

# Convolutional Neural Network Compression via Tensor Decomposition

by

Mateusz Faltyn

B.Arts Sc. (Honours), McMaster University, 2021

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF  
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Mathematics)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

February 2023

© Mateusz Faltyn 2023

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Convolutional Neural Network Compression via Tensor Decomposition

submitted by Mateusz Faltyn in partial fulfillment of the requirements for

the degree of Master of Science

in Mathematics

**Examining Committee:**

Elina Robeva, Assistant Professor, Mathematics, UBC  
Supervisor

Yaniv Plan, Associate Professor, Mathematics, UBC  
Supervisory Committee Member

# Abstract

Computer vision models, such as image and video classifiers, are increasingly prevalent in Internet-of-Things systems. Since the advent of the AlexNet neural network model in 2012, convolutional neural networks have been demonstrated to be very effective at performing many computer vision tasks. However, convolutional neural networks' high computational and storage costs hinder the wider adoption of computer vision models in smaller Internet-of-Things devices such as mobile phones or embedded systems. As larger neural network models increase hardware costs, industry and academia have come together to tackle the problem of how to compress convolutional neural networks. Convolutional neural network compression via tensor decomposition has been shown to reduce the memory and storage requirements for devices to perform computer vision tasks successfully. In this work, we first review the preliminaries of tensor decomposition and define the four major types of tensor decompositions and their related decomposition algorithms in Chapter 1. Afterward, we introduce the building blocks of neural networks and describe convolutional neural networks in Chapter 2. Finally, we overview the different tensor decomposition approaches for convolutional neural network compression and display the results of two experiments using the PyTorch-TedNet CIFAR10 and CIFAR100 model benchmarks in Chapter 3.

# Lay Summary

Computer vision allows computers to understand and analyze images and videos. Cameras and special software allow computers to see and understand the world around us just like humans. For example, a computer with computer vision can be programmed to recognize different objects in an image, like a car, or a person. Neural networks are computer programs designed to process and analyze large amounts of data. They are called *neural networks* because they are inspired by how the human brain works. One way neural networks can be made more efficient is by *compressing* them. This means reducing the number of neurons and connections in the network, while still keeping its overall function and performance. This text explores how the neural networks we use for computer vision can be compressed with mathematical structures called tensors.

# Preface

This thesis is an original and independent work by the author, Mateusz Faltyn, under the supervision of Dr. Elina Robeva. Parts of Chapter 1 have been published as a section in the course material entitled *A Graduate Course in Tensor Decompositions and Applications* co-authored with Dr. Robeva for the *MATH 605D: Tensor Decompositions and their Applications* course held at the University of British Columbia in Fall 2022.

# Table of Contents

<b>Abstract</b>	iii
<b>Lay Summary</b>	iv
<b>Preface</b>	v
<b>Table of Contents</b>	vi
<b>List of Tables</b>	viii
<b>List of Figures</b>	ix
<b>List of Programs</b>	x
<b>Acknowledgements</b>	xi
<b>Dedication</b>	xii
<b>1 Tensors and Tensor Decomposition</b>	1
1.1 Preliminaries	1
1.1.1 Basic Definitions and Operations	1
1.1.2 Tensor Rank	5
1.1.3 Uniqueness	9
1.2 Tensor Decomposition and Algorithms	11
1.2.1 CP Decomposition	11
1.2.2 Tucker Decomposition	13
1.2.3 Tensor-Train Decomposition	14
1.2.4 Tensor-Ring Decomposition	14
<b>2 Neural Networks and Deep Learning</b>	16
2.1 Fundamentals of Neural Networks	16
2.1.1 Preliminaries	16
2.1.2 Definitions	18

## Table of Contents

---

2.2	Fundamentals of Deep Learning . . . . .	20
2.2.1	Data Acquisition . . . . .	20
2.2.2	Data Preprocessing . . . . .	21
2.2.3	Data Splitting . . . . .	21
2.2.4	Model Construction . . . . .	22
2.2.5	Model Training and Evaluation . . . . .	23
2.3	Learning from Data . . . . .	24
2.3.1	Neural Network Expressivity . . . . .	24
2.3.2	SGD and BP . . . . .	25
<b>3</b>	<b>Convolutional Neural Networks and Compression . . . . .</b>	<b>28</b>
3.1	Convolutional Neural Networks . . . . .	28
3.1.1	Convolutions . . . . .	28
3.1.2	Finding Edges . . . . .	30
3.1.3	Deep CNNs . . . . .	31
3.2	Compression Methods . . . . .	33
3.2.1	Benchmarks . . . . .	34
3.2.2	Precision Reduction . . . . .	34
3.2.3	Network Pruning . . . . .	35
3.2.4	Compact Architecture Construction . . . . .	35
3.3	CNN Compression via Tensor Decomposition . . . . .	36
3.3.1	CP Compression . . . . .	36
3.3.2	Tucker Compression . . . . .	37
3.3.3	Tensor-Train Compression . . . . .	37
3.3.4	Tensor-Ring Compression . . . . .	37
3.4	PyTorch-TedNet Model Benchmarks . . . . .	37
3.4.1	TedNet Tensorized Layers . . . . .	38
3.4.2	ResNet-32 . . . . .	39
3.4.3	CIFAR-10 Benchmark . . . . .	41
3.4.4	CIFAR-100 Benchmark . . . . .	44
3.4.5	Conclusion . . . . .	46
	<b>Bibliography . . . . .</b>	<b>48</b>

# List of Tables

3.1	Common CNN compression benchmark datasets and models.	34
3.2	CIFAR-10 benchmark results (50 Epochs, Single GPU). . . .	41
3.3	CIFAR-100 benchmark results (50 Epochs, Single GPU). . . .	44



# List of Figures

2.1	A feedforward neural network, [44]. . . . .	18
2.2	Standard deep learning workflow. . . . .	20
2.3	Neural network learning process, [42]. . . . .	24
3.1	Output of different types of convolutional kernels applied on an image, [6]. . . . .	28
3.2	Example convolution of an image (left) with a kernel (right), [49]. . . . .	30
3.3	LeNet-5 architecture. . . . .	33

# List of Programs

1	ResNet-32 - Import libraries . . . . .	39
2	ResNet-32 - Model, [19] . . . . .	40
3	Import libraries . . . . .	41
4	Configure GPU . . . . .	42
5	CIFAR-10 - Load dataset, [22] . . . . .	42
6	CIFAR-10 - Define train and test processes . . . . .	43
7	CIFAR-10 - Define model . . . . .	43
8	CIFAR-10 - Train and evaluate model, [40, 39] . . . . .	44
9	CIFAR-100 - Load dataset [22] . . . . .	44
10	CIFAR-100 - Define train and test processes . . . . .	45
11	CIFAR-100 - Define model . . . . .	45
12	CIFAR-100 - Train and evaluate model, [40, 39] . . . . .	46

# Acknowledgements

To begin, I would like to thank my supervisor, Dr. Elina Robeva, for her guidance throughout my Master of Science in Mathematics journey. Her insights and expertise have been invaluable, and I have learned much from working with her.

Furthermore, I am deeply indebted to Dr. Ryan Van Lieshout, who has supported my professional development since the beginning of my academic career.

Lastly, I would like to thank Dr. Yaniv Plan for his time and efforts in reading and commenting on my thesis.

# Dedication

To Nina, Asia, Mama, Tata, and Stella.

# Chapter 1

## Tensors and Tensor Decomposition

### 1.1 Preliminaries

The history of tensor decomposition can be traced to Hitchcock in 1927 [16], and Cattell in 1944 [4], with the extension of matrix decomposition techniques to higher-dimensional arrays. However, decomposing tensors did not become widespread until 1970, when researchers in the psychometrics community started using them to analyze and understand high-dimensional data. There were two papers on the same tensor decomposition in 1970: in one of these papers, Carroll and Chang [3] called the decomposition *CAN-DECOMP*; in the other, Harshman, called it *PARAFAC* [15]. That is why it is called CANDECOMP/PARAFAC- or CP-decomposition today. This section introduces the notion of a tensor, basic tensor arithmetic, and CP-decomposition.

#### 1.1.1 Basic Definitions and Operations

While linear algebra is concerned with the thorough study of matrices (2-dimensional arrays) and vectors (1-dimensional arrays). Here, we consider  $d$ -dimensional arrays for any  $d \geq 1$ . Thus, we study *multilinear* algebra. We begin with the formal definition of a tensor.

**Definition 1.1.1. (Tensor)** A *tensor* is a multidimensional array  $T \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$  (or  $\mathbb{C}^{n_1 \times n_2 \times \dots \times n_d}$ ). We say that  $T$  has *order*  $d$ , or is a  $d$ -way tensor. We refer to its entries by  $T_{i_1, \dots, i_d}$  where:

$$1 \leq i_1 \leq n_1, \dots, 1 \leq i_d \leq n_d.$$

The *size* of  $T$  is  $n_1 \times \dots \times n_d$ . We refer to the  $d$  dimensions of  $T$  as its *modes*.

**Definition 1.1.2. (Tensor Slice)** The *slices* of  $T$  are the subtensors obtained after fixing one or more of the indices of  $T$ .

**Example 1.1.1.** If  $T$  is a 3-way tensor of size  $n_1 \times n_2 \times n_3$ , then,  $T_{i_1..}$  is an  $n_2 \times n_3$  slice for any  $i_1 \in \{1, \dots, n_1\}$ ,  $T_{.i_2.}$  is an  $n_1 \times n_3$  slice for any  $i_2 \in \{1, \dots, n_2\}$ , and  $T_{..i_3}$  is an  $n_1 \times n_2$  slice for any  $i_3 \in \{1, \dots, n_3\}$ . These are called the horizontal, lateral, and frontal slices, respectively.

The slices of tensors can be combined to form a matrix.

**Definition 1.1.3. (Matricization/Flattening of a Tensor)** We call *matricization* (or *flattening*) the process of reordering the elements of a tensor  $T$  of order  $d$  into a matrix. If  $T$  is an  $n_1 \times \dots \times n_d$  tensor, then for any subset  $\emptyset \subsetneq I \subsetneq \{1, \dots, d\}$  we can define the flattening  $T^{I|(\{1, \dots, d\} \setminus I)}$  as a  $\Pi_{i \in I} n_i \times \Pi_{j \notin I} n_j$  matrix, whose rows correspond to the indices in  $I$  and whose columns correspond to the indices in  $\{1, \dots, d\} \setminus I$ .

**Example 1.1.2.** A  $3 \times 4 \times 2$  tensor  $T$  can be rearranged as a  $6 \times 4$  matrix, a  $3 \times 8$  matrix, or a  $2 \times 12$  matrix [21]. Consider  $T \in \mathbb{R}^{3 \times 4 \times 2}$  with slices  $T_{..1}$  and  $T_{..2}$ :

$$T_{..1} = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix},$$

$$T_{..2} = \begin{bmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{bmatrix}.$$

We can write  $T$  as  $T = [T_{..1} | T_{..2}]$ .  $T$  can be flattened in three ways:

$$T^{1|23} = \begin{bmatrix} 1 & 4 & 7 & 10 & 13 & 16 & 19 & 22 \\ 2 & 5 & 8 & 11 & 14 & 17 & 20 & 23 \\ 3 & 6 & 9 & 12 & 15 & 18 & 21 & 24 \end{bmatrix} \in \mathbb{R}^{3 \times 8},$$

$$T^{2|13} = \begin{bmatrix} 1 & 2 & 3 & 13 & 14 & 15 \\ 4 & 5 & 6 & 16 & 17 & 19 \\ 7 & 8 & 9 & 19 & 20 & 21 \\ 10 & 11 & 12 & 22 & 23 & 24 \end{bmatrix} \in \mathbb{R}^{4 \times 6},$$

$$T^{2|13} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \end{bmatrix} \in \mathbb{R}^{2 \times 12}.$$

We will now define a few important tensor operations used throughout this text. Since the set of tensors of a given size is a vector space, addition, subtraction, and scalar multiplication are well-defined.

**Definition 1.1.4. (Hadamard Product)** We call element-wise matrix multiplication the Hadamard product. Let  $A$  and  $B$  be matrices of size  $I \times J$ . The Hadamard product is denoted as  $A \star B$  and the result is of size

$$A \star B = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1J}b_{1J} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2J}b_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}b_{I1} & a_{I2}b_{I2} & \dots & a_{IJ}b_{IJ} \end{bmatrix}.$$

**Definition 1.1.5. ( $k$ -Mode Product)** We call the multiplication of a tensor  $T \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$  by a matrix  $U \in \mathbb{R}^{J_k \times n_k}$  the  $k$ -mode product denoted by  $\bullet_k$ . The result is of size  $n_1 \times \dots \times n_{k-1} \times J_k \times n_{k+1} \times \dots \times n_d$ :

$$(T \bullet_k U)_{i_1 \dots i_{k-1} j i_{k+1} \dots i_d} = \sum_{i_k=1}^{n_k} T_{i_1 \dots i_{k-1} i_k i_{k+1} \dots i_d} u_{j i_k}.$$

We call the multiplication of a tensor  $T \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$  by a vector  $v \in \mathbb{R}^{n_k}$  the  $k$ -mode (vector) product denoted by  $\bullet_k$ . The result is of size  $n_1 \times \dots \times n_{k-1} \times n_{k+1} \times \dots \times n_d$ :

$$(X \bullet_k v)_{i_1 \dots i_{k-1} i_{k+1} \dots i_d} = \sum_{i_k=1}^{n_k} x_{i_1 \dots i_d} v_{i_k}.$$

**Definition 1.1.6. (Kronecker Product)** We call the *Kronecker product* of matrices  $A \in \mathbb{R}^{I \times J}$  and  $B \in \mathbb{R}^{K \times L}$  the operation denoted as  $A \otimes B$  resulting in a matrix of size  $(IK) \times (JL)$ :

$$\begin{aligned} A \otimes B &= \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1J}B \\ a_{21}B & a_{22}B & \dots & a_{2J}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}B & a_{I2}B & \dots & a_{IJ}B \end{bmatrix} \\ &= [a_1 \otimes b_1 \quad a_1 \otimes b_2 \quad a_1 \otimes b_3 \quad \dots \quad a_J \otimes b_{L-1} \quad a_J \otimes b_L]. \end{aligned}$$

**Definition 1.1.7. (Khatri-Rao Product)** Let  $A \in \mathbb{R}^{n_1 \times n_3}$  and  $B \in \mathbb{R}^{n_2 \times n_3}$  be matrices. The *Khatri-Rao product*, denoted by  $A \odot B$ , is of size  $(n_1 n_2) \times n_3$  and is denoted as follows:

$$A \odot B = [\text{vec}(a_1 \otimes b_1) \quad \text{vec}(a_2 \otimes b_2) \quad \dots \quad \text{vec}(a_K \otimes b_K)],$$

where  $\text{vec}(M)$  arranges the entries of an  $m \times n$  matrix  $M$  into a vector of length  $mn$ .

We can define a norm for a tensor analogous to the Frobenius norm for matrices.

**Definition 1.1.8. (Norm)** We define the norm  $\|T\|$  of a tensor  $T$  as the square root of the sum of the squares of all its entries:

$$\|T\| = \sqrt{\sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \dots \sum_{i_d=1}^{n_d} x_{i_1, \dots, i_d}^2}.$$

Furthermore, the notion of an inner product of matrices easily extends to tensors:

**Definition 1.1.9. (Inner Product)** The inner product of two tensors of the same size  $T, S \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$  is:

$$\langle T, S \rangle = \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \dots \sum_{i_d=1}^{n_d} x_{i_1, \dots, i_d} y_{i_1, \dots, i_d}.$$

Then,  $\langle T, T \rangle = \|T\|^2$ .

We now define special classes of tensors that may appear throughout this manuscript and related works.

**Definition 1.1.10. (Symmetric)** A  $d$ -way tensor  $T$  of size  $n \times \dots \times n$  is called *symmetric* if its elements remain unchanged under any index permutation. In other words,

$$T_{i_1, \dots, i_d} = T_{i_{\pi(1)}, \dots, i_{\pi(d)}},$$

for any permutation  $\pi$  on  $\{1, \dots, d\}$  and any  $i_1, \dots, i_d \in \{1, \dots, n\}$ .

**Example 1.1.3.** A tensor  $T \in \mathbb{R}^{n \times n \times n}$  is symmetric if:

$$T_{ijk} = T_{ikj} = T_{jik} = T_{jki} = T_{kij} = T_{kji}, \text{ for all } i, j, k = 1, \dots, n.$$

**Definition 1.1.11. (Diagonal Tensors)** We call a tensor  $T \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$  *diagonal* if:

$$T_{i_1 i_2 \dots i_n} \neq 0 \text{ only if } i_1 = i_2 = \dots = i_d.$$



### 1.1.2 Tensor Rank

We now introduce the concept of a rank-1 tensor:

**Definition 1.1.12. (Rank-1 Tensors)** We call a  $d$ -way tensor  $T \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$  *rank-1* if it can be written as the outer product of  $d$  vectors:

$$T = a^{(1)} \otimes a^{(2)} \otimes \dots \otimes a^{(d)} \text{ where } a^{(i)} \in \mathbb{R}^{n_i}.$$

Entry-wise, this means that:

$$T_{i_1, \dots, i_d} = a_{i_1}^{(1)} \otimes a_{i_2}^{(2)} \otimes \dots \otimes a_{i_d}^{(d)} \text{ for all } i_1, \dots, i_d.$$

**Example 1.1.4.** If  $T \in \mathbb{R}^{n_1 \times n_2 \times n_3}$  then  $T$  has rank-1 if  $T = a \otimes b \otimes c$  for some vectors  $a \in \mathbb{R}^{n_1}, b \in \mathbb{R}^{n_2}, c \in \mathbb{R}^{n_3}$ . Then, the entry at position  $(i, j, k)$  of  $T$  is equal to  $T_{ijk} = a_i b_j c_k$ .

The definition of a rank of a tensor closely follows the definition of rank-1:

**Definition 1.1.13. (Rank)** The *rank* of a tensor  $T$ , denoted  $\text{rank}(T)$ , is the smallest number  $r$  such that  $T$  can be written as a sum of  $r$  rank-1 tensors:

$$T = \sum_{i=1}^r a^{(1,i)} \otimes a^{(2,i)} \otimes \dots \otimes a^{(d,i)}.$$

Here  $a^{(j,i)} \in \mathbb{R}^{n_j}$  for  $j = 1, \dots, d$  and  $i = 1, \dots, r$ .

**Definition 1.1.14. (CP-Decomposition)** Let  $T$  be a  $d$ -order tensor. A *CP-decomposition* of  $T$  has the form:

$$T = \sum_{i=1}^r a^{(1,i)} \otimes a^{(2,i)} \otimes \dots \otimes a^{(d,i)} = \llbracket A^{(1)}, A^{(2)}, \dots, A^{(d)} \rrbracket,$$

where vectors  $a^{(1,i)} \in \mathbb{R}^{n_1}, a^{(2,i)} \in \mathbb{R}^{n_2}, \dots, a^{(d,i)} \in \mathbb{R}^{n_d}$  for  $i = 1, \dots, r$ . In other words, a CP-decomposition of  $T$  is a sum of  $r$  rank-1 tensors. The smallest  $r$  for which such a decomposition exists is the rank (or CP-rank) of  $T$ .

Upon further investigation, we see that the rank of a tensor depends on the field. Let  $T \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$  and  $T = \sum_{i=1}^r a^{(1,i)} \otimes a^{(2,i)} \otimes \dots \otimes a^{(d,i)}$ :

- $\text{rank}_{\mathbb{R}}(T)$  is the smallest  $r$  such that all  $a^{(1,i)}, \dots, a^{(d,i)}$  for all  $i = 1, \dots, r$  are in  $\mathbb{R}$ .

- $\text{rank}_{\mathbb{C}}(T)$  is the smallest  $r$  such that all  $a^{(1,i)}, \dots, a^{(d,i)}$  for all  $i = 1, \dots, r$  are in  $\mathbb{C}$ .

It's clear that  $\text{rank}_{\mathbb{R}}(T) \geq \text{rank}_{\mathbb{C}}(T)$  for every tensor  $T$ . However, the following example shows that equality does not always hold.

**Example 1.1.5.** Let  $T \in \mathbb{R}^{2 \times 2 \times 2}$  such that:

$$T = \left[ \begin{array}{cc|cc} 1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \end{array} \right].$$

We see that  $\text{rank}_{\mathbb{R}}(T) \geq 3$  but  $\text{rank}_{\mathbb{C}}(T) = 2$ :

$$T = \frac{1}{2} \left( \left[ \begin{array}{c} 1 \\ -i \end{array} \right] \otimes \left[ \begin{array}{c} 1 \\ i \end{array} \right] \otimes \left[ \begin{array}{c} 1 \\ -i \end{array} \right] + \left[ \begin{array}{c} 1 \\ i \end{array} \right] \otimes \left[ \begin{array}{c} 1 \\ -i \end{array} \right] \otimes \left[ \begin{array}{c} 1 \\ i \end{array} \right] \right).$$

This is not true for matrices. Another difference between matrices and tensors that causes many issues is the following lemma:

**Lemma 1.1.1.** The set of tensors with rank at most  $r$  is not closed if  $r \geq 2$ .

**Example 1.1.6.** For every  $n \in \mathbb{N}$ , consider the following tensors

$$\begin{aligned} S_n &= \left[ \begin{array}{cc|cc} n & 1 & 1 & \frac{1}{n} \\ 1 & \frac{1}{n} & \frac{1}{n} & \frac{1}{n^2} \end{array} \right] = \frac{1}{n^2} \left[ \begin{array}{c} n \\ 1 \end{array} \right] \otimes \left[ \begin{array}{c} n \\ 1 \end{array} \right] \otimes \left[ \begin{array}{c} n \\ 1 \end{array} \right], \\ R_n &= \left[ \begin{array}{cc|cc} -n & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right] = \left[ \begin{array}{c} -n \\ 0 \end{array} \right] \otimes \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \otimes \left[ \begin{array}{c} 1 \\ 0 \end{array} \right], \\ T_n &= S_n + R_n = \left[ \begin{array}{cc|cc} 0 & 1 & 1 & \frac{1}{n} \\ 1 & \frac{1}{n} & \frac{1}{n} & \frac{1}{n^2} \end{array} \right]. \end{aligned}$$

For every  $n \in \mathbb{N}$ ,  $T_n$  is a rank-2 tensor. The sequence  $\{T_n\}$  converges to:

$$T = \left[ \begin{array}{cc|cc} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{array} \right].$$

Suppose we have tensor  $T$  and want to approximate  $T$  by a low-rank, say rank- $r$ , tensor with respect to the norm  $\|T\| = \sqrt{\sum_{i_1 \dots i_d} |T_{i_1 \dots i_d}|^2}$ . Considering Lemma 2.1, the best rank- $r$  approximation might not exist. The Eckart-Young theorem allows us always to find the best low-rank approximation for a matrix as follows:

**Theorem 1.1.2. (Eckart-Young, [10])** Let  $A$  be a matrix of rank  $r$  with the singular value decomposition  $A = \sum_{i=1}^r \sigma_i u_i \otimes v_i$ , where  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r$  are the positive singular values of  $A$ . Suppose that  $k < r$  and we want to find the best rank- $k$  approximation of  $A$ , i.e. a matrix  $B$  with rank at most  $k$  such that:

$$\|A - B\| = \min_{\text{rank}(B) \leq k} \|A - B\|.$$

Then,

$$B = \sum_{i=1}^k \sigma_i u_i \otimes v_i.$$

For tensors, their best low-rank approximation has nothing to do with their CP-decomposition in general. For example, Kolda and Bader [21] shows that the best rank-1 approximation of a tensor like:

$$T = \sigma_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \sigma_2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

is

$$S = \gamma x \otimes y \otimes z,$$

where

$$x, y, z \neq \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

**Definition 1.1.15. (Maximal Rank)** The *maximal rank* of  $n_1 \times n_2 \times \dots \times n_d$  tensors over  $\mathbb{R}$  (or  $\mathbb{C}$ ) is the maximal possible rank of such a tensor over  $\mathbb{R}$  (or  $\mathbb{C}$ ).

**Definition 1.1.16. (Typical Rank)** The *typical rank* of  $n_1 \times n_2 \times \dots \times n_d$  tensors over  $\mathbb{R}$  (or  $\mathbb{C}$ ) is any rank  $r$  that occurs with positive probability, i.e. on a subset of  $\mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$  (or  $\mathbb{C}^{n_1 \times n_2 \times \dots \times n_d}$ ) with positive Lebesgue measure.

**Note:**

- Maximal rank and typical rank of  $n_1 \times n_2$  matrices are equal to  $\min(n_1, n_2)$ .
- Maximal rank and typical rank of  $n_1 \times n_2 \times n_3 \times n_4$  tensors for  $d \geq 3$  may be different.
- For  $n_1 \times n_2 \times n_3 \times n_4$  tensors with  $d \geq 3$ , there may be more than one typical rank over  $\mathbb{R}$ .

- For  $n_1 \times n_2 \times n_3 \times n_4$  tensors, there is always one typical rank over  $\mathbb{C}$ , which is called *generic rank*.

**Example 1.1.7.** For  $2 \times 2 \times 2$  tensors  $T \in \mathbb{R}^{2 \times 2 \times 2}$ ,

- The maximal rank over  $\mathbb{R}$  and the maximal rank over  $\mathbb{C}$  are both 3.
- The typical ranks over  $\mathbb{R}$  are 2 (79%) and 3 (21%) when choosing random  $2 \times 2 \times 2$  real tensors whose entries are independent standard Gaussians.
- The typical rank over  $\mathbb{C}$  is 2.

**Note:** Tensors of rank 1 have probability 0.

We will count the degrees of freedom of tensors of rank at most  $r$  and find the smallest  $r$  where the number of degrees of freedom is at least as large as the dimension of  $\mathbb{C}^{n_1 \times n_2 \times \dots \times n_d}$  which is  $n_1 \dots n_d$ .

**Lemma 1.1.3. (Degrees of Freedom in a Rank-1 Tensor)** The set of rank-1 tensors  $a^{(1)} \otimes \dots \otimes a^{(d)}$  has dimension:

$$(n_1 - 1) + \dots + (n_d - 1) + 1 = n_1 + \dots + n_d - d + 1.$$

*Proof.* It is immediately clear that  $a^{(1)} \otimes \dots \otimes a^{(d)}$  has at most  $n_1 + \dots + n_d$  degrees of freedom, but note that this is up to a constant scalar factor in each term. Instead, require that  $\|a^{(i)}\| = 1$  and consider:

$$\lambda a^{(1)} \otimes \dots \otimes a^{(d)},$$

which has  $1 + (n_1 - 1) + \dots + (n_d - 1) = n_1 + \dots + n_d - d + 1$  degrees of freedom.  $\square$

Now we define the *expected generic rank* of a tensor in  $\mathbb{C}^{n_1 \times \dots \times n_d}$  to be the smallest  $r$  such that the number of degrees of freedom of tensors of the form  $\sum_{i=1}^r a^{(1,i)} \otimes \dots \otimes a^{(d,i)}$  is more than or equal to the number of degrees of freedom of the whole space  $\mathbb{C}^{n_1 \times \dots \times n_d}$ , which is  $n_1 \dots n_d$ . Thus, we need the smallest  $r$  such that  $r(n_1 + \dots + n_d - d + 1) \geq n_1 \dots n_d$ .

**Definition 1.1.17. (Expected Generic Rank)** The *expected generic rank* in  $\mathbb{C}^{n_1 \times \dots \times n_d}$  is:

$$\left\lceil \frac{n_1 \dots n_d}{n_1 + \dots + n_d - d + 1} \right\rceil.$$

In particular, if  $n_i = n \forall i$ , then the expected generic rank is

$$\frac{n^d}{dn - d + 1} = \frac{n^d}{d(n - 1) + 1} \sim \frac{n^{d-1}}{d},$$

which is smaller than the known maximal rank bound of  $n^{d-1}$ .

**Theorem 1.1.4. (Landsberg, [25])** In general, the generic rank is not the same as the expected generic rank.

1. **(Strassen, [47])** The generic rank of a tensor in  $\mathbb{C}^{3 \times 3 \times 3}$  is 5.
2. **(Strassen-Lickteig, [31])** For  $n \neq 3$ , the generic rank of a tensor in  $\mathbb{C}^{n \times n \times n}$  is the same as the expected generic rank of  $\lceil \frac{n^3}{3n-2} \rceil$ .
3. The generic rank of tensors in  $\mathbb{C}^{2 \times 2 \times 3}$  and  $\mathbb{C}^{2 \times 3 \times 3}$  is the expected generic rank 3.

The set of tensors of a given rank is not closed, i.e. there may exist sequences of tensors  $T_n, n \geq 1$  of rank  $r$  for which  $T_n \rightarrow T$  as  $n \rightarrow \infty$ , but  $T$  has rank  $> r$ . Motivated by this, we define the *border rank* of a tensor  $T$  below.

**Definition 1.1.18. (Border Rank)** For some tensor  $T$ , define its *border rank* to be:

$$\begin{aligned} \underline{\text{rank}}(T) &= \min\{r | T \text{ is the limit of tensors of rank } \leq r\} \\ &= \min\{r | \forall \epsilon > 0, \exists S \text{ s.t. } \text{rank}(S) \leq r \text{ and } \|T - S\| \leq \epsilon\}. \end{aligned}$$

Here,  $\|\cdot\|$  is the Frobenius norm for tensors, i.e.,  $\|T\| = \sqrt{\sum_{i_1, \dots, i_d} T_{i_1, \dots, i_d}^2}$ .

### 1.1.3 Uniqueness

Matrix decompositions are not unique. To see this, consider a matrix  $M \in \mathbb{C}^{n \times m}$  of rank  $r$ . We may form a decomposition:

$$\begin{aligned} M &= AB^\top = \sum_{i=1}^r a_i b_i^\top = \sum_{i=1}^r a_i \otimes b_i, \text{ where} \\ A &= \begin{bmatrix} | & & | \\ a_1 & \dots & a_r \\ | & & | \end{bmatrix}, \\ B &= \begin{bmatrix} | & & | \\ b_1 & \dots & b_r \\ | & & | \end{bmatrix}. \end{aligned}$$

However, for any  $r \times r$  orthogonal matrix  $U$ , we can have a different decomposition:

$$M = A(UU^\top)B^\top = (AU)(BU^\top).$$

We first define the Kruskal rank of a matrix:

**Definition 1.1.19. (Kruskal Rank, [24])** Consider a matrix:

$$W = \begin{bmatrix} | & & | \\ w_1 & \dots & w_n \\ | & & | \end{bmatrix}.$$

We say that the columns of  $W$  are in  $k$ -general linear position if any  $k$  columns are linearly independent. The *Kruskal rank* of  $W$ , denoted  $\kappa_W$ , is the maximal  $r$  such that the columns of  $W$  are in  $r$ -general linear position.

Note that the Kruskal rank of a matrix is different from the conventional notion of rank: rank  $r$  means that there *exist*  $r$  linearly independent columns, but Kruskal rank  $r$  requires that *any collection* of  $r$  columns is linearly independent.

**Theorem 1.1.5. (Kruskal, [24])** Let  $T$  be a  $n_1 \times n_2 \times n_3$  tensor, and suppose we may write it as follows:

$$T = \sum_{i=1}^r a_i \otimes b_i \otimes c_i.$$

Let

$$A = \begin{bmatrix} | & & | \\ a_1 & \dots & a_r \\ | & & | \end{bmatrix}, B = \begin{bmatrix} | & & | \\ b_1 & \dots & b_r \\ | & & | \end{bmatrix}, C = \begin{bmatrix} | & & | \\ c_1 & \dots & c_r \\ | & & | \end{bmatrix}.$$

If  $r \leq \frac{1}{2}(\kappa_A + \kappa_B + \kappa_C) - 1$ , then  $T$  has rank  $r$  and its rank- $r$  decomposition is unique up to permutation and scaling.

We also have the following generalization of Kruskal's theorem.

**Theorem 1.1.6. (Sidiropoulos and Bro, [45])** Let  $T$  be a  $n_1 \times \dots \times n_d$  tensor that can be written as follows:

$$T = \sum_{i=1}^r a^{(1,i)} \otimes \dots \otimes a^{(d,i)}.$$

Let

$$A^{(j)} = \begin{bmatrix} \begin{array}{c} | \\ a^{(j,1)} \\ | \end{array} & \dots & \begin{array}{c} | \\ a^{(j,r)} \\ | \end{array} \end{bmatrix}.$$

Then, the above decomposition is unique, and  $T$  has rank  $r$  if

$$r \leq \frac{1}{2} \left( \sum_{i=1}^d \kappa_{A^{(i)}} \right) - \frac{d-1}{2}.$$

**Theorem 1.1.7.** The Kruskal rank conditions are sufficient. They are also necessary conditions for uniqueness if rank is 2 or 3.

**Theorem 1.1.8. (Chiantini and Ottaviani, [5])** If  $T \in \mathbb{C}^{n_1 \times n_2 \times n_3}$  with

$$n_1 \leq n_2 \leq n_3 \text{ and } \underline{\text{rank}}(T) \leq \frac{n_1 n_2}{16},$$

then with probability 1  $T$  has a unique decomposition into a sum of  $\underline{\text{rank}}(T)$  rank-1 terms, i.e.  $\text{rank}(T) = \underline{\text{rank}}(T)$ .

## 1.2 Tensor Decomposition and Algorithms

In this section, we provide an overview of four important tensor decompositions and their respective algorithms.

### 1.2.1 CP Decomposition

There are three major types of algorithms that are commonly used to compute the CP decomposition (defined in Section 1.1.2) of a tensor: Alternating Least Squares, Jennrich's Algorithm, and the Tensor Power Method.

---

**Algorithm 1** Alternating Least Squares (ALS), [3, 15]

---

**Require:** Tensor  $T = \llbracket A^{(1)}, A^{(2)}, \dots, A^{(d)} \rrbracket \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$ .

**Initialize:**  $A_1, A_2, \dots, A_n$  randomly.

**repeat**

**for**  $i = 1, 2, \dots, d$ . **do**

$$A_i = \arg \min_X \mathcal{L}(A_1, A_2, \dots, A_{i-1}, X, A_{i+1}, \dots, A_d).$$

**end for**

**until** convergence.

**return**  $A_1, A_2, \dots, A_d$ .

---

## 1.2. Tensor Decomposition and Algorithms

---

### Algorithm 2 Jennrich's Algorithm, [15]

---

**Require:** Tensor  $T \in \mathbb{R}^{n \times m \times p}$ .

**Initialize:**  $M_a = T_{..a}, M_b = T_{..b}$  s.t.  $a, b \sim N(0, \frac{1}{p})^p$  i.i.d.

Set eigenvectors  $\{u_i : i \in [k]\}$  of  $k$  largest eigenvalues of  $M_a(M_b)^\dagger$ .

Set eigenvectors  $\{v_i : i \in [k]\}$  of  $k$  largest eigenvalues of  $((M_b)^\dagger M_a)^T$ .

Pair  $\{u_i, v_i\}$  if corresponding eigenvalues are (approximately) reciprocal.

Solve  $T = \sum_{i=1}^k u_i \otimes v_i \otimes w_i$  for vectors  $w_i$ .

**return** factor matrices  $U \in \mathbb{R}^{n \times k}, V \in \mathbb{R}^{m \times k}, W \in \mathbb{R}^{p \times k}$ .

---



---

### Algorithm 3 Tensor Power Method, [2]

---

**Require:**  $\|T\| \neq 0$

**for**  $i = 1, 2, \dots$  **do**

Randomly sample and normalize  $\theta \in S^d(\mathbb{R}^n)^n$  s.t.  $\|\theta\| = 1$ .

Run power iteration starting with  $\theta$  on eigenvector  $v_i \in \mathbb{R}^n$ .

**until** convergence.

**Eigenvalue:**  $\lambda_i \leftarrow T \cdot v_i^d = \langle T, v_i^{\otimes d} \rangle$ .

**Deflate:**  $T \leftarrow T - \lambda_i v_i^{\otimes d}$ .

**end for**

**return**  $\{v_1, \dots, v_r\}$  and  $\{\lambda_1, \dots, \lambda_r\}$ .

---



### 1.2.2 Tucker Decomposition

Here we define the Tucker decomposition: factorizing a tensor into a *core tensor* and matrices that scale the core for each mode. As the Tucker decomposition provides a summary of the data, it can be seen as a higher-order principal component analysis (PCA). We also define the two commonly used algorithms to compute the Tucker decomposition of a tensor: Higher Order Singular Value Decomposition (HOSVD) and Higher Order Orthogonal Iteration (HOOI).

**Definition 1.2.1. (Tucker Decomposition)** Let  $T$  be a  $d$ -order tensor. A *Tucker decomposition* of  $T$  has the form:

$$T = \sum_{i_1=1}^{r_1} \sum_{i_2=1}^{r_2} \cdots \sum_{i_d=1}^{r_d} g_{i_1 i_2 \dots i_d} a^{(1,i)} \otimes a^{(2,i)} \otimes \cdots \otimes a^{(d,i)} = \llbracket \mathcal{G}; A^{(1)}, A^{(2)}, \dots, A^{(d)} \rrbracket,$$

where vectors  $a^{(1,i)} \in \mathbb{R}^{n_1}$ ,  $a^{(2,i)} \in \mathbb{R}^{n_2}$ ,  $\dots$ ,  $a^{(d,i)} \in \mathbb{R}^{n_d}$  for  $i_1 = 1, \dots, r_1$  to  $i_d = 1, \dots, r_d$ . In other words, a Tucker decomposition of  $T$  is the decomposition into a core tensor  $\mathcal{G}$  multiplied by a single matrix  $A^{(1)}, \dots, A^{(d)}$  along each of its  $d$  modes.

---

**Algorithm 4** Higher Order Singular Value Decomposition (HOSVD), [26]

---

**Require:** Tensor  $T = \llbracket A^{(1)}, A^{(2)}, \dots, A^{(d)} \rrbracket \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$ .  
**for**  $i = 1, 2, \dots, d$  **do**  
     $A^i \leftarrow r_i$  leading left singular vectors of matrices  $X_i$ .  
**end for**  
 $\mathcal{G} \leftarrow T \times_1 A^{(1)T} \times_2 A^{(2)T} \times_3 \cdots \times_d A^{(d)T}$ .  
**return**  $\mathcal{G}, A^{(1)}, A^{(2)}, \dots, A^{(d)}$ .

---



---

**Algorithm 5** Higher Order Orthogonal Iteration (HOOI), [27]

---

**Require:** Tensor  $T = \llbracket A^{(1)}, A^{(2)}, \dots, A^{(d)} \rrbracket \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$ .  
**Initialize**  $A^i \in \mathbb{R}^{I_i \times r}$  for  $i = 1, \dots, d$  via HOSVD.  
**repeat**  
    **for**  $i = 1, 2, \dots, d$  **do**  
        Tensor  $S \leftarrow T \times_1 A^{(1)T} \times_2 \cdots \times_{i-1} A^{(i-1)T} \times_{i+1} A^{(i+1)T} \times_{i+2} \cdots \times_d A^{(d)T}$ .  
         $A^i \leftarrow r_i$  leading left singular vectors of matrices  $Y_i$ .  
    **end for**  
**until** stopping criterion satisfied.  
 $\mathcal{G} \leftarrow T \times_1 A^{(1)T} \times_2 A^{(2)T} \times_3 \cdots \times_d A^{(d)T}$ .  
**return**  $\mathcal{G}, A^{(1)}, A^{(2)}, \dots, A^{(d)}$ .

---

### 1.2.3 Tensor-Train Decomposition

Using tensor network notation, one can arrive at many tensor decompositions. One of the most important tensor network decompositions is the Tensor-Train decomposition and its associated singular-value decomposition algorithm.

**Definition 1.2.2. (Tensor-Train Decomposition)** Let  $T$  be a  $d$ -order tensor. A *Tensor-Train (TT) decomposition* of  $T$  has the form:

$$\begin{aligned} T(i_1, \dots, i_d) &= \sum_{\alpha_0, \dots, \alpha_d} G_1(\alpha_0, i_1, \alpha_1) G_2(\alpha_1, i_2, \alpha_2) \dots G_d(\alpha_{d-1}, i_d, \alpha_d), \\ &= G_1[i_1] G_2[i_2] \dots G_d[i_d], \end{aligned}$$

where  $G_1[i_1]$  is size  $1 \times r_1$ ,  $G_2[i_2]$  is size  $r_1 \times r_2$ , and  $G_d[i_d]$  is size  $r_{d-1} \times 1$ . We refer to matrices  $G_i$  as the *TT-cores* and  $r_i$  as the *TT-ranks*. We denote  $r = \max r_i$  to be the *maximal TT-rank*. The TT-format is also referred to as the *Matrix Product State (MPS)*.

---

**Algorithm 6** Tensor-Train Singular Value Decomposition (TT-SVD), [48]

---

**Require:**  $d$ -order tensor  $T$  and prescribed accuracy  $\epsilon$ .

**Initialize**  $\delta = \frac{\epsilon}{\sqrt{d-1}} \|T\|_F$ ,  $C = T$ , and  $r_0 = 1$ .

**for**  $k = 1$  to  $d - 1$  **do**

$C = \text{reshape}(C, [r_k n_k, \frac{\text{number of elements of } C}{r_{k-1} n_k}])$ .

Compute  $\delta$ -truncated SVD of  $C : C = U \sum V^T + E$  s.t.  $\|E\|_F \leq \delta$ .

$r_k = \text{rank}(U \sum V^T)$ .

Set  $G_k = \text{reshape}(U, [r_{k-1}, n_k, r_k])$ .

$C = \sum V^T$ .

**end for**

**return** tensor  $S$  in TT-format with cores  $G_1, \dots, G_d$ .

---

### 1.2.4 Tensor-Ring Decomposition

The Tensor-Ring decomposition is a generalization of the TT decomposition such that the trace operation  $r_1 = r_{d+1} = 1$  condition is not necessary. Here we present the decomposition as well as its associated singular-value decomposition algorithm.

**Definition 1.2.3. (Tensor-Ring Decomposition)** Let  $T$  be a  $d$ -order

## 1.2. Tensor Decomposition and Algorithms

---

tensor. A *Tensor-Ring (TR) decomposition* of  $T$  has the form:

$$\begin{aligned} T(i_1, \dots, i_d) &= \text{Trace}(G_1[i_1] \dots G_d[i_d]), \\ &= \sum_{\alpha_0=1}^{r_0} \dots \sum_{\alpha_{d-1}=1}^{r_{d-1}} G_1(\alpha_0, i_1, \alpha_1) \dots G_d(\alpha_{d-1}, i_d, \alpha_0), \end{aligned}$$

where  $G_k[i_k]$  is size  $r_{k-1} \times r_k$  for each  $i_k$  such that  $1 \leq i_k \leq n_k$ . We refer to the 3-order tensors  $G_k$  as the *TR-cores* and the vector  $(r_0, r_1, \dots, r_d)$  such that  $r_0 = r_d$  as the *TR-rank*. The TR decomposition can be seen as a generalization of the TT decomposition where  $r_0 = r_d = 1$ .

---

### Algorithm 7 Tensor-Ring Singular Value Decomposition (TR-SVD), [36]

---

**Require:**  $d$ -order tensor  $T$  and prescribed accuracy  $\epsilon$ .

$C = \text{reshape}(T, [n_1, \frac{\text{number of elements of } C}{n_1}])$ .

$[U, \Sigma, V] = \text{SVD}_\delta(C)$ .

Put  $r_1 = \text{rank}(\Sigma)$ .

$G_1 = \text{permute}(\text{reshape}(U, [n_1, r_0, r_1]), [2, 1, 3])$ .

$C = \text{permute}(\text{reshape}(\Sigma V^T, [r_0, r_1, \prod_{j=2}^d n_j]), [2, 3, 1])$ .

Merge the last two indices by  $C = \text{reshape}(C, [r_1, \prod_{j=2}^{d-1} n_j, n_d r_0])$ .

**for**  $k = 2$  to  $d - 1$  **do** **do**

$C = \text{reshape}(C, [r_{k-1} n_k, \frac{\text{number of elements of } C}{r_{k-1} n_k}])$ .

$[U, S, V] = \text{SVD}_\delta(C)$ .

$r_k = \text{rank}(\Sigma)$ .

$G_k = \text{reshape}(U, [r_{k-1}, n_k, r_k])$ .

$C = \Sigma V^T$ .

**end for**

$G_d = \text{reshape}(C, [r_{d-1}, n_d, r_0])$ .

**return** tensor  $S$  in TR-format with cores  $G_1, \dots, G_d$ .

---

## Chapter 2

# Neural Networks and Deep Learning

### 2.1 Fundamentals of Neural Networks

#### 2.1.1 Preliminaries

Artificial neural networks (ANNs) are computational models inspired by the structure and function of the human central nervous system. They are composed of interconnected *neurons* that can process and transmit information. Similar to tensor decomposition, ANNs have a long and varied history, with early ideas and concepts dating back to the 1940s.

The first conceptualization of an artificial neural network is often attributed to McCulloch and Pitts, who published a paper in 1943 outlining the basic principles of a neural network [35]. They proposed that neurons in the brain could be modeled as simple logic gates, and that complex computations could be performed by arranging these neurons in specific patterns.

In the 1950s and 1960s, many researchers built upon this foundation and developed early prototypes of artificial neural networks. Some of the key figures in this field include Frank Rosenblatt, who developed the perception [43], a simple type of neural network capable of binary classification, and Bernard Widrow and Marcian Hoff, who developed the ADALINE (Adaptive Linear Element) and MADALINE (Multiple ADALINE) neural networks, which were capable of learning and adapting to new data [50].

Despite early successes, the field of ANNs faced several challenges and setbacks in the 1970s and 1980s. One of the main challenges was the lack of sufficient computing power to train large, complex neural networks. Another challenge was the lack of effective algorithms for training neural networks, which made it difficult to achieve good performance on real-world tasks [50].

In the late 1980s and early 1990s, advances in computing technology and the development of new training algorithms, such as backpropagation, helped to revitalize the field of ANNs. These advances made it possible to train large, complex neural networks on various tasks, including image and

speech recognition, natural language processing, and control systems [50].

Since then, ANNs have become an important tool in various fields, including computer science, engineering, and the natural sciences. They have been used to solve many problems, from simple tasks like recognizing handwritten digits to complex tasks like driving a car. Today, ANNs are a fundamental component of many modern artificial intelligence systems and continue to be an active area of research and development [12].

At a high level, an ANN consists of the following components:

- **Input layer:** The input layer receives input data and transfers it to the next layer. The size of the input layer (i.e., the number of neurons) is determined by the input data size. For example, if the input data consists of images with 28x28 pixels, then the input layer would have 784 neurons (one for each pixel).
- **Hidden layers:** The hidden layers are located between the input and output layers. They perform intermediate computations on the input data and pass the results on to the next layer. There can be any number of hidden layers in a neural network, and each layer can have any number of neurons.
- **Output layer:** The output layer produces the final result of the neural network's computations. The size of the output layer (i.e., the number of neurons) depends on the task the neural network tries to perform. For example, if the task is to classify images into 10 different categories, then the output layer would have 10 neurons (one for each category).

Each neuron in an ANN is connected to other neurons in the network through weights. These weights determine the strength of the connection between neurons and the extent to which one neuron's output influences another's input. During the training process, the neurons' weights are adjusted to optimize the neural network's performance on a given task.

In addition to the weights, each neuron has an activation function, which determines the output of the neuron given its input. The activation function can be a simple function, such as a binary threshold function or a sigmoid function, or a more complex function, such as a rectified linear unit (ReLU) or a hyperbolic tangent (tanh) function.

Finally, an artificial neural network also has an error function that measures the difference between the desired output and the actual output of the neural network. The error function guides the training process by indicating how well the neural network performs and how the weights should be adjusted to improve performance.

Throughout the following two chapters, we will examine three important references: *Understanding Machine Learning* by Shalev-Shwartz and Ben-David provides a rigorous and detailed treatment of the mathematical and statistical concepts that underlie machine learning algorithms [44]; *Deep Learning* by Goodfellow, Bengio, and Courville covers a wide range of topics in deep learning, from the basics of neural networks and supervised learning to more advanced topics such as unsupervised learning, reinforcement learning, and generative models [12]; *Linear Algebra and Learning from Data* by Strang provides a clear and intuitive explanation of how linear algebra concepts can be used to understand and solve problems in machine learning [46].

### 2.1.2 Definitions

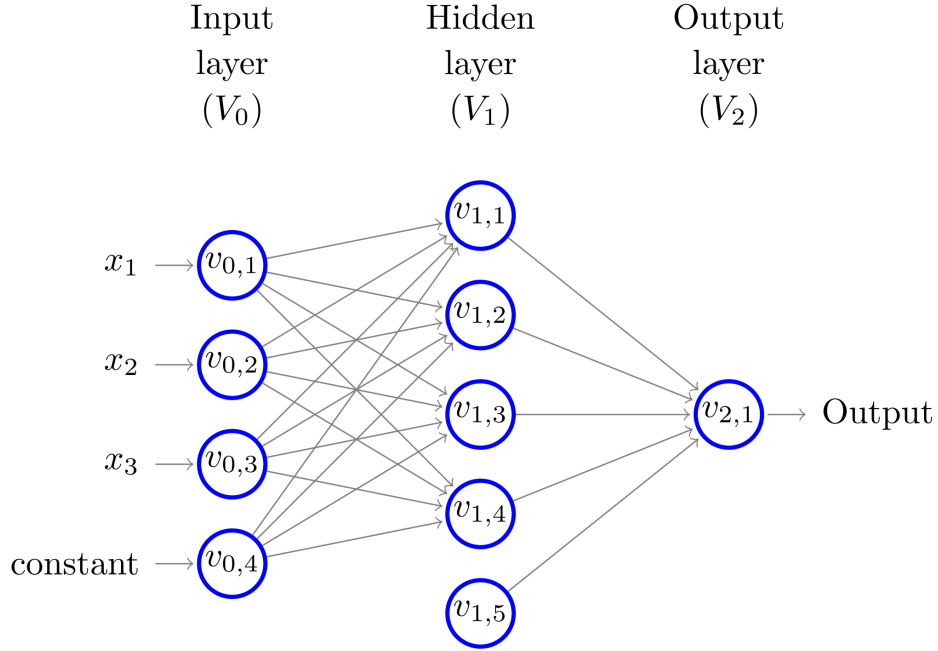


Figure 2.1: A feedforward neural network, [44].

**Definition 2.1.1. (Feedforward Neural Network, [44])** A *feedforward neural network* is a directed acyclic graph  $G = (V, E)$  with a weight function  $w : E \rightarrow \mathbb{R}$  over its edges. Every node in the graph corresponds to a

## 2.1. Fundamentals of Neural Networks

---

*neuron* and is modeled by a scalar function  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  called the *activation function*. Every edge connects the output of some neuron to the input of another. The input of a neuron is the weighted sum of all of the outputs of all of the neurons connected to it. Unless stated otherwise, the term *neural network* refers to a feedforward neural network. See Figure 2.1 to see a visual depiction of a neural network.

**Definition 2.1.2. (Neural Network Layers, [44])** A neural network is organized in *layers*. Each layer consists of the union of nonempty and disjoint subsets of nodes,  $V = \cup_{t=0}^T V_t$ , such that every edge in  $E$  connects some node in  $V_{t-1}$  to some node in  $V_t$ , for some  $t \in [T]$  where  $T$  denotes the *depth* or number of layers in the network (excluding  $V_0$ ). The size of the network is  $|V|$  while the width of the network is  $\max_t |V_t|$ . The layers of a neural network can be grouped into three sections: the *input layer*  $V_0$  with  $n + 1$  neurons where  $n$  is the dimensionality of the input space, *hidden layers*  $V_1, \dots, V_{T-1}$  where computation occurs, and the output layer  $V_T$ .

**Definition 2.1.3. (Neural Network Computation, [44])** Suppose we have a neural network with  $n + 1$  neurons in  $V_0$ . For every  $i \in [n]$ , the output of neuron  $i$  in  $V_0$  is simply  $x_i$ . The last neuron in  $V_0$  is the constant neuron, which always outputs 1. We denote by  $v_{t,i}$  the  $i$ th neuron of the  $t$ th layer and by  $o_{t,i}(x)$  the output of  $v_{t,i}$  when the network is fed with the input vector  $x$ . Therefore, for  $i \in [n]$  we have  $o_{0,i}(x) = x_i$  and for  $i = n + 1$  we have  $o_{0,i}(x) = 1$ . We now proceed with the calculation in a layer-by-layer manner. Suppose we have calculated the outputs of the neurons at layer  $t$ . Then, we can calculate the outputs of the neurons at layer  $t + 1$  as follows. Fix some  $v_{t+1,j} \in V_{t+1}$ . Let  $a_{t+1,j}(x)$  denote the input to  $v_{t+1,j}$  when the network is fed with the input vector  $x$ . Then,

$$a_{t+1,j}(x) = \sum_{r:(v_{t,r}, v_{t+1,j}) \in E} w((v_{t,r}, v_{t+1,j})) o_{t,r}(x),$$

and

$$o_{t+1,j}(x) = \sigma(a_{t+1,j}(x)).$$

## 2.2 Fundamentals of Deep Learning

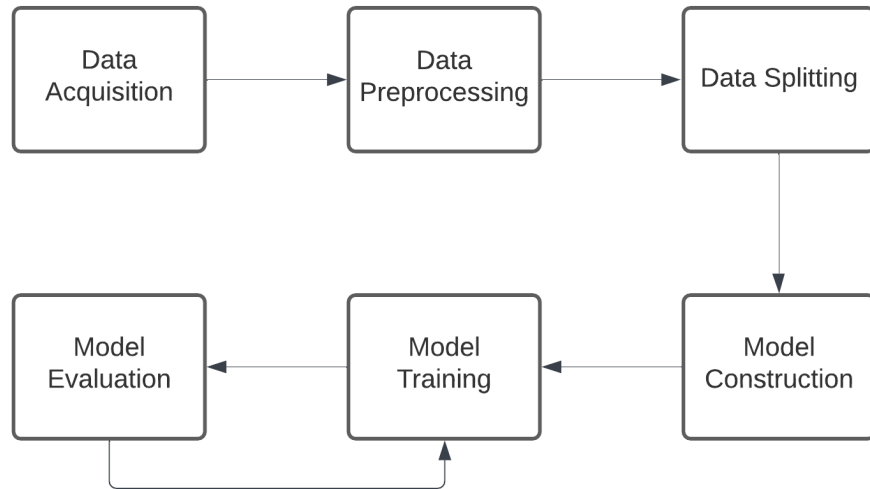


Figure 2.2: Standard deep learning workflow.

This section provides an overview of the deep learning workflow in an academic context based on a standard approach [12]. See Figure 2.2 for a visual workflow summary.

### 2.2.1 Data Acquisition

Data acquisition is an essential step in deep learning, as it involves collecting and preparing the data that will be used to train a deep learning model. Here are the main steps involved in data acquisition in deep learning:

1. Define the problem: The first step in data acquisition is to define the problem that the deep learning model will be used to solve. This will help determine the type and amount of data needed to train the model.
2. Identify sources of data: Various data sources are available for training a deep learning model, including public datasets, private datasets, and web scraping. Identifying the most relevant and reliable data sources for the problem at hand is important.



3. Collect the data: Once the data sources have been identified, the data must be collected. This can involve downloading datasets, scraping websites, or collecting data from other sources.
4. Store the data: Finally, the data should be stored in a secure and easily accessible location, such as a local drive or a cloud storage service. This will make it easy to access and use data during training.

### 2.2.2 Data Preprocessing

Once the dataset is stored, the data must be processed before training. Data preprocessing involves several tasks, including cleaning and formatting the data, splitting it into training and testing sets, and normalizing or standardizing it. Here are the main steps involved in data preprocessing in deep learning:

1. Data cleaning: Data cleaning involves identifying and correcting any errors, inconsistencies, and missing values in the data. This is important because these issues can affect the accuracy and effectiveness of the deep learning model.
2. Data formatting: Data formatting involves organizing the data in a way suitable for training a deep learning model. This can involve converting data to a numerical format, encoding categorical variables, and removing unnecessary columns.
3. Data normalization: Data normalization involves scaling the data so that all features have the same scale. Normalization may improve the performance of the deep learning model, as it can prevent certain features from dominating the training process.
4. Data standardization: Data standardization involves transforming the data with unit variance and zero mean. This can also help to improve the performance of the deep learning model, as it can ensure that all features are treated equally.

### 2.2.3 Data Splitting

Once the data has been appropriately preprocessed, it must be split into a training and a test set. Here are the main steps involved in data splitting in deep learning:

1. Determine the size of the training set: The first step in data splitting is to determine the size of the training set. This will depend on the size and complexity of the data, as well as the desired level of performance of the deep learning model. As a general rule, a larger training set will result in a more accurate model, but it will also require more computational resources to train.
2. Partition data into training and testing sets: Once the size of the training set has been determined, the data can be divided into separate training and testing sets. Many techniques exist to perform this task, such as random sampling, stratified sampling, or cross-validation.
3. Shuffle data: It is generally a good idea to shuffle the data before dividing it into training and testing sets. This can help to ensure that the data is randomly distributed across the two sets, which can improve the performance of the deep learning model.
4. Store the training and testing sets: Once the data has been divided into training and testing sets, it is important to store the sets in a secure and easily accessible location. This will make it easy to access and use data during training.

### 2.2.4 Model Construction

Model construction involves several steps, including selecting a neural network architecture, initializing the model's weights, and defining the training process. Here are the main steps involved in model construction in deep learning:

1. Select a neural network architecture: The first step in model construction is to select a neural network architecture suitable for the problem at hand. There are many different types of neural network architectures to choose from, including feedforward networks, convolutional neural networks, recurrent neural networks, and autoencoders, to name a few.
2. Initialize the model's weights: Once a neural network architecture has been selected, the next step is to initialize the model's weights. The weights of a deep learning model are the parameters learned during the training process, and they determine the model's behavior. Therefore, it is essential to initialize the weights carefully, as this can affect the model's performance.

3. Define the training process: A deep learning model's training process involves adjusting the model weights to optimize its performance on a given task. The training process typically involves several steps, including defining the loss function, selecting an optimizer, and setting the learning rate.
4. Compile the model: Once the training process has been defined, the model can be compiled. Compiling the model involves specifying the loss function and optimizer used during training and any additional metrics that will be used to evaluate the model's performance.

### 2.2.5 Model Training and Evaluation

Once we have constructed a model, it is time to start training and evaluation. Model training and evaluation involve many steps, including feeding the training data to the model, adjusting the weights based on the loss function and optimizer, and evaluating the model's performance. Here are the main steps involved in model training in deep learning:

1. Feed the training data to the model: The first step in model training is to feed the training data to the model. This involves passing the data through the model and computing the output of the model based on the current weights of the model.
2. Compute the loss: The next step is to compute the loss, which is a measure of the difference between the desired output and model output. The loss function is defined during the model compilation step and is used to guide the training process.
3. Adjust the weights: Once the loss has been computed, the model weights are adjusted to reduce the loss. This is done using an optimizer, defined during the model compilation step. The optimizer adjusts the weights based on the gradient of the loss function of the weights.
4. Evaluate the model's performance: After the weights have been adjusted, the model's performance is evaluated on the training data. Depending on the task, some metrics include accuracy, precision, and recall.
5. Repeat the process: The process of feeding the training data to the model, computing the loss, adjusting the weights, and evaluating the

model's performance is repeated until the model reaches a satisfactory level of performance on the training data.

## 2.3 Learning from Data

In this section, we provide an overview of how neural networks learn from data. We begin with the definition of a hypothesis class.

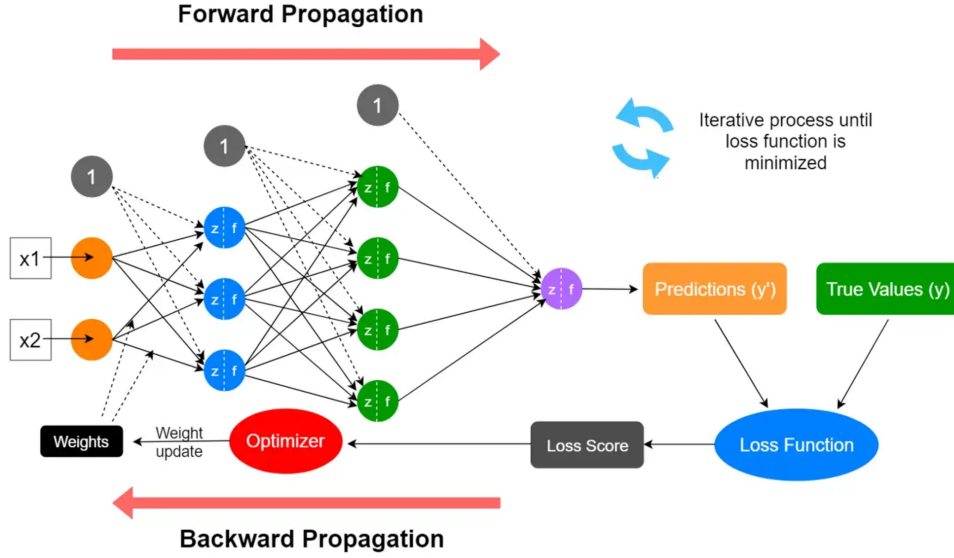


Figure 2.3: Neural network learning process, [42].

**Definition 2.3.1. (Hypothesis Class, [44])** Suppose we have a feed-forward neural network  $(V, E, \sigma, w)$  with function  $h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \rightarrow \mathbb{R}^{V_T}$ . We can fix the graph and activation function of the neural network to obtain the *architecture*  $(V, E, \sigma)$  of the network. We can define the *hypothesis class* for learning by the set of any such functions:

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w : E \rightarrow \mathbb{R}\}.$$

In summary, the weights over the edges of a neural network are the parameters specifying a hypothesis in the hypothesis class.

### 2.3.1 Neural Network Expressivity

Neural networks are extremely powerful tools for approximating functions. To show their expressiveness, we will begin by showing that every Boolean

function can be approximated using a very small neural network:

**Theorem 2.3.1.** (Theorem 20.1 in [44]) For every  $n$ , there exists a graph  $(V, E)$  of depth 2, such that  $\mathcal{H}_{V,E,sign}$  contains all functions from  $\{\pm 1\}^n$  to  $\{\pm 1\}$ .

However, we run into our first theoretical obstacle: the number of nodes in the hidden layer is exponentially large for such neural networks:

**Theorem 2.3.2.** (Theorem 20.2 in [44]) For every  $n$ , let  $s(n)$  be the minimal integer such that there exists a graph  $(V, E)$  with  $|V| = s(n)$  such that the hypothesis class  $\mathcal{H}_{V,E,sign}$  contains all the functions from  $\{0, 1\}^n$  to  $\{0, 1\}$ . Then,  $s(n)$  is exponential in  $n$ . Similar results hold for other activation functions.

Yet, we can show that a network of size  $O(T(n)^2)$  can approximate all Boolean functions in time  $O(T(n))$ :

**Theorem 2.3.3.** (Theorem 20.3 in [44]) Let  $T : \mathbb{N} \rightarrow \mathbb{N}$  and for every  $n$ , let  $\mathcal{F}_n$  be the set of functions that can be implemented using a Turing machine using runtime of at most  $T(n)$ . Then, there exist constants  $b, c\mathbb{R}_+$  such that for every  $n$ , there is a graph  $(V_n, E_n)$  of size at most  $cT(n)^2 + b$  such that  $\mathcal{H}_{V_n, E_n, sign}$  contains  $\mathcal{F}_n$ .

Moving beyond Boolean functions, we can show that neural networks are universal function approximators:

**Theorem 2.3.4.** (Theorem 20.5 in [44]) Fix some  $\epsilon \in (0, 1)$ . For every  $n$ , let  $s(n)$  be the minimal integer such that there exists a graph  $(V, E)$  with  $|V| = s(n)$  such that the hypothesis class  $\mathcal{H}_{V,E,\sigma}$ , with  $\sigma$  being the sigmoid function, can approximate, to within precision of  $\epsilon$ , every 1-Lipschitz function  $f : [-1, 1]^n \rightarrow [-1, 1]$ . Then,  $s(n)$  is exponential in  $n$ .

### 2.3.2 SGD and BP

Stochastic gradient descent (SGD) and backpropagation (BP) are the two fundamental algorithms used in learning via neural networks. We begin by defining the gradient of a differential function:

**Definition 2.3.2. (Gradient, [44])** The *gradient* of a differentiable function  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  at  $w$ , denoted  $\nabla f(w)$ , is the vector of partial derivatives of  $f$ , namely:

$$\nabla f(w) = \left( \frac{\partial f(w)}{\partial w[1]}, \dots, \frac{\partial f(w)}{\partial w[d]} \right).$$

### 2.3. Learning from Data

---

We want to show that gradient descent can be calculated for non-differentiable functions. To do this, we will show that gradient descent can be applied to the subgradient of  $f(w)$  at  $w^{(t)}$ . First, we characterize convexity as the existence of a tangent that lies below  $f$ :

**Lemma 2.3.5.** (Lemma 14.3 in [44]) Let  $S$  be an open convex set. A function  $f: S \rightarrow \mathbb{R}$  is convex if and only if for every  $w \in S$ , there exists  $v$  such that  $\forall u \in S$ :

$$f(u) \geq f(w) + \langle u - w, v \rangle.$$

From this lemma, we arrive at the definition of a subgradient:

**Definition 2.3.3. (Subgradient, [44])** A vector  $v$  that satisfies:

$$f(u) \geq f(w) + \langle u - w, v \rangle,$$

is called the *subgradient* of  $f$  at  $w$ . The set of subgradients of  $f$  at  $w$  is called the *differential set* and is denoted  $\partial f(w)$ .

Thus, gradient descent can be generalized to non-differentiable functions. From this, we move onward to the SGD algorithm, which minimizes the loss function of a neural network using noise:

---

**Algorithm 8** Stochastic Gradient Descent (SGD) for Minimizing  $f(w)$ , [44]

---

**Require:** scalar  $\eta > 0$ , integer  $T > 0$   
 $w^{(1)} = 0$   
**for**  $t = 1, 2, \dots, T$  **do**  
    Choose  $v_t$  at random from a distribution s.t.  $\mathbb{E}[v_t | w^{(t)}] \in \partial f(w^{(t)})$   
    Update  $w^{(t+1)} = w^{(t)} - \eta v_t$   
**end for**  
**return**  $\bar{w} = \frac{1}{T} \sum_{t=1}^T w^{(t)}$

---



---

**Algorithm 9** SGD for Neural Networks, [44]

---

**Require:** integer  $\tau$ , integers  $\eta_1, \dots, \eta_\tau$ , float  $\lambda > 0$ , layered graph  $(V, E)$ , differentiable activation function  $\sigma: \mathbb{R} \rightarrow \mathbb{R}$   
Choose  $w^{(1)} \in \mathbb{R}^{|E|}$  at random  
**for**  $i = 1, 2, \dots, \tau$  **do**  
    Sample  $(x, y) \sim \mathcal{D}$   
    Calculate gradient  $v_i = \text{backpropagation}(x, y, w, (V, E), \sigma)$   
    Update  $w^{(i+1)} = w^{(i)} - \eta_i(v_i + \lambda w^{(i)})$   
**end for**  
**return**  $\bar{w}$  is the best performing  $w^{(i)}$  on a validation set

---

### 2.3. Learning from Data

---

How do we calculate the gradient of a loss function? Backpropagation readily solves this issue:

---

**Algorithm 10** Backpropagation, [44]

---

**Require:** example  $(x, y)$ , weight vector  $w$ , layered graph  $(V, E)$ , activation function  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$

Denote layers of the graph  $V_0, \dots, v_T$  where  $V_t = \{v_{t,1}, \dots, v_{t,k_t}\}$

Define  $W_{t,i,j}$  as the weight of  $(v_{t,j}, v_{t+1,i})$

**Forward:**

Set  $o_0 = x$

**for**  $t = 1, 2, \dots, T$  **do**

**for**  $i = 1, 2, \dots, k_t$  **do**

        Set  $a_{t,i} = \sum_{j=1}^{k_{t-1}} W_{t-1,i,j} o_{t-1,j}$

        Set  $o_{t,i} = \sigma(a_{t,i})$

**end for**

**end for**

**Backward:**

Set  $\delta_T = o_T - y$

**for**  $t = T - 1, T - 2, \dots, 1$  **do**

**for**  $i = 1, 2, \dots, k_t$  **do**

$\delta_{t,i} = \sum_{j=1}^{k_{t+1}} W_{t,j,i} \delta_{t+1,j} \sigma'(a_{t+1,j})$

**end for**

**end for**

**return** For all edges  $(v_{t-1,j}, v_{t,i})$ , set the partial derivative to  $\delta_{t,i} \sigma'(a_{t,i}) o_{t-1,j}$

---

Figure 2.3 provides an excellent visual summary of the neural network learning process.

## Chapter 3

# Convolutional Neural Networks and Compression

### 3.1 Convolutional Neural Networks

A convolutional neural network, often referred to as CovNet or CNN, is a specific neural network architecture that is extremely useful for solving computer vision problems such as image and video recognition. In this section, we explore the mathematics that makes CNNs so special.





<i>Original</i>	<i>Gaussian Blur</i>	<i>Sharpen</i>	<i>Edge Detection</i>
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
			

Figure 3.1: Output of different types of convolutional kernels applied on an image, [6].

#### 3.1.1 Convolutions

We begin by slowly building up to the definition of convolution in the context of CNNs.

**Definition 3.1.1. (Toeplitz Matrix, [46])** We call an  $n \times m$  matrix  $M$  a *Toeplitz matrix* if it is banded shift-invariant, i.e.,  $A_{i,j} = A_{i+1,j+1}$ . For



example:

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & 0 & 0 & 0 \\ 0 & a_0 & a_1 & a_2 & 0 & 0 \\ 0 & 0 & a_0 & a_1 & a_2 & 0 \\ 0 & 0 & 0 & a_0 & a_1 & a_2 \end{bmatrix}.$$

**Definition 3.1.2. (Block Toeplitz, [46])** We call an  $n \times m$  matrix  $M$  *Block Toeplitz* if it can be partitioned into two or more square blocks or submatrices of which some are Toeplitz. For example:

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & 0 & 0 & 0 \\ 0 & a_0 & a_1 & a_2 & 0 & 0 \\ 0 & 0 & a_0 & a_1 & a_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

**Definition 3.1.3. (Kernel, [46])** In the context of image recognition, we call a (typically Block Toeplitz)  $E \times E$  matrix  $K$  a *kernel*. In the literature, the term *filter* is often used as a synonym for kernel; however, some authors use the term *kernel* only to describe a two-dimensional matrix and *filter* to describe a higher-dimensional tensor. The kernel matrix elements are called *weights* and are typically learned during the training process as defined in Chapter 2.

**Definition 3.1.4. (Kernel Window, [46])** The *kernel window* is the region of the input image  $W_{nm}$  that is covered by the kernel at any given time. As the kernel slides across the input image, it covers a different image window at each position. For example, let  $B$  be a  $4 \times 4$  image pixel matrix and  $K$  a  $3 \times 3$  kernel. The kernel window  $W_{22}$  is:

$$B = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}, W_{22} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

The number of pixels by which the kernel window moves each time it slides across the image is called the *stride*. To move the kernel across the entire image, the kernel is positioned at the top left corner of the image. Next, the convolution operation (defined below) is performed using the pixels in the kernel window. The kernel is then moved to the right by the stride number, and the convolution operation is performed again using the new kernel window. This process is repeated until the kernel has covered the entire image.

**Definition 3.1.5. (Convolution, [46])** Suppose we have an  $E \times E$  kernel  $K$  and an  $n \times m$  image pixel matrix  $B$ . We define the two-dimensional *convolution*  $A = K * B$  to output an  $(E + n - 1) \times (E + m - 1)$  matrix  $C$ :

$$A = K * B = C_{xy} = \sum_u \sum_v k_{uv} b_{x-u+1, y-v+1},$$

where  $u$  and  $v$  range over all legal subscripts for  $k_{uv}$  and  $b_{x-u+1, y-v+1}$ . See Figure 3.1 for examples of outputs of different kernels convolved with an image.

If one looks closely at the definition above, the subscripts suggest a 180-degree counter-clockwise kernel rotation to simplify the operation. In practice, a convolution in deep learning leads to an output of the same size as the kernel and is defined as  $\text{sum}(\text{dot}(B, K))$ , where  $B$  is an image pixel matrix, and  $K$  is a kernel. In other words, it is the sum of the dot product of column-wise vectors of the kernel window and the kernel as displayed in Figure 3.2.

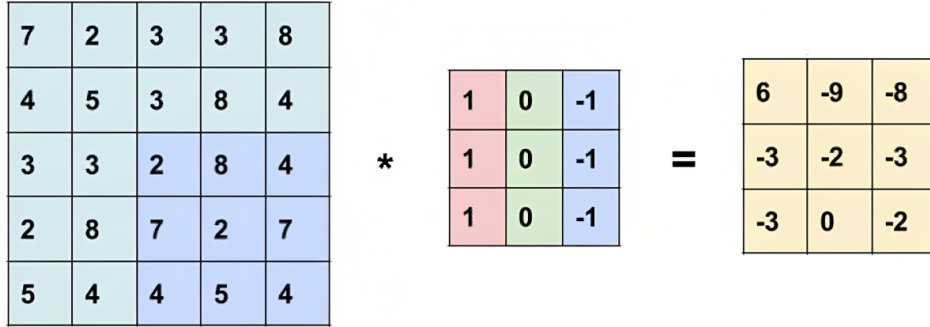


Figure 3.2: Example convolution of an image (left) with a kernel (right), [49].

### 3.1.2 Finding Edges

With a solid understanding of the algebra underpinning convolutions and filters, we can explore the many purposes filters serve in CNNs.

**Definition 3.1.6. (Smoothing, [46])** For a matrix  $f$ , we call a convolution with a Gaussian *smoothing*:

$$Gf(x, y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} * f.$$

For small  $\sigma^2$ , noise removal makes smoothing signal details clearer.

**Definition 3.1.7. (Sobel Operators for Gradient Detection, [46])** In the  $E = 3$  case, we call the following two matrices *Sobel operators* :

$$\frac{\partial}{\partial x} \approx \frac{1}{2} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \frac{\partial}{\partial y} \approx \frac{1}{2} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

The convolution of a Sobel operator with an image  $A$  produces another image  $G_x$  or  $G_y$  that contains either the horizontal or vertical derivative approximation, respectively. Sobel operators are used in gradient detection as a precursor to edge detection in image recognition tasks.

**Definition 3.1.8. (Laplacians of Gaussians for Edge Detection, [46])** We call the following filters the *Laplacians of Gaussians*:

$$Ef(x, y) = [\nabla^2 g(x, y)] * f(x, y),$$

where

$$\nabla^2 G = (x^2 + y^2 - 2\sigma^2)e^{-(x^2+y^2)/2\sigma^2}/\pi\sigma^4.$$

These filters are used in *Canny Edge Detection*.

**Definition 3.1.9. (Padding)** We define *padding* as the extension of the input layer for the input and output layers to be of equal dimension. Currently, the most popular padding approach is *zero-padding*, which simply sets all additional row and column components as zeros. Without padding, the spatial dimensions of the input image will typically shrink as the kernel slides across the image. This can be undesirable for certain image processing tasks, where the spatial dimensions of the input image are important. Adding padding to the borders of the input image makes it possible to preserve the spatial dimensions of the input image after the convolution operation has been applied.

#### 3.1.3 Deep CNNs

Now that all of the main mechanisms behind CNNs are explored, we can build up towards LeNet-5 - one of the first and simplest CNN implementations.

**Definition 3.1.10. (Input Channels)** Suppose we have an  $n \times m \times p$  image pixel tensor  $B$ . We refer to the size of  $p$  as the number of *input channels*.

For example, a grayscale image has only one channel, which encodes the intensity of each pixel. On the other hand, a color image typically has three channels: one for red, one for green, and one for blue. These channels are used to represent the different colors of the image. When applying a kernel to an input image, it is necessary to specify the number of input channels in the kernel. The kernel will be applied separately to each input channel, and the results will be combined to produce the output of the convolution operation.

**Theorem 3.1.1. (Karpathy’s Formula, [46])** Suppose we have an  $E \times E$  kernel  $K$ , an  $N \times N \times 1$  image pixel matrix  $B$ , stride  $S$ , and amount of zero padding  $P$ . Then, the size of the output  $O$  will be:

$$O = \frac{N - E + 2P}{S} + 1.$$

**Example 3.1.1. (LeNet-5, [29])** In general, a *deep CNN* is a deep neural network that has more than one filter or convolutional layer. As there are many variants of Deep CNNS, we will focus on the foundational LeNet-5 architecture proposed by LeCun et al in 1989. LeNet-5 is a simple 5-layer deep CNN that was created to solve the image recognition task of recognizing ten handwritten and machine-printed characters. The *input layer*  $I_0$  is a  $32 \times 32 \times 1$  image pixel matrix  $B$  with one input channel as it is grayscale. The *first convolutional layer* has 6 filters, each of size  $5 \times 5$ . It takes in input  $I_1 = I_0$  and returns output  $O_1$  of size  $28 \times 28 \times 6$ . Afterward, average pooling is applied to  $O_1$  resulting in a reduction in size by half without affecting the channel number. Thus,  $O_{1'}$  is  $14 \times 14 \times 6$ . The *second convolutional layer* has 16 filters, each of size  $5 \times 5$ . It takes in input  $I_2 = O_{1'}$  and outputs  $O_2$  which is  $10 \times 10 \times 16$ . Again, average pooling is applied to  $O_2$  resulting in  $O_{2'}$  of size  $5 \times 5 \times 16$ . The *third convolutional layer* has 120 filters, each of size  $5 \times 5$ . It takes in input  $I_3 = O_{2'}$  and outputs  $O_3$  of size  $1 \times 1 \times 120$ . Finally, we have a fully connected layer of 84 neurons followed by an output layer of 10 neurons representing the ten image classes. See Figure 3.3 for a diagram of the full LeNet-5 architecture.

It is important to note that the weights of the convolutional filters are updated using backpropagation. During training, the CNN is fed a batch of training data, and the output of the CNN is compared to the ground truth labels using a loss function. The gradient of the loss function with respect to the weights of the convolutional filters is then calculated using backpropagation. The weights are then updated in the opposite direction of the gradient, which helps to reduce the loss. This process is repeated for

### 3.2. Compression Methods

---

multiple epochs until the CNN has learned to classify the training data to the desired accuracy. It is also common to use weight initialization techniques and regularization techniques, such as dropout, to improve the training of the CNN and prevent overfitting.

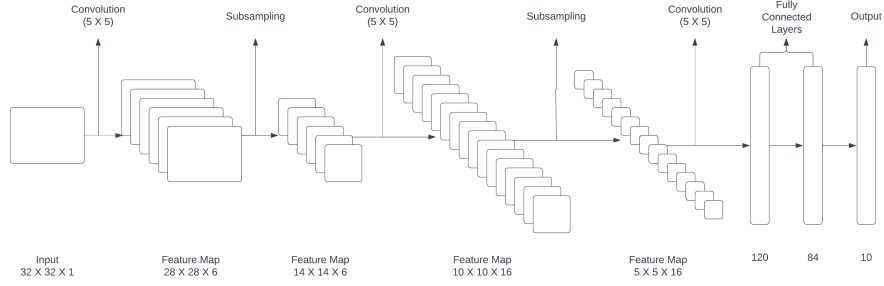


Figure 3.3: LeNet-5 architecture.

## 3.2 Compression Methods

The question of whether or not to compress CNNs is a natural one. From the point of view of a fully-connected feedforward neural network, a CNN is already a highly compressed model. Take the LeNet-5 architecture introduced in the previous section. The convolution from the input layer to the first feature map reduces the parameters to  $28 \cdot 28 \cdot (5 \cdot 5 + 1) \cdot 6 = 122,304$  at that stage of the computation, while a fully-connected layer would produce  $(32 \cdot 32 + 1) \cdot (28 \cdot 28) \cdot 6 = 4,821,600$  parameters. Unfortunately, most CNN architectures still involve fully-connected layers before producing an output, which massively increases the number of model parameters. As more model parameters increase hardware costs and decrease training speeds, industry and academia have come together to tackle the problem of how to compress CNNs further.

Many methods exist for compressing CNNs. Fortunately for those who wish to compare methods, most of the field has focused on two common values for assessing compression techniques: 1) percentage accuracy change from baseline, and 2) compression ratio (CR). Percentage accuracy change from baseline is simply the measure of the compressed CNN performance against its uncompressed counterpart (i.e., the baseline model) on some

### 3.2. Compression Methods

task(s) using a shared metric. We call a compressed model *lossless* if there is only a small decrease in percentage accuracy ( $\leq 1\%$ ). CR is a ratio calculated by dividing the number of parameters (i.e., weights) of the baseline model by the number of parameters of the compressed model. Other values, such as memory usage, power usage, and training speedup, are also seen across the literature. In this section, we first provide a table of commonly used benchmark datasets and models for CNN compression and then cover the three most common compression methods (excluding tensor decomposition-based methods) for CNNs: 1) precision reduction, 2) network pruning, and 3) compact architecture construction. Finally, we will highlight two key papers of interest for each method that capture the development of each technique since its emergence.

#### 3.2.1 Benchmarks

Dataset	Associated CNN Models
MNIST [8]	LeNet-5 [29], AlexNet [23]
ImageNet [7]	ResNet-18, ResNet-20, ResNet-32 [19]
CIFAR-10 [22]	ResNet-18, ResNet-20, ResNet-32
CIFAR-100 [22]	ResNet-18, ResNet-20, ResNet-32

Table 3.1: Common CNN compression benchmark datasets and models.

#### 3.2.2 Precision Reduction

**Definition 3.2.1. (Linear Quantization)** We call the conversion of weight representation from floating-point to fixed-point or dynamic fixed-point *linear quantization*.

Lotric and Bulic [34] were among the first research teams to demonstrate effective and efficient linear quantization of neural networks. Using an iterative logarithmic multiplier for fixed-point multiplication, Lotric and Bulic maintained an accuracy within 1% error of the baseline neural network model with more than 10% memory and 20% power reduction. In 2016, Gysel et al [13] presented Ristretto, a CNN approximation framework that could convert 32-bit floating point to 8-bit fixed point operations while maintaining less than 1% accuracy error from baseline.

**Definition 3.2.2. (Non-linear Quantization)** We call the non-uniform (typically logarithmic) encoding of a CNN *non-linear quantization*.

While linear quantization proved to be an effective initial compression scheme, neural network operations could not be encoded down any further than to 8-bit fixed point without suffering significant performance deterioration. Miyashita et al [37] introduced a scheme that uses non-uniform base-2 logarithmic encoding of CNN operations and weights that achieves similar levels of accuracy compared to baseline models while compressing the operations down to 5-bit fixed point. Zhou et al [52] created an incremental network quantization framework that maintained similar accuracy to baseline while compressing operations down to 3-bit fixed point, leading to a CR of 89x.

#### 3.2.3 Network Pruning

**Definition 3.2.3. (Unstructured Pruning)** We call the removal of redundant weights in a CNN *unstructured pruning*.

Han et al [14] were among the first to successfully compress CNNs by pruning unimportant connections. Without loss of accuracy, they achieved CRs of 9 to 13 on various models. Dettmers and Zettlemoyer [9] introduced sparse momentum, an algorithm capable of achieving lossless accuracy against baseline with a CR of up to 20x and a training speedup of over 5x on various CNN models.

**Definition 3.2.4. (Structured Pruning)** We call the removal of specific filters or channels in a CNN *structured pruning*. This is usually done by *Transfer Learning* - first pretraining a neural network on a large but distinct dataset and then fine-tuning it on the target dataset.

Through Transfer Learning, Molchanov et al [38] could remove entire feature maps of a CNN while retaining lossless accuracy. This method achieved a training speedup of over 5x on ImageNet. Lin et al [32] used a generative adversarial learning approach to prune both neurons and filters in an end-to-end approach achieving lossless accuracy on datasets such as MNIST, CIFAR-10, and ImageNet with training time speedup of over 3.7x and a CR from 2x to 9x.

#### 3.2.4 Compact Architecture Construction

If quantization and pruning are insufficient compression methods for the target IoT device due to memory, communication, or storage constraints,

constructing a new compact architecture is a common solution. Two of the most popular compact CNN architectures are SqueezeNet [18] and MobileNet [17]. SqueezeNet was constructed with autonomous vehicles in mind. On ImageNet, it achieves lossless accuracy as compared to AlexNet while having a CR of 50x and being 510x smaller in terms of storage. MobileNet is slightly larger than SqueezeNet in terms of parameters, yet it has less than 5% of SqueezeNet’s operations. MobileNet can also achieve lossless accuracy on numerous tasks, including object and face detection.

## 3.3 CNN Compression via Tensor Decomposition

In this section, we cover tensor decomposition-based CNN compression methods using the aforementioned four tensor decompositions: CP, Tucker, Tensor-Train, and Tensor-Ring. Similar to the previous section, we cover two key papers of interest for each tensor-based technique. Most, if not all, of the approaches covered in this section involve *tensorizing* CNNs: converting specific layers of a CNN into tensor format via a tensor decomposition to reduce the layer’s number of parameters. In particular, it is very useful to compress the fully-connected layers near the end of the CNN as they disproportionately increase the number of parameters in the entire network. While much more difficult and less impactful than fully-connected layers, Convolutional layers are still compressible.

### 3.3.1 CP Compression

A popular method for using CP decomposition to compress CNNs is to replace the CNN’s kernels with low-rank tensor approximations. Lebedev et al [28] were among the first to show promising results by decomposing kernels via non-linear least squares CP decomposition and fine-tuning via backpropagation. On ImageNet, their approach was able to speed up the training time by 4x with trivial decreases in accuracy. However, such approaches often suffer from diverging component degeneracy: the phenomena where at least two rank-1 tensors exist such that their Frobenius norms are high but cancel each other. This degeneracy commonly leads to a trade-off between reducing approximation error and increasing estimation stability. On large CNN models training on complex datasets such as ILSVRC-12 and CIFAR-100, Phan et al [41] achieved lossless accuracy with a CR of nearly 3x through their stable CP decomposition method.



### 3.3.2 Tucker Compression

Unlike CP decomposition which focuses on compressing kernel layers, one of the more successful CNN compression methods using Tucker decomposition is the compression of every layer in the network. Kim et al [20] were among the first to demonstrate the power of Tucker decomposition through their three-stage compression scheme that consists of 1) rank selection, 2) tucker decomposition on kernel tensor, and 3) fine-tuning. Their method achieved lossless accuracy with a CR of 5.46x on AlexNet, 7.40x on VGG-S, and 1.09x on VGG-16. Liu and Ng [33] provide a modern extension of Kim et al’s work and achieve lossless accuracy for ResNet-18 on CIFAR-10 with a CR and speedup of 11.82x and 5.48x, respectively.

### 3.3.3 Tensor-Train Compression

Garipov et al [11] were one of the first groups to achieve near lossless accuracy on the CIFAR-10 dataset: 1.1% accuracy drop with a CR of 80x. Their method consisted of 1) reshaping the 4-dimensional kernel of each convolutional layer, 2) performing TT-decomposition on each convolutional layer, and 3) tensorizing the fully-connected layers by storing the weight matrices in TT-format. Using a TT-SVD-based projection algorithm and an ADMM-regularized training procedure, Yin et al [51] were able to achieve lossless and even improved accuracy on several datasets including MNIST, CIFAR-10, CIFAR-100, and ImageNet with CRs ranging from 4.6x to 18x.

### 3.3.4 Tensor-Ring Compression

While TT-decomposition compression methods still achieve the best performance, a few promising TR-decomposition compression methods exist for CNNs. Aggarwal et al [1] demonstrated that a tensor ring net with compressed convolutional and fully-connected layers could achieve lossless accuracy on MNIST via LeNet-5 with a CR of 11x and a CR of 243x on CIFAR-10 using WideResNet28 with only a 2.3% decrease in accuracy. Li et al [30] were able to improve upon the previous work, demonstrating a CR of 141x with only a 1.9% decrease in accuracy on CIFAR-10 using WideResNet28.

## 3.4 PyTorch-TedNet Model Benchmarks

In this section, we first review the CNN tensorization methodology of Pan et al [39] for all four tensor decomposition types. Then, we verify the analyses

performed in the study on the CIFAR-10 and CIFAR-100 benchmarks. To verify the analyses, we begin by creating ResNet-32 using PyTorch [40]. Afterward, we train and test all four TedNet compressed models (CP, Tucker2, TT, and TR) on the benchmarks. All source codes and results are provided below.

### 3.4.1 TedNet Tensorized Layers

In this section, we only focus on tensorizing the fully-connected layers, which can be formulated as  $y = Wx$  although TedNet can compress convolutional layers. We can reformulate a fully-connected layer in a CNN as:

$$\mathbf{y}_{j_1, \dots, j_M} = \sum_{i_1, \dots, i_N=1}^{I_1, \dots, I_N} \mathbf{W}_{i_1, \dots, i_N, j_1, \dots, j_M} \mathbf{x}_{i_1, \dots, i_N}.$$

**Definition 3.4.1. (CP Layer, [39])** For a fully-connected layer in a CNN, the following is a tensorized representation called the *CP layer*:

$$\mathbf{y}_{j_1, \dots, j_M} = \sum_{i_1, \dots, i_N=1}^{I_1, \dots, I_N} \sum_{r=1}^R g_r a_{i_1, r}^{(1)} \dots a_{i_N, r}^{(N)} a_{j_1, r}^{(N+1)} \dots a_{j_M, r}^{(N+M)} \mathbf{x}_{i_1, \dots, i_N},$$

where  $R$  is the CP-rank.

**Definition 3.4.2. (Tucker2 Layer, [39])** For a fully-connected layer in a CNN, the following is a tensorized representation called the *Tucker2 layer*:

$$\begin{aligned} \mathbf{y}_{j_1, \dots, j_M} &= \star, \\ \star &= \sum_{i_1, \dots, i_N=1}^{I_1, \dots, I_N} \sum_{r_1=1}^{R_1} \dots \sum_{r_N=1}^{R_N} g_{r_1, \dots, r_N} a_{i_1, r_1}^{(1)} \dots a_{i_N, r_N}^{(N)} a_{j_1, r_1}^{(N+1)} \dots a_{j_M, r_N}^{(N+M)} \mathbf{x}_{i_1, \dots, i_N}, \end{aligned}$$

where  $R_1, \dots, R_N$  can be different ranks.

**Definition 3.4.3. (TT Layer, [39])** For a fully-connected layer in a CNN, the following is a tensorized representation called the *TT layer*:

$$\begin{aligned} \mathbf{y}_{j_1, \dots, j_M} &= \star, \\ \star &= \sum_{i_1, \dots, i_N=1}^{I_1, \dots, I_N} \sum_{r_1, \dots, r_N=1}^{R_1, \dots, R_N} g_{i_1, j_1, r_1}^{(1)} \dots g_{i_N, j_N, r_N}^{(N)} \mathbf{x}_{i_1, \dots, i_N}, \end{aligned}$$

where  $R_1, \dots, R_{N-1}$  are the TT-ranks.

### 3.4. PyTorch-TedNet Model Benchmarks

---

**Definition 3.4.4. (TR Layer, [39])** For a fully-connected layer in a CNN, the following is a tensorized representation called the *TR layer*:

$$\mathbf{y}_{j_1, \dots, j_M} = \star,$$

$$\star = \sum_{i_1, \dots, i_{N+M}=1}^{I_1, \dots, I_{N+M}} \sum_{r_0, \dots, r_{N+M}-1}^{R_0, \dots, R_{N+M}-1} g_{r_0, i_1, r_1}^{(1)} \cdots g_{r_N, j_1, r_{N+1}}^{(N+1)} \cdots g_{r_{N+M-1}, j_M, r_0}^{(N+M)} \mathbf{x}_{i_1, \dots, i_N}$$

where  $R_0, \dots, R_N$  are the TR-ranks.

#### 3.4.2 ResNet-32

We begin by creating the ResNet-32 model using PyTorch.

---

##### Program 1 ResNet-32 - Import libraries

---

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 import torch.nn.init as init
5 from torch.autograd import Variable

```

---

### 3.4. PyTorch-TedNet Model Benchmarks

---

#### Program 2 ResNet-32 - Model, [19]

---

```
1 def ResNet32():
2     def _weights_init(m):
3         classname = m.__class__.__name__
4         #print(classname)
5         if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
6             init.kaiming_normal_(m.weight)
7     class LambdaLayer(nn.Module):
8         def __init__(self, lambd):
9             super(LambdaLayer, self).__init__()
10            self.lambd = lambd
11        def forward(self, x):
12            return self.lambd(x)
13    class BasicBlock(nn.Module):
14        expansion = 1
15        def __init__(self, in_planes, planes, stride=1, option='A'):
16            super(BasicBlock, self).__init__()
17            self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
18            self.bn1 = nn.BatchNorm2d(planes)
19            self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)
20            self.bn2 = nn.BatchNorm2d(planes)
21            self.shortcut = nn.Sequential()
22            if stride != 1 or in_planes != planes:
23                self.shortcut = LambdaLayer(lambda x:
24                    F.pad(x[:, :, ::2, ::2], (0, 0, 0, 0, planes//4, planes//4), "constant", 0))
25        def forward(self, x):
26            out = F.relu(self.bn1(self.conv1(x)))
27            out = self.bn2(self.conv2(out))
28            out += self.shortcut(x)
29            out = F.relu(out)
30            return out
31    class ResNet(nn.Module):
32        def __init__(self, block, num_blocks, num_classes=100):
33            super(ResNet, self).__init__()
34            self.in_planes = 16
35            self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1, bias=False)
36            self.bn1 = nn.BatchNorm2d(16)
37            self.layer1 = self._make_layer(block, 16, num_blocks[0], stride=1)
38            self.layer2 = self._make_layer(block, 32, num_blocks[1], stride=2)
39            self.layer3 = self._make_layer(block, 64, num_blocks[2], stride=2)
40            self.linear = nn.Linear(64, num_classes)
41            self.apply(_weights_init)
42        def _make_layer(self, block, planes, num_blocks, stride):
43            strides = [stride] + [1]*(num_blocks-1)
44            layers = []
45            for stride in strides:
46                layers.append(block(self.in_planes, planes, stride))
47            self.in_planes = planes * block.expansion
48            return nn.Sequential(*layers)
49        def forward(self, x):
50            out = F.relu(self.bn1(self.conv1(x)))
51            out = self.layer1(out)
52            out = self.layer2(out)
53            out = self.layer3(out)
54            out = F.avg_pool2d(out, out.size()[3])
55            out = out.view(out.size(0), -1)
56            out = self.linear(out)
57            return out
58    return ResNet(BasicBlock, [5, 5, 5])
```

---

### 3.4.3 CIFAR-10 Benchmark

In Table 3.2, we see that only the TedNet TT-tensorized ResNet-32 model achieves lossless performance.

	Max Accuracy	Rank	Parameters	CR
ResNet-32	81	-	0.46M	1x
CP	63	10	0.026M	18x
TK2	68	3	0.012M	40x
TT	80	3	0.012M	40x
TR	76	3	0.010M	44x

Table 3.2: CIFAR-10 benchmark results (50 Epochs, Single GPU).

---

#### Program 3 Import libraries

---

```

1  # Import libraries
2
3  from managpu import GpuManager
4  import random
5  import tednet
6  import tednet.tnn.cp
7  import tednet.tnn.tucker2
8  import tednet.tnn.tensor_ring
9  import tednet.tnn.tensor_train
10 import numpy as np
11 import torch
12 import torch.nn as nn
13 import torch.nn.functional as F
14 import torch.optim as optim
15 from torchvision import datasets, transforms

```

---

### 3.4. PyTorch-TedNet Model Benchmarks

---

#### Program 4 Configure GPU

---

```
1  # Configure GPU
2
3  my_gpu = GpuManager()
4  my_gpu.set_by_memory(1)
5  use_cuda = torch.cuda.is_available()
6  device = torch.device("cuda" if use_cuda else "cpu")
7  seed = 123
8  random.seed(seed)
9  np.random.seed(seed)
10 torch.manual_seed(seed)
11 if use_cuda:
12     torch.cuda.manual_seed_all(seed)
13     torch.backends.cudnn.benchmark = True
14     torch.backends.cudnn.deterministic = True
```

---

---

#### Program 5 CIFAR-10 - Load dataset, [22]

---

```
1  # Load dataset
2
3  kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
4  train_loader = torch.utils.data.DataLoader(
5      datasets.CIFAR10('./data', train=True, download=True,
6                      transform=transforms.Compose([
7                          transforms.ToTensor(),
8                          transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))
9                      ])),
10     batch_size=128, shuffle=True, **kwargs)
11 test_loader = torch.utils.data.DataLoader(
12     datasets.CIFAR10('./data', train=False, transform=transforms.Compose([
13         transforms.ToTensor(),
14         transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))
15     ])),
16     batch_size=128, shuffle=True, **kwargs)
```

---

### 3.4. PyTorch-TedNet Model Benchmarks

---

#### Program 6 CIFAR-10 - Define train and test processes

---

```
1  # Define train and test processes
2
3  def train(model, device, train_loader, optimizer, epoch, log_interval=200):
4      model.train()
5      for batch_idx, (data, target) in enumerate(train_loader):
6          data, target = data.to(device), target.to(device)
7          optimizer.zero_grad()
8          output = model(data)
9          loss = F.cross_entropy(output, target)
10         loss.backward()
11         optimizer.step()
12         if batch_idx % log_interval == 0:
13             print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
14                 epoch, batch_idx * len(data), len(train_loader.dataset),
15                 100. * batch_idx / len(train_loader), loss.item()))
16
17 def test(model, device, test_loader):
18     model.eval()
19     test_loss = 0
20     correct = 0
21     with torch.no_grad():
22         for data, target in test_loader:
23             data, target = data.to(device), target.to(device)
24             output = model(data)
25             test_loss += F.cross_entropy(output, target, reduction='sum').item() # sum up batch loss
26             pred = output.argmax(dim=1, keepdim=True) # get the index of the max log-probability
27             correct += pred.eq(target.view_as(pred)).sum().item()
28
29     test_loss /= len(test_loader.dataset)
30
31     print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
32         test_loss, correct, len(test_loader.dataset),
33         100. * correct / len(test_loader.dataset)))
```

---

---

#### Program 7 CIFAR-10 - Define model

---

```
1  # Define model
2
3  model = resnet32()
4  ## model = tednet.tnn.cp.CPResNet32([10, 10, 10, 10, 10, 10, 10],10)
5  ## model = tednet.tnn.tucker2.TK2ResNet32([3, 3, 3, 3, 3, 3, 3],10)
6  ## model = tednet.tnn.tensor_train.TTResNet32([3, 3, 3, 3, 3, 3, 3],10)
7  ## model = tednet.tnn.tensor_ring.TRResNet32([3, 3, 3, 3, 3, 3, 3],10)
8  model.to(device)
9  optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9, weight_decay=0.0001)
```

---

### 3.4. PyTorch-TedNet Model Benchmarks

---

#### Program 8 CIFAR-10 - Train and evaluate model, [40, 39]

---

```

1  # Begin training
2
3  for epoch in range(20):
4      train(model, device, train_loader, optimizer, epoch)
5      test(model, device, test_loader)

```

---

#### 3.4.4 CIFAR-100 Benchmark

In Table 3.3, we see that neither the TedNet TT-tensorized ResNet-32 or TedNet TR-tensorized ResNet-32 models achieve lossless performance.

	Max Accuracy	Rank	Parameters	CR
ResNet-32	54	-	0.47M	1x
TT	48	6	0.038M	13x
TR	43	6	0.035M	13x

Table 3.3: CIFAR-100 benchmark results (50 Epochs, Single GPU).

---

#### Program 9 CIFAR-100 - Load dataset [22]

---

```

1  # Load dataset
2
3  kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
4  train_loader = torch.utils.data.DataLoader(
5      datasets.CIFAR100('./data', train=True, download=True,
6          transform=transforms.Compose([
7              transforms.ToTensor(),
8              transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))
9          ])),
10     batch_size=128, shuffle=True, **kwargs)
11  test_loader = torch.utils.data.DataLoader(
12      datasets.CIFAR100('./data', train=False, transform=transforms.Compose([
13          transforms.ToTensor(),
14          transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))
15      ])),
16     batch_size=128, shuffle=True, **kwargs)

```

---



### 3.4. PyTorch-TedNet Model Benchmarks

---

#### Program 10 CIFAR-100 - Define train and test processes

---

```
1  # Define train and test processes
2
3  def train(model, device, train_loader, optimizer, epoch, log_interval=200):
4      model.train()
5      for batch_idx, (data, target) in enumerate(train_loader):
6          data, target = data.to(device), target.to(device)
7          optimizer.zero_grad()
8          output = model(data)
9          loss = F.cross_entropy(output, target)
10         loss.backward()
11         optimizer.step()
12         lr=optimizer.param_groups[0]["lr"]
13         if batch_idx % log_interval == 0:
14             print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
15                 epoch, batch_idx * len(data), len(train_loader.dataset),
16                 100. * batch_idx / len(train_loader), loss.item()))
17
18  def test(model, device, test_loader):
19      model.eval()
20      test_loss = 0
21      correct = 0
22      with torch.no_grad():
23          for data, target in test_loader:
24              data, target = data.to(device), target.to(device)
25              output = model(data)
26              test_loss += F.cross_entropy(output, target, reduction='sum').item() # sum up batch loss
27              pred = output.argmax(dim=1, keepdim=True) # get the index of the max log-probability
28              correct += pred.eq(target.view_as(pred)).sum().item()
29
30      test_loss /= len(test_loader.dataset)
31
32      print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
33          test_loss, correct, len(test_loader.dataset),
34          100. * correct / len(test_loader.dataset)))
```

---

---

#### Program 11 CIFAR-100 - Define model

---

```
1  # Define model
2
3  model = ResNet32()
4  ## model = tednet.tnn.tensor_train.TTResNet32([6, 6, 6, 6, 6, 6],100)
5  ## model = tednet.tnn.tensor_ring.TTResNet32([6, 6, 6, 6, 6, 6],100)
6  model.to(device)
7  optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9, weight_decay=5e-4)
```

---

### 3.4. PyTorch-TedNet Model Benchmarks

---

---

**Program 12** CIFAR-100 - Train and evaluate model, [40, 39]

---

```
1  # Begin training
2
3  scheduler = torch.optim.lr_scheduler.StepLR(optimizer,5, gamma=0.2)
4
5  epochs=20
6  lrs=[]
7  for epoch in range(1,epochs+1):
8      train(model, device, train_loader, optimizer, epoch)
9      test(model, device, test_loader)
10     scheduler.step()
```

---

#### 3.4.5 Conclusion

As demonstrated in the CIFAR-10 and CIFAR-100 benchmark results, tensor decomposition can drastically decrease the number of model parameters of large CNNs such as ResNet-32 while maintaining similar performance. In particular, the TT-tensorized ResNet-32 performed the best on both benchmarks against all other compressed models. One of the main benefits of TT-decomposition for compressing CNNs is that it can effectively capture the low-rank structure of convolutional filters, typically used in CNNs to extract features from images. Convolutional filters often have a low-rank structure because they only need to capture a small number of patterns or features in an image, rather than the entire image itself. By using the TT-decomposition to approximate these filters, it is possible to represent them using fewer parameters, leading to a more compact and efficient CNN. Additionally, the TT-decomposition can be efficiently computed using algorithms such as alternating least squares, making it practical for real-world applications. Moreover, it can provide good reconstruction performance, meaning that the original tensor can be accurately reconstructed from the decomposed cores. This is important for preserving the accuracy of the CNN when compressing it. Finally, TT decomposition is also flexible in that it can be applied to a wide range of tensor sizes and dimensions, making it suitable for compressing CNNs with different architectures.

There are several benefits to performing CNN compression via tensor decomposition. Tensor decomposition can significantly reduce the number of parameters in a CNN, leading to a smaller model size. This is particularly useful for deploying CNNs on devices with limited storage or reducing the transmission time when transferring the model over the network. Furthermore, tensor decomposition can also improve the efficiency of a CNN by reducing the number of calculations required to process a given input.

This can lead to faster inference times and lower power consumption, making the CNN more suitable for real-time applications. In some cases, tensor decomposition can also improve the generalization performance of a CNN, meaning that it can perform better on unseen data. This is because the decomposition process can remove redundant or irrelevant information from the model, leading to a more compact and efficient data representation.

Several directions exist for future research in the area of CNN compression via tensor decomposition:

1. Developing more efficient algorithms: One area of focus could be developing more efficient algorithms for computing tensor decompositions, such as the TT-decomposition. This could involve finding ways to reduce the computational complexity of the algorithms or developing new algorithms that are better suited for compressing CNNs.
2. Improving the accuracy of compressed models: Another area of focus could be improving the accuracy of compressed CNNs. This could involve developing methods for selecting the most important parameters to keep during the compression process or finding ways to preserve more of the information in the original model.
3. Investigating the generalization performance of compressed models: Researchers could also study the generalization performance of compressed CNNs, which refers to how well the model performs on unseen data. This could involve evaluating the trade-off between model compression and generalization performance to determine the optimal compression level for a given task.

# Bibliography

- [1] V. Aggarwal, W. Wang, B. Eriksson, Y. Sun, and W. Wang. Wide compression: Tensor ring nets. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9329–9338, 2018.
- [2] A. Anandkumar, R. Ge, and M. Janzamin. Analyzing tensor power method dynamics in overcomplete regime. *J. Mach. Learn. Res.*, 18(1):752–791, 2017.
- [3] J. D. Carroll and J. J. Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of “eckart-young” decomposition. *Psychometrika*, 35(3):283–319, 1970.
- [4] R. B. Cattell. Parallel proportional profiles” and other principles for determining the choice of factors by rotation. *Psychometrika*, 9:267–283, 1944.
- [5] L. Chiantini and G. Ottaviani. On generic identifiability of 3-tensors of small rank. *SIAM Journal on Matrix Analysis and Applications*, 33(3):1018–1037, 2012.
- [6] Pratik Choudhari. Understanding ”convolution” operations in cnn, May 2020.
- [7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.
- [8] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.
- [9] T. Dettmers and L. Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *ArXiv*, abs/1907.04840, 2019.

- [10] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1:211–218, 1936.
- [11] T. Garipov, D. Podoprikin, A. Novikov, and D. Vetrov. Ultimate tensorization: compressing convolutional and fc layers alike. *ArXiv*, 2016.
- [12] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. <http://www.deeplearningbook.org>.
- [13] P. Gysel, M. Motamedi, and S. Ghiasi. Hardware-oriented approximation of convolutional neural networks. *ArXiv*, abs/1604.03168, 2016.
- [14] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural network. *ArXiv*, abs/1506.02626, 2015.
- [15] A. Harshman. Foundations of the parafac procedure: Models and conditions for an “explanatory” multi-modal factor analysis. *UCLA Working Papers in Phonetics*, 16:1–84, 1970.
- [16] F. L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Studies in Applied Mathematics*, (1-4):164–189, 1927.
- [17] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017.
- [18] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. H., W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and 1mb model size. *ArXiv*, abs/1602.07360, 2016.
- [19] H. Kaiming, Z. Xiangyu, R. Shaoqing, and S. Jian. Deep residual learning for image recognition. *ArXiv*, 2015.
- [20] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *ArXiv*, 2015.
- [21] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.
- [22] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

- [23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.
- [24] J.B. Kruskal. Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics. *Linear Algebra and its Applications*, 2:95–138, 1977.
- [25] J. M. Landsberg. *Tensors: Geometry and Applications*. Graduate Studies in Mathematics, 2012.
- [26] L. De Lathauwer, B. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.
- [27] L. De Lathauwer, B. De Moor, and J. Vandewalle. On the best rank-1 and rank-( $r_1, r_2, \dots, r_n$ ) approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1324–1342, 2000.
- [28] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempit-sky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *ArXiv*, 2014.
- [29] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput*, 1(4):541–551, 1989.
- [30] N. Li, Y. Pan, Y. Chen, Z. Ding, D. Zhao, and Z. Xu. Towards efficient tensor decomposition-based dnn model compression with optimization framework. *ArXiv*, 2020.
- [31] T. Lickteig. Rank and optimal computation of generic tensors. *Linear Algebra and its Applications*, 69:95–120, 1985.
- [32] S. Lin, R. J., C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. S. Doermann. Towards optimal structured cnn pruning via generative adversarial learning. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2785–2794, 2019.
- [33] Y. Liu and M. K. Ng. Deep neural network compression by tucker decomposition with nonlinear response. *Knowledge-Based Systems*, 241, 2022.

- [34] U. Lotrič and P. Bulić. Applicability of approximate multipliers in hardware neural networks. *Neurocomputing*, 96:57–65, 2012. Adaptive and Natural Computing Algorithms.
- [35] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.
- [36] O. Mickelin and S. Karaman. On algorithms for and computing with the tensor ring decomposition. *Numer Linear Algebra Appl*, 27(3), 2020.
- [37] D. Miyashita, H. Lee E, and B. Murmann. Convolutional neural networks using logarithmic data representation. *ArXiv*, abs/1603.01025, 2016.
- [38] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *ArXiv*, abs/1611.06440, 2016.
- [39] Y. Pan, M. Wang, and Z. Xu. Tednet: A pytorch toolkit for tensor decomposition networks. *Neurocomputing*, 469:234–238, 2022.
- [40] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. *ArXiv*, 2019.
- [41] A. Phan, K. Sobolev, K. Sozykin, D. Ermilov, J. Gusak, P. Tichavsky, V. Glukhov, I. Oseledets, and A. Cichocki. Stable low-rank tensor decomposition for compression of convolutional neural network. *ArXiv*, 2020.
- [42] Rukshan Pramoditha. Overview of a neural network’s learning process, Feb 2022.
- [43] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.
- [44] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

- [45] N.D. Sidiropoulos and R. Bro. On the uniqueness of multilinear decomposition of n-way arrays. *Journal of Chemometrics*, 14:229–239, 2000.
- [46] G. Strang. *Linear Algebra and Learning from Data*. Wellesley-Cambridge Press, 2019.
- [47] V. Strassen. Rank and optimal computation of generic tensors. *Linear Algebra and its Applications*, 52-53:645–685, 1983.
- [48] A. Vijayaraghavan. Efficient tensor decomposition. *ArXiv*, 2020.
- [49] Rijul Vohra. Convolutional neural networks, Sep 2019.
- [50] B. Widrow and M.A. Lehr. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.
- [51] M. Yin, Y. Sui, S. Liao, and B. Yuan. Towards efficient tensor decomposition-based dnn model compression with optimization framework. *ArXiv*, 2021.
- [52] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *ArXiv*, abs/1702.03044, 2017.