NeuKron: Constant-Size Lossy Compression of Sparse Reorderable Matrices (e.g., Bipartite Graphs) and Tensors

NOTE: If a preview does not appear properly, please download this file.

ABSTRACT

Many real-world data are naturally represented as a sparse reorderable matrix, whose rows and columns can be arbitrarily ordered (e.g., the adjacency matrix of a bipartite graph). Storing a sparse matrix in conventional ways requires an amount of space linear in the number of non-zeros, and lossy compression of sparse matrices (e.g., Truncated SVD) typically requires an amount of space linear in the number of rows and columns. In this work, we propose NeuKron for compressing a sparse reorderable matrix into a constant-size space. NeuKron generalizes Kronecker products using a recurrent neural network with a constant number of parameters. NeuKron updates the parameters so that a given matrix is approximated by the product and reorders the rows and columns of the matrix to facilitate the approximation. The updates take time linear in the number of non-zeros in the input matrix, and the approximation of each entry can be retrieved in logarithmic time. We also extend NeuKron to compress sparse reorderable tensors (e.g. multi-layer graphs), which generalize matrices. Through experiments on ten real-world datasets, we show that NeuKron is **(a) Compact:** requiring up to five orders of magnitude less space than its best competitor with similar approximation errors, **(b) Accurate:** giving up to 10× smaller approximation error than its best competitors with similar size outputs, and **(c) Scalable:** successfully compressing a matrix with over 230 million non-zero entries.

CCS CONCEPTS

• **Information systems** → **Data mining**; **Data compression**.

KEYWORDS

Data Compression, Bipartite Graph, Sparse Matrix, Sparse Tensor

ACM Reference Format:

NOTE: If a preview does not appear properly, please download this file.. 2023. NeuKron: Constant-Size Lossy Compression of Sparse Reorderable Matrices (e.g., Bipartite Graphs) and Tensors. In *Proceedings of The Web Conference (TheWebConf '23).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

We consider a matrix to be *sparse* if the number of non-zero entries is much smaller than that of all entries. Sparse matrices naturally represent many types of data from various domains, including the following examples:

- **E-commerce**: User-item matrices represent how many times each user purchased each item [15, 30].
- **Search Engines**: Document-keyword matrices represent how many times each document contains each keyword [29]. User-ad matrices indicate how many times each user clicked each ad given by search engines [34]. The adjacency matrices of web graphs represent hyperlinks between documents [4].
- **Social Media**: The adjacency matrices of social networks indicate friendship between users [9, 33]. User-group matrices indicate which user belongs to each group [44].
- **Bibliography**: Author-paper matrices represent who authored each paper [35]. The adjacency matrices of collaboration networks represent co-authorships between authors [44].

Despite their sparsity, many real-world matrices require considerable space. Examples include user-ad matrices [40] and the adjacency matrices of web graphs [4] with billions of rows or columns; and keyword-document matrices [29] and the adjacency matrices of online social networks [9, 33] with tens of billions of non-zeros.

Compression of such large sparse matrices becomes important as smartphones and IoT devices become popular. Such memory-limited mobile devices are often required to process a large amount of data without sending them to clouds or servers, due to potential privacy risks [22]. Moreover, as the size of large-scale matrices grows rapidly, storing them is challenging also in desktops and servers [3, 9, 33], and for federate learning, compressing matrices is required to reduce communication costs [18]. As a result, a large number of lossy matrix-compression techniques [3, 10, 39] have been developed over the last few decades.

To the best of our knowledge, existing lossy-compression methods for sparse matrices create outputs whose sizes are at least linear in the numbers of rows and columns of the input matrix. For example, given an N-by-M matrix $\mathbf{A}$ and a positive integer K, truncated singular value decomposition (T-SVD) [12, 38] outputs two matrices of which the numbers of entries are $O(KN)$ and $O(KM)$. Recent methods [3, 10, 39] have the same limitations, while they provide a better trade-off between space and information loss than T-SVD.

Can we compress a matrix into a constant-size space, which can even be smaller than the number of rows and columns? In this paper, we exploit the fact that **many real-world sparse matrices are *reorderable***, i.e., the rows and columns of the matrices can be arbitrarily ordered.[1] All of the matrices discussed in the first paragraph, which are essentially bipartite graphs (nodes of one type correspond to rows, and nodes of the other type correspond to columns), are reorderable. For example, in the case of user-item matrix built based on e-commerce data, which user (item) comes next to which user (item) does not matter. Our key idea is to **order rows and columns** to facilitate our model to learn and exploit meaningful patterns in the input matrix for compression.

[1] A matrix is *non-reorderable* if the orders of rows and columns in it convey information. For example, images and multivariate time series are non-reorderable matrices since the orders of rows and columns in them indicate spatial and temporal adjacency.

TheWebConf '23, May 1–5, 2023, Austin, TX, USA

© 2023 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00

https://doi.org/XXXXXXX.XXXXXXX

Specifically, we present NeuKron, a constant-size lossy compression method for sparse reorderable matrices. It consists of a machine-learning model and novel training schemes. The model generalizes the Kronecker power and enhances its expressive power using a recurrent neural network with a constant number of parameters. The training scheme, which is crucial for performance, is to reorder rows and columns in the input matrix to create patterns that the machine-learning model can exploit for better compression. Consider an N-by-M matrix with L non-zeros, where $N \leq M$ without loss of generality. The model and the training schemes are designed carefully so that each training epoch takes $O(M+L\log M)$ time, and after training, the approximation of each entry can be retrieved in $O(\log M)$ time. Note that the time complexity of training depends only on the number of non-zeros instead of all entries.

In addition, we extend NeuKron for lossy-compression of sparse reorderable tensors while maintaining its strengths. Tensors (i.e., multi-dimensional arrays) generalize matrices to higher dimensions, and in other words, matrices are 2-order tensors. Sparse tensors have been used widely for various purposes, including context-aware recommender systems [20] and knowledge base completion [23]), and for lossy compression of them, tensor decomposition methods (e.g., CP [2, 6] and Tucker [2, 41]) have been developed.

For evaluation, we perform extensive experiments using ten real-world matrices (spec., bipartite graphs) and tensors. The results reveal the following advantages of NeuKron:

- **Compact:** Its output is up to **5 orders of magnitude smaller** than competitors' with similar approximation error.
- **Accurate:** It achieves up to **10.1× smaller approximation error** than its best competitors that give similar-size outputs.
- **Scalable:** Its running time is **linear** in the number of non-zero entries, and it successfully compresses matrices with **over 230 millions of non-zero entries** on commodity GPUs.

Reproducibility: The code and datasets are available at [1].

Remarks on non-reorderable matrices: While we focus on reorderable matrices in this paper, NeuKron can also be applied to non-reorderable matrices if the mapping between the original and new orders of rows and columns are stored additionally. **Even with this extra space requirement**, which is linear in the number of rows and columns, **NeuKron still gives the best trade-off** between size and approximation error, as shown in Appendix C.

2 RELATED WORKS

In this section, we review lossy-compression methods for matrices and tensors. Those for lossy compression of sparse matrices or tensors of any size are compared in Table 1 and also in Section 6.

Factorization-based matrix compression: Given a matrix $\mathbf{A} \in \mathbb{R}^{N\times M}$, singular value decomposition (SVD) [13] decomposes $\mathbf{A}$ into $\mathbf{U}\Sigma\mathbf{V}^T$ where $\mathbf{U} \in \mathbb{R}^{N\times R}$, $\mathbf{V} \in \mathbb{R}^{M\times R}$, Σ is a diagonal matrix with its singular values, and R is its rank. Truncated SVD (T-SVD) [12, 38] outputs the K ($\leq R$) most largest singular values and the corresponding vectors of $\mathbf{U}$ and $\mathbf{V}$ from which the rank-K approximation of $\mathbf{A}$ best in terms of the Frobenius norm can be obtained [38]. Its outputs have $O(K(M+N))$ real values, and typically most of them are non-zero. For further compression, CUR decomposition [10] aims to yield sparse outputs. Specifically, a sparse matrix $\mathbf{A}$ is decomposed into CUR (i.e., $\mathbf{A} \approx \mathbf{CUR}$), where $\mathbf{C} \in \mathbb{R}^{N\times K}$ and $\mathbf{R} \in \mathbb{R}^{K\times M}$ are constructed by sampling K columns and rows from $\mathbf{A}$, respectively. The matrix $\mathbf{U} \in \mathbb{R}^{K\times K}$ is dense but small, and it is determined by $\mathbf{C}$ and $\mathbf{R}$ so that the approximation error is minimized. Compact matrix decomposition (CMD) [39] keeps only unique columns and rows in $\mathbf{C}$ and $\mathbf{R}$ for further efficiency.

Co-clustering-based matrix compression: ACCAMS and bACCAMS [3] use an additive combination of small co-clusterings to approximate a given matrix. While the numbers of parameters of them are linear in the numbers of rows and columns, they produce intermediate results whose size is linear in the number of (potentially zero) known entries. Thus, they are computationally and memory inefficient when most entries are known but zero, as in matrices considered in this work.

Kronecker product-based matrix compression: The adjacency matrix of a Kronecker graphs [26] is a Kronecker power of a fixed seed matrix (e.g., 2-by-2 matrix). KronFit [27, 28] searches for a seed matrix whose Kronecker power approximates the adjacency matrix of a given graph. While KronFit is designed for adjacency matrices, it can be easily extended to matrices of any size, and the output seed matrix can be considered as a constant-size lossy compression of a given matrix. However, the approximation error is considerable, even when the seed matrix is large, due to the inflexibility of the Kronecker product, as shown in Section 6.2.

Tensor compression: CP decomposition (CP) [6] and Tucker decomposition (Tucker) [41] generalize the aforementioned T-SVD to higher-order tensors. They approximate a given tensor using the sums and products (e.g., outer product and n-mode product) of much smaller low-rank tensors and matrices, which can be considered as a lossy compression of the given tensor. There exist quasi-optimal methods for Tucker [14, 42]. Moreover, efficient CP and Tucker methods for sparse tensors have been developed [2]. For lossless compression of sparse tensors, compressed sparse fiber (CSF) [36, 37] is available.

Other related works: Unipartite-graph summarization algorithms [24, 25, 31] can be used for compressing adjacency matrices of unipartite graphs, while they cannot be directly applied to weighted and/or non-symmetric matrices, which we aim to compress. Matrix sketching methods replace a given large matrix with a more compact matrix that follows the properties of the input matrix, for example, by leaving only important columns (rows) of the input matrix [10, 11]. These methods, however, cannot be applied to our problem (i.e., Problem 1 in Section 3.2) because the entries of the input matrix cannot be estimated directly from their outputs. For federated learning, the weights of neural networks, which are typically dense matrices or tensors, can be compressed [18].

3 NOTATIONS AND PROBLEM DEFINITION

In this section, we introduce basic concepts and give a formal problem definition. See Table 2 for common notations.

3.1 Notations and Concepts

Sparse reorderable matrix and tensor: A *matrix* $\mathbf{A} \in \mathbb{R}^{N\times M}$ is a 2-dimensional array with N rows and M columns, and real entries. A D-order *tensor* $\mathcal{X} \in \mathbb{R}^{N_1\times\cdots\times N_D}$ is a D-dimensional array of size $N_1 \times \cdots \times N_D$ with real entries. We use a_{ij} or $A(i,j)$ to denote the (i,j)-th entry of $\mathbf{A}$, and we use $x_{i_1,\cdots,i_D}$ to denote the $(i_1,\cdots,i_D)$-th entry of $\mathcal{X}$. We consider a matrix or a tensor to be *sparse* if the number of non-zero entries is much smaller than that of all

Table 1: Comparison of lossy-compression methods for sparse matrices and tensors. For simplicity, we treat the tensor order and all hyperparameters as constants. Comparisons are relative, and we provide details in [1].

Methods	Space & Accuracy Trade-off	Training Complexity (per iteration)	Inference Complexity (per entry)	Number of Hyper-parameters	Training Time (total)
NeuKron	Strong	∝ #non-zeros	∝ $\log(N_{\max})$*	4**	Long
T-SVD [12, 42]	Weak	∝ #non-zeros	constant	1	Short
CMD [39]	Moderate	∝ #non-zeros	constant	2	Moderate
CUR [10]	Moderate	∝ #non-zeros	constant	2	Moderate
ACCAMS [3]	Moderate	∝ #all-entries	constant	2	Moderate
bACCAMS [3]	Moderate	∝ #all-entries	constant	4	Long
KronFit [27, 28]	Weak	∝ #non-zeros	∝ $\log(N_{\max})$*	4	Long
CP [6]	Weak	∝ #non-zeros	constant	1	Moderate
Tucker [41]	Weak	∝ #non-zeros	constant	1	Moderate

* Here $N_{\max} = \max(N_1, \cdots, N_D)$ is the maximum dimensionality (i.e., mode length).
** The learning rate, the optimizer, the weight parameter for the criterion of switching, and the size of hidden dimensions in LSTM.

entries.[2] We call a matrix *reorderable* if its rows and columns can be arbitrarily ordered. We provide some examples of reorderable matrices where the orders of rows and columns do not convey any information and some examples of non-reorderable ones (see Footnote 1) in Section 1. Similarly, we call a tensor reorderable if the indices in each mode can be arbitrarily ordered.

Approximation error: The *Frobenius norm* is a function $\|\cdot\|_F : \mathbb{R}^{N\times M} \to \mathbb{R}$ defined as the square root of the square sum of all entries in the given matrix. Similarly, the Frobenius norm of a tensor is defined as the square root of the square sum of all entries in the given tensor. The *approximation error* of a matrix $\tilde{\mathbf{A}}_\Theta$ that approximates $\mathbf{A}$ is defined as $\|\mathbf{A} - \tilde{\mathbf{A}}_\Theta\|_F^2$. Similarly, the approximation error of $\tilde{\mathcal{X}}_\Theta$ that approximates $\mathcal{X}$ is defined as $\|\mathcal{X} - \tilde{\mathcal{X}}_\Theta\|_F^2$.

Kronecker product and power: Given two matrices $\mathbf{A} \in \mathbb{R}^{N\times M}$ and $\mathbf{B} \in \mathbb{R}^{P\times Q}$, the *Kronecker product* $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{NP\times MQ}$ is a large matrix formed by multiplying $\mathbf{B}$ by each element of $\mathbf{A}$, i.e.,

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1M}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{N1}\mathbf{B} & \cdots & a_{NM}\mathbf{B} \end{bmatrix}.$$

We denote the l-th *Kronecker power* of $\mathbf{A}$ as $\mathbf{A}^{\otimes l}$, where $\mathbf{A}^{\otimes l} = \mathbf{A}^{\otimes (l-1)} \otimes \mathbf{A}$ and $\mathbf{A}^{\otimes 1} = \mathbf{A}$.

3.2 Problem Definition

The constant-size lossy matrix compression problem that we address in this paper is defined in Problem 1. It should be noted that the given constant k can be even smaller than N and M. The problem of *constant-size lossy compression of a sparse reorderable tensor* can be defined by simply replacing the matrix $\mathbf{A}$ with a tensor $\mathcal{X}$ and $\|\mathbf{A} - \tilde{\mathbf{A}}_\Theta\|_F^2$ with $\|\mathcal{X} - \tilde{\mathcal{X}}_\Theta\|_F^2$.

Problem 1. (Constant-size Lossy Compression of a Sparse Reorderable Matrix)
- ***Given:*** *(1) a sparse and reorderable matrix* $\mathbf{A} \in \mathbb{R}^{N\times M}$, *(2) a constant* $k = O(1)$,
- ***Find:*** *a model* Θ
- ***to Minimize:*** *the approximation error* $\|\mathbf{A} - \tilde{\mathbf{A}}_\Theta\|_F^2$, *where* $\tilde{\mathbf{A}}_\Theta$ *is the matrix approximated from* Θ.
- ***Subject to:*** *the number of parameters in* Θ *is at most* k.

[2] The ratio is at most 0.0046 in the datasets considered in the paper.

Table 2: Frequently-used notations

Symbol	Definition
$\mathbf{A} \in \mathbb{R}^{N\times M}$	an N-by-M sparse matrix
a_{ij} or $\mathbf{A}(i,j)$	(i,j)-th entry of $\mathbf{A}$
$\mathbf{A}_{i,:}$, $\mathbf{A}_{:,i}$	i-th row of $\mathbf{A}$, i-th column of $\mathbf{A}$
$\mathcal{X} \in \mathbb{R}^{N_1\times\cdots\times N_D}$	tensor
D	order of $\mathcal{X}$
$x_{i_1,\cdots,i_D}$	$(i_1,\cdots,i_D)$-th entry of $\mathcal{X}$
$\mathrm{nnz}(\mathbf{A})$, $\mathrm{nnz}(\mathcal{X})$	number of non-zero entries in $\mathbf{A}$ and $\mathcal{X}$
$\|\mathbf{A}\|_F$, $\|\mathcal{X}\|_F$	Frobenius norm of $\mathbf{A}$ and $\mathcal{X}$
$\otimes$	Kronecker product
$\mathbf{A}^{\otimes l}$, $\mathcal{X}^{\otimes l}$	l-th Kronecker power of $\mathbf{A}$ and $\mathcal{X}$
Θ	a NeuKron model which compresses $\mathbf{A}$ and $\mathcal{X}$
$\tilde{\mathbf{A}}_\Theta$, $\tilde{\mathcal{X}}_\Theta$	approximated matrix and tensor of $\mathbf{A}$ and $\mathcal{X}$ by Θ
q	a parameter for the scale of model outputs
h	hidden dimension in LSTM
$[n]$	a set of integers from 1 to n (i.e., $\{1, 2, \cdots, n\}$)

4 PROPOSED METHOD

In this section, we present NeuKron, a constant-space lossy compression method for sparse reorderable matrices and tensors. We first describe its neural network model and then the training strategies for it. After that, we analyze the computational complexity of NeuKron. For ease of explanation, we assume that the input is a matrix through the section, and then we describe the extensions for tensors in Section 5.

4.1 Model

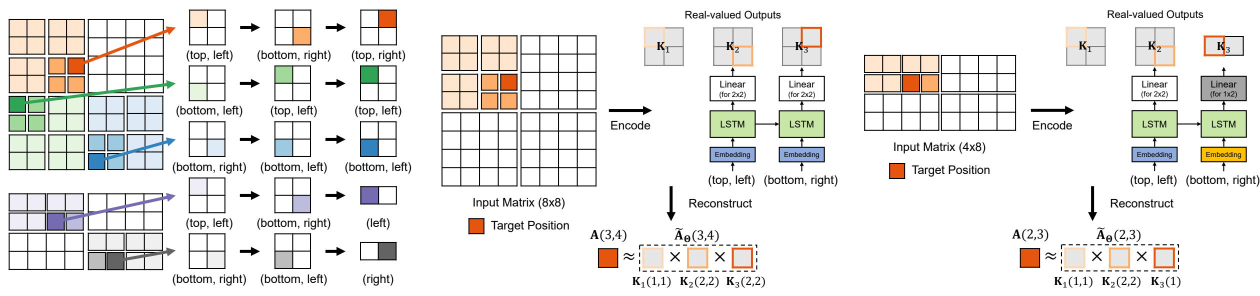
4.1.1 Overview. When designing a neural network model Θ for NeuKron, we aim to achieve the following goals:

- **G1. Constant Size:** The number of parameters of the model should be constant, regardless of the size of the input matrix.
- **G2. Exploitation of Sparsity:** It should be possible to fit the model to the input by accessing only non-zero entries.
- **G3. Fast Approximation:** From the trained model, it should be possible to approximate each entry of the input matrix in sublinear time (preferably, in constant or logarithmic time).

For **G1**, given a matrix $\mathbf{A}$ to be compressed, we encode the position (i,j) of each entry a_{ij} as a sequence and use an auto-regressive sequence model, specifically LSTM [16], which has a constant number of parameters, to process the sequence. For our purpose, LSTM performs similarly with GRU [8] and outperforms the decoder layer of Transformer [43], as shown empirically in [1]. For an entry a_{ij}, the sequence encoding the position (i,j) is fed into LSTM, and the outputs of LSTM are combined for its approximation $\tilde{a}_{ij}$ in logarithmic time, achieving **G3** (see Theorem 1 in Section 4.3). Moreover, regarding **G2**, the outputs of LSTM are combined so that the sparsity can be exploited for efficient computation of the objective and its gradient (see Section 4.2.2). The details of encoding inputs and combining outputs are described in the following subsections. Regarding **G3**, it should be noticed that many factorization-based methods approximate each entry even in constant time (see Table 1).

4.1.2 Encoding inputs (lines 1-3 of Algorithm 1). For simplicity, we assume an input matrix $\mathbf{A} \in \mathbb{R}^{N\times M}$ where $N = M = 2^l$ (see Section 4.1.4 for generalization to matrices of any size). Algorithm 1 depicts how NeuKron approximates such $\mathbf{A}$.

For each entry a_{ij} of $\mathbf{A}$, NeuKron encodes its position (i,j) in a sequence of length $l = \log_2 M$ by recursively subdividing $\mathbf{A}$ in a top-down manner. NeuKron first chooses the partition where



(a) Encoding Examples (b) Model for Square Matrices (c) Model for Matrices of Any Size

Figure 1: The overall approximation process of NeuKron. It encodes the input position into a sequence by recursively dividing the input matrix. The sequence is fed into LSTM, and the outputs of LSTM are aggregated based on the Kronecker product.

Algorithm 1: Approximation process of NeuKron for an N-by-$N(=2^l)$ matrix $\mathbf{A}$

```
Input: (a) a position: $(i,j) \in [N] \times [N]$ where $N = 2^l$
       (b) parameters of Embedding, LSTM, and the linear layer $(\mathbf{W}, \mathbf{b})$
       (c) scale parameter $q$ and the first matrix of Kronecker products $\mathbf{K}_1$
Output: an approximation $\tilde{a}_{ij}$ of $a_{ij}$, which is the $(i,j)$-th entry of
        the input matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$
1 for $k \leftarrow 1$ to $l$ do
2     $\mathbf{x}_k \leftarrow \mathrm{Embedding}(t(i,k), t(j,k))$        ▷ Sect. 4.1.2
3 $\mathbf{y}_2, \cdots, \mathbf{y}_l \leftarrow \mathrm{LSTM}(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{l-1})$
4 for $k \leftarrow 2$ to $l$ do
5     $\mathbf{K}_k \leftarrow \mathrm{Softplus}(\mathbf{W}\mathbf{y}_k + \mathbf{b})$        ▷ Sect. 4.1.3
6 return $\tilde{a}_{ij} \leftarrow \sqrt{q} \cdot \prod_{k=1}^{l} \mathbf{K}_k(t(i,k), t(j,k)) / \|\mathbf{K}_k\|_F$
```

a_{ij} lies when $\mathbf{A}$ is divided into 2×2 partitions of the same size (i.e., $2^{l-1} \times 2^{l-1}$). Each division gives four partitions at top left (TL), top right (TR), bottom left (BL), and bottom right (BR). Then, NeuKron repeats the process on the chosen partition until only the target entry a_{ij} is left. The sequence of the positions of the chosen partition is used to encode a_{ij}. In our implementation, each entry of the sequence, which is a position, is converted into a tuple in $\{1,2\} \times \{1,2\}$. Specifically, the k-th entry of the sequence that encodes the position (i,j) is $(t(i,k), t(j,k))$ where

$$t(i,k) := \left(\left\lfloor \frac{(i-1)}{2^{l-k}} \right\rfloor \bmod 2 \right) + 1. \tag{1}$$

Example 1 (Encoding in Square Matrices). *Suppose we encode the position $(3,4)$ of the square matrix in Figure 1(a), where $l = 3$. The position $(3,4)$ is located in the top-left partition of the input matrix, and it is located in the bottom-right part of the chosen partition. Lastly, the position $(3,4)$ is located at the top-right one of the lastly chosen partition. Thus, the position $(3,4)$ is encoded in the sequence TL $\rightarrow$ BR $\rightarrow$ TR, which becomes $(1,1) \rightarrow (2,2) \rightarrow (1,2)$ based on t (Eq. (1)).*

Each tuple in the sequence, except for the last one, goes through an embedding layer (line 2) to be converted into a corresponding embedded vector of size h, where h is a hyperparameter. Then, the vector is fed into LSTM (line 3).

4.1.3 Handling outputs (lines 4-6 of Algorithm 1). Below, we present how NeuKron produces an approximation. See Figure 1(b) for a pictorial description. We again assume an input matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ where $N = M = 2^l$ for ease of explanation. Given the position (i,j) of a target entry a_{ij}, NeuKron creates $\mathbf{K}_1 \in \mathbb{R}^{2 \times 2}, \cdots, \mathbf{K}_l \in \mathbb{R}^{2 \times 2}$. Specifically, given the sequence of tuples that encode (i,j) (see Section 4.1.2 for encoding), for each $k \in [l-1]$, the k-th LSTM cell receives the embedding of the k-th tuple, and then the hidden state of the cell goes through the linear layer and the Softplus activation to produce $\mathbf{K}_{k+1}$ (line 5). The entries of $\mathbf{K}_1$ are separate learnable parameters. The approximation $\tilde{a}_{ij}$ is computed from the (i,j)-th entry of their Kronecker product $\mathbf{K}_1 \otimes \cdots \otimes \mathbf{K}_l$ as follows (line 6):

$$\tilde{a}_{ij} := \sqrt{q} \cdot \prod_{k=1}^{l} \mathbf{K}_k(t(i,k), t(j,k)) / \|\mathbf{K}_k\|_F, \tag{2}$$

where $\prod_{k=1}^{l} \mathbf{K}_k(t(i,k), t(j,k))$ is the (i,j)-th entry of the Kronecker product, and q is a learnable parameter. It should be noticed that the entire Kronecker product does not have to be computed. By combining the outputs of LSTM using Eq.(2), **G2** in Section 4.1.1 can be achieved. Specifically, using Eq.(2) enables the exploitation of the sparsity of the input matrix $\mathbf{A}$ for linear-time training, as described in detail in Section 4.2.2 (see Lemma 1).

4.1.4 Handling matrices of any size. Below, we describe how the above processes of NeuKron are generalize to compress a matrix of any size. For a given matrix $A \in \mathbb{R}^{N \times M}$, we consider integers l_{row} and l_{col} such that $2^{l_{\text{row}}} \geq N$ and $2^{l_{\text{col}}} \geq M$. Then, $A \in \mathbb{R}^{N \times M}$ is extended to the $2^{l_{\text{row}}}$-by-$2^{l_{\text{col}}}$ matrix with additional rows and columns filled with zeros. Specifically, NeuKron sets l_{row} to $\lceil \log_2 N \rceil$ and set l_{col} to $\lceil \log_2 M \rceil$ so that the number of new entries is minimized.

Without loss of generality, we assume $N \leq M$ and thus $l_{\text{row}} \leq l_{\text{col}}$. If $l_{\text{row}} = l_{\text{col}}$, the extended square matrix is considered as the input and processed as described in Sections 4.1.2 and 4.1.3. Otherwise (i.e., if $l_{\text{row}} < l_{\text{col}}$), to encode the position (i,j) of a target entry a_{ij}, NeuKron first recursively divides $\mathbf{A}$ into 2×2 partitions, l_{row} times, to obtain a partition has a size of $1 \times 2^{l_{\text{col}} - l_{\text{row}}}$, and then it recursively divides the partition into two partitions of the same size (i.e., 1×2), $l_{\text{col}} - l_{\text{row}}$ times. Each division gives two partitions at left (L) and right (R). Specifically, the k-th entry of the sequence that encodes the position (i,j) is $(t_{\text{row}}(i,k), t_{\text{col}}(j,k))$, where $\forall d \in \{\text{row}, \text{col}\}$,

$$t_d(i,k) = \begin{cases} \left(\left\lfloor (i-1)/2^{l_d - k} \right\rfloor \bmod 2 \right) + 1, & \text{if } k \leq l_d, \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Example 2 (Encoding in Rectangular Matrices). *Suppose we encode the position $(2,3)$ of the non-square matrix in Figure 1(a), where $(l_{row}, l_{col}) = (2,3)$. The position $(2,3)$ is located in the top-left partition and in the bottom-right partition, respectively, in the first two divisions. In the last division, the position $(2,3)$ is located in the left one. Thus, the position $(2,3)$ is encoded in the sequence TL $\rightarrow$ BR $\rightarrow$ L, which becomes $(1,1) \rightarrow (2,2) \rightarrow (0,1)$ based on t_d (Eq. (3)).*

As in Section 4.1.3, NeuKron produces an approximation of a_{ij} using the (i, j)-th entry of the modified Kronecker product in Eq. (2) $\mathbf{K}_1 \otimes \cdots \otimes \mathbf{K}_{l_{\text{col}}}$. The only difference is that $\mathbf{K}_{l_{\text{row}}+1}, \cdots, \mathbf{K}_{l_{\text{col}}}$ are matrices of size 1×2, and for them, a separate embedding and linear layers are used, as described in Figure 1(c).

4.1.5 Comparison with Kronecker Graphs. Our model Θ generalizes the Kronecker graph model [27, 28] in two ways:

- While the Kronecker graph model uses the power of a single seed matrix, Θ uses the Kronecker product of potentially different matrices (i.e., $\mathbf{K}_1, \cdots, \mathbf{K}_l$) for approximation.
- In Θ, the matrices $\mathbf{K}_1, \cdots, \mathbf{K}_l$ may vary depending on the position of the target entry to be approximated. Specifically, $\tilde{a}_{ij}$ is computed using the (i, j)-th entry of $\mathbf{K}_1^{(f_1(i),f_1(j))} \otimes \mathbf{K}_2^{(f_2(i),f_2(j))} \otimes \cdots \otimes \mathbf{K}_l^{(f_l(i),f_l(j))}$, where $f_k(i) = \lfloor (i-1)/2^{l-k} \rfloor$.

This generalization leads to a significantly better trade-off between parameter size and approximation error in practice, as shown in Section 6.2. Notably, there are also two differences:

- While the Kronecker graph model is trained under a log-likelihood objective, Θ uses the squared Frobenius norm and normalizes the matrices to apply the tricks in Eq. (5) and Eq. (6).
- As specified in Eq. (2), each matrix (i.e., $\mathbf{K}_1, \cdots, \mathbf{K}_l$) is normalized and mapped onto the unit hypersphere.

4.2 Training Strategies

In this subsection, we propose novel training schemes for NeuKron's model Θ. We first present how to fit Θ to a given sparse reorderable matrix while exploiting its sparsity. Then, we present how to reorder the rows and columns of the input matrix so that Θ can be better fit to it. These two steps are alternated until convergence, as described in Algorithm 2. Below, we assume a matrix $\mathbf{A} \in \mathbb{R}^{N\times M}$ where $(N, M) = (2^{l_{\text{row}}}, 2^{l_{\text{col}}})$. As described in Section 4.1.4, a matrix of any size can be extended by zero-padding to satisfy this condition. We also assume $N \le M$, without loss of generality.

4.2.1 Update of row/column orders. It is crucial to properly order the rows and columns of a given reorderable matrix for NeuKron's model Θ better fit the matrix. This is because proper ordering reveals patterns (e.g., self-similarity and co-clusters), which Θ can exploit for accurate compression.

Overall process: For initialization, any co-clustering algorithms can be used. In our implementation, the matrix reordering scheme in [19] is used (see Section 6.3 for the effect of initialization). After initialization, NeuKron repeats (a) sampling two rows (or columns), (b) measuring the change in the approximation error (i.e., $\|\mathbf{A} - \tilde{\mathbf{A}}_\Theta\|_F^2$), and (c) determining whether to swap the sampled rows (or columns) or not probabilistically using the following criterion:

$$u < \exp(-\gamma \cdot \Delta), \tag{4}$$

where $u \sim U(0, 1)$, Δ is the change in the approximation error, and $\gamma > 0$ is a hyperparameter that controls the probability of accepting swaps that increase the approximation error.

Similarity-aware sampling: Below, we describe how NeuKron samples candidate pairs of rows (or columns) to be potentially swapped. Compared to a naive uniform sampling, the proposed sampling method has two advantages: **(a) effective**: it samples pairs based on the similarity of rows (or columns) so that swapping

Algorithm 2: Overall training process of NeuKron

```
Input: (a) a sparse reorderable matrix A
       (b) a number $T_p$ of permutation updates
Output: a NeuKron model $\Theta$
1 Initialize $\Theta$
2 while not converged do
3     for $k \leftarrow 1$ to $T_p$ do
4         $\mathbf{A} \leftarrow$ UpdateRowOrder($\mathbf{A}$)        ▷ Sect. 4.2.1
5         $\mathbf{A} \leftarrow$ UpdateColOrder($\mathbf{A}$)        ▷ Sect. 4.2.1
6     $\Theta \leftarrow$ UpdateModel($\mathbf{A}$, $\Theta$)        ▷ Sect. 4.2.2
7 return $\Theta$
```

the pairs is likely to reduce the approximation error, and **(b) easy-to-parallelize**: it samples disjoint pairs, which can be processed in parallel. The main idea is to select candidate pairs so that swapping pairs is likely to make similar rows (or columns) close to each other and thus to make them encoded in similar sequences in Section 4.1.2. Below, we describe the sampling method step by step for sampling row pairs. Column pairs are sampled similarly.

- **Estimating similarity:** In order to quickly estimate the similarity, min-hashing [5] is used. Specifically, for a uniform random bijective function $h_{\text{col}} : [M] \to [M]$ for the columns, the shingle $\min_{a_{ij}\neq 0}(h_{\text{col}}(j))$ of each i-th row is computed. It can be shown that two rows have the same shingle with probability proportional to the Jaccard similarity of the column indices of their non-zeros [5].
- **Locating similar rows/cols nearby:** We match rows with the same shingle disjointly, and for each matched rows, we sample pairs of rows to be swapped so that they are located in *nearby positions*, which we define as positions whose binary representations differ in only 1 bit. Let $p(i, k)$ be the position whose binary representation differs with that of i only in the k-th bit. Specifically, if two rows in the i_1-th and i_2-th positions are matched, we sample $(i_1, p(i_2, k))$ and $(i_2, p(i_1, k))$ so that i_1 and i_2 become nearby after swaps. The position $k \in [l_{\text{col}}]$ is sampled probabilistically (see Appendix B for details).
- **Pairing unmatched rows:** The rows remaining unmatched are randomly matched, and for each matched rows, we sample pairs as described above.

We describe the entire process of reordering for rows in Algorithm 3 in Appendix B.

4.2.2 Update of model parameters. The objective function of optimization is $\|\mathbf{A} - \tilde{\mathbf{A}}_\Theta\|_F^2$, as in Problem 1. Naively computing it takes $\Omega(NM \log M)$ time since all NM entries are approximated and approximating each entry takes $\Theta(\log M)$ time (see Theorem 1 in Section 4.3).

For its efficient computation, we reformulate the error as

$$\|\mathbf{A} - \tilde{\mathbf{A}}_\Theta\|_F^2 = \sum_{i=1}^{N}\sum_{j=1}^{M}(a_{ij} - \tilde{a}_{ij})^2 = \sum_{a_{ij}\neq 0}(a_{ij} - \tilde{a}_{ij})^2 \tag{5}$$

$$+ \sum_{a_{ij}=0}\tilde{a}_{ij}^2 = \sum_{a_{ij}\neq 0}\left((a_{ij} - \tilde{a}_{ij})^2 - \tilde{a}_{ij}^2\right) + \sum_{i=1}^{N}\sum_{j=1}^{M}\tilde{a}_{ij}^2.$$

In our model Θ, the last term, (i.e., the sum of squares) can be immediately computed from a learnable parameter $q \in \mathbb{R}^+$ (which is used in Eq. (2)), as formalized in Lemma 1.

Lemma 1. *For approximation by Eq.* (2), *Eq.* (6) *always holds.*

$$\sum_{i=1}^{2^{l_{row}}}\sum_{j=1}^{2^{l_{col}}}\tilde{a}_{ij}^2 = q^{l_{col}} \tag{6}$$

Proof. To prove this lemma, we use an induction. For $(l_{\text{row}}, l_{\text{col}}) = (1,1)$ and $(l_{\text{row}}, l_{\text{col}}) = (0,1)$, the statement holds trivially. Suppose the statement holds when $(l_{\text{row}}, l_{\text{col}}) = (0, l_2)$. For $(l_{\text{row}}, l_{\text{col}}) = (0, l_2+1)$, the statement also holds since

$$\begin{aligned}\sum_{i=1}^{2^{l_{\text{row}}}}\sum_{j=1}^{2^{l_2+1}}\tilde{a}_{ij}^2 &= \frac{q\mathbf{K}_1(1,1)^2}{\|\mathbf{K}_1\|_F^2}\sum_{i=1}^{2^{l_{\text{row}}}}\sum_{j=1}^{2^{l_2}}\frac{\tilde{a}_{ij}^2}{q\mathbf{K}_1(1,1)^2/\|\mathbf{K}_1\|_F^2} \\ &+ \frac{q\mathbf{K}_1(1,2)^2}{\|\mathbf{K}_1\|_F^2}\sum_{i=1}^{2^{l_{\text{row}}}}\sum_{j=2^{l_2}+1}^{2^{l_2+1}}\frac{\tilde{a}_{ij}^2}{q\mathbf{K}_1(1,2)^2/\|\mathbf{K}_1\|_F^2} \\ &= q^{l_2}\left(\frac{q\mathbf{K}_1(1,1)^2}{\|\mathbf{K}_1\|_F^2}+\frac{q\mathbf{K}_1(1,2)^2}{\|\mathbf{K}_1\|_F^2}\right) = q^{l_2}\cdot q = q^{l_2+1}\end{aligned}$$

Similarly, if the statement holds for $(l_{\text{row}}, l_{\text{col}}) = (l_1, l_2)$ and $l_1 \le l_2$, the statement also holds for $(l_{\text{row}}, l_{\text{col}}) = (l_1+1, l_2+1)$. By induction, the statement holds for all $0 \le l_{\text{row}} \le l_{\text{col}}$. □

This property follows from our careful design of Eq. (2), which is based on the Kronecker product. While q can be set so that the square sum of entries of $\tilde{\mathbf{A}}_{\Theta}$ is equal to that of $\mathbf{A}$, making it learnable leads to better compression since this gives more degrees of freedom to the model (see Section 6.3). As a result, the error becomes $\sum_{a_{ij}\neq 0}((a_{ij}-\tilde{a}_{ij})^2-\tilde{a}_{ij}^2)+q^{l_{col}}$, and thus the error and its gradient can be computed in time proportional to the number of non-zeros, without having to approximate zero entries in $\mathbf{A}$ explicitly (see Theorem 2 in Section 4.3). It should be noticed that we do use the loss function that encourages the model to fit to all entries including zeros, and we speed up its computation without changing it. Gradient descent is used for updating the model parameters.

Implementation in practice: Since candidate pairs are disjoint, processing them, including computing Eq. (4), is performed in parallel in our implementation. Shingles are also computed in parallel.

4.3 Theoretical Analysis

We analyze the time and space complexity of NeuKron. We assume that (a) $N \le M$ for the input matrix $\mathbf{A} \in \mathbb{R}^{N\times M}$ and (b) the dimension h of LSTM is a constant (i.e., $O(1)$), which is a user-defined hyperparameter. NeuKron requires logarithmic time for approximation (Theorem 1), as confirmed empirically in Section 2 of [1]. For training, it requires time proportional to the number of non-zero entries of $\mathbf{A}$, denoted by nnz($\mathbf{A}$) (Theorem 2).

Theorem 1 (Approximation Time for Each Entry). *The approximation of each entry by NeuKron takes* $\Theta(\log M)$ *time.*

Proof. First, we need to encode the position of the given entry. Since we need the subdivision $\Theta(\log M)$ times, the time complexity of the encoding step is $\Theta(\log M)$. The computational cost to approximate an entry only depends on the length of the input of the LSTM, so the time complexity for inference is $\Theta(\log M)$. □

Theorem 2 (Training Time). *Each training epoch in NeuKron takes* $O(nnz(\mathbf{A})\cdot\log M)$ *time.*

Proof. The time complexity for inference is $O(\log M)$ for each input. Thus, computing the approximation error takes $O(\text{nnz}(\mathbf{A})\cdot\log M)$ with Eq. (5) (see Lemma 1). The time complexity for computing the gradients is also $O(\text{nnz}(\mathbf{A})\cdot\log M)$, since the gradient of each component in the model, such as matrix multiplication and taking a non-linearity, does not require a greater time complexity. For optimizing the orders of rows and columns, computing the shingle values for rows and columns takes $O(\text{nnz}(\mathbf{A}))$ time since we need to look up all non-zero entries. Matching the rows and the columns as pairs requires $O(N+M)$ time. Only the entries of the output that correspond to non-zero entries are changed due to swaps and inference of a single element takes $O(\log M)$ time. Thus, checking the criterion in Eq. (4) takes $O(\text{nnz}(\mathbf{A})\cdot\log M)$ time. Therefore, the overall training time per epoch is $O(\text{nnz}(\mathbf{A})\cdot\log M)$. □

While NeuKron requires space proportional to the number of non-zero entries in the input matrix during training (Theorem 4), it gives a constant-size compression. (Theorem 3).

Theorem 3 (Space Complexity of Outputs). *The number of model parameters of NeuKron is* $\Theta(1)$.

Proof. In NeuKron, the number of the parameters for LSTM is $\Theta(h^2)$. The embedding layer before the LSTM and the linear layers after LSTM requires $\Theta(h)$ of parameters. The number of parameters for $\mathbf{K}_1$ in Algorithm 1 is 4, which is the number of entries. We consider $\Theta(h) = \Theta(1)$ as h is a constant; thus, the number of parameters is $\Theta(1)$. □

Theorem 4 (Space Complexity during Training). *NeuKron requires* $O(nnz(\mathbf{A})+M)$ *space during training.*

Proof. Refer to Appendix D. □

5 EXTENSION TO TENSORS

We extend NeuKron to sparse reorderable tensors. Theoretical analyses are available at Section 3 of [1].

5.1 Model

For a given D-order tensor $\mathcal{X} \in \mathbb{R}^{N_1\times\cdots\times N_D}$ (we assume $N_1 \le \cdots \le N_D$ without loss of generality), we first compute $l_i = \lceil \log_2 N_i \rceil$ for each $i \in [D]$ and extend $\mathcal{X}$ to the tensor of size $2^{l_1}\times\cdots\times 2^{l_D}$ with additional entries filled with zeros. As in Section 4.1.4, for encoding, NeuKron first recursively divides the extended tensor into 2^D partitions l_1 times to obtain a partition has a size of $1\times 2^{l_2-l_1}\times\cdots\times 2^{l_D-l_1}$. Then, it recursively divides the partition as it handles a $(D-1)$-order tensor. As a result, the k-th entry of the encoded sequence for the position $(i_1,\cdots i_D)$ is $(t_1(i_1,k),\cdots t_D(i_D,k))$, where t_d is identical to Eq. (3). We provide an example of NeuKron on an 3-order tensor in Figure 2. After encoding, NeuKron produces an approximation using the Kronecker product $\mathcal{K}_1\otimes\cdots\otimes\mathcal{K}_{l_D}$ from D linear layers for handling tensors of D different sizes.

5.2 Training Strategies

The main difference in training strategies lies in computing shingles. For a D-order tensor, D random bijective functions are used, thus each mode index has $D-1$ shingles from those functions except for the function of the same mode. In our extension, we match positions i_1 and i_2 as a pair only if the $D-1$ shingles of i_1 and those of index

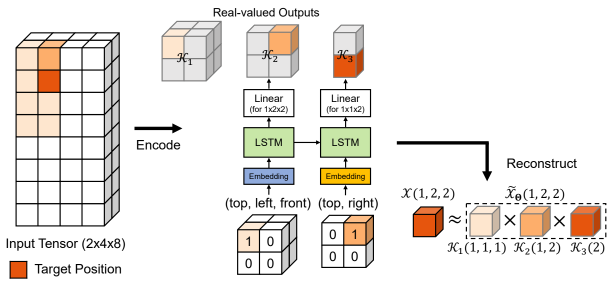


Figure 2: Example of NeuKron on an 3-order tensor $\mathcal{X}$.

j are all the same, and the orders of indices are randomly initialized. All other procedures are identical to the original NeuKron.

6 EXPERIMENTS

We conducted experiments to answer the following questions:

- Q1. **Compression Performance:** Does NeuKron perform more compact and accurate compression than its best competitors?
- Q2. **Ablation Study:** How effective are NeuKron's training strategies for the compression performance?
- Q3. **Scalability and Speed:** Does NeuKron scale linearly with the number of non-zero entries of an input data?
- Q4. **Approximation Analysis:** How does the approximation error of NeuKron vary depending on entry values?
- Q5. **Effects of Data Properties:** How do the skewness, order, and dimension of the input affect the approximation error?

The answers for Q3, Q4, and Q5 are provided in Appendix A.

6.1 Experiment Specifications

Machine: We ran experiments for NeuKron on a machine with 4 RTX 2080Ti GPUs and 128GB RAM. For competitors, which do not require GPUs, we ran experiments on a desktop with a 3.8GHz AMD Ryzen 3900X CPU and 128GB RAM. **Note that outputs and compression ratios do not depend on machine specifications.**

Datasets: We used six real-world matrices and four real-world tensors listed in Table 3. The matrices are essentially (weighted) bipartite graphs.[3] All the datasets are weighted (i.e., non-binary matrices and tensors) except for the email and threads datasets. Detailed semantics and structural properties of the datasets are provided in Table 3 and Table 7 of [1], respectively.

Competitors: For matrices, we compared NeuKron with KronFit [27], T-SVD (truncated SVD), CMD [39], ACCAMS [3], CUR [10], and bCCAMS [3]. In order to compress matrices of any size, we extended KronFit so that it (a) fits a non-square seed matrix, (b) permutes rows and columns separately, and (c) aims to minimize the approximation error in Problem 1. We did not consider methods designed for unipartite and/or unweighted graphs (e.g., [24, 25, 31]) as competitors since they are not applicable to most of the datasets. For tensors, we compared NeuKron with CP [2] and Tucker [21] decompositions and CSF [36], which is lossless. The competitors are described in Section 2, and see [1] for implementation details.

Experimental Setup: We trained NeuKron and its competitors under the following stopping condition with the patience of 100 epochs: $\frac{\mathcal{E}_{\min}-\mathcal{E}_{\text{curr}}}{\mathcal{E}_{\min}} < 10^{-5}$, where $\mathcal{E}_{\min}$ is the lowest approximation error so far, and $\mathcal{E}_{\text{curr}}$ is the current approximation error. For all experiments, we set T_p in Algorithm 2 to 2, and set γ in Eq. (4) to 10, after a preliminary study (see Section 6 of [1]). NeuKron was trained by Adam optimizer whose learning rate was set to 10^{-3} for the email and threads datasets, and 10^{-2} for the others. Unless otherwise stated, we set the hidden dimension h to 30 in the email, nyc, and tky datasets and to 60 in the kasandr, nips, and threads datasets. For the other datasets, we set h to 90. We ran all experiments 5 times with different random seeds and reported the average error. The setups for the competitors are depicted in [1].

[3]Nodes of one type in a bipartite graph correspond to rows, and nodes of the other type correspond to columns.

Table 3: Real-world datasets used in the paper. All datasets are publicly available, and links to them are available in [1].

Type	Name	Size	# of non-zeros
Matrix	email	$1,005 \times 25,919$	92,159
	nyc	$1,083 \times 38,333$	91,024
	tky	$2,293 \times 61,858$	211,955
	kasandr	$414,520 \times 503,702$	903,366
	threads	$176,445 \times 595,778$	1,457,727
	twitch	$790,100 \times 15,524,309$	234,422,289
Tensor	nips	$2,482 \times 2,862 \times 14,036$	3,101,609
	4-gram	$48K \times 54K \times 55K \times 58K$	7,495,550
	3-gram	$88K \times 100K \times 110K$	9,778,281
	enron	$5,699 \times 6,066 \times 244K$	31,312,375

6.2 Q1. Compression Performance

We compared the (a) size in bytes[4] and (b) approximation error of the compressed output obtained by the considered algorithms. We varied the hidden dimension h of NeuKron from 5 to 30 for the email, nyc, and tky datasets and from 10 to 60 for the kasandr, nips and threads datasets. For the others, we varied h from 15 to 90. Similarly, we varied the hyperparameters of each competitor as to reveal its trade-off between the size and error (refer to [1]).

For all datasets, NeuKron achieved the best trade-off between the approximation error and the compressed size. As seen in Figure 3, the size was up to **five orders of magnitude smaller** in NeuKron than in the competitors when their errors were similar. The error was also up to **10.1× smaller** in NeuKron than in the competitors when the outputs were of similar size. Note that the errors of KronFit do not always decrease as the number of parameters increases, as previously reported in [27].

Performance on non-reorderable data: NeuKron can also be applied to non-reorderable data if the mapping between the original and new orders of rows and columns are stored additionally. Even when we assume that the datasets are non-reorderable and consider the extra cost, NeuKron gives by far the best trade-off between size and approximation error, as shown in Figure 9.

6.3 Q2. Ablation Study

On the four smallest matrices and the two smallest tensors, we demonstrate the effectiveness of the components of NeuKron illustrated in Section 4 by comparing it with the following variants:

(a) NeuKron: the proposed method with all components.
(b) NeuKron-H (N-H): a variant that uniformly samples pairs of rows and columns without using min-**h**ashing.
(c) NeuKron-I (N-I): a variant that randomly **i**nitializes the orders of rows and columns without using the scheme in [19].

[4]In our implementation, each floating-point number took 4 bytes.

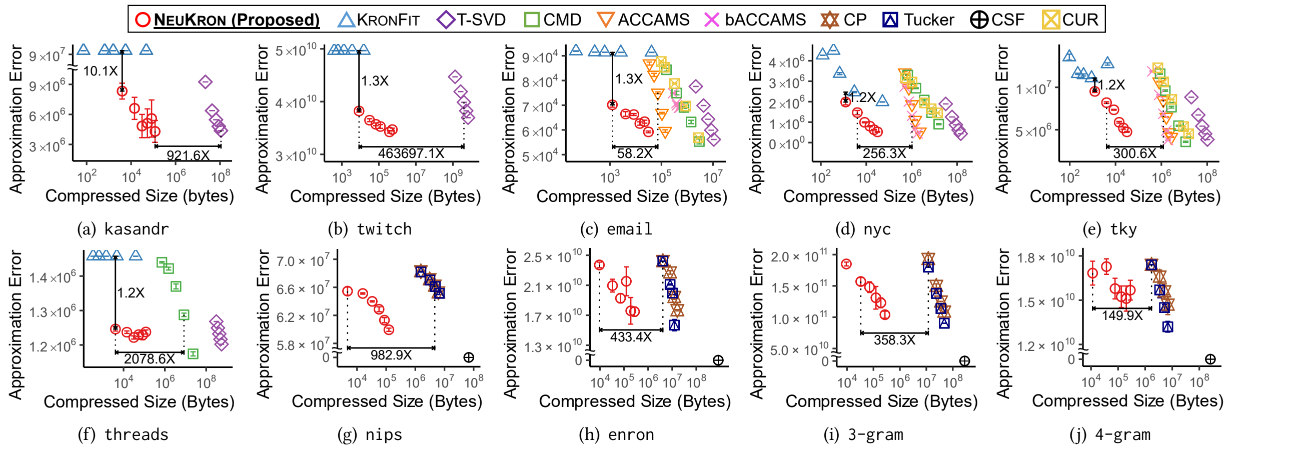
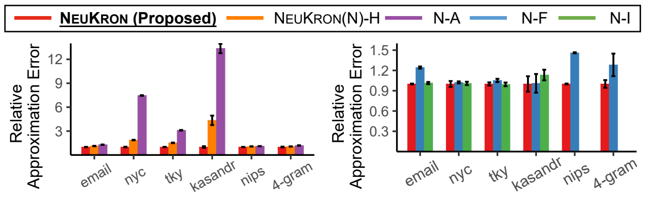


Figure 3: <u>NeuKron provides concise and accurate compressions.</u> The outputs of NeuKron are up to five orders of magnitude smaller that those of the competitors when the approximation errors in them are similar. When the sizes of the outputs are similar, the approximation error was up to 10.1× smaller in the outputs of NeuKron than those in the competitors. ACCAMS, bACCAMS, CUR, and CMD ran out of memory in some datasets, and their results do not appear in the corresponding plots. Note that the errors of KronFit do not always decrease as the number of parameters increase, as previously reported in [27].



(a) Effect of min-hashing and the auto-regressive architecture

(b) Effect of q and initialization

Figure 4: <u>Effectiveness of the components of NeuKron.</u> We report the approximation errors of variants relative to that of NeuKron. Results of NeuKron-I on tensors are omitted, since for tensors, NeuKron also randomly initializes orders.

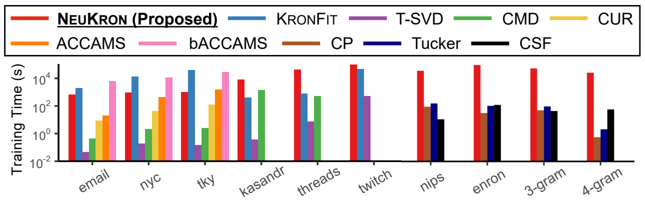


Figure 5: <u>Training time of NeuKron and the competitors.</u> Note that NeuKron requires much longer training time than many competitors, while it provides the best tradeoff between space and accuracy (Figure 3). See Appendix A.1 for detailed hyperparameter settings for each method.

(d) NeuKron-F (N-F): a variant that **f**ixes q to the sum of the squares of all entries in the input.
(e) NeuKron-A (N-A): a variant without any **a**uto-regressive architecture. It only uses two learnable matrices $\mathbf{K}_{\text{square}} \in \mathbb{R}^{2\times 2}$ and $\mathbf{K}_{\text{rect}} \in \mathbb{R}^{1\times 2}$, as in KronFit, to compute $\mathbf{K}_{\text{square}}^{\otimes l_{\text{row}}} \otimes \mathbf{K}_{\text{rect}}^{\otimes (l_{\text{col}}-l_{\text{row}})}$ for approximation. Similarly, it uses D learnable tensors to approximate N-order tensors.

As seen in Figure 4, NeuKron outperformed NeuKron-A and NeuKron-H, which indicates that the auto-regressive architecture (i.e., LSTM) and the min-hashing technique are crucial to enhance the performance of NeuKron. Moreover, making q learnable was effective especially on the `email`, `nips`, and `4-gram` datasets. For the order initialization, NeuKron-I showed comparable or slightly poor performance than NeuKron, implying that how the rows and columns are initialized can affect the compression quality.

<u>Extra Results:</u> For details results regarding Q3-Q5, refer to Appendix A. Importantly, the training time of NeuKron is significantly longer than that of many competitors, as shown in Figure 5.

7 CONCLUSION

In this work, we focus on compressing sparse reorderable matrices and tensors, including the adjacency matrices of bipartite graphs, into a constant-size space. Our contributions are three-fold:

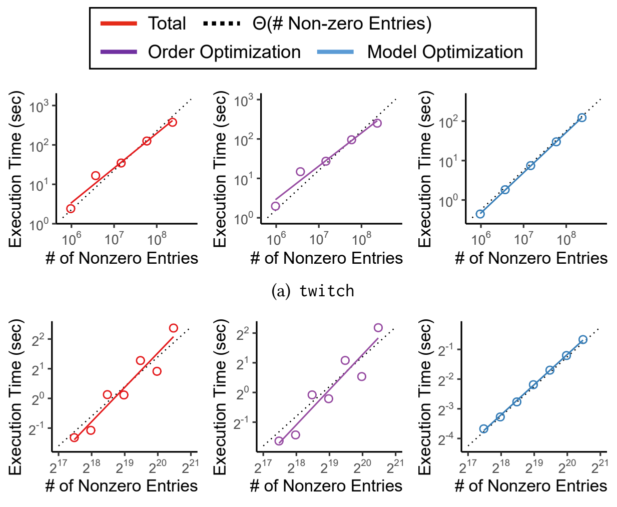
- **Compact yet Accurate Method:** We proposed NeuKron, which lossily compresses matrices and fixed-order tensors of any size with a constant number of parameters. NeuKron provided an output that is up to five orders of magnitude smaller than the outputs of the best competitors when the approximation errors in them are similar (Figure 3).
- **Theoretical Analysis:** We carefully designed NeuKron so that, for sparse reorderable matrices and fixed-order tensors of any size (a) the number of parameters is constant, (b) each entry is approximated in a logarithmic time, and (c) the model is fitted to an input in time proportion to the number of non-zero entries in it. We proved these desirable properties (Theorems 1-3).
- **Extensive Experiments:** Through extensive experiments on 10 real-world datasets, we demonstrated the effectiveness and scalability of NeuKron (Figures 3 and 6). Especially, we showed that NeuKron successfully compressed a matrix with up to 230 millions non-zero entries.

<u>Future directions:</u> We expect that NeuKron can be used for web-related applications, including fraud detection in online social networks [32] and recommendation [17], where lossy matrix/tensor compression (e.g., T-SVD) has been successful. These applications can be considered as future research directions.

<u>Reproducibility:</u> The code and datasets used are available at [1].

REFERENCES

[1] 2022. *Code, Datasets, and Appendix (Anonymous).* https://anonymous.4open.science/r/22-NeuKron-6D50
[2] Brett W Bader and Tamara G Kolda. 2008. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231.
[3] Alex Beutel, Amr Ahmed, and Alexander J Smola. 2015. Accams: Additive co-clustering to approximate matrices succinctly. In *WWW*.
[4] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW*.
[5] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. 2000. Min-wise independent permutations. *JCSS* 60, 3 (2000), 630–659.
[6] J Douglas Carroll and Jih-Jie Chang. 1970. Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition. *Psychometrika* 35, 3 (1970), 283–319.
[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *SDM*.
[8] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *EMNLP* (2014).
[9] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing graphs and indexes with recursive graph bisection. In *KDD*.
[10] Petros Drineas, Ravi Kannan, and Michael W Mahoney. 2006. Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix. *SIAM Journal on computing* 36, 1 (2006), 158–183.
[11] Petros Drineas, Michael W Mahoney, and Shan Muthukrishnan. 2008. Relative-error CUR matrix decompositions. *SIAM J. Matrix Anal. Appl.* 30, 2 (2008), 844–881.
[12] Carl Eckart and Gale Young. 1936. The approximation of one matrix by another of lower rank. *Psychometrika* 1, 3 (1936), 211–218.
[13] Gene H Golub and Christian Reinsch. 1971. Singular value decomposition and least squares solutions. In *Linear algebra*. Springer, 134–151.
[14] Wolfgang Hackbusch. 2012. *Tensor spaces and numerical tensor calculus*. Vol. 42. Springer.
[15] Ruining He and Julian McAuley. 2016. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*.
[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[17] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *ICDM*.
[18] Nam Hyeon-Woo, Moon Ye-Bin, and Tae-Hyun Oh. 2022. FedPara: Low-rank Hadamard Product for Communication-Efficient Federated Learning. In *ICLR*.
[19] Jinhong Jung and Lee Sael. 2020. Fast and accurate pseudoinverse with sparse matrix reordering and incremental approach. *Machine Learning* 109, 12 (2020), 2333–2347.
[20] Alexandros Karatzoglou, Xavier Amatriain, Linas Baltrunas, and Nuria Oliver. 2010. Multiverse recommendation: n-dimensional tensor factorization for context-aware collaborative filtering. In *RecSys*.
[21] Tamara G Kolda and Jimeng Sun. 2008. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*.
[22] Jakub Konečnỳ, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. 2016. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527* (2016).
[23] Timothée Lacroix, Nicolas Usunier, and Guillaume Obozinski. 2018. Canonical tensor decomposition for knowledge base completion. In *ICML*.
[24] Kyuhan Lee, Hyeonsoo Jo, Jihoon Ko, Sungsu Lim, and Kijung Shin. 2020. Ssumm: Sparse summarization of massive graphs. In *KDD*.
[25] Kristen LeFevre and Evimaria Terzi. 2010. GraSS: Graph structure summarization. In *SDM*.
[26] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. 2005. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *ECML/PKDD*.
[27] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker graphs: an approach to modeling networks. *JMLR* 11, 2 (2010).
[28] Jure Leskovec and Christos Faloutsos. 2007. Scalable modeling of real graphs using kronecker multiplication. In *ICML*.
[29] Chao Liu, Fan Guo, and Christos Faloutsos. 2009. Bbm: bayesian browsing model from petabyte-scale data. In *KDD*.
[30] Julian McAuley, Jure Leskovec, and Dan Jurafsky. 2012. Learning attitudes and attributes from multi-aspect reviews. In *ICDM*.
[31] Matteo Riondato, David García-Soriano, and Francesco Bonchi. 2017. Graph summarization with quality guarantees. *DMKD* (2017).
[32] Neil Shah, Alex Beutel, Brian Gallagher, and Christos Faloutsos. 2014. Spotting suspicious link behavior with fbox: An adversarial perspective. In *ICDM*.
[33] Kijung Shin, Amol Ghoting, Myunghwan Kim, and Hema Raghavan. 2019. Sweg: Lossless and lossy summarization of web-scale graphs. In *WWW*.
[34] Sumit Sidana, Charlotte Laclau, Massih R Amini, Gilles Vandelle, and André Bois-Crettez. 2017. KASANDR: a large-scale dataset with implicit feedback for recommendation. In *SIGIR*.
[35] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. 2015. An Overview of Microsoft Academic Service (MAS) and Applications. In *WWW*.
[36] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. 1–7.
[37] Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 61–70.
[38] Gilbert W Stewart. 1993. On the early history of the singular value decomposition. *SIAM review* 35, 4 (1993), 551–566.
[39] Jimeng Sun, Yinglian Xie, Hui Zhang, and Christos Faloutsos. 2007. Less is more: Compact matrix decomposition for large sparse graphs. In *SDM*.
[40] Daniel Ting. 2018. Count-min: Optimal estimation and tight error bounds using empirical error distributions. In *KDD*.
[41] Ledyard R Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31, 3 (1966), 279–311.
[42] Nick Vannieuwenhoven, Raf Vandebril, and Karl Meerbergen. 2012. A new truncation strategy for the higher-order singular value decomposition. *SIAM Journal on Scientific Computing* 34, 2 (2012), A1027–A1052.
[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*.
[44] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *ICDM*.



(a) `twitch`

(b) `threads`

Figure 6: <u>The training process of NeuKron is scalable.</u> Both model and order optimizations scale near-linearly with the number of non-zeros in the input.

A ADDITIONAL EXPERIMENTAL RESULTS

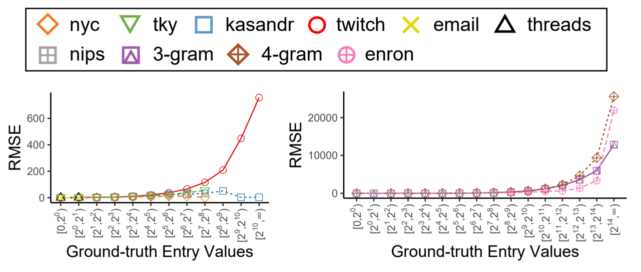
A.1 Q3. Scalability and Speed

In order to evaluate the scalability of NeuKron, we generated multiple matrices of various sizes from the `threads` and `twitch` datasets by tracking their evolutions over time. In them, we measured the training time per epoch for model and order optimizations in addition to total training time per epoch. The hidden dimension h was fixed to 60. As shown in Figure 6, the individual and overall training processes of NeuKron scaled **linearly with the number of non-zeros**, which is consistent with the theoretical results in Section 4.3. We further confirmed the linear scalability of NeuKron on tensor datasets and in hidden dimensions in Section 8 of [1].

We compared the training time of NeuKron and the competitors in Figure 5. We followed the hyperparameter settings in Section 6.2. For NeuKron, we reported the result with the smallest hidden dimensions that we considered. For all competitors except for KronFit, we reported their results when their approximation errors are closest to that of NeuKron. For KronFit, we reported its result when its output size is closest to that of NeuKron. Since our optimization problem is a mixed discrete-continuous optimization problem, which is notoriously difficult, the convergence of NeuKron takes much longer than that of factorization-based methods. While the convergence took long, the approximation error dropped rapidly in early iterations in most cases. The detailed training curves are given in Figure 2 of [1].

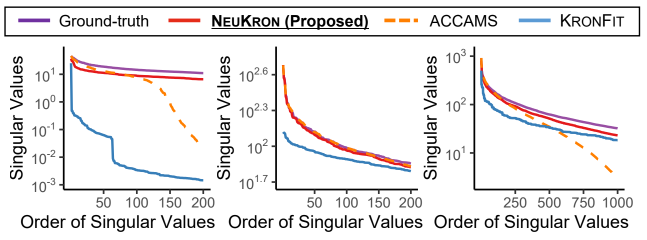
A.2 Q4. Approximation Analysis

We analyzed how the approximation error by NeuKron varies depending on the ground-truth value of approximated entries. In each dataset, we grouped the approximated entries by log-binning of their ground-truth values, and for each group, we computed the root mean squared error (RMSE) of the approximation errors. As seen in Figure 7, RMSE tended to increase with respect to ground-truth entry values. We also checked at most 1,000 largest singular values of matrices obtained by NeuKron and the two strongest competitors. For each method, we used the hyperparameter settings that led to the least approximation error in Figures 3 and 9. As seen in Figure 8, the singular values obtained by NeuKron were closest to the singular values of the input matrices.



(a) RMSE on sparse matrices (b) RMSE on sparse tensors

Figure 7: <u>Analysis of approximation errors of NeuKron.</u> The errors tend to increase with respect to the ground truth values of approximated entries. The x-axis is in log scale.



(a) `email` (b) `nyc` (c) `tky`

Figure 8: <u>NeuKron preserves singular values well.</u> The singular values of the matrix obtained by NeuKron are closest to the ground-truth ones. We used the smallest datasets for this experiment since computing singular values requires approximating all entries, including zeros.

A.3 Q5. Effects of Data Properties

We investigated the effects of properties of an input tensor $\mathcal{X}$ on the performance of NeuKron. For this experiment, we synthetically generated tensors using the multi-dimensional extension R-MAT [7]. Specifically, we first split each mode of a tensor into two partitions, and then chose either the first partition with probability p or the second one with probability $1-p$. This process is repeated until the target position is determined. As a default setting, we set (a) p to 0.8, (b) the order D to 3, (c) the sum of all tensor entries to 10^6, and (d) the number of entries to 2^{30}. We measured *fitness*, which is defined as $1 - \|\mathcal{X} - \tilde{\mathcal{X}}_\Theta\|_F / \|\mathcal{X}\|_F$ (the higher, the better). The fitness is widely used to compare the errors of approximations to different tensors. We varied the skewness p from 0.65 to 0.85. Note that increasing p makes the distribution of non-zero entries more skewed with distinct patterns, and decreasing p makes the distribtuion more uniform without patterns. As seen in Figure 10(a), the fitness increased as p increased, implying that NeuKron provides better performance on skewed tensors with distinct patterns. Next, we changed the order D from 2 to 6, but no significant effect of D was observed, as shown in Figure 10(b). Lastly, we analyzed the effect of dimension (i.e., the number of indices in each mode) by changing the dimension of tensors while fixing the number of non-zeros. As seen in Figure 10(c), the fitness of NeuKron decreased as

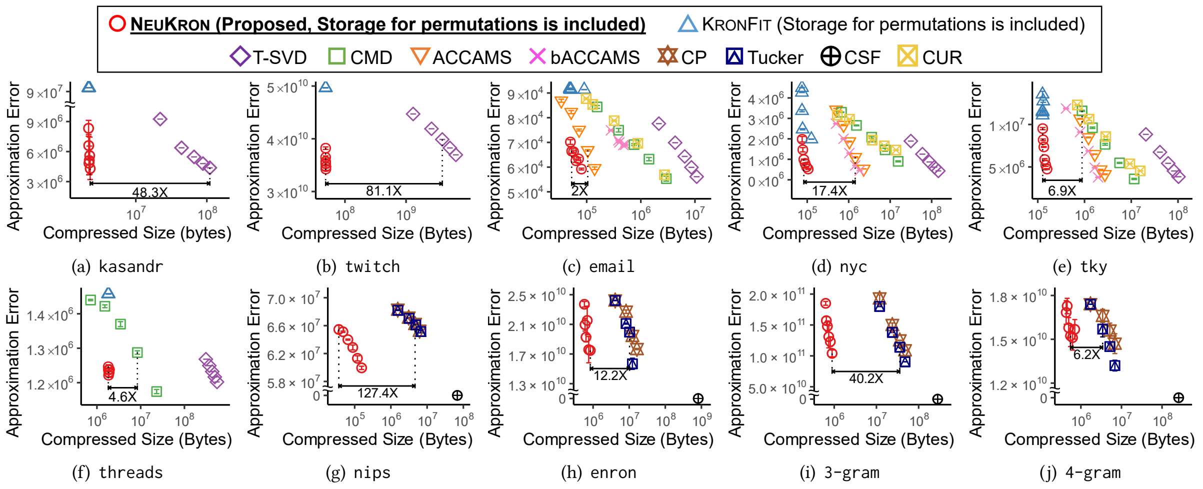


Figure 9: <u>NeuKron significantly outperforms the competitors even if we assume that matrices and tensors are *non-reorderable* and separately store the permutations of indices for all modes.</u> Note that the outputs of NeuKron require up to two orders of magnitude smaller space than those of the competitors with similar approximation error.

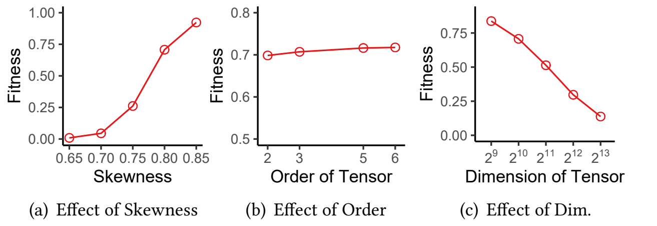


Figure 10: <u>Effects of data properties on NeuKron.</u> (a) The fitness increases as the skewness p increases. (b) The order of tensors does not significantly affect the fitness. (c) As tensors become bigger, the fitness decreases.

tensors became bigger and thus more entries were approximated by a fixed number of parameters.

B PSEUDOCODE FOR ORDERING ROWS

The pseudocode of UpdateRowOrder described in Section 4.2.1, is given in Algorithm 3, where binary representations start from 0, while row indices start from 1.

C EFFECTIVENESS OF NEUKRON ON NON-REORDERABLE DATA

NeuKron can also be applied to non-reorderable matrices and tensors if the mapping between the original and new orders of mode indices are stored additionally. Even with this additional space requirement, NeuKron still yielded the best trade-off between the approximation error and the compressed size, as seen in Figure 9, where we assume that the input matrices and tensors are not reorderable. Especially, the compression size of NeuKron was two orders of magnitude smaller than those of the competitors with similar approximation error on the `twitch` and `nips` datasets. Remark that KronFit also needs space for storing orders when it is applied to non-reorderable matrices.

Algorithm 3: UpdateRowOrder

```
Input: (a) a reorderable matrix $\mathbf{A}$, (b) a hyperparameter $\gamma$ in Eq. (4)
Output: updated matrix $\mathbf{A}$
1  Sample $k \in \mathbb{N} \cup \{0\}$ so that $\mathrm{P}(k = i) = 1/2^{i+1}$
2  $k \leftarrow \min(k_{\text{row}}, l_{\text{row}} - 1)$; $R \leftarrow \emptyset$; $P \leftarrow \emptyset$
3  Generate a random hash bijective functions $h_{\text{col}}$
4  foreach $i \in \{i \in [n] : (i-1) \text{ AND } 2^k = 0\}$ do
5      $u \sim U(0, 1)$
6      if $u < 1/2$ then $R \leftarrow R \cup \{i\}$
7      else $R \leftarrow R \cup \{i + 2^k\}$
8  foreach $i \in R$ do
9      $f_{row}(i) \leftarrow \min_{a_{ij} \neq 0}(h_{\text{col}}(j))$
10 while $\exists (i_1, i_2)$ s.t. $f_{row}(i_1) = f_{row}(i_2)$ do
11     $P \leftarrow P \cup \{(i_1, (i_2 - 1) \text{ XOR } 2^k + 1), (i_2, (i_1 - 1) \text{ XOR } 2^k + 1)\}$
12     $R \leftarrow R \setminus \{i_1, i_2\}$
13 $R \leftarrow R \cup \{(r - 1) \text{ XOR } 2^{k_{\text{row}}} + 1 : r \in R\}$
14 while $R \neq \emptyset$ do
15     Randomly sample $(i_1, i_2)$ from $R$
16     $P \leftarrow P \cup \{(i_1, i_2)\}$;  $R \leftarrow R \setminus \{i_1, i_2\}$
17 $P_{\text{accept}} \leftarrow \emptyset$
18 foreach $(i_1, i_2) \in P$ do
19     $u \sim U(0, 1)$
20     $\Delta \leftarrow$ change in the approximation error
21     if $u \geq exp(-\gamma \cdot \Delta)$ then $P_{\text{accept}} \leftarrow P_{\text{accept}} \cup \{(i_1, i_2)\}$
22 foreach $(i_1, i_2) \in P_{accept}$ do
23     $\mathbf{A}_{i_1,:}, \mathbf{A}_{i_2,:} \leftarrow \mathbf{A}_{i_2,:}, \mathbf{A}_{i_1,:}$
```

D PROOF OF THEOREM 4

Proof. Storing the input matrix in a sparse format requires $O(\text{nnz}(\mathbf{A}))$ space. Changing the orders of rows and columns requires $O(M + N)$ space for saving random hash functions, shingles, and changes of losses. If we assume that the batch size and the number of parameters of LSTM as a constant, its memory usage during inference and backpropagation is $O(\log M)$ since the input length is $O(\log M)$. Thus, the overall space complexity during training is $O(\text{nnz}(\mathbf{A}) + M)$. □