CoMERA: Computing- and Memory-Efficient Pre-Training via Rank-Adaptive Tensor Optimization

Zi Yang
University at Albany
zyang8@albany.edu

Samridhi Choudhary
Amazon Alexa AI
samridhc@amazon.com

Xinfeng Xie
Meta
xinfeng@meta.com

Cao Gao
Meta
caogao@meta.com

Siegfried Kunzmann
Amazon Alexa AI
kunzman@amazon.com

Zheng Zhang
University of California at Santa Barbara
zhengzhang@ece.ucsb.edu

Abstract

Pre-training large AI models such as deep learning recommendation systems and foundation language (or multi-modal) models costs massive GPUs and computing time. The high training cost has become only affordable to big tech companies, meanwhile also causing increasing concerns about the environmental impact. This paper presents CoMERA, a **Co**mputing- and **M**emory-**E**fficient pre-training method via **R**ank-**A**daptive tensor optimization. CoMERA achieves end-to-end rank-adaptive tensor-compressed training via a multi-objective optimization formulation, and improves the training to provide both high compression ratio and excellent accuracy in the training process. Our optimized numerical computation (e.g., optimized tensorized embedding and tensor-vector contractions) and GPU implementation eliminate part of the run-time overhead in the tensorized training on GPU. This leads to, for the first time, $2-3\times$ speedup per ~~each~~ training epoch compared with standard pre-training. CoMERA also outperforms the recent GaLore in terms of both memory and computing efficiency. Specifically, CoMERA is $2\times$ faster per training epoch and $9\times$ more memory-efficient on a tested six-encoder transformer with single-batch training. We further show that the efficiency can be further boosted by employing low-/mixed-precision low-rank tensor computation. With further GPU and HPC optimization, ~~we are optimistic that~~ CoMERA can dramatically reduce the pre-training cost of large-scale AI foundation models on industry-scale training infrastructure.

1 Introduction

Deep neural networks have gained great success in solving a wide range of practical engineering problems. These approaches usually use a huge number of variables to parametrize a neural network, and require high-volume data sets and massive hardware resources to train the model. For instance, the Deep Learning Recommendation Model (DLRM) released by Meta (which is smaller than the practical model used in Meta's product) [37] has 4.2 billion model parameters; the GPT-3 model [4] has 175 billion parameters. Due to the huge model size, high-performance computing (HPC) platforms are usually required to handle the computation- and memory-expensive training. OpenAI shows that the computing power required for key AI tasks has doubled every 3.4 months [1] since 2012. From 2012 to May 2018, this metric grew by more than $300,000\times$. ChatGPT, which is a fine-tuned version of GPT-3, is estimated to have cost over 10,000 GPUs in the training [3] and over \$4 million in each training session. Meta's largest LLaMA model released Feb. 2023, for example, used 2,048 Nvidia A100 GPUs to train on 1.4 trillion tokens, taking about 21 days [2].

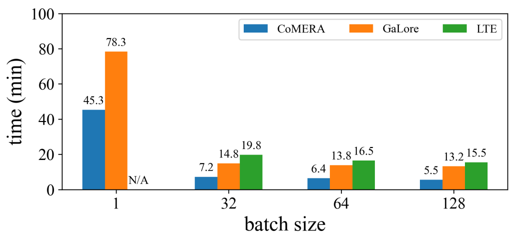
Building ever-larger AI models entails a tremendous amount of energy expenditure and thus carbon emissions, leading to ~~an~~ energy consumption growing at a breathtaking rate. The study [44] estimated that training a transformer for NLP can generate up to 626,155 pounds of CO_2 emissions—roughly equal to the total lifetime carbon footprint of five cars. As a comparison, the average American generates 36,156 pounds of CO_2 emissions in a year. The study in [42] further shows that training GPT-3 takes 1.287 gigawatt hours (or about as much electricity as 120 US homes would consume in a year) and generates 502 tons of carbon emissions (or about as much as 110 US cars emit in a year in the research). Pre-traning a large AI model normally requires many training runs, indicating that the total energy cost can be even tremendously higher.

Large AI models often have much redundancy among the model parameters. Therefore, extensive methods have been developed to reduce the cost of AI inference (e.g., post-training model compression [5, 26, 9], quantization [15, 18], pruning [36, 34], and knowledge distilation [19]). However, pre-training large AI models remains an extremely challenging task. Actually, the pre-training cost of state-of-the-art AI models (e.g., language/vision foundation models) are so high that only a small number of big tech companies can afford the cost. While pruning techniques [38, 49] can generate memory-efficient models for inference, they do not always reduce the memory and energy cost of pre-training because the sparsity pattern is unknown *a-priori*. Low-precision arithmetic [22, 28, 14, 45] has been widely used in on-device training. However, the memory reduction of low-precision methods is quite limited, and it is hard to utilize ultra low-precision computing on HPC platforms since the GPU cards only support limited precision formats. Chen [7] successfully extended the idea of robust matrix factorization to reduce the runtime and energy cost of training AI models. Recently, similar low-rank matrix approximation techniques have been applied to pre-train large AI models including large language models [53, 23]. Among them, the GaLore method [53] can pre-train the 7B LLaMMA model on a RTX 4090 GPU using a single-batch setting. However, the single-batch setting can lead to extremely long training time, which is infeasible in practical applications.

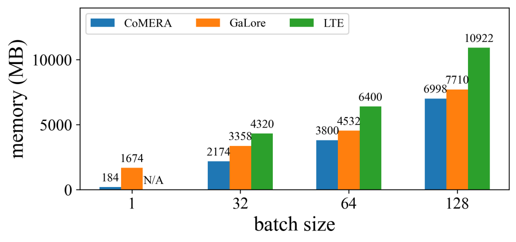
Compared with low-rank matrix approximation, low-rank tensor approximation has achieved much higher compression ratios in post-training compression of various neural networks [39, 30, 51, 46, 24, 26]. This idea has also been studied in fixed-rank end-to-end training [39, 5, 25] and zeroth-order on-device training [55] surpassing low-rank matrix approximation in terms of both accuracy and memory cost. Recently, the tensor-compressed adaptation method [50] has also shown superior performance than the popular LoRA method [21] in fine-tuning large language models (LLMs). A major challenge of tensor-compressed training is the determination of tensor ranks in a training process, which was addressed by the recent Bayesian rank-adaptive training method [16, 17]. However, to achieve reduction in both memory and pre-training time (especially on LLMs), two open problems need to be addressed. Firstly, a more robust rank-adaptive tensor-compressed training model is desired~~,~~ since the method in [16, 17] replies on a heuristic fixed-rank warm-up training. Secondly, while modern GPUs are well optimized for large-size matrix computations in standard neural network training, they are unfriendly for low-rank tensor-compressed training. Specifically, most operations in tensor-compressed training are small-size tensor contractions. These computation patterns can cause significant runtime overhead on GPUs even though the theoretical computing FLOPS of tensor-compressed training is much lower than standard uncompressed training. As a result, as far as we know no papers have reported real training speedup on GPU. This issue was also observed in [20].

Paper Contributions. In this work, we propose CoMERA, a tensor-compressed pre-training method that can achieve, for the first time, *simultaneous reduction of both memory and runtime on GPU*. Our specific contributions are summarized as follows.

- **Multi-Objective Optimization for Rank-Adaptive Tensor-Compressed Pre-Training**. It is crucial to achieve a good balance between the compression ratio and model accuracy in tensor-compressed pre-training. We propose a multi-objective optimization formulation to achieve such a trade-off and to customize the model for a specific resource requirement. One by-product of this method is the partial capability of automatic architecture search: some layers are ~~are~~ identified as unnecessary and can be completely removed by the rank-adaptive training.
- **Performance Optimization of Tensor-Compressed Training.** While tensor-compressed training can greatly reduce the memory cost and computing FLOPS, it often slows down the practical training on GPU. This is because GPU is well optimized for standard training with large-size matrix computations, but tensorized pre-training involves lots of small-size low-rank tensor operations which can lead to much unnecessary back-end runtime overhead. In this work, we propose three approaches to achieve real pre-training speedup on GPU: ① optimizing the lookup process of



(a) Training time per epoch.



(b) Peak memory consumption.

Figure 1: Training time and total memory cost of CoMERA, GaLore [53] and LTE [23] on a six-encoder transformer with varying batch sizes. The experiment is done on Nvidia RTX 3090 GPU.

tensorized embedding tables, ② optimizing the tensor-vector contractions in both forward and backward propagation, ③ eliminating the GPU backend overhead via CudaGraph.

- **Experimental Results.** We evaluate the performance of our method on the end-to-end training of a medium-size transformer model with six encoders and the deep learning recommendation system model (DLRM) released by Meta. On these benchmarks, our method achieves $80\times$ compression ratio on the transformer model and $99\times$ compression on the DLRM model while maintaining the testing accuracy compared with standard uncompressed training methods. Due to the computationally efficient low-rank tensor-network contractions used in the forward and backward propagation, CoMERA also achieves $2-3\times$ speedup per training epoch compared with standard training methods (i.e., uncompressed training using Adam) on the transformer model. We also show that further speedup can be obtained by employing mixed-precision low-rank tensor contractions in CoMERA.

Figure 1 shows the comparison of our CoMERA with GaLore [53] and the recent LoRA-based pre-training method LTE [23] on the six-encoder transformer. The experiment is done on Nvidia RTX 3090 GPU. When data and back-end memory cost are considered, CoMERA's memory consumption is $9\times$ less than GaLore when using the single-batch training adopted in [53], and it uses the least memory under all batch sizes. Furthermore, our method is $2-3\times$ faster than GaLore and LTE in each training epoch, although CoMERA has not yet been fully optimized on GPU.

While this work focuses on reducing the memory and computing cost of pre-training, it can also reduce the communication cost by orders of magnitude: only low-rank tensorized model parameters and gradients need to be communicated in a distributed setting. The CoMERA framework can also be implemented on resource-constraint edge devices to achieve energy-efficient on-device learning.

2 Background and Literature Review

Notation. Throughout the paper, lower-case letters (e.g., a) denote scalars; lower-case bold letters (e.g., $\mathbf{a}$) denote vectors; upper-case bold letters (e.g., $\mathbf{A}$) denote matrices; upper-case calligraphic bold letters (e.g., $\boldsymbol{\mathcal{A}}$) denote tensors. $\mathbf{1}$ or $\mathbf{0}$ are a vector/matrix whose entries are all 1 or 0, respectively. $\mathbf{I}_n$ is an n-by-n identity matrix. We use $[n]$ to denote the set of integers $\{1, 2, \ldots, n\}$. For a vector $\mathbf{v}$, $\|\mathbf{v}\|$ and $\|\mathbf{v}\|_1$denote its Euclidean norm and 1-norm, respectively. $\|\mathbf{v}\|_0$ is the 0-norm, that is the number of nonzero entries in $\|\mathbf{v}\|_0$. For a matrix $\mathbf{A}$, $\mathbf{A}^T$ denotes its transpose; $\|\mathbf{A}\|$ represents the Frobenius norm, and the spectrum norm $\|\mathbf{A}\|_2$ is the largest singular value of $\mathbf{A}$. We use MATLAB-style indexing to denote submatrices. For instance, $\mathbf{A}(i_1 : i_2, j_1 : j_2)$ denotes the submatrix consisting of the rows from i_1 to i_2 and the columns from j_1 to j_2.

2.1 Tensor and Tensor Contraction

Tensors are generalizations of matrices to higher orders and can be represented by multi-dimensional data arrays [27, 29]. A real tensor of order d and dimension $n_1, \ldots, n_d$ can be denoted as $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$. In tensor networks, the order-d tensor $\boldsymbol{\mathcal{A}}$ is represented as a node with d edges. Order-1 tensors are vectors, and order-2 tensors are matrices. Tensors of various orders with corresponding tensor network representations are illustrated in Fig. 2 (a). The tensor

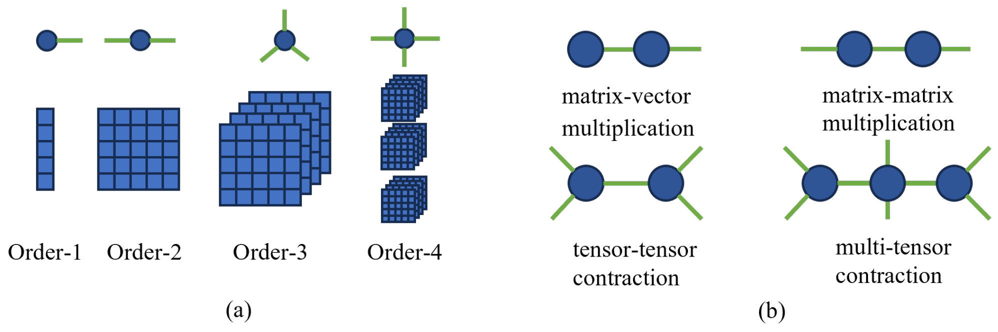

Order-1
Order-2
Order-3
Order-4
(a)
matrix-vector multiplication
matrix-matrix multiplication
tensor-tensor contraction
multi-tensor contraction
(b)

Figure 2: (a) Tensors. (b) Tensor contractions.

$\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ is indexed as $\mathcal{A} = (a_{i_1 \cdots i_d})_{1 \le i_j \le n_j}$. The Frobenius norm of tensor $\mathcal{A}$ is defined as $\|\mathcal{A}\| := \sqrt{\sum_{i_1,\ldots,i_d}^{n_1,\ldots,n_d} a_{i_1 \cdots i_d}^2}$.

Tensor Contraction. Let $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ and $\mathcal{B} \in \mathbb{R}^{l_1 \times l_2 \times \cdots \times l_m}$ be two tensors with $n_s = l_t$. The tensor contraction product between $\mathcal{A}$ and $\mathcal{B}$ on indices s, t is an order-$(d+m-2)$ tensor $\mathcal{C} = \mathcal{A} \times_{s,t} \mathcal{B}$ with dimension $\Pi_{i \neq s} n_i \times \Pi_{j \neq t} l_j$ whose entries are

$$c_{(i_p)_{p \neq s},(j_p)_{p \neq t}} = \sum_{i_s = j_t = 1}^{n_s} a_{i_1 \cdots i_s \cdots i_m} b_{j_1 \cdots j_t \cdots j_k}. \tag{1}$$

The tensor contraction for a pair of indices can be naturally generalized to multiple pairs. For two index sets $S = \{s_1, \ldots, s_p\}, T = \{t_1, \ldots, t_p\}$ with same cardinality and $n_{s_i} = l_{t_i}$, the tensor contraction $\mathcal{A} \times_{S,T} \mathcal{B}$ produces an order-$(d+m-2p)$ tensor of dimension $\Pi_{i \notin S} n_i \times \Pi_{j \notin T} l_j$ that is similarly defined as (1). Figure 2 has illustrated some tensor contractions. Generally, we use the PyTorch einsum notation to denote the operation among multiple tensors. An einsum is in the form of

$$\mathcal{B} = \textbf{einsum}(S_1, \ldots, S_m \Rightarrow T, [\mathcal{A}_1, \ldots, \mathcal{A}_m]), \tag{2}$$

where each S_i is a string of characters that specifies the dimension of $\mathcal{A}_i$. Let $n_1, \ldots, n_m$ be dimensions corresponding to the characters in the string T, then the output tensor $\mathcal{B}$ has the dimension $n_1, \ldots, n_m$ and it is obtained by summing over all other dimensions that are not in T. In the following, we show a few commonly used einsum operations.

- The matrix product between $\mathbf{A} \in \mathbb{R}^{m \times k}, \mathbf{B} \in \mathbb{R}^{k \times n}$ can be written as
$$\mathbf{C} = \mathbf{AB} = \textbf{einsum}(mk, kn \Rightarrow mn, [\mathbf{A}, \mathbf{B}]).$$
- For the batched matrices $\mathcal{A} \in \mathbb{R}^{b \times m \times k}, \mathcal{B} \in \mathbb{R}^{b \times k \times n}$, the batched matrix multiplication is
$$\mathcal{C} = \mathcal{AB} = \textbf{einsum}(bmk, bkn \Rightarrow bmn, [\mathcal{A}, \mathcal{B}]), \quad \text{where } \mathcal{C}[i,:,:] = \mathcal{A}[i,:,:]\mathcal{B}[i,:,:].$$
- The Tensor-Train decomposition as in (3) is
$$\mathcal{A} = \textbf{einsum}(n_1 r_1, \ldots, r_{d-1} n_d \Rightarrow n_1 n_2 \cdots n_d, [\mathcal{G}_1, \mathcal{G}_2 \ldots, \mathcal{G}_d]).$$

2.2 Tensor Decomposition

There exist many tensor decomposition formats, e.g., CP decomposition [27], Tucker decomposition [27], Tensor-Ring decomposition [54]. In this paper, we will mainly use Tensor-Train [41] and Tensor-Train matrix [40] decomposition for compressed neural network pre-training.

The Tensor-Train (TT) decomposition [41] represents the tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ as a set of small-size cores $\mathcal{G}_1, \ldots, \mathcal{G}_d$ such that $\mathcal{G}_i \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$ and

$$\mathcal{A} = \mathcal{G}_1 \times_{3,1} \mathcal{G}_2 \times_{3,1} \cdots \times_{3,1} \mathcal{G}_d. \tag{3}$$

The tuple $(r_0, r_1, \ldots, r_d)$ is the TT rank of the TT decomposition (3) and must satisfy $r_0 = r_d = 1$. Figure 3 shows the TT decomposition of an order-3 tensor and Figure 4(a) shows the TT decomposition in the form of a tensor network.

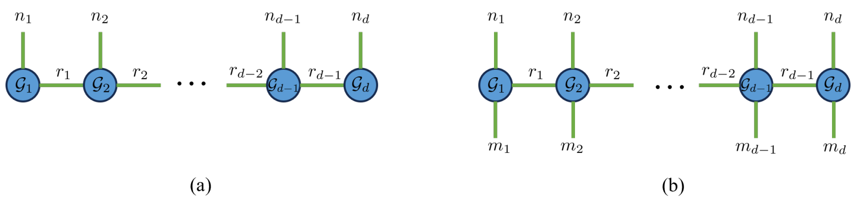


Figure 4: Tensor networks for (a) tensor-train and (b) tensor-train-matrix decompositions.

A related decomposition format is the Tensor-Train-Matrix (TTM) decomposition [40]. It considers the tensor $\boldsymbol{\mathcal{B}}$ of order $2d$ and dimension $m_1 \times \cdots \times m_d \times n_1 \times \cdots \times n_d$. The tensor $\boldsymbol{\mathcal{B}}$ is reshaped to $\widetilde{\boldsymbol{\mathcal{B}}}$ of dimension $m_1 \times n_2 \times \cdots \times m_d \times n_d$. The TTM decomposition represents the tensor $\widetilde{\boldsymbol{\mathcal{B}}}$ as

$$\widetilde{\boldsymbol{\mathcal{B}}} = \boldsymbol{\mathcal{F}}_1 \times_{4,1} \boldsymbol{\mathcal{F}}_2 \times_{4,1} \cdots \times_{4,1} \boldsymbol{\mathcal{F}}_d, \qquad (4)$$

where $\boldsymbol{\mathcal{F}}_i \in \mathbb{R}^{r_{i-1} \times m_i \times n_i \times r_i}$ for $i = 1, \ldots, d$ and $r_0 = r_d = 1$. A tensor-network representation of the TTM decomposition is shown in Figure 4(b).

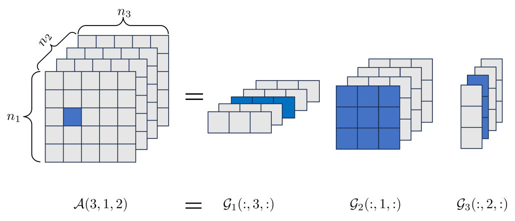


Figure 3: Tensor-Train decomposition.

In tensor-compressed neural networks, large weight matrices are compressed into small tensor cores to reduce memory and computation costs for training and inference. Let $\mathbf{W} \in \mathbb{R}^{m \times n}$ be a weight in the neural network. It is first reshaped to a tensor $\boldsymbol{\mathcal{W}}$ of dimension $m_1 \times \cdots \times m_d \times n_1 \times \cdots \times n_d$, where $m = \Pi_{i=1}^{d} m_i, n = \Pi_{i=1}^{d} n_i$. Then, the TT decomposition (3) or the TTM decomposition (4) are applied to represent $\boldsymbol{\mathcal{W}}$ as small tensor cores. When $d = 1$, the TT decomposition reduces to a low-rank matrix factorization. Compared to low-rank matrix factorization, TT and TTM decompositions can reach much higher compression ratios especially when the ranks are small.

In practice, the weights of linear layers are often compressed into the TT format due to its efficiency in tensor-vector multiplications. The TTM format is more suitable for embedding tables whose dimension is highly unbalanced.

2.3 Related Work

The increasing size of LLMs dramatically boosts the model performance on general tasks and downstream tasks by storing more general knowledge during the pretraining stage. However, it also brings significant challenges, particularly in the huge demands of computing resources and energy consumption for training and inference. As a result, memory- and computing-efficient pre-training has become increasingly important.

Low-rank adapters and sub-space learning. The low-rank adapter (LoRA) was first proposed in [21] to fine-tune large language models. LoRA applies a low-rank adapter to each weight matrix. The low-rank adapters are much smaller than original weights. During training, the original weights are fixed and only the small adapters are trained. This approach dramatically reduces the memory footprint for fine-tuning large ~~language models.~~ LoRA can be integrated with low-precision computation for memory reductions [12]. The works [33, 23] further generalize LoRA to pretrain large AI models from scratch. Subspace learning projects the parameters into a low-rank subspace and train the model in the low-dimensional subspace. Hence, subspace learning significantly reduce the number of trainable parameters, thereby the training memory and costs. We refer to [31, 6, 48] for recent works in subspace learning for AI models.

Memory-efficient optimization. The optimzier Adam is most promising optimizer used to train the state-of-the-art AI models. It stores first-order and second-order statistics to adaptively determine the learning rate, which requires the memory of two times the model size. There exist works focusing reducing the memory footprint of Adam. The work Adafactor [43] factorizes second-order statistics into low-rank factors, achieving a sublinear memory cost. Quantization can be integrated into the Adam optimizer to reduce memory by representing gradients and stored statistics in low-precision. Recent works of quantized optimizers can be found in [11, 32, 8]. GaLore [53] projects the gradient matrix into a low-dimensional subspace. Hence, the Adam optimizer only stores the statistics for the low-dimensional projected gradients, requiring much less space than storing full gradients.

Low-rank tensor-compressed training. Low-rank tensor decomposition has achieved remarkable success in both post-training compression and end-to-end compressed training [40, 16, 17, 47, 35]. Most works focus on fixed-rank training [39, 5, 25], which often produce poor accuracy if the tensor ranks are not properly chosen. Recent works [16, 17] employ Bayesian methods for automatic rank determination in the training process. The work [52] further developed a quantization-aware methods to obtain quantized low-rank tensorized models. While the comparison ratio is high, many existing methods do not necessarily reduce the memory cost because they reconstruct the full-size model parameters in forward and backward propagation. Furthermore, many of them actually have slowed down the training due to the lack of efficient implementation of low-rank tensor operations on GPU.

3 The CoMERA Training Framework

The size of the tensor-compressed neural networks can be adjusted by modifying the tensor ranks. However, it also brings in an important problem: *how can we determine the tensor ranks automatically for a given resource limit?* In this section, we propose to formulate the rank-adaptive tensor-compressed training as a multi-objective optimization problem.

3.1 Multi-Objective Training Model

A Modified TT Representation. We consider the tensor-compressed training for a generic neural network. Suppose that the neural network is parameterized as $f(\mathbf{x}|\{\boldsymbol{\mathcal{G}}_1^i,\ldots,\boldsymbol{\mathcal{G}}_{d_i}^i\}_{i=1}^P)$, where $\{\boldsymbol{\mathcal{G}}_j^i \in \mathbb{R}^{r_{j-1}^i\times n_j^i\times r_j^i}\}_{j=1}^{d_i}$ compress the original weight $\mathbf{W}_i$. Let $\{\mathbf{x}_k, y_k\}_{k=1}^N$ be training data and $\mathcal{L}$ be the loss function. The training is to minimize the following objective function

$$\min_{\{\boldsymbol{\mathcal{G}}_1^i,\ldots,\boldsymbol{\mathcal{G}}_{d_i}^i\}_{i=1}^P} L := \sum_{k=1}^N \mathcal{L}(y_k, f(\mathbf{x}_k|\{\boldsymbol{\mathcal{G}}_1^i,\ldots,\boldsymbol{\mathcal{G}}_{d_i}^i\}_{i=1}^P)). \tag{5}$$

We modify the TT compression, and control the ranks of $\boldsymbol{\mathcal{G}}_1^i,\ldots,\boldsymbol{\mathcal{G}}_{d_i}^i$ by a set of diagonal matrices $\{\mathbf{D}_j^i \in \mathbb{R}^{r_j^i\times r_j^i}\}_{j=1}^{d-1}$. Specifically, the compression of $\mathbf{W}_i$ is

$$\boldsymbol{\mathcal{W}}_i = \boldsymbol{\mathcal{G}}_1^i \times_{3,1} \mathbf{D}_1^i \times_{2,1} \boldsymbol{\mathcal{G}}_2^i \times_{3,1} \cdots \times_{3,1} \mathbf{D}_{d_i-1}^i \times_{2,1} \boldsymbol{\mathcal{G}}_{d_i}^i, \tag{6}$$

where $\boldsymbol{\mathcal{W}}_i$ is the reshape of $\mathbf{W}_i$. After compression, the tensor cores for $\boldsymbol{\mathcal{W}}_i$ have the size

$$S_i = n_1^i\|\mathbf{D}_1^i\|_0 + n_{d_i}^i\|\mathbf{D}_{d_i-1}^i\|_0 + \sum_{j=2}^{d_i-1} n_j^i\|\mathbf{D}_{j-1}^i\|_0\|\mathbf{D}_j^i\|_0. \tag{7}$$

For simplicity, we denote $\boldsymbol{\mathcal{G}} := \{\boldsymbol{\mathcal{G}}_1^i,\ldots,\boldsymbol{\mathcal{G}}_{d_i}^i\}_{i=1}^P$ and $\mathbf{D} := \{\mathbf{D}_1^i,\ldots,\mathbf{D}_{d_i-1}^i\}_{i=1}^P$.

Multi-Objective Optimization. In the pre-training stage we intend to minimize both the loss and tensor-compressed network size, which can be formulated as a multi-objective optimization problem

$$\min_{\boldsymbol{\mathcal{G}},\mathbf{D}} \{L(\boldsymbol{\mathcal{G}},\mathbf{D}), S(\mathbf{D})\}, \tag{8}$$

where $S(\mathbf{D}) := \sum_{i=1}^P S_i(\mathbf{D})$. In most cases, we cannot find a point that minimizes the loss and model size simultaneously. Therefore, we look for a point such that one objective cannot be further improved without sacrificing the other objective. This leads to the definition of Pareto points. A point $(\boldsymbol{\mathcal{G}}^*,\mathbf{D}^*)$ is called a Pareto point of (8) if there does not exist $\boldsymbol{\mathcal{G}}$ and $\mathbf{D}$ such that $L(\boldsymbol{\mathcal{G}},\mathbf{D}) \le L(\boldsymbol{\mathcal{G}}^*,\mathbf{D}^*)$, $S(\mathbf{D}) \le S(\mathbf{D}^*)$, and at least one of inequalities is strict.

3.2 Training Methods

It is normally challenging to directly solve a multi-objective optimization problem [13, 10]. A widely used method is the scalarization method, that converts a mutli-objective optimization to a single-objective one such that its optimizers are Pareto points. In CoMERA, we use different scalarization methods at the early and late stage of training. The late stage is optional, and it can be used to further compress the pre-trained model to enable efficient deployment on resource-constraint platforms.

Algorithm 1 Solve relaxed scalarization problem (13)

```
    Input: Initializations $\mathcal{G}_0, \mathbf{D}_0$, constants $L_0, S_0, w_1, w_2, \rho, \beta$, and an optimization algorithm $\mathcal{O}$.
    Output: Tensor cores $\mathcal{G}_T$ and rank-control parameters $\mathbf{D}_T$.
for $t = 0, \ldots, T-1$ do
    if $w_1(L(\mathcal{G}_t, \mathbf{D}_t) - L_0) \geq w_2(S(\mathbf{D}_t) - S_0)$ then
        The optimization algorithm $\mathcal{O}$ runs one step on the problem (14).
    else
        The optimization algorithm $\mathcal{O}$ runs one step on the problem (15).
    end if
end for
```

Early Stage. At the early stage of CoMERA, aggressively pruning ranks dramatically hurts the convergence. Hence, we start the pre-training with the following linear scalarization formulation [10]

$$\min_{\mathcal{G}, \mathbf{D}} \; L(\mathcal{G}, \mathbf{D}) + \gamma S(\mathbf{D}). \tag{9}$$

It is still hard to solve (9) since $S(\mathbf{D})$ uses $\|\cdot\|_0$ whose derivative is 0 almost everywhere. Therefore, we replace $\|\cdot\|_0$ by the ℓ_1 norm $\|\cdot\|_1$ and get the convex relaxation

$$\hat{S}(\mathbf{D}) := \sum_{i=1}^{P} \left(\hat{S}_i(\mathbf{D}) := \sum_{i=1}^{P} n_1^i \|\mathbf{D}_1^i\|_1 + n_{d_i}^i \|\mathbf{D}_{d_i-1}^i\|_1 + \sum_{j=2}^{d_i-1} n_j^i \|\mathbf{D}_{j-1}^i\|_1 \|\mathbf{D}_j^i\|_1 \right). \tag{10}$$

We note that $\hat{S}(\mathbf{D})$ can be arbitrarily close to 0 while keeping $L(\mathcal{G}, \mathbf{D})$ unchangeable, since the corresponding slices of TT factors can be scaled accordingly. Therefore, a direct relaxation of the scalarization (12) does not permits a minimizer. To address this issue, we add an ℓ_2 regularization $\|\mathcal{G}\|^2 := \sum_{i=1}^{P} \sum_{j=1}^{d_i} \|\mathcal{G}_j^i\|^2$ to the relaxation and get the formulation

$$\min_{\mathcal{G}, \mathbf{D}} \; L(\mathcal{G}, \mathbf{D}) + \gamma \hat{S}(\mathbf{D}) + \beta \|\mathcal{G}\|^2. \tag{11}$$

In practice, optimizing (11) effectively controls tensor ranks without hurting the model performance.

Late Stage (Optional). In the late stage of CoMERA, we have an option to continue training the model towards a preferred loss L_0 and a preferred model size S_0 to make it deployable on a resource-constraint platform. This can be achieved by choosing an achievement scalarization [10] that leads to a Pareto point close to (L_0, S_0):

$$\min_{\mathcal{G}, \mathbf{D}} \; \max\{w_1(L(\mathcal{G}, \mathbf{D}) - L_0), w_2(S(\mathbf{D}) - S_0)\} + \rho(L(\mathcal{G}, \mathbf{D}) + S(\mathbf{D})). \tag{12}$$

Here $w_1, w_2 > 0$ scale the objectives into proper ranges, and $\rho > 0$ is a small constant. After relaxing $S(\mathbf{D})$ to $\hat{S}(\mathbf{D})$ and adding the regularization term, we get the following problem

$$\min_{\mathcal{G}, \mathbf{D}} \; \max\{w_1(L(\mathcal{G}, \mathbf{D}) - L_0), w_2(S(\mathbf{D}) - S_0)\} + \rho(L(\mathcal{G}, \mathbf{D}) + \hat{S}(\mathbf{D})) + \beta \|\mathcal{G}\|^2, \tag{13}$$

where $\beta > 0$ is a positive constant. Note that the $S(\mathbf{D})$ inside $\max$ is not relaxed now for accurate comparisons. When $w_1(L(\mathcal{G}, \mathbf{D}) - L_0) \geq w_2(S(\mathbf{D}) - S_0)$, we consider the following problem

$$\min_{\mathcal{G}, \mathbf{D}} \; w_1(L(\mathcal{G}, \mathbf{D}) - L_0) + \rho(L(\mathcal{G}, \mathbf{D}) + \hat{S}(\mathbf{D})) + \beta \|\mathcal{G}\|^2. \tag{14}$$

When $w_1(L(\mathcal{G}, \mathbf{D}) - L_0) < w_2(S(\mathbf{D}) - S_0)$, we relax the $S(\mathbf{D})$ again and get the following problem

$$\min_{\mathcal{G}, \mathbf{D}} \; w_2(\hat{S}(\mathbf{D}) - S_0) + \rho(L(\mathcal{G}, \mathbf{D}) + \hat{S}(\mathbf{D})) + \beta \|\mathcal{G}\|^2. \tag{15}$$

Finally, the scalarization problem (12) is solved by Algorithm 1. The late stage optimization can be independently applied to a trained tensor-compressed model for further model size reductions.

4 Performance Optimization of CoMERA

While CoMERA can greatly reduce the number of optimization variables and memory cost, it does not automatically speed up training on GPU. GPUs are well optimized for large-size matrix-matrix computations which are the computational bottleneck in standard uncompressed neural network training. However, the low-rank and small-size tensor operations in CoMERA are not efficiently supported by GPU. This section presents three methods to achieve real training speedup on GPU.

4.1 Performance Optimization of TTM Embedding Tables

Embedding tables are widely used in transformers and DLRM. An embedding table usually has a very unbalanced shape, i.e., the row size is much larger than the column size, making TTM compression more suitable than the TT format. In the following, we use an order-4 TTM embedding table to illustrate how to accelerate the lookup process.

We consider a generic embedding table $\mathbf{T} \in \mathbb{R}^{m \times n}$. Given a set of indices $\mathcal{I} = \{i_k\}_{k=1}^{b}$, the lookup operation selects the submatrix $\mathbf{T}[\mathcal{I}, :] \in \mathbb{R}^{b \times n}$. This operation is fast and inexpensive. However, the full embedding table itself is extremely memory-consuming. For instance, the largest embedding table in DLRM has over ten million rows. Suppose that $m = m_1 m_2 m_3 m_4, n = n_1 n_2 n_3 n_4$, then we reshape $\mathbf{T}$ into tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{m_1 \times n_1 \times \cdots \times m_4 \times n_4}$ and represent it in the TTM format

$$\boldsymbol{\mathcal{T}} = \boldsymbol{\mathcal{G}}_1 \times_{4,1} \boldsymbol{\mathcal{G}}_2 \times_{4,1} \boldsymbol{\mathcal{G}}_3 \times_{4,1} \boldsymbol{\mathcal{G}}_4. \tag{16}$$

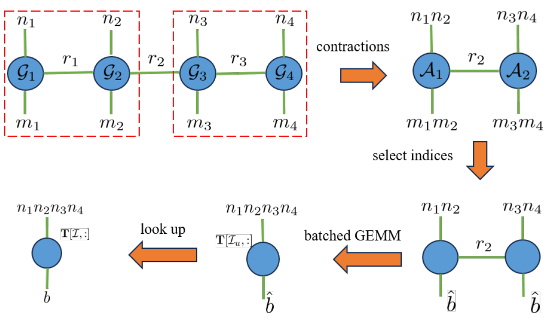


Figure 5: Optimized TTM embedding table lookup.

The compressed embedding table does not have the matrix $\mathbf{T}$ explicitly. To obtain $\mathbf{T}[\mathcal{I}, :]$ with a low memory cost, we convert each row index $i_k \in \mathcal{I}$ to a tensor index vector $(z_1^k, z_2^k, z_3^k, z_4^k)$ with $z_t^k \in [1, m_t]$ for each dimension t. We denote $\mathcal{Z}_t = \{z_t^k\}_{k=1}^{b}$, then $\mathbf{T}[\mathcal{I}, :]$ can be computed as

$$\begin{aligned}\mathbf{T}[\mathcal{I}, :] = \textbf{einsum}(&r_0 b n_1 r_1, r_1 b n_2 r_2, r_2 b n_3 r_3, r_3 b n_4 r_4 \Rightarrow b(n_1 n_2 n_3 n_4),\\ &[\boldsymbol{\mathcal{G}}_1[:, \mathcal{Z}_1, :, :], \boldsymbol{\mathcal{G}}_2[:, \mathcal{Z}_2, :, :], \boldsymbol{\mathcal{G}}_3[:, \mathcal{Z}_3, :, :], \boldsymbol{\mathcal{G}}_4[:, \mathcal{Z}_4, :, :]]).\end{aligned} \tag{17}$$

Each tensor $\boldsymbol{\mathcal{G}}_t[:, \mathcal{Z}_t, :, :]$ has size $r_{t-1} \times b \times n_i \times n_t$. Compared to the original tensor $\boldsymbol{\mathcal{G}}_t$, $\boldsymbol{\mathcal{G}}_t[:, \mathcal{Z}_t, :, :]$ stores many duplicated values especially when the size b of the index set $\mathcal{I}$ is large. Therefore, directly computing the tensor contractions (17) can cause much computing and memory overhead.

We optimize (17) by eliminating the redundant computation at two levels.

- **Row-index level.** The set $\mathcal{I}$ has many repeated row indices. Therefore, we construct the index set $\mathcal{I}_u = \{i_k\}_{k=1}^{\hat{b}}$ containing all unique indices in $\mathcal{I}$. We can easily obtain $\mathbf{T}[\mathcal{I}, :]$ from $\mathbf{T}[\mathcal{I}_u, :]$.
- **Tensor-index level.** The reduced row index set $\mathcal{I}_u$ leads to $\hat{b}$ associated tensor index vectors $(z_1^k, z_2^k, z_3^k, z_4^k)$, but at most $m_1 m_2$ pairs of (z_1^k, z_2^k) and $m_3 m_4$ pairs of (z_3^k, z_4^k) are unique. In practice, $\hat{b}$ is much larger than $m_1 m_2$ and $m_3 m_4$. Therefore, we can consider all unique pairs (z_1^k, z_2^k) and (z_3, z_4) and compute two intermediate tensors

$$\boldsymbol{\mathcal{A}}_1 = \textbf{einsum}(r_0 m_1 n_1 r_1, r_1 m_2 n_2 r_2 \Rightarrow (m_1 m_2)(n_1 n_2) r_2, [\boldsymbol{\mathcal{G}}_1, \boldsymbol{\mathcal{G}}_2]), \tag{18}$$

$$\boldsymbol{\mathcal{A}}_2 = \textbf{einsum}(r_2 m_3 n_3 r_3, r_3 m_4 n_4 r_4 \Rightarrow r_2 (m_3 m_4)(n_3 n_4), [\boldsymbol{\mathcal{G}}_3, \boldsymbol{\mathcal{G}}_4]). \tag{19}$$

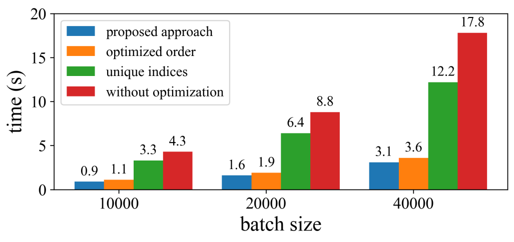
For each $i_k \in \mathcal{I}_u$, let (j_1^k, j_2^k) be the coordinate of i_k for size $(m_1 m_2, m_3 m_4)$. We denote $\mathcal{J}_1 = \{j_1^k\}_{k=1}^{\hat{b}}$ and $\mathcal{J}_2 = \{j_2^k\}_{k=1}^{\hat{b}}$, then compute the unique rows of $\mathbf{T}$ as

$$\mathbf{T}[\mathcal{I}_u, :] = \textbf{einsum}(\hat{b}(n_1 n_2) r_2, r_2 \hat{b}(n_3 n_4) \Rightarrow \hat{b}(n_1 n_2 n_3 n_4), [\boldsymbol{\mathcal{A}}_1[\mathcal{J}_1, :, :], \boldsymbol{\mathcal{A}}_1[:, \mathcal{J}_2, :]]). \tag{20}$$

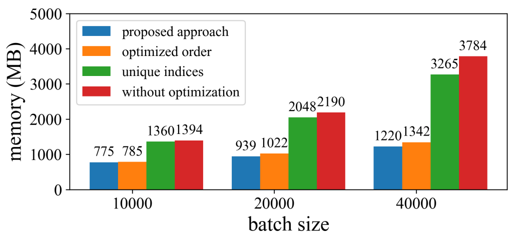
Figure 5 summarizes the whole process of TTM embedding table look up.

This approach can be easily applied to higher-order embedding tables. The general method is to first group some small tensor cores to obtain intermediate tensors and then utilize the intermediate tensors to compute unique row vectors. For the TTM embedding table with tensor cores $\boldsymbol{\mathcal{G}}_1, \ldots, \boldsymbol{\mathcal{G}}_6$, we may first compute $\boldsymbol{\mathcal{G}}_1 \times_{4,1} \boldsymbol{\mathcal{G}}_2, \boldsymbol{\mathcal{G}}_3 \times_{4,1} \boldsymbol{\mathcal{G}}_4, \boldsymbol{\mathcal{G}}_5 \times_{4,1} \boldsymbol{\mathcal{G}}_6$ or $\boldsymbol{\mathcal{G}}_1 \times_{4,1} \boldsymbol{\mathcal{G}}_2 \times_{4,1} \boldsymbol{\mathcal{G}}_3, \boldsymbol{\mathcal{G}}_4 \times_{4,1} \boldsymbol{\mathcal{G}}_5 \times_{4,1} \boldsymbol{\mathcal{G}}_6$. The choice depends on the shapes of the tensor cores.

Performance. We demonstrate the optimized TTM embedding tables on a single RTX 3090 GPU. We consider an embedding table of TTM shape $[[80, 50, 54, 50], [4, 4, 6, 8]]$ and rank 32, which is extracted from a practical DLRM model. We test the execution time for 100 lookups under various settings. As shown in Figure 6, our proposed method achieves about $4 - 5\times$ speed-up and $2 - 3\times$ memory saving than the standard TTM embedding without any optimization.



(a) Speed-up



(b) Memory reduction

Figure 6: Performance of optimized TTM embedding table lookup. The labels *proposed approach, optimized order, unique indices, without optimization* represent the new method in 4.1, the method that only uses the unique order, the method that only uses the unique indices, and the method without optimization, respectively.

4.2 Contraction Path Optimization for TT-Vector Multiplications

Next, we optimize the forward- and back- propagation of linear layers in the TT format. We consider a standard linear layer $\mathbf{Y} = \mathbf{X}\mathbf{W}$, where $\mathbf{Y} \in \mathbb{R}^{b\times N_2}, \mathbf{W} \in \mathbb{R}^{N_1\times N_2}, \mathbf{X} \in \mathbb{R}^{b\times N_1}$. The weight matrix $\mathbf{W}$ is compressed into the Tensor-Train format: $\boldsymbol{\mathcal{W}} = [\![\boldsymbol{\mathcal{G}}_1, \dots, \boldsymbol{\mathcal{G}}_{2d}]\!] \in \mathbb{R}^{n_1\times\cdots\times n_{2d}}$,where $\boldsymbol{\mathcal{G}}_i \in \mathbb{R}^{r_{i-1}\times n_i\times r_i}$, $r_0 = r_{2d} = 1$, and $N_1 = n_1 \cdots n_d, N_2 = n_{d+1}\cdots n_{2d}$.

The forward-propagation can be written using the einsum notation as

$$\mathbf{Y} = \mathbf{X}\mathbf{W} = \textbf{einsum}(bn_1 \dots n_d, S_1, \dots, S_{2d} \Rightarrow bn_{d+1}\dots n_{2d}, [\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{G}}_1, \dots, \boldsymbol{\mathcal{G}}_{2d}]) \tag{21}$$

where $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{b\times n_1\times\cdots\times n_d}$ is the reshaping of $\mathbf{X}$ and S_i denotes $r_{i-1}n_ir_i$. Suppose that the gradient to $\mathbf{Y}$ is $\mathbf{g}_{\mathbf{Y}}$, then the gradients to $\boldsymbol{\mathcal{G}}_i$ and $\boldsymbol{\mathcal{X}}$ can be computed as follows:

$$\begin{aligned} \mathbf{g}_{\boldsymbol{\mathcal{G}}_i} = \textbf{einsum}\big(&bn_1\dots n_d, bn_{d+1}\dots n_{2d}, S_1,\dots,S_{i-1},S_{i+1},\dots,S_{2d} \Rightarrow S_i, \\ &[\boldsymbol{\mathcal{X}}, \mathbf{g}_{\boldsymbol{\mathcal{Y}}}, \boldsymbol{\mathcal{G}}_1,\dots,\boldsymbol{\mathcal{G}}_{i-1},\boldsymbol{\mathcal{G}}_{i+1},\dots,\boldsymbol{\mathcal{G}}_{2d}]\big), \end{aligned} \tag{22}$$

$$\mathbf{g}_{\boldsymbol{\mathcal{X}}} = \textbf{einsum}\big(bn_{d+1}\dots n_{2d}, S_1,\dots,S_{2d} \Rightarrow bn_1\dots n_d, [\mathbf{g}_{\boldsymbol{\mathcal{Y}}}, \boldsymbol{\mathcal{G}}_1,\dots,\boldsymbol{\mathcal{G}}_{2d}]\big). \tag{23}$$

In total, $2d + 2$ contraction sequences are needed for the TT-format forward- and back- propagation. To reduce the computational costs, it is critical to find an optimal or near-optimal contraction path.

Large batch case. Given a large batch size b, an empirically near-optimal contraction path is to first contract $\boldsymbol{\mathcal{G}}_1,\dots,\boldsymbol{\mathcal{G}}_d$ and $\boldsymbol{\mathcal{G}}_{d+1},\dots,\boldsymbol{\mathcal{G}}_{2d}$, and then contract the results with $\boldsymbol{\mathcal{X}}$ and $\mathbf{g}_{\boldsymbol{\mathcal{Y}}}$. For convenience, we denote

$$\boldsymbol{\mathcal{A}}_i := \boldsymbol{\mathcal{G}}_1 \times\cdots\times \boldsymbol{\mathcal{G}}_i, \qquad \boldsymbol{\mathcal{A}}_{-i} = \boldsymbol{\mathcal{G}}_{d-i+1}\times\cdots\times\boldsymbol{\mathcal{G}}_d \tag{24}$$

$$\boldsymbol{\mathcal{B}}_i := \boldsymbol{\mathcal{G}}_{d+1} \times\cdots\times \boldsymbol{\mathcal{G}}_{d+i}, \qquad \boldsymbol{\mathcal{B}}_{-i} = \boldsymbol{\mathcal{G}}_{2d-i+1}\times\cdots\times\boldsymbol{\mathcal{G}}_{2d} \tag{25}$$

which are all computed sequentially. In practical implementations, we only need to compute $\boldsymbol{\mathcal{A}}_d, \boldsymbol{\mathcal{A}}_{-d}, \boldsymbol{\mathcal{B}}_d, \boldsymbol{\mathcal{B}}_{-d}$ and store the intermediate results. The forward-propagation (21) is then computed in the following way

$$\boldsymbol{\mathcal{T}}_1 = \textbf{einsum}(bn_1\dots n_d, n_1\dots n_dr_d \Rightarrow br_d, [\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{A}}_d]) \tag{26}$$

$$\boldsymbol{\mathcal{Y}} = \textbf{einsum}(br_d, r_dn_{d+1}\dots n_{2d} \Rightarrow bn_1\dots n_d, [\boldsymbol{\mathcal{T}}_1, \boldsymbol{\mathcal{B}}_d]). \tag{27}$$

In backward propagation, the gradients are computed in the following way:

- The gradient $\mathbf{g}_{\boldsymbol{\mathcal{X}}}$ is computed as

$$\boldsymbol{\mathcal{U}}_1 = \textbf{einsum}(bn_{d+1}\dots n_{2d}, r_dn_{d+1}\dots n_{2d} \Rightarrow br_d, [\mathbf{g}_{\boldsymbol{\mathcal{Y}}}, \boldsymbol{\mathcal{B}}_d]) \tag{28}$$

$$\mathbf{g}_{\boldsymbol{\mathcal{X}}} = \textbf{einsum}(br_d, n_1\dots n_dr_d \Rightarrow bn_1\dots n_d, [\boldsymbol{\mathcal{U}}_1, \boldsymbol{\mathcal{A}}_d]). \tag{29}$$

  The tensors $\boldsymbol{\mathcal{T}}_1, \boldsymbol{\mathcal{U}}_1$ are reused to compute gradients $\mathbf{g}_{\boldsymbol{\mathcal{G}}_i}$.

- The gradients $\mathbf{g}_{\boldsymbol{\mathcal{G}}_i}$ for $i \geq d+1$ can be computed as

$$\boldsymbol{\mathcal{T}}_2 = \textbf{einsum}(br_d, bn_{d+1}\dots n_{2d} \Rightarrow r_dn_{d+1}\dots n_{2d}, [\boldsymbol{\mathcal{T}}_1, \mathbf{g}_{\boldsymbol{\mathcal{Y}}}]) \tag{30}$$

$$\begin{aligned} \mathbf{g}_{\boldsymbol{\mathcal{G}}_i} = \textbf{einsum}(&r_dn_{d+1}\dots n_{2d}, r_dn_{d+1}\dots n_{i-1}r_{i-1}, r_in_{i+1}\dots n_{2d} \Rightarrow \\ &r_{i-1}n_ir_i,\ [\boldsymbol{\mathcal{T}}_2, \boldsymbol{\mathcal{B}}_{i-1-d}, \boldsymbol{\mathcal{B}}_{-(2d-i)}]). \end{aligned} \tag{31}$$

Algorithm 2 Empirical path for tensor-compressed forward- and back- propagation

```
Forward Input: Tensor cores $\mathcal{G}_1, \ldots, \mathcal{G}_{2d}$ and input matrix $\mathbf{X}$.
Forward Output: Output matrix $\mathbf{Y}$, and intermediate results.
1: Reshape the matrix $\mathbf{X}$ to the tensor $\mathcal{X}$.
2: Compute $\mathcal{A}_d, \mathcal{A}_{-d}, \mathcal{B}_d, \mathcal{B}_{-d}$ in the sequential order and store intermediate results of
   $\{\mathcal{A}_i, \mathcal{A}_{-i}, \mathcal{B}_i, \mathcal{B}_{-i}\}_{i=1}^{d}$ in (24) and (25) for back-propagation.
3: Compute $\mathcal{T}_1$ as in (26) to store it for back-propagation.
4: Compute $\mathcal{Y}$ as in (27) and reshape it to the appropriate matrix $\mathbf{Y}$.
Backward Input: Inputs of Forward, stored results from Forward, and output gradient $\mathbf{g}_{\mathbf{Y}}$.
Backward Output: Gradients $\mathbf{g}_{\mathbf{X}}, \mathbf{g}_{\mathcal{G}_1}, \ldots, \mathbf{g}_{\mathcal{G}_{2d}}$.
1: Reshape the gradient $\mathbf{g}_{\mathbf{Y}}$ to the tensor $\mathbf{g}_{\mathcal{Y}}$.
2: Compute $\mathcal{U}_1$ and $\mathbf{g}_{\mathcal{X}}$ as in (28), (29) and store $\mathcal{U}_1$ for future use.
3: Compute $\mathbf{g}_{\mathcal{G}_i}$ for $i \geq d+1$ as in (30), (31) using stored tensors.
4: Compute $\mathbf{g}_{\mathcal{G}_i}$ for $i \leq d$ as in (26), (33) using stored tensors.
```

- Similarly, the gradients $\mathbf{g}_{\mathcal{G}_i}$ for $i \leq d$ can be computed as

$$\mathcal{U}_2 = \mathbf{einsum}(br_d, bn_1 \ldots n_d \Rightarrow r_d n_1 \ldots n_d, [\mathcal{U}_1, \mathcal{X}]) \quad (32)$$

$$\mathbf{g}_{\mathcal{G}_i} = \mathbf{einsum}(r_d n_1 \ldots n_d, n_1 \ldots n_{i-1} r_{i-1}, r_i n_{i+1} \ldots n_d r_d \Rightarrow r_{i-1} n_i r_i,\ [\mathcal{U}_2, \mathcal{A}_i, \mathcal{A}_{-(d-i)}]). \quad (33)$$

The contraction paths of forward- and back- propagation are summarized in Algorithm 2.

Small batch case. We may search for a better path using a greedy search algorithm to minimize the total operations. In each iteration, we prioritize the pairs that output the smallest tensors. Such a choice can quickly eliminate large intermediate dimensions to reduce the total number of operations. When the batch size is large, the searched path is almost identical to the empirical path in Algorithm 2 which eliminates the batch size dimension b early. The searched path may differ from Algorithm 2 for small batch sizes, but their execution times on GPU are almost same. This is because the tensor contractions for smaller batch sizes have minor impact on the GPU running times. Consequently, despite certain tensor contractions in the empirical path being larger than those in the optimal path, the actual GPU execution times between them exhibit only negligible differences. Therefore, the empirical contraction path in Algorithm 2 is adopted for all batch sizes in CoMERA.

4.3 GPU Performance Optimization via Cuda Graph

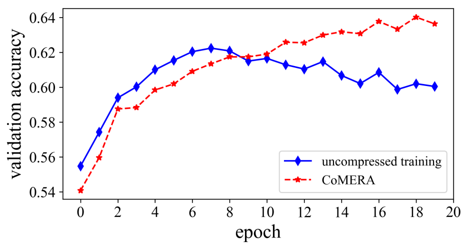
While tensor-compressed pre-training consumes theoretically much less computing FLOPS than standard uncompressed training, it can be slower on GPU in practice if not implemented carefully. Therefore, it is crucial to optimize the GPU performance to achieve real speedup.

Modern GPUs are highly optimized for large-size matrix multiplications which dominimates the cost of standard neural network training. However, CoMERA involves many small-size tensor contractions. These operations are not yet optimized on GPU and require many small-size GPU kernels, causing significant runtime overhead. In each training iteration, many GPU kernels are launched sequentially, causing launching overhead in the back end. Such overhead is negligible in standard neural network training. However, the back end overhead has a dramatic impact on the overall runtime of CoMERA, since the TT-vector multiplication is much less computing intensive. Furthermore, the small-size tensor contraction may also under-utilize the GPU resources.

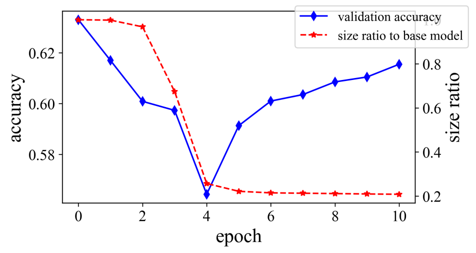
In this work, we employ Cuda Graph to reduce the launching overhead and improve the GPU utilization. Cuda Graph records all kernels required for the training and builds a computational graph with various computing kernels. During the pre-training, we launch and execute the whole graph rather than lunching a large number of kernels sequentially. CoMERA uses 4 to 6$\times$ more kernels than standard training. As a result, using Cuda Graph can eliminate lots of back-end overhead and lead to significant training speedup. We remark that this is just an initial step of GPU optimization. We expect that a more dedicated and comprehensive GPU optimization and algorithm/hardware co-design can achieve more significant training speedup.

Table 1: Tensorized setting for the Transformer model in CoMERA.

	format	linear shape	tensor shape	rank
embedding	TTM	(30527,768)	(64,80,80,60)	30
attention	TT	(768,768)	(12,8,8,8,8,12)	30
feed-forward	TT	(768,3072)	(12,8,8,12,16,16)	30



(a) Behavior of early-stage CoMERA training on the MNLI dataset.



(b) Model size and accuracy of late-stage CoMERA with a target size ratio 0.2 on MNLI.

Figure 7: Training performance of CoMERA on the the six-encoder transformer (batch size=128).

5 Results

In this section, we test the performance of CoMERA on a transformer and a DLRM benchmark. Our experiments are run on a Nvidia RTX 3090 GPU with 24GB RAM.

5.1 A Medium-Size Transformer with Six Encoders

We first consider a six-encoder Transformer model. The embedding tables and all linear layers in encoders are represented as tensor cores in the training process, as shown in Table 1. We train this model on the MNLI dataset with the maximum sequence length 128 and compare the accuracy, resulting model size, and training time of CoMERA with the standard uncompressed training.

CoMERA Accuracy and Compression Performance. Table 2 summarizes the training results. The **early-stage training** of CoMERA achieves $74\times$ compression ratio on all tensorized layers, and the validation accuracy even is higher than the uncompressed training. Figure 7a shows the validation accuracy of CoMERA. In the **late stage** of CoMERA, we set different target compression ratios for more aggressive rank pruning. The target compression ratios are for the tensorized layers rather than for the whole model. The late-stage training can reach the desired compression ratio with very little accuracy drop. The smallest model has an compression ratio of $80\times$ for the whole model due to a $361\times$ compression on the tensorized layers, and the obtained accuracy is only slightly worse than the uncompressed training. Fig. 7b shows the change of model size and validation accuracy in the late-stage training of CoMERA. When the model size does not reach the target size ratio 0.2, the model size keeps decreasing, and the validation accuracy drops a little bit. After the model size approaches the target ratio 0.2, CoMERA starts to recover the lost performance.

Architecture Search Capability of CoMERA. A fundamental challenge in the pre-training stage is architecture search: shall we keep certain blocks or layers of a model? Interestingly, CoMERA has some capability of automatic architecture search. Specifically, the ranks of some linear layers become zero in the pre-training of CoMERA, and hence the whole layer can be removed from the model. For the target compression ratio 0.2, the whole second last encoder and some linear layers in other encoders are completely removed after late-stage rank-adaptive training. The change of ranks of layers in the fifth encoder block is shown in Table 3.

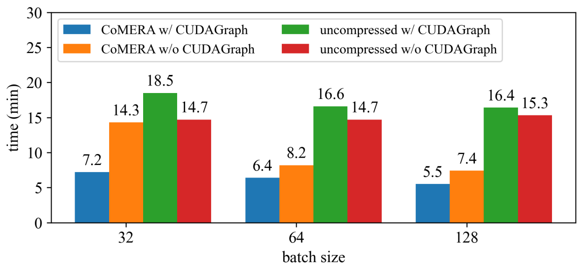


Figure 8: Training time per epoch for the six-encoder transformer model on the MNLI dataset.

Training Time. As shown in Figure 8, CoMERA with CudaGraph achieves around $3\times$ speed-up than uncompressed training for various batch sizes. CoMERA without CudaGraph can take much

Table 2: Result of Transformer on MNLI of batch size 128.

	validation	total size (MB)	compressed size (MB)
uncompressed training	62.2%	256 (1×)	253 (1×)
CoMERA (early stage)	63.3%	5.9 (43×)	3.4 (74×)
CoMERA (late stage), target ratio: 0.8	62.2%	4.9 (52×)	2.4 (105×)
CoMERA (late stage), target ratio: 0.5	62.1%	3.9 (65×)	1.4 (181×)
CoMERA (late stage), target ratio: 0.2	61.5%	3.2 (80×)	0.7 (361×)

Table 3: The change of ranks of layers in the fifth encoder block.

	before training	early-stage rank	late-stage rank
Q-layer in attention	$(12,30,30,30,12)$	$(12,30,30,30,12)$	$(0,0,0,0,0)$
K-layer in attention	$(12,30,30,30,12)$	$(12,30,30,30,12)$	$(0,0,0,0,0)$
V-layer in attention	$(12,30,30,30,12)$	$(12,30,30,30,12)$	$(9,11,11,7,9)$
FC-layer in attention	$(12,30,30,30,12)$	$(12,30,29,30,12)$	$(9,8,10,8,8)$
#1 linear-layer in Feed-Forward	$(12,30,30,30,16)$	$(0,0,0,0,0)$	$(0,0,0,0,0)$
#2 linear-layer in Feed-Forward	$(16,30,30,30,12)$	$(0,0,0,0,0)$	$(0,0,0,0,0)$

longer time in small batch-size setting due to the launching overhead of too many small kernels. The uncompressed training with CudaGraph takes longer time than the one without CudaGraph. This is because CudaGraph requires all batches to have the same sequence length, and the consequent computing overhead is more than the time reduction of CudaGraph. In contrary, CoMERA has much less computing FLOPS due to the low-rank tensor contractions in its forward and backward propagation, and the computing part accounts for a much smaller portion of the overall runtime.

5.2 A DLRM Model with 4-GB Model Size

We further test CoMERA with the DLRM model [37] released by Meta on the Criteo Ad Kaggle dataset. This model has 26 embedding tables to process categorical features. We compress the ten largest embedding tables into the TTM format as described in Section 4.1. All fully connected layers with sizes > 128 are compressed into the TT format. The model is trained for two epochs.

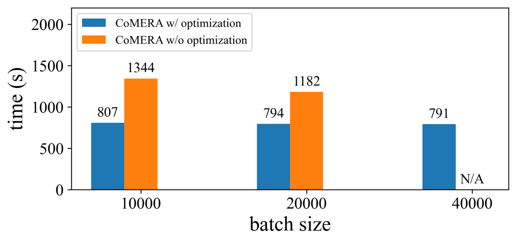
Effect of Optimized TTM Embedding. The training time per epoch and peak memory consumption are shown in Figure 9. Our optimized TTM lookup speeds up the whole training process by around $2\times$ and remarkably reduces the memory cost by $4-5\times$, especially for large batch training.

Overall Performance of CoMERA. Table 4 shows the testing accuracy, testing loss (measured as normalized CE), memory costs, and model sizes of CoMERA and uncompressed training. CoMERA achieves the similar accuracy as the uncompressed training. Furthermore, CoMERA compresses the whole DLRM model by $99\times$ and saves $7\times$ peak memory cost (with consideration of the data and backend overhead in a large-batch setting) in the training process. The reduction of model size and memory cost mainly comes from the compact TTM tensor representation of large embedding tables.

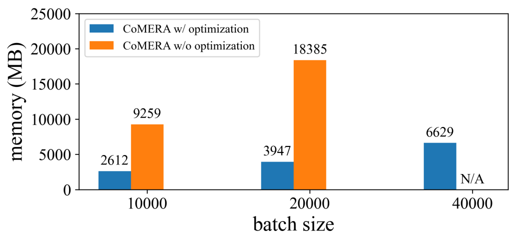
5.3 Comparison with GaLore and LTE

We compare our method with two recent low-rank compressed pre-training frameworks: GaLore [53] and LTE [23]. GaLore [53] reduces the memory cost by compressing the gradient of each weight matrix with singular-value decomposition (SVD), and LTE represents the model weights as the sum of some parallel low-rank matrix factorizations. We evaluate the memory costs and training times per epoch of various pre-training methods on the six-encoder transformer model under different batch sizes. We do not compare the total training time because the training epochs of various methods are highly case dependent. For instance, LTE [23] fails to converge for this benchmark.

Training Time Per Epoch. We use rank 128 for the low-rank gradients in GoLore, and rank 32 and head number 16 for the low-rank adapters in LTE. For fair comparison, all methods are executed with Cuda graph to reduce the overhead of lunching CUDA kernels. The runtimes per training epochs are reported in Figure 1(a). For the LTE, we only report the results for batch sizes $32, 64, 128$ since it requires the batch size to be a multiple of the head number. Overall, our CoMERA is about $2\times$ faster than GaLore and $3\times$ faster than LTE for all batch sizes, because the forward and backward propagation using low-rank tensor-network contractions dramatically reduce the computing FLOPS.



(a) Training time per epoch.



(b) Memory cost.

Figure 9: Performance of optimized CoMERA on training DLRM.

Table 4: Training results on the DLRM model with a batch size $10,000$.

	accuracy	normalized CE	model size (GB)	peak memory (GB)
uncompressed	78.68%	0.793	4.081	18.275
CoMERA	78.76%	0.792	0.041 (**99**$\times$)	2.612 (**7**$\times$)

Memory Cost. Figure 1 (b) shows the memory cost of all three pre-training methods. In the single-batch setting as used in [53], our CoMERA method is $9\times$ more memory-efficient than Galore on the tested case (with consideration of data and back-end cost). As the batch size increases, the memory overhead caused by data and activation functions becomes more significant, leading to less memory reduction ratios. However, our proposed CoMERA still uses the least memory.

6 Discussion: Mixed-Precision CoMERA

Modern GPUs offer low-precision computation to speed up the training and inference. It is natural to combine low-rank tensor compression and quantization to achieve the best training efficiency. However, CoMERA involves many small-size low-rank tensor contractions, and a naive low-precision implementation may even slow down the training due to the overhead caused by precision conversions.

To resolve the above issue, we implement mixed-precision computation in CoMERA based on one simple observation: large-size contractions enjoy much more benefits of low-precision computation than small-size ones. This is because the overhead caused by precision conversions can dominate the runtime in small-size contractions. In large-batch tensor-compressed training, small- and large-size tensor contractions can be distinguished by whether the batch size dimension b is involved. In general, a contraction with the batch b is regarded as large ~~and~~ is computed in a low precision. Otherwise, it is regarded small and is computed in full-precision. The actual mixed-precision algorithm depends on the contraction path used in the forward- and back- propagations of CoMERA.

Runtime. We evaluate the mixed-precision forward and backward propagations of CoMERA in a FP8 precision on the NVIDIA L4 GPU. We consider a single linear layer. The shapes $(1024, 1024)$ and $(1024, 4096)$ are converted to the TT shapes $(16, 8, 8, 8, 8, 16)$ and $(16, 8, 8, 16, 16, 16)$ respectively, and the ranks are both 32. The total execution time for 1000 forward and backward propagations are shown in Table 5. The FP8 tensor-compressed linear layer has about $3\times$ speed-up compared to the FP8 vanilla linear layer when the batch size and layer size are large. When the batch size is small, the FP8 vanilla linear layer is even faster. This is because the tensor-compressed linear layer consists of a few sequential computations that are not well supported by current GPU kernels. We expect to see a more significant acceleration after optimizing the GPU kernels.

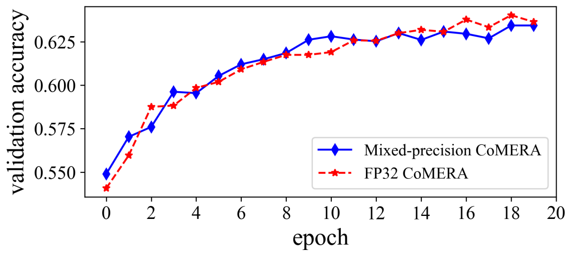


Figure 10: Convergence of mixed-precision CoMERA on the six-encoder transformer.

Convergence. We use the mixed-precision CoMERA to train the DLRM model and the six-encoder transformer. The result on DLRM is shown in Table 6. The convergence curve of the six-encoder

Table 5: Speed-up of mixed-precision computation on tensor-compressed linear layers.

shape (b,m,n)	tensor-vector		matrix-vector	
	FP8-mix	FP32	FP8	FP32
(10000,1024,1024)	1.95	1.63	1.02	2.82
(20000,1024,1024)	2.02	3.37	1.93	5.41
(40000,1024,1024)	2.55	6.82	4.27	10.93
(10000,1024,4096)	1.97	3.96	3.69	10.28
(20000,1024,4096)	2.96	8.32	7.26	20.93
(40000,1024,4096)	5.47	17.13	15.27	45.60

Table 6: Training results of mixed-precision CoMERA on DLRM (batch size=10,000).

	accuracy	normalized CE
FP32 CoMERA	78.76%	0.792
FP8/FP32 mixed-precision CoMERA	78.88%	0.793

transformer is shown in Figure 10. The experiments demonstrate that the accuracy of FP8 training is similar to FP32 training. However, we did not see much acceleration of using FP8 in the experiments. This is mainly because of ① the computation overhead of slow data type casting between FP32 and FP8; ② the sequential execution of small tensor contractions that are not well supported by current GPUs; ③ the relatively small sizes of linear layers in the tested models. We will investigate these problems in the future, and are optimistic to see significant acceleration on larger models.

7 Conclusions and Future work

In this work, we have presented the CoMERA framework to reduce the memory and computing time of pre-training AI models. We have investigated rank-adaptive training via a multi-objective optimization to meet specific model sizes while maintaining as much performance as possible. We have achieved real training speedup on GPU via three optimization: optimizing the tensorized embedding tables, optimizing the contraction path in tensorized forward and backward propagation, and optimizing the GPU latency via CudaGraph. The experiments on a six-encoder transformer model demonstrated that CoMERA can achieve $2-3\times$ speedup per training epoch. The model sizes of the transformer and a DLRM model have been reduced by $43\times$ to $99\times$ in the training process, leading to significant peak memory reduction (e.g., $7\times$ total reduction in large-batch training of DLRM). Our method has also outperformed the latest GaLore and LTE frameworks in both memory and runtime efficiency. We have also observed further speedup by combining CoMERA with mixed-precision computation.

Unlike large-size matrix operations, the small low-rank tensor-based operations used in CoMERA are not yet well-supported by existing GPUs and kernels. We believe that the performance of CoMERA can be furthr boosted significantly after a comprehensive GPU optimization. The existing optimizers, e.g. Adam, are well studied for uncompressed training. However, CoMERA has a very different optimization landscape due to the tensorized structure. Therefore, it is also worth studying the optimization algorithms specifically for CoMERA in the future.

References

[1] AI and compute. `https://openai.com/blog/ai-and-compute/`. OpenAI in 2019.

[2] ChatGPT and generative AI are booming, but the costs can be extraordinary. `https://www.cnbc.com/2023/03/13/chatgpt-and-generative-ai-are-booming-but-at-a-very-expensive-price.html`. Accessed: 2023-03-15.

[3] Update: ChatGPT runs 10K Nvidia training GPUs with potential for thousands more. `https://www.fierceelectronics.com/sensors/chatgpt-runs-10k-nvidia-training-gpus-potential-thousands-more`. Accessed: 2023-03-15.

[4] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[5] Giuseppe G Calvi, Ahmad Moniri, Mahmoud Mahfouz, Qibin Zhao, and Danilo P Mandic. Compression and interpretability of deep neural networks via tucker tensor layer. *arXiv:1903.06133*, 2019.

[6] Arslan Chaudhry, Naeemullah Khan, Puneet Dokania, and Philip Torr. Continual learning in low-rank orthogonal subspaces. *Advances in Neural Information Processing Systems*, 33:9900–9911, 2020.

[7] Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Re. Pixelated Butterfly: Simple and efficient sparse training for neural network models. *arXiv preprint arXiv:2112.00029*, 2021.

[8] Congliang Chen, Li Shen, Haozhi Huang, and Wei Liu. Quantized adam with error feedback. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 12(5):1–26, 2021.

[9] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[10] Kalyanmoy Deb, Karthik Sindhya, and Jussi Hakanen. Multi-objective optimization. In *Decision sciences*, pages 161–200. CRC Press, 2016.

[11] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. *arXiv preprint arXiv:2110.02861*, 2021.

[12] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

[13] Nyoman Gunantara. A review of multi-objective optimization: Methods and its applications. *Cogent Engineering*, 5(1):1502242, 2018.

[14] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[15] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[16] Cole Hawkins, Xing Liu, and Zheng Zhang. Towards compact neural networks via end-to-end training: A bayesian tensor approach with automatic rank determination. *SIAM Journal on Mathematics of Data Science*, 4(1):46–71, 2022.

[17] Cole Hawkins and Zheng Zhang. Bayesian tensorized neural networks with automatic rank selection. *Neurocomputing*, 453:172–180, 2021.

[18] Qinyao He, He Wen, Shuchang Zhou, Yuxin Wu, Cong Yao, Xinyu Zhou, and Yuheng Zou. Effective quantization methods for recurrent neural networks. *arXiv preprint arXiv:1611.10176*, 2016.

[19] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[20] Oleksii Hrinchuk, Valentin Khrulkov, Leyla Mirvakhabova, Elena Orlova, and Ivan Oseledets. Tensorized embedding layers for efficient model compression. *arXiv preprint arXiv:1901.10787*, 2019.

[21] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[22] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[23] Minyoung Huh, Brian Cheung, Jeremy Bernstein, Phillip Isola, and Pulkit Agrawal. Training neural networks from scratch with parallel low-rank adapters. *arXiv preprint arXiv:2402.16828*, 2024.

[24] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[25] Valentin Khrulkov, Oleksii Hrinchuk, Leyla Mirvakhabova, and Ivan Oseledets. Tensorized embedding layers for efficient model compression. *arXiv preprint arXiv:1901.10787*, 2019.

[26] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[27] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, Aug. 2009.

[28] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *NIPS*, pages 1742–1752, 2017.

[29] Joseph M Landsberg. Tensors: geometry and applications. *Representation theory*, 381(402):3, 2012.

[30] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[31] Yoonho Lee and Seungjin Choi. Gradient-based meta-learning with learned layerwise metric and subspace. In *International Conference on Machine Learning*, pages 2927–2936. PMLR, 2018.

[32] Bingrui Li, Jianfei Chen, and Jun Zhu. Memory efficient optimizers with 4-bit states. *Advances in Neural Information Processing Systems*, 36, 2024.

[33] Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. Relora: High-rank training through low-rank updates. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ NeurIPS 2023)*, 2023.

[34] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.

[35] Xindian Ma, Peng Zhang, Shuai Zhang, Nan Duan, Yuexian Hou, Ming Zhou, and Dawei Song. A tensorized transformer for language modeling. *Advances in neural information processing systems*, 32, 2019.

[36] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11264–11272, 2019.

[37] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[38] Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry P Vetrov. Structured Bayesian pruning via log-normal multiplicative noise. In *NIPS*, pages 6775–6784, 2017.

[39] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in neural information processing systems*, pages 442–450, 2015.

[40] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems 28*, pages 442–450, 2015.

[41] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[42] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[43] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.

[44] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[45] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi Viji Srinivasan, and Kailash Gopalakrishnan. Ultra-low precision 4-bit training of deep neural networks. *NIPS*, 33, 2020.

[46] Andros Tjandra, Sakriani Sakti, and Satoshi Nakamura. Compressing recurrent neural network with tensor train. *CoRR*, abs/1705.08052, 2017.

[47] Andros Tjandra, Sakriani Sakti, and Satoshi Nakamura. Compressing recurrent neural network with tensor train. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4451–4458, 2017.

[48] Xiao Wang, Tianze Chen, Qiming Ge, Han Xia, Rong Bao, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. Orthogonal subspace learning for language model continual learning. *arXiv preprint arXiv:2310.14152*, 2023.

[49] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016.

[50] Yifan Yang, Jiajun Zhou, Ngai Wong, and Zheng Zhang. LoRETTA: Low-rank economic tensor-train adaptation for ultra-low-parameter fine-tuning of large language models. *arXiv preprint arXiv:2402.11417*, 2024.

[51] Yinchong Yang, Denis Krompass, and Volker Tresp. Tensor-train recurrent neural networks for video classification. In *Proc. the 34th International Conference on Machine Learning*, volume 70, pages 3891–3900, Sydney, Australia, 06–11 Aug 2017.

[52] Zi Yang, Samridhi Choudhary, Siegfried Kunzmann, and Zheng Zhang. Quantization-aware and Tensor-compressed Training of Transformers for Natural Language Understanding. In *Proc. INTERSPEECH 2023*, pages 3292–3296, 2023.

[53] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient LLM training by gradient low-rank projection, 2024.

[54] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor ring decomposition. *arXiv preprint arXiv:1606.05535*, 2016.

[55] Yequan Zhao, Xinling Yu, Zhixiong Chen, Ziyue Liu, Sijia Liu, and Zheng Zhang. Tensor-compressed back-propagation-free training for (physics-informed) neural networks. *arXiv preprint arXiv:2308.09858*, 2023.