy. It uses PaliGemma (Beyer et al., 2024) as a VLM backborne, and Gemma (Team et al., 2024) as a flow-matching diffusion policy. Trained via distillation from a larger foundation model on diverse robot interaction data, it achieves strong zero-shot generalization while remaining lightweight and deployable for real-world manipulation tasks. Parameters are listed in Table 14. It is licensed under the Gemma terms of use.



(a) LIBERO-Spatial task 0: "pick up the black bowl between the plate and the ramekin and place it on the plate"

(b) LIBERO-Object task 0: "pick up the alphabet soup and place it in the basket"

(c) LIBERO-Goal task 0: "open the middle drawer of the cabinet"

(d) LIBERO-Long task 0: "put both the alphabet soup and the tomato sauce in the basket"

Figure 17. Robot manipulation video snapshots for the compressed VLA-JEPA model on LIBERO benchmarks.

Table 9. Gemma3 instruction-tuned text transformer parameters (Gemma Team et al., 2025)

Model	layers	heads	KV heads	hidden size	head dim	MLP dim	Huggingface ID
270M	18	4	1	640	256	2048	google/gemma-3-270m-it
1B	26	4	1	1152	256	6912	google/gemma-3-1b-it
4B	34	8	4	2560	256	10240	google/gemma-3-4b-it
12B	48	16	8	3840	256	15360	google/gemma-3-12b-it
27B	62	32	16	5376	256	21504	google/gemma-3-27b-it

VLA-JEPA VLA-JEPA (Vision–Language–Action with Joint Embedding Predictive Architecture) (Sun et al., 2026) augments VLA policies with a latent predictive world model based on V-JEPA2 (Assran et al., 2025), enabling the agent to learn action-relevant future representations without reconstructing pixels. Built on a Qwen3-VL-2B VLM backbone (Bai et al., 2025) and a V-JEPA2 latent encoder based on ViT-L, it predicts future latent states and generates robot actions using a diffusion policy based on DiT-B (Peebles & Xie, 2023). By jointly learning perception, language grounding, and latent dynamics, VLA-JEPA achieves strong generalization and robustness across manipulation tasks while improving data efficiency and transfer to unseen environments. Parameters are listed in Table 15. It is released under the Apache-2.0 license.

T. Datasets

Wikitext-2 (WT2) The WikiText language modeling dataset (Merity et al., 2016) is a collection of over 100 million tokens extracted from the set of verified good and featured articles on Wikipedia. The dataset is available under the CC BY-SA-4.0 license. The wikitext-2-raw-v1 contains 36,718, 3,760, and 4,358 samples for train, validation, and test splits, respectively. We use `https://huggingface.co/datasets/mindchain/wikitext2`.

Table 10. Phi-3 instruction-tuned text transformer parameters (Haider et al., 2024)

Model	layers	heads	KV heads	hidden size	head dim	MLP dim	Huggingface ID
mini (3.8B)	32	32	32	3072	96	8192	microsoft/Phi-3.5-mini-instruct
small (7B)	32	32	8	4096	128	—	microsoft/Phi-3-small-8k-instruct
medium (14B)	40	40	10	5120	128	17920	microsoft/Phi-3-medium-4k-instruct

Table 11. Phi-4 instruction-tuned text transformer parameters (Abouelenin et al., 2025; Aneja et al., 2026)

Model	layers	heads	KV heads	hidden size	head dim	MLP dim	Huggingface ID
mini (3.8B)	32	32	8	3072	128	8192	microsoft/Phi-4-mini-instruct
reasoning (14B)	40	40	10	5120	128	17920	microsoft/Phi-4-reasoning

GSM8K The Grade School Math 8K (GSM8K) dataset (Cobbe et al., 2021) is a benchmark for evaluating mathematical reasoning in LLMs. It consists of 8,500 high-quality grade-school-level word problems written by human annotators, each paired with a detailed natural-language solution and a final numeric answer. The dataset is designed to assess multi-step arithmetic reasoning rather than factual recall, requiring models to generate intermediate reasoning steps to solve problems correctly. GSM8K is released under the MIT license and contains 7,473 training samples and 1,319 test samples. We use `https://huggingface.co/datasets/openai/gsm8k`.

TextVQA TextVQA (Singh et al., 2019) requires VLM models to read and reason about text in images to answer questions about them. Specifically, models need to incorporate the new modality of text present in the images and reason over it to answer TextVQA questions. TextVQA dataset contains 45,336 questions over 28,408 images from the OpenImages dataset. We use `https://huggingface.co/datasets/lmms-lab/textvqa`, licensed under CC-BY-4.0.

LIBERO The LIBERO dataset (Liu et al., 2023a) is a benchmark for robotic vision-language-action (VLA) learning that evaluates long-horizon, compositional manipulation in simulated household environments. It provides five benchmark suites: spatial; object; goal; long, and short. The short task suite has 90 task variants and the other suites have 10 different tasks. Each task includes multimodal data—video observations, language instructions, and low-level control trajectories—enabling end-to-end learning from vision and language to robot actions. We use `https://huggingface.co/datasets/lerobot/libero`, licensed under Apache-2.0.

U. Libraries

We partly use some public libraries as below.

- Pytorch `https://github.com/pytorch/pytorch`; version 2.9.1; BSD-3-Clause license
- opt_einsum `https://github.com/dgasmith/opt_einsum`; version 3.3.0; MIT license
- einops `https://github.com/arogozhnikov/einops`; version 0.8.2; MIT license
- tensorly `https://github.com/tensorly/tensorly`; version 0.9.0; BSD-3 Clause license
- transformers `https://github.com/huggingface/transformers`; version 5.8.0; Apache-2.0 license
- datasets `https://github.com/huggingface/datasets`; version 4.85; Apache-2.0 license
- SciPy `https://github.com/scipy/scipy`; version 1.17.1; BSD-3-Clause license
- PyEDA `https://github.com/cjdrake/pyeda`; version 0.29.0; BSD-2-Clause license
- RevKit `https://github.com/msoeken/revkit`; MIT license
- Galois `https://github.com/mhostetter/galois`; version 0.4.11; MIT license
- LeRobot `https://github.com/huggingface/lerobot`; version 0.5.1; Apache-2.0 license

Table 12. Qwen3-VL transformer parameters (Bai et al., 2025)

Model	layers	heads	KV heads	hidden size	head dim	MLP dim	Hugginface ID
2B	24	16	8	2048	128	11008	Qwen/Qwen3-VL-2B-Instruct
4B	32	32	8	4096	128	22016	Qwen/Qwen3-VL-4B-Instruct
8B	36	32	8	4096	128	22016	Qwen/Qwen3-VL-8B-Instruct
32B	64	40	8	5120	128	27392	Qwen/Qwen3-VL-32B-Instruct

Table 13. X-VLA transformer parameters (Zheng et al., 2025): Huggingface ID lerobot/xvla-libero

Module	layers	heads	KV heads	hidden size	head dim	MLP dim
VLM: Florence-2-Large	24	16	16	1024	64	4096
Vision Encoder: DaViT	12	64	64	2048	32	8192

Table 14. $\pi_{0.5}$ VLA transformer parameters (Intelligence et al., 2025): Huggingface ID lerobot/pi05_libero_finetuned

Module	layers	heads	KV heads	hidden size	head dim	MLP dim
LLM: Gemma-2B	18	8	1	2048	256	16384
Vision Encoder: SigLIP ViT-L	24	16	16	1024	64	4096
Diffusion Policy: Gemma-300M	18	8	1	1024	128	4096

Table 15. VLA-JEPA transformer parameters (Sun et al., 2026): Huggingface ID lerobot/VLA-JEPA-LIBERO

Module	layers	heads	KV heads	hidden size	head dim	MLP dim
VLM: Qwen3-VL-2B	36	16	8	1536	96	8960
World model: V-JEPA2 ViT-L	24	16	16	1024	64	4096
Diffusion Policy: DiT-B	12	12	12	768	64	3072