APPENDIX A
EXTRA RELATED WORK (RELATED TO SECTION III)

In this section, we review methods rooted in tensor compression, focusing on their applications in matrix completion and neural network compression.

Matrix Completion: Lossy compression methods for matrices have been widely employed for matrix compression, whose objective is to impute missing entries of a partially observed matrix, such as a user-item rating matrix in recommender systems. SVD [12] extends T-SVD with bias terms and regularization terms. SVD++ [11] is an extension of SVD that incorporates implicit features into the item factors to better capture implicit feedback.

Recent methods have introduced deep learning techniques for this task, categorized into (1) GNN-based completion and (2) auto-encoder-based completion. GNN-based methods express an input matrix as a graph based on values or additional features given for each index, and then predict the missing entries using graph neural networks (GNNs). For example, GC-MC [37] formulates the problem as a link classification problem in bipartite graphs, and uses graph auto-encoder (GAE) as the backbone model to predict the missing values by predicting labels of the corresponding links between rows and columns. IGMC [34] extracts an enclosing subgraph for each entry and formulates the problem as graph classification. For auto-encoder-based completion methods, each row or column is fed into an auto-encoder whose training objective is to minimize the difference between the inputs and the outputs. AutoRec [38] is the pioneering auto-encoder-based method, where the rows of the target matrices are fed into the auto-encoder, which subsequently produces outputs that include the missing entries in the input rows. SparseFC [32] re-parametrizes the weight matrices of the auto-encoder architecture through kernel functions to improve the prediction performance. GLocal-K [33] further improves the performance by introducing a global kernel applied to the input matrix with convolution operations.

Neural-network Compression: The tensor compression task is directly applied to neural-network compression that aims to reduce the number of parameters of a neural network while striving to minimize any performance degradation. FWSVD [14] and TFWSVD [30] have been proposed to efficiently compress language models by replacing their matrix-form parameters with low-rank approximations, i.e., the inner products of two small matrices approximating the original parameters. Thakker et al. [15] utilized Kronecker products of two small matrices to reduce the number of parameters of Recurrent Neural Networks. NimbleGNN [16] adopts TTD to compactly compress hidden embeddings of Graph Neural Networks.

APPENDIX B
DETAILS OF ORDER-BASED INDEX MODEL IN NEUKRON AND TENSORCODEC (RELATED TO SECTION IV)

When approximating an entry, NEUKRON and TENSORCODEC first encode the position of the entry as a sequence, and then feed the sequence into their model to compute the approximation result. Specifically, the encoding process of TensorCodec, so-called TT-tensor format [19], includes the following procedure:

- Each index has an encoded sequence of length d', which can also be viewed as d'-dimensional feature, where $d' \in \mathbb{N}$ is a hyperparameter. For each $(j,k) \in [d] \times [d']$, there is a hyperparameter $n_{j,k} \in \mathbb{N}$, and $\prod_{k=1}^{d'} n_{j,k} = N_j$ holds for every $j \in [d]$.
- For each $j \in [d]$, there is a learnable permutation $\pi_j : [N_j] \to [N_j]$ and a fixed one-to-one function $h_j : [N_j] \to [n_{j,1}] \times \cdots \times [n_{j,d'}]$ defined as $h_j\left(\sum_{k=1}^{d'}\left(m_k \cdot \prod_{l=k+1}^{d'} n_l\right)+1\right) = (m_1, \cdots, m'_d)$. With these functions, each index $(i_1, \cdots, i_d) \in [N_1] \times \cdots \times [N_d]$ is mapped to a d'-dimensional feature. Specifically, the feature of the $(i_1, \cdots, i_d)$-th entry becomes $(((h_1 \circ \pi_1)(i_1)_1, \cdots, (h_d \circ \pi_d)(i_d)_1), \cdots, ((h_1 \circ \pi_1)(i_1)_{d'}, \cdots, (h_d \circ \pi_d)(i_d)_{d'}))$.

Then, it uses the output of h to compute the indices in the folded tensor of which the order is d'. NeuKron uses the special case of TT-tensor format which $n_{j,k}$ is 1 or 2 for all pairs of (j,k). The encoding process is too restrictive and this leads to the limited expressiveness. Since h_j's and pi_j's are one-to-one functions, all indices in the same mode should have different features from one another. Furthermore, the indices are always split into the groups of fixed sizes, as we described in L1 of Section IV-A.

APPENDIX C
PROOFS (RELATED TO SECTIONS IV-D, IV-E)

A. Proof of Theorem 1

Proof. To reconstruct the original tensor from the compressed output, qELiCiT requires (1) the quantized features of the indices, (2) the candidate values for quantization, (3) the corresponding values of the reference states, (4) the scale parameter γ, and (5) the bias parameter β. Note that we do not need to store the reference states since they are deterministic. As described in Section IV-C4, for each mode j and feature k, we need $N_j \cdot q + 2^q \cdot t$ bits for storing the quantized features and the candidate values. For the corresponding values of the reference states, we need to store $2^d \cdot r$ real numbers. Thus, the total compressed output size is $(t \cdot r \cdot 2^d) + \sum_{j=1}^{d}\sum_{k=1}^{r}(N_j \cdot q + t \cdot 2^q) = r \cdot (q \cdot \sum_{i=1}^{d} N_i + t(d \cdot 2^q + 2^d)) + 2t$ bits. $\square$

B. Proof of Theorem 2

Proof. In Algorithm 2, computing $\mathbf{P}_{j,k}$ for each $j \in [d], k \in [r]$ requires $O(2^q)$ time, since we need to find l^* among $[2^q]$. The time complexity of computing the inputs of g, i.e., $x_1, \cdots, x_k$, is $O(rd \cdot 2^l)$. Therefore, the approximation time for each entry is $O(rd \cdot 2^q + rd \cdot 2^d) = O(rd \cdot (2^q + 2^d))$. $\square$

C. Proof of Theorem 3

Proof. By Theorem 2, the approximation time for each entry is $O(rd \cdot 2^q + rd \cdot 2^d) = O(rd \cdot (2^q + 2^d))$. And generate the permutation π requires $O(\prod_{j=1}^{d} N_j)$ time. Since qELiCiT utilizes gradient descent-based update, and does not require an additional process for updating the parameters, e.g., order optimization in NeuKron and TensorCodec, the overall training time complexity per epoch is $O(rd \cdot (2^q + 2^d) \cdot \prod_{j=1}^{d} N_j)$. □

D. Proof of Theorem 4

Proof. In Algorithm 1, the random permutation π and the input tensor $\mathcal{X}$ requires $O(\prod_{j=1}^{d} N_j)$ memory. For approximating each entry, qELiCiT requires $O(rd \cdot 2^q)$ to compute $\mathbf{P}_{j,k}$'s, and $O(rd \cdot 2^l)$ to compute x_k's. Lastly, the reduced function g consumes $O(r)$ memory, if we set g to the function in Eq. 2. Since qELiCiT approximates b entries in parallel, the overall memory requirements of qELiCiT during training is $O(brd \cdot (2^q + 2^d) + \prod_{j=1}^{d} N_j)$. □

E. Proof of Theorem 5

Proof. Suppose $\mathcal{X}$ be the summation of r rank-1 tensors, i.e., $\mathcal{X} = \sum_{k=1}^{r} \mathbf{x}_k^{(1)} \circ \cdots \circ \mathbf{x}_k^{(d)}$, where $\circ$ denotes an outer product and $\mathbf{x}_k^{(j)} \in \mathbb{R}^{N_j}$ for each $(k,j) \in [r] \times [d]$. Let $m_k^{(j)}$ and $M_k^{(j)}$ be the minimum and maximum value of $\mathbf{x}_k^{(j)}$, respectively, and $\tilde{\mathbf{x}}_k^{(j)} = (\mathbf{x}_k^{(j)} - m_k^{(j)} \mathbf{1}_{N_j})/(M_k^{(j)} - m_k^{(j)}) \in [0,1]^{N_j}$ be the normalized vector of $\mathbf{x}_k^{(j)}$ by $m_k^{(j)}$ and $M_k^{(j)}$, where $\mathbf{1}_n \in \mathbb{R}^n$ is the n-dimensional vector whose values are all one. If we set $\mathbf{F}_k^{(j)}(i_j)$ to the i_j-th value $\tilde{\mathbf{x}}_k^{(j)}(i_j)$ of $\tilde{\mathbf{x}}_k^{(j)}$ and $v_{k,l}$ to $\prod_{j=1}^{d} (m_k^{(j)} + s_l^{(j)} \cdot (M_k^{(j)} - m_k^{(j)}))$,

$$\begin{aligned}
\tilde{\mathcal{X}}(i_1, \cdots, i_d) &= g\left(\sum_{l=1}^{2^d} w_{(i_1,\cdots,i_d),1,l} v_{1,l}, \cdots, \sum_{l=1}^{2^d} w_{(i_1,\cdots,i_d),r,l} v_{r,l}\right) \\
&= \sum_{k=1}^{r} \sum_{l=1}^{2^d} \prod_{j=1}^{d} y_{(i_1,\cdots,i_d),k,l} (m_k^{(j)} + s_l^{(j)} \cdot (M_k^{(j)} - m_k^{(j)})) \\
&= \sum_{k=1}^{r} \prod_{j=1}^{d} \left((1 - \mathbf{F}_k^{(j)}(i_j)) m_k^{(j)} + \mathbf{F}_k^{(j)}(i_j) M_k^{(j)}\right) \\
&= \sum_{k=1}^{r} \prod_{j=1}^{d} m_k^{(j)} + \tilde{\mathbf{x}}_k^{(j)}(i_j)(M_k^{(j)} - m_k^{(j)}) \\
&= \sum_{k=1}^{r} \prod_{j=1}^{d} x_k^{(j)}(i_j) = \mathcal{X}(i_1, \cdots, i_j).
\end{aligned}$$

Therefore, ELiCiT with r features generalizes CPD of rank r and T-SVD of rank r, since T-SVD can be expressed with CPD. □

F. Proof of Theorem 6

Proof. For each $k \in [d']$, let L_k be $\lceil \log_2(\max n_{j,k}) \rceil$. For the k-th feature of the design provided in Section IV-E, ELiCiT can fully represent the feature using from the $(\sum_{i=1}^{k-1} L_i) + 1$-th feature to the $(\sum_{i=1}^{k-1} L_i) + L_k$-th feature if we restrict the features to be binaries. Therefore, ELiCiT with $r = \sum_{i=1}^{d'} L_i$ features generalizes the designs used in NeuKron and TensorCodec. □

Appendix D
Comparison with ACCAMS (Related to Section IV-E)

Similar to ELiCiT, ACCAMS [13] used a feature-based design for approximating matrices. For each iteration, it identifies co-clusters for a given input matrix $\mathbf{A}$, by obtaining sets C_r and C_c of the clusters of rows and columns in $\mathbf{A}$, respectively, using k-means clustering. For each (i,j)-th entry, it finds the cluster r_i containing the index i from C_r and the cluster c_j containing the index j from C_c, and then assigns the average of the values of the entries located in the co-clusters. It repeats the procedure and approximates the value using the summation of the assigned values during iterations. Compared to ACCAMS, ELiCiT has the following advantages:

- While ACCAMS can only handle matrices, ELiCiT can handle any higher-order tensors.
- ACCAMS performs additive co-clustering, in other words, it determines the k-th feature after the previous $k-1$ features are all determined. On the other hand, ELiCiT computes the features simultaneously, thereby allowing to use parallel computing.
- Even though the design can be used for tensors, there is a strong limitation on the number of clusters for efficiency, since it significantly affects the number of the parameters. For example, if we set the number of clusters to 16 for each index of the 4-order tensor, the number of parameters is at least $16^4 \cdot r$, where r is the number of features, and it may significantly increase even by the small increase in r. However, this issue does not apply to qELiCiT since the number of parameters for the corresponding values of the reference point does not depend on q.

In addition, ELiCiT with r features theoretically generalizes ACCAMS with r features and 2×2 co-clusters for each feature. Refer to Theorem 7 for the details.

Theorem 7. ELiCiT *generalizes ACCAMS with $t = r$ and $k_m = k_n = 2$, where t is the number of iterations, and k_m and k_n are the maximum numbers of clusters of rows and columns of matrices, respectively.*

Proof. For each iteration $k \in [t]$, there are two disjoint clusters $C_{r,1}^{(k)}, C_{r,2}^{(k)} \subseteq [N_1]$ of rows, and two disjoint clusters $C_{c,1}^{(k)}, C_{c,2}^{(k)} \subseteq [N_2]$ of columns. Let $x_{i,j}^{(k)} \in \mathbb{R}$ be the value assigned to the co-cluster $(C_{r,i}^{(k)}, C_{c,j}^{(k)})$, for each $i,j \in [2]$. Then, the ELiCiT model with $r = t$ generalizes ACCAMS with $t = r$ and $k_m = k_n = 2$, if we set $\mathbf{F}_k^{(1)}(i_1)$ to $\mathbb{1}\{i_1 \in C_{r,2}^{(k)}\}$, $\mathbf{F}_k^{(2)}(i_2)$ to $\mathbb{1}\{i_2 \in C_{c,2}^{(k)}\}$, and $v_{k,(2l_1+l_2-2)}$ to $x_{l_1,l_2}^{(k)}$, for each $k \in [t]$ and $l_1, l_2 \in [2]$, where $\mathbb{1}\{A\}$ is the indicator whose value is 1 if the statement A is true, and 0 otherwise. □

Appendix E
Detailed explanation on the variants of qELiCiT to real-world applications (Related to Section V)

We present the details of variants of ELiCiT for matrix completion and neural-network compression.

Algorithm 3: The Overall Training Process of qELICIT++

```
   Input: (a) a masked matrix $\bar{\mathbf{A}} \in \mathbb{R}^{N_1 \times N_2}$
          (b) a mask $\mathbf{M}$ indicating whether each entry in $\mathbf{A}$ is observed
   Output: the trained parameters of ELICIT (or qELICIT)
1  for $i_1 \leftarrow 1$ to $N_1$ do
2      $I^{(1)}_{i_1} \leftarrow \{e \in [N_2] : \mathbf{M}(i_1, e) = 1\}$
3  for $i_2 \leftarrow 1$ to $N_2$ do
4      $I^{(2)}_{i_2} \leftarrow \{e \in [N_1] : \mathbf{M}(e, i_2) = 1\}$
5  $\Theta \leftarrow$ initialized parameters of a qELICIT++ model
6  while not converged do
7      $\hat{A} \leftarrow$ predict($\Theta$)          ▷ refer to the subroutine below
8      Compute $\mathcal{L}(\hat{\mathbf{A}}, \Theta)$ using Eq. (4)
9      Update $\Theta$ to minimize $\mathcal{L}(\hat{\mathbf{A}}, \Theta)$, i.e., $\Theta \leftarrow$
         $\Theta - \alpha \cdot \frac{\partial}{\partial \Theta}\mathcal{L}(\hat{\mathbf{A}}, \Theta)$
10 return $\Theta$
11 Subroutine predict($\Theta$)
12     for $j \leftarrow 1$ to 2 do
13         for $i \leftarrow 1$ to $N_j$ do
14             for $k \leftarrow 1$ to $r$ do
15                 $l_* \leftarrow \mathrm{argmin}_{l \in [2^q]} \left\{ \left| c^{(j)}_{k,l} - \mathbf{F}^{(j)}_k(i) \right| \right\}$
16                 $l'_* \leftarrow \mathrm{argmin}_{l \in [2^q]} \left\{ \left| c'^{(j)}_{k,l} - \mathbf{F}'^{(j)}_k(i) \right| \right\}$
17                 $\mathbf{Q}^{(j)}_{i,k} \leftarrow \left( \mathbf{F}^{(j)}_k(i) + c^{(j)}_{k,l_*} \right) - \tilde{\mathbf{F}}^{(j)}_k(i)$
18                 $\mathbf{Q}'^{(j)}_{i,k} \leftarrow \left( \mathbf{F}'^{(j)}_k(i) + c'^{(j)}_{k,l'_*} \right) - \tilde{\mathbf{F}}'^{(j)}_k(i)$
19     for $j \leftarrow 1$ to 2 do
20         for $i \leftarrow 1$ to $N_j$ do
21             for $k \leftarrow 1$ to $r$ do
22                 $\mathbf{P}^{(j)}_{i,k} \leftarrow$
                     $\mathbf{Q}^{(j)}_{i,k} + |I^{(j)}_i|^{-1/2} \sum_{e \in I_i} (\mathbf{Q}'^{(3-j)}_{e,k} - 0.5)$
23     for $i_1 \leftarrow 1$ to $N_1$ do
24         for $i_2 \leftarrow 1$ to $N_2$ do
25             for $k \leftarrow 1$ to $r$ do
26                 $x_k \leftarrow (1 - \mathbf{P}^{(1)}_{i_1,k})(1 - \mathbf{P}^{(2)}_{i_2,k})v_{k,1} + (1 -$
                     $\mathbf{P}^{(1)}_{i_1,k})\mathbf{P}^{(2)}_{i_2,k}v_{k,2} + \mathbf{P}^{(1)}_{i_1,k}(1 - \mathbf{P}^{(2)}_{i_2,k})v_{k,3} +$
                     $\mathbf{P}^{(1)}_{i_1,k}\mathbf{P}^{(2)}_{i_2,k}v_{k,4}$
27             $\hat{\mathbf{A}}(i_1, i_2) \leftarrow g(x_1, \cdots, x_r)$
28     return $\hat{\mathbf{A}}$
```

A. *qELICIT++ for Matrix Completion*

For the matrix completion problem, we present qELICIT++, an extended version of qELICIT that incorporates implicit features motivated from SVD++ [11], an extended version of SVD. Specifically, SVD++ with r features makes the predictions $\hat{\mathbf{A}}(i,j)$ for the missing entries using the following equation, where the terms in green are also used to predict the value in the original SVD, and the terms in red are the newly added term in SVD++:

$$\hat{\mathbf{A}}(i_1, i_2) = \mu + a_{i_1} + b_{i_2} + n_{i_1}^T m_{i_2} + \left(n_{i_1} + |I_{i_1}|^{-\frac{1}{2}} \sum_{i_2 \in I_{i_1}} z_{i_2} \right)^{\top} m_{i_2},$$

where μ is the average value of all the visible entries, $a_{i_1} \in \mathbb{R}$ is the bias for the i_1-th row, $b_{i_2} \in \mathbb{R}$ is the bias for the i_2-th column, $n_{i_1} \in \mathbb{R}^r$ is the feature vector of the i_1-th row, $m_{i_2} \in \mathbb{R}^r$ is the feature vector of the i_2-th column, I_{i_1} is the set of column indices of the visible entries whose row index is i_1, and $z_{i_2} \in \mathbb{R}^r$ is the implicit feature vector of the i_2-th column for capturing implicit ratings.

In Algorithm 3, we provide the pseudocode for the overall training process of qELICIT++. For qELICIT++, we use additional parameters to capture implicit ratings: $\mathbf{F}'^{(j)}_k(i_j)$ for the implicit feature of index i_j in mode j for each j and i_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 $i_j \in [N_j]$. The prediction process is similar to the approximation process of qELICIT. For each feature $\mathbf{F}^{(j)}_k(i_j)$ and implicit feature $\mathbf{F}'^{(j)}_k(i_j)$, qELICIT++ conducts the quantization process as shown in lines 3-4 of Algorithm 2 and obtains the quantized features $\mathbf{Q}_{j,k}$ and $\mathbf{Q}'_{j,k}$, respectively (lines 12-18). Subsequently, it merges the quantized feature for the original features and the implicit features using the following equation (lines 19-22) and conducts the rest of the approximation process of qELICIT (lines 24-28):

$$\mathbf{P}^{(j)}_{i,k} = \mathbf{Q}^{(j)}_{i,k} + |I^{(j)}_{i_j}|^{-1/2} \sum_{e \in I_{i_j}} \left(\mathbf{Q}'^{(3-j)}_{e,k} - 0.5 \right), \quad (3)$$

where $I^{(j)}_{i_j}$ is the set of indices in the $(3-j)$-th mode of the visible entries whose mode-j index is i_j.

When training qELICIT++, it is crucial to prevent overfitting, since our objective in this problem is to predict the values of unseen entries, rather than precisely approximating those of the observed entries during training. Specifically, we applied L_2 regularization to qELICIT++. By including the regularization term, the training loss function $\mathcal{L}$ of qELICIT++ can be represented as follows:

$$\begin{aligned}\mathcal{L}(\hat{\mathbf{A}}, \Theta) &= \|\mathbf{M} \odot (\bar{\mathbf{A}} - \hat{\mathbf{A}})\|_F^2 + \lambda_1 \cdot \sum_{k=1}^{r} \sum_{l=1}^{4} v_{k,l}^2 \\ &+ \lambda_2 \cdot \sum_{j=1}^{2} \sum_{i=1}^{N_i} \sum_{k=1}^{r} \left(\|\mathbf{F}^{(j)}_k(i) - 0.5\|_F^2 + \|\mathbf{F}'^{(j)}_k(i) - 0.5\|_F^2 \right) \\ &+ \lambda_3 \cdot \sum_{j=1}^{2} \sum_{i=1}^{N_i} \sum_{k=1}^{r} \sum_{l=1}^{2^q} \left(\left(c^{(j)}_{k,l} - 0.5 \right)^2 + \left(c'^{(j)}_{k,l} - 0.5 \right)^2 \right),\end{aligned} \quad (4)$$

where $\lambda_1, \lambda_2, \lambda_3 \in \mathbb{R}^+$ are the hyperparameters to control the scales of the regularization terms. For qELICIT++, we use $g = \mathrm{sum}(\cdot)$ instead of the reduce function introduced in Section IV-C5, since the performance difference was negligible in our preliminary studies.

B. TFW-*qELICIT for Neural-network Compression*

For the neural-network compression problem, we present TFW-qELICIT, an extended version of qELICIT inspired by TFWSVD [30]. The training process of TFW-qELICIT consists of two steps. First, it compresses each parameter represented as a tensor in Θ using qELICIT and replaces Θ to $\Theta(\Phi)$. Next, it fine-tunes Φ using the function $p(\cdot)$ to optimize the performance of the compressed model.

In the first step, instead of the approximation error, which is the original objective function of qELICIT, we used

Algorithm 4: Overview of TFW-qELıCıT

```
Input: (a) a neural network $f_\Theta$ parameterized by the trained parameters
       (b) a function $p(\cdot)$ evaluates the performance
Output: the compersssed parameters $\Phi$ from $\Theta$ by TFW-qELıCıT
1  $\Phi \leftarrow \{\}$
2  foreach $\mathcal{X} \in \Theta$ do
3      Compute $\mathcal{W}_\mathcal{X}$ using Eq. (6)
4      $\Psi \leftarrow$ initialized parameters of a qELıCıT model
5      while not converged do
6          $L \leftarrow 0$
7          foreach $(i_1, \cdots, i_d) \in [N_1] \times \cdots [N_d]$ do
8              $\tilde{x} \leftarrow$ APPROXIMATE$((i_1, \cdots, i_d), \Psi)$ ▷ Algorithm 2
9              $e \leftarrow e + \mathcal{W}_\mathcal{X}(i_1, \cdots, i_d) \cdot (\mathcal{X}(i_1, \cdots, i_d) - \tilde{x})^2$
10         Update $\Psi$ to minimize $e$, i.e., $\Psi \leftarrow \Psi - \alpha \cdot \frac{\partial e}{\partial \Psi}$
11     $\Phi(\mathcal{X}) \leftarrow \Psi$
12 while not converged do
13     $f_\Phi \leftarrow$ a compressed neural network parameterized by $\Phi$
14     Update $\Phi$ to maximize $p(f_{\Theta(\Phi)})$
15 return $\Phi$
```

the weighted approximation error introduced in TFWSVD. Specifically, the weighted approximation error $\mathcal{L}_{\text{approx}} : \mathbb{R}^{N_1 \times \cdots \times N_d} \rightarrow \mathbb{R}^+$ is defined as

$$\mathcal{L}_{\text{approx}}(\tilde{\mathcal{X}}) = \|\mathcal{W}_\mathcal{X} \odot (\mathcal{X} - \tilde{\mathcal{X}}) \odot (\mathcal{X} - \tilde{\mathcal{X}})\|_1, \quad (5)$$

where $\mathcal{W}_\mathcal{X} \in \mathbb{R}^{N_1 \times \cdots \times N_d}$ is a non-negative tensor storing the weight of each entry in $\mathcal{X}$. When every entry in $\mathcal{W}_\mathcal{X}$ is 1, the above equation is equivalent to the original approximation error $\|\mathcal{X} - \tilde{\mathcal{X}}\|_F^2$. For TFWSVD and TFW-qELıCıT, the empirical Fisher information for each tensor-shaped parameter $\mathcal{X} \in \Theta$ estimated as Eq. (6) is used as the weight $\mathcal{W}_\mathcal{X}$ of the parameter $\mathcal{X}$.

$$\mathcal{W}_x = \frac{1}{|D|} \sum_{i=1}^{|D|} \left(\frac{\partial}{\partial \mathcal{X}} \mathcal{L}_{\text{train}}(d_i; \mathcal{X}) \right)^2, \quad (6)$$

where D is a training dataset, d_i is the i-th data in D, and $\mathcal{L}_{\text{train}}(d_i; \mathcal{X})$ is the output of training loss function where the input data is d_i.

The process for the second step follows the learning method of a typical neural network. Unlike qELıCıT++, we utilized the reduce function introduced in Section IV-C5 to TFW-qELıCıT. Thus, the model design of TFW-qELıCıT is identical to qELıCıT, while the objective function differs from that in TFW-qELıCıT. In Algorithm 4, we provide the overall training procedure of TFW-qELıCıT.

Appendix F
Detailed Experimental Settings

A. Tensor Compression (Related to Section VI-A)

Datasets: In Table IV, we provide some statistics and semantics of the datasets used in the paper.

Training Protocol: For the decomposition-based methods trained by the alternating least squares (ALS) process, we stopped the training process when the fitness gain from an epoch is smaller than 10^{-4}. For deep-learning-based methods and qELıCıT with the gradient descent optimization process, we stopped the training process when the fitness gained from τ epochs was less than 10^{-4} or when the number of epochs reaches e, where $\tau, e \in \mathbb{N}$ are the hyperparameters depending on the methods.

Hyperparameter Settings: For NeuKron and TensorCodec, we chose the learning rate values, among $\{1, 0.1, 0.01, 0.001\}$, that maximize the fitness for each setting. For the tolerance τ and the number of epochs e, we followed the settings in [19]. Specifically, we set τ to 10 for the `Absorb`, `Action`, `Activity` datasets and 100 for the other datasets. In addition, we set the maximum number of epochs e to 500 for `Stock`, 1000 for `PEMS`, 1400 for `Airquality`, 1500 for `NYC`, and 5000 for the other datasets. For qELıCıT, we used two learning rates lr_1 and lr_2, where lr_1 is the learning rate for the values of the reference states, and lr_2 is the learning rate for the other learnable parameters. We chose both learning rate values, among $\{1, 0.1, 0.01, 0.001\}$, that maximize the fitness for each setting. For all the datasets, we set τ to 10 and e to 500. In Table V, for all methods, we provide the settings of the hyperparameters that directly control the trade-off between approximation error and compressed size.

B. Matrix Completion (Related to Section VI-G)

Datasets: We considered 5 real-world datasets that contain ratings of movies provided by users. Basic statistics of the dataset are provided in Table VI. The train/valid/test split ratio is 7:1:2 for the `ML-100K`, `ML-1M`, `ML-10M` datasets, and 8:1:1 for the `Douban` and `Flixster` datasets. For `ML-100K`, we used the train/test split used in [33], and then split the train and validation set again from the training set with a 7:1 ratio. For `ML-1M` and `ML-10M`, we used a random split of ratings with a 7:1:2 ratio. For `Douban` and `Flixster`, we used the train/test split provided in the official implementation of [34], and then we split the train and validation set again from the training set with an 8:1 ratio.

Evaluation Protocol: For `SVD`, `SVD++`, and qELıCıT++, we stopped the training process when the RMSE on the validation set did not decrease for 100 epochs. For the other methods, we stopped the training process when the RMSE on the validation set was not decreased for 10 epochs or when the process was performed for 100 epochs since they required much more training time per epoch, compared to SVD, SVD++, and qELıCıT++. To find the optimal hyperparameters, we used Bayesian optimizations on the validation set with 200 trials. Specifically, we searched for the optimal values of the regularization terms in [1e-4, 1e4] and the optimal values of the learning rates in [1e-4, 1e-0]. Similar to qELıCıT for tensor compression, we used different learning rates for the values of the reference states and the other learnable parameters.

C. Neural-network Compression (Related to Section VI-G)

Subtasks: In order to evaluate the performances, we used 7 subtasks from the GLUE benchmark listed below:

TABLE IV
BASIC STATISTICS AND SEMANTICS OF REAL-WORLD DATASETS USED IN THE PAPER. IN THIS TABLE, DENSITY STANDS FOR THE PROPORTION OF NON-ZERO ENTRIES.

Order	Name	Shape	Description of value	#Entries	Density
4	Absorb	192 (longitude) × 288 (latitude) × 30 (altitude) × 120 (time)	Measurement of aerosol absorption	199.1M	1.000
	NYC	265 (source) × 265 (destination) × 28 (day) × 35 (month)	Count of trip records	68.8M	0.118
3	Action	100 (frame) × 570 (feature) × 567 (video)	Extracted feature value	32.3M	0.393
	Activity	337 (frame) × 570 (feature) × 320 (video)	Extracted feature value	61.5M	0.569
	Airquality	$5,600$ (hour) × 362 (location) × 6 (pollutant)	Measurement of air pollutants	12.2M	0.917
	PEMS	963 (station) × 144 (timestamp) × 440 (day)	Rate of use of car lanes	61.0M	0.999
	Stock	$1,317$ (time) × 88 (feature) × 916 (stock)	Feature value of stocks	106.2M	0.816
	Uber	183 (date) × 24 (hour) × $1,140$ (latitude)	Count of Uber pickups	5.0M	0.138

TABLE V
THE SETTINGS OF HYPERPARAMETERS THAT DIRECTLY CONTROL THE TRADE-OFF BETWEEN APPROXIMATION AND COMPRESSED OUTPUT SIZE.

Method	Hyperparameters	Datasets							
		Absorb	NYC	Action	Activity	Airquality	PEMS	Stock	Uber
qELiCiT	r	4, 10, 16, 20	4, 10, 14, 20	10, 18, 28, 38	10, 20, 26, 38	6, 14, 20, 28	4, 6, 10, 14, 20	4, 6, 8, 10	10, 20, 30, 40
TensorCodec	(TT rank, Hidden dimension)	(6, 4), (4, 8), (6, 10), (8, 11)	(2, 5), (5, 7), (6, 9), (10, 9)	(6, 8), (11, 9), (13, 11), (13, 14)	(8, 6), (12, 8), (9, 13), (11, 15)	(7, 11), (7, 20), (8, 25), (17, 21)	(6, 5), (5, 9) (7, 11), (14, 8)	(7, 3), (5, 7), (8, 8), (8, 10)	(7, 8), (7, 13), (12, 13)
NeuKron	Hidden dimension	5, 8, 11, 14	5, 8, 11, 14	10, 15, 19, 23	10, 15, 19, 23	14, 23, 29, 35	7, 11, 14, 17	5, 8, 11, 14	11, 16, 21, 24
CPD	Rank	1, 3, 5, 10, 14, 18, 22, 25	1, 2, 4, 7, 14, 21, 28, 35	1, 3, 6, 10, 15, 20, 25, 30, 35	1, 3, 5, 11, 17, 23, 29, 35, 41	1, 2, 3, 4, 5, 10, 15, 20, 25	1, 2, 3, 4, 5, 6, 10, 15, 20, 25	1, 2, 3, 4, 5, 6, 9, 13	1, 2, 3, 4, 5, 6, 15, 20, 25, 27
Tucker	Rank	1, 2, 3, 4, 7, 9, 11, 13, 15	1, 2, 4, 6, 8, 10, 12	1, 3, 6, 9, 14, 19, 24, 29	1, 3, 5, 10, 16, 22, 28, 33	1, 2, 3, 4, 5	1, 2, 3, 4 6, 11, 16, 20	1, 2, 3, 4 5, 6, 9, 12	1, 2, 3, 4, 5, 9, 11, 13, 15
TTD	Rank	1, 2, 3, 4, 6, 8, 10, 12	1, 2, 3, 4, 8, 11	1, 2, 3, 4, 9, 14, 19, 24, 29	1, 2, 3, 4, 10, 16, 22, 28, 33	1, 2, 3, 4, 7, 10, 13, 16, 19	1, 2, 3, 4, 6, 9, 12, 15, 18, 20	1, 2, 3, 4, 5, 6, 9, 12	1, 2, 3, 4, 9, 11, 13, 15
TRD	Target error	0.9, 0.8, 0.72, 0.6	0.8, 0.6, 0.5, 0.43, 0.41, 0.33, 0.27	0.6, 0.47, 0.43, 0.395, 0.32, 0.28, 0.25, 0.23, 0.22	0.7, 0.6, 0.5, 0.4, 0.38, 0.34, 0.3, 0.28, 0.26	0.6, 0.55, 0.5, 0.46, 0.36, 0.33, 0.31	0.7, 0.65, 0.6, 0.57, 0.44	1.4, 1.3, 1.2, 1.1, 1.05, 0.85, 0.78, 0.71	0.6, 0.45, 0.41, 0.29, 0.28

TABLE VI
BASIC STATISTICS OF MATRIX COMPLETION DATASETS CONSIDERED IN THE PAPER.

Name	Shape	# of ratings	Sparsity	Range of values
ML-10M	71,567 × 10,677	10,000,054	0.0131	0.5, 1, $\cdots$, 4.5, 5
ML-1M	6,040 × 3,952	1,000,209	0.0419	1, 2, 3, 4, 5
ML-100K	943 × 1,682	100,000	0.0630	1, 2, 3, 4, 5
Douban	3,000 × 3,000	136,891	0.0152	1, 2, 3, 4, 5
Flixster	3,000 × 3,000	26,173	0.0029	0.5, 1, $\cdots$, 4.5, 5

- CoLA is a subtask to predict whether the given sentence is grammatically correct or not.
- When two hypotheses are given, the goal of MNLI is to identify whether the first hypothesis sentence entails or contradicts the second hypothesis sentence.
- MRPC is a subtask that predicts whether the given two sentences are paraphrases of each other or not.
- Each query of QQP and STSB consists of two sentences, and the models need to predict the similarity of the two sentences. QQP is a binary classification problem to predict whether the given questions are similar to each other or not. The objective of STSB is to maximize the correlation between the ground-truth similarity scores and the predicted similarity scores.
- QNLI is a subtask to predict whether the second given sentence is answerable to the question in the first given sentence.
- The sentences in SST-2 are movie reviews. The goal of the task is to classify whether the sentiment of the given movie review is positive, negative, or neutral.

Evaluation Protocol: For the target neural network, we used BERT$_{base}$ [36], which is a transformer-based model containing 109.5M parameters. To initialize the network, we used the publicly available pre-trained weights from HuggingFace[2]. To fine-tune the performance of BERT$_{base}$ for each downstream task, we selected the optimal hyperparameters maximizing the performance on the downstream task. Specifically, we chose the initial learning rate among {2e-5, 3e-5, 5e-5}, the weight decay coefficient between {0, 0.01}, and the number of epochs among {2, 3, 4}. During training, we decayed the learning rate linearly so that the learning rate at the end of the training became 0. From the fine-tuned parameters, we compressed the trained parameters using the compression algorithms and then fine-tuned the compressed parameters to optimize their performance. For SVD and FWSVD, we used the PyTorch library to perform matrix decomposition. For TFWSVD, TFW-qELiCiT, and TFW-ELiCiT, we trained the model through gradient-decent updates. Specifically, For TFWSVD, we optimized the model for $50,000$ steps using the optimizer suggested in [30] with the learning rate 2e-5. For TFW-qELiCiT, we optimized the model for $5,000$ steps using the Adam optimizer with the learning rate 1e-3 for the values of the reference states and 1e-2 for the other learnable parameters. After compression, we fine-tuned the compressed parameters with the same procedure as that of BERT$_{base}$. As in [14], [30], we reported the performance on the validation set.

[2]https://huggingface.co/bert-base-uncased

TABLE VII

QELICIT++ OUTPERFORMS THE AUTO-ENCODER-BASED COMPETITORS EVEN WITH LARGER BUDGET RANGES. WE MARKED THE CASES IN BOLD WHEN THE RMSE OF OUR METHOD IS LOWER THAN THE LOWEST RMSE AMONG THE AUTO-ENCODER-BASED METHODS.

(a) Flixster

Method	Budget	RMSE
qELiCiT++ (Proposed)	16N	**0.884±0.002**
	32N	**0.885±0.003**
	64N	**0.883±0.002**
	128N	**0.886±0.001**
SparseFC	256N	0.984±0.002
	512N	0.983±0.002
GLocal-K	256N	0.980±0.000
	512N	0.979±0.000

(b) Douban

Method	Budget	RMSE
qELiCiT++ (Proposed)	16N	**0.726±0.001**
	32N	**0.729±0.003**
	64N	**0.726±0.001**
	128N	**0.723±0.000**
GLocal-K	256N	0.750±0.015
	512N	0.743±0.011
SparseFC	256N	0.731±0.001
	512N	0.730±0.001

(c) ML-100K

Method	Budget	RMSE
qELiCiT++ (Proposed)	16N	0.926±0.011
	32N	0.902±0.004
	64N	**0.899±0.001**
	128N	**0.899±0.001**
GLocal-K	256N	0.943±0.003
	512N	0.918±0.001
SparseFC	256N	0.905±0.001
	512N	0.903±0.000

(d) ML-1M

Method	Budget	RMSE
qELiCiT++ (Proposed)	16N	**0.850±0.005**
	32N	**0.846±0.004**
	64N	**0.838±0.001**
	128N	**0.835±0.001**
GLocal-K	256N	0.911±0.008
	512N	0.902±0.004
SparseFC	256N	0.882±0.011
	512N	0.857±0.002

(e) ML-10M

Method	Budget	RMSE
qELiCiT++ (Proposed)	16N	**0.793±0.001**
	32N	**0.796±0.002**
	64N	**0.785±0.001**
	128N	**0.794±0.003**
GLocal-K	256N	O.O.M
	512N	O.O.M
SparseFC	256N	0.872±0.004
	512N	0.900±0.071

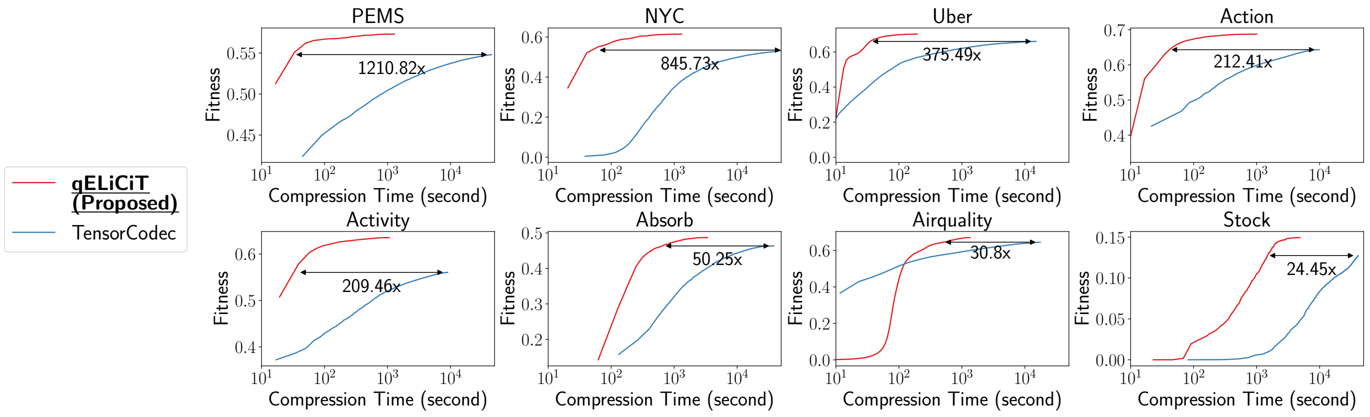

Fig. 9. qELICIT shows significantly faster compression speed than TENSORCODEC. The compression speed of qELICIT in reaching the maximum fitness achieved by TENSORCODEC is up to 1200× faster than that of TENSORCODEC with a similar compression size.

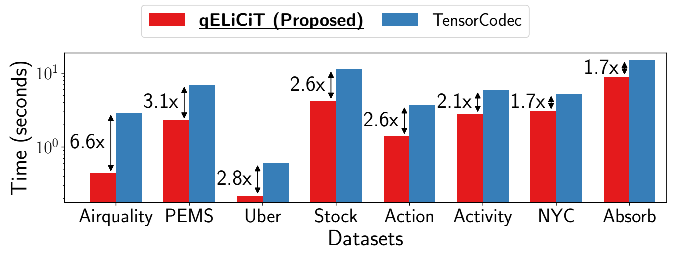

Fig. 10. qELICIT shows significantly faster approximation speed than TENSORCODEC. The approximation speed of qELICIT is up to 6.6× faster than that of TENSORCODEC with a similar compression size.

APPENDIX G ADDITIONAL EXPERIMENT RESULTS (RELATED TO SECTION VI)

A. Q3. Application (Matrix Completion)

We provide the results with additional budget ranges $\{256N, 512N\}$ for the auto-encoder-based methods in Table VII. In most cases, qELICIT++ achieved lower RMSE than the auto-encoder-based competitors even with the larger budget ranges.

B. Compression Time and Approximation Time

In Section VI-C, we conducted a comparison of the compression speed for qELICIT and its competitors. Additionally, for qELICIT and TENSORCODEC, we further measured how fitness changes over time during the compression process and approximation time for the entire tensor. For a fair comparison, we set the compressed output sizes to be similar to the smallest output sizes of TENSORCODEC in Figure 3.

Consistent with the result in Section VI-C, the compression speed of qELICIT was significantly faster than that of TENSORCODEC. Specifically, while TENSORCODEC required considerable time to achieve satisfactory accuracy, qELICIT was up to 1200× faster in reaching the maximum fitness achieved by TENSORCODEC. In addition, ELiCiT was up to 6.6 times faster than TensorCodec in terms of approximation speed, as seen in Figure 10.

C. Performance of ELICIT without quantization

We additionally compared ELICIT (without quantization) with decomposition-based competitors. Note that our clustering-based quantization is not compatible with the parameters of TENSORCODEC and NEUKRON, which consist of permutations and LSTM parameters, and it aims to directly replace the LSTM component in the deep-learning-based compression methods, as explained in Section 4.1[3].

As seen in Figure 11, ELICIT outperformed the decomposition-based competitors except for the Uber dataset, even without clustering-based quantization. Furthermore, ELICIT consistently outperformed CPD, which ELICIT theoretically generalizes, on all the datasets. Specifically, it offers up to 1.95× smaller outputs with better fitness (when comparing the result with the third largest size of ELICIT and

[3]While we might consider applying general quantization approaches to the parameters of deep-learning-based methods for further compression, they can also be applied equally to the output of qELiCiT.

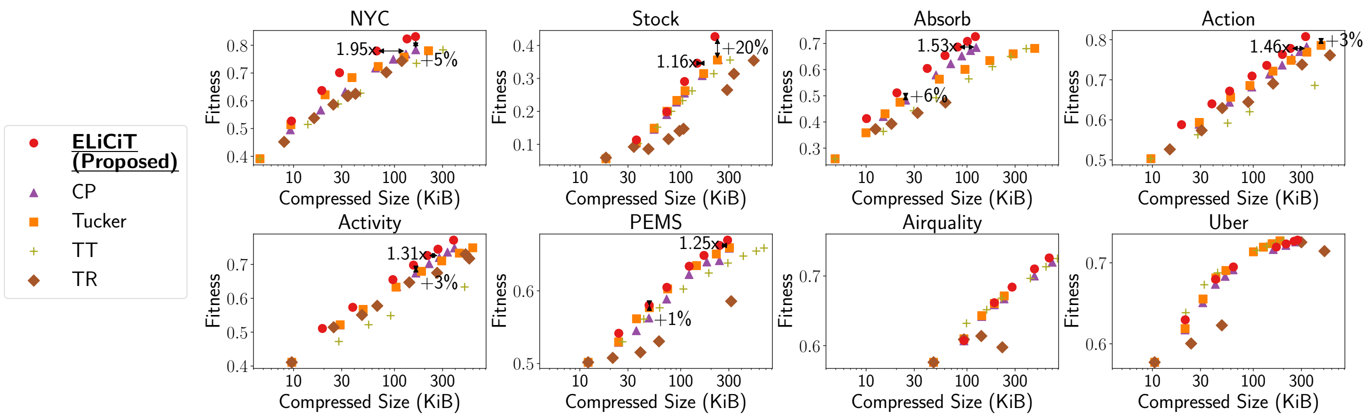

Fig. 11. ELıCıT without quantization still provides more compact and accurate compression of tensors than decomposed- based methods. The compressed size of ELıCıT is up to $1.95\times$ smaller than that of the most compact competitor with similar fitness. Its fitness is up to 20% higher than that of the most accurate baseline while achieving a similar compressed size.

the second largest size of CPD on the `NYC` dataset). For a similarly sized output, it achieves up to 20% better fitness (when comparing the result with the largest sizes of ELıCıT and CPD on the `Stock` dataset). On the `Uber` dataset, TTD showed the best tradeoff, and CP decomposition also exhibited a relatively poor trade-off, similar to ELıCıT.