ELiCiT: Effective and Lightweight Lossy Compression of Tensors

Jihoon Ko[1], Taehyung Kwon[1], Jinhong Jung[2], and Kijung Shin[1]
[1]Kim Jaechul Graduate School of AI, KAIST, [2]School of Software, Soongsil University
jihoonko@kaist.ac.kr, taehyung.kwon@kaist.ac.kr, jinhong@ssu.ac.kr, kijungs@kaist.ac.kr

***Abstract*—Many real-world data from various domains can be represented as tensors, and a significant portion of them is large-scale. Thus, tensor compression is crucial for their storage and transmission. Recently, deep-learning-based methods have emerged to enhance compression performance. However, they require considerable compression time to fulfill their performance. In this work, to achieve both speed and performance, we develop ELICIT, an effective and lightweight lossy tensor compression method. When designing ELICIT, we avoid deep auto-regressive neural networks and index reordering, which incur high computational costs of deep-learning-based tensor compression. Specifically, instead of using the orders of indices as parameters, we introduce a feature-based model for indices, which enhances the model's expressive capacity and simplifies the overall end-to-end training procedure. Moreover, to reduce the size of the parameters and computational cost for inference, we adopt end-to-end clustering-based quantization, as an alternative to deep auto-regressive architecture. As a result, ELICIT becomes easy to optimize with enhanced expressiveness. We prove that it (partially) generalizes deep-learning-based methods and also traditional ones. Using eight real-world tensors, we show that ELICIT yields compact outputs that fit the input tensor accurately. Compared to the best competitor with similar fitness, it offers 1.51-5.05× smaller outputs. Moreover, compared to deep-learning-based compression methods, ELICIT is 11.8-96.0× faster with 5-48% better fitness for a similarly sized output. We also demonstrate that ELICIT is extended to matrix completion and neural network compression, providing the best trade-offs between model size and application performance.**

***Index Terms*—Tensor, Data Compression, Matrix Completion, Neural-network Compression**

I. INTRODUCTION

Many real-world data from various domains are naturally represented as tensors. Examples include predicates in knowledge bases structured as (subject, verb, object) [1], e-commerce purchases in the format of (user, item, time), and traffic histories structured as (source, destination, time). Neural network parameters are also structured in tensor formats. For instance, the weights of a fully-connected layer are represented as a matrix (i.e., 2-order tensor), and the weights of a 2D convolutional layer are in the format of a 4-order tensor.

These real-world tensors can be large-scale, and their sizes have been expanding over time. For instance, e-commerce purchase histories for hundreds of millions of users have accumulated, and large language models (LLMs) [2], [3], whose parameters are composed of tensors, typically have more than billions of parameters.

As the size increases, the expenses of storing and transmitting tensors also increase; consequently, tensor compression becomes increasingly essential to mitigate these costs. Especially, since tensors are widely used for various types of data, general tensor compression methods that do not rely on specific data property assumptions are increasingly important. For instance, video compression methods, which heavily depend on the continuity of video data [4], [5], cannot be applied to tensors in the above examples.

Decomposition-based methods [6]–[10] have commonly been employed for lossy compression of general tensors. For matrices (i.e., 2-order tensors), truncated singular value decomposition (T-SVD) [6] has been widely adopted, and for tensors, CP [8] and Tucker [7] Decompositions have been popular. These methods are not only useful for tensor compression but have also been used for other applications, including tensor completion [11]–[13] and neural-network compression [14]–[16]. Recently, several deep-learning-based tensor-compression methods [17], [18] have been proposed to achieve better compression performance.

NEUKRON [17] and TENSORCODEC [18] extend the Kroncker graph model [19] and Tensor-Train Decomposition (TTD) [9], respectively, and they are dependent on the order of mode indices (e.g., rows and columns). Moreover, they commonly employ a deep auto-regressive neural network to keep the number of parameters manageable. While these methods achieve unprecedented compression performance, they suffer from long compression time attributed to their reliance on heavyweight neural networks and order optimization. Specifically, their compression time can be up to five orders of magnitude slower than decomposition-based methods, as demonstrated in Section VI.

Are deep learning techniques really indispensable for achieving superior compression performance? Can we accelerate tensor compression while maintaining or even improving compression performance? In this work, we propose ELICIT (**E**ffective and **Li**ghtweight Lossy **C**ompress**i**on of **T**ensors) as an answer to these questions. In the development of ELICIT, we refrain from using deep auto-regressive neural networks and order optimization, which incur high computational costs of the aforementioned deep-learning-based tensor-compression methods. Specifically, ELICIT is based on a feature-based model for indices that is independent of the orders of indices. Moreover, ELICIT reparameterizes the model with a manageable number of parameters through end-to-end clustering-based quantization, instead of relying on deep auto-regressive neural networks. As a result, ELICIT becomes easily optimized in an end-to-end manner using gradient descent with enhanced expressiveness. Specifically, we prove that its design

TABLE I
FREQUENTLY-USED SYMBOLS.

Symbol	Description
$\mathcal{X} \in \mathbb{R}^{N_1 \times \cdots \times N_d}$	tensor where d is the order of $\mathcal{X}$
$\mathcal{X}(i_1, \cdots, i_d)$	$(i_1, \cdots, i_d)$-th entry of $\mathcal{X}$
$[n] = \{1, \cdots, n\}$	set of consecutive integers from 1 to n
$\tilde{\mathcal{X}}$	approximated tensor of $\mathcal{X}$
r	number of latent features
$\mathbf{F}^{(j)}(i_j) \in [0,1]^r$	feature vector of index i_j of the j-th mode
$\mathbf{F}_k^{(j)}(i_j) \in [0,1]$	k-th feature of index i_j of the j-th mode
$\mathbf{F}(i_1, \cdots, i_d) \in [0,1]^{r \times d}$	feature matrix of $\mathcal{X}(i_1, \cdots, i_d)$
$\mathbf{F}_k(i_1, \cdots, i_d) \in [0,1]^d$	k-th feature vector of $\mathcal{X}(i_1, \cdots, i_d)$
$\mathbf{s}_l \in \{0,1\}^d$	l-th reference state
$v_{k,l} \in \mathbb{R}$	value of s_l for k-th feature
$g(\cdot)$	reduce function for approximation
q	quantization level
$c_{k,l}^{(j)} \in [0,1]$	l-th candidate for k-th feature and j-th mode

(partially) generalizes the aforementioned deep-learning-based methods [17], [18] and also CP Decomposition [8]. Furthermore, we extend ELiCiT's applicability to matrix completion and neural-network compression.

We evaluate ELiCiT using eight real-world tensors from various application domains. Our extensive experiment results reveal the following advantages of ELiCiT:

- **Compact and Accurate:** It consistently achieves a better trade-off between compressed size and approximation error than all considered competitors. Specifically, ELiCiT compresses tensors to sizes 1.51-5.05× smaller than competitors while achieving similar fitness. It also achieves 5-48% better fitness than competitors with similar output sizes.
- **Fast:** While giving similar outputs with better fitness, ELiCiT is 11.8-96.0× faster than deep-learning methods.
- **Applicable:** It is successfully applied to matrix completion and neural network compression, providing a better trade-off between model size and application performance, compared to state-of-the-art competitors for these applications.

Reproducibility: The code and datasets used in the paper and the supplementary material are available at https://github.com/jihoonko/icdm24-elicit.

II. Preliminaries

In this section, we introduce some important notations and concepts. Then, we formulate the tensor compression problem. Some frequently-used symbols are listed in Table I.

A. Notations

Tensor: A *tensor* $\mathcal{X} \in \mathbb{R}^{N_1 \times \cdots \times N_d}$ is a multi-dimensional array of real numbers, where d denotes the order of $\mathcal{X}$, i.e., the dimension of the array. A *matrix* $\mathbf{A} \in \mathbb{R}^{N_1 \times N_2}$ is the special case of a tensor where $d = 2$. We use $\mathcal{X}(i_1, \cdots, i_d)$ to represent the value in the $(i_1, \cdots, i_d)$-th position of $\mathcal{X}$.
Frobenius norm: The *Frobenius norm* $\|\mathcal{X}\|_F$ of $\mathcal{X}$ is defined as the squared root of the squared sum of all the entries in $\mathcal{X}$.
Approximation error: The *approximation error* of the compressed model Θ on the input tensor $\mathcal{X}$ is defined as $\|\mathcal{X} - \tilde{\mathcal{X}}\|_F^2$, where $\tilde{\mathcal{X}} \in \mathbb{R}^{N_1 \times \cdots \times N_d}$ is the approximation of $\mathcal{X}$ from Θ. The *fitness* of $\tilde{\mathcal{X}}$ on $\mathcal{X}$ is defined as $1 - \|\mathcal{X} - \tilde{\mathcal{X}}\|_F / \|\mathcal{X}\|_F$. Note that maximizing the fitness is equivalent to minimizing the approximation error.

B. Problem Formulation

We define the lossy tensor compression problem as follows:

Problem 1. (Lossy Compression of a Tensor)
- ***Given:*** *a tensor* $\mathcal{X} \in \mathbb{R}^{N_1 \times \cdots \times N_d}$,
- ***Find:*** *the compression model* Θ *(which leads to* $\tilde{X}$*)*
- ***to Minimize:*** *(1) the number of the parameters of* Θ
  (2) the approximation error $\|\mathcal{X} - \tilde{\mathcal{X}}\|_F^2$.

III. Related Work

In this section, we review previous studies for compressing matrices and tensors. Their **applications to matrix completion and neural-network compression** are reviewed in Appendix A of [20].
Compression via Low-rank Decomposition: Truncated Singular Value Decomposition (T-SVD) [6], [21] is one of the most popular decomposition methods for lossy matrix compression. For the input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and a desired rank $r < \text{rank}(\mathbf{A})$, T-SVD compresses $\mathbf{A}$ into $\mathbf{U}\Sigma\mathbf{V}^\top$ where $\Sigma \in \mathbb{R}^{r \times r}$ is the diagonal matrices whose diagonal entries are top-r largest singular values, and $\mathbf{U} \in \mathbb{R}^{m \times r}$ and $\mathbf{V} \in \mathbb{R}^{n \times r}$ consist of the corresponding left singular vectors and right singular vectors, respectively. T-SVD provides the optimal rank r approximation to $\mathbf{A}$ in terms of the Frobenius norm.

While T-SVD is specifically designed for matrices (or 2-order tensors), CP Decomposition (CPD) [8] and Tucker Decomposition (TKD) [7] are designed for higher-order tensors, extending the concept of SVD to tensors. Given the tensor $\mathcal{X} \in \mathbb{R}^{N_1 \times \cdots \times N_d}$ and a specified rank r, CPD compresses $\mathcal{X}$ into the sum of r rank-1 tensors, where each of these tensors is represented as the outer product of d vectors, each with dimensions $N_1, \cdots, N_d$. TKD factorizes $\mathcal{X}$ into a small core tensor $\mathcal{T} \in \mathbb{R}^{r_1 \times \cdots \times r_d}$ and factor matrices $\mathbf{U}_1, \cdots, \mathbf{U}_d$, where r_k is the rank of k-th mode and $\mathbf{U}_k$ has dimensions $N_k \times r_k$. Tensor-train Decomposition (TTD) [9] overcomes several drawbacks [9], [22], [23] of CPD and TKD. TTD compresses $\mathcal{X}$ by approximating each $(i_1, \cdots, i_d)$-th entry in $\mathcal{X}$ by the product of a sequence of matrices assigned to each mode index. Tensor Ring Decomposition (TRD) [10] also approximates $\mathcal{X}$ by computing the traces of the inner products of the small matrices.
Compression via Co-clustering: ACCAMS [13] employs an approximation technique involving the combination of small co-clusters to approximate matrices. Specifically, for each iteration, it identifies co-clusters in a given input matrix $\mathbf{A}$, and then it assigns a value to each entry depending on which co-cluster the entry is located. It repeats the procedure, and for approximation, it uses the summation of the assigned values during iterations. Similarly to T-SVD, it is specialized for matrices and cannot be applied to higher-order tensors. From a feature-based clustering perspective, ELiCiT and ACCAMS share some similarities, as further discussed in Appendix D.
Compression using Deep Learning: Deep-learning-based methods have been recently proposed for better compression performance. For compressing sparse reorderable tensors, NeuKron [17] generalizes the Kronecker model [19] for

enhanced expressiveness. It further introduces an index reordering technique, leveraging min-hashing on the input tensor, for better performance. However, it is designed for non-negative sparse tensors, showing lower performance on dense tensors with positive and negative values. To address the issue, Kwon et al. [18] extended TTD with an index reordering algorithm suitable for such general tensors. Notably, both generalized models are reparameterized as the output of deep auto-regressive neural networks (spec., LSTM) to reduce the number of parameters (i.e., size of compressed outputs). While these methods exhibit exceptional compression performance, compared to the aforementioned traditional methods, they are time-consuming for compression and leave room for improvement in terms of expressive power, as discussed in detail in Sections IV-A and IV-E.

Completion using Deep Learning: While they are not initially designed for compression, we can adapt deep-learning-based tensor completion models for compression by fitting them to training data, without regularization. For efficiency with fewer parameters, CostCo [24] approximates each entry using two 1D convolutional layers and fixed-size features for each mode index. M^{2}DMTF [25] decomposes tensors similarly to TKD but determines each factor matrix for each mode using individual neural networks with fewer parameters, while TKD uses a single relatively large factor matrix, instead.

IV. Proposed Method

In this section, we present ELiCiT, our proposed model for **E**ffective and **Li**ghtweight Lossy **C**ompress**i**on of **T**ensors.

A. Motivations

Deep-learning-based methods, specifically NeuKron [17] and TensorCodec [18], exhibit remarkable tensor compression performance (i.e., balance between compressed size and approximation error). However, they possess several limitations due to their model design and training procedure.

L1. **Limited Expressiveness of Order-dependent Models.** The models rely on the order of mode indices (e.g., rows and columns) within the input tensor. Essentially, they recursively divide mode indices into equal-sized groups based on their orders (see Appendix B of [20] for details). However, dividing mode indices into equally sized groups may not be optimal (e.g., consider dividing locations in NYC in the NYC dataset; see Section VI-A).

L2. **High Computational Cost of Heavyweight Design.** They reparameterize their models using deep auto-regressive neural networks, particularly LSTM. While this is effective in reducing the overall parameter size (i.e., the size of compressed outputs), it requires heavy computational costs during training.

L3. **Complexity of Order Optimization.** Since the models are dependent of the orders of mode indices, it is crucial to optimize the orders for enhanced approximation. However, order optimization is notoriously challenging, since the orders cannot be optimized concurrently with other parameters. Thus, they must undergo separate optimization, resulting in inefficiency.

Algorithm 1: Overview of ELiCiT and qELiCiT

```
Input: a tensor $\mathcal{X} \in \mathbb{R}^{N_1 \times \cdots \times N_d}$
Output: the trained parameters of ELiCiT (or qELiCiT)
1  $I \leftarrow [N_1] \times \cdots \times [N_d]$
2  $\Theta \leftarrow$ initialized parameters of a ELiCiT (or qELiCiT) model
3  while not converged do
4      Generate a random permutation $\pi$ of $I$
5      for $i \leftarrow 1$ to $|I|$ step $b$ do
6          $e \leftarrow 0$
7          for $j \leftarrow i$ to $\min\{i+b-1, |I|\}$ do
8              $\tilde{x} \leftarrow$ APPROXIMATE$(\pi(j), \Theta)$        ▷ Algorithm 2
9              $e \leftarrow e + (\mathcal{X}(\pi(j)) - \tilde{x})^2$
10         Update $\Theta$ to minimize $e$, i.e., $\Theta \leftarrow \Theta - \alpha \cdot \frac{\partial e}{\partial \Theta}$
11 return $\Theta$
```

Consequently, the deep-learning-based methods are time-consuming for compression, being much slower than decomposition-based methods (refer to Section VI-C). To address these limitations, we design ELiCiT based on the following ideas. **Related to L1,** we model mode indices as latent features (called *feature-based index model*), which enables ELiCiT to be independent of their orders. Moreover, since the features are interpreted as groups of mode indices with arbitrary sizes, our model can generalize the order-based models that divide the indices into equal-sized groups, offering enhanced expressiveness (refer to Section IV-E). **Related to L2,** we quantize the index features through end-to-end clustering with few parameters (called *clustering-based quantization*), which further reduces compression size while avoiding the high costs incurred by LSTM-based reparameterization. Consequently, **related to L3,** we can easily optimize all learnable parameters using gradient descent, which eliminates the need for separate order optimization, leading to faster compression. **Based on these ideas, our goal is to achieve (a) an improved balance between compressed size and approximation error, and (b) reduced compression time.**

B. Feature-based Index Model and Overview

In this subsection, we introduce the feature-based index model of ELiCiT, and based on it, we outline ELiCiT.

Feature-based Index Model: For each of the d modes of the input tensor $\mathcal{X}$, ELiCiT uses latent features to model each index and the corresponding tensor entries. By incorporating learnable latent features, we aim to capture hidden patterns within the data and thus efficiently compress tensors. In ELiCiT, each index i_j of the j-th mode has an r-dimensional continuous *latent feature vector* $\mathbf{F}^{(j)}(i_j) \in [0,1]^r$, and its k -th entry $\mathbf{F}^{(j)}_k(i_j)$ denotes the k-th *latent feature* of index i_j. For brevity, we will refer to *latent feature* as *feature* henceforth in this paper. The features of entries are determined based on the features of indices. Specifically, $\mathcal{X}(i_1, \cdots, i_d)$ has its feature matrix $\mathbf{F}(i_1, \cdots, i_d)$ whose j-th column is $\mathbf{F}^{(j)}(i_j)$, for every $j \in [d]$. The k-th feature vector $\mathbf{F}_k(i_1, \cdots, i_d)$ of $\mathcal{X}(i_1, \cdots, i_d)$ is defined as the k-th row of $\mathbf{F}(i_1, \cdots, i_d)$, which is $(\mathbf{F}^{(1)}_k(i_1), \cdots, \mathbf{F}^{(d)}_k(i_d))$.

Example 1 (Feature Design). *Suppose we model the* $(1,2)$-*th entry of the matrix (2-order tensor) in Figure 1, where* $r = 2$. *In this example, index* 1 *of the first mode has a feature*

$\mathbf{F}^{(1)}(1) = (1,1)$*, and index* 2 *of the second mode has a feature* $\mathbf{F}^{(2)}(2) = (0,1)$*. Thus, the first feature vector* $\mathbf{F}_1(1,2)$ *and the second feature vector* $\mathbf{F}_2(1,2)$ *for the* $(1,2)$*-th entry become* $(1,0)$ *and* $(1,1)$*, respectively.*

This feature-based model is simple but theoretically generalizes the models used in the deep-learning-based methods [17], [18], as elaborated in Section IV-E.

Feature-based Approximation: For each entry, its approximation by ELICIT is obtained from its features. Specifically, for each feature index k, there is a differentiable function $f_k : \mathbb{R}^d \to \mathbb{R}$ that maps the k-th feature vector of each entry to a real value, and ELICIT finally approximates each entry using the outputs of $f_1, \cdots, f_r$ and a function $g : \mathbb{R}^r \to \mathbb{R}$ that reduces the r values to a scalar output. To encourage the features to better model indices, we make these functions exhibit a reasonable characteristic: the more similar the two feature vectors are, the more similar the outputs become. In the extreme, if the feature vectors of the two entries are the same, the outputs corresponding to the two entries should also be the same. Thus, if the outputs for the entries whose index of the j-th mode is i_1 and those for the entries whose index of the j-th mode is i_2 are similar to each other, the features $\mathbf{F}^{(j)}(i_1)$ and $\mathbf{F}^{(j)}(i_2)$ are also likely to be similar to each other.

Learning of Features: In ELICIT, the features are continuous, and all the computation process for approximation is differentiable. Therefore, when fitting the parameters of ELICIT to the input tensor, they can be updated simultaneously through gradient descent-based update iterations, as outlined in Algorithm 1. That is, the training procedure is much simpler, compared to those of the previous deep-learning-based methods (refer to Section IV-A). In our experiments, we used mini-batch training with the Adam optimizer.

Roadmap: In Section IV-C, we provide the details of the approximation process of ELICIT. We theoretically analyze the (1) time complexity, (2) space complexity, and (3) compressed output size of ELICIT in Section IV-D. In Section IV-E, we demonstrate the theoretical expressive power of ELICIT, through comparisons with existing methods.

C. Approximation Process of ELICIT

In ELICIT, there are some *reference* states and the corresponding values for approximating each entry. Specifically, there are 2^d distinct reference states, $\mathbf{s}_1, \cdots, \mathbf{s}_{2^d} \in \{0,1\}^d$, shared across all the features. Each l-th reference state $\mathbf{s}_l$ is represented as the d dimensional binary vector representing $l-1$, which can be obtained by Eq. (1), and there is a value $v_{k,l} \in \mathbb{R}$ corresponding to s_l for each feature index $k \in [r]$.

$$\mathbf{s}_l = \left(s_l^{(1)}, \cdots, s_l^{(d)}\right) = \left(\left\lfloor \frac{l-1}{2^{d-1}} \right\rfloor \bmod 2, \left\lfloor \frac{l-1}{2^{d-2}} \right\rfloor \bmod 2, \cdots, \left\lfloor \frac{l-1}{2^{0}} \right\rfloor \bmod 2\right). \quad (1)$$

1) Handling binary features: For simplicity, we first assume that each mode index has a single binary feature, which is the special case of our feature design. In this case, for every index $i_1, \cdots, i_d$ of the tensor, there exists a unique $l \in [2^d]$ such that the feature vector $\mathbf{F}_1(i_1, \cdots, i_d)$ is identical to the

Algorithm 2: Approximation Process of (q)ELICIT

```
▷ The red and green lines are executed only in qELICIT and
  ELICIT, respectively. The black lines are executed in both
  algorithms.
Input: (a) a position $(i_1, \cdots, i_d) \in [N_1] \times \cdots \times [N_d]$
(b) the feature matrix $\mathbf{F}(i_1, \cdots, i_d)$ of $(i_1, \cdots, i_d)$-th position
(c) the set of candidate values $\{c_{k,l}^{(j)} : (j,k,l) \in [d] \times [r] \times [2^q]\}$
(d) the corresponding values $v_{1,1}, \cdots, v_{1,2^d}, \cdots, v_{k,1}, \cdots, v_{k,2^d}$
    of the reference states $\mathbf{s}_1, \cdots, \mathbf{s}_{2^d}$
(e) the reduce function $g$
Output: the approximated $(i_1, \cdots, i_d)$-th entry $\tilde{\mathcal{X}}(i_1, \cdots, i_d)$
1  for $j \leftarrow 1$ to $d$ do
2      for $k \leftarrow 1$ to $r$ do
3          $l^* \leftarrow \mathrm{argmin}_{l \in [2^q]} \left\{ \left| c_{k,l}^{(j)} - \mathbf{F}_k^{(j)}(i_j) \right| \right\}$      ▷ Section IV-C4
4          $\mathbf{P}_k^{(j)} \leftarrow \left(\mathbf{F}_k^{(j)}(i_j) + c_{k,l^*}^{(j)}\right) - \tilde{\mathbf{F}}_k^{(j)}(i_j)$
5          $\mathbf{P}_k^{(j)} \leftarrow \mathbf{F}_k^{(j)}(i_j)$
6  for $k \leftarrow 1$ to $r$ do
7      $x_k \leftarrow 0$      ▷ Section IV-C2
8      for $l \leftarrow 1$ to $2^d$ do
9          $w \leftarrow 1$
10         for $j \leftarrow 1$ to $d$ do
11             $w \leftarrow w \cdot \left((2\mathbf{P}_k^{(j)} - 1)s_l^{(j)} + (1 - \mathbf{P}_k^{(j)})\right)$
12         $x_k \leftarrow x_k + w \cdot v_{k,l}$
13 return $g(x_1, \cdots, x_r)$      ▷ Section IV-C5
```

l-th reference state $\mathbf{s}_l$. In this case, the approximated value becomes the corresponding value $v_{1,l}$ of $\mathbf{s}_l$ (see Example 2). This process is equivalent to accessing a memory address to retrieve its value in low-level programming.

Example 2 (Approximating Entries with Binary Features)**.** *Suppose we approximate the* $(1,2)$*-th entry of the 2-order tensor in Figure 1. Since* $\mathbf{F}_1(1,2)$ *is identical to* $\mathbf{s}_3 = (1,0)$*,* $\tilde{\mathcal{X}}(1,2)$ *becomes* $v_{1,3} = 3$.

However, the above process for binary features cannot be directly applied to continuous features, which are used by ELICIT, as described below. Specifically, when the continuous feature vector is given, there may be no reference state identical to the feature vector. For example, there is no reference state identical to $\mathbf{F}_1(3,4) = (0.4, 0.7)$ of the 2-order tensor in Figure 1. Thus, the above process needs to be extended to continuous features to be used in ELICIT.

2) Handling continuous features: Below, we describe how to extend the approximation process to continuous features. To address the aforementioned challenge, for the $(i_1, \cdots, i_d)$-th entry, we use the weighted sum of $v_{1,l}$ for every $l \in [2^d]$ instead. The weight $w_{(i_1,\cdots,i_d),l} \in \mathbb{R}$ of each $v_{1,l}$ is determined by the difference between the reference point s_l and the feature vector $\mathbf{F}_1(i_1, \cdots, i_d)$. Specifically, we use the following equations to compute the approximation result $\tilde{\mathcal{X}}(i_1, \cdots, i_d)$:

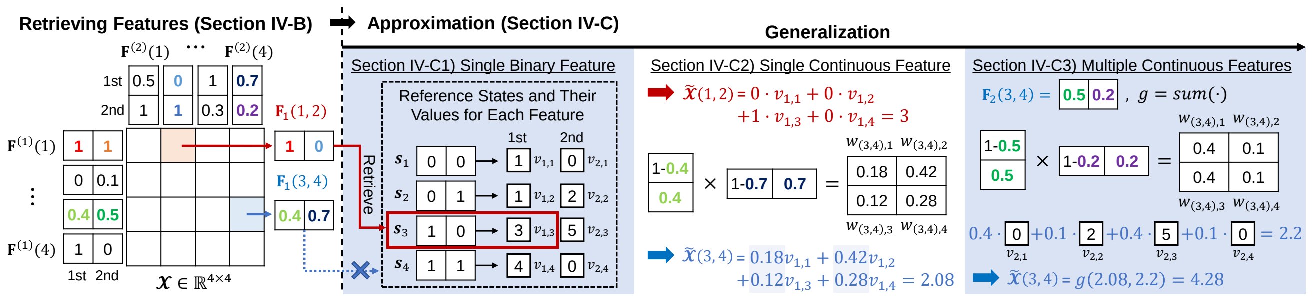
$$y_{(i_1,\cdots,i_d),l,j} := \begin{cases} \mathbf{F}_1^{(j)}(i_j), & \text{if } s_l^{(j)} = 1 \\ 1 - \mathbf{F}_1^{(j)}(i_j), & \text{otherwise} \end{cases}$$

$$w_{(i_1,\cdots,i_d),l} := \prod_{j=1}^{d} y_{(i_1,\cdots,i_d),l,j},$$

$$\tilde{\mathcal{X}}(i_1, \cdots, i_d) := \sum_{l=1}^{2^d} w_{(i_1,\cdots,i_d),l} v_{1,l}.$$

Note that the computed weights have the following properties:

- Since $\mathbf{F}_1^{(j)}(i_j) \in [0,1]$, each weight $w_{(i_1,\cdots,i_d),l}$ is also in



Retrieving Features (Section IV-B)
Approximation (Section IV-C)
Generalization
Section IV-C1) Single Binary Feature
Section IV-C2) Single Continuous Feature
Section IV-C3) Multiple Continuous Features
Reference States and Their Values for Each Feature
Retrieve
1st
2nd
$\mathcal{X} \in \mathbb{R}^{4\times4}$
$\tilde{\mathcal{X}}(1,2) = 0 \cdot v_{1,1} + 0 \cdot v_{1,2} + 1 \cdot v_{1,3} + 0 \cdot v_{1,4} = 3$
$\tilde{\mathcal{X}}(3,4) = 0.18v_{1,1} + 0.42v_{1,2} + 0.12v_{1,3} + 0.28v_{1,4} = 2.08$
$\mathbf{F}_2(3,4) =$ 0.5 0.2, $g = sum(\cdot)$
$0.4 \cdot 0 + 0.1 \cdot 2 + 0.4 \cdot 5 + 0.1 \cdot 0 = 2.2$
$\tilde{\mathcal{X}}(3,4) = g(2.08, 2.2) = 4.28$

Fig. 1. The approximation process of ELICIT. To approximate each entry, it first retrieves the feature vectors of the entries from the features of the indices. Next, it computes the weighted sums of the values assigned to the reference states, based on the features of the entry. The weighted sums are reduced by the reduce function $g(\cdot)$ to compute the approximation result.

$[0,1]$. Moreover, $\sum_{l=1}^{2^d} w_{(i_1,\cdots,i_d),l}$ is always 1, and it can be easily shown by the distributive property.

- The closer the feature vector $\mathbf{F}_1(i_1,\cdots,i_d)$ is to $\mathbf{s}_l$, the larger the corresponding weight $w_{(i_1,\cdots,i_d),l}$ becomes.
- If the features are binary, as in Section IV-C1, $w_{(i_1,\cdots,i_d),l} = 1$ if $\mathbf{F}_1(i_1,\cdots,i_d) = \mathbf{s}_l$ and 0 otherwise.

By the last property, the above process generalizes the process for binary features.

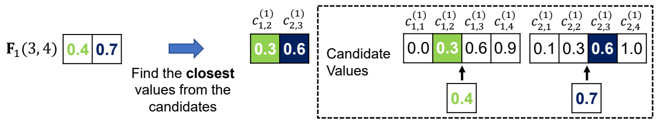
Example 3 (Approximating Entries with Continuous Features)**.** *Suppose we approximate the* $(3,4)$*-th entry of the 2-order tensor in Figure 1. Since* $\mathbf{F}_1(3,4) = (0.4, 0.7)$*, the weights* $w_{(3,4),1}, w_{(3,4),2}, w_{(3,4),3}, w_{(3,4),4}$ *for* $\mathbf{s}_1$, $\mathbf{s}_2$, $\mathbf{s}_3$, $\mathbf{s}_4$ *are* $(1-0.4)\cdot(1-0.7) = 0.18$, $(1-0.4)\cdot 0.7 = 0.42$, $0.4\cdot(1-0.7) = 0.12$*, and* $0.4 \cdot 0.7 = 0.28$*, respectively. Thus,* $\tilde{\mathcal{X}}(3,4)$ *is* $0.18v_{1,1} + 0.42v_{1,2} + 0.12v_{1,3} + 0.28v_{1,4} = 2.08$.

Advantages: This design generalizes the design for binary features, enhancing expressiveness. Its differentiable process allows ELICIT to be easily optimized via gradient descent, addressing Limitation **L3** (see Section IV-A) of prior methods.

3) Handling multiple features (i.e., $r > 1$): To further enhance expressiveness, the final version of ELICIT uses multiple features. For each feature index k, ELICIT computes the output for a target entry using the above process. After collecting r outputs from each feature, a function g reduces these values into a scalar, and any differentiable g (e.g., $\text{sum}(\cdot)$) can be employed for this purpose. We provide specific details about its design in Section IV-C5.

Example 4 (Approximating Entries with Multiple Features)**.** *Suppose we approximate the* $(3,4)$*-th entry of the 2-order tensor in Figure 1, where* $r = 2$ *and* $g = \text{sum}(\cdot)$*. As in Example 3, we can obtain 2.08 from* $\mathbf{F}_1(3,4) = (0.4, 0.7)$*. Similarly, from* $\mathbf{F}_2(3,4) = (0.5, 0.2)$*, we can obtain the weights* $(w_{(3,4),1}, w_{(3,4),2}, w_{(3,4),3}, w_{(3,4),4}) = (0.4, 0.1, 0.4, 0.1)$ *and the output* $0.4v_{2,1} + 0.1v_{2,2} + 0.4v_{2,3} + 0.1v_{2,4} = 2.2$*. Thus,* $\tilde{\mathcal{X}}(3,4)$ *becomes* $g(2.08, 2.2) = 2.08 + 2.2 = 4.28$.

Remaining Challenges: However, the compressed output size of ELICIT increases proportionally to the number of real values in the feature vectors and reference states, which significantly affects the trade-off between the compressed size and the accuracy. Specifically, if we assume the `double` datatype for storing the features and values, ELICIT requires $\sum_{i=1}^{d} N_i \cdot r \cdot 64$ bits for feature vectors for all indices and $2^d \cdot r \cdot 64$ bits for all the corresponding values of the reference states.



$\mathbf{F}_1(3,4)$ 0.4 0.7
Find the closest values from the candidates
$c_{1,2}^{(1)}$ $c_{2,3}^{(1)}$ 0.3 0.6
Candidate Values
$c_{1,1}^{(1)}$ $c_{1,2}^{(1)}$ $c_{1,3}^{(1)}$ $c_{1,4}^{(1)}$ 0.0 0.3 0.6 0.9
$c_{2,1}^{(1)}$ $c_{2,2}^{(1)}$ $c_{2,3}^{(1)}$ $c_{2,4}^{(1)}$ 0.1 0.3 0.6 1.0
0.4
0.7

Fig. 2. The quantization process of qELICIT. For each feature, qELICIT finds the closest candidate value. After quantization, it follows the approximation process of ELICIT.

In most real-world tensors, including the datasets considered in the paper, $\sum_{i=1}^{d} N_i$ is much bigger than 2^d, which indicates that the size of features is the bottleneck of the compressed output size of ELICIT. Furthermore, the permutation-based ordering used in NEUKRON and TENSORCODEC only needs $\sum_{i=1}^{d} N_i \log N_i$ bits, which is smaller than the total feature size of ELICIT even where $r = 1$. Thus, the size of features needs to be reduced for better compression performance.

4) Clustering-based Quantization: To handle the problem posed by our feature-based model, we propose clustering-based quantization on top of ELICIT, which is inspired by a clustering-based approximation method [26]. Specifically, it aims to reduce the size of compressed outputs, while minimizing the performance degradation. **We name this enhanced version qELICIT** to distinguish it from ELICIT without quantization. Note that **quantization replaces LSTM-based reparameterization**, which causes Limitation **L2** (see Section IV-A) of prior methods, with a closely-aligned objective.

Goal: The size of original features for the j-th mode is $N_j \cdot 64$ bits. We aim to further compress the size to be $N_j \cdot (q + o(1))$ bits, where q is small enough to be considered as a constant.

Overview: To represent each feature $\mathbf{F}_k^{(j)}(i_j)$ with $q + o(1)$ bits in expectation, for each $j \in [d]$ and $k \in [r]$, we additionally prepare 2^q learnable candidates $c_{k,1}^{(j)}, \cdots, c_{k,2^q}^{(j)} \in [0,1]$, where $2^q \ll N_j$. After the quantization process, each feature $\mathbf{F}_k^{(j)}(i_j)$ is replaced with $c_{k,l}^{(j)}$ where $l = \text{argmin}_{l \in 2^q}\{|c_{k,l}^{(j)} - \mathbf{F}_k^{(j)}(i,j)|\}$. For each i_j, if we store each integer index of the candidates in $\log_2 N_j$ bits, instead of real-valued quantization vectors, **the size of the compressed features becomes** $2^q \cdot 64 + N_j \cdot \log(2^q) = N_j \cdot (q + o(1))$ **bits** by the assumption (i.e., $2^q \ll N_j$), which satisfies our goal.

Details: During training, since qELICIT replaces the features with the closest candidates and then performs approximation, the original features are not directly used for approximating the entries. Hence, they are unable to be updated with gradient descent iterations. To tackle the issue, qELICIT uses a "straight-through" trick when replacing the values. Specifically, when replacing $\mathbf{F}_k^{(j)}(i_j)$, if the closest candidate is $c_{k,l}^{(j)}$, qELICIT replaces the feature with $\left(c_{k,l}^{(j)} + \mathbf{F}_k^{(j)}(i_j)\right) - \tilde{\mathbf{F}}_k^{(j)}(i_j)$, where $\tilde{\mathbf{F}}_k^{(j)}(i_j)$ is the fixed constant having the same value with $\mathbf{F}_k^{(j)}(i_j)$. Using this trick, qELiCiT can simultaneously update the features and the candidates with a common goal of maximizing the fitting performance of qELiCiT. Note that, instead of our quantization scheme, it is possible to apply an off-the-shelf clustering-based approximation method [26]. However, this replacement introduces an additional hyperparameter[1] and associated tuning cost, without yielding performance gains, as demonstrated in Section VI-D.

Example 5 (Approximation Process of qELiCiT)**.** *Suppose we approximate the* $(3,4)$*-th entry of the 2-order tensor in Figure 2, where* $r = q = 2$ *and* $g = \text{sum}(\cdot)$*. First,* qELiCiT *quantizes the feature matrix* $\mathbf{F}(3,4)$*. To quantize the feature* $\mathbf{F}_1^{(1)}(3)$*, it selects the closest candidate among* $\{c_{1,1}^{(1)}, c_{1,2}^{(1)}, c_{1,3}^{(1)}, c_{1,4}^{(1)}\} = \{0.0, 0.3, 0.6, 0.9\}$*. Thus, the quantized feature becomes* 0.3*. Similarly,* $\mathbf{F}_1^{(2)}(4)$*,* $\mathbf{F}_2^{(1)}(3)$*, and* $\mathbf{F}_2^{(2)}(4)$ *are quantized to* 0.6*,* 0.5*, and* 0.1*, respectively. The rest of the process is the same as that of* ELiCiT.

We found that even if we set q to 4, the accuracy of qELiCiT is not significantly degraded. Thus, we set q to 4 for all experiments, and we further analyze how performance changes with q in Section VI-F.

5) Reduce Function: Below, we present the details of the reduce function g. Our preliminary study reveals that the following reduce function [27], which incorporates non-linearity, yields superior compression performance, compared to simpler ones (e.g., sum$(\cdot)$):

$$g(x_1, x_2, \cdots, x_r) = \left(\sum_{k=1}^{\lfloor r/2 \rfloor} x_k\right) \cdot \tanh\left(\sum_{k=\lfloor r/2 \rfloor + 1}^{r} x_k\right).$$

Since real-world tensors contain values at various scales, in order to adapt to the scale of the input tensors, we additionally introduce two learnable parameters: γ for controlling scale, and β for controlling bias of the output. This leads to the following function used in ELiCiT:

$$g(x_1, x_2, \cdots, x_r) = e^{\gamma} \cdot \left(\sum_{k=1}^{\lfloor r/2 \rfloor} x_k\right) \cdot \tanh\left(\sum_{k=\lfloor r/2 \rfloor + 1}^{r} x_k\right) + \beta. \tag{2}$$

In Section VI-D, we demonstrate the effectiveness of the above design of g, compared to sum$(\cdot)$ and LSTM$(\cdot)$.

D. Theoretical Analysis

We theoretically analyze the output size, approximation and training time, and space for training of our full-fledged method qELiCiT. **All proofs are provided in Appendix C of [20].**

Theorem 1 (Compressed Output Size)**.** *The size of the compressed output of* qELiCiT *is* $r \cdot (q \cdot \sum_{i=1}^{d} N_i + t(d \cdot 2^q + 2^d)) + 2t$ *bits, where* t *is the size of the datatype for a real number.*

[1]If applied to our problem, it treats $\left(c_{k,l}^{(j)} + \mathbf{F}_k^{(j)}(i_j)\right)$ as a constant and $\tilde{\mathbf{F}}_k^{(j)}(i_j)$ as a learnable parameter. As a result, it requires an additional regularization term for cluster centroids to be updated.

Theorem 2 (Approximation Time for Each Entry)**.** *The approximation for each entry takes* $O(rd \cdot (2^q + 2^d))$ *time.*

Theorem 3 (Training Time)**.** *The training time complexity of* qELiCiT *per epoch is* $O(rd \cdot (2^q + 2^d) \cdot \prod_{j=1}^{d} N_j)$*.*

Theorem 4 (Space Usage)**.** *The overall memory requirement of* qELiCiT *during training is* $O(brd \cdot (2^q + 2^d) + \prod_{j=1}^{d} N_j)$*.*

E. Comparison with Existing Methods

Below, we show ELiCiT (partially) generalizes prior tensor-compression methods, demonstrating its expressive power. **For a comparison with ACCAMS, see Appendix D of [20].**

1) T-SVD [21] and CPD [8]: ELiCiT with r features and $g = \text{sum}(\cdot)$ provably generalizes CPD of rank r and T-SVD of rank r (when the input tensor is of order 2, i.e., a matrix).

Theorem 5. ELiCiT *with* r *features and* $g = \text{sum}(\cdot)$ *is a generalization of CPD of rank* r *and T-SVD of rank* r*.*

Proof Sketch. This can be demonstrated by constructing reference states using the rank-1 tensor through a cross-product of length-2 vectors composed of the min-max values from each factor in the CP decomposition output. Features are then set to normalized values after max-min normalization. See Appendix C of [20] for details. □

2) NeuKron *[17] and* TensorCodec *[18]:* Their order-based index models (see Appendix B of [20] for details) are generalized by the feature-based index model used in ELiCiT.

Theorem 6. *The feature-based index model of* ELiCiT *is a generalization of the order-based index models used in* NeuKron *and* TensorCodec*.*

Proof Sketch. We can show this by representing each element in the encoded sequences of their models in binary and concatenating the transformed feature to generate the features of our model. See Appendices B and C of [20] for details. □

V. Extension to Other Real-world Applications

Below, we extend ELiCiT for two real-world applications: (1) matrix completion and (2) neural-network compression.

A. Matrix Completion

Problem Formulation: The goal of the matrix completion problem is to accurately predict the missing entries in the input matrix based on the observed entries.

Problem 2. (Matrix Completion)

- ***Given:*** *a masked matrix* $\bar{\mathbf{A}} = \mathbf{M} \odot \mathbf{A}$*, where* $\mathbf{A} \in \mathbb{R}^{N_1 \times N_2}$ *is the ground-truth matrix,* $\mathbf{M} \in \{0,1\}^{N_1 \times N_2}$ *is a mask indicating the observability of the entries in* $\mathbf{A}$*, and* $\odot$ *is an element-wise multiplication operation,*
- ***Find:*** *the prediction model* Θ *that generates* $\hat{\mathbf{A}}$
- ***to Minimize:*** *prediction error* $\|(\mathbf{1} - \mathbf{M}) \odot (\mathbf{A} - \hat{\mathbf{A}})\|_F^2$ *on the missing entries, where* $\mathbf{1}$ *is the* $N_1 \times N_2$ *matrix of ones.*

Proposed Method: For the matrix completion problem, we develop qELiCiT++, an extended version of qELiCiT that incorporates implicit features motivated from SVD++ [11]. qELiCiT++ uses additional parameters to capture implicit ratings: $\mathbf{F}'^{(j)}_k(i_j)$ for the implicit feature of index i_j in mode j, and the candidate values $c'^{(j)}_{k,1}, \cdots, c'^{(j)}_{k,2^q}$ for the implicit feature, for each $(j,k) \in [2] \times [r]$ and $i_j \in [N_j]$. See Appendix E-A of [20] for the detailed training and prediction processes.

B. Neural-network Compression

Problem Formulation: This problem aims to minimize the number of parameters of a neural-network model while minimizing the performance degradation of the compressed model on the given downstream tasks, formally described as follows:

Problem 3. (Neural-network Compression)

- ***Given:*** *(1) a neural network f_Θ with the trained parameters Θ, (2) a function $p(\cdot)$ that evaluates the performance of neural networks,*
- ***Find:*** *a compressed parameter Φ that parameterizes Θ*
- ***to Minimize:*** *(1) the size of Φ, (2) the amount of performance degradation, i.e., $p(f_\Theta) - p(f_{\Theta(\Phi)})$.*

Proposed Method: We propose TFW-qELɪCɪT, an extended version of qELɪCɪT inspired by TFWSVD [28]. It first compresses each tensor-shaped parameter $\mathcal{X}$ by setting the weighted approximation error as the objective function, which is defined as $\mathcal{L}_{\text{approx}}(\tilde{\mathcal{X}}) = \|\mathcal{W}_{\mathcal{X}} \odot (\mathcal{X} - \tilde{\mathcal{X}}) \odot (\mathcal{X} - \tilde{\mathcal{X}})\|_1$, where $\mathcal{W}_{\mathcal{X}} \in \mathbb{R}^{N_1 \times \cdots \times N_d}$ is the empirical Fisher information. Afterward, it fine-tunes Φ for the function $p(\cdot)$ to optimize the performance of the compressed model, following typical training methods for neural networks. Refer to Appendix E-B of [20] for the details of $\mathcal{W}_{\mathcal{X}}$ and the training process.

VI. Experiments

We performed experiments to answer the following questions:

Q1. **Compression Performance:** How compactly and accurately does qELɪCɪT compress tensors, compared to existing decomposition- and deep-learning-based methods?

Q2. **Speed:** How fast does qELɪCɪT compress tensors, compared to the existing deep-learning-based methods?

Q3. **Ablation Study:** How do the techniques utilized in qELɪCɪT affect its accuracy and compression time?

Q4. **Scalability:** Is compression and approximation by qELɪCɪT scalable w.r.t. to the number of entries?

Q5. **Hyperparameter Sensitivity:** How does the quantization level q affect the compression performance of qELɪCɪT?

Q6. **Application:** How effective are the extensions of qELɪCɪT for matrix completion and neural-network compression?

A. Experimental Settings

Datasets: We used 8 real-world tensors listed in Table II. As shown in [18], the datasets have various properties, such as application domains, smoothness, and density. The semantics of the datasets are provided in Appendix F-A of [20].

Baseline Methods: For deep-learning-based methods, we considered TᴇɴsᴏʀCᴏᴅᴇᴄ [18], NᴇᴜKʀᴏɴ [17], CostCo [24], and M^{2}DMTF [25], as competitors. Note that the latter two were originally designed for completion. For decomposition-based methods, we considered CPD [8], TKD [7], TTD [9], and TRD [10] as competitors. See Section III for their details.

Implementation: We utilized Tensor Toolbox [29] in MATLAB for the implementation of CPD and TKD. For TTD,

TABLE II
Real-world datasets used in the paper. The detailed semantics of the datasets are provided in Appendix F-A of [20].

Order	Name	Shape	#Entries	Description
4	`Absorb`	$192 \times 288 \times 30 \times 120$	199.1M	Climate
	`NYC`	$265 \times 265 \times 28 \times 35$	68.8M	Traffic volume
3	`Action`	$100 \times 570 \times 567$	32.3M	Video features
	`Activity`	$337 \times 570 \times 320$	61.5M	Video features
	`Airquality`	$5,600 \times 362 \times 6$	12.2M	Climate
	`PEMS`	$963 \times 144 \times 440$	61.0M	Traffic volume
	`Stock`	$1,317 \times 88 \times 916$	106.2M	Stock
	`Uber`	$183 \times 24 \times 1,140$	5.0M	Traffic volume

TRD, and M^{2}DMTF, we employed the original implementation in MATLAB provided by the authors. For TᴇɴsᴏʀCᴏᴅᴇᴄ, NᴇᴜKʀᴏɴ, and CostCo, we used the implementation in PyTorch or TensorFlow provided by the authors. We implemented our methods including qELɪCɪT and its variants using PyTorch. As demonstrated in Section VI-D, limiting the feature values to the range $[0,1]$ slightly slows down compression speed with marginal differences in compression performance. Therefore, in our implementation, we allowed the feature values to take any real value.

Evaluation Protocol: We used fitness, aforementioned in Section II, to measure the accuracy of a compressed result. We determined the compression output size for each method using the double-precision (64-bit) floating format. We performed each experiment 5 times, and reported the average and standard deviation. Details regarding the hyperparameters and training protocols are provided in Appendix F-A of [20].

Machines: For deep-learning-based methods except for M^{2}DMTF, we conducted all experiments on a server with 4 RTX 2080Ti GPUs. For the baselines that do not use GPUs in their implementation, we ran them on a desktop with an i5-9600K CPU and 64GB RAM.

B. Q1. Compression Performance

We evaluated the compression performance of qELɪCɪT compared to its competitors in terms of compression size and fitness. For each method, we conducted experiments with at least 4 settings yielding different compressed output sizes (except for TᴇɴsᴏʀCᴏᴅᴇᴄ on the `Uber` dataset). To directly compare qELɪCɪT with the deep-learning-based methods, we set the compressed output size to be as slightly smaller as possible. Since the implementation of M^{2}DMTF only supports 3-order tensors, we could not test it on the Absorb and NYC datasets, which are 4-order tensors.

For all datasets, qELɪCɪT demonstrated the best trade-offs between compressed size and fitness, as shown in Figure 3. Specifically, the compressed output size of qELɪCɪT was up to $5.05\times$ smaller than that of the most compact competitor with a comparable fitness. Moreover, its fitness was up to 48% higher than that of the most accurate competitor with a similarly-sized output. CostCo and M^{2}DMTF, which are originally designed for completion but adapted for compression, performed comparably to decomposition-based compression methods overall and worse in a few cases.

C. Q2. Speed

We measured the compression speed for all tested methods. For a fair comparison, we set the compressed output sizes to

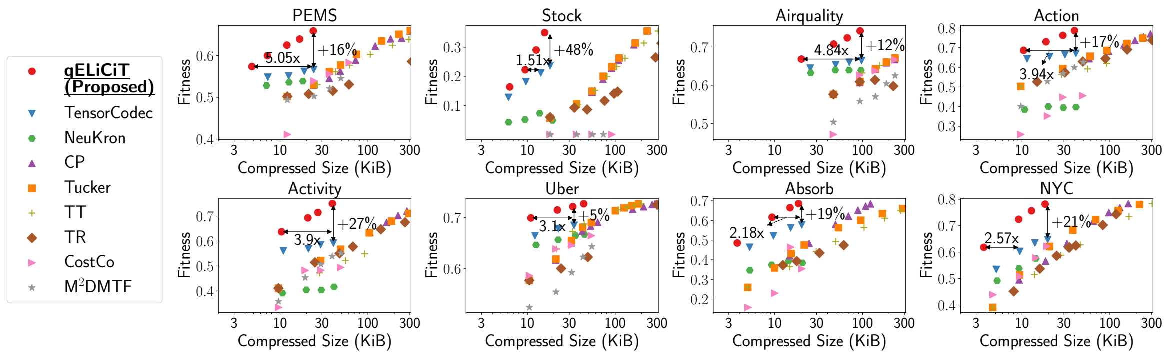


Fig. 3. Our qELiCiT provides compact and accurate compression of tensors. The compressed size of qELiCiT is up to $5.05\times$ smaller than that of the most compact competitor with similar fitness. Its fitness is up to 48% higher than that of the best fitness baseline with a similar compressed size.

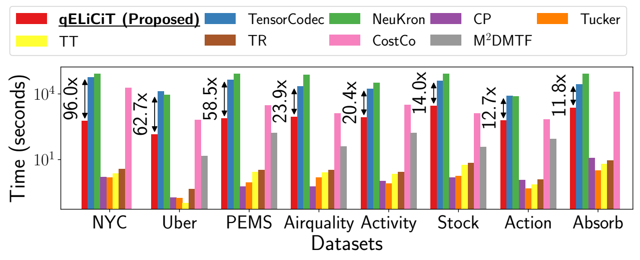


Fig. 4. Our qELiCiT is significantly faster than the existing deep-learning-based compression methods (i.e., TensorCodec and NeuKron). For a similar compression size, compression by qELiCiT is up to $96\times$ faster than compression by TensorCodec.

be similar (spec. similar to the smallest output sizes of TensorCodec in Figure 3). **Our quantized method qELiCiT was significantly faster than the deep-learning-based compression methods.** As seen in Figure 4, compression by qELiCiT was up to $96\times$ faster than that by TensorCodec, the fastest deep-learning-based compression method, despite the superior compression performance of qELiCiT in Section VI-B.

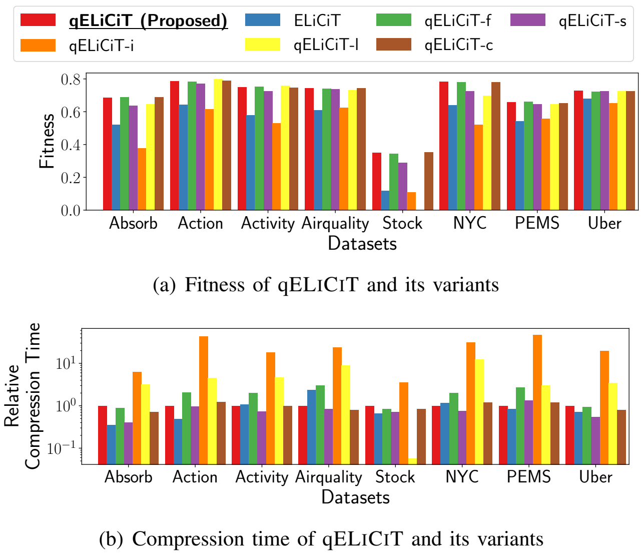
D. Q3. Ablation Study

We performed an ablation study to demonstrate the importance of the techniques applied to qELiCiT. Specifically, we considered the following variants of qELiCiT:

- **ELiCiT:** a variant of qELiCiT without quantization, presented in Section IV-C.
- **qELiCiT-f:** a variant of qELiCiT where feature values are restricted to the range of $[0, 1]$, as discussed in Section VI-A.
- **qELiCiT-s:** a variant of qELiCiT, where $g = \text{sum}(\cdot)$.
- **qELiCiT-i:** a variant of qELiCiT with the order-based index model used in NeuKron and TensorCodec.
- **qELiCiT-l:** a variant of qELiCiT, where g is parameterized by LSTM, as used in NeuKron and TensorCodec. We set the hidden dimension of LSTM to 16.
- **qELiCiT-c:** a variant of qELiCiT, where we use the clustering-based quantization approach used in [26], as discussed in Section IV-C4.

To ensure a fair comparison, we set the compressed output sizes of qELiCiT and its variants to be similar to the largest compressed size of qELiCiT in Section VI-B for each dataset.

Every technique applied to qELiCiT enhances compression performance in terms of accuracy and compression time, as seen in Figure 5. ELiCiT, qELiCiT-s, and qELiCiT-i showed relatively lower fitness in most of the datasets. While qELiCiT-f and qELiCiT-l achieved similar fitness to qELiCiT in most cases, their compression speed is significantly slower, except for qELiCiT-f on `Absorb`, `Stock`, and `Uber` datasets. Lastly, qELiCiT-c achieved similar fitness and compression speed to qELiCiT. Note that qELiCiT-c incorporates an additional regularization term, as we discussed in Section 4.4. Thus, tuning the regularization coefficient is necessary for qELiCiT-c, while the original qELiCiT does not require such a process.


(a) Fitness of qELiCiT and its variants

(b) Compression time of qELiCiT and its variants

Fig. 5. Every technique used in qELiCiT is effective. The full-fledged version, qELiCiT, achieves the best fitness, and it demonstrates the fastest compression speed among those with high fitness (i.e., qELiCiT-{f,c,l}).

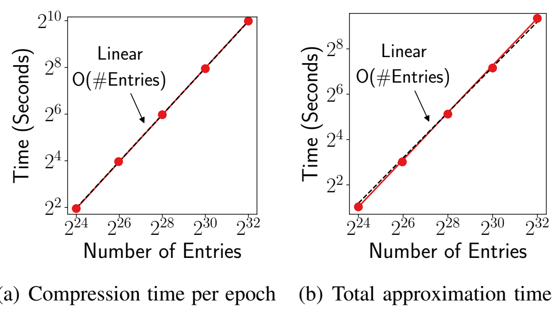

(a) Compression time per epoch (b) Total approximation time

Fig. 6. qELiCiT is near-linearly scalable with the number of entries. Both per-epoch compression time and approximation time increase near-linearly with the number of entries.

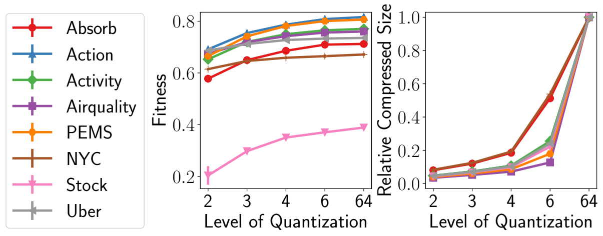


Fig. 7. Hyperparameter sensitivity. As the quantization level q increases, the fitness and the size of the compressed output of qELiCiT also increase, and in most datasets, $q = 4$ offers a good balance between fitness and compressed size. In the figure, we considered ELiCiT without quantization when $q = 64$.

E. Q4. Scalability

We measured the compression time per epoch and the approximation time given tensors with different sizes. We generated 5 synthetic 4-order tensors, varying the length of each mode in the range of $\{64, 90, 128, 181, 256\}$. The largest generated tensor contains about 4.3B entries and occupies 32 GiB when stored on disk. As seen in Figure 6, both the compression time per epoch and the approximation time for the entire tensor were near-linear in the number of entries.

F. Q5. Hyperparameter Sensitivity

To analyze how the hyperparameter q (i.e., quantization level, or the number of candidates) affects the compression performance, we measured fitness depending on its value. For each dataset, we fixed the size r of latent features to the setting that yielded the largest compressed output size in Section VI-B, and then we measured the performance of qELiCiT by varying q. As shown in Figure 7, as the level q of quantization increases, both the fitness and the size of the compressed output of ELiCiT increase. In addition, we observed that these changes were rapid around $q = 4$, where it provides competitive accuracy with sufficiently compact compression.

G. Q6. Applications

We show the effectiveness of our qELiCiT extensions, i.e., qELiCiT++ and TFW-qELiCiT, to their applications described in Section V. For all methods using GPUs, we used a server with an RTX 3090 GPU. For the other methods that do not use GPUs, we used the desktop aforementioned in Section VI-A. Since all the competitors used the single-precision floating point format, unlike the evaluation setup for tensor compression, we used the same format for our methods. Detailed settings for these experiments are provided in Appendices F-B and F-C of [20].

1) Matrix Completion: We compared qELiCiT++ with SVD [12], SVD++ [11], SparseFC [30], GLocal-K [31], and IGMC [32]. For all methods excluding the auto-encoder-based methods, we ran experiments with 4 different model sizes as budgets: $\{16N, 32N, 64N, 128N\}$ in bytes where $N = N_1 + N_2$, and N_1 and N_2 are the numbers of rows and columns of a matrix to be completed, respectively. For GLocal-K and IGMC, we used budgets in the range of $\{64N, 128N\}$ bytes due to their higher model size requirements compared to the other methods. For all experiments in this subsection, we used RMSE as the standard evaluation metric for this task. We split every dataset into train/valid/test sets and performed Bayesian optimization to tune hyperparameters on the valid set with 200 trials. The split ratio is 7:1:2 for `ML-100K`, `ML-1M`, and `ML-10M`, and 8:1:1 for `douban` and `flixster`.

TABLE III
OUR TFW-ELiCiT PROVIDES THE BEST COMPRESSION FOR THE NEURAL NETWORK BERT$_{\text{BASE}}$ ACROSS MOST SUBTASKS, DELIVERING THE HIGHEST ACCURACY FOR NEARLY THE SAME COMPRESSION SIZE. THE BEST SCORE IS IN BOLD, AND THE SECOND-BEST SCORE IS UNDERLINED FOR EACH TASK.

Model (Size)	`CoLA`	`MNLI`	`MRPC`	`QNLI`	`QQP`	`SST-2`	`STSB`	Avg.
BERT$_{\text{base}}$ (418MiB)	59.1 ±1.9	84.7 ±0.2	90.5 ±0.7	91.7 ±0.1	88.1 ±0.3	92.8 ±0.4	89.4 ±0.3	85.2
SVD (256MiB)	44.4 ±1.7	82.9 ±0.3	86.8 ±0.6	89.7 ±0.2	87.5 ±0.3	<u>91.3</u> ±0.6	86.9 ±0.5	81.3
FWSVD (256MiB)	50.2 ±1.1	83.3 ±0.4	88.1 ±0.9	90.3 ±0.2	**87.6** ±0.3	91.0 ±0.5	88.1 ±0.3	82.7
TFWSVD (134MiB)	4.7 ±7.4	79.0 ±0.4	83.7 ±0.6	86.1 ±0.5	85.7 ±0.3	87.4 ±0.9	84.7 ±0.5	73.0
TFWSVD (175MiB)	42.2 ±2.4	81.5 ±0.2	86.9 ±0.9	88.8 ±0.2	86.9 ±0.2	89.4 ±0.5	87.0 ±0.4	80.4
TFWSVD (256MiB)	53.8 ±1.6	**83.5** ±0.2	<u>89.9</u> ±0.9	<u>90.4</u> ±0.2	87.4 ±0.3	90.7 ±0.4	<u>88.6</u> ±0.5	83.5
TFW-qELiCiT (**116MiB**)	<u>55.3</u> ±1.6	83.3 ±0.3	89.8 ±0.5	<u>90.4</u> ±0.2	87.4 ±0.3	91.1 ±0.6	<u>88.6</u> ±0.4	<u>83.7</u>
TFW-ELiCiT (256MiB)	**57.4** ±0.9	**83.5** ±0.5	**90.0** ±0.6	**90.6** ±0.3	<u>87.5</u> ±0.3	**91.4** ±0.9	**88.7** ±0.4	**84.1**

As shown in Figure 8, under all size budgets, except for the largest budget on the `ML-10M` dataset, qELiCiT++ outperformed its competitors. In addition, qELiCiT++ even achieved lower RMSE than the auto-encoder-based competitors that used larger size budgets. Refer to Appendix G-A of [20] for the results with additional budget ranges $\{256N, 512N\}$ for GLocal-K and SparseFC.

2) Neural-network Compression: For this task, we compared our TFW-qELiCiT with FWSVD [14] and TFWSVD [28]. We further compared it with TFW-ELiCiT, a variant of TFW-qELiCiT without quantization, to check the effectiveness of the quantization. As in [14], [28], we evaluated the methods on 7 subtasks in the GLUE benchmark [33].

For training the models, we followed the overall procedure of [28]. First, we fine-tuned the parameters of BERT$_{\text{base}}$ [34] to each subtask. From the fine-tuned parameters, we performed the process described in Algorithm 4 of [20], i.e., after compressing the trained parameters using a compression algorithm, we fine-tuned the compressed parameters to optimize the performance of the compressed model. For each subtask, we used its standard evaluation metric: Matthews correlation for `CoLA`, Pearson correlation for `STSB`, F1 for `MRPC` and `QQP`, and accuracy for `MNLI`, `QNLI`, and `SST-2`.

Table III shows the sizes and average performances along with standard deviations of the neural-network compression methods for each subtask. TFW-ELiCiT outperformed its competitors with nearly identical compressed sizes in all subtasks except `QQP`. Additionally, its quantized version TFW-qELiCiT achieved a compression size of 116MiB, which is 54.7% smaller than 256MiB of TFWSVD, while showing competitive accuracy. Note that the models compressed by TFWSVD with sizes smaller than 200MiB performed worse than those compressed by SVD and FWSVD.

Extra Experiments: Extra results on approximation and compression speed, as well as performance comparisons excluding the effects of quantization, are in Appendix G of [20].

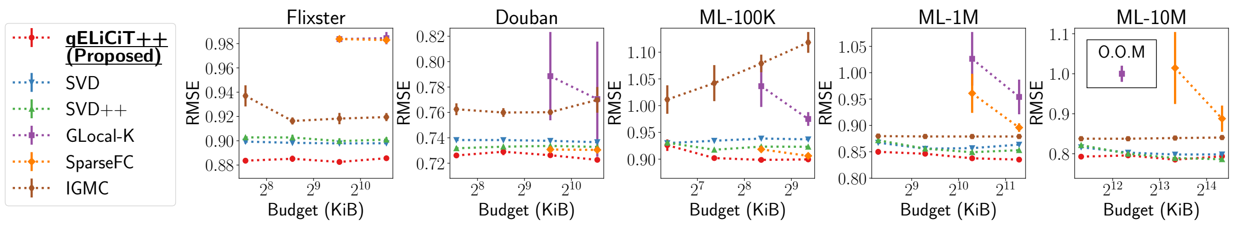


Fig. 8. Our qELiCiT++ provides the best trade-off between parameter size budget and accuracy in matrix completion. Given the same size budget, it outperforms all other competitors in terms of RMSE across all settings, except for the largest budget on `ML-10M`.

VII. Conclusion

In this work, we propose ELiCiT, an effective and lightweight lossy tensor compression method. ELiCiT achieves both speed and performance with lightweight yet theoretically expressive model design and end-to-end clustering-based quantization. Using eight real-world tensors and two real-world applications of tensor compression, we demonstrate the following strengths of ELiCiT (spec., qELiCiT):

- **Compact and Accurate (Figure 3):** While achieving comparable fitness, qELiCiT compresses tensors 1.51-5.05× smaller than the most compact competitor.
- **Fast (Figure 4):** qELiCiT is 11.8-96.0× faster with 5-48% better fitness than the latest deep-learning-based competitors with similar-size outputs.
- **Applicable (Figure 8 and Table III):** The variants of qELiCiT for matrix completion and neural-network compression outperform state-of-the-art competitors for these applications, in terms of the trade-offs between the model size and application performance.

For **reproducibility**, the code and datasets used in the paper are available at https://github.com/jihoonko/icdm24-elicit.

Acknowledgements: This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00438638, EntireDB2AI: Foundations and Software for Comprehensive Deep Representation Learning and Prediction on Entire Relational Databases, 90%) (No. RS-2019-II190075, Artificial Intelligence Graduate School Program (KAIST), 10%).

References

[1] T. Mitchell, W. Cohen, E. Hruschka, P. Talukdar, B. Yang, J. Betteridge, A. Carlson, B. Dalvi, M. Gardner *et al.*, "Never-ending learning," *Communications of the ACM*, vol. 61, no. 5, pp. 103–115, 2018.
[2] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.
[3] OpenAI, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.
[4] V. Bhaskaran and K. Konstantinides, *Image and video compression standards: algorithms and architectures*. Springer Science & Business Media, 1997.
[5] S. Ma, X. Zhang, C. Jia, Z. Zhao, S. Wang, and S. Wang, "Image and video compression with neural networks: A review," *IEEE TCSVT*, vol. 30, no. 6, pp. 1683–1698, 2019.
[6] G. W. Stewart, "On the early history of the singular value decomposition," *SIAM review*, vol. 35, no. 4, pp. 551–566, 1993.
[7] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.
[8] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.
[9] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.
[10] Q. Zhao, M. Sugiyama, L. Yuan, and A. Cichocki, "Learning efficient tensor representations with ring-structured networks," in *ICASSP*, 2019.
[11] Y. Koren, "Factorization meets the neighborhood: a multifaceted collaborative filtering model," in *KDD*, 2008.
[12] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.
[13] A. Beutel, A. Ahmed, and A. J. Smola, "Accams: Additive co-clustering to approximate matrices succinctly," in *WWW*, 2015.
[14] Y.-C. Hsu, T. Hua, S. Chang, Q. Lou, Y. Shen, and H. Jin, "Language model compression with weighted low-rank factorization," in *ICLR*, 2021.
[15] U. Thakker, I. Fedorov, C. Zhou, D. Gope, M. Mattina, G. Dasika, and J. Beu, "Compressing rnns to kilobyte budget for iot devices using kronecker products," *ACM JETC*, vol. 17, no. 4, pp. 1–18, 2021.
[16] C. Yin, D. Zheng, I. Nisa, C. Faloutsos, G. Karypis, and R. Vuduc, "Nimble gnn embedding with tensor-train decomposition," in *KDD*, 2022.
[17] T. Kwon, J. Ko, J. Jung, and K. Shin, "Neukron: Constant-size lossy compression of sparse reorderable matrices and tensors," in *WWW*, 2023.
[18] ——, "Tensorcodec: Compact lossy compression of tensors without strong data assumptions," in *ICDM*, 2023.
[19] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: an approach to modeling networks." *Journal of Machine Learning Research*, vol. 11, no. 2, 2010.
[20] "Supplementary materials," 2024. [Online]. Available: https://github.com/jihoonko/icdm24-elicit
[21] C. Eckart and G. Young, "The approximation of one matrix by another of lower rank," *Psychometrika*, vol. 1, no. 3, pp. 211–218, 1936.
[22] J. Håstad, "Tensor rank is np-complete," in *Automata, Languages and Programming: 16th International Colloquium Stresa, Italy, July 11–15, 1989 Proceedings 16*. Springer, 1989, pp. 451–460.
[23] V. De Silva and L.-H. Lim, "Tensor rank and the ill-posedness of the best low-rank approximation problem," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 3, pp. 1084–1127, 2008.
[24] H. Liu, Y. Li, M. Tsang, and Y. Liu, "Costco: A neural tensor completion model for sparse tensors," in *KDD*, 2019.
[25] J. Fan, "Multi-mode deep matrix and tensor factorization," in *ICLR*, 2021.
[26] T. Chen, L. Li, and Y. Sun, "Differentiable product quantization for end-to-end embedding compression," in *ICML*, 2020.
[27] H.-W. Nam, Y.-B. Moon, and T.-H. Oh, "Fedpara: Low-rank hadamard product for communication-efficient federated learning," in *ICLR*, 2022.
[28] T. Hua, Y.-C. Hsu, F. Wang, Q. Lou, Y. Shen, and H. Jin, "Numerical optimizations for weighted low-rank estimation on language models," in *EMNLP*, 2022.
[29] "Tensor toolbox," 2024. [Online]. Available: https://www.tensortoolbox.org/
[30] L. Muller, J. Martel, and G. Indiveri, "Kernelized synaptic weight matrices," in *ICML*, 2018.
[31] S. C. Han, T. Lim, S. Long, B. Burgstaller, and J. Poon, "Glocal-k: Global and local kernels for recommender systems," in *CIKM*, 2021.
[32] M. Zhang and Y. Chen, "Inductive matrix completion based on graph neural networks," in *ICLR*, 2019.
[33] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," in *ICLR*, 2018.
[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.