eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Master Thesis

Máster Universitario en Ingeniería Computacional y Sistemas Inteligentes

Tensor Decompositions for Neural Networks Compression

Unai Sainz de la Maza Gamboa

Advisors
Jose A. Pascual Saiz

July 10, 2023

Abstract

As the demand for deploying machine learning models on resource-constrained devices grows, neural network compression has become an important area of research. Tensor decomposition is a promising technique for compressing neural networks, as it enables the representation of the network weights in a lower-dimensional format, while maintaining their accuracy and performance. In this work, we explore the application of tensor decomposition techniques, including Canonical Polyadic decomposition, Tucker decomposition, and Tensor Train decomposition, for neural network compression. We provide an exhaustive overview of the various tensor decomposition methods and compare their performance in terms of compression rates and accuracy. We implement and evaluate the different compression methods on the benchmark dataset CIFAR-10, using popular models such as ResNet and VGG. Our results show that tensor decomposition can significantly reduce the number of parameters of neural networks, while reducing minimally their accuracy. Finally, we discuss the challenges and opportunities of using tensor decomposition for neural network compression and highlight some open research questions in this field.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In recent years, Deep Neural Networks (DNNs) have become a powerful and valuable tool in many industrial and commercial applications. In this manner, DNNs have been successfully used in various domains such as computer vision, audio processing, speech recognition, etc. In computer vision, Convolutional Neural Networks (CNNs) have achieved the state-of-the-art on several tasks like image classification [2], object detection [3], semantic segmentation [4], and human pose estimation [5].

One of the key feature behind these methods is over-parametrization, for which there is evidence that helps to find a local minima [6]. However, over-parametrization leads to issues like redundancy, or making generalization harder, because it excessively increases the number of parameters. Also, increasing the number of parameters have also and impact in terms of storage and computational requirements. Because of that, the deployment of these overparametrized models on devices with limited computational resources, such as mobile devices and edge computing, is sometimes unfeasible.

In this way, neural network compression is the process of reducing the size and complexity of a neural network, while maintaining its accuracy and improving its performance. Compressed neural networks have a smaller memory footprint, require less computational resources for training and inference, and can be deployed more efficiently on low-power devices such as mobile phones or embedded systems. Several approaches have been proposed to reduce the redundancy and improve the efficiency of the models such as quantization, network pruning, weight sharing, knowledge distillation and low-rank factorization. For the interested reader, in [7] a general survey about neural network compression methods is provided.

Tensor methods are mathematical techniques for representing high-dimensional data using lower-dimensional structures which can significantly reduce the storage and computational requirements. Tensors are multidimensional arrays and a core mathematical object in multilinear algebra that arise naturally in a wide range of applications such as quantum physics simulations [8], signal processing [9], numerical linear algebra [10], neuroscience [11], graph analysis [10], data mining [12], and more. In this way, tensor methods have been used in recent years for compressing neural networks, taking advantage of the inherent multidimensional structure of neural networks to achieve compression by decomposing

the weight tensors of the neural network into low-dimensional core tensor and a set of matrices, or into a collection of lower dimensional tensors. Indeed, tensor decompositions can be applied to the weights of neural network layers to compress them, and in some cases, speed them up [13].

The Tucker decomposition (or Higher-order singular value decomposition) [14] is one of the earliest methods used for compressing neural networks, where we decompose the high-dimensional tensor into a core tensor multiplied by a matrix along each mode. However, other popular methods such as Tensor Train (TT) decomposition [15], where we decompose a high-dimensional tensor into a series of lower-dimensional tensors along each mode, or Canonical Polyadic (CP) decomposition [16], where we factorize a tensor into a sum of outer products of vectors are also used to reduce the number of parameters in the neural networks while preserving its performance. Other tensor decomposition algorithms have been used in the recent literature for neural network compression, such as Tensor Ring (TR) [17], Hierarchical Tucker (HT) [18], and Block-Term decomposition [19].

In this work, we explore the tensorization of different models based on CNNs such as ResNet [20] and VGG [21]. We have focused our work on tensorizing the convolutional layers of these models using tensor decompositions, including Tucker, Canonical Polyadic, and Tensor-Train decompositions. We also explore the strategy used for the tensorization, following two main strategies, the first, where we tensorize pre-trained network layers weights and then we make a fine-tuning to recover the performance that may have been lost due to tensorization, and the second, where we tensorize the network layers before we train them from scratch. Finally, we compare different approaches to select the rank of the tensorization, and we compare both tensorization strategies in terms of compression rate and accuracy trade-off.

The rest of the work is organized as follows. In Chapter 2, we revise some of the most popular methods proposed for neural network compression, and we review the recent developments in tensor methods applied to neural network compression. In Chapter 3, we introduce our work proposal and the aims of the project. In Chapter 4, we introduce the theoretical background necessary to follow the work. In Chapter 5, we provide a general introduction to tensor methods, and we facilitate a comprehensive overview of the most important tensor decomposition algorithms. In Chapter 6, we explain how tensor decompositions are used for neural network compression, including important aspects such as the complexity analysis, or the rank selection. In Chapter 7, we experimentally measure the performance of the proposed method to support the theoretical framework. Finally, in Chapter 8 and Chapter 9, we end the work by discussing some conclusions and highlighting potential opportunities for this research line.

CHAPTER 2

Related work

In this chapter, we provide a brief review of the different techniques used for compressing neural networks.

Neural network compression has been an active research area in recent years due to the growing demand for deploying machine learning models on resource-constrained devices. Various techniques have been proposed to compress neural networks, including quantization, pruning, knowledge distillation, and tensor decomposition.

Quantization is concerned with quantizing the weights and/or the features of a neural network [22], in this straightforward way, we can speed up neural network computations and minimize memory requirements. Network pruning is an approach to reduce a heavy network to obtain a light-weight form by removing redundancy in the heavy network, different types of pruning have been proposed, e.g., structured pruning [23], unstructured pruning [24], and many others. Knowledge distillation [25], is the process of transferring knowledge from a large model to a smaller one, the key idea behind this approach is that while large models have higher knowledge capacity than small models, this capacity might not be fully utilized.

Tensor decomposition [10], in particular, has received increasing attention as an effective method for compressing neural networks. Tensor decomposition techniques aim to represent the weights of a neural network in a lower-dimensional format, by decomposing the weight tensor into a set of factor matrices along each mode. This approach can significantly reduce the storage and computational requirements of the network, which is crucial for deployment on edge devices and embedded systems.

Several tensor decomposition techniques have been proposed in the literature, including CP decomposition [16], Tucker decomposition [14], HT decomposition [18], and TT decomposition [15]. Where each of these techniques has its strengths and weaknesses, and the choice of the decomposition method depends on the specific use case. CP decomposition represents a tensor as a sum of rank-one tensors and is particularly useful for compressing convolutional layers. Tucker decomposition decomposes a tensor into a small core tensor and factor matrices and is suitable for compressing fully connected layers. Hierarchical Tucker decomposition is an extension of Tucker decomposition that allows for a more flexible decomposition of tensors with high modes. Tensor-Train decomposition

decomposes a tensor into a set of TT cores and is particularly useful for compressing very high-dimensional tensors.

The use of tensor decompositions for neural network compression was first investigated in [26], where CP decomposition was proposed to compress a 4-dimensional convolution kernel. Additionally, low-rank matrix factorization was used to speed up the inference time of the CNNs. In a similar vein, [13] treated weight matrices as multi-dimensional tensors and applied the TT decomposition algorithm to compress feed-forward layers. The authors achieved high compression ratios without significant loss of accuracy, and provided theoretical support for the proposed method. Following this approach, in [27], the authors propose a TT decomposition based tensorization, both for the feed-forward and the convolutional layers of different CNN models. More recently, this approach of using TT decomposition based tensorization have been studied from a physics perspective in [28], and extended to compress 3D convolutional neural networks for video classification in [29].

Inspired in [26], other works have tried to compress CNNs using CP decomposition. For example, in [30], the authors improve the CP decomposition algorithm and they propose a Tensor Convolutional Neuro-Network (TCNN) for anomaly detection. In [31], they compress 3DCNNs using CP decomposition with an application to spatio-temporal facial emotion analysis. In [32], after tensorizing the convolutional layers via CP decompostion, they analyze the value of the decomposed kernels to guide feature selection. Also, in works such as [33], the authors use Tucker decomposition with nonlinear response to compress the convolutional layers, [34] propose a hybrid tensor decomposition scheme, where they combine TT decomposition for feed-forward layers and hierarchical tucker for the convolutional layers. In [35] the authors propose T-Net, a fully parametrized CNN with a single high-order low-rank tensor using TT and Tucker decompositions, this allows them to regularize the whole network thanks to the low-rank structure imposed on the weight tensor and drastically reduce the number of parameters.

Recent works have develop more sophisticated algorithms, for example, in [36], the authors dynamically and adaptively adjust the model size and decomposition structure without retraining, i.e., using a data-driven adaptive tensor decomposition approach. In [37], the authors propose an iterative approach, which alternates low-rank factorization, rank selection and fine-tuning. Also, novel tensor decompositions have been proposed, for example, in [34], the authors use sequences of Kronecker products to generalize widely used methods CP, TT, Tucker, etc. In [38], they propose a novel global compression framework that automatically analyzes each layer to identify the optimal per-layer compression ratio, while simultaneously achieving the desired overall compression. Other approaches have studied the combination of different neural network compression methods, e.g., [33] combines quantization and tensorization to compress deep neural networks. Additionally, in [39], the authors have introduced a novel weight initialization paradigm, which generalizes Xavier and Kaiming methods, making it widely applicable to any tensor-based CNNs.

Tensor decompositions have been also used for compressing other types of neural networks such as Recurrent Neural Networks (RNNs) [40], Long Short-Term Memory (LSTM) [41], and Transformers architecture [42, 43, 44, 45]. For example, in [44], they propose a hardware-aware tensor decomposition for Transformers architecture, i.e., given the exponential space of possible decompositions, they automate the choice of tensorization shape and decomposition rank with hardware-aware co-optimization. Another works like

[46], also have tried to compress transformers architecture, but in this case, the authors propose a novel compressed self-attention mechanism called multi-linear attention using block-term tensor decomposition.

The use of tensor methods in deep learning has not been limited to the use of tensor decompositions for neural network compression. Tensor methods have been successfully used for a wide range of tasks such as deep neural networks interpretability [47, 48], multi-dimensional data analysis [49], information fusion [50], and many others. In [51], they introduce a novel Tensor Contraction Layer (TCL) that reduces the dimensionality while maintaining the multi-linear structure of the data, being parsimonious in terms of number of parameters. In this manner, [52] generalized the Linear Regression concept to higher order, and [53] introduced Tensor Dropout, a randomization in the tensor rank for robustness. However, all these applications are out of the scope of this work, but for the interested reader, a recent comprehensive survey can found in [54].

CHAPTER 3

The aims of the project

The project's primary goal of this work is to investigate the use of tensor decompositions for neural network compression. By evaluating the effectiveness of various tensor decomposition techniques for compressing CNNs, we aim to identify the most effective methods for achieving high compression rates while maintaining accuracy and performance. To accomplish this task, we need to investigate and understand the concept of neural network compression. Furthermore, we aim to provide a comprehensive understanding of neural network compression techniques and their importance in reducing model size and computational requirements.

Secondly, we need to explore tensor decompositions for neural network compression, studying different tensor decomposition methods and their applicability to compressing neural networks. This includes understanding the mathematical foundations, advantages, and limitations of various tensor decomposition techniques.

The third step is to evaluate the impact of tensor decompositions on model compression. That is, analyze the trade-off between compression ratios and model performance, i.e., the impact of compression of factors such as model size reduction, computational efficiency, memory footprint, and inference speed. Additionally, we will investigate the usage of different rank selection strategies that have been proposed in the literature.

Finally, to evaluate the potential of using tensor decompositions for neural network compression, we will need to include a comparative analysis between tensor decomposition-based compressed models and raw models, where the tensorization of both pretrained and trained from scratch models is considered. This analysis should take into account metrics such as accuracy, generalization, and training convergence to assess the trade-offs between compression and performance.

All in all, the main objectives of the work can be summarised as follows:

- Investigate and understand the concept of neural network compression, and methods proposed.
- Study the mathematical foundations required to understand tensor methods, and explore tensor decompositions such as CP, Tucker and TT decompositions.

- Study the applicability of tensor decompositions to neural network compression and analyze their impact in terms of compression ratios, computational efficiency, memory footprint, etc.
- Provide a comparative analysis between tensor decomposition-based compressed models and raw models performance measured by metrics such as accuracy, generalization, on standard datasets like CIFAR-10.
- Identify the open research questions and challenges in the field of neural network compression using tensor decompositions.

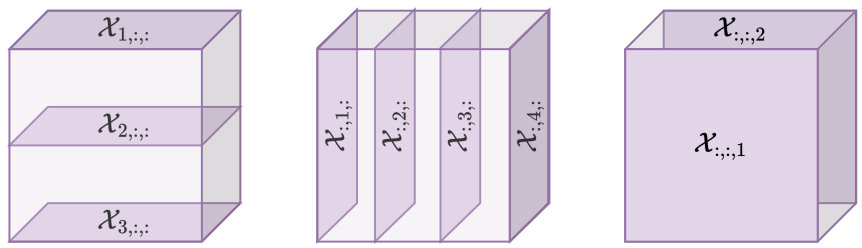
CHAPTER 4

Notation and preliminaries

To make the work self-contained and give the reader an understanding of the mathematical tools employed in tensor methods, we first introduce the notation conventions used in this work and then review some of the fundamental concepts of linear and multilinear algebra.

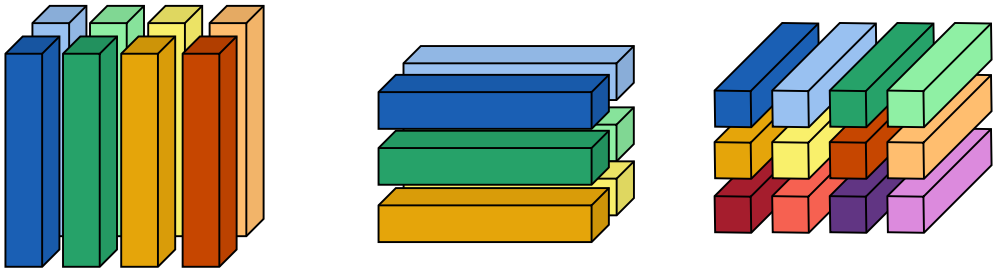
4.1 Notation

Tensors [55], also known as multi-way arrays, can be seen as higher-order extensions of vectors (1^{st}-order tensors), or matrices (2^{nd}-order tensors). In the same way as rows and columns in a matrix, an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$ has N modes (i.e., orders, ways, or indices) whose dimensions (i.e., lengths) are represented by $I_1, \ldots, I_N$. An element $(i_1, i_2, \ldots, i_N)$ of tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$ is accessed as: $\mathcal{X}_{i_1, i_2, \ldots, i_N}$ or $\mathcal{X}(i_1, i_2, \ldots, i_N)$. Also, given a set of N matrices (or vectors) that correspond to each mode of $\mathcal{X}$, the n^{th} matrix (or vector) is denoted as $\mathbf{U}^{(n)}$ (or $\mathbf{u}^{(n)}$). Furthermore, *fibers* are the higher-order generalization of the concept of rows and columns of matrices to tensors, obtained by fixing all indices but one, where to represent all the elements of a mode, we use a colon. For instance, if $\mathcal{X}$ is a third order tensor, then its mode-1 (column) fibers can be denoted as $\mathcal{X}_{:,j,k}$ (see Fig. 4.2). *Slices* are two-dimensional sections of a tensor, defined by fixing all but two indices. As shown in Fig. 4.1, the horizontal, lateral and frontal slices of a third-order tensor $\mathcal{X}$ are denoted by $\mathcal{X}_{i_1::}$, $\mathcal{X}_{:i_2:}$, and $\mathcal{X}_{::i_3}$, respectively.



(a) Horizontal slices: $\mathcal{X}_{i,:,:}$ (b) Lateral slices: $\mathcal{X}_{:,j,:}$ (c) Frontal slices: $\mathcal{X}_{:,:,k}$

Figure 4.1: Slices of a third order tensor of size $3 \times 4 \times 2$.



(a) mode-1 (column) fibers: $\mathcal{X}_{:,j,k}$ (b) mode-2 (row) fibers: $\mathcal{X}_{i,:,k}$ (c) mode-3 (tube) fibers: $\mathcal{X}_{i,j,:}$

Figure 4.2: Fibers of a third order tensor.

As shown in Table 4.1, lowercase letters denote scalars, e.g., n, boldface lowercase letters denote vectors, e.g., $\mathbf{v}$, boldface capital letters denote matrices, e.g., $\mathbf{M}$, and boldface Euler script letters denote tensors of order 3 or greater, e.g., $\mathcal{X}$.

Notation	Definition
n	scalar
$\mathbf{v}$	vector
$\mathbf{M}$	matrix
$\mathcal{X}$	tensor
A	dimensionality

Table 4.1: Tensor notations.

4.2 Mathematical background

4.2.1 Transforming tensors intro matrices or vectors

Definition 4.1 (Tensor unfolding). *Given a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$, its mode-$n$ unfolding is a matrix $\mathbf{X}_{[n]} \in \mathbb{R}^{I_N \times I_M}$, with $M = \prod_{\substack{k=1 \\ k \neq n}}^{N} I_k$, and is defined by the mapping from the tensor element $(i_1, i_2, \ldots, i_N)$ to matrix element (i_n, j), where $j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^{N} (i_k - 1) + \prod_{\substack{m=k+1 \\ m \neq n}}^{N} I_m$.*

Tensor unfolding, also known as *matricization* or *flattening*, is the process of reordering the fibers of a tensor as the columns of a matrix. See Fig. 4.3 for a illustrative example of a mode-1 unfolding of the tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$. Notice that the mode-$n$ unfolding introduced here is an special case of a more general matricization; see [56] for further details.

Definition 4.2 (Tensor vectorization). *Given a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$, we can transform it into a vector $vec(\mathcal{X})$ of size $I_1 \cdot I_2 \cdot \cdots \cdot I_N$, with a mapping from tensor element $(i_1, i_2, \ldots, i_N)$ to $vec(\mathcal{X})$ element j, where $j = 1 + \sum_{k=1}^{N} (i_k - 1) \times \prod_{m=k+1}^{N} I_m$.*

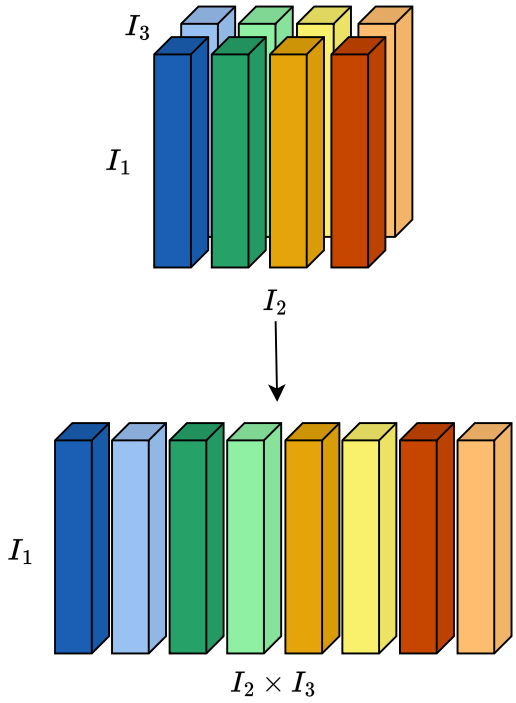


Figure 4.3: Mode-1 unfolding of a third order tensor.

4.2.2 Tensor and matrix products

Definition 4.3 (n-mode product). *Given a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$ and a matrix $\mathbf{M} \in \mathbb{R}^{R \times I_n}$, the n-mode (*matrix*) product of a tensor is denoted as $\mathcal{X} \times_n \mathbf{M}$, where the resulting tensor is of size $I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \times \cdots \times I_N$. The n-mode product of a tensor with a matrix can be seen as change of basis when a tensor defines a multilinear operator [10]. Also, the operation can also be defined using the tensor unfolding of $\mathcal{X}$, and the dot product as*

$$\mathcal{X} \times_n \mathbf{M} = \mathbf{M}\mathbf{X}_{[n]} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \times \cdots \times I_N}. \tag{4.1}$$

Similarly, the n-mode vector *product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$ with a vector $\mathbf{v} \in \mathbb{R}^{I_n}$ is denoted as $\mathcal{X} \bar{\times}_n \mathbf{v} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N}$, where the order of the resulting tensor is $N - 1$. This can be defined elementwise as*

$$(\mathcal{X} \bar{\times}_n \mathbf{v})_{i_1 \ldots i_{n-1} i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \ldots i_N} v_{i_n}. \tag{4.2}$$

The idea here is to compute the inner product of each mode-n fiber with vector v.

Definition 4.4 (Tensor contraction). *Let $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$ and $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_M}$ be the N- and M-ord3er tensors, respectively. The mode-$\binom{m}{n}$ contraction of $\mathcal{X}$ and $\mathcal{Y}$ with $I_n = I_m$ is defined by $\mathcal{Z} = \mathcal{X} \times_n^m \mathcal{Y}$ [10], where its entries are defined elementwise as*

$$\mathcal{Z}_{i_1,\ldots,i_{n-1},i_{n+1},\ldots,i_N,j_1,\ldots,j_{m-1},j_{m+1},\ldots,j_M} = \\ = \sum_{i_n=1}^{I_n} \mathcal{X}_{i_1,\ldots,i_{n-1},i_n,i_{n+1},\ldots,i_N} \mathcal{Y}_{j_1,\ldots,j_{m-1},i_n,j_{m+1},\ldots,j_M}. \tag{4.3}$$

Notice that if we consider two matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{J \times K}$, their matrix multiplication can be also described as a tensor contraction denoted by

$$\mathbf{A} \times_2^1 \mathbf{B}. \tag{4.4}$$

Definition 4.5 (Matrix Kronecker product). *Given two matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{K \times L}$, their Kronecker product is denoted as*

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1J}\mathbf{B} \\ a_{21}\mathbf{B} & \cdots & a_{2J}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{I1}\mathbf{B} & \cdots & a_{IJ}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{I \cdot K \times J \cdot L}. \tag{4.5}$$

Definition 4.6 (Khatri-Rao product). *Given two matrices $\mathbf{A} \in \mathbb{R}^{I \times K}$ and $\mathbf{B} \in \mathbb{R}^{J \times K}$, with the same number of columns, their Khatri-Rao (or matching Kronecker columnwise) product is denoted as*

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} \mathbf{A}_{:,1} \otimes \mathbf{B}_{:,1}, & \mathbf{A}_{:,2} \otimes \mathbf{B}_{:,2}, & \ldots, & \mathbf{A}_{:,K} \otimes \mathbf{B}_{:,K} \end{bmatrix} \in \mathbb{R}^{I \cdot J \times K}. \tag{4.6}$$

If $\mathbf{a}$ and $\mathbf{b}$ are vectors, then Khatri-Rao and Kronecker products are the same, i.e., $\mathbf{a} \otimes \mathbf{b} = \mathbf{a} \odot \mathbf{b}$.

Definition 4.7 (Hadamard product). *Given matrices $\mathbf{A}$ and $\mathbf{B}$, both of size $I \times J$, their Hadamard product is the elementwise product defined as*

$$\mathbf{A} * \mathbf{B} = \mathbf{A}_{i,j}\mathbf{B}_{i,j} = \begin{bmatrix} a_{11}b_{11} & \cdots & a_{1J}b_{1J} \\ a_{12}b_{21} & \cdots & a_{2J}b_{2J} \\ \vdots & \ddots & \vdots \\ a_{I1}b_{I1} & \cdots & a_{IJ}b_{IJ} \end{bmatrix} \in \mathbb{R}^{I \times J}. \tag{4.7}$$

Definition 4.8 (Outer product). *Given a set of N vectors $\{x^{(n)}\}_{n=1}^{N}$, their outer product is denoted by*

$$\mathcal{X} = x^{(1)} \circ x^{(2)} \circ \cdots \circ x^{(N)} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}, \tag{4.8}$$

which defines a rank-one N^{th}-order tensor.

Definition 4.9 (Inner product). *The inner product of two same-sized tensors $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, is the sum of products of their entries defined as*

$$\langle \mathcal{X}, \mathcal{Y} \rangle = \sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_n=1}^{I_N} x_{i_1 i_2 \ldots i_n} y_{i_1 i_2 \ldots i_n}, \tag{4.9}$$

where $\langle \mathcal{X}, \mathcal{X} \rangle = ||\mathcal{X}||^2$.

4.2.3 Matrix and tensor rank

Definition 4.10 (Matrix rank). *Given a matrix of real numbers $\mathbf{X} \in \mathbb{R}^{I \times J}$, the rank of $\mathbf{X}$, also denoted as $rank(\mathbf{X})$, is defined as*

- *The number of linearly independent columns of $\mathbf{X}$.*

- *The number of linearly independent rows of* $\mathbf{X}$.

This definition implies that $rank(\mathbf{X}) \leq min(I, J)$*, and if* $rank(\mathbf{X}) = min(I, J)$*,* $\mathbf{X}$ *is full-rank.*

Notice that the rank of a matrix can be defined equivalently in several ways [57], here we use one of the simplest definitions available.

Definition 4.11 (Tensor rank)**.** *The rank of a tensor* $\mathcal{X}$*, denoted by* $rank(\mathcal{X})$*, is defined as the smallest number of rank-one tensors that generates* $\mathcal{X}$ *as their sum [58].*

The definition of tensor rank is analogous to that of matrix rank, but the properties of matrix and tensor ranks differ significantly. One notable distinction is that the rank of a real-valued tensor may vary over $\mathbb{R}$ and $\mathbb{C}$, as shown in [10]. Another significant difference between matrix and tensor ranks is that, except in special cases, there is no straightforward algorithm for determining the rank of a given tensor, and in fact, the problem is NP-hard [59].

4.3 Neural networks

As this work aims to be self-contained, here we briefly recap fully-connected and convolutional layers, which are then compressed through tensorization and used to discuss DNNs.

4.3.1 Fully-connected layer

We start from a fully-connected layer, which is commonly defined as a linear layer $f(\cdot)$ followed by an activation function $\sigma(\cdot)$. Given an input matrix $\mathbf{X} = \{\mathbf{X_1}, \ldots, \mathbf{X_m}\} \in \mathbb{R}^{n^2 \times m}$, obtained by stacking the data points as columns, and the bias matrix $\mathbf{B} = \{\mathbf{b_1}, \ldots, \mathbf{b_m}\} \in \mathbb{R}^{c \times m}$, we can define the fully-connected layer as

$$\hat{\mathbf{Y}} = \sigma(f(\mathbf{X}; \mathbf{W}, \mathbf{B})) = \sigma(\mathbf{WX} + \mathbf{B}), \tag{4.10}$$

where $\hat{\mathbf{Y}} \in \mathbb{R}^{c \times m}$ is the output of the fully-connected layer, which maps each data point from the input space $\mathbb{R}^{n^2}$ to the output space, $\mathbb{R}^c$.

4.3.2 Fully-connected neural networks

If we stack several fully-connected layers introduced in Section 4.3.1, we can form the so called fully-connecte neural networks. This multilayered model consists of the input layer, the output layer, and the internal layers in between, commonly called the hidden units because their intermediate results do not show up in the final result. Fig. 4.4 shows an example of a fully-connected neural network with three layers (ignoring the input layer).

We denote the input layer as layer 0, the layer that follows it as layer 1, and so on. The weights and biases of each layer are denoted by $\mathbf{W}^j$ and $\mathbf{B}^j$, where j is the layer number. Each of the two hidden layers computes a linear mapping followed by a nonlinear activation function. We denote the result before the activation function is applied as $\mathbf{Z}^j$ and the layer's output as $\mathbf{A}^j$. Therefore, given a training dataset $\mathbf{X} \in \mathbb{R}^{n^2 \times m}$, we calculate the

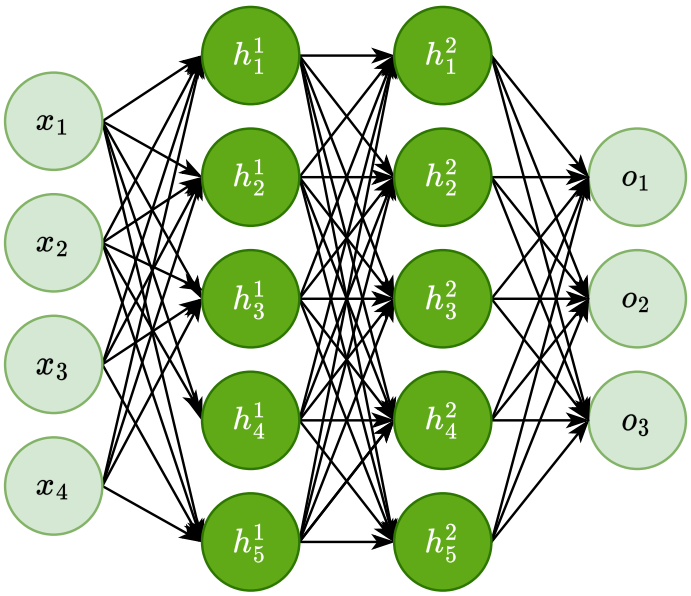


Figure 4.4: Illustration of a fully-connected neural network with three layers. Where $x_1, \ldots, x_4$ and $o_1, \ldots, o_4$ are the input and output units, respectively, h_n^m are the hidden units, where m represents the layer number and n the unit number.

output of the neural network by passing the data through the hidden units h_j sequentially, i.e., from left to right. The computation done in an N-layer network is denoted as

$$\mathbf{Z}^j = \mathbf{W}^j \mathbf{A}^{j-1} + \mathbf{B}^j, \tag{4.11}$$

where $\mathbf{A}^j = \sigma(\mathbf{Z}^j)$ and $j \in [N]$. Should be noted that for the input layer, $\mathbf{A}^0$ is simply the training data $\mathbf{X}$.

4.3.3 Convolutional layer

The convolutional layer conducts the convolution operation between the inputs and the weights. The weights of the convolutional layer are commonly referred to as the kernel tensor, or the filters. Typically, the kernel tensor $\mathcal{K}$ is composed of T square-shaped filters, each having a size of $d \times d$, and T is the number of output channels. This convolutional layer takes in an input tensor $\mathcal{U} \in \mathbb{R}^{X \times Y \times C}$, where X, Y are the spatial dimensions and C is the number of input channels. The output of the convolutional layer is a tensor $\mathcal{V} \in \mathbb{R}^{X' \times Y' \times T}$, where $X' = \frac{X-d+2*P}{S+1}$ and $Y' = \frac{Y-d+2*P}{S+1}$. Padding P and stride S are two quantities that control how far the kernel moves after each dot product and how much it is allowed go out of the edge. The mathematical definition of the convolution operation is provided later in Eq. (6.3).

4.3.4 Convolutional neural networks

CNNs have architectures very similar to those of ordinary fully-connected neural networks: they have a sequential layout of layers followed by activation functions and the same input/output behaviour. The only difference is that the convolutional layers in CNNs are often placed before the fully connected layers.

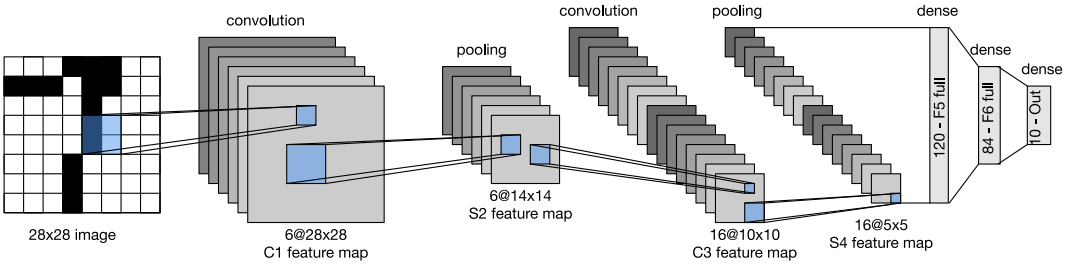


Figure 4.5: LeNet (or LeNet-5) architecture (image sourced from [1]).

One example of this type of neural networks is LeNet [60], which takes 2D grayscale images as inputs. Two convolutional layers and two pooling layers extract the input images features, followed by two fully connected layers that classify the inputs. The network outputs a vector $\hat{y} \in \mathbb{R}^{10}$, with each dimension corresponding to a class to predict. An illustrative visualization of this network is shown in Fig. 4.5.

CHAPTER 5

Tensor methods

Tensor Networks (TNs) [61], are a class of techniques that deal with very large tensors by representing them as collections of small interconnected tensors, also called "blocks", "cores", "factor", or "components". The primary goal of TNs is to approximate the large tensors in a compressed and distributed manner, overcoming the curse of dimensionality associated with these tensors [62].

5.1 Tensor diagrams

The intricate structures of advanced TNs are difficult to comprehend using only mathematical notation. To aid in the comprehension of the interconnections between tensors within TNs, Roger Penrose introduced TN diagrams in the early 1970s [63]. These diagrams represent tensors as nodes with edges, providing a simple graphical representation of complex tensors [8]. Therefore, TN diagrams are a practical tool for visually presenting and conveniently representing complex tensors. Furthermore, the potential of tensor diagrams as a versatile tool for network analysis in the field of deep learning is noteworthy [47].

As illustrated in Fig. 5.1, a tensor is denoted as a node with edges. The number of edges denotes the modes of a tensor, and the value of the edge represents the dimension of the corresponding mode. For example, a node with one edge represents a vector $\mathbf{v} \in \mathbb{R}^{I}$, a node with two edges represents a matrix $\mathbf{M} \in \mathbb{R}^{I \times J}$, and a node with three edges represents a third-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, where in general a node with N edges represents a Nth-order tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \cdots \times I_N}$.

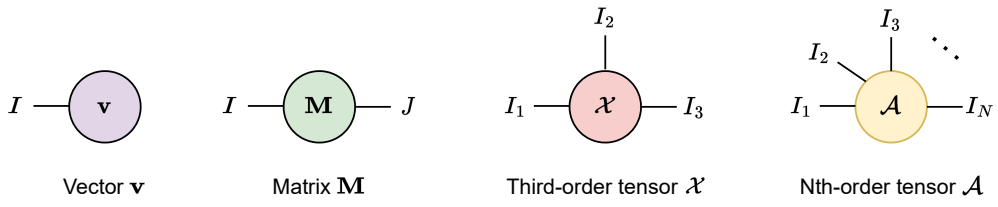


Figure 5.1: Basic symbols for TN diagrams.

As illustrated earlier, tensors are linked by tensor contractions, which involve combining two tensors into one by pairing their corresponding indices. As a consequence, the connected edges vanish, while the dangling edges remain. See Fig. 5.2, where the diagram of the matrix multiplication operation described by equation Eq. (4.4) is shown. It is worth mentioning that matrix multiplication is the most common form of tensor contraction. In general, tensor contraction can be formulated as a tensor product, see Eq. (4.3). To compute tensor contractions among multiple tensors, such as in complex TNs, it is necessary to perform contractions sequentially between each pair of tensors. However, determining the order of these contractions is crucial to achieve better calculation efficiency. [64] highlights the importance of this step in achieving better performance in tensor computations.

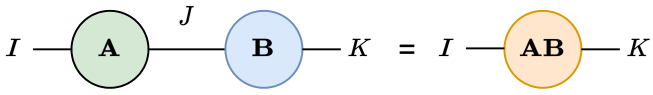


Figure 5.2: Diagram of the most common tensor contraction form, i.e., matrix multiplication.

5.2 Tensor decompositions

In this work, we adopt a unified approach to the terminology of "tensor decomposition" (TD) and "tensor network" as they are equivalent. TD models, such as CP [16] and Tucker decomposition [14], are considered basic types of TNs. It should be noted that different terminologies have been used for the same model, as TNs and TDs originated from different research fields. For instance, the Matrix Product State (MPS) decomposition [65] is also referred to as TT decomposition [15]. Here, we briefly introduce some of the most significant TDs, while in Fig. 5.3, we illustrate each TD by employing TN diagrams.

5.2.1 CANDECOMP/PARAFAC Decomposition

CP factorization factorizes a higher-order tensor $\mathcal{X}$ into a sum of several rank-1 tensor components. The rank of the tensor $\mathcal{X}$ is the minimum number of rank-1 tensors that sum to $\mathcal{X}$, also known as the CP rank. This generalizes the notion of matrix rank to higher-order tensors. For instance, given an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$, each of its elements in the CP format can be expressed as

$$\mathcal{X}_{i_1,i_2,\ldots,i_N} \approx \sum_{r=1}^{R} \mathcal{G}_r \prod_{n=1}^{N} \mathcal{A}^{(n)}_{i_n,r}, \tag{5.1}$$

where R denotes the CP rank, $\mathcal{G}$ represents the diagonal core tensor (consisting of R non-zero elements on the superdiagonal), and $\mathcal{A}^{(n)} \in \mathbb{R}^{I_n \times R}$ represents a series of factor matrices. See (a) in Fig. 5.3 for the TN diagram that describes this decomposition.

One of the issues with this method is that computing the tensor rank, i.e., the number of rank-1 tensor components, is a NP-hard problem [10]. Therefore, a predefined CP rank R should be known in advance, this can be seen as fixing hyperparameters and using them to fit different CP-based models [26]. Other possible ways of computing the rank, such as estimating the rank from the data, e.g., using Bayesian approaches [66], or using deep neural networks [67] are also possible. An illustrative example can be found in Fig. 5.4, where CP decomposition is applied to a third-order tensor $\mathcal{X}$.

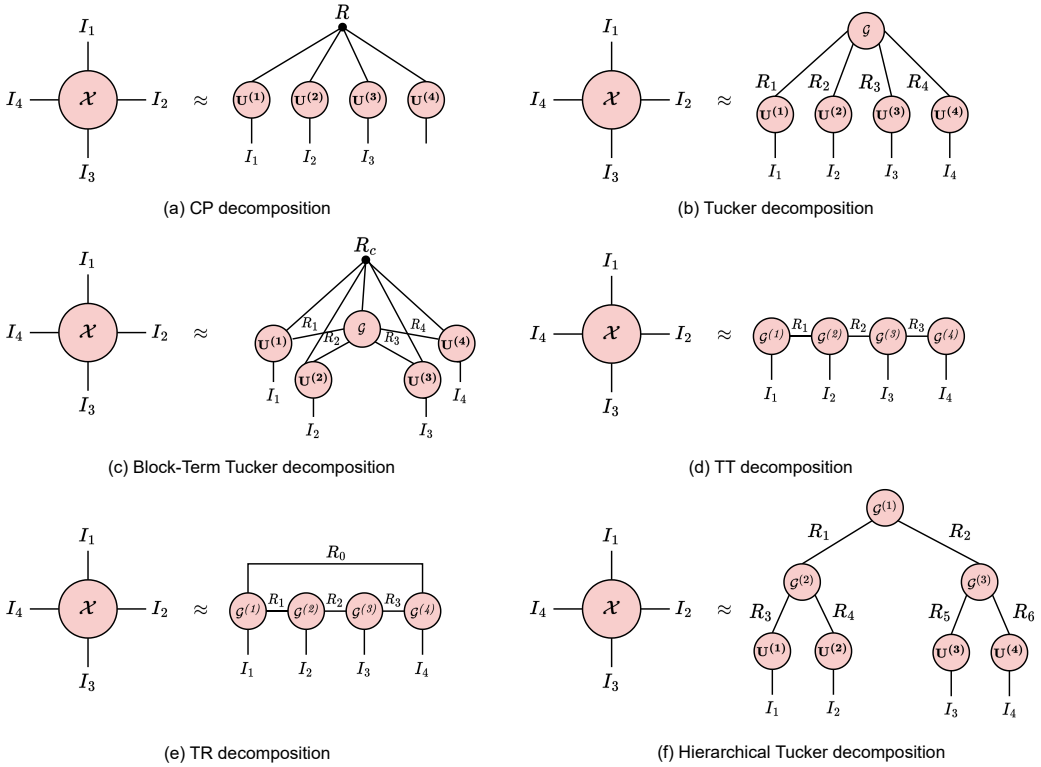


Figure 5.3: TN diagrams of different tensor decompositions.

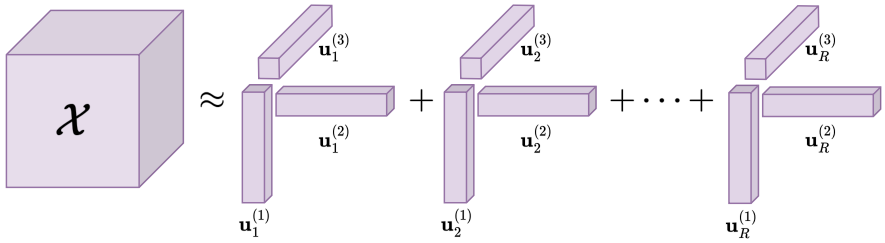


Figure 5.4: CP decomposition of a third-order tensor $\mathcal{X}$ into a sum of rank-1 tensors.

5.2.2 Tucker Decomposition

The Tucker decomposition [14] factorizes a higher-order tensor non-uniquely into a core tensor multiplied by a corresponding factor matrix along each mode. More precisely, given an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$, the Tucker decomposition can be formulated for each element as

$$\mathcal{X}_{i_1,i_2,\ldots,i_N} \approx \sum_{r_1,\ldots,r_N=1}^{R_1,\ldots,R_N} \mathcal{G}_{r_1,r_2,\ldots,r_N} \prod_{n=1}^{N} \mathcal{A}^{(n)}_{i_n,r_n}, \tag{5.2}$$

where $R = \{R_1, R_2, \ldots, R_N\}$ denotes the Tucker ranks, $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \cdots \times R_N}$ denotes the core tensor, and $\mathcal{A}^{(n)} \in \mathbb{R}^{I_n \times R_n}$ denotes a factor matrix. The core tensor captures interactions between the columns of factor matrices, and if $R_n \ll I_n, \forall n$, the core tensor can be viewed as a compressed version of $\mathcal{X}$. It should be noted that in this case, $R_1, R_2, \ldots, R_N$

can take different values, which leads to the non-uniqueness of its decomposition results; see Fig. 5.5 for an illustrative example, and (b) in Fig. 5.3 to visualize the TN diagram of this decomposition. Finally, its worth be noting that by imposing the factor matrices to be orthonormal, the Tucker decomposition is known as the higher-order singular value decomposition (HOSVD) [10].

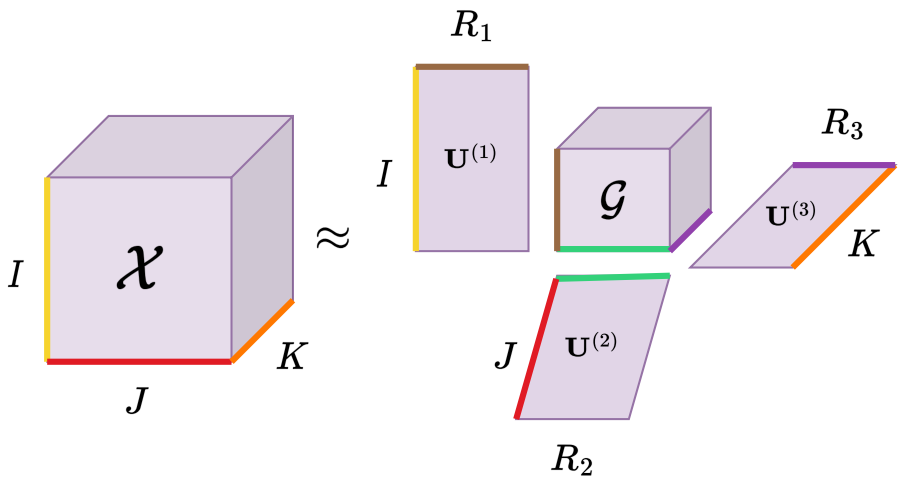


Figure 5.5: Tucker decomposition of a third-order tensor $\mathcal{X}$.

5.2.3 Block-Term Tucker Decomposition

Block-Term Tucker (BTT) decomposition is a more generalized factorization proposed to make a tradeoff between CP and Tucker decompositions [19]. Notice that both CP and Tucker methods decompose a tensor into a core tensor multiplied by a matrix along each mode, where CP also imposes an additional superdiagonal constraint on the core tensor, i.e., it tries to simplify the structural information on the core tensor. BTT imposes a block diagonal constraint on Tucker's core tensor, more specifically, BTT aims to decompose a tensor $\mathcal{X}$ into a sum of several Tucker decompositions with low Tucker ranks. For example, given a fourth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times I_4}$, the BTT decomposition for each element can be formulated as

$$\mathcal{X}_{i_1,i_2,i_3,i_4} \approx \sum_{r_C=1}^{R_C} \sum_{r_1,r_2,r_3,r_4=1}^{R_T,R_T,R_T,R_T} \mathcal{G}_{r_C,r_1,r2,r_3,r_4} \mathcal{A}^{(1)}_{r_C,i_1,r_1} \mathcal{A}^{(2)}_{r_C,i_2,r_2} \mathcal{A}^{(3)}_{r_C,i_3,r_3} \mathcal{A}^{(4)}_{r_C,i_4,r_4}, \quad (5.3)$$

where $\mathcal{G} \in \mathbb{R}^{R_C \times R_T \times R_T \times R_T \times R_T}$ denotes the R_C core tensors of Tucker decompositions, each $\mathcal{A}^{(n)} \in \mathbb{R}^{R_C \times I_n \times R_T}$ denotes the R_C corresponding factor matrices of the Tucker decompositions, R_T denotes the Tucker rank (R_T, R_T, R_T, R_T), and R_C denotes the CP rank. This decomposition combines the benefits of both CP and Tucker methods, where we also have the flexibility to degenerate the BTT decomposition to get CP decomposition when Tucker rank is equal to 1, and Tucker decomposition when CP rank is equal to 1. See (c) of Fig. 5.3 for the TN diagram that defines this decomposition.

5.2.4 Tensor Train Decomposition

The Tensor-Train decomposition [15], also known as MPS decomposition in quantum physics [65], factorizes a higher-order tensor $\mathcal{X}$ into a linear multiplication of a series of third-order tensors. For example, given an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$, the TT decomposition can be formulated elementwise as

$$\mathcal{X}_{i_1,i_2,\ldots,i_N} \approx \sum_{r_1,r_2,\ldots,r_{N-1}=1}^{R_1,R_2,\ldots,R_{N-1}} \mathcal{G}^{(1)}_{1,i_1,r_1} \mathcal{G}^{(2)}_{r_1,i_2,r_2} \mathcal{G}^{(3)}_{r_2,i_3,r_3} \ldots \mathcal{G}^{(N)}_{r_{N-1},i_N,1}, \tag{5.4}$$

where $R = \{R_1, R_2, \ldots, R_{N-1}\}$ denotes the TT ranks, $\mathcal{G}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$ denotes a third-order tensor and $R_0 = R_N = 1$, which means that $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(N)}$ are actually two matrices, i.e., the boundary conditions (*open boundary conditions*) of the TT decomposition; see (d) in Fig. 5.3 for the TN diagram of this decomposition, and Fig. 5.6 for a visual example. It should be noted that the TT decomposition can be computed easily by applying SVD recursively.

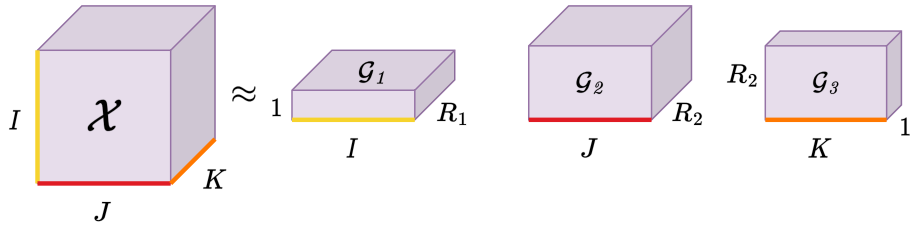


Figure 5.6: Tensor Train decomposition of a third-order tensor $\mathcal{X}$ into a series of third-order cores, where the boundary condition restricts $R_1 = R_{N+1} = 1$.

5.2.5 Tensor Ring Decomposition

As introduced in Section 5.2.4, TT decomposition suffers from its two endpoints, which hinder the representation ability and flexibility of the TT-based model. To alleviate this issue, the researchers have released the full power of the linear structure by connecting (linking via tensor contraction) the two endpoints, forming the Tensor Ring decomposition. The formulation of the TR decomposition of the tensor $\mathcal{X}$ can be expressed elementwise as

$$\mathcal{X}_{i_1,i_2,\ldots,i_N} \approx \sum_{r_0,r_1,\ldots,r_{N-1}}^{R_0,R_1,\ldots,R_{N-1}} \mathcal{G}^{(1)}_{r_0,i_1,r_1} \mathcal{G}^{(2)}_{r_1,i_2,r_2} \mathcal{G}^{(3)}_{r_2,i_3,r_3} \ldots \mathcal{G}^{(N)}_{r_{N-1},i_N,r_0}, \tag{5.5}$$

where $R = \{R_0, R_1, \ldots, R_N\}$ denotes the TR ranks, each $\mathcal{G}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$ is a third-order tensor, and $R_0 = R_N$. Notice that in this case it is not necessary to follow a strict order when multiplying its nodes $\mathcal{G}^{(n)}$. See (e) in Fig. 5.3 for the TN diagram that represents this decomposition.

5.2.6 Hierarchical Tucker Decomposition

Hierarchical Tucker decomposition is an special case of tensor decomposition, where a tree-like structure is used, i.e., hierarchical levels with respect to the order of the tensor.

Recursive decomposition of HT involves breaking it down into intermediate components, which are referred to as frames. This process follows a top-to-bottom binary tree structure, where every frame corresponds to a distinct node, and each node possesses its own set of associated dimensions. More specifically, for a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$, we can build a binary tree with a root node associated with $S_{set} = \{1, I_2, \ldots, N\}$ and $\mathcal{U}_{S_{set}}$ as a root frame. The index sets associated with left child node $\mathcal{U}_{S_{set1}}$ and right child node $\mathcal{U}_{S_{set2}}$ are denoted as $S_{set1}, S_{set2} \subseteq S_{set}$, respectively. Notice that these child nodes are decomposed in a similar manner, e.g., the left child node $\mathcal{U}_{S_{set1}} \in \mathbb{R}^{R_1 \times I_{min(S_{set1})} \times \cdots \times I_{max(S_{set1})}}$ can also be recursively decomposed into its left child node $\mathcal{U}_{D_{set1}}$ and right child node $\mathcal{U}_{D_{set2}}$. The first three steps of the explained recursion can be defined as

$$\mathcal{U}_{S_{set}} \approx \mathcal{G}_s \times_1^2 \mathcal{U}_{S_{set1}} \times_1^2 \mathcal{U}_{S_{set2}}, \tag{5.6}$$

$$\mathcal{U}_{S_{set1}} \approx \mathcal{G}_{s1} \times_1^2 \mathcal{U}_{D_{set1}} \times_1^2 \mathcal{U}_{D_{set2}}, \tag{5.7}$$

$$\mathcal{U}_{S_{set2}} \approx \mathcal{G}_{s2} \times_1^2 \mathcal{U}_{D_{set3}} \times_1^2 \mathcal{U}_{D_{set4}} \tag{5.8}$$

where $\mathcal{G}_s \in \mathbb{R}^{R_1 \times R_2}$, $\mathcal{G}_{s1} \in \mathbb{R}^{R_1 \times R_3 \times R_4}$, $\mathcal{G}_{s2} \in \mathbb{R}^{R_2 \times R_5 \times R_6}$ are the transfer tensors, and $R = \{R_1, \ldots, R_6\}$ are the hierarchical ranks. An illustrative example can be found in (f) of Fig. 5.3.

CHAPTER 6

Network compression using TDs

In this chapter, we will explore the application of tensor decomposition techniques to compress feed-forward (or fully-connected) and convolutional layers in deep neural networks. More precisely, we will explain how some of the TDs introduced in Section 5.2 such as CP, Tucker and TT decompositions, can be used to compress neural networks effectively.

6.1 Tensorization of fully-connected layers

To introduce the compression of fully-connected layers with tensor decompositions, we first provide an illustrative example using the popular (truncated) Singular Value Decomposition (SVD) method [68], and then extend this technique to higher-order decomposition. Since fully-connected layer weights are stored in matrices instead of tensors, it is natural to apply the (truncated) SVD algorithm to approximate these weight matrices.

Given a weight matrix $\mathbf{W} \in \mathbb{R}^{S \times T}$, we can decompose $\mathbf{W} \approx \mathbf{U}\mathbf{S}\mathbf{V}^T$ using (truncated) SVD algorithm, where $\mathbf{U} \in \mathbb{R}^{S \times R}$, $\mathbf{S} \in \mathbb{R}^{R \times R}$, $\mathbf{V} \in \mathbb{R}^{T \times R}$, and $R \in \mathbb{Z}^+$. Therefore, we can describe the original j^{th}-fully connected layers operation using the approximated form as

$$\mathbf{A}^j = \sigma(\mathbf{U}^j\mathbf{S}^j(\mathbf{V}^j)^T\mathbf{A}^j + \mathbf{B}^j), \tag{6.1}$$

where $\mathbf{A}^{j-1}$ is the output of the previous layer (or the input matrix $\mathbf{X}$, if $j - 1 = 0$). Notice that if the choice of the rank R if significantly smaller than S and T, this approximated form allows us to reduce the number of parameters in the weight matrix. While the SVD algorithm is simple and easy to implement, it is not well-suited for compressing large-scale weight matrices. Modern neural networks commonly employ weight matrices with a substantial number of parameters, ranging from tens to hundreds of thousands. This necessitates a higher compression rate than what the SVD algorithm can provide.

The utilization of tensor decompositions in neural networks through the tensorization of fully-connected layers was initially proposed in [13], where the term Tensor Neural Networks (TNNs) was coined. As said before, the weights of fully-connected layers are represented by matrices, direct application of tensor decompositions is not feasible. Therefore, the weight matrices need to be reshaped in order to obtain a higher-order tensor. Specifically, given an input matrix $\mathbf{W} \in \mathbb{R}^{I \times J}$, with dimensions expressed as $I = I_1 \times I_2 \times \cdots \times I_N$

and $J = J_1 \times J_2 \times \cdots \times J_N$, $\mathbf{W}$ is first tensorized by reshaping it into a higher-order tensor of size $I_1 \times I_2 \times \cdots \times I_N \times J_1 \times J_2 \times \cdots \times J_N$. Then, by permuting the dimensions and reshaping it once more, an Nth-order tensor of size $I_1 J_1 \times I_2 J_2 \times \cdots \times I_N J_N$ is obtained. This tensor is subsequently compressed using the corresponding tensor decomposition format, such as the TT format. Finally, during the forward pass, there are two main strategies to consider:

- Reconstructed: Reconstruct the complete (approximated) weights and perform a regular linear layer forward pass.
- Factorized: Contract the input tensor with the factors of the decomposition. This can be faster for very small ranks, i.e., when the factorization factors are very small.

6.2 Tensorization of convolutional layers

Unlike in fully-connected layers, the weights of convolutional layers are naturally represented as tensors. For example, a 2D convolution is directly represented by a 4^{th}-order tensor. Therefore, we can directly apply tensor decompositions to compress convolutional layers.

While the utilization of tensor decomposition in CNNs is a recent and currently relevant development, the concept of representing linear operators through separable representations using tensor decomposition is not new (refer to [62] for more details). In this way, the motivation behind depthwise separable convolutions is to improve the efficiency and effectiveness of CNNs.

6.2.1 Depthwise separable convolutions

Traditional convolutional layers in CNNs apply a single filter to all input channels, resulting in a large number of computations. Depthwise separable convolutions aim to reduce this computational cost by decomposing the convolution operation into two separate steps: depthwise convolution and pointwise convolution.

Depthwise convolution applies a separate filter to each input channel independently, capturing channel-wise spatial correlations. This step significantly reduces the number of parameters and computations compared to traditional convolutions.

Pointwise convolution, also known as 1×1 convolution, which can be viewed as a tensor contraction, is a fundamental example of convolution. It is commonly employed in deep neural networks to introduce data bottlenecks where the channel-wise information obtained from the depthwise convolution is combined, as seen in architectures like MobileNet [69] and Inception [70]. Let's consider a 1×1 convolution operator denoted as Φ. It is defined by a kernel tensor $\mathcal{W} \in \mathbb{R}^{T \times C \times 1 \times 1}$ and is applied to an activation tensor $\mathcal{X} \in \mathbb{R}^{C \times H \times W}$. The squeezed version of the kernel along the first mode is represented as $\mathbf{W} \in \mathbb{R}^{T \times C}$. Thus, we can express this as follows:

$$\Phi(\mathcal{X})_{t,y,x} = \mathcal{X} \star \mathcal{W} = \sum_{k=1}^{C} \mathcal{X}_{t,k,y,x} \mathcal{W}_{k,y,x} = \mathcal{X} \times_1 \mathcal{W}, \tag{6.2}$$

where $x = y = 1$ as we are considering a 1×1 convolution.

The depthwise separable convolution operation can be viewed as a form of tensor decomposition, where the original kernel tensor is decomposed into a set of smaller tensors.

6.2.2 Kruskal convolutions

CP decomposition allows separating modes of a convolutional kernel, resulting in a Kruskal form. This was proposed in [26], where the authors start from a pretrained convolutional kernel and apply CP decomposition to it in order to obtain a separable convolution. In this case, the CP decomposition was achieved by minimizing the reconstruction error between the pretrained weights and the corresponding CP approximation. The authors demonstrated both space savings and computational speedups. Using a CP decomposition

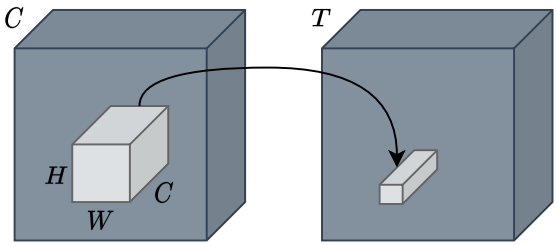


Figure 6.1: Standard convolution operation.

to factorize the kernel offers a notable benefit: the resulting factors can serve as parameters for a set of efficient depthwise separable convolutions, effectively replacing the original convolution operation [26]. Let's consider a standard convolution illustrated in Fig. 6.1 and denoted as:

$$\mathcal{F}_{t,y,x} = \sum_{k=1}^{C}\sum_{j=1}^{H}\sum_{i=1}^{W} \mathcal{W}(t,k,j,i)\mathcal{X}(k,j+y,i+x) \tag{6.3}$$

It's Kruskal convolution is obtained by expanding the kernel $\mathcal{W}$ in the CP form as

$$\mathcal{F}_{t,y,x} = \underbrace{\sum_{r=1}^{R} \mathbf{U}_{t,r}^{(T)} \underbrace{\left[\sum_{i=1}^{W} \mathbf{U}_{i,r}^{(W)} \underbrace{\left(\underbrace{\sum_{j=1}^{H} \mathbf{U}_{j,r}^{(H)} \left[\underbrace{\sum_{k=1}^{C} \mathbf{U}_{k,r}^{(C)} \mathcal{X}(k,j+y,i+x)}_{1\times 1 \text{ convolution}} \right]}_{\text{depthwise convolution}} \right)}_{\text{depthwise convolution}} \right]}_{1\times 1 \text{ convolution}}, \tag{6.4}$$

This expression is explained as follows: an initial 1×1 convolution reduces input channels to the rank (blue). Two depthwise convolutions are then applied to the height and width of the activation tensor (red and green). Lastly, a second 1×1 convolution restores the number of channels from the rank of the CP decomposition to the desired output channel count (black) (see Fig. 6.2).

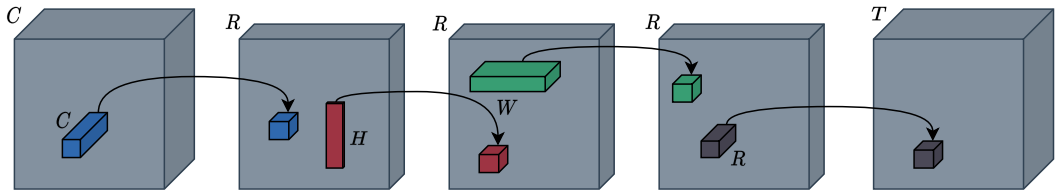


Figure 6.2: Convolution operation with a CP-decomposed kernel, i.e., Kruskal form.

6.2.3 Tucker convolutions

As previously, we consider the convolution $\mathcal{F} = \mathcal{X} \star \mathcal{W}$ described in Eq. (6.3). However, in this case we consider using Tucker decomposition to compress convolutional layers as proposed in [71].

First, instead of a Kruskal structure, the convolution kernel is assumed to admit Tucker format described as follows:

$$\mathcal{W}(t,s,j,i) = \sum_{r_1=0}^{R_1}\sum_{r_2=0}^{R_2}\sum_{r_3=0}^{R_3}\sum_{r_4=0}^{R_4} \mathcal{G}_{r_1,r_2,r_3,r_4}\mathbf{U}^{(T)}_{t,r_1}\mathbf{U}^{(C)}_{s,r_2}\mathbf{U}^{(H)}_{j,r_3}\mathbf{U}^{(W)}_{i,r_4}, \tag{6.5}$$

where $\mathbf{U}^{(n)}$, with $n = \{T, C, H, W\}$, are the four factor matrices, and $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times R_3 \times R_4}$ is the core tensor that represents the interactions between the modes.

Using the decomposed kernel tensor, we can create an efficient reformulation [71]: initially, the factors along the spatial dimensions are incorporated into the core tensor by expressing $\mathcal{H} = \mathcal{G} \times_3 \mathbf{U}^{(H)}_{j,r_3} \times_4 \mathbf{U}^{(W)}_{i,r_4}$. Rearranging the terms, we observe that a Tucker convolution is equivalent to sequentially handling the channel count transformation, performing a small convolution, and then restoring the channel dimension from the rank to the desired number of channels:

$$\mathcal{F}_{t,y,x} = \underbrace{\sum_{r_1=1}^{R_1} \mathbf{U}^{(T)}_{t,r_1} \underbrace{\left[\sum_{j=1}^{H}\sum_{i=1}^{W}\sum_{r_2=1}^{R_2} \mathcal{H}_{r_1,r_2,j,i}, \underbrace{\left(\sum_{k=1}^{C} \mathbf{U}^{(C)}_{k,r_2} \mathcal{X}(k, j+y, i+x) \right)}_{1\times 1 \text{ convolution}} \right]}_{H\times W \text{ convolution}}}_{1\times 1 \text{ convolution}}. \tag{6.6}$$

This expression can be described more precisely as follows: after decomposing the full kernel, the factors corresponding to input and output channels are utilized to parameterize 1 x 1 convolutions, applied to the initial (blue) and final (green) stages, respectively. The remaining two factors are integrated into the core tensor and employed to parameterize a (small) regular 2D convolution (red) (see Fig. 6.3).

6.2.4 TT-based convolutions

A direct (and straightforward) approach to represent convolutional kernels in TT-format is by applying the TT decomposition directly to the kernel tensor $\mathcal{W}$ and dividing the forward

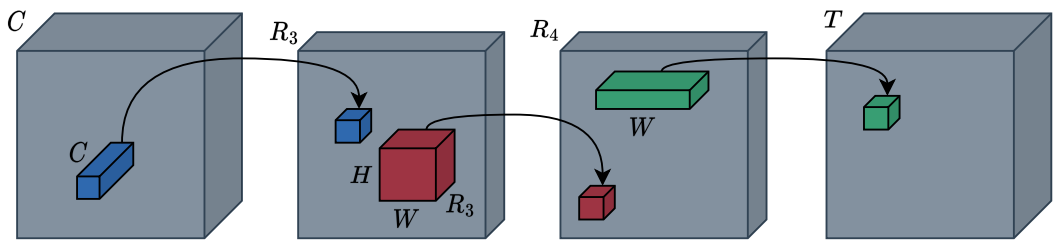


Figure 6.3: Convolution operation with a Tucker-decomposed kernel.

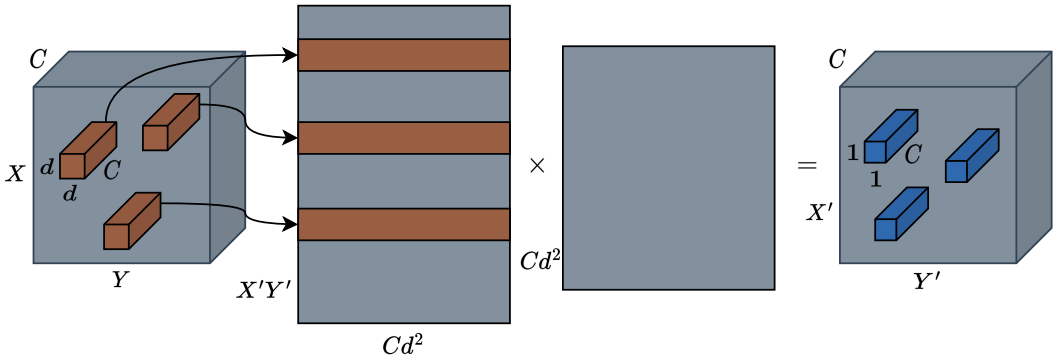


Figure 6.4: Convolution operation with a TT-decomposed kernel, where the leftmost tensor represents the input tensor $\mathcal{X}$, the next two matrices are $\mathbf{U}$ and $\mathbf{K}$, respectively, and the last tensor represents the output tensor $\mathcal{V}$.

pass into sequential steps. However, in [27], the authors demonstrated the inadequate performance of this approach and introduced a novel method inspired by the notion that a convolutional layer can be expressed as a matrix-by-matrix multiplication [27].

As shown in Fig. 5.6, the 2D convolution between a third-order input tensor $\mathcal{X} \in \mathbb{R}^{X \times Y \times C}$, and a fourth-order kernel tensor $\mathcal{W} \in \mathbb{R}^{d \times d \times C \times T}$, is equivalent to the matrix-by-matrix multiplication [27]. More precisely, we introduce two matrices, $\mathbf{U}, \mathbf{K}$ to hold the same data in matrix format:

$$\mathcal{X}(x+i-1, y+j-1, c) = \mathbf{U}(x + X'(y-1), i + d(j-1) + d^2(c-1)), \tag{6.7}$$

$$\mathcal{W}(i, j, c, t) = \mathbf{K}(i + d(j-1) + d^2(t-1), c), \tag{6.8}$$

where we assume that $X' = X - d + 1$ and $Y' = Y - d + 1$. Then, we use $\mathbf{U}$ and $\mathbf{K}$ matrices to compute the intermediate output matrix $\mathbf{V}$ as

$$\mathbf{V} = \mathbf{U}\mathbf{K}. \tag{6.9}$$

Finally, by reshaping the matrix $\mathbf{V}$ back into tensor format, we recover the output tensor $\mathcal{V} \in \mathbb{R}^{X' \times Y' \times T}$ as

$$\mathcal{V}(x, y, t) = \mathbf{V}(x, X'(y-1), t). \tag{6.10}$$

Once the input and kernel tensors are matricized, we proceed by reshaping the kernel tensor into a high-dimensional tensor denoted as $\mathcal{W}' \in \mathbb{R}^{d^2 \times c_1 t_1 \times c_2 t_2 \times \cdots \times c_d t_d}$, where $C = \prod_{i=1}^{d} C_i$ and $T = \prod_{i=1}^{d} T_i$. This factorization assumes the existence of such a decomposition. To ensure that the factorization is always feasible, we introduce empty

channels as needed. In summary, the kernel matrix $\mathbf{K}$ is transformed into a $(d-1)$-dimensional tensor, where the first dimension has a size of d^2, and the remaining dimensions have sizes of C_iT_i for i ranging from 1 to d. This tensorization process is mathematically defined as

$$\begin{aligned}\mathbf{K}(x+d(y-1)+d^2(c'-1),c) &= \mathcal{K}'((x+d(y-1),1,(c_1,t_1),\ldots,(c_d,t_d))) \\ &\approx \mathcal{G}^0(x+d(y-1),1)\mathcal{G}^1(c_1,t_1)\ldots\mathcal{G}^d(c_d,t_d),\end{aligned} \tag{6.11}$$

It should be noted that the tuple (c_k, t_k) represents the compound index for the k-th dimension, where c_k is utilized as the row index and t_k serves as the column index. See Fig. 6.5 for an more illustrative example of the matricization, tensorization and decomposition process Finally, we can re-formulate the forward pass using the compound index defined

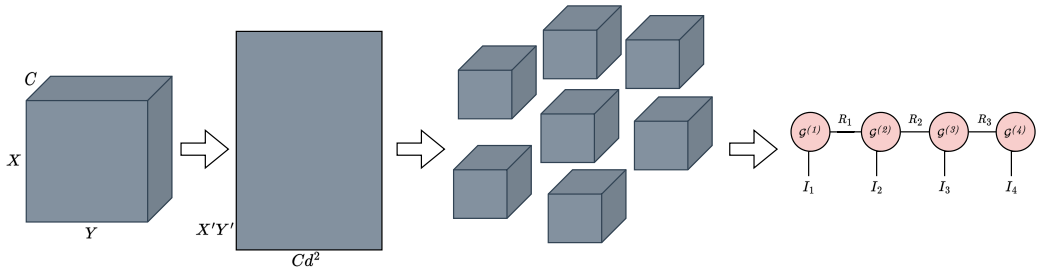


Figure 6.5: Matricization, tensorization and decomposition process.

in Eq. (6.11). That is, we reshape the data matrix $\mathbf{U}$ into a tensor $\mathcal{U}' \in \mathbb{R}^{X\times Y\times C_1\times\ldots C_d}$ and compute the output tensor $\mathcal{V}' \in \mathbb{R}^{X'\times Y'\times T_1\times\cdots\times T_d}$ as follows

$$\begin{aligned}\mathcal{V}' = \sum_{i,j}^{d}\sum_{c_1,\ldots,c_d}\mathcal{U}'(i+x-1,j+y-1,c_1,\ldots,c_d)\times \\ \mathcal{G}^0(x+d(y-1),1)\mathcal{G}^1(c_1,t_1)\ldots\mathcal{G}^d(c_d,t_d).\end{aligned} \tag{6.12}$$

Then, we reshape the intermediate output tensor $\mathcal{V}'$ into the final output tensor $\mathcal{V} \in \mathbb{R}^{X'\times Y'\times T}$. Should be noted that the tensor cores $\mathcal{G}$'s are trained as the layer's parameters, and the optimization process can be carried out utilizing any autograd package [72].

6.3 Complexity analysis

Let's consider the input tensor $\mathcal{U} \in \mathbb{R}^{X\times Y\times C}$, where X, Y are the spatial dimensions and C is the number of input channels. If we assume that there is no bias term, an ordinary convolution has $TCHW$ parameters, where in most of the cases $H = W$. Thus, if $d = H = W$, we can rewrite the number of parameters as TCd^2, and the number of multiplication-addition operations are defined by $TCd^2(X-d+1)(Y-d+1)$.

By applying CP decomposition, the parameter count reduces to $R(C+2d+T)$, and the number of multiplication-addition operations decreases to $R(C+2d+T)(X-d+1)(Y-d+1)$. Hence, if we assume that the chosen rank R is significantly smaller than C and T, with $R \approx \frac{CT}{C+T}$, the compression scheme exhibits a complexity improvement on the order of d^2. The compression ratio of the CP decomposition is defined as

$$C_{\text{cp}} = \frac{TCHW}{R(C+2d+T)}, \tag{6.13}$$

where the speed-up ratio can be defined in the same way as

$$S_{\text{cp}} = \frac{TCd^2(X-d+1)(Y-d+1)}{R(C+2d+T)(X-d+1)(Y-d+1)}. \tag{6.14}$$

According to [73], with Tucker decomposition reduces the number of parameters to $CR_3+TR_4+R_3R_4d^2$, and the number of multiplication-addition operations to $CR_3XY+(R_3d^2+T)R_4(X-d+1)(Y-d+1)$. We can define the compression ratio of the Tucker decomposition as

$$C_{\text{tucker}} = \frac{TCHW}{CR_3+TR_4+R_3R_4d^2}, \tag{6.15}$$

and the speed-up ratio as

$$S_{\text{tucker}} = \frac{TCd^2(X-d+1)(Y-d+1)}{CR_3XY+(R_3d^2+T)R_4(X-d+1)(Y-d+1)}, \tag{6.16}$$

where both of them are bounded by $\frac{TC}{R_3R_4}$.

Following the calculations in [74], TT decomposition reduces the number of parameters to $CR_3+dR_3R+dR_4R+R_4T$, while also reducing the number of multiplication-addition operations to $CR_3XY+dR_3RXY+dR_4RX'Y'+R_4TX'Y'$, where $X'=(X-d+1)$ and $Y'=(Y-d+1)$. Therefore, we can define the compression ratio for the TT decomposition as

$$C_{\text{TT}} = \frac{TCHW}{CR_3+dR_3R+dR_4R+R_4T}, \tag{6.17}$$

and the speed-up ratio is calculated as

$$S_{\text{TT}} = \frac{TCd^2(X-d+1)(Y-d+1)}{CR_3XY+dR_3RXY+dR_4RX'Y'+R_4TX'Y}. \tag{6.18}$$

Furthermore, a summary of the complexities for each tensor decomposition can be found in Table 6.1.

Method	**Space complexity**	**Computational complexity**
Standard convolution	$TCHW$	$TCd^2(X-d+1)(Y-d+1)$
CP convolution	$R(C+2d+T)$	$R(C+2d+T)(X-d+1)(Y-d+1)$
Tucker convolution	$CR_3+TR_4+R_3R_4d^2$	$CR_3XY+(R_3d^2+T)R_4(X-d+1)(Y-d+1)$
TT convolution	$CR_3+dR_3R+dR_4R+R_4T$	$CR_3XY+dR_3RXY+dR_4RX'Y'+R_4TX'Y'$

Table 6.1: Comparison of complexities.

6.4 Rank selection

In this work, many tensor methods discussed rely on knowledge or estimation of the decomposition's rank. However, determining the rank of a tensor is generally a computationally difficult problem (NP-hard) [59, 75], and practical applications often resort to heuristic approaches. While the precise rank selection is less critical within a deep framework, since all parameters are jointly trained end-to-end, the task of rank selection remains a significant challenge.

Moreover, the chosen rank for the decomposition strongly impacts performance and compression ratio, effectively controlling the trade-off between these two metrics. It directly influences the expressiveness and complexity of the decomposition model. Unfortunately, the process of selecting optimal rank values through trial and error is highly inefficient and time-consuming. Additionally, it is worth noting that rank selection also affects the computational performance achieved. By considering hardware characteristics or restrictions, selecting appropriate ranks can bridge the gap between theoretical advancements and practical implementations [44].

Given the importance of rank selection (or estimation), various approaches have been proposed. For instance, in practical scenarios, one may initially select a rank that preserves around 80% of the parameters and gradually decrease it using multi-stage compression techniques [37]. When working with pretrained networks, it is also possible to approximate the rank by employing Bayesian matrix factorization [76] on the unfolding of the weight tensor, thereby obtaining an estimate of the rank [71, 37]. Other approaches such as [67] have used neural networks and reinforcement learning to estimate the rank. Alternatively, one can incorporate a Lasso-type penalty into the loss function to automatically force certain components to zero, aiding in rank determination [77].

CHAPTER 7

Results

This chapter presents a comprehensive benchmarking analysis aimed at evaluating the performance of tensor decompositions for compressing neural networks. Specifically, we assess the effectiveness of these methods using two distinct tensorization strategies. The first strategy involves training tensorized models from the ground up, while the second strategy involves applying tensorization to pretrained models.

7.1 Experimental setup

All the models were built with Pytorch [78] and trained with the help of Pytorch Lightning [79] primitives. As a experimental platform, we use several hardware accelerators, more precisely, we use some Nvidia RTX 3090 and Nvidia A5000 GPUs equipped with 24GB of VRAM.

For the tensorization, there exist many libraries prepared for working with tensors easily such as Tensorly [80], TensorNetwork [81], or cuTensor [82], furthermore, there are some other libraries specifically designed for working with tensor decompositions for neural networks compression. In our case, we decided to use tensorly-torch [80], a wrapper of Tensorly and Pytorch that facilitates the tensorization of neural networks through several tensor decompositions, using a unified interface and providing several backends, e.g., Pytorch, Numpy, JAX, CuPy, and so on. However, notice that other libraries such as Tednet [83], which is a solid alternative that provide similar functionalities.

7.1.1 Dataset

The CIFAR-10 dataset is composed of 60,000 images, with each color image (RGB) having a resolution of 32x32 pixels. These images are evenly distributed across ten object classes, with each class containing 6,000 images. The ten classes are as follows: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

The CIFAR-10 dataset [84] is divided into two subsets: a training set and a test set. The training set consists of 50,000 images, with each class having 5,000 samples, where we randomly select 5000 examples for the validation set. Note that these splits are made using the same random seed to maintain the reproducibility of the experiments. The remaining

10,000 images form the test set, serving as an independent benchmark to evaluate the generalization and performance of the trained models.

7.1.2 Training and evaluation framework

In order to ensure fairness during the training process of all the models presented in Section 7.2.1, several measures were taken. Specifically, all models were trained for 50 epochs, utilizing a batch size of 128 and a consistent learning rate of $1e{-}3$. The widely recognized Adam optimizer [85] was employed, with default hyperparameters set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

In order to assess the performance of the models, we employ identical evaluation metrics for models tensorized using both tensorization strategies, i.e., tensorized from scratch in Section 7.2.1 and tensorized from pretrained models in Section 7.2.2. Firstly, considering the nature of our task as a balanced multi-class classification problem, we utilize the accuracy metric, which is defined as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}, \tag{7.1}$$

where *TP* are true positives, *TN* true negatives, *FP* false positives, and *FN* false negatives. Additionally, as we are interested in measuring the trade-off between accuracy and parameter reduction, we need to define a metric to measure this reduction. Therefore, we define the compression ratio metric, which measures the size reduction achieved dividing the number of parameters in the uncompressed model by the number of parameters in the compressed model. This is defined mathematically as

$$\text{Compression ratio} = \frac{\#\text{params uncompressed}}{\#\text{params compressed}}. \tag{7.2}$$

We also measure the model size, where if we assume that the floating point precision used is 32, we can calculate the model size in Megabytes (MB) as

$$\text{Model size} = \#\text{params} * (32/8) * 1e{-}6. \tag{7.3}$$

7.2 Analysis of the results

To test the performance of each decomposition scheme, we apply each of the methods to the convolutional layers of two different architectures like ResNet [86], and VGG [21]. More precisely, we use several size configurations of these two architectures, i.e., ResNet18, ResNet50, VGG11, and VGG19 networks.

VGG11 and VGG19 networks are built sequentially stacking 8 and 16 convolutional layers, respectively, followed by 3 fully-connected layers. In the same way, ResNet18 is built stacking 17 convolutional layers, followed by a single fully-connected layer. On the other hand, ResNet50, which is composed by 49 convolutional layers followed by a fully-connected layer, divides the convolutional layers into several blocks, with each block containing a series of convolutions, i.e., 1×1 convolution, 3×3 convolution and 1×1 convolution, known as data bottlenecks [86].

To tensorize the networks, we replace all the convolutional layers but the first one of the different architectures by the TD-based convolutions, i.e., CP-decomposed (Section 6.2.2),

Tucker-decomposed (Section 6.2.3) and TT-decomposed (Section 6.2.4) convolutions. A summary of the different configurations is shown in Table 7.1.

We also tested two additional intermediate configurations, more precisely, ResNet34 and VGG16, but as the results are very similar to those of the other configurations, we have not included them in this analysis, and they can be found in Appendix A.

Network	Baseline configuration	Tensorized configuration
VGG11	8 convolutional layers 3 fully-connected layers	1 convolutional layer 7 TD-convolutional layers 3 fully-connected layers
VGG19	16 convolutional layers 3 fully-connected layers	1 convolutional layer 15 TD-convolutional layers 3 fully-connected layers
ResNet18	17 convolutional layers 1 fully-connected layer	1 convolutional layer 16 TD-convolutional layers 1 fully-connected layer
ResNet50	49 convolutional layers 1 fully-connected layer	1 convolutional layer 48 TD-convolutional layers 1 fully-connected layer

Table 7.1: Networks configuration summary.

7.2.1 Tensorization from scratch

In this tensorization strategy, we begin by selecting a baseline network, such as an untrained ResNet18 model, as our starting point. Subsequently, we apply tensorization techniques to the baseline model at various compression ratios, and proceed to train these tensorized models from the beginning. The objective of this experiment is to assess whether the tensorized networks can preserve the representational capacity exhibited by the original networks using fewer parameters.

7.2.1.1 Results on ResNet

The results presented in Table 7.2 demonstrate the effective compression of the ResNet18 network using both CP and Tucker decompositions. In comparison to the uncompressed baseline, the CP-based model achieves a compression ratio of 1.965x by halving the number of parameters while only reducing the accuracy by 0.0052. Furthermore, retaining only 10% of the parameters and reducing the model size from 44.695MB to 5.153MB, the model experiences a minor accuracy reduction of 0.0243. Similarly, the Tucker decomposition exhibits similar behavior, with medium-high parameter retention such as 50% and 80% outperforming CP decomposition and maintaining the baseline accuracy. On the other hand, we observe that TT decomposition achieves similar compression ratios as CP or Tucker, but is not able to maintain accuracy, dropping around 10% for all the compression ratios.

Moving on to ResNet50, although CP decomposition is typically discouraged due to unstable optimization [87], Table 7.3 reveals that it performs slightly better than Tucker de-

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	11173962	44.695	1x	0.9043
CP	0.1	1288468	5.153	8.672x	0.88
CP	0.2	2389079	9.556	4.677x	0.8888
CP	0.5	5687499	22.749	1.965x	0.8991
CP	0.8	8987725	35.951	1.243x	0.8997
Tucker	0.1	1288513	5.154	8.672x	0.8875
Tucker	0.2	2387722	9.551	4.68x	0.8871
Tucker	0.5	5690166	22.761	1.964x	0.9007
Tucker	0.8	8983194	35.932	1.244x	0.9036
TT	0.1	1288398	5.153	8.673x	0.7907
TT	0.2	2384384	9.556	4.686x	0.8067
TT	0.5	5691839	22.767	1.963x	0.7906
TT	0.8	8965400	35.861	1.246x	0.7968

Table 7.2: Results compressing the convolutional layers of a ResNet18 model, and training the tensorized model from scratch.

composition. For instance, with a parameter retention of 10%, the CP-based model achieves a model size of 37.946MB compared to the uncompressed model's size of 94.080MB, resulting in an accuracy reduction of only 0.0092. On the other hand, the best performance for the Tucker-based models is observed at a parameter retention of 80%, yielding an accuracy of 0.9018. In this case, we see that TT decomposition is able to achieve a higher compression ratio than CP and Tucker, i.e., 10.186x, with a model size of 9.235MB, while reducing the accuracy by 0.0254.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	23520842	94.080	1x	0.9106
CP	0.1	9486467	37.946	2.479x	0.9014
CP	0.2	11050056	44.200	2.129x	0.8936
CP	0.5	15732335	62.929	1.495x	0.9029
CP	0.8	20409706	81.639	1.152x	0.9038
Tucker	0.1	9488912	37.956	2.479x	0.8953
Tucker	0.2	11046237	44.185	2.129x	0.8867
Tucker	0.5	15737527	62.950	1.495x	0.8859
Tucker	0.8	20410051	81.640	1.152x	0.9018
TT	0.1	2308938	9.235	10.186x	0.8852
TT	0.2	11040926	44.163	2.13x	0.9001
TT	0.5	15735579	62.942	1.495x	0.8956
TT	0.8	20391852	81.567	1.153x	0.8921

Table 7.3: Results compressing the convolutional layers of a ResNet50 model, and training the tensorized model from scratch.

Overall, considering that the ResNet architecture already employs more efficient convolutional layers, such as bottlenecks [86], we observe that TD-based compression significantly reduces the size of the models while minimally impacting the accuracy of the uncompressed baselines.

7.2.1.2 Results on VGG

In both Table 7.4 and Table 7.5, we can observe that the VGG architecture, which solely employs standard convolutional layers, achieves higher compression ratios compared to the ResNet models. This observation is particularly evident when analyzing the CP-based models.

In the case of VGG11, the CP-based model reduces the size from 36.924MB to 3.75MB, resulting in a significant model reduction. However, this reduction in size comes at the cost of a decrease in accuracy by 0.0379. Similarly, the CP-based VGG19 model demonstrates even higher model reduction, reducing the size from 80.162MB to 8.108MB, with a compression ratio of 9.887x. However, this also leads to a reduction in accuracy by 0.0311.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	9231114	36.924	1x	0.8803
CP	0.1	937456	3.750	9.847x	0.8424
CP	0.2	1860182	7.441	4.962x	0.8565
CP	0.5	4627272	18.509	1.995x	0.8777
CP	0.8	7394881	29.580	1.248x	0.8747
Tucker	0.1	938922	3.756	9.832x	0.8437
Tucker	0.2	1858301	7.433	4.968x	0.8556
Tucker	0.5	4631515	18.526	1.993x	0.8691
Tucker	0.8	7395593	29.582	1.248x	0.8671
TT	0.1	938326	3.753	9.838x	0.5102
TT	0.2	1856280	7.425	4.973x	0.6694
TT	0.5	4630781	18.523	1.993x	0.6916
TT	0.8	7376404	29.50	1.251x	0.6757

Table 7.4: Results compressing the convolutional layers of a VGG11 model, and training the tensorized model from scratch.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	20040522	80.162	1x	0.8937
CP	0.1	2026948	8.108	9.887x	0.8626
CP	0.2	4031038	16.124	4.972x	0.8687
CP	0.5	10040663	40.163	1.996x	0.8812
CP	0.8	16051973	64.208	1.248x	0.892
Tucker	0.1	2029861	8.119	9.873x	0.8625
Tucker	0.2	4028238	16.113	4.975x	0.8594
Tucker	0.5	10052199	40.209	1.994x	0.8834
Tucker	0.8	16056695	64.227	1.248x	0.8784
TT	0.1	2029550	8.118	9.874x	0.5542
TT	0.2	4022440	16.089	4.982x	0.3919
TT	0.5	10052593	40.210	1.994x	0.5006
TT	0.8	16015504	64.062	1.251x	0.5781

Table 7.5: Results compressing the convolutional layers of a VGG19 model, and training the tensorized model from scratch.

It is worth noting that compared to the ResNet models, VGG models experience a more considerable decrease in accuracy when using a small parameter retention setting. This

suggests that the impact on accuracy is more pronounced in VGG models when employing aggressive compression techniques.

On the other hand, Tucker decomposition generally achieves worse results overall compared to the CP-based models. It fails to reach the accuracy obtained by the CP-based models, even when considering different parameter retention settings. This gets even worse for the case of TT decomposition, which obtains a severe reduction in the accuracy, and making the TT-based models unusable in practice.

In summary, the VGG architecture demonstrates higher compression ratios at the same accuracy, especially with CP-based models. However, the reduction in accuracy is more substantial in VGG models when using lower parameter retention settings. Additionally, Tucker decomposition does not achieve the same accuracy levels as the CP-based models across all parameter retention settings. Finally, we see that TT-based models are affected by a drastic reduction in the accuracy for all the compression ratios.

7.2.2 Tensorization from pretrained

In this tensorization strategy applied to pretrained models, we commence by choosing a trained baseline network, such as a pretrained ResNet18 model, as our initial framework. Afterward, we apply tensorization techniques to the baseline model at different compression ratios and proceed to fine-tune these tensorized models to recover the accuracy lose due to the compression. The purpose of this experiment is to evaluate whether the tensorized networks can approximate accurately the original trained networks with fewer parameters.

7.2.2.1 Results on ResNet

The results presented in Table 7.6 demonstrate that tensorization of a pretrained ResNet18 model yields satisfactory outcomes when utilizing CP and Tucker decompositions, particularly when the parameter retention exceeds 50%. However, as the compression ratio increases and the parameter retention decreases, there is a significant decline in accuracy. The optimal trade-off between compression and accuracy is achieved with the CP decomposition and a parameter retention of 50%, which reduces the model size from 44.695 MB to 22.749 MB, with a marginal accuracy drop of 0.0026. In contrast, the Tucker decomposition, at the same parameter retention, achieves an accuracy of 0.8915, which is 0.0128 lower than the original. Notably, the TT decomposition fails to accurately approximate the original pretrained model for any compression ratio.

In the case of the pretrained ResNet50 model, the results shown in Table 7.7 reveal that all decompositions struggle to achieve the original accuracy when using a parameter retention of 10%. The CP decomposition achieves an accuracy of 0.4648, the Tucker decomposition reaches 0.2015, and the TT decomposition falls to 0.1285. With a parameter retention of 20% and a reduction in model size from 94.080 MB to 44.200 MB, only the CP decomposition successfully approximates the original network, achieving an accuracy of 0.8627. The Tucker decomposition begins to perform reasonably well with parameter retentions higher than 50%, but it does not match the performance of the CP decomposition for the same level of compression. Similar to ResNet18, the TT decomposition fails to effectively compress the pretrained network and experiences a significant drop in accuracy.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	11173962	44.695	1x	0.9043
CP	0.1	1288468	5.153	8.672x	0.8066
CP	0.2	2389079	9.556	4.677x	0.8807
CP	0.5	5687499	22.749	1.965x	0.9017
CP	0.8	8987725	35.951	1.243x	0.904
Tucker	0.1	1288513	5.154	8.672x	0.7742
Tucker	0.2	2387722	9.551	4.68x	0.8436
Tucker	0.5	5690166	22.761	1.964x	0.8915
Tucker	0.8	8983194	35.932	1.244x	0.8992
TT	0.1	723529	2.894	15.444x	0.1002
TT	0.2	1252568	5.010	8.921x	0.1776
TT	0.5	1592209	6.369	7.018x	0.1565
TT	0.8	1728997	6.916	6.463x	0.2874

Table 7.6: Results tensorizing the convolutional layers of a ResNet18 model from a pretrained model and without fine-tuning.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	23520842	94.080	1x	0.9106
CP	0.1	9486467	37.946	2.479x	0.4648
CP	0.2	11050056	44.200	2.129x	0.8627
CP	0.5	15732335	62.929	1.495x	0.9055
CP	0.8	20409706	81.639	1.152x	0.9094
Tucker	0.1	9488912	37.956	2.479x	0.2015
Tucker	0.2	11046237	44.185	2.129x	0.2853
Tucker	0.5	15737527	62.950	1.495x	0.8586
Tucker	0.8	20410051	81.640	1.152x	0.8972
TT	0.1	8699744	34.799	2.704x	0.1285
TT	0.2	9453069	37.812	2.488x	0.1208
TT	0.5	10627659	42.511	2.213x	0.1018
TT	0.8	11643420	46.574	2.020x	0.1009

Table 7.7: Results tensorizing the convolutional layers of a ResNet50 model from a pretrained model and without fine-tuning.

7.2.2.2 Results on VGG

The results presented in Table 7.8 indicate that for the pretrained VGG11 network, both CP and Tucker decompositions result in a drop in accuracy of approximately 0.10-0.15 when using a parameter retention of 10%. However, these decompositions successfully compress the original pretrained network when using a parameter retention higher than 20%, and in some cases, even surpass the original accuracy. For instance, the CP decomposition improves the accuracy slightly from 0.8803 to 0.8829 while achieving a compression ratio of 1.995x and reducing the model size from 36.924 MB to 18.509 MB. Tucker decomposition also matches the original accuracy but requires a parameter retention of 80%. Similar to the previous findings, the TT decomposition fails to accurately compress the original pretrained network.

A similar behavior is observed for the pretrained VGG19 network, as shown in Table 7.9.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	9231114	36.924	1x	0.8803
CP	0.1	937456	3.750	9.847x	0.7367
CP	0.2	1860182	7.441	4.962x	0.8562
CP	0.5	4627272	18.509	1.995x	0.8829
CP	0.8	7394881	29.580	1.248x	0.8855
Tucker	0.1	938922	3.756	9.832x	0.704
Tucker	0.2	1858301	7.433	4.968x	0.8419
Tucker	0.5	4631515	18.526	1.993x	0.8778
Tucker	0.8	7395593	29.582	1.248x	0.8822
TT	0.1	452943	1.812	20.380x	0.1
TT	0.2	883615	3.534	10.447x	0.1
TT	0.5	1137418	4.550	8.116x	0.1123
TT	0.8	1231198	4.925	7.498x	0.23

Table 7.8: Results tensorizing the convolutional layers of a VGG11 model from a pretrained model and without fine-tuning.

Similar to the case of ResNet50, due to their larger size, all decompositions suffer when using smaller parameter retentions such as 10%. Tucker decomposition performs well with parameter retentions above 50%, reaching an accuracy of 0.8906 with a parameter retention of 80% and a compression ratio of $1.248\times$. CP decomposition slightly exceeds the original accuracy, achieving an accuracy of 0.8942 with a compression ratio of $1.248\times$ and reducing the model size from 80.162 MB to 64.208 MB. The same behavior is observed for TT decomposition, as it does not achieve a good trade-off between compression and accuracy.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	20040522	80.162	1x	0.8937
CP	0.1	2026948	8.108	9.887x	0.3189
CP	0.2	4031038	16.124	4.972x	0.8617
CP	0.5	10040663	40.163	1.996x	0.8912
CP	0.8	16051973	64.208	1.248x	0.8942
Tucker	0.1	2029861	8.119	9.873x	0.6849
Tucker	0.2	4028238	16.113	4.975x	0.7666
Tucker	0.5	10052199	40.209	1.994x	0.8774
Tucker	0.8	16056695	64.227	1.248x	0.8906
TT	0.1	1027433	4.110	19.505x	0.1
TT	0.2	2013630	8.055	9.952x	0.1
TT	0.5	2564808	10.259	7.814x	0.207
TT	0.8	2767236	11.069	7.242x	0.3525

Table 7.9: Results tensorizing the convolutional layers of a VGG19 model from a pretrained model and without fine-tuning.

7.2.3 Fine-tune tensorized pretrained models

Seeing that some of the results obtained from tensorizing pretrained models are not as good as expected, we have tried to recover the accuracy drop with a fine-tuning process. This

process is done trying to make it as fair as possible, setting a shared fine-tune framework for all the models. More precisely, we set a fixed learning rate of $1e{-}5$, and we use a training technique called *early stopping*, where we monitor the validation loss metric, and we stop the fine-tuning process once this metric stop improving. The number of checks with no improvement after which training will be stopped is set to 3, and the maximum number of epochs is set to 50.

7.2.3.1 Results on ResNet

After fine-tuning, we observe significant improvements in the initial results for ResNet18 (Table 7.6). Starting from a parameter retention of 10% and a model size of 5.153MB, the accuracy has been increased from 0.8066 to 0.897, which is very close to the original accuracy of 0.9043. With higher parameter retentions, both Tucker and CP decompositions are capable of surpassing the original accuracy while utilizing fewer parameters. In the case of TT decomposition, higher compression ratios can be achieved, but it results in a notable drop in accuracy.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	11173962	44.695	1x	0.9043
CP	0.1	1288468	5.153	8.672x	0.897
CP	0.2	2389079	9.556	4.677x	0.9203
CP	0.5	5687499	22.749	1.965x	0.9189
CP	0.8	8987725	35.951	1.243x	0.9148
Tucker	0.1	1288513	5.154	8.672x	0.894
Tucker	0.2	2387722	9.551	4.68x	0.9138
Tucker	0.5	5690166	22.761	1.964x	0.9237
Tucker	0.8	8983194	35.932	1.244x	0.9283
TT	0.1	723529	2.894	15.444x	0.5416
TT	0.2	1252568	5.010	8.921x	0.7227
TT	0.5	1592209	6.369	7.018x	0.8328
TT	0.8	1728997	6.916	6.463x	0.8675

Table 7.10: Results after fine-tuning a tensorized pretrained ResNet18 model.

For the case of ResNet50, as shown in Table 7.11, we observe a similar behavior to ResNet18, where we can recover the original accuracy and even slightly surpass it. Specifically, with parameter retentions higher than 20%, both CP and Tucker decompositions achieve the same or better performance than the original pretrained network while utilizing fewer parameters. For example, CP decomposition is able to obtain an accuracy of 0.9237, which is 0.0194 higher than the orignal, while reducing the model size from 94.080 MB to 62.929 MB. On the other hand, although TT decomposition achieves higher compression ratios, it does not yield to as good results as CP or Tucker decompositions.

7.2.3.2 Results on VGG

The initial results of tensorizing a pretrained VGG11 network are quite promising. However, when using small parameter retentions, tensor decompositions struggle to recover the original accuracy of 0.8803. After fine-tuning the tensorized network, significant improvements are observed. This is demonstrated in Table 7.12, where it can be seen that both Tucker

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	23520842	94.080	1x	0.9106
CP	0.1	9486467	37.946	2.479x	0.901
CP	0.2	11050056	44.200	2.129x	0.9176
CP	0.5	15732335	62.929	1.495x	0.9237
CP	0.8	20409706	81.639	1.152x	0.9241
Tucker	0.1	9488912	37.956	2.479x	0.8788
Tucker	0.2	11046237	44.185	2.129x	0.9
Tucker	0.5	15737527	62.950	1.495x	0.915
Tucker	0.8	20410051	81.640	1.152x	0.9226
TT	0.1	8699744	34.799	2.704x	0.8453
TT	0.2	9453069	37.812	2.488x	0.8537
TT	0.5	10627659	42.511	2.213x	0.8525
TT	0.8	11643420	46.574	2.020x	0.8657

Table 7.11: Results after fine-tuning a tensorized pretrained ResNet50 model.

and CP decompositions either match or even surpass the original accuracy. For instance, Tucker decomposition achieves an accuracy of 0.8971 while reducing the model size by half from 36.924 MB to 18.509 MB. On the other hand, although TT decomposition achieves higher compression ratios, it is unable to achieve the same level of performance as CP and Tucker decompositions.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	9231114	36.924	1x	0.8803
CP	0.1	937456	3.750	9.847x	0.865
CP	0.2	1860182	7.441	4.962x	0.8816
CP	0.5	4627272	18.509	1.995x	0.8925
CP	0.8	7394881	29.580	1.248x	0.8919
Tucker	0.1	938922	3.756	9.832x	0.8587
Tucker	0.2	1858301	7.433	4.968x	0.8851
Tucker	0.5	4631515	18.526	1.993x	0.8971
Tucker	0.8	7395593	29.582	1.248x	0.8969
TT	0.1	452943	1.812	20.380x	0.3675
TT	0.2	883615	3.534	10.447x	0.5445
TT	0.5	1137418	4.550	8.116x	0.7679
TT	0.8	1231198	4.925	7.498x	0.8032

Table 7.12: Results after fine-tuning a tensorized pretrained VGG11 model.

The initial results of tensorizing a pretrained VGG19 network are not as satisfactory (Table 7.9), as all tensor decompositions struggle to recover the original accuracy when using low parameter retentions. However, as shown in Table 7.13, fine-tuning the tensorized VGG19 network leads to significant improvements, particularly for CP and Tucker decompositions, which are able to match or even surpass the original accuracy of 0.8937 with relatively small parameter retentions. For instance, CP decomposition achieves an accuracy of 0.9007 while utilizing approximately 20% of the parameters and reducing the model size from 80.162 MB to 16.124 MB. Despite some improvements, the results obtained by TT decomposition are still far from those achieved by Tucker or CP decompositions.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	20040522	80.162	1x	0.8937
CP	0.1	2026948	8.108	9.887x	0.8627
CP	0.2	4031038	16.124	4.972x	0.9007
CP	0.5	10040663	40.163	1.996x	0.9062
CP	0.8	16051973	64.208	1.248x	0.8965
Tucker	0.1	2029861	8.119	9.873x	0.8642
Tucker	0.2	4028238	16.113	4.975x	0.8864
Tucker	0.5	10052199	40.209	1.994x	0.9097
Tucker	0.8	16056695	64.227	1.248x	0.9139
TT	0.1	2029550	8.118	9.874x	0.317
TT	0.2	4022440	16.089	4.982x	0.5517
TT	0.5	10052593	40.210	1.994x	0.7641
TT	0.8	16015504	64.062	1.251x	0.8175

Table 7.13: Results after fine-tuning a tensorized pretrained VGG19 model.

CHAPTER 8

Conclusions

This study investigates the effectiveness of tensor decompositions (TDs) as a method for compressing convolutional layers in VGG and ResNet architectures. The results demonstrate that TDs offer a promising approach for neural network compression by significantly reducing model size while having minimal impact on accuracy.

The research explores three widely used TD methods: CP, Tucker, and TT decompositions. By decomposing the weight tensors of convolutional layers into smaller rank tensors, a substantial reduction in model size is achieved without sacrificing accuracy.

Experimental findings reveal that TDs can effectively capture important features and patterns in convolutional layers, enabling compressed models to retain their discriminative power. Despite the reduction in accuracy, the compressed models still achieve comparable performance to their original counterparts.

The reduction in model size through TDs is particularly noteworthy, as it decreases the number of parameters and memory footprint. This has significant implications for resource-constrained environments like mobile devices or edge computing scenarios, where storage and memory limitations often pose challenges.

TDs offer a compelling approach to neural network compression by achieving substantial reductions in model size while preserving accuracy. Compressed models not only require less storage space, but also consume fewer computational resources during inference. This leads to improved efficiency and faster execution times, which are highly desirable in real-world applications.

In conclusion, this study demonstrates the effectiveness of CP, Tucker, and TT decompositions for compressing convolutional layers in VGG and ResNet architectures. These findings contribute to the growing body of research on efficient and compact deep learning models, facilitating practical and resource-efficient deployments of neural networks across various domains.

CHAPTER 9

Future work

In this chapter, we outline potential areas of future work that can advance the field of tensor decompositions and their application to neural networks compression.

One area of focus is the development of customized and optimized kernels for tensor decompositions. This research involves investigating and developing efficient algorithms and computational techniques tailored to tensor decomposition, i.e., designing customized kernels specific to the target hardware architecture. In this way, in [88], the authors proposed a code generation framework that creates efficient kernels for inference. However, this work only considers Tucker-based models, so its extension to other types of decompositions may be a good line of research. Additionally, parallel computing techniques could be explored to accelerate the execution of tensor decomposition algorithms while considering energy efficiency.

Another promising direction is the deployment of TD-based CNNs in energy-efficient platforms with a focus on minimizing their carbon footprint. By integrating tensor decompositions with convolutional neural networks, we can reduce the number of parameters and improve computational efficiency. This research could explore the feasibility of deploying TD-based CNNs on platforms like RISC-V or FPGA, which offer energy-efficient computing capabilities. An important aspect of this work should be measuring and analyzing the energy consumption and carbon footprint of TD-based CNNs at inference time, comparing their efficiency to traditional CNN architectures.

Furthermore, we propose investigating the combination of tensor decompositions with quantization techniques. This research seeks to achieve further parameter reduction and computational efficiency. Methods will be developed to integrate tensor decompositions and quantization in a synergistic manner, maximizing the benefits of both approaches while minimizing the overall carbon footprint.

Although this work primarily focuses on utilizing tensor decompositions to compress convolutional layers in CNNs, it is worth noting that the application of tensor decompositions is not restricted to this specific type of neural network. The potential of employing tensor decompositions in neural networks, such as transformers that predominantly consist of fully-connected layers and encompass a significant number of parameters, can be explored. Furthermore, it is important to recognize that even recurrent neural net-

works or long short-term memory networks can be compressed effectively using tensor decompositions.

Another promising avenue is to employ more sophisticated tensor decompositions, such as advanced tensor networks like MERA or PEPS, which are extensively utilized in quantum physics [89], for tensorizing various neural network architectures. These complex tensor decompositions possess a greater capacity to capture intricate relationships within the data. Hence, we can leverage this enhanced capacity to effectively compress neural networks.

In conclusion, the future work outlined above presents exciting opportunities for advancing the field of tensor decompositions. By developing customized kernels, deploying TD-based CNNs on energy-efficient platforms, and exploring the combination with quantization, researchers can contribute to the development of more efficient and resource-friendly algorithms with reduced environmental impact.

APPENDIX A

Appendix

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	21282122	85.128	1x	0.912
CP	0.1	2307890	9.232	9.221x	0.8869
CP	0.2	4421281	17.685	4.814x	0.8868
CP	0.5	10751273	43.005	1.979x	0.8994
CP	0.8	17088005	68.352	1.245x	0.9002
Tucker	0.1	2304279	9.217	9.236x	0.8899
Tucker	0.2	4423928	17.696	4.811x	0.8801
Tucker	0.5	10756892	43.028	1.978x	0.8946
Tucker	0.8	17068610	68.274	1.247x	0.8961
TT	0.1	1270025	5.08	16.757x	0.8321
TT	0.2	2329272	9.317	9.137x	0.8401
TT	0.5	3009193	12.037	7.072x	0.8273
TT	0.8	3293677	13.175	6.462x	0.8149

Table A.1: Results compressing the convolutional layers of a ResNet34 model, and training the tensorized model from scratch.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	21282122	85.128	1x	0.912
CP	0.1	2307890	9.232	9.221x	0.5712
CP	0.2	4421281	17.685	4.814x	0.8951
CP	0.5	10751273	43.005	1.979x	0.9125
CP	0.8	17088005	68.352	1.245x	0.9107
Tucker	0.1	2304279	9.217	9.236x	0.633
Tucker	0.2	4423928	17.696	4.811x	0.8576
Tucker	0.5	10756892	43.028	1.978x	0.9026
Tucker	0.8	17068610	68.274	1.247x	0.9103
TT	0.1	1270025	5.08	16.757x	0.1
TT	0.2	2329272	9.317	9.137x	0.1
TT	0.5	3009193	12.037	7.072x	0.2295
TT	0.8	3293677	13.175	6.462x	0.4592

Table A.2: Results tensorizing the convolutional layers of a ResNet34 model from a pretrained model and without fine-tuning.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	21282122	85.128	1x	0.912
CP	0.1	2307890	9.232	9.221x	0.906
CP	0.2	4421281	17.685	4.814x	0.919
CP	0.5	10751273	43.005	1.979x	0.9233
CP	0.8	17088005	68.352	1.245x	0.9224
Tucker	0.1	2304279	9.217	9.236x	0.8968
Tucker	0.2	4423928	17.696	4.811x	0.9178
Tucker	0.5	10756892	43.028	1.978x	0.9243
Tucker	0.8	17068610	68.274	1.247x	0.9277
TT	0.1	1270025	5.08	16.757x	0.5721
TT	0.2	2329272	9.317	9.137x	0.7427
TT	0.5	3009193	12.037	7.072x	0.8349
TT	0.8	3293677	13.175	6.462x	0.8796

Table A.3: Results after fine-tuning a tensorized pretrained ResNet34 model.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	14728266	58.913	1x	0.9006
CP	0.1	1491744	5.967	9.873x	0.8528
CP	0.2	2964470	11.858	4.968x	0.8735
CP	0.5	7380522	29.522	1.996x	0.8923
CP	0.8	11797740	47.191	1.248x	0.8897
Tucker	0.1	1493972	5.976	9.858x	0.8593
Tucker	0.2	2962034	11.848	4.972x	0.8758
Tucker	0.5	7388040	29.552	1.994x	0.88
Tucker	0.8	11800251	47.201	1.248x	0.8798
TT	0.1	745616	2.982	19.753x	0.5243
TT	0.2	1459699	5.839	10.09x	0.5014
TT	0.5	1868502	7.474	7.882x	0.5992
TT	0.8	2019774	8.079	7.292x	0.5905

Table A.4: Results compressing the convolutional layers of a VGG16 model, and training the tensorized model from scratch.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	14728266	58.913	1x	0.9006
CP	0.1	1491744	5.967	9.873x	0.5928
CP	0.2	2964470	11.858	4.968x	0.8727
CP	0.5	7380522	29.522	1.996x	0.9017
CP	0.8	11797740	47.191	1.248x	0.9005
Tucker	0.1	1493972	5.976	9.858x	0.6584
Tucker	0.2	2962034	11.848	4.972x	0.7988
Tucker	0.5	7388040	29.552	1.994x	0.8796
Tucker	0.8	11800251	47.201	1.248x	0.8906
TT	0.1	745616	2.982	19.753x	0.1
TT	0.2	1459699	5.839	10.09x	0.1274
TT	0.5	1868502	7.474	7.882x	0.1185
TT	0.8	2019774	8.079	7.292x	0.3186

Table A.5: Results tensorizing the convolutional layers of a VGG16 model from a pretrained model and without fine-tuning.

Method	% Parameters	#Parameters	Model size (MB)	Compression ratio	Accuracy
Uncompressed	1.0	14728266	58.913	1x	0.9006
CP	0.1	1491744	5.967	9.873x	0.8707
CP	0.2	2964470	11.858	4.968x	0.9029
CP	0.5	7380522	29.522	1.996x	0.9107
CP	0.8	11797740	47.191	1.248x	0.8937
Tucker	0.1	1493972	5.976	9.858x	0.8681
Tucker	0.2	2962034	11.848	4.972x	0.893
Tucker	0.5	7388040	29.552	1.994x	0.9102
Tucker	0.8	11800251	47.201	1.248x	0.9129
TT	0.1	745616	2.982	19.753x	0.3455
TT	0.2	1459699	5.839	10.09x	0.5315
TT	0.5	1868502	7.474	7.882x	0.7782
TT	0.8	2019774	8.079	7.292x	0.8338

Table A.6: Results after fine-tuning a tensorized pretrained VGG16 model.

Bibliography

[1] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021. See pages v, 15.

[2] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A ConvNet for the 2020s. Technical report. See page 1.

[3] Zhong-Qiu Zhao, Peng Zheng, Shou-Tao Xu, and Xindong Wu. Object Detection With Deep Learning: A Review. *IEEE Transactions on Neural Networks and Learning Systems*, 30(11):3212–3232, 2019. See page 1.

[4] Yujian Mo, Yan Wu, Xinneng Yang, Feilin Liu, and Yujun Liao. Review the state-of-the-art technologies of semantic segmentation based on deep learning. *Neurocomputing*, 493:626–646, 2022. See page 1.

[5] Qi Dang, Jianqin Yin, Bin Wang, and Wenqing Zheng. Deep learning based 2D human pose estimation: A survey. *Tsinghua Science and Technology*, 24(6):663–676, 2019. See page 1.

[6] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A Convergence Theory for Deep Learning via Over-Parameterization. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 242–252. PMLR, 2 2019. See page 1.

[7] Brian R Bartoldson, Bhavya Kailkhura, and Davis Blalock. Compute-Efficient Deep Learning: Algorithmic Trends and Opportunities. *Journal of Machine Learning Research*, 24:1–77, 2023. See page 1.

[8] Román Orús. Tensor networks for complex quantum systems. *Nature Reviews Physics*, 1(9):538–550, 2019. See pages 1, 17.

[9] Nicholas D Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E Papalexakis, and Christos Faloutsos. Tensor Decomposition for Signal Processing and Machine Learning. *IEEE Transactions on Signal Processing*, 65(13):3551–3582, 2017. See page 1.

[10] Tamara G Kolda and Brett W Bader. Tensor Decompositions and Applications. *SIAM Review*, 51(3):455–500, 8 2009. See pages 1, 3, 11, 13, 18, and 20.

[11] Fengyu Cong, Qiu-Hua Lin, Li-Dan Kuang, Xiaofeng Gong, Piia Astikainen, and Tapani Ristaniemi. Tensor decomposition of EEG signals: A brief review. *Journal of Neuroscience Methods*, 248:59–69, 2015. See page 1.

[12] Kijung Shin and U Kang. Distributed Methods for High-Dimensional and Large-Scale Tensor Factorization. *2014 IEEE International Conference on Data Mining*, pages 989–994, 2014. See page 1.

[13] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing Neural Networks. *arXiv:1509.06569 [cs]*, 12 2015. See pages 2, 4, and 23.

[14] Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966. See pages 2, 3, 18, and 19.

[15] I V Oseledets. Tensor-Train Decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 1 2011. See pages 2, 3, 18, and 21.

[16] J Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart-Young" decomposition. *Psychometrika*, 35(3):283–319, 1970. See pages 2, 3, and 18.

[17] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor Ring Decomposition. *ArXiv*, abs/1606.05535, 2016. See page 2.

[18] Daniel Kressner and Christine Tobler. Algorithm 941: Htucker—A Matlab Toolbox for Tensors in Hierarchical Tucker Format. *ACM Trans. Math. Softw.*, 40(3), 4 2014. See pages 2, 3.

[19] Lieven De Lathauwer. Decompositions of a Higher-Order Tensor in Block Terms—Part I: Lemmas for Partitioned Matrices. *SIAM Journal on Matrix Analysis and Applications*, 30(3):1022–1032, 2008. See pages 2, 20.

[20] Kaiming He, X Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2015. See page 2.

[21] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. See pages 2, 32.

[22] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *arXiv: Computer Vision and Pattern Recognition*, 2015. See page 3.

[23] M Xia, Zexuan Zhong, and Danqi Chen. Structured Pruning Learns Compact and Accurate Models. In *Annual Meeting of the Association for Computational Linguistics*, 2022. See page 3.

[24] Trevor Gale, Erich Elsen, and Sara Hooker. The State of Sparsity in Deep Neural Networks. *ArXiv*, abs/1902.09574, 2019. See page 3.

[25] Jianping Gou, B Yu, Stephen J Maybank, and Dacheng Tao. Knowledge Distillation: A Survey. *International Journal of Computer Vision*, 129:1789 – 1819, 2020. See page 3.

[26] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, I Oseledets, and Victor S Lempitsky. Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition. *CoRR*, abs/1412.6553, 2014. See pages 4, 18, and 25.

[27] Timur Garipov, Dmitry Podoprikhin, Alexander Novikov, and Dmitry Vetrov. Ultimate tensorization: compressing convolutional and FC layers alike, 11 2016. See pages 4, 27.

[28] Richik Sengupta, Soumik Adhikary, Ivan Oseledets, and Jacob Biamonte. Tensor networks in machine learning. 7 2022. See page 4.

[29] Dingheng Wang, Guangshe Zhao, Guoqi Li, Lei Deng, and Yang Wu. Compressing 3DCNNs Based on Tensor Train Decomposition. *Neural Networks*, 131:215–230, 11 2020. See page 4.

[30] Xiaolong Wu. TCNN: a Tensor Convolutional Neuro-Network for big data anomaly detection. page 7. See page 4.

[31] Jean Kossaifi, Antoine Toisoul, Adrian Bulat, Yannis Panagakis, Timothy Hospedales, and Maja Pantic. Factorized Higher-Order CNNs with an Application to Spatio-Temporal Emotion Estimation, 3 2020. See page 4.

[32] Yinan Wang, Weihong "Grace" Guo, and Xiaowei Yue. Tensor decomposition to Compress Convolutional Layers in Deep Learning. *IISE Transactions*, pages 1–60, 4 2021. See page 4.

[33] Ye Liu and Michael K Ng. Deep neural network compression by Tucker decomposition with nonlinear response. *Knowledge-Based Systems*, 241:108171, 4 2022. See page 4.

[34] Bijiao Wu, Dingheng Wang, Guangshe Zhao, Lei Deng, and Guoqi Li. Hybrid tensor decomposition in neural network compression. *Neural Networks*, 132:309–320, 12 2020. See page 4.

[35] Jean Kossaifi, Adrian Bulat, Georgios Tzimiropoulos, and Maja Pantic. T-Net: Parametrizing Fully Convolutional Nets with a Single High-Order Tensor, 4 2019. See page 4.

[36] Yanwei Zheng, Yang Zhou, Zengrui Zhao, and Dongxiao Yu. Adaptive Tensor-Train Decomposition for Neural Network Compression. In Yong Zhang, Yicheng Xu, and Hui Tian, editors, *Parallel and Distributed Computing, Applications and Technologies*, Lecture Notes in Computer Science, pages 70–81, Cham, 2021. Springer International Publishing. See page 4.

[37] Julia Gusak, Maksym Kholiavchenko, Evgeny Ponomarev, Larisa Markeeva, Ivan Oseledets, and Andrzej Cichocki. MUSCO: Multi-Stage Compression of neural networks, 11 2019. See pages 4, 30.

[38] Lucas Liebenwein, Alaa Maalouf, Oren Gal, Dan Feldman, and Daniela Rus. Compressing Neural Networks: Towards Determining the Optimal Layer-wise Decomposition. See page 4.

[39] Yu Pan, Zeyong Su, Ao Liu, Jingquan Wang, Nannan Li, and Zenglin Xu. A Unified Weight Initialization Paradigm for Tensorial Convolutional Neural Networks, 7 2022. See page 4.

[40] Miao Yin, Siyu Liao, Xiao-Yang Liu, Xiaodong Wang, and Bo Yuan. Towards Extremely Compact RNNs for Video Recognition with Fully Decomposed Hierarchical Tucker Structure. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12080–12089, 2021. See page 4.

[41] Yu Gong, Miao Yin, Lingyi Huang, Chunhua Deng, Yang Sui, and Bo Yuan. Algorithm and Hardware Co-Design of Energy-Efficient LSTM Networks for Video Recognition with Hierarchical Tucker Tensor Decomposition, 12 2022. See page 4.

[42] Xindian Ma, Peng Zhang, Shuai Zhang, Nan Duan, Yuexian Hou, Ming Zhou, and Dawei Song. A Tensorized Transformer for Language Modeling. See page 4.

[43] Sunzhu Li, P Zhang, Guobing Gan, Xiuqing Lv, Benyou Wang, Junqiu Wei, and Xin Jiang. Hypoformer: Hybrid Decomposition Transformer for Edge-friendly Neural Machine Translation. In *Conference on Empirical Methods in Natural Language Processing*, 2022. See page 4.

[44] Jiaqi Gu, Ben Keller, Jean Kossaifi, Anima Anandkumar, Brucek Khailany, and David Z Pan. HEAT: Hardware-Efficient Automatic Tensor Decomposition for Transformer Compression, 11 2022. See pages 4, 30.

[45] Peiyu Liu, Ze-Feng Gao, Wayne Xin Zhao, Z Y Xie, Zhong-Yi Lu, and Ji-rong Wen. Enabling Lightweight Fine-tuning for Pre-trained Language Model Compression based on Matrix Product Operators. In *Annual Meeting of the Association for Computational Linguistics*, 2021. See page 4.

[46] 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. See page 5.

[47] Yoav Levine, Noam Wies, Or Sharir, Nadav Cohen, and Amnon Shashua. Chapter 7 - Tensors for deep learning theory: Analyzing deep learning architectures via tensorization. In Yipeng Liu, editor, *Tensors for Data Processing*, pages 215–248. Academic Press, 2022. See pages 5, 17.

[48] Nadav Cohen, Or Sharir, Yoav Levine, Ronen Tamari, David Yakira, and Amnon Shashua. Analysis and Design of Convolutional Networks via Hierarchical Tensor Decompositions, 6 2018. See page 5.

[49] Namgil Lee and Andrzej Cichocki. Fundamental tensor operations for large-scale data analysis using tensor network formats. *Multidimensional Systems and Signal Processing*, 29:921–960, 2018. See page 5.

[50] Zhun Liu, Ying Shen, Varun Bharadhwaj Lakshminarasimhan, Paul Pu Liang, Amir Zadeh, and Louis-Philippe Morency. Efficient Low-rank Multimodal Fusion With Modality-Specific Factors. *ArXiv*, abs/1806.00064, 2018. See page 5.

[51] [1706.00439] Tensor Contraction Layers for Parsimonious Deep Nets. See page 5.

[52] Jean Kossaifi, Zachary Chase Lipton, Aran Khanna, Tommaso Furlanello, and Anima Anandkumar. Tensor Regression Networks. *ArXiv*, abs/1707.08308, 2017. See page 5.

[53] Arinbjörn Kolbeinsson, Jean Kossaifi, Yannis Panagakis, Adrian Bulat, Anima Anandkumar, Ioanna Tzoulaki, and Paul Matthews. Tensor Dropout for Robust Learning, 12 2020. See page 5.

[54] Maolin Wang, Yu Pan, Xiangli Yang, Guangxi Li, and Zenglin Xu. Tensor Networks Meet Neural Networks: A Survey. 1 2023. See page 5.

[55] Cesar F Caiafa and Andrzej Cichocki. Generalizing the column–row matrix decomposition to multi-way arrays. *Linear Algebra and its Applications*, 433(3):557–573, 2010. See page 9.

[56] Tamara Gibson Kolda. Multilinear operators for higher-order decompositions. Technical report, Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA ..., 2006. See page 10.

[57] Sheldon Axler. *Linear algebra done right*. Springer Science & Business Media, 1997. See page 13.

[58] Frank L Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, 6(1-4):164–189, 1927. See page 13.

[59] Johan Håstad. Tensor rank is np-complete. *J. Algorithms*, 11:644–654, 1989. See pages 13, 29.

[60] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. See page 15.

[61] Jacob Biamonte and Ville Bergholm. Tensor networks in a nutshell. *arXiv preprint arXiv:1708.00006*, 2017. See page 17.

[62] Andrzej Cichocki, Anh-Huy Phan, Qibin Zhao, Namgil Lee, Ivan Oseledets, Masashi Sugiyama, Danilo P Mandic, and others. Tensor networks for dimensionality reduction and large-scale optimization: Part 2 applications and future perspectives. *Foundations and Trends® in Machine Learning*, 9(6):431–673, 2017. See pages 17, 24.

[63] Roger Penrose. Applications of negative dimensional tensors. *Combinatorial mathematics and its applications*, 1:221–244, 1971. See page 17.

[64] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, 2021. See page 18.

[65] David Perez-Garcia, Frank Verstraete, Michael M Wolf, and J Ignacio Cirac. Matrix product state representations. *arXiv preprint quant-ph/0608197*, 2006. See pages 18, 21.

[66] Shinichi Nakajima, Masashi Sugiyama, S Derin Babacan, and Ryota Tomioka. Global analytic solution of fully-observed variational bayesian matrix factorization. *The Journal of Machine Learning Research*, 14(1):1–37, 2013. See page 18.

[67] Zhiyu Cheng, Baopu Li, Yanwen Fan, and Yingze Bao. A novel rank selection scheme in tensor ring decomposition based on reinforcement learning for deep neural networks. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3292–3296, 2020. See pages 18, 30.

[68] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011. See page 23.

[69] Mark Sandler, Andrew G Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. See page 24.

[70] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016. See page 24.

[71] 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, 2 2016. See pages 26, 30.

[72] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML workshop*, volume 238, 2015. See page 28.

[73] Xiao-Yang Liu, Yiming Fang, Liuqing Yang, Zechu Li, and Anwar Walid. Chapter 9 - High-performance tensor decompositions for compressing and accelerating deep neural networks. In Yipeng Liu, editor, *Tensors for Data Processing*, pages 293–340. Academic Press, 1 2022. See page 29.

[74] Jiahao Su, Jingling Li, Bobby Bhattacharjee, and Furong Huang. Tensorial neural networks: Generalization of neural networks and application to model compression. *arXiv preprint arXiv:1805.10352*, 2018. See page 29.

[75] Christopher J Hillar and Lek-Heng Lim. Most tensor problems are np-hard. *Journal of the ACM (JACM)*, 60(6):1–39, 2013. See page 29.

[76] Shinichi Nakajima, Ryota Tomioka, Masashi Sugiyama, and S Babacan. Perfect dimensionality recovery by variational bayesian pca. *Advances in neural information processing systems*, 25, 2012. See page 30.

[77] Jean Kossaifi, Antoine Toisoul, Adrian Bulat, Yannis Panagakis, Timothy M Hospedales, and Maja Pantic. Factorized higher-order cnns with an application to spatio-temporal emotion estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6060–6069, 2020. See page 30.

[78] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. See page 31.

[79] William Falcon and The PyTorch Lightning team. PyTorch Lightning, March 2019. See page 31.

[80] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. Tensorly: Tensor learning in python. *Journal of Machine Learning Research*, 20(26):1–6, 2019. See page 31.

[81] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning, 2019. See page 31.

[82] Paul Springer and Chen-Han Yu. cutensor: High-performance cuda tensor primitives. In *NVIDIA GPU Technology Conference 2019*, 2019. See page 31.

[83] Yu Pan, Maolin Wang, and Zenglin Xu. Tednet: A pytorch toolkit for tensor decomposition networks. *Neurocomputing*, 469:234–238, 2022. See page 31.

[84] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009. See page 31.

[85] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. See page 32.

[86] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. See pages 32, 34.

[87] Vin De Silva and Lek-Heng Lim. Tensor rank and the ill-posedness of the best low-rank approximation problem. *SIAM Journal on Matrix Analysis and Applications*, 30(3):1084–1127, 2008. See page 33.

[88] Lizhi Xiang, Miao Yin, Chengming Zhang, Aravind Sukumaran-Rajam, P Sadayappan, Bo Yuan, and Dingwen Tao. Tdc: Towards extremely efficient cnns on gpus via hardware-aware tucker

decomposition. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 260–273, 2023. See page 45.

[89] Glen Evenbly and Guifré Vidal. Tensor network states and geometry. *Journal of Statistical Physics*, 145:891–918, 2011. See page 46.