UNIVERSITAT DE BARCELONA
Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

SEARCH OF OPTIMAL LOW-RANK APPROXIMATIONS USING TENSOR NETWORKS

Autor: Aran Roig

Director: Dr. Nahuel Statuto
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, June 7, 2025

Abstract

Tensor network structure search has been interesting research topic since the raise on complexity of deep learning models and quantum mechanics. This Bachelor Thesis main goal is to give an automated search of an optimal tensor network structure that gives the best low-rank approximation of a given tensor.

For these purpose we first give an introduction to some basic tensor algebra, we present tensor networks and tensor network states. Then we present algorithms that find the cores of a tensor network that better represent an objective tensor. We also present an algorithm for finding optimal tensor network structures that can guarantee that we will find the most optimal cores.

Finally, we prove that these algorithms converge under certain assumptions and then we perform some practical experiments that empirically proves that it is possible to find other more optimized low-rank decompositions of tensors without significant losses on performance and accuracy.

Resum

La recerca de l'estructura òptima de xarxes de tensors ha estat un tema d'interès des de l'augment en la complexitat dels models d'aprenentatge profund i de la mecànica quàntica. Aquest treball de final de grau té com a objectiu principal oferir una cerca automatitzada d'una estructura òptima d'una xarxa de tensors que representi millor un tensor amb una aproximació de baix rank.

Per aconseguir aquest propòsit, primer oferim una introducció als tensors, a les xarxes tensorials. Aleshores presentem algoritmes que troben els nuclis de la xarxa tensorial que millor representen un tensor objectiu. També presentem un algoritme per trobar quina estructura de xarxa tensorial ens pot garantir que poguem trobar els nuclis més optims.

Finalment demostrem que aquests algoritmes convergeixen sota algunes suposicions i després fem uns quants experiments que empíricament mostren que és possible trobar altres descomposicions de baix rang de tensors sense que suposin una pèrdua significant en rendiment i precissió.

2020 Mathematics Subject Classification. 11G05, 11G10, 14G10

Agraïments

Vull agrair a ...

Contents

Chapter 1

Introduction

Over the last years, high-dimensional data has become ubiquitous in modern science and engineering, arising in areas such as quantum many-body physics and machine learning. However, as the dimensionality of the data increases, so does the burden of the increase of computational and storage cost often referred as the curse of dimensionality. To address this challenge, low-rank approximation techniques have emerged as powerful tools, enabling the efficient representation, compression, and manipulation of large-scale tensors. Recent research has proven promising results in the capabilities of compression of tensor networks. Qing et. al. for example see empirically that tensor networks can be very useful for neural network parameter compression [1] or even image recovering [2]

Sometimes, the low-rank approximation of a given tensor is not guaranteed to be represented from a fixed tensor network structure. There has been a lot of recent research on solving the problem on finding the optimal structures. On [3] Li et. al. present an algorithm that aims to find an optimal tensor network permutation. We will see in more detail the TnALE (Tensor network Alternating Local Enumeration) algorithm presented by Li. et. al on [4]. The presented algorithm aims to solve the rank search of the core tensors and the optimal permutation. We will extend this algorithm for also solving the structure search of tensor networks.

The concept of tensor networks appeared from the study of many-body quantum systems [5] and recently has attracted significant interest on machine learning. Foundamental research on tensor networks started in 1971 by the paper "Applications of negative dimensional tensors". In which he introduced the Penrose Notation for representing contractions of various tensors, and how this diagrammatic approach could be used in various applications in physics [6]. Forwarding to recent years, on 2014, Román Orus [7] and later in 2019 presented the first idea of applying them in machine learning [5].

The main goal of this thesis will be to compress tensors by getting a low-rank approximation of them by finding an approximate representation of a given tensor network state. We will present two different algorithms for achieving this goal: the TnALS algorithm and backpropagation. We will compare these two algorithm and we will discuss their use cases. Then, we will present the TnALE algorihtm and we will prove its convergence presenting beforehand zero gradient theory.

Finally, we will do some experiments with these algorithms: we will perform lossy compression on images and also we will compress the parameters of a simple neural network.

1.1 Thesis structure

The second chapter of the thesis will focus on the preliminaries about tensor algebra. We will introduce tensors from a mathematical standpoint presenting the tensor product space. We will see that the tensor product space forms vector space and then we will describe some operations between tensors.

After that, we will also describe some reshaping operations that can be applied to tensors to transform its orders i.e their number of dimensions. These reshaping operations will be useful on the following chapters since we will need them for presenting the TnALS algorithm and also for the last chapter of applications, when we transform matrices into tensors and then apply all the theory that we have presented.

On the third chapter we will present the basics of the Penrose Notation and we will give a formal definition to tensor networks and tensor network states (TNS). We will give some common examples of tensor networks and after that we will define the notion of the rank of a tensor related to a TNS, called G-rank. We will see that the G-rank can be a lot smaller than the traditional tensor rank as it is discussed in [8], proving that there exists better compression methods than the canonical polyadic decomposition, which we will see on the same chapter.

On the fourth chapter we will describe the main algorithms of this thesis. First, we will give two algorithms that fixed a TN, finds a low-rank decomposition of smaller tensors that can represent our original tensor with a certain error. These algorithms are the tensor network alternating least squares (TnALE) and we will also solve this problem using backpropagation.

Then, we will unfix the tensor structure and we will present the TnALE algorithm which finds the most optimal tensor network structure for representing a tensor T by evaluating a loss function that depends on the structure itself. The advantage of TnALE respect other algorithms is that it also saves some evaluations of some tensor network structure candidates. Lastly we will prove that TnALE converges to an optimal structure, and also we will prove the fact of skipping certain evaluations, all of this under some given assumptions.

Finally on the fifth chapter we will apply all of these algorithms for practical cases such as compressing images, compressing the weight matrices of fully connected neural networks.

Chapter 2

Tensors

In this chapter we will construct tensors in a formal way and we will lay down the basics of tensor algebra. We will define the notion of rank of a tensor and then, we will describe some basic tensor reshaping operations. All of this will be crucial for then presenting tensor contractions and tensor networks on the following chapter.

2.1 The tensor product space

From now on, we will denote $\mathbb{V}_1, \ldots, \mathbb{V}_n$ as finite vector spaces over a field $\mathbb{K}$ ($\mathbb{R}$ if unspecified) of dimension $\dim \mathbb{V}_i = N_i \; \forall i = 1, \ldots, n$.

We will now present the definition of a tensor. One of the most simple definitions for presenting a tensor are multilinear maps:

Definition 2.1. *A **multilinear map** is a mapping $T : \mathbb{V}_1 \times \cdots \times \mathbb{V}_n \to \mathbb{K}$ which satisfies:*

1. $T(v_1, \ldots, \lambda v_i, \ldots, v_n) = \lambda \cdot T(v_1, \ldots, v_i, \ldots, v_n)$
2. $T(v_1, \ldots, v_i + u, \ldots, v_n) = T(v_1, \ldots, v_i, \ldots, v_n) + T(v_1, \ldots, u, \ldots, v_n)$
   $\forall i = 1, \ldots, n, \; u \in \mathbb{V}_i, \lambda \in \mathbb{K}$

We will call multilinear maps tensors. The order of a tensor will be defined as the value of n. For example, kinear maps and bilinear maps are specific cases of tensors with $n = 1$ and $n = 2$ respectively.

This definition gives a clear intuition about what is a tensor, but it doesn't extract all of the properties that tensor have. For example, following this definition it is not obvious that all multilinear maps $T : \mathbb{V}_1 \times \cdots \mathbb{V}_n \to \mathbb{K}$ form a vector space. The next thing that we are going is transform this definition so that we can define tensors as an element of a given vector space. We will achieve this define by defining a relation between all the vectors in the free vector space over the cartesian product of $\mathbb{V}_1 \times \cdots \times \mathbb{V}_n$ that capture the properties of multilinear maps.

The first ingredient that we will need is defining what are the free vector spaces over a set:

Definition 2.2. *Let S be a set and $\mathbb{K}$ a field. We denote the free vector space over S as $\mathbb{K}[S]$, which is the vector space containing all linear combinations of elements from S*

with coefficients of $\mathbb{K}$*, i.e:*

$$\mathbb{K}[S] = \left\{ \sum_{i=1}^{n} a_i s_i \mid s_i \in S, n \in \mathbb{Z}_+, a_i \in \mathbb{K} \right\}$$

Example 2.3. If we take $\mathbb{K} = \mathbb{R}$ and $S = (x, y)$, then the elements of $\mathbb{R}[S]$ have the form $ax + by$ with $a, b \in \mathbb{R}$. We can clearly see that $\mathbb{R}[S]$ is a vector space of dimension 2 over the field $\mathbb{R}$ and therefore isomorphic to $\mathbb{R}^2$

Now we will present the relation of each element of the free vector space of $\mathbb{V}_1 \times \cdots \times \mathbb{V}_n$ and multilinear maps.

Let $\mathbb{L} = \mathbb{K}[\mathbb{V}_1 \times \cdots \times \mathbb{V}_n]$. We define relation $\sim_R$ as the smallest equivalence relation that satisfies:

$$(v_1, \ldots, \alpha v_i, \ldots, v_n) \sim_R \alpha(v_1, \ldots, v_n) \; \forall i = 1, \ldots, n, \forall \alpha \in \mathbb{K}$$

$$(v_1, \ldots, v_i + u_i, \ldots, v_n) \sim_R (v_1, \ldots, v_i \ldots, v_n) + (v_1, \ldots, u_i, \ldots, v_n) \; \forall i = 1, \ldots, n$$

Proposition 2.4. $[0]_R$ *is a subspace of* $\mathbb{K}[\mathbb{V}_1 \times \cdots \mathbb{V}_n]$

Proof. $0 \in [0]_R$. It is sufficient to see that for all $\lambda \in \mathbb{K}$ and $u, v \in [0]_R$ then $u + \lambda v \in [0]_R$. We can write $u + \lambda v \sim_R (u_1 + v_1, \ldots, u_i + \lambda v_i, \ldots, u_n + v_n) \sim_R (u_1, \ldots, u_n) + (v_1, \ldots, \lambda v_i, \ldots, v_n) \sim_R 0 + \lambda 0 \sim_R 0$ and therefore $u + \lambda v \sim_R 0$ since R is an equivalence relation. □

Because of this proposition, the quotient space $\mathbb{L}/[0]_R$ is well defined.

Definition 2.5. *The* ***tensor product space*** $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ *is defined as the quotient* $\mathbb{L}/[0]_R$*. Each element of the tensor product space is the image of the quotient mapping* $\varphi : \mathbb{L} \to \mathbb{L}/[0]_R$ *defined by sending each element* $(v_1, \ldots, v_n) \in \mathbb{L}$ *to its equivalence class that we will write as* $v_1 \otimes \cdots \otimes v_n$*.*

We will give an small example of a tensor product space:

Example 2.6. Given $\mathbb{R}^2$ and $\mathbb{R}^3$ with its canoncial vector space structures, $\mathbb{R}^2 \otimes \mathbb{R}^3$ is defined by the equivalence classes that follow the relations

$$(\alpha u, v) \sim \alpha(u, v) \sim (u, \alpha v) \quad \forall \alpha \in \mathbb{K}, \; u \in \mathbb{R}^2, v \in \mathbb{R}^3$$

$$(u + u', v) \sim (u, v) + (u' + v) \quad \forall u, u' \in \mathbb{R}^2, v \in \mathbb{R}^3$$

$$(u, v + v') \sim (u, v) + (u + v') \quad \forall u \in \mathbb{R}^2, v, v' \in \mathbb{R}^3$$

And an element of $\mathbb{R}^2 \otimes \mathbb{R}^3$ would be the representant of $[((1, 2, 3), (0, 4))]$ in which for example other representants of the same equivalence class would be $2 \cdot ((1, 2, 3), (0, 2))$ or $((1, 0, 3), (0, 2)) + ((0, 2, 0), (0, 2))$

We will now state and prove the universal property of the tensor product:

Theorem 2.7 (Universal property of the tensor product)**.** *Let φ be the quotient mapping from $\mathbb{V}_1 \times \cdots \times \mathbb{V}_n$ to $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$. For every multilinear map $h : \mathbb{V}_1 \times \cdots \times \mathbb{V}_n \to X$ where X is any vector space there exists an unique linear map $\tilde{h} : \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n \to X$ such that the following diagram commutes:*

$$\begin{array}{ccc} \mathbb{V}_1 \times \cdots \times \mathbb{V}_1 & \xrightarrow{\varphi} & \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n \\ & \searrow^{h} & \downarrow \tilde{h} \\ & & X \end{array}$$

Proof. Let $\varphi(v_1, \ldots, v_n) := [(v_1, \ldots, v_n)] \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$. Let $h : \mathbb{V}_1 \times \cdots \mathbb{V}_n \to X$ be a multilinear map. We define $\tilde{H} : \mathbb{K}[\mathbb{V}_1 \times \cdots \times \mathbb{V}_n] \to X$ by:

$$\tilde{H}\left(\sum_{i=1}^{p} a_i(v_i^1, \ldots, v_i^n)\right) := \sum_{i=1}^{p} a_i h(v_i^1, \ldots, v_i^n)$$

Consider now the vector subspace $[0]_R$ of $\mathbb{K}[\mathbb{V}_1 \times \cdots \times \mathbb{V}_n]$. Since h is multilinear, we can see that $\tilde{H}$ sends every element of $[0]_R$ to $0 \in X$, therefore $W \subseteq \ker \tilde{H}$ and hence $\tilde{H}$ induces a well-defined linear map $\tilde{h} : \mathbb{K}[\mathbb{V}_1 \times \cdots \times \mathbb{V}_n]/R = \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n \to X$ that satisfies $\tilde{h}(v_1 \otimes \cdots \otimes v_n) = h(v_1, \ldots, v_n)$

Suppose that exists another mapping $f : \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n \to X$ such that $f(v_1 \otimes \cdots \otimes v_n) = h(v_1, \ldots, v_n)$, then we would have

$$f\left(\sum_{i=1}^{p} a_i v_i^1 \otimes \cdots \otimes v_i^n\right) = \sum_{i=1}^{p} a_i h(v_i^1, \ldots, v_i^n) = \tilde{h}\left(\sum_{i=1}^{n} a_i v_i^1 \otimes \cdots \otimes v_i^n\right)$$

therefore, the linear mapping $\tilde{h}$ is unique □

An interesting observation of the universal property of tensor products is that if we take any multilinear map $T : \mathbb{V}_1 \times \cdots \times \mathbb{V}_n \to X$ and we fix X to be the field $\mathbb{K}$, then there exists an unique linear map $\tilde{T} : \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n \to \mathbb{K}$ which is an element of the dual space $(\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n)^*$ which is isomorphic to $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$, proving that the multilinear maps that we have defined at Definition 2.1 are isomorphic to the elements of the tensor product space $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$

Also, thanks to this theorem we explicitly construct the vector space $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ and a basis of it:

Proposition 2.8. *Let $\{e_1^i, e_2^i, \ldots, e_{N_i}^i\}$ be basis for each $\mathbb{V}_i$ and $N_i = \dim \mathbb{V}_i$. Then the set*

$$\mathcal{B}_\otimes = \{e_{i_1}^1 \otimes \cdots \otimes e_{i_n}^n = [(e_{i_1}^1, \ldots, e_{i_n}^n)]_R : 1 \leqslant i_j \leqslant N_j, 1 \leqslant j \leqslant n\}$$

is a basis of $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$.

Proof. If we have $v_1 \otimes \cdots \otimes v_n \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$, if we write $v_i = \sum_{j=1}^{N_i} \lambda_j^i e_j^i$ then, because of the multilinearity of R we get that

$$v_1 \otimes \cdots \otimes v_n = \left(\sum_{i=1}^{N_1} \lambda_1^i e_1^i\right) \otimes \cdots \otimes \left(\sum_{i=1}^{N_n} \lambda_n^i e_n^i\right) = \sum_{s_1, \ldots, s_n}^{N_1, \ldots, N_n} \lambda_1^{s_1} \cdots \lambda_n^{s_n} (e_1^{s_1} \otimes \cdots \otimes e_n^{s_n})$$

And since all elements of $\mathbb{V}_1 \otimes \mathbb{V}_n$ can be written in this form, $\mathcal{B}_\otimes$ spans the entire space. For proving the independance of each element of $\mathcal{B}_\otimes$, suppose that we have a linear combination such that

$$\sum_{s_1,\dots,s_n}^{N_1,\dots,N_n} \lambda_{s_1,\dots,s_n} e^1_{s_1} \otimes \cdots \otimes e^n_{s_n} = 0 \tag{2.1.1}$$

Let $\{{}^*e^i_1, \dots, {}^*e^i_n\}$ be the dual basis for $\mathbb{V}^i$ such that ${}^*e^i_j(e^i_k) = \delta_{jk}$. We define now the multilinear map

$$\begin{aligned} f_{(k_1,\dots,k_n)} : \mathbb{V}_1 \times \cdots \times \mathbb{V}_n &\longrightarrow \mathbb{K} \\ (v_1, \dots, v_n) &\longmapsto {}^*e^1_{k_1}(v_1) \cdot {}^*e^2_{k_2}(v_2) \cdots {}^*e^n_{k_n}(v_n) \end{aligned}$$

With $1 \leqslant k_i \leqslant N_i$. The image of $f_{(k_1,\dots,k_n)}$ extracts the coefficient $\lambda_{k_1,\dots,k_n}$ of any tensor $v_1 \otimes \cdots \otimes v_n$.

Applying the universal property of the tensor product, there exists an unique linear map $\tilde{f}_{(s_1,\dots,s_n)} : \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n \to \mathbb{K}$ such that

$$\tilde{f}_{(k_1,\dots,k_n)}(e^1_{j_1} \otimes \cdots \otimes e^n_{j_n}) = f_{(k_1,\dots,k_n)}(e^1_{j_1}, \dots, e^n_{j_n}) = \delta_{j_1 k_1} \cdot \delta_{j_2 k_2} \cdots \delta_{j_n k_n}$$

If we now apply the linerar combination to $\tilde{f}_{(s_1,\dots,s_n)}$ we get

$$\tilde{f}_{(k_1,\dots,k_n)} \left(\sum_{s_1,\dots,s_n}^{N_1,\dots,N_n} \lambda_{s_1,\dots,s_n} e^1_{s_1} \otimes \cdots \otimes e^n_{s_n} \right) = \sum_{s_1,\dots,s_n}^{N_1,\dots,N_n} \lambda_{s_1,\dots,s_n} \cdot \delta_{k_1 s_1} \cdot \delta_{k_2 s_2} \cdots \delta_{k_n s_n} = \lambda_{k_1,\dots,k_n}$$

But from (2.1.1) $\lambda_{k_1,\dots,k_n} = 0$ for all $1 \leqslant k_i \leqslant N_i$. Therefore, the elements of $\mathcal{B}_\otimes$ are independent and form a basis. □

From now on we can directly work with the tensor space as if it was a regular vector space with a well defined basis. The dimension of $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ is $N_1 \cdot N_2 \cdots N_n$ and its elements can be expressed as

$$T = \sum_{s_1,\dots,s_n}^{N_1,\dots,N_n} T_{s_1,\dots,s_n} \cdot e^1_{s_1} \otimes \cdots \otimes e^n_{s_n} \tag{2.1.2}$$

Using the basis that we defined in the last proposition. If we wanted to store a tensor in a computer program (we will do it on the following chapters) it would be enough to save all the $T_{s_1,\dots,s_n}$ entries since it is an element of a vector space with a well defined basis.

Definition 2.9. *We will define the **size** of the tensor $T \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ as* $\text{Size}(T) = N_1 N_2 \cdots N_n$. *We will also say that the **order** of a tensor T is n, which is the same as the order of its respective multilinear map $T : \mathbb{V}_1 \times \cdots \times \mathbb{V}_n \to \mathbb{K}$.*

We can now start presenting basic tensor algebra. Adding and multiplying by an scalar is already defined by the underlying vector space. We will define the tensor product between two tensors of different tensor product spaces. We will need this operation later because when we define tensor networks, we will need to contract tensors of different orders.

Suppose that $T \in \mathbb{V}_1 \otimes \cdots \mathbb{V}_n$ and $U \in \mathbb{W}_1 \otimes \cdots \otimes \mathbb{W}_m$. We want that the tensor product operation results in a tensor $T \otimes U \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n \otimes \mathbb{W}_1 \otimes \cdots \otimes \mathbb{W}_m$.

Definition 2.10 (Tensor product)**.** *With the above notation, let* $\dim \mathbb{V}_i = N_i$, $\dim \mathbb{W}_j = M_j$ *and some basis* $\{e_1^i, \dots, e_{N_i}^i\}$ *of each* $\mathbb{V}_i$ *and* $\{p_1^j, \dots, p_{M_j}^j\}$ *of each* $\mathbb{W}_j$*, we define the tensor product* $T \otimes U$ *as*

$$T \otimes U = \sum_{i_1,\dots,i_n}^{N_1,\dots,N_n} \sum_{j_1,\dots,j_m}^{M_1,\dots,M_m} T_{i_1,\dots,i_n} U_{j_1,\dots,j_m} \cdot e_{i_1}^1 \otimes \cdots \otimes e_{i_n}^n \otimes p_{j_1}^1 \otimes \cdots \otimes p_{j_m}^m \tag{2.1.3}$$

Which naturally is an element of $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n \otimes \mathbb{W}_1 \otimes \cdots \otimes \mathbb{W}_m$ because we are creating a linear combination of elements of the basis of that space. If we do the tensor product for two tensors of order 1 (which are isomorphic to vectors), this operation is called the **outer product**, so we can state that the tensor product is a generalization of it.

Example 2.11. Let $\{a_1, a_2\} \subset \mathbb{V}_1$, $\{b_1, b_2\} \subset \mathbb{V}_2$ and $\{c_1, c_2\} \subset \mathbb{W}_1$, $\{d_1, d_2\} \subset \mathbb{W}_2$ be basis of their corresponding vector spaces. Let $T \in \mathbb{V}_1 \otimes \mathbb{V}_2$ and $U \in \mathbb{W}_1 \otimes \mathbb{W}_2$ defined as:

$$T = 2(a_1 \otimes b_1) + 3(a_2 \otimes b_1) \qquad U = c_1 \otimes d_1 + c_2 \otimes d_2$$

Then, the tensor product $T \otimes U$ would be:

$$T \otimes U = 2(a_1 \otimes b_1 \otimes c_1 \otimes d_1) + 2(a_1 \otimes b_1 \otimes c_2 \otimes d_2) +$$
$$3(a_2 \otimes b_1 \otimes c_1 \otimes d_1) + 3(a_2 \otimes b_1 \otimes c_2 \otimes d_2)$$

We will proceed to introduce a tensor norm since in the following chapters we will want to know if a tensor is "small" or "big", and also but not less important, we want to know if two tensors are near each other to test convergence for algorithms involving tensor networks. In our case we will stick to the Frobenius norm:

Definition 2.12. *We define the frobenius norm as:*

$$\|\cdot\|_F : \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n \longrightarrow \mathbb{R}_+$$
$$\left(\sum_{s_1,\dots,s_n}^{N_1,\dots,N_n} T_{s_1,\dots,s_n} \cdot e_{s_1}^1 \otimes \cdots \otimes e_{s_n}^n \right) \longmapsto \sqrt{\sum_{s_1,\dots,s_n}^{N_1,\dots,N_n} T_{s_1,\dots,s_n}^2}$$

Now, we will define another important operation of tensors that will be an important pillar on decomposing tensors with high dimensionality: the rank of a tensor or the traditional rank. It will serve as the extension of the matrix rank, which can be defined as the dimension of the vector space spanned by all the vectors on its columns. For the matrix rank, we say that if it is spanned by a single vector, the rank is 1. We will extend this concept and we will say that a tensor has rank 1 if it is spanned by a single element of a basis. In other words, a tensor t is a rank-1 tensor if it can be written as

$$t = \lambda v^1 \otimes \cdots \otimes v^n$$

with $v^i \in \mathbb{V}_i$ and $\lambda \in \mathbb{K}$. Therefore, the rank of an tensor T will be r if it can be spanned from r rank-1 tensors. This gives us the following definition:

Definition 2.13. *We say that a tensor* T *has rank* r *as* $\operatorname{rank} T = r$ *with* $r \in \mathbb{Z}_+$ *if* r *is the minimum value such that we can write* T *as the following form*

$$T = \sum_{p=1}^{r} \lambda_p v_p^1 \otimes \cdots \otimes v_p^n \tag{2.1.4}$$

where $v_1^i, \dots, v_r^i \in \mathbb{V}_i, i = 1, \dots, n$ *and* $\lambda_p \in \mathbb{K}$

The rank of a tensor is bounded by $\prod_{i=1}^{n} N_i$ since we can decompose every tensor as the sum of the elements of a basis of the tensor product space, and so the rank of the tensor does not exceed the number of elements of the basis.

Unlike matrices, determining the rank of a tensor is an NP-hard problem (See Section 8 of [9]). Finding the maximum rank, i.e determining $\max_{T \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n} \operatorname{rank} T$ still remains an unresolved problem. [10]

Decomposing a tensor T in rank-1 tensors as in eq. (2.1.4) is known as **tensor rank decomposition**. One could ask if given a tensor T and fixed r, can we construct a tensor T' of rank t such that $\|T - T'\|_F$ is minimum. The decomposition of T' is called **canonical polyadic decomposition** and it is an special case of a tensor network that we will see on the following section.

2.2 Reshaping operations

In this section we will introduce reshaping operations for tensors with the object to try to extend some numeric algorithms for matrices and to tensors by reshaping the tensors into matrices, doing operations with the matrix-shaped tensors and then recover back the original tensor shape. This will be later use for example when we introduce the alternating least squares algorithms for tensor networks.

Any tensor $T \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ can be identified as an n-dimensional array. In other words, for each tensor T we can define a discrete function $\mathcal{T}$ that encodes the representation in a basis of the tensors T as:

$$\begin{aligned} \mathcal{T} : \prod_{i=1}^{n} \{1, \ldots, N_i\} &\longrightarrow \mathbb{K} \\ (i_1, \ldots, i_n) &\longmapsto T_{i_1, \ldots, i_n} \end{aligned}$$

From now on we will identify the set of all images of $\mathcal{T}$ as an element of $\mathbb{K}^{N_1 \times \cdots \times N_n}$. We will often write a tensor $T \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ as an element $T \in \mathbb{K}^{N_1 \times \cdots \times N_n}$ with $\dim \mathbb{V}_i = N_i$. Sometimes we will write $T(i_1, \ldots, i_n)$ as the image of $\mathcal{T}$ of $(i_1, \ldots, i_n)$. Since now we can see a tensor as an n-dimensional array thanks to the mapping $\mathcal{T}$, we can start reshaping tensors. But before that, we will define what is a mode of a tensor:

Definition 2.14. *We define the j-th of a tensor as its j-th dimension. A tensor of order n has n different modes.*

For example, having a 3-order tensor $T \in \mathbb{R}^{N_1 \times N_2 \times N_3}$. We can write each entry of the tensor as $T(i_1, i_2, i_3)$. The first mode of T is i_1, the second one, i_2 and the third i_3. In other words, when we say the j-mode of a tensor we are referring at one argument of the discrete function $\mathcal{T}$

Now we will introduce the linearization operation which simplify the notation a lot when we define reshaping operations.

Definition 2.15 (Linearization)**.** *Fixed $N_1, \ldots, N_n \in \mathbb{Z}_+$, given $i_1, \ldots, i_n \in \mathbb{Z}_+$ such that $1 \leqslant i_1, \leqslant N_1, \ldots, 1 \leqslant i_n \leqslant N_n$, we define the **linearization** of the indices $i_1, \ldots, i_n$ as*

the mapping $\prod_{i=1}^{n}\{1,\dots,N_i\} \to \{1,\dots,\prod_{i=1}^{n} N_i\}$ *and with its images defined as:*

$$\overline{i_1, i_2, \dots, i_n} = \sum_{j=2}^{n} \left((i_j - 1) \prod_{k=1}^{j} N_k \right) + i_1$$

The purpose of the linearization mapping is to give a bijection within each element of the form $(i_1,\dots,i_n) \in \prod_{i=1}^{n}\{1,\dots,N_i\}$ to a positive natural number. For example, consider $n = 3$ and $N_1 = N_2 = N_3 = 3$. The encoding of the tuple $(1,1,1)$ corresponds to 1, the tuple $(2,1,1)$ to 2, the tuple $(1,2,3)$ to $1+(2-1)\cdot 3+(3-1)\cdot 3 = 10$. One should keep in mind that when defining an array in computer science, usually the first element starts at the index 0, and our linearization operation starts with the indices at 1. Throughout the thesis we will stick to array indices starting at 1 for consistency, but by doing a change of variables $i'_j = i_j - 1$ before applying the linearization and then subtracting 1 also on the image, we will get the same operation but for arrays starting at 0.

We will now define the vectorization operation, which reshapes a tensor $T \in \mathbb{K}^{N_1 \times N_2 \times \cdots \times N_n}$ to a vector of $\mathbb{K}^{N_1 \cdot N_2 \cdots N_n}$

Definition 2.16. *We define the* ***vectorization*** *of* T *as the first order tensor (or vector)* $\mathcal{V} \in \mathbb{K}^{N_1 N_2 \cdots N_n}$ *defined entrywise as*

$$\mathcal{V}(\overline{i_1 i_2 \dots i_n}) = T(i_1, i_2, \dots, i_n)$$

We will write the vectorization of T *as* $\operatorname{vec} T$

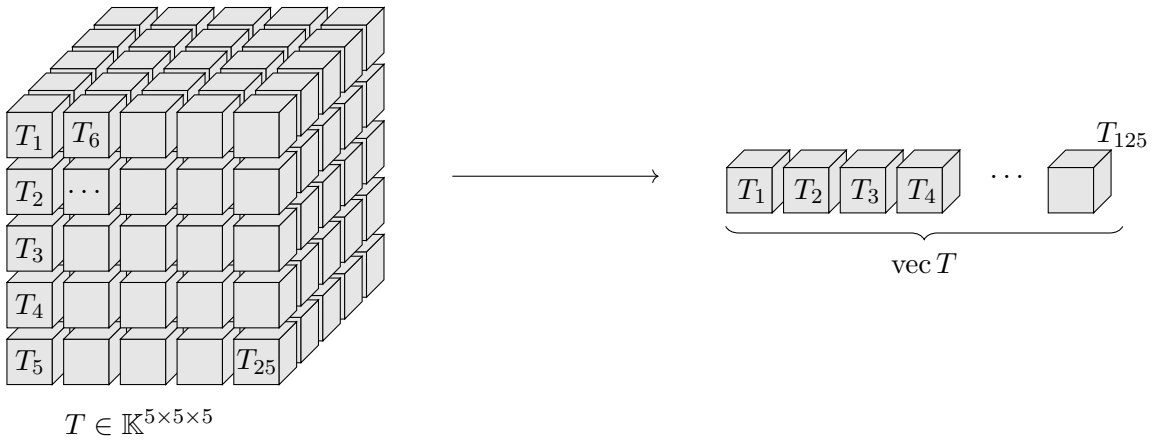


Figure 2.1: Representation of tensor vectorization. Source: own elaboration

We will also define the permutation of two modes of a tensor. That means swapping arguments of the function $\mathcal{T}$ following some permutation $\sigma \in S_n$.

Definition 2.17 (Tensor permutation)**.** *Given a permutation* $\sigma \in S_n$*, and a tensor* $T \in \mathbb{K}^{N_1 \times N_2 \times \cdots \times N_n}$ *we define* $T_\sigma \in \mathbb{K}^{N_{\sigma(1)} \times N_{\sigma(2)} \times \cdots \times N_{\sigma(n)}}$ *entrywise as*

$$T_\sigma(i_1,\dots,i_n) = T(i_{\sigma^{-1}(1)}, i_{\sigma^{-1}(2)}, \dots, i_{\sigma^{-1}(n)})$$

Example 2.18. Let $N_1, N_2 \in \mathbb{Z}_+$ and $M \in \mathbb{K}^{N_1 \times N_2}$. The matrix transposition M^T can be written as $M_{(2,1)}$

Example 2.19. Suppose that we have $X \in \mathbb{R}^{3\times2\times2}$ defined as

$$X = \left[\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, \begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}, \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}\right]$$

(3, 1, 2) Then the tensor $X_{(3,1,2)} \in \mathbb{R}^{2\times2\times3}$ would be written entrywise as

$$X_{(3,1,2)}(i_1, i_2, i_3) = X(i_3, i_1, i_2)$$

And therefore,

$$X_{(3,1,2)} = \left[\begin{pmatrix}1 & 1 & 1\\ 0 & 0 & 0\end{pmatrix}, \begin{pmatrix}0 & 1 & 0\\ 1 & 1 & -1\end{pmatrix}\right]$$

Reshaping tensors onto matrices will also be very useful since it will let us treat high order tensors as matrices and then apply numerical algorithms there.

Definition 2.20 (Tensor unfolding)**.** *Let T be a tensor of order n with $n \geqslant 2$. Let $\sigma \in S_n$ be a permutation of $(1, 2, \ldots, n)$. We define the* ***unfolding*** *of the tensor T as the 2nd-order tensor or matrix $\mathcal{U} \in \mathbb{R}^{\prod_{i=1}^{d} N_{\sigma(i)} \times \prod_{i=d+1}^{n} N_{\sigma(i)}}$ entrywise as*

$$\mathcal{U}(\overline{i_{\sigma(1)}, \ldots, i_{\sigma(d)}}, \overline{i_{\sigma(d+1)}, \ldots, i_{\sigma(n)}}) = T(i_1, \ldots, i_n)$$

We will write $\mathcal{U} = \text{unfold}\,(\mathcal{T}, (p_1, \ldots, p_d), (p_{d+1}, \ldots, p_n))$

Example 2.21. Suppose that we have X defined as in Example 2.19. Then unfold $(X, (1, 2), (3))$ would result in a matrix $\mathbb{R}^{3\times4}$ with its elements defined as $\mathcal{U}(\overline{i_1 i_2}, \overline{i_3}) = X(i_1, i_2, i_3)$. Computing each entry gives

$$\text{unfold}\,(X, (1, 2), (3)) = \begin{pmatrix}1 & 0 & 0 & 1\\ 1 & 0 & 1 & 1\\ 1 & 0 & 0 & -1\end{pmatrix}$$

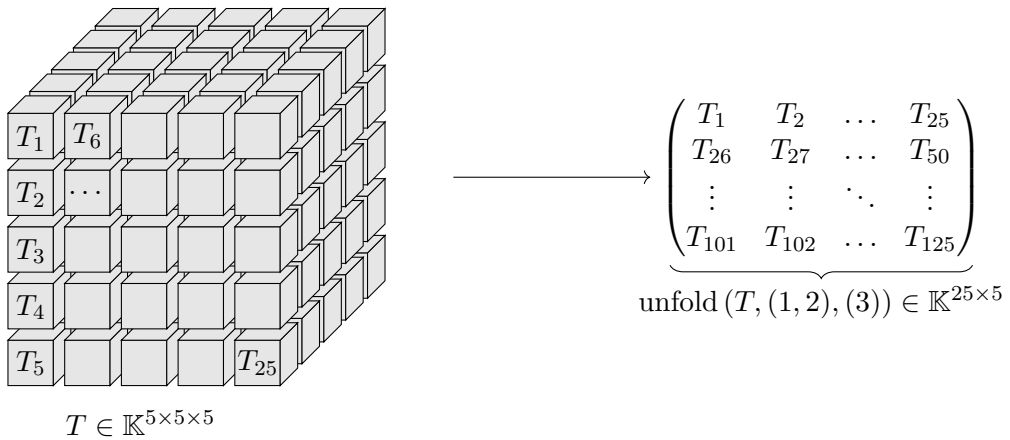


Figure 2.2: Representation of tensor unfolding. Source: own elaboration

Now we will introduce the reverse operation of the tensor unfolding: tensor folding.

Definition 2.22 (Tensor folding)**.** *Let $\sigma \in S_n$, $1 \leqslant d \leqslant n$. Let $N = \prod_{i=1}^{d}$, $\mathcal{U} = \prod i = d+1^n$ and M be a matrix of $\mathbb{K}^{N\times M}$. We define the **folding** of M following σ as the tensor $T \in \mathbb{K}^{N_{\sigma(1)}\times\cdots\times N_{\sigma(n)}}$ defined entrywise as:*

$$T(i_1,\ldots,i_n) = \mathcal{U}(\overline{i_{\sigma^{-1}(1)},\ldots,i_{\sigma^{-1}(d)}}, \overline{i_{\sigma^{-1}(d+1)},\ldots,i_{\sigma^{-1}(n)}})$$

We will write $T = \text{fold}(\mathcal{U}, (\sigma(1),\ldots,\sigma(d)), (\sigma(d+1),\ldots,\sigma(n)))$

Now that we have defined tensors and all the basic reshaping operations we will go towards constructing tensor networks. We will start by defining the contraction between two tensors:

2.3 Tensor contraction and the Penrose Notation

In this section we will define the tensor contraction operation, which is one of the most important operations in tensor algebra and is the core concept behind tensor networks. The idea behind tensor contraction is that given two tensors from different spaces, say for example $T \in \mathbb{R}^{N_1\times\cdots\times N_n}$ and $U \in \mathbb{R}^{M_1\times\cdots\times M_m}$, we pick one dimension of each tensor N_i and M_j with the same size $N_i = M_j$ and then we obtain a new tensor of order $n+m-2$ that contains all dimensions except for N_i and M_j, and each element of the resulting tensor is obtained by generalizing the matrix product opertion but along N_i and M_j dimensions.

So, the elements of the tensor contraction $T \times_j^i U$ would be, element-wise:

$$\sum_{k=1}^{N_i} T(s_1,\ldots,s_{i-1},k,s_{i+1},\ldots s_n)\cdot U(t_1,\ldots,t_{j-1},k,t_{j+1},\ldots,t_m)$$

Where $T \times_j^i U \in \mathbb{K}^{N_1\times\cdots\times N_{i-1}\times N_{i+1}\times\cdots\times N_n\times M_1\times\cdots\times M_{j-1}\times M_{j+1}\times\cdots\times M_m}$

Since the notation of tensor contractions is often very tedius, we will introduce the Penrose Notation for representing tensor contractions in a more compact and elegant way. The Penrose notation dates back from at least the early 1970s and was firstly used by Robert Penrose, to which the name is owed. [11]

Given an nth-order tensor $T \in \mathbb{K}^{N_1\times\cdots\times N_n}$ we represent it using the Penrose notation is as a circle with as many edges as the order of the tensor, as seen in fig. 2.3

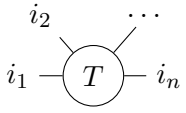


Figure 2.3: Representation of a tensor $T \in \mathbb{K}^{N_1\times\cdots\times N_n}$ using the Penrose notation. Source: own elaboration

The dimensions of the tensor are not explicitly written in the Penrose notation. Instead, only the order of the indices is preserved. If they are not explicitly set on the labels of the edges, the order of the indexes of the tensor will be determined by their orientation respect to the circle: the order starts from the left and then follows a clockwise rotation. The order in which we encounter the edges will be the order of the indexes. For example, in fig. 2.3, the order would be $i_1, i_2, \ldots, i_n$

Figure 2.4: The identity matrix represented using the Penrose Notation

Then, we represent a contraction of two tensors on the Penrose notation by joining to edges of different tensors. The edges that are joined will be the edges that the contraction is performed. As seen in fig. 2.5.

We represent a free edge that is unconnected to anything as the identity matrix.

Now, we will give a more formal definition of the tensor contraction since there is a natural way in which tensor contraction definition appears from applying a vector space with its dual with different tensors. As before, we take two tensors T and U with $T \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ and $U \in \mathbb{W}_1 \otimes \cdots \otimes \mathbb{W}_m$. Suppose that there exists some vector space of the dimensions of U that is the dual of some space of the tensor T, in other words, suppose that exists some i and j in which $\mathbb{W}_j = \mathbb{V}_i^*$. Then, the tensor contraction is doing the tensor product of T and U and then applying $\mathbb{V}_i$ with its dual afterwards. Doing this gives the following definition:

Definition 2.23. *Let $P \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ and suppose that there exists some i and j such that $\mathbb{V}_j = \mathbb{V}_i^*$ The following map*

$$\begin{aligned} \mathcal{C}_j^i : \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n &\longrightarrow \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_{i-1} \otimes \mathbb{V}_{i+1} \otimes \cdots \otimes \mathbb{V}_{j-1} \otimes \mathbb{V}_{j+1} \otimes \cdots \otimes \mathbb{V}_n \\ v_1 \otimes \cdots \otimes v_n &\longmapsto (v_1 \otimes \cdots \otimes v_{i-1} \otimes v_{i+1} \otimes \cdots \otimes v_{j-1} \otimes v_{j+1} \otimes \cdots \otimes v_n)\, v_j(v_i) \end{aligned}$$

where $v_j \in \mathbb{V}_i^*$. We define $\mathcal{C}_j^i$ as the tensor contraction mapping of $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ over the indices i and j. We call $\mathcal{C}_i^j(T)$ the contraction of T by indices (i, j).

Now, we define the **contraction of two tensors** T and U defined as before with some space $\mathbb{W}_j$ being the dual of some $\mathbb{V}_i$ as $\mathcal{C}_j^i(T \otimes U)$. We will sometimes write $T \times_j^i U$

We will represent the contraction between two tensors as their representation in the Penrose notation with the edges that represent the indexes that are contracting by joining them, as seen in fig. 2.5.

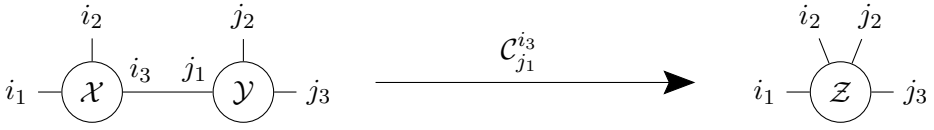


Figure 2.5: Representation in the Penrose notation of the contraction between two tensors $\mathcal{X} \in \mathbb{K}^{N_1 \times N_2 \times N_3}, \mathcal{Y} \in \mathbb{K}^{M_1 \times M_2 \times M_3}$ by their indices i_3 and j_1 with $N_1, N_2, N_3, M_1, M_2, M_3 \in \mathbb{Z}_+$ and $N_3 = M_1$. Source: own elaboration

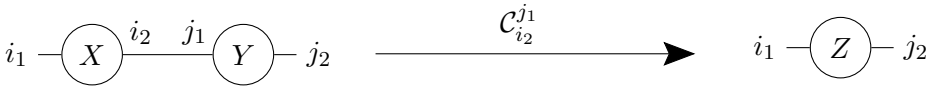
If we fix basis for $\mathbb{V}_1, \ldots, \mathbb{V}_p, \mathbb{W}_1, \ldots, \mathbb{W}_q$ and we represent X, Y as discrete functions by its representations in those basis, we get a way for computing $\mathcal{C}_k^l(X \otimes Y)$ as:

$$\begin{aligned} &\mathcal{C}_k^l(i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_p, j_1, \ldots, j_{l-1}, j_{l+1}, \ldots, j_q) \\ &= \sum_{s=1}^{N_k} \mathcal{X}(i_1, \cdots, i_{k-1}, s, i_{k+1}, \cdots, i_p)\mathcal{Y}(j_1, \cdots, j_{l-1}, s, j_{l+1}, \cdots, j_q) \end{aligned} \tag{2.3.1}$$

Example 2.24. The tensor contraction for two tensors $M_1 \in \mathbb{K}^{N_1 \times N_2}$ and $M_2 \in \mathbb{K}^{N_2 \times N_3}$ over one edge of each tensor yields the matrix multiplication. Applying eq. (2.3.1) we get that $Z = C_2^2(X \otimes Y)$ is defined entry-wise as:

$$Z(i_1, j_2) = \sum_{s=1}^{N_2} M_1(i_1, s) M_2(s, j_2)$$

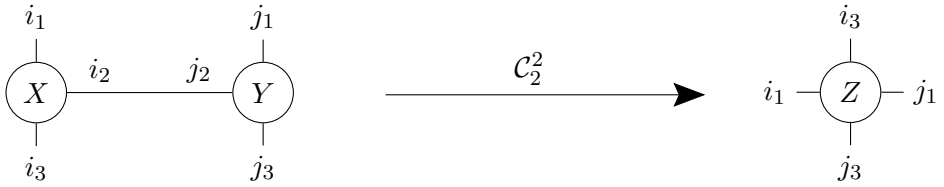
And that is identical to the conventional matrix product. Visually, we can represent it using the Penrsone notation:



Example 2.25. Suppose that we take $X \in \mathbb{R}^{3\times2\times2}$ from Example 2.21 and $Y \in \mathbb{R}^{2\times2\times3}$ defined as:

$$X = \left[\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, \begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}, \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}\right] \qquad Y = \left[\begin{pmatrix}1 & 0 & -1\\ 0 & 2 & -2\end{pmatrix}, \begin{pmatrix}1 & 1 & 0\\ 2 & 0 & 2\end{pmatrix}\right]$$

We want to compute the contraction of X and Y from the second index of each tensor $Z = C_2^2(X \otimes Y)$. Note that $Z \in \mathbb{R}^{3\times2\times2\times3}$. Using the Penrose Notation, the contraction would look like



And each element of the contracted tensor Z would be defined as:

$$Z(i_1, i_3, j_1, j_3) = \sum_{s=1}^{2} X(i_1, s, i_3) Y(j_1, s, j_3)$$

For example, by fixing $i_1 = i_3 = 1$ we can compute the first "inner matrix" of Z:

$$Z(1, 1, j_1, j_3) = \sum_{s=1}^{2} X(1, s, 1) Y(j_1, s, j_3)$$

This sum is equal to

$$X(1,1,1)Y(j_1,1,j_3) + X(1,2,1)Y(j_1,2,j_3) = Y(j_1,1,j_3) = \begin{pmatrix}1 & 0 & -1\\ 1 & 1 & 0\end{pmatrix}$$

If we do the same for the rest of indices i_1, i_3, we get

$$Z = \begin{bmatrix} \begin{pmatrix}1 & 0 & -1\\ 1 & 1 & 0\end{pmatrix} & \begin{pmatrix}0 & 2 & -2\\ 2 & 0 & 2\end{pmatrix} \\ \begin{pmatrix}1 & 2 & -3\\ 3 & 1 & 2\end{pmatrix} & \begin{pmatrix}0 & 2 & -2\\ 2 & 0 & 2\end{pmatrix} \\ \begin{pmatrix}1 & 0 & -1\\ 1 & 1 & 0\end{pmatrix} & \begin{pmatrix}0 & -2 & 2\\ -2 & 0 & -2\end{pmatrix} \end{bmatrix}$$

One can clearly see that the contraction that we defined on the Penrose notation (joining two edges) is the same as the formal definition of the contraction of two tensors.

Now we will be extending the Penrose notation for representing some special cases:

Identity tensors

Identity tensors on the Penrose notation are represented as free edges that are not connected to any tensor. We denote by I_n as the tensor identity of order n. I_2 are identity matrices.

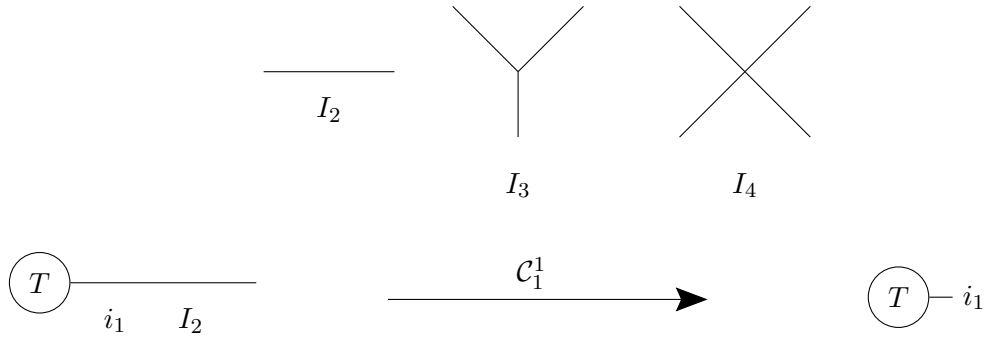


Figure 2.6: The identity matrix represented using the Penrose Notation. Contracting a mode over the identity yields the same tensor. Source: own elaboration.

This makes complete sense since contracting some tensor along some mode over the identity matrix yields the same tensor. This is ilustrated in fig. 2.6

Trace

There may be the case that when contracting a series of tensors, we might end up as what we see as a loop in the Penrose Notation. Contracting over these two indexes we get the trace of the tensor $\mathcal{T}$ respect the indices i_k and i_p and we denote it as $\mathrm{Tr}_p^k(T)$ (See fig. 2.7). This operation is well defined since contracting over two modes of the same tensor is doing a tensor contraction using Definition 2.23.

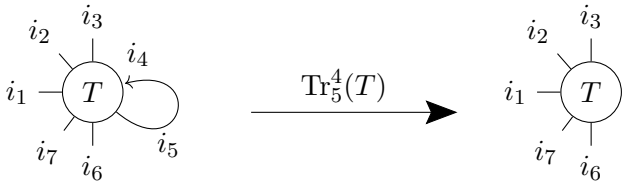


Figure 2.7: Representation of the trace of a tensor using the Penrose notation. Source: own elaboration.

Diagonal tensors

A diagonal tensor $\Lambda \in \mathbb{K}^{N\times\cdots\times N}$ of order n is a tensor which only has entries in his diagonals, i.e, there are $\lambda_1, \ldots, \lambda_n \in \mathbb{K}$ such that

$$\Lambda(i_1, \ldots, i_n) = \begin{cases} \lambda_j & \text{if } i_1 = \ldots i_n = j \\ 0 & \text{otherwise} \end{cases}$$

We will draw diagonal tensors as identity tensors but with an small circle on the middle.

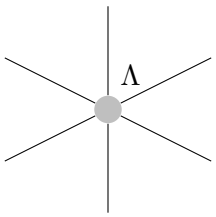


Figure 2.8: Diagonal tensor represented using the Penrose Notation. Source: own elaboration.

Outer product

And if two tensors are unconnected, we can represent it as one tensor but we do not perform any contraction between them since they are not connected. In this case, the outer product of the two tensors is performed.

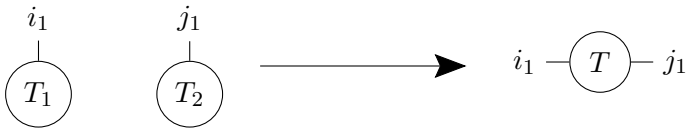


Figure 2.9: Representation of the outer product using the Penrose notation. Source: own elaboration.

As we previously said, eq. (2.3.1) is equivalent to a matrix product, therefore it is possible to compute a tensor contraction by multiplying two tensors unfolded in a concrete way. the tensors $\mathcal{X}$ and $\mathcal{Y}$, we can compute $\mathcal{C}_l^k(X \otimes Y)$ as a matrix product.

Corollary 2.26. *Let* $X \in \mathbb{K}^{N_1\times\cdots\times N_k\times\cdots\times N_n}, Y \in \mathbb{K}^{M_1\times\cdots\times M_l\times\cdots\times M_m}$ *with* $1 \leqslant k \leqslant n$, $1 \leqslant l \leqslant m$, $N_k = M_l$. *The matrix product*

$$\mathcal{X}(\overline{i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n}, \overline{i_k}) \cdot \mathcal{Y}(\overline{j_l}, \overline{j_1, \ldots, j_{l-1}, j_{l+1}, \ldots, j_m})$$

Results in a $\left(\prod_{i=1}^{n} N_i\right)/N_k \times \left(\prod_{i=1}^{m} M_i\right)/M_l$ *matrix, which can be reshaped back onto* $C_l^k(X \otimes Y)$.

Example 2.27. Following from Example 2.25, we will reshape the tensors X and Y for computing Z by only performing a matrix product. We need to first compute the matrices $\mathcal{X} = \text{unfold}\,(X, (1,3), (2)) \in \mathbb{R}^{6\times 2}$ and $\mathcal{Y} = \text{unfold}\,(Y, (2), (1,3)) \in \mathbb{R}^{2\times 6}$. These unfoldings result in:

$$X = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & -1 \end{pmatrix} \qquad Y = \begin{pmatrix} 1 & 0 & -1 & 1 & 1 & 0 \\ 0 & 2 & -2 & 2 & 0 & 2 \end{pmatrix}$$

Therefore:

$$Z = XY = \begin{pmatrix} 1 & 0 & -1 & 1 & 1 & 0 \\ 0 & 2 & -2 & 2 & 0 & 2 \\ 1 & 2 & -3 & 3 & 1 & 2 \\ 0 & 2 & -2 & 2 & 0 & 2 \\ 1 & 0 & -1 & 1 & 1 & 0 \\ 0 & -2 & 2 & -2 & 0 & -2 \end{pmatrix}$$

And we can now reshape $Z \in \mathbb{R}^{9\times 9}$ as the tensor in $\mathbb{R}^{3\times 3\times 3\times 3}$ as

$$Z(i_1, i_3, j_1, j_3) = Z(\overline{i_1, i_3}, \overline{j_1, j_3})$$

which in fact, is identical to Z in Example 2.25

2.4 Tensor Networks

In this section we will present tensor networks formally from a mathematically formal standpoint. Then, we will present the notion of the tensor G-rank that will be useful for determining if a tensor network can represent a tensor. We will present some common tensor network structures. After that, we will discuss over the ordering in which its the most optimal to contract a tensor network and we will give an algorithm that finds an optimal order of contraction, and finally, we will present the alternating least squares algorithm applied to generic tensor networks for explicitly finding the tensor cores of a tensor network.

The concept of tensor networks originated from a physics background. Roger Penrose described how its diagrammatic language could be used in various applications of physics [11]. Later, in 1992, Steven R. White developed de Density Matrix Renormalization Group (DRMG) algorithm for quantum lattice systems. It was considered the first successfull tensor network application [12].

A tensor network is a directed graph G. With a vector space $\mathbb{V}_i$ encoded at each node and two spaces $\mathbb{E}_j$ and $\mathbb{E}_j^*$ encoded in each edge of the graph. Each node will represent a tensor that contains in a tensor product space that at least contains $\mathbb{V}_i$. Each other of the spaces that the tensor product space is spanned, are combinations of some $\mathbb{E}_j$ and $\mathbb{E}_k^*$.

Since we have already seen that contractions give the same result, we will often write G as an undirected graph. But we will not do that at the moment because with a directed graph the contraction order is unequivocally well defined.

Definition 2.28. *Given a directed graph $G = (V, \bar{E})$ and a vertex $i \in V$ we define*

$$\mathrm{IN}(i) = \{j \in V : (j, i) \in \bar{E}\} \qquad \mathrm{OUT}(i) = \{j \in V : (i, j) \in \bar{E}\}$$

Let $\mathbb{V}_1, \dots, \mathbb{V}_d$ be vector spaces with $\dim \mathbb{V}_i = N_i, i = 1, \dots, d$. Let $\mathbb{E}_1, \dots, \mathbb{E}_c$ be finite vector spaces with $\dim \mathbb{E}_i = R_i, i = 1, \dots, c$. For each vertex $i \in V$ we will associate the tensor product space

$$\xi_i := \left(\bigotimes\nolimits_{j \in \mathrm{IN}(i)} \mathbb{E}_j \right) \otimes \mathbb{V}_i \otimes \left(\bigotimes\nolimits_{j \in \mathrm{OUT}(i)} \mathbb{E}_j^* \right)$$

A **tensor network** or **tensor network structure** is completly defined by the graph G and all the tensor product spaces $\xi_1, \dots, \xi_n$. We will also associate to the tensor network a contraction mapping $\mathcal{C}_G$ defined by contracting factors in $\mathbb{E}_j$ with factors of $\mathbb{E}_j^*$:

$$\mathcal{C}_G : \bigotimes\nolimits_{i=1}^{n} \xi_i \longrightarrow \bigotimes\nolimits_{i=1}^{d} \mathbb{V}_i$$

Note that we have defined the tensor product spaces of the tensor network structure ξ_i this way because when we contract the whole graph, we will get a tensor of $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$. Since every directed edge (i, j) must point out of a vertex i and point into a vetex j, each copy of $\mathbb{E}_j$ is paired with one copy of $\mathbb{E}^*$, so the contraction $\mathcal{C}_G$ is well defined and it results in a tensor of $\mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ (See fig. 2.10)

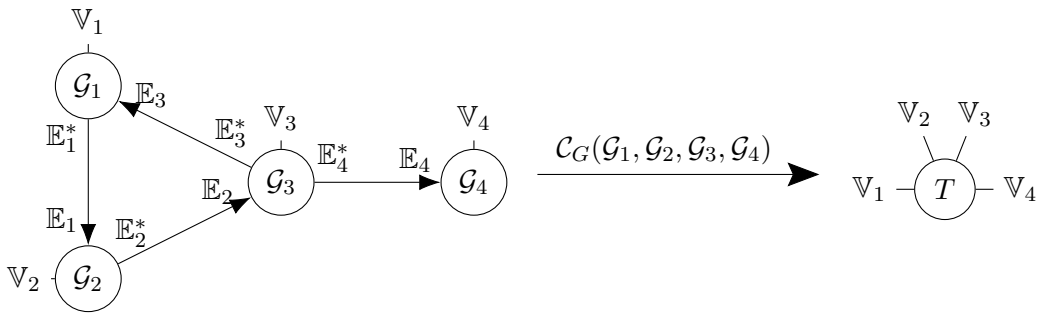


Figure 2.10: Example of a tensor network and a tensor network state evaluation using the Penrose notation. Source: own elaboration.

By picking some tensors $\mathcal{G}_i \in \xi_i$ and evaluating them through $\mathcal{C}_G$ we get a tensor T. We will call a **tensor network state** the set of all contracted tensors that are possible to obtain by varying $\mathcal{G}_1, \dots, \mathcal{G}_n$. We will also call the tensors $\mathcal{G}_i$ **core tensors**.

Definition 2.29. *We will define the set of all possible tensor states of a tensor network as the set* $\mathrm{TNS}(G; \mathbb{E}_1, \dots, \mathbb{E}_c, \mathbb{V}_1, \dots, \mathbb{V}_n)$*, i.e*

$$\mathrm{TNS}(G; \mathbb{E}_1, \dots, \mathbb{E}_c, \mathbb{V}_1, \dots, \mathbb{V}_n) := \left\{ \kappa_G(\mathcal{G}_1 \otimes \cdots \otimes \mathcal{G}_n) \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n : \mathcal{G}_i \in \xi_i \right\}$$

Is not very important to distinguish the vector spaces $\mathbb{E}_1, \dots, \mathbb{E}_c, \mathbb{V}_1, \dots, \mathbb{V}_n$ on the Penrose representation of the tensor network so we will not write them. This is because for each dangling edge of each node is an space $\mathbb{V}_i$ and we already know that each edge encodes a contraction. That means that by looking only at the directed graph we can already know the shape of the core tensors. Also, since all vector spaces are determined up to isomorphism by its dimension, we will write the tensor network as $\mathrm{TNS}(G; R_1, \dots, R_c, N_1, \dots, N_n)$. Where $R_i = \dim \mathbb{E}_i$ are called the **ranks of the tensor network state** and $N_i = \dim \mathbb{V}_i$ are called the **dimensions of the tensor network states**.

Sometimes we will also omit the dimensions of the tensor network states since n is equal to the number of vertices of G and c is equal to the number of edges of G, we will write $\mathrm{TNS}(G; R, N)$ for a more compact notation, with R and N being tuples of size c and n respectively.

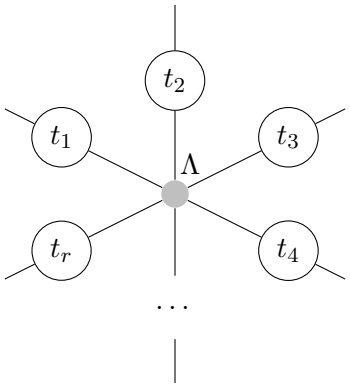
To give some examples of tensor network structures, we will now see some widely used structures and we will discuss some of their properties.

2.5 Common Tensor network structures

Example 2.30. [Canonical polyadic decomposition] As we have seen, canonocal polyadic decomposition consists on decomposing T as a sum of r 1-rank tensors. We can write

$$\sum_{p=1}^{r} \lambda_p v_i^1 \otimes v_i^2 \otimes \cdots \otimes v_i^n$$

As the contraction of a diagonal tensor Λ of order r with its enties being λ_p and the other 1-rank tensors. If we denote $t_i = v_i^1 \otimes v_i^2 \otimes \cdots \otimes v_i^n$ so that the decomposition can be written as $\sum_{p=1}^{r} \lambda_p t_p$, the resulting tensor network has the following star shape:



Example 2.31. [Tensor Train (TT) Decomposition] A **tensor train decomposition** or **matrix product state** of T are a set of 3th-order tensors $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$ with $\mathcal{G}_i \in \mathbb{K}^{R_{i-1} \times N_i \times R_i}$ and $R_0 = R_n = 1$ such that every element of T is written in the form

$$T(i_1, i_2, \ldots, i_n) = \sum_{r_0, \ldots, r_n}^{R_0, \ldots, R_n} \mathcal{G}_1(r_0, i_1, r_1)\mathcal{G}_2(r_1, i_2, r_2) \cdots \mathcal{G}_n(r_{n-1}, i_n, r_n) \tag{2.5.1}$$

We denote $R_0, R_1, \ldots, R_n$ as the ranks of the tensor train decomposition, or TT-ranks.

We can easily see that the TT decomposition is obtained by our definition of a tensor network when G is a path, also the contraction of the whole network yields eq. (2.5.1)

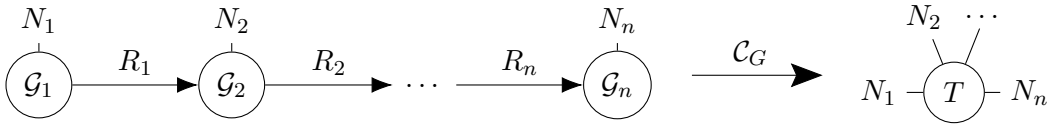


Figure 2.11: Tensor Train decomposition. Source: own elaboration.

Tensor train decomposition are very well studied because they are a chain of products of matrices, there are a lot of mathematical theory that can be applied in this structure,

for example, for finding the cores of the TT decomposition one could fix a core, reshape and unfold onto a matrix the objective tensor T, apply then SVD and update the cores according to the decomposition. A more detailed algorithm can be found in [13]. Since in this thesis is centered in genereal tensor structures will not give more details about this algorithm.

Example 2.32. [Tensor Ring Decomposition] **Tensor ring decomposition** (or TR) or also known a **matrix product state with periodic boundary conditions**, is obtained when G is a cycle.

Tensor Ring decomposition is considered generalization of Tensor Train decomposition, it's contraction is the same as eq. (2.5.1) but removing the condition $R_0 = R_1 = 1$. Zhao et. al. explain that Tensor Rings have also been widely studied because of their circular invariance: shifting the cores on the tensor network results in an equivalent tensor network that once its evaluated results in the objective tensor with its dimensions fixed. Thanks to this fact, one can still apply SVD decomposition on Tensor Rings [14].

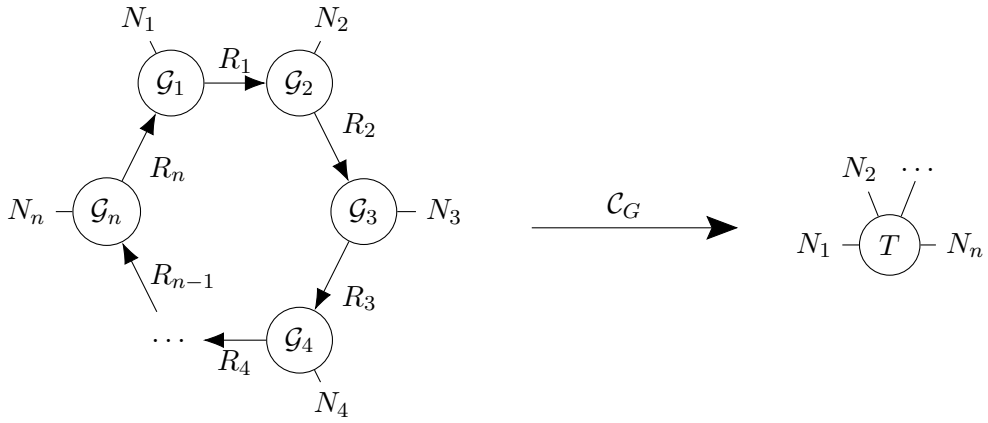


Figure 2.12: Tensor Ring (TR) decomposition. Source: own elaboration.

Example 2.33. The fully connected tensor network decomposition is obtenied when G is a complete graph.

Example 2.34. A tree tensor network decomposition is obtained when G is a tree graph. We will see that tensor tree networks have the the nice property that if we can represent a tensor T in some state of the network, the G-rank of T is unique.

Now that we have seen some examples we will define another type of tensor rank that depends on the tensor network structure itself. We will finish the section showing that there are tensor network decompositions that results over this rank i.e the decomposition a tensor that is inside a tensor network state can be of lower dimension than the one that results from the traditional rank.

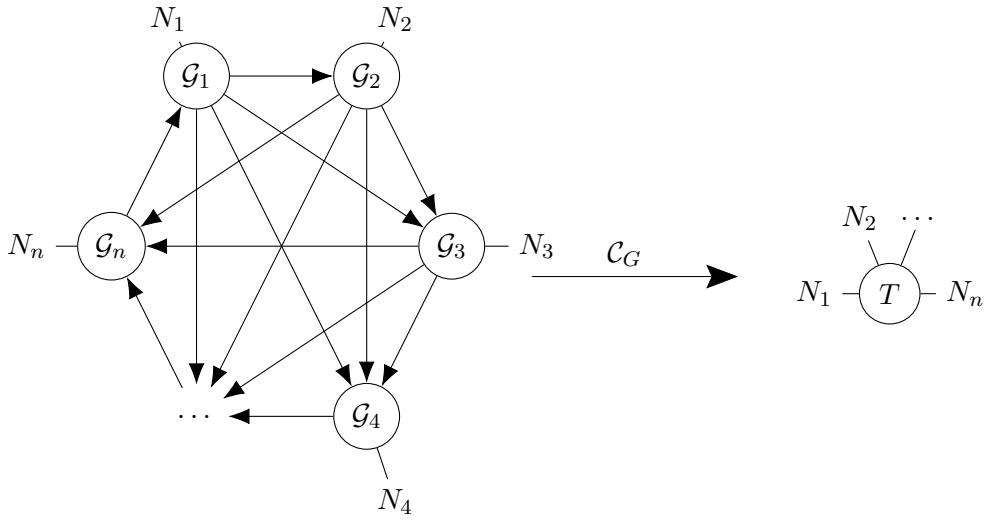


Figure 2.13: Fully connected tensor network decomposition (FCTN). Source: own elaboration.

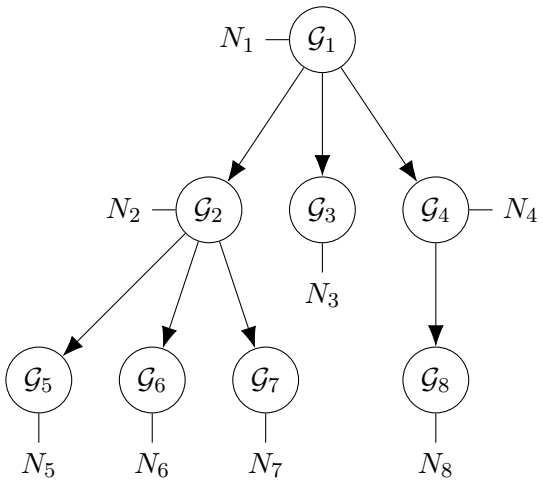


Figure 2.14: A tree tensor network

2.6 Tensor Network Ranks

In this section we will present the notion of G-rank of a tensor. The G-rank of a tensor is defined as the minimum set of ranks $(R_1, \ldots, R_c)$ that a tensor T can be represented by a tensor state. Knowing the G-rank of a tensor will be useful because we could find cores $\mathcal{G}_1, \ldots, \mathcal{G}_n$ such that we could represent T, and surely enough, with a enough sotisficated algorithm, it could converge to these core tensors.

Our main goal will be to try to estimate the G-rank of a tensor, since finding the G-rank explicitly is a very hard task. Approximating the G-rank will be enough for us because in the next chapter we will slightly vary the approximated G-rank for finding a good representation, thanks to the TnALE algorithm.

As we defined the rank for a tensor T as the minimum number of 1-rank tensors that compose the tensor T, we will define the G-rank as the minimum ranks that a tensor

network state needs for representing T. More formally,

Definition 2.35 (Tensor G-rank)**.** *Given a graph G, we define the tensor rank respect to a G or G-**rank** as*

$$\operatorname{rank}_G(T) = \min\{(R_1, \dots, R_c) \in \mathbb{Z}_+^c : T \in \operatorname{TNS}(G; R_1, \dots, R_c, N_1, \dots, N_d)\}$$

Where $\min(S)$ with $S \subset \mathbb{Z}_+^c$ denotes the minimal elements of S. We treat $\mathbb{Z}_+^c$ with its usual partial order:

$$(a_1, \dots, a_c) \leqslant (b_1, \dots, b_c) \iff a_1 \leqslant b_1, a_2 \leqslant b_2, \dots, a_c \leqslant b_c$$

So for example if $S = \{(3,4,5), (2,1,3), (1,3,2)\}$, then $\min(S) = \{(2,1,3), (1,3,2)\}$

Now, we will see that $\operatorname{rank}_G(T)$ is a finite set. The following theorem gives us that if we make $R_1, \dots, R_c$ big enough, every tensor T can be a state of $\operatorname{TNS}(G; R_1, \dots, R_c, N_1, \dots, N_n)$. In fact, these values that guarantee that T is an state are $R_1 = \dots R_c = \operatorname{rank} T$

Theorem 2.36. *Let $T \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ and let G be a connected graph with n vertices and c edges. There exists $R_1, \dots, R_c \in \mathbb{Z}_+$ such that*

$$T \in \operatorname{TNS}(G; R_1, \dots, R_c, N_1, \dots, N_d)$$

in fact, we can choose $R_1 = \cdots = R_c = \operatorname{rank} T$

Proof. Let $r = \operatorname{rank} T$. Then there exist $v_1^{(i)}, \dots, v_r^{(i)} \in \mathbb{V}_i, i = 1, \dots, n$ such that

$$T = \sum_{p=1}^{r} v_1^{(p)} \otimes \cdots \otimes v_n^{(p)}$$

We take $R_1 = \cdots = R_c = r$ we take for each $i = 1, \dots, n$

$$\mathcal{G}_i = \sum_{p=1}^{r} \left(\bigotimes\nolimits_{j \in \operatorname{IN}(i)} e_p^{(j)} \right) \otimes v_p^{(i)} \otimes \left(\bigotimes\nolimits_{j \in \operatorname{OUT}(i)} e_p^{(j)*} \right)$$

Now observe that for each $i = 1, \dots, n$ there exists an unique h such that whenever $j \in \operatorname{IN}(i) \cap \operatorname{OUT}(i)$, $e_p^{(j)}$ and $e_p^{(j)*}$ contract and give δ_{pq}, therefore the summand vanishes except when $p = q$. This together with the assumption that G is a connected graph implies that $\kappa_G(\mathcal{G}_1 \otimes \cdots \otimes \mathcal{G}_n)$ reduces to a sum of terms of the form $v_p^{(1)} \otimes \cdots \otimes v_p^{(d)}$ for $p = 1, \dots, r$, which is of course T □

Now, by picking the ranks as $R_1 = \cdots = R_c = \operatorname{rank} T$ is usually not optimal, since we will end up that our tensor network will use more memory to represent T than the memory we need to write T itself.

We will proceed to show that the G-rank can be a lot more smaller than the tensor rank, and in fact, we will show some examples of tensor networks that can represent tensors in a more efficient way than the canonical polyadic decomposition that we have seen in chapter 2.

The following theorem says that there exists some tensor networks and some tensors that make this claim true:

Theorem 2.37. *For $n \geqslant 3$ there exists a connected simple graph G with n vertices and c edges such that there exists a tensor $T \in \mathbb{V}_1 \otimes \cdots \otimes \mathbb{V}_n$ where its tensor rank* $\operatorname{rank}(T) = r$ *is significantly larger than its G-rank* $\operatorname{rank}_G(T) = (r_1, \ldots, r_n)$*. More specifically,*

$$r \gg r_1 + \cdots + r_n$$

The proof of this theorem can be found in [8]. In the paper the authors find that for representing a tensor network that contracts to a tensor $T \in (\mathbb{C}^{n\times n})^* \otimes (\mathbb{C}^{n\times n})^* \otimes \mathbb{C}^{n\times n} \cong \mathbb{C}^{n\times n\times n}$ picking G as C_3 and P_3 the inequality holds for $n = 3$. Then, the proof constructs from T for $n > 3$. This theorem shows that the G-rank can offer a more compact representation of tensors than the classical tensor rank. Nontheless, the theorem is not proven for generic tensor product spaces, in which proving that the G-rank could be smaller than the classical tensor rank are more difficult.

In this chapter we have presented tensors, tensor networks and tensor network states. We have proved that sometimes this representations of a tensor can be more compact than the traditional rank decomposition. What remains now is how to find the cores $\mathcal{G}_1, \ldots, \mathcal{G}_n$ of a given tensor network structure that give this compact decomposition. and also finding what tensor structure would be more optimal for finding these cores, and also what we mean by a more "optimal" structure.

Chapter 3

Tensor network state search

In this chapter we will give algorithms that aim to solve the following two problems:

1. Given a fixed TNS$(G; R)$ and an objective tensor T, find the best cores $\mathcal{G}_1, \ldots, \mathcal{G}_c$ such that $\mathcal{C}_G(\mathcal{G}_1, \ldots, \mathcal{G}_c) \approx T$.
2. Given an objective tensor T find the optimal (G^*, R^*) such that guarantees that exists some core tensors $\mathcal{G}_1, \ldots, \mathcal{G}_{c^*}$ such that they minimize a loss function that depends on R^* and the relative error of $\mathcal{C}_G(\mathcal{G}_1, \ldots, \mathcal{G}_c)$ and T.

We will present two algorithms that solve (1): the TnALS algorithm and the backpropagation algorithm. We will see that each one has some pros and caveats. For solving (2) we will present the TnALE algorithm. We will also prove the convergence of these algorithms under some assumptions.

3.1 Core finding algorithms

In this section we will present the algorithms that aim to solve problem (1).

3.1.1 The TnALS algorithm

In this subsection we will discuss the Alternating Least Squares algorithm that Malik et. al. present on section 5 of [15]

We will denote $T \in \mathbb{K}^{N_1 \times \cdots \times N_n}$ as our objective tensor. Let $G = (V, E)$, c the number of edges $|E|$ and $R = (r_1, \ldots, r_c)$. We would want to find some cores $\mathcal{G}_1, \ldots, \mathcal{G}_c$ of TNS$(G; R)$ such that minimize

$$\|T - \mathcal{C}_G(\mathcal{G}_1, \ldots, \mathcal{G}_c)\|_F$$

If we impose that all the tensor cores remain fixed except $\mathcal{G}_m$, our problem would become

$$\arg\min_{\mathcal{G}_m} \|T - \mathcal{C}_G(\mathcal{G}_1, \ldots, \mathcal{G}_n)\|_F$$

Now, we apply our contraction mapping $\mathcal{C}_G$ for all cores excluding $\mathcal{G}_m$ (figs. 3.1 and 3.2). We will call this contracted tensor $\mathcal{G}^{\neq m}$.

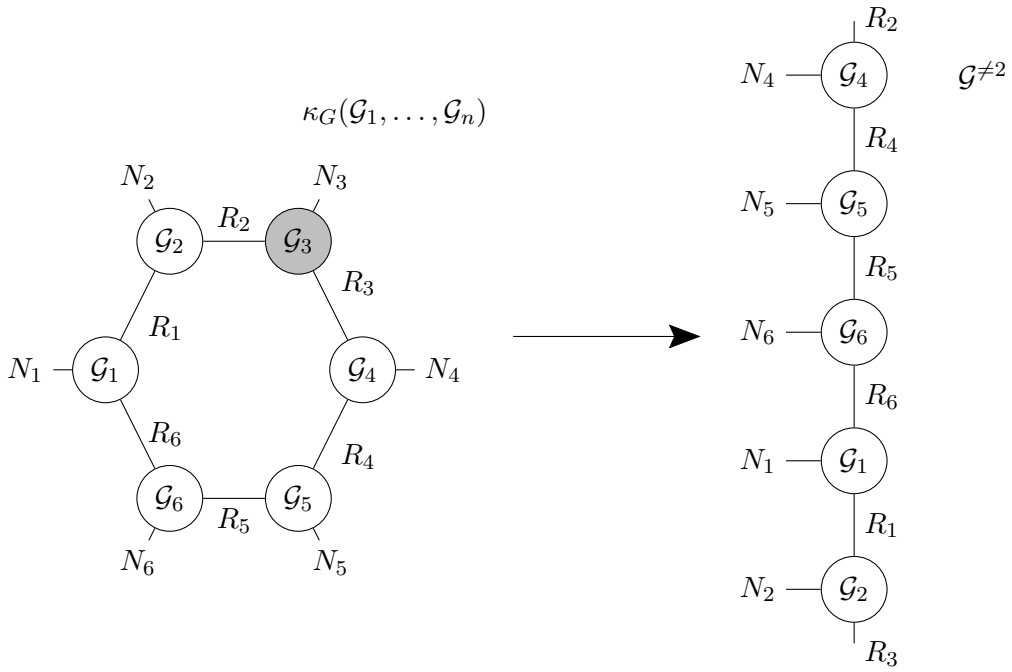


Figure 3.1: The representation of $\mathcal{G}^{\neq m}$ on the TR decomposition, with $m = 2$. Source: own elaboration.

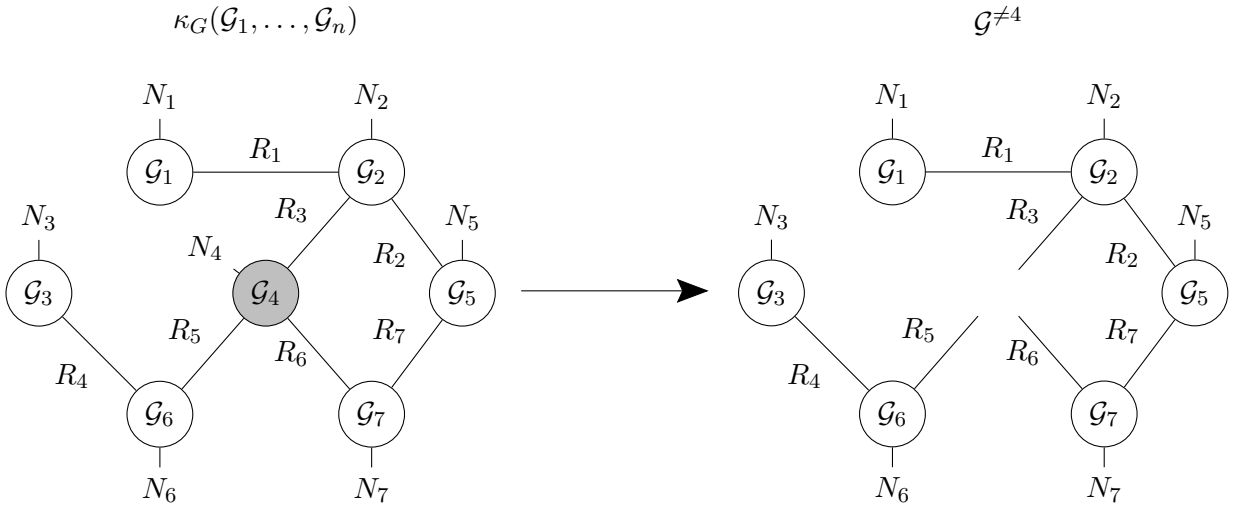


Figure 3.2: The representation of $\mathcal{G}^{\neq 4}$ over some arbitrary TN. Source: own elaboration.

The last contractions that remain between $\mathcal{G}^{\neq m}$ and $\mathcal{G}_m$ are equivalent to evaluating the whole network. Now, if we consider appropiate matricizations $T^{(m)}$, $G^{\neq m}$ and G_m, these contractions can be computed calculating the matrix product $G^{\neq m}G_m$, so our minimization problem is now equivalent to solving the least squares problem. That is, given $A \in \mathbb{K}^{m\times n}$ and $B \in \mathbb{K}^{m\times k}$, find $X \in \mathbb{K}^{n\times k}$ such that

$$\arg\min_X \|AX - B\|_2 \tag{3.1.1}$$

In our case, $A = G^{\neq m}, X = G_m$ and $B = T^{(m)}$. Let $x^{(i)}$ be the i-th column of G_m and

$y^{(i)}$ the i-th column of $T^{(m)}$. Solving 3.1.1 means solving for each i

$$\arg\min_{x^{(i)}} \|G^{\neq m}x^{(i)} - y^{(i)}\|_2 \tag{3.1.2}$$

Solving the least squares problem by using normal equations

Note that $G^{\neq m}$ does not have to be a square matrix, so for minimizing the (3.1.2) we can not directly compute its inverse. Instead, we can solve the **normal equation** of the least square problem. The normal equation is the linear system

$$A^TAx = A^Tb$$

A^TA is an square matrix and now we can already solve this system by traditional means.

Proposition 3.1. *If x satisfies $A^T(Ax - b) = 0$ then x is a solution to the least-squares problem, i.e, x minimizes $\|Ax - b\|_2$*

Proof. Let $y \in \mathbb{R}^n$ be any vector. Then,

$$\begin{aligned}
\|A(x+y) - b\|_2^2 &= [A(x+y) - b]^T[A(x+y) - b] \\
&= (Ax - b)^T(Ax - b) + 2(Ay)^T(Ax - b) + (Ay)^T(Ay) \\
&= \|Ax - b\|_2^2 + 2y^TA^T(Ax - b) + \|Ay\|_2^2 \\
&= \|Ax - b\|_2^2 + \|Ay\|_2^2 \\
&\geqslant \|Ax - b\|_2^2
\end{aligned}$$

□

Thanks to this proposition, any problem of the form $\arg\min_x \|Ax - b\|_2$ will have as a solution a vector x such that $A^TAx = A^Tb$. In our case, we get that solving eq. (3.1.2) is equivalent to solving the linear system

$$(G^{\neq m})^TG^{\neq m}x^{(i)} = (G^{\neq m})^Ty^{(i)} \tag{3.1.3}$$

Nontheless, we can't still gurantee that this system has a solution since we don't know if $(G^{\neq m})^TG^{\neq m}$ is invertible. We could solve this inconvinience applying the Tikhonov regularization. Instead of solving eq. (3.1.3), we will solve the lineal system

$$((G^{\neq m})^TG^{\neq m} + \gamma I)\,\hat{x}^{(i)} = (G^{\neq m})^Ty^{(i)}$$

where $\gamma > 0$ is called the **Tikhonov factor**. We will pick a small Tikhonov factor (usually $\gamma < 10^{-5}$) to assure that it doesn't smooth too much the solutions of the least squares problem.

Keep in mind that by adding any small perturbance to a linear system will not lead to the exact solution of $x^{(i)}$, but instead to a very close approximation $\hat{x}^{(i)}$. Therefore, the convergence of the algorithm can be affected by adding the Tikhonov factor.

Solving the least squares problem by the pseudoinverse of the SVD

Another approach on solving (3.1.3) that does not depend about if A is singular or not, is to apply the single value decomposition. The following method that we will describe is the same that uses the implementation of the function `torch.linalg.lstsq` of the PyTorch library that is used for solving the least squares problem [16] as a fallback (if the matrix A is well-conditioned, PyTorch uses more efficient methods such as QR factorization).

Given a matrix $A \in \mathbb{K}^{n\times m}$ we denote its conjugate transpose as A^*. If $\mathbb{K} = \mathbb{R}$ then $A^* = A^T$

Theorem 3.2 (SVD Factoriation)**.** *Any matrix $A \in \mathbb{K}^{n\times m}$ can be decomposed as*

$$A = U\Sigma V^*$$

Where $U \in \mathbb{K}^{n\times n}, V \in \mathbb{K}^{m\times m}$ are unitary matrices (i.e, $U^*U = UU^* = I_n$ and $V^*V = VV^* = I_m$) and $\Sigma \in \mathbb{K}^{n\times m}$ is a rectangular diagonal matrix, i.e:

$$\Sigma = \operatorname{Diag}(\sigma_1, \dots, \sigma_r, 0, \dots, 0)$$

*The entries of Σ are called **singular values** of A. They are usually ordered as $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_r \geqslant 0$ where $r = \operatorname{rank}(A)$ We call this factorization the **SVD factorization** of A.*

Proof. We can suppose that $A \neq 0$ because if $A = 0$ we could choose $U = V = I$ and $\Sigma = 0$. We will prove by induction over n. For $n = 1$, consider the matrix $A = (a) \neq 0$ with $a \in \mathbb{K}^m$, i.e A is a $m \times 1$ matrix. If we define $\sigma = \|a\|_2$ then there exists an ortogonal matrix $Q \in \mathbb{K}^{m\times m}$ such that $Q^*a = (\sigma, 0, \dots, 0)^T$, therefore

$$A = Q \begin{pmatrix} \sigma \\ 0 \end{pmatrix}$$

Therefore we can define the SVD of A as $U = Q$, $V = (1, \dots, 1)$ and $\Sigma = \begin{pmatrix} \sigma \\ 0 \end{pmatrix}$. Now, suppose that we can do the SVD factorization for any matrix of $n-1$ rows. Let $A \in \mathbb{K}^{m\times n}$, $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ be two unitary vectors such that $Ax = \sigma y$ with $\sigma = \|A\|_2$. We know that there exists some matrices $V_2 \in \mathbb{K}^{n\times(n-1)}$ and $U_2 \in \mathbb{K}^{m\times(m-1)}$ such that $V = (xV_2) \in \mathbb{K}^{n\times n}$ and $U = (yU_2) \in \mathbb{K}^{m\times m}$ are ortogonals. Then,

$$U^*AV = \begin{pmatrix} y^* \\ U_2^* \end{pmatrix} A \begin{pmatrix} x & V_2 \end{pmatrix} = \begin{pmatrix} y^*Ax & y^*AV_2 \\ U_2^*Ax & U_2^*AV_2 \end{pmatrix}$$

Therefore if we call $B = U_2^*AV_2$ and $w^* = y^*AV_2$ then

$$U^TAV = \begin{pmatrix} \sigma & w^* \\ 0 & B \end{pmatrix} = A_1 \tag{3.1.4}$$

And since

$$\left\| A_1 \begin{pmatrix} \sigma \\ w \end{pmatrix} \right\|_2^2 \geqslant (\sigma^2 + w^*w)^2$$

Then we have that $\|A_1\|_2^2 \geqslant \sigma^2 + w^Tw$ and since $\|A\|_2^2 = \|A_1\|_2^2 =^2$ we get that $w = 0$ and replacing w in (3.1.4) we can find an SVD representation for the matrix A_1. □

Definition 3.3 (Moore-Penrose pseudoinverse)**.** *Let $A \in \mathbb{K}^{n\times m}$. The pseudoinverse of A is a matrix $A^+ \in \mathbb{K}^{m\times n}$ such that satisfies the following four criteria, known as the Moore-Penrose conditions [17]:*

1. $AA^+A = A$
2. $A^+AA^+ = A$
3. $(AA^+)^* = AA^+$
4. $(A^+A)^* = A^+A$

Where A^* denotes the conjugate transposition of A.

Now we make the claim that $\Sigma^+ = \text{Diag}\left(\frac{1}{\sigma_1}, \ldots, \frac{1}{\sigma_r}, 0, \ldots, 0\right) \in \mathbb{K}^{n\times m}$

$$\Sigma = \begin{pmatrix} \overbrace{\begin{matrix} \sigma_1 & 0 & \ldots & 0 \\ 0 & \sigma_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \sigma_r \end{matrix}}^{m} \\ \begin{matrix} 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 \end{matrix} \end{pmatrix} \Bigg\} n \qquad \Sigma^+ = \begin{pmatrix} \overbrace{\begin{matrix} \frac{1}{\sigma_1} & 0 & \ldots & 0 & 0 & \ldots & 0 \\ 0 & \frac{1}{\sigma_2} & \ldots & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \frac{1}{\sigma_r} & 0 & \ldots & 0 \end{matrix}}^{n} \end{pmatrix} \Bigg\} m$$

By plugging each matrix to each Moore-Penrose condition we can see that, in fact, Σ^+ is the pseudoinverse of Σ. The following Lemma will give us a way to compute the Pseudoinverse if we have an SVD factorization:

Lemma 3.4. *If $A = U\Sigma V^T$, then $A^+ = V\Sigma^+U^T$*

Proof. It is sufficient to check if the Moore-Penrose conditions are satisfied. By the properties of transpositions and using that $VV^T = V^TV = I_n$ and $U^TU = UU^T = I_m$:

1. $AA^+A = (U\Sigma V^T)(V\Sigma^+U^T)(U\Sigma V^T) = U\Sigma(V^TV)\Sigma^+(UU^T)\Sigma V^T = U\Sigma I_m\Sigma^+I_n\Sigma V^T = U\Sigma\Sigma^+\Sigma V^T = U\Sigma V^T = A$
2. $A^+AA^+ = (V\Sigma^+U^T)(U\Sigma V^T)(V\Sigma^+U^T) = V\Sigma^+(U^TU)\Sigma(V^TV)\Sigma^+U^T = V\Sigma^+\Sigma\Sigma^+U^T = V\Sigma^+U^T = A^+$
3. $(AA^+)^T = (U\Sigma V^TV\Sigma^+U^T)^T = (U\Sigma\Sigma^+U^T)^T = U\Sigma\Sigma^+U^T = AA^+$
4. $(A^+A)^T = (V\Sigma^+U^TU\Sigma V^T)^T = (V\Sigma^+\Sigma V^T)^T = V\Sigma^+\Sigma V^T = A^+A$

□

Now, for solving (3.1.1) we can use the pseudoinverse of the SVD decomposition of the matrix A:

Proposition 3.5. *$x = V\Sigma^+U^*b$ minimizes $\|Ax - b\|_2$*

Proof. By applying the SVD factorization of the matrix A we get:

$$\|Ax - b\|_2^2 = \|U\Sigma V^* x - b\|_2^2 = \|U^*(U\Sigma V^* x - b)\|_2^2 = \|\Sigma(V^* x) - U^* b\|_2^2$$

In the last equality we used that U is an unitary matrix and $\|Ux\|_2 = \|x\|_2$. Let $y = V^*x$ and $c = U^*b$. Then the problem of minimizing $\|Ax - b\|_2$ reduces to solving

$$\min_y \|\Sigma y - c\|_2^2$$

. Since Σ is a diagonal matrix, if we pick an y with each component y_i of the form

$$y_i = \begin{cases} \dfrac{c_i}{\sigma_i} & \text{if } \sigma_i \neq 0 \\ \lambda_i & \text{if } \sigma_i = 0 \end{cases}$$

With $\lambda_i \in \mathbb{K}$ being any value, then $\|\Sigma y - c\|_2^2 = 0$. We will pick $\lambda_i = 0$ for all i, since this will give us the vector y that minimizes $\|Ax - b\|_2$ with the least norm, thus we have that $y = \Sigma^+ c$. By reversing our change of variables, we get that

$$x = Vy = V\Sigma^+ c = V\Sigma^+ U^* b$$

□

Once we have solved (3.1.1) with any of these two described methods, we can reshape back G_m to $\mathcal{G}_m$. We can iteratively change the varying core tensor $\mathcal{G}_i$ until the algorithm converges.

Algorithm 1 Tensor Network ALS

Input: A tensor $T \in \mathbb{K}^{N_1 \times \cdots \times N_n}$ and some fixed error ϵ
Output: Core tensors $\mathcal{G}_1, \ldots, \mathcal{G}_n$

Initialize tensors $\mathcal{G}_1, \ldots, \mathcal{G}_n$
while $\|T - \kappa_G(\mathcal{G}_1, \ldots, \mathcal{G}_n)\|_F > \epsilon$ **do**
    for $k = 1, \ldots, n$ **do**
        $\mathcal{G}_m \leftarrow \arg\min_{G_m} \|G^{\neq m} G_m - T^{(m)}\|_2$
return $\mathcal{G}_1, \ldots, \mathcal{G}_n$

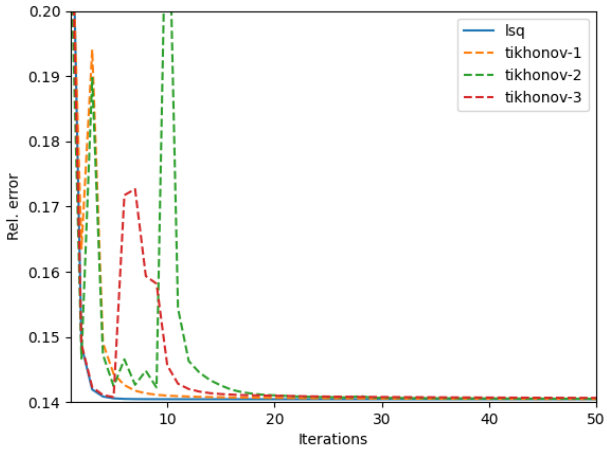
We will not enter in more detail about the explicit order of the modes when folding or unfolding the tensors. On the source code attached to this thesis, there is the complete implementation of this algorithm, detailing all the foldings and unfoldings.

As we can see, the ALS algorithm converges to a certain set of cores $\mathcal{G}_1^*, \ldots, \mathcal{G}_n^*$. If we denote $\Theta_k = \{\mathcal{G}_1^k, \ldots, \mathcal{G}_n^k\}$ as the set of cores $\mathcal{G}_i$ at the k-th iteration of the the algorithm, and if we denote $\mathcal{L}(\Theta_k) = \|G^{\neq k} G_k - T^{(k)}\|$ with $G^{\neq k}$, G_k and $T^{(k)}$ depending on Θ_k. Then, at each step we are minimizing $\mathcal{L}(\Theta_k)$, and so at the next iteration of the algorithm the loss function only can decrease, since by not varying the other core tensors we would at least stay at the same value of $\mathcal{L}(\Theta_k)$, therefore

$$\mathcal{L}(\Theta_{k+1}) \leqslant \mathcal{L}(\Theta_k)$$

Since the sequence $\{\mathcal{L}(\Theta_{(k)})\}_{k=0}^{\infty}$ is bounded by $0 \leqslant \mathcal{L}(\Theta_{(k)})$, therefore the algorithm converges to a fixed point Θ^*

Note that as we have previously said we can not guarantee the convergence of the algorithm if we use the Tikhonov regularization since it doesn't give the exact solution to the system in eq. (3.1.3), and therefore the solution might not solve the least squares problem. Nontheless, the Tikhonov regularization speeds up the computation, as we can see in the following figure:



	Time
lsq	4.47s
tikhonov-1	0.56s
tikhonov-2	0.58s
tikhonov-3	0.60s

Figure 3.3: Comparation between the ALS algorithm when the Tikhonov regularization is used and when not. The dashed lines represent different executions of the algorithm and the continuous line represents an iteration of the algorithm when using the least squares method for solving 3.1.2. The TN used is a TT with an objective tensor of shape $(8, 8, 16, 8, 8)$ and ranks $R = (8, 8, 8, 8)$. We can see that since in some cases when the linear system is near to zero, the Tikhonov regularization makes the relative error skyrocket. With enough iterations the Tikhonov regularization achieves the same relative error as the LSQ method. Source: own elaboration. Source: own elaboration.

3.1.2 Backpropagation

Since our problem of finding tensor cores is defined by minimizing a loss function, a valid idea would be to apply backpropagation.

In this subsection we will give another way of finding the tensor cores $\mathcal{G}_1, \ldots, \mathcal{G}_n$, but with backpropagation. The backpropagation algorithm is one of the core algorithms of deep learning. It consists on trying to find the variation that each parameter does to the loss function by exploiting the chain rule and then do some optimization with the gradient, such as gradient descent. We will now explain briefly the backpropagation algorithm in the context of tensor networks.

Suppose that we have a set D (a dataset), and a loss function $\pi : D \times \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_n} \to \mathbb{R}_+$ The core idea behind backpropagation is to tune the parameters in a way that we minimize π_D. We do this by computing the gradient of all of the parameters of each $\mathcal{G}_i$ and moving the parameters in the same direction as the gradient.

In the case that $D = \{T_0\}$ the backpropagation problem is the same as finding the best TNS that represents T_0, and if for example, dataset has more tensors, it can be applied for example as a deep learning problem, as we will see later in chapter 5.

Let $d = T_0$. We want to compute the gradient of

$$\pi(T_0, \mathcal{C}_G(\mathcal{G}_1, \ldots, \mathcal{G}_n)) \tag{3.1.5}$$

Suppose that we reorder $\mathcal{G}_1, \ldots, \mathcal{G}_n$ in a sense that the contractions for evaluating the tensor network become first contracting $\mathcal{G}_n$ and $\mathcal{G}_{n-1}$, then the resulting tensor with $\mathcal{G}_{n-2}$ and so on. The function in 3.1.5 becomes

$$\pi(T_0, C(\mathcal{G}_1, \mathcal{C}(\mathcal{G}_2, \mathcal{C}(\ldots, \mathcal{C}(\mathcal{G}_{n-1}, \mathcal{G}_n))))) \tag{3.1.6}$$

Let $X_n = \mathcal{C}(\mathcal{G}_{n-1}, \mathcal{G}_n)$ and let $X_k = \mathcal{C}(\mathcal{G}_k, X_{k+1})$ for all $1 \leqslant k \leqslant n-1$. We can rewrite 3.1.6 as $\pi(T_0, X_1)$.

We would want to find the derivative of $\mathcal{G}_k$ respect of the loss function π. Consider the intermidiate contraction

$$X_k = \mathcal{C}(\mathcal{G}_k, X_{k+1})$$

Applying the chain rule we get

$$\frac{\partial \pi}{\partial \mathcal{G}_k} = \frac{\partial \pi}{\partial X_1} \cdot \frac{\partial X_1}{\partial X_2} \cdots \frac{\partial X_{k-1}}{\partial \mathcal{G}_k} \tag{3.1.7}$$

The partial derivative of the core tensor $\mathcal{G}_k$ respect to the loss function tells us in which

If we want to compute the derivative of the loss function respect to an individual element $\mathcal{G}_k^{i_1,\ldots,i_d}$ of the core tensor, using again the chain rule as before we get:

$$\frac{\partial \pi}{\partial \mathcal{G}_k^{i_1,\ldots,i_d}} = \sum_{b_1,\ldots,b_m} \frac{\partial \pi}{\partial X_k^{b_1,\ldots,b_m}} \cdot \frac{\partial X_k^{b_1,\ldots,b_m}}{\partial \mathcal{G}_i^{i_1,\ldots,i_d}} \tag{3.1.8}$$

So, we just have seen that by computing the partials of each intermidiate contraction we can obtain the partial derivative of the loss function respect to each individual parameter of the core tensor.

After knowing each derivative, we can compute the gradient of π in respect to all entries of the core tensors

$$\nabla \pi = \left(\frac{\partial \pi}{\partial \mathcal{G}_1}, \frac{\partial \pi}{\mathcal{G}_2}, \ldots, \frac{\partial \pi}{\partial \mathcal{G}_n} \right) \tag{3.1.9}$$

With each entry $\frac{\partial \pi}{\partial \mathcal{G}_i}$ being a tensor as the same shape of $\mathcal{G}_i$. We can reshape $\nabla \pi$ as a vector of the form

$$\nabla \pi = \left(\frac{\partial \pi}{\partial \mathcal{G}_1^{a_1,\ldots,a_p}}, \ldots, \frac{\partial \pi}{\partial \mathcal{G}_n^{b_1,\ldots,b_m}} \right) \tag{3.1.10}$$

Once we computed $\nabla \pi$, now we have an idea that in a local environment, how adjusting each entry of each core tensor affects on the change of the loss function π. Let $w = (\operatorname{vec} \mathcal{G}_1, \ldots, \operatorname{vec} \mathcal{G}_n)$ be a vector containing all the entries of the core tensors arrenged in order. Then, we can update each core tensor by "descending" by the gradient $\nabla \pi$ i.e updating w by a tiny amount on the direction that minimizes the value of the loss function

$$w^* = w - \eta \cdot \nabla \pi(w) \tag{3.1.11}$$

Where η is a small positive number called the learning rate. If we keep iterating as in 3.1.11, eventually w will reach a local minima of the loss function.

We implemented this algorithm using the PyTorch python library, since it has a built-in auto differentiation engine called `torch.autograd`. For for every operation that we apply to any PyTorch parameter, it remembers the operations that has been applied to the tensor and then it is able to compute the gradient of all of our computations [18].

On section A.1 we can find a comparison between the ALS and the backpropagation algorithms between some tensor network. As we can observe in fig. A.3, on most cases the TN-ALS surpasses the backpropagation algorithm for finding core tensors with less relative error.

3.2 Structure search

In this section we will describe the TnALE algorithm, that aims to solve problem (2).

On the last section we supposed that we had found some optimal G and R for representing T, and then we described some algorithms for finding the cores $\mathcal{G}_1, \ldots, \mathcal{G}_n$. In this section, we will try to find these optimal (G, R).

This problem is known as the *tensor network structure search problem* and its objective is to find the most optimal tensor network that compresses our objective T while maintaining the expressivity of the network, i.e that the network is capable to represent a closer result to the actual objective.

We will define the **loss function** $\mathcal{L} : \mathbb{G} \times \mathbb{F}_G \to \mathbb{R}_+$ as

$$\mathcal{L}(G, R) = \phi(G, R) + \lambda \cdot O(G, R)$$

Where $\mathbb{G}$ is the space of all simple connected graphs of n nodes, $R = (r_1, \ldots, r_K) \in \mathbb{F}_G \subseteq \mathbb{Z}_+^K$ are the ranks of the tensor network, $\phi(G, R)$ represents a function that determines the complexity of the tensor network, $\lambda > 0$ is a tuning parameter and $O : \mathbb{G} \times \mathbb{F}_G \to \mathbb{R}_+$ is the **evaluation function** defined as:

$$O(G, R) = \min_{\mathcal{Z} \in TNS(G,R)} \pi_D(\mathcal{Z}) \tag{3.2.1}$$

We will define the tensor network structure search problem as solving the following discrete optimization problem:

$$\min_{(G,R) \in \mathbb{G} \times \mathbb{F}_G} \mathcal{L}(G, R) \tag{3.2.2}$$

The intuition behind presenting this optimization problem this way is that since the evaluation function depends on the tuning parameter λ we can adjust a balance between giving priority to how expressive is the network or on simplifying its complexity. We will pich ϕ as the **complexity function** as a function $\phi : \mathbb{G} \times \mathbb{F}_G \to \mathbb{R}_+$ In our case, we will pick the inverse of the compression rate. I.e,

$$\phi(G, R) = \frac{\text{Size}(\mathcal{G}_1) \cdot \text{Size}(\mathcal{G}_2) \cdots \text{Size}(\mathcal{G}_n)}{\text{Size}(T)}$$

We call the problem 3.2.2 as Tensor Network Structure Selection (TN-SS). TN-SS is a very generalized problem. There has been a lot of research on solving it under certain conditions:

- Tensor Network Rank Selection (TN-RS), it restricts G to be a fixed graph, and its objective is to find the tensor network ranks $R \in \mathbb{F}_G$.
- Tensor Network Permutation Selection (TN-PS) fixes the ranks R and search over the set of all the simple graphs that are isomorphic to G.
- Lastly, Tensor Network Topology Selection (TN-TS) searches over the set of all simple graphs of N vertices and the tensor ranks R are fixed

In this chapter we will present the tensor network structure search algorithm using alternating local enumeration (TnALE) introduced by Li et. al. on [19].

3.2.1 The TnALE algorithm

The purpose of the TnALE algorithm is to solve 3.2.2 by locally alternating the discrete parameters that define (G, R).

We will start by describing the algorithm itself, and then we will present all the theory for proving its convergence.

The idea behind the TnALE algorithm is very simple: pick a random graph and random ranks G and R and then, for each iteration modifiy the rank R_1 a certain value and then pick the one that minimizes the loss function. Then do the same for R_2 and so on until R_c. Then, update G by adding or removing some edges and stay with the structure that is minimizes the loss function. Then modify $R_c, R_{c_1}, \dots R_2$ as before.

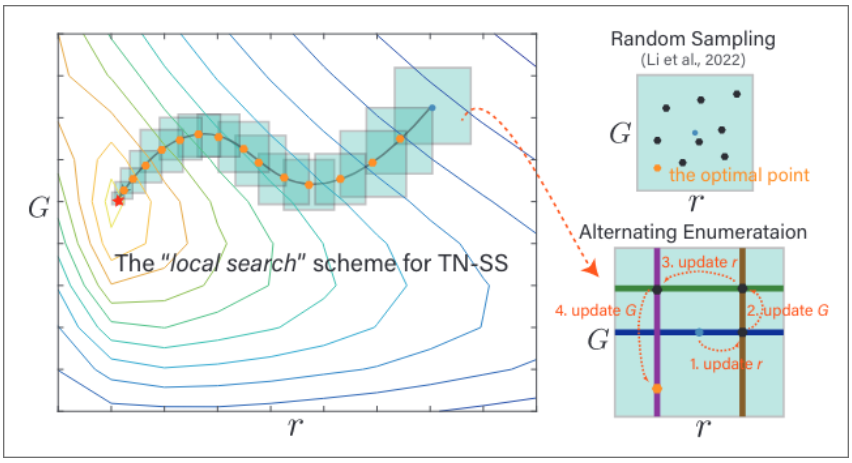


Figure 3.4: A representation of the TnALE algorithm and the difference of its sampling respect to TNLS [3]. r and G denote the structure related variables and each square represents a neighborhood of a given point in the iteration of TnALE. Source: [4]

Once this first pass is finished, the graph G is modified and each "local neighborhood" of the graph G is evaluated. Once again we save the best evaluation overwritting G and R.

Lastly, we do the first pass of varying the ranks $R_1, \dots, R_c$ but in the reverse order $R_c, \dots, R_1$. Doing this "round-trip" over the search of the ranks empirically results in a faster convergence rate to the optimal TNS structure.

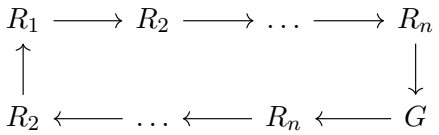


Figure 3.5: The "round-trip" of updating the optimal structure. Source: own elaboration

Algorithm 2 Tensor Network Alternating Local Enumeration (Tn-ALE)

Input: A starting point $(G^{(0)}, R^{(0)})$ with $R^{(0)} = (R_1^{(0)}, \dots, R_K^{(0)})^T \in \mathbb{Z}_+^K$ and a rank-related radius $r \in \mathbb{Z}_+$ and the number of "round-trips" D

Output: The optimal (G, R) that minimizes eq. (3.2.2)

1: Initialize $(G, R) = (G^{(0)}.R^{(0)})$ with $R = (R_1, \dots, R_K)^T$
2: **for** $d = 1, \dots D$ **do**
3: **for** $k = 1, \dots K$ **do**
4: **for** $i = -r, \dots, 0, \dots r$ **do**
5: Copy $(\bar{G}, \bar{R}) \leftarrow (G, R)$
6: Update $(\bar{G}, \bar{R})$ by $\bar{R}_k = R_k + r$
7: $h(i) \leftarrow O(G, R)$ `# We can apply linear interpolation`
8: Update (G, R) by $R_k = \arg\min_i h(i)$
9: Take the neighborhood of G as $N(G)$
10: **for all** $G' \in N(G)$ **do**
11: Update (G, R) by $G = G'$ (Insert or delete an element of R if necessary)
12: $h(G') \leftarrow O(G, R)$
13: Update (G, R) by $G = \arg\min_{G'} h(G')$
14: **for** $k = K, K-1, \dots 2$ **do**
15: **for** $i = -r, \dots, 0, \dots r$ **do**
16: Copy $(\bar{G}, \bar{R}) \leftarrow (G, R)$
17: Update $(\bar{G}, \bar{R})$ by $\bar{R}_k = R_k + r$
18: $h(i) \leftarrow O(G, R)$ `# We can apply linear interpolation`
19: Update (G, R) by $R_k = \arg\min_i h(i)$
20: **return** (G, R)

In Alg. 2 the pseudocode of TnALE is given. During the next subsection, we will need to introducte some theory of gradient-less optimization for proving the convergence of the Tn-ALE algorithm.

Keep in mind that for each time we try to vary some rank R_k or the graph G we will need to evaluate (3.2.1), and it can be computationally very expensive. In the case that we are varying a rank (See steps 7 and 21 of Alg. 2), we can approximate $O(G, R)$ by doing a linear interpolation between evaluating at the points $R_k - r, R_k, R_k + r$. Since by increasing ranks empirically yields to a better representation of the objective tensor [4] and therefore we get less relative error (and the other way around is generally also true).

We will now formally prove that the ALS algorithm converges. For that, we will need to introduce the gradientless descent.

3.2.2 Gradientless optimization

What we want to see in this subsection is to extend the notion of the gradient onto discrete functions to then be able to apply a discrete gradient descent over our loss function.

We will start by rewriting the loss function (3.2.2) in a more general form. We define $\mathbb{P}$ as the set of all functions of the form $p : \mathbb{Z}_+^K \to \mathbb{K}_+^L$

$$\min_{x \in \mathbb{Z}_+^K, p \in \mathbb{P}} f_p(x) := f \circ p(x) \tag{3.2.3}$$

Where $f_p : \mathbb{Z}_+^L \to \mathbb{R}_+$ is a generalization of the loss function, and $p : \mathbb{Z}_+^K \to \mathbb{Z}_+^L \in \mathbb{P}$ correspond to the topology-related variable (G, R).

We can show that our tensor network structure search problem corresponds to a concrete case of this general form: since G is completly defined by its adjacency matrix A which is defined as

$$A_{ij} := \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

We can encode the ranks R on the adjacency matrix by setting on each edge its corresponding rank:

$$A_{ij} := \begin{cases} R_{(i,j)} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Where $R_{(i,j)}$ denotes the rank of the edge $(i.j)$. We will denote this matrix as A_R

Example 3.6. Let $G = P_4$ and $R = (1, 2, 3)$. We associate the edges of G with the ranks R as $R_{(1,2)} = r_1 = 1$, $R_{(2,3)} = r_2 = 2$ and $R_{(3,4)} = r_3 = 3$. We consider $\mathrm{TNS}(G; R)$. The adjacency matrix of G and the encoded rank matrix of $\mathrm{TNS}(G; R)$ is, respectively:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad A_R = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 3 \\ 0 & 0 & 3 & 0 \end{pmatrix}$$

Given G and R, we can encode $x \in \mathbb{R}_+^K$ as a vector with the rank of each edge of the graph and $p \in \mathbb{P}$ as a permutation matrix. By varying only p we get all the permutations of the graph G, and by x we get vary the ranks of the edges.

Example 3.7.

$$A_R^1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 3 \\ 0 & 0 & 3 & 0 \end{pmatrix} \Leftrightarrow p_1(x) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 2 \\ 0 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 2 \\ 0 \\ 3 \end{pmatrix}$$

$$A_R^2 = \begin{pmatrix} 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 0 & 2 & 0 \end{pmatrix} \Leftrightarrow p_1(x) = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 2 \\ 0 \\ 3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 3 \\ 1 \\ 0 \\ 2 \end{pmatrix}$$

So, our problem is now a particular case of (3.2.3). We can now apply theory of gradientless optimizations in the real domain.

The goal of this section is to try to extend the gradient and its properties into discrete valued functions $f : \mathbb{Z}_+^L \to \mathbb{R}$. It is based on the theory that Golovin et. al. present on [20], and also on the discussion of the TnALE algorithm itself in the appendix of [4]. We will end this chapter by proving that if we take some assumptions of f like being convex and also smooth we can prove that the TnALE algorithm can converge. For that, we will define what is a discrete gradient, what is α-strong convergence and (β_1, β_2)-smoothness, we will prove some properties about strong convergence and smoothness.

Before we begin, since we will be working with vectors of $\mathbb{Z}_+^L$, we will introduce a norm for them:

Definition 3.8. *The l^2 norm of a vector $x = (x_1, \ldots, x_n)^T$ is defined as*

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

From now on, if x is a vector we will represent $\|x\|$ as $\|x\|_2$. We can now start by defining the concept of finite gradient:

Definition 3.9. *For any function $f : \mathbb{Z}_+^L \to \mathbb{R}$ its finite gradient $\nabla f : \mathbb{Z}_+^L \to \mathbb{R}$ at a point $x \in \mathbb{L}_+^L$ is defined as*

$$\Delta f(x) = [f(x + e_1) - f(x), \ldots, f(x + e_L) - f(x)]^T$$

With e_i being the unit vector of $\mathbb{Z}_+^L$ defined with the i-th entry being 1 and the rest of entries zeros.

Lemma 3.10. *Let $x \in \mathbb{Z}_+^L$. Then $\Delta\|x\|_2^2 = 2x + \mathbf{1}$ where $\mathbf{1}$ denotes the unit vector of $\mathbb{Z}_+^L$ that all of its entries are 1.*

Proof.

$$\Delta\|x\|_2^2 = \Delta\left(\sum_{i=1}^{L} |x_i|^2\right) = \sum_{i=1}^{L} \Delta|x_i|^2 = \sum_{i=1}^{L} 2x_i + e_i = 2x + \mathbf{1}$$

Where in the penultimate equality we used that

$$\Delta|x_i|^2 = |x_i + e_i|^2 - |x_i|^2 = |x_i|^2 + 2\langle x_i, e_i\rangle + |e_i|^2 - |x_i|^2 = 2\langle x_i, e_i\rangle + e_i = 2x_i + e_i$$

□

Now we will extend the strong convexity using the finite gradient. Recalling, a differentiable function $f : X \subseteq \mathbb{R}^n \to \mathbb{R}$ is convex if and only if for all $x, y \in X$

$$f(y) \geqslant f(x) - \langle \Delta f(x), y - x\rangle$$

Where $\Delta f(x)$ denotes the gradient of f. If f also satisfies that

$$f(y) \geqslant f(x) + \langle \Delta f(x) - \frac{\alpha}{2}\mathbf{1}, y - x\rangle + \frac{\alpha}{2}\|y - x\|_2^2 \tag{3.2.4}$$

For some $\alpha \geqslant 0$ e say that f is α-strongly convex.

Definition 3.11. *We say that $f : \mathbb{Z}_+^L \to \mathbb{R}$ is α-strongly convex with $\alpha \geqslant 0$ if it satisfies eq. (3.2.4) with $\Delta f(x)$ being the finite gradient for all $x, y \in \mathbb{Z}_+^L$ if the inequality holds for $\alpha = 0$, we will say that f is convex.*

We will prove now some properties of α-strongly convex functions. The following lemma gives us the equivalence between α-stronglt convex functions and convex functions:

Lemma 3.12. *$f : \mathbb{Z}_+^L$ is an α-strongly convex function iff $g(x) = f(x) - \frac{\alpha}{2}\|x\|_2^2$ is convex $\forall x \in \mathbb{Z}_+^L$*

Proof. ($\Rightarrow$) The first statement is equivalent to proving

$$g(y) \geqslant g(x) + \langle \Delta g(x), y - x\rangle \quad \forall x, y \in \mathbb{Z}_+^L \tag{3.2.5}$$

Applying that $f(x)$ is α-strongly convex we get that

$$\begin{aligned} g(y) - g(x) - \langle \Delta g(x), y - x\rangle &= f(y) - \frac{\alpha}{2}\|y\|_2^2 - f(x) + \frac{\alpha}{2}\|x\|_2^2 - \langle \Delta g(x), y - x\rangle \\ &= f(y) - \frac{\alpha}{2}\|y\|_2^2 - f(x) + \frac{\alpha}{2}\|x\|_2^2 - \langle \Delta f(x) - \alpha(2x + \mathbf{1}), y - x\rangle \\ &= f(y) - f(x) - \langle \Delta f(x) - \frac{\alpha}{2}\mathbf{1}, y - x\rangle - \frac{\alpha}{2}\|y\|_2^2 + \frac{\alpha}{2}\|x\|_2^2 + \frac{\alpha}{2}\langle 2x, y - x\rangle \\ &\geqslant \frac{\alpha}{2}\|y - x\|_2^2 - \frac{\alpha}{2}\|y\|_2^2 + \frac{\alpha}{2}\|x\|_2^2 + \frac{\alpha}{2}\langle 2x, y - x\rangle \\ &= \frac{\alpha}{2}\left(\|y - x\|_2^2 - \|y\|_2^2 - \|x\|_2^2 + 2\langle x, y\rangle\right) = 0 \end{aligned}$$

($\Leftarrow$) From eq. (3.2.5) we get

$$\begin{aligned} & f(y) - \frac{\alpha}{2}\|y\|_2^2 \geqslant f(x) - \frac{\alpha}{2}\|x\|_2^2 + \langle \Delta f(x) - \frac{\alpha}{2}(2x + \mathbf{1}), y - x\rangle \\ \Leftrightarrow\ & f(y) \geqslant f(x) - \frac{\alpha}{2}(\|x\|_2^2 + \|y\|_2^2) + \langle \Delta f(x) - \frac{\alpha}{2}\mathbf{1}, y - x\rangle - \frac{\alpha}{2}\langle 2x, y - x\rangle \\ \Leftrightarrow\ & f(y) \geqslant f(x) + \langle \Delta f(x) - \frac{\alpha}{2}\mathbf{1}, y - x\rangle + \frac{\alpha}{2}\|y - x\|_2^2 \end{aligned}$$

The last inequality gives us that f is α-strongly convex. □

The following lemma tells us that α-strongly convex functions have a more strict inequality of the monotone gradient property (a function is convex iff its gradient is monotone) than convex functions. The second inequality is a result of applying the Cauchy-Swartz inequality to the first one:

Lemma 3.13. *Given f an α-strongly convex function in $\mathbb{Z}_+^L$, then:*

1. $\langle \Delta f(x) - \Delta f(y), x - y\rangle \geqslant \alpha\|x - y\|_2^2 \quad \forall x, y \in \mathbb{Z}_+^L$
2. $\|\Delta f(x) - \Delta f(y)\|_2 \geqslant \alpha\|x - y\|_2 \quad \forall x, y \in \mathbb{Z}_+^L$

Proof. Let $g(x)$ be the same as in Lemma 3.12. To prove (1), we know that

$$\langle \Delta g(x) - \Delta g(y), x - y\rangle \geqslant 0, \forall x, y \in \mathbb{Z}_+^L$$

by the monotone gradient property of the convexity, which is true in both continuous and discrete scenarios. By substituting $\Delta g(x)$ we get that

$$\langle \Delta f(x) - \frac{\alpha}{2}(2x + \mathbf{1}) - \Delta f(y) + \frac{\alpha}{2}(2y + \mathbf{1}), x - y \rangle \geqslant 0$$

Simplifying we obtain that for all $x, y \in \mathbb{Z}_+^L$

$$\langle \Delta f(x) - \Delta f(y), x - y \rangle \geqslant \alpha \|x - y\|_2^2 \tag{3.2.6}$$

For proving (2) Consider the following inequality:

$$\|\Delta f(x) - \Delta f(y)\|_2 \|x - y\|_2 \geqslant \langle \Delta f(x) - \Delta f(y), x - y \rangle \geqslant \alpha \|x - y\|_2^2$$

Where in the first inequality we used the Cauchy-Swartz inequality and the last inequality follows from (3.2.6) □

The following lemma tells us that if we have a β-Lipschitz discrete function, the gradient is bounded. This will be one of the main assumptions that we will make about our loss function.

Lemma 3.14. *If $\|f(x) - f(y)\|_2 \leqslant \beta \|x - y\|$ for all $x, y \in \mathbb{Z}_+^L$ then the norm of the finite gradient is bounded by β i.e $\|\Delta f(x)\|_\infty = \max(|\Delta f(x)_1|, \ldots, |\Delta f(x)_L|) \leqslant \beta$*

Proof. Denote $\Delta f(x)_i$ the i-th entry of $\Delta f(x)$. Then for all $1 \leqslant i \leqslant L$,

$$|\Delta f(x)_i| = |f(x + e_i) - f(x)| \leqslant \beta \|x + e_i - x\| = \beta$$

□

Definition 3.15. *We say f is (β_1, β_2)-smooth for $\beta_1, \beta_2 > 0$ if*

1. *$|f(x) - f(y)| \leqslant \beta \|x - y\|_2$ for all $x, y \in \mathbb{Z}_+^L$*
2. *The function $l(x) := \frac{\beta_2}{2}\|x\|_2^2 - f(x)$ is convex*

The first item makes f to a β_1-Lipschitz function implying the "continuity" of the function, while the second item ensures an upper bound to the change of the gradient, because of the following lemma:

Lemma 3.16. *If $l(x) = \frac{\beta_2}{2}\|x\|_2^2 - f(x)$ is convex, then $\forall x, y \in \mathbb{Z}_+^L$ the following inequality satisfies:*

$$\langle \Delta f(x) - \Delta f(y), x - y \rangle \leqslant \beta \|x - y\|_2^2$$

Proof. By the convexity of $l(x)$ we get that

$$l(y) \geqslant l(x) + \langle \Delta l(x), y - x \rangle$$

Similarly we get

$$l(x) \leqslant l(y) + \langle \Delta l(y), x - y \rangle$$

Summing the two inequalities, we get that

$$l(y) + l(x) \geqslant l(x) + l(y) + \langle \Delta l(y) - \Delta l(x), x - y \rangle$$

And therefore

$$\langle \Delta l(x) - \Delta l(y), x - y \rangle \geqslant 0$$

Applying $l(x)$ we get that

$$\langle \beta(2x + \mathbf{1}) - \Delta f(x) - \beta(2y + \mathbf{1}) + \Delta f(y), x - y \rangle \geqslant 0$$

And by simplifying the inequality we have that

$$\langle \Delta f(x) - \Delta f(y), x - y \rangle \leqslant \beta \|x - y\|_2^2$$

□

3.2.3 Proof of the convergence of TnALE

After presenting α-strongly convex functions and (β_1, β_2)-smooth functions, in this subsection we will now prove that by setting the assumptions that our original problem from 3.2.3 the function f_p is α-strongly convex and (β_1, β_2)-smooth with $0 \leqslant \alpha \leqslant \beta_1 \leqslant \beta_2 \leqslant 1$ and that the minimum point of the discrete gradient which will be a fixed point of the TnALE algorithm does in fact exists and that its gradient is bounded by a very small factor.

We will start by defining the sub-level set of a function at a point. This definition will be useful for later proving that the TnALE algorithm can descend:

Definition 3.17 (Sub-level set)**.** *The level set of f at a point $x \in \mathbb{Z}_+^L$ is defined as the set $\mathbb{L}_x(f) = \{y \in \mathbb{Z}_+^L : f(y) = f(x)\}$ The sub-level set of f at a point x is defined as the set $\mathbb{L}_x^{\downarrow} = \{y \in \mathbb{Z}_+^L : f(y) \leqslant f(x)\}$*

Now, with the following lemma we will see that we can always find a neighbourhood that is contained inside $\mathbb{L}_x^{\downarrow}$, this will mean that we would be able to find a succession of points $(x_n)_{n=1}^{\infty}$ such that each $f(x_i) \leqslant f(x_j)$ if $i > j$

Lemma 3.18. *Let $f : \mathbb{Z}_+^L \to \mathbb{R}$ be a α-strongly convex, (β_1, β_2)-smooth and that its minimum value $f(x^*)$ satisfies that $\|\frac{\beta_2}{2}\mathbf{1} - \Delta f(x^*)\| \leqslant \gamma$ where γ is a constant and $0 \leqslant \gamma \leqslant \alpha$. Then $\forall x \in \mathbb{Z}_+^L$ there exists a L-dimensional cube which is of edge length $\frac{2(\alpha-\gamma)}{\beta_2\sqrt{L}}\|x - x^*\|$, tangent at x and inside the sub-level set $\mathbb{L}_x^{\downarrow}(f)$*

The proof of this lemma can be found at lemma B.8 from the appendix of [4]

Lemma 3.19 (Convex combination in the discrete domain)**.** *Suppose that $q = \theta x + (1-\theta)y$ for all $x, y \in \mathbb{Z}_+^L$ and $\theta \in [0, 1]$ and that there is a $\hat{q} \in \mathbb{Z}_+^L$ such that if $\Lambda = q - \hat{q}$ and we suppose that f is α-strongly convex then*

$$\theta f(x) + (1 - \theta) f(y) \geqslant f(\hat{q}) + \langle \Delta f(\hat{q}) - \frac{\alpha}{2}\mathbf{1}, \Lambda \rangle + \frac{\alpha}{2}\|\Lambda\|_2^2$$

Note that this lemma justifies that we can use the linear interpolation trick when evaluating tensor networks, since by picking $\hat{q}$ as the local minimum when varying one parameter of the rank R_i, since we are moving across one line all of our real evaluations that we are skipping will be bounded below our linear interpolation.

Proof. By the definition of the α-strong convexity we get

$$f(x) \geqslant f(\hat{q}) + \left\langle \Delta f(\hat{q}) - \frac{\alpha}{2}\mathbf{1}, x - \hat{q} \right\rangle + \frac{\alpha}{2}\|x - \hat{q}\|_2^2$$

$$f(y) \geqslant f(\hat{q}) + \left\langle \Delta f(\hat{q}) - \frac{\alpha}{2}\mathbf{1}, y - \hat{q} \right\rangle + \frac{\alpha}{2}\|y - \hat{q}\|_2^2$$

And therefore,

$$\begin{aligned}\theta f(x) + (1-\theta)f(y) &\geqslant f(\hat{q}) + \left\langle \Delta f(\hat{q}) - \frac{\alpha}{2}\mathbf{1}, \Lambda \right\rangle + \frac{\alpha}{2}(\theta\|x\|^2 + (1-\theta)\|y\|^2 + \|\hat{q}\|^2 - 2\langle q, \hat{q}\rangle) \\ &\geqslant f(\hat{q}) + \left\langle \Delta f(\hat{q}) - \frac{\alpha}{2}\mathbf{1}, \Lambda \right\rangle + \frac{\alpha}{2}(\|q\|^2 + \|\hat{q}\|^2 - 2\langle q, \hat{q}\rangle) \\ &= f(\hat{q}) + \left\langle \Delta f(\hat{q}) - \frac{\alpha}{2}\mathbf{1}, \Lambda \right\rangle + \frac{\alpha}{2}\|\Lambda\|^2\end{aligned}$$

□

Theorem 3.20. *Let $f : \mathbb{Z}_+^K \to \mathbb{R}_+$ is α-strongly convex, (β_1, β_2)-smooth and let the minimum of eq.* (3.2.3) *be (p^*, x^*). Assume that f satisifies $\|\Delta f_{p^*}(x^*) - \frac{\beta_2}{2}\mathbf{1}\|_2 \leqslant \gamma$ where $0 \leqslant \gamma < \alpha \leqslant \beta_1 \leqslant \beta_2 \leqslant 1$.*

Then, if we let p fixed as p^*, and $0 \leqslant \theta \leqslant 1$ and for any x with $\|x - x^*\|_\infty \leqslant c$ we can find a neighborhood $B_\infty(x, r_X)$ where $r_X \geqslant \theta c + \frac{1}{2}$ such that there exists an element $y \in B_\infty(x, r_X)$ satisfying

$$f_{p^*}(y) - f_{p^*}(x^*) \leqslant (1-\theta)(f_{p^*}(x) - f_{p^*}(x^*)) + \frac{7}{8}K$$

Proof. Since we are letting p to be fixed as p^*, the general problem $\min_{x\in\mathbb{Z}_+^K, p\in\mathbb{P}} f_p(x)$ can be simplified and equivalently rewritten as $\min_{x\in\mathbb{Z}_+^K} f(x)$ Where $f : \mathbb{Z}_+^K \to \mathbb{R}_+$ represents the objective function. By Lemma 3.19 we have

$$f(\hat{q}) - f(x^*) \leqslant (1-\theta)(f(x) - f(x^*)) + \left\langle \frac{\alpha}{2}\mathbf{1} - \Delta f(\hat{q}), \Delta \right\rangle - \frac{\alpha}{2}\|\Delta\|^2$$

Now we will see that there exists an element y inside a neighborhood $B(x, r_x)$ with $r_x \in \mathbb{R}_+$ such that y also belongs to the sub-level cube tangent at $\hat{q}$ from Lemma 3.18 so that the inequality $f(y) \leqslant f(\hat{q})$ holds. We will prove the existance of this point y by showing that the intersection between the sub-level cube tangent at $\hat{q}$ and $B(x, r_x)$ is not empty. For that we will find the distance between $\hat{q}$ and x satisfies that

$$\|x - \hat{q}\|_\infty = \|x - q + \Lambda\|_\infty \leqslant \|x - q\|_\infty + \|\Lambda\|_\infty = \theta\|x - x^*\|_\infty + \|\Lambda\|_\infty \leqslant \theta c + \frac{1}{2}$$

The last inequality follows from $\|\Lambda\| \leqslant \frac{1}{2}$ which holds because we can always find $\hat{q} \in \mathbb{Z}_+^K$ by rounding the entries of q to its closest integers. Therefore we have proven that if $r_x \geqslant \theta c + \frac{1}{2}$ the intersection of the sub-level cube tangent at $\hat{q}$ and $B(x, r_x)$ is not empty, proving the existance of the point y.

Lastly, if we take an element y from the intersection we get that

$$\begin{aligned}
f(y) - f(x^*) &\leqslant f(\hat{q}) - f(x^*) \leqslant (1-\theta)(f(x) - f(x^*)) + \left\langle \frac{\alpha}{2}\mathbf{1} - \Delta f(\hat{q}), \Lambda \right\rangle - \frac{\alpha}{2}\|\Lambda\|^2 \\
&\leqslant (1-\theta)(f(x) - f(x^*)) + \left|\left\langle \frac{\alpha}{2}\mathbf{1}, \Lambda \right\rangle\right| + |\langle \Delta f(\hat{q}), \Lambda\rangle| + \frac{\alpha}{2}\|\Lambda\|^2 \\
&\leqslant (1-\theta)(f(x) - f(x^*)) + \frac{\alpha}{4}K + \|\Delta f(\hat{q})\|_\infty \|\Lambda\|_1 + \frac{\alpha}{2}\|\Lambda\|^2 \\
&\leqslant (1-\theta)(f(x) - f(x^*)) + \frac{\alpha}{4}K + \frac{\beta_1}{2}K + \frac{\alpha}{8}K \\
&\leqslant (1-\theta)(f(x) - f(x^*)) + \frac{3\alpha + 4\beta_1}{8}K \\
&\leqslant (1-\theta)(f(x) - f(x^*)) + \frac{7}{8}K
\end{aligned}$$

The inequality of the fourth line follows from $\|\Delta f(x)\|_\infty \leqslant \beta_1$ and from $\|\Lambda\|_\infty \leqslant \frac{1}{2}$ and the last line comes from that $\alpha \leqslant \beta_1 \leqslant 1$ □

The assumption of the satisfaction of the inequality $\|\Delta f_{p^*}(x^*) - \frac{\beta_2}{2}\mathbf{1}\|_2 \leqslant \gamma$ implies that $\Delta f_{p^*}(x^*)$ should be sufficiently smaller than the constant vector $\frac{\beta_2}{2}\mathbf{1}$, which can be understood as a discrete version of the zero-gradient stationary points of the continuous domain.

The constant K that appears on the proof is due to the fact that $\|\Lambda\|_1 \leqslant K\|\Lambda\|_\infty \leqslant K/2$ and that $\|\Lambda\|_2 \leqslant K\|\Lambda\|_\infty \leqslant \sqrt{K}/2$. This gives us that the norms $\|\Lambda\|_1$ and $\|\Lambda\|_2$ can become larger as the dimension K grows, which it is inevitable except if $\|\Lambda\|_\infty = 0$, which implies as we defined Λ the conventional convex optimizitation over the continuous domain.

Keeping all of the assumption of f, we can prove that local sampling methods, in concrete, TnALE converges:

Corollary 3.21. *Let $(x_n)_{n=0}^\infty$ be a succession of $\mathbb{Z}_+^K$ with x_0 randomly chosen. Suppose that p^* is known and that x_n is equal to the y of Theorem 3.20 for all $n > 0$. Then, if $\Omega(1/K) \leqslant \theta \leqslant 1$, i.e θ scales at least as $1/K$, we have that*

$$\lim_{n\to\infty}(f_{p^*}(x_n) - f_{p^*}(x^*)) = O(1)$$

Proof. Let $C_K = (7/8)K$. By Theorem 3.20,

$$\begin{aligned}
f_{p^*}(x_n) - f_{p^*}(x^*) &\leqslant (1-\theta)(f_{p^*}(x_{n-1}) - f_{p^*}(x^*)) + C_K \\
&\leqslant (1-\theta)^2(f_{p^*}(x_{n-2}) - f_{p^*}(x^*)) + C_K + C_K(1-\theta) \\
&\leqslant (1-\theta)^3(f_{p^*}(x_{n-3}) - f_{p^*}(x^*)) + C_K + C_K(1-\theta) + C_K(1-\theta)^2 \\
&\leqslant \dots \\
&\leqslant (1-\theta)^n(f_{p^*}(x_0) - f_{p^*}(x^*)) + C_K\sum_{m=1}^{n}(1-\theta)^{m-1}
\end{aligned}$$

Using that $\Omega(1/K) \leqslant \theta \leqslant 1$ we get that

$$\lim_{n\to\infty}(f_{p^*}(x_n) - f_{p^*}(x^*)) \leqslant \lim_{n\to\infty}(1-\theta)^n(f_{p^*}(x_0) - f_{p^*}(x^*)) + C_K\sum_{m=1}^{n}(1-\theta)^{m-1}$$

$$= 0 + C_K \frac{1}{\theta} = K \frac{7}{8\theta} \leqslant O(1)$$

Since in the last equality the term θ scales at least as $1/K$, therefore everything does not scale depending in K or θ, making the limit converge to some constant. □

And with this corollary we have proven that the TnALE algorithm converges. On the last corollary we have taken in account that K could also grow during the limit. In our case K would be the fixed value of our dimension that depends on the number of nodes from the graph space $\mathbb{G}$ that we are considering (See Example 3.7).

In this chapter we have seen how to search optimal structures of tensor networks that guarantee that when we search the optimal cores these minimize a loss function π_D. For achieving this we have seen first that if we fix the tensor network structure to some graph and ranks (G, R) we can apply the TN-ALS algorithm or we can use backpropagation for obtaining the cores $\mathcal{G}_1, \ldots, \mathcal{G}_n$ that minimize the loss function fixed (G, R).

Then, we have presented the TnALE algorithm, which objective is to give a general solution to the TN-RS, TN-PS and TN-TS problems, giving us an optimal tensor network structure (G, R) and saving some evaluations when testing for tensor network structure candidates.

Finally, we have proved that under certain assumptions about the loss function, the TnALE algorithm does converge to a certain structure. We have presented gradientless optimization theory to prove this fact.

On the following chapter we will see some applications of all of these algorithms combined. We will use them to compress some images and we will also use them for compressing the weights of fully connected neural network layers.

Chapter 4

Applying TN low-rank approximations on FCNN

4.1 Neural networks

Neural networks are modeled after how real neurons work, but in a simplified manner: each neural network contains a set of neurons which are split in different layers. In a fully connected neural network, each neuron has an output that is connected to each neuron of the next layer, and all neurons of a layer have as inputs the outputs of all the neurons of the previous layer. The first layer is called the input layer and the last layer is called the output layer. All the layers in between are called hidden layers.

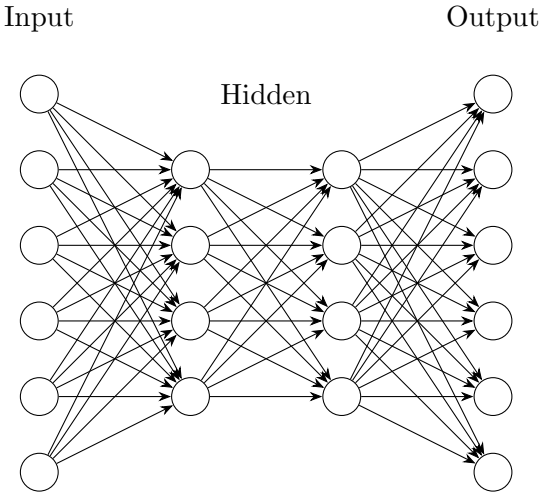


Figure 4.1: A representation of a fully connected neural network. Each dot is a neuron and each edge represents a connection. Source: own elaboration

Neural networks are modeled after how real neurons work, but in a simplified manner: each neural network contains a set of neurons which are split in different layers. A fully connected neural network is composed on 3 different parts: an **input layer**, a central part with some layers called **hidden layers** and an **output layer**. The idea of neural networks is that they recieve an encoded input through the input layer and this input

will be successively processed by the hidden layers and then it will return a certain result through the output layer. (See fig. 4.1)

A fully connected neural network is composed on 3 different parts: an input layer, a central part with some layers called hidden layers and an output layer. The idea of neural networks is that they recieve an encoded input through the input layer and this input will be successively processed by the hidden layers and then it will return a certain result through the output layer. (See fig. 4.1)

Let $n_1, \dots, n_L$ be the number of neurons on each layer. Each layer first does a pre-activation stage when it takes into account the activation of the previous layer, it adds some weights to each connection and it adds some bias to some of them.

The preactivation of the entire layer is computed by:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

Where $z^{(l)} \in \mathbb{R}^{n_l}$ is the pre-activation of the layer, $W^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix, $a^{(l-1)} \in \mathbb{R}^{l-1}$ is the activation of the previous layer and and $b^{(l)} \in \mathbb{R}^{n_l}$ is the bias of the layer l. Note that each element of the vector $z^{(l)}$ denotes the preactivation of one neuron of the layer l. Then, once the preactivation is computed, the activation of a neuron is defined by composing the preactivation with some non-linear activation function $\sigma : \mathbb{R} \to \mathbb{R}$:

$$a^{(l)} = \sigma(z^{(l)})$$

Where in $\sigma(z^{(l)})$ is the vector resulting of applying σ to each element of $z^{(l)}$. Some examples of common activation functions are the Sigmoid $S(x)$, the ReLU $R(x)$ and $T(x)$.

$$S(x) = \frac{1}{1+e^{-x}} \qquad R(x) = \max(x, 0) \qquad T(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

So, a neural network can be described as a composite function

$$N(x; \Theta) = \sigma^{(L)}(W^{(L)} \cdot \sigma^{(L-1)}(W^{(L-1)} \dots \sigma^{(1)}(W^{(1)}x + b^{(1)}) + b^{(2)} \dots) + b^{(L)})$$

Where $\Theta = \{W^{(l)}, b^{(l)}\}_{l=1}^{L}$ denotes the set of all adjustable parameters of the network. By varying Θ we can change the outputs that $N(x; \Theta)$. The parameters start at some given values and then they are adjusted usually using the backpropagation algorithm, the idea behind the backpropagation algorithm is very similar as we have described in the Subsection 3.1.2: we will have a dataset D with a series of pairs of inputs and outputs (x_i, y_i) and we will define some loss function $\pi_D : \mathbb{R}^{n_l} \to \mathbb{R}_+$ that measures the error of the output of the neural network $N(x)$ respect of the desired output y_i.

Then, exploiting the chain rule we can compute the contribution of each parameter to the total loss function $\frac{\delta \pi}{\delta W_{i,j}^{(l)}}$ and $\frac{\delta \pi}{\delta b_i^{(l)}}$. While computing these derivatives, we start by the layer $l-1$ and then compute the layer $l-2$ until we get to the input layer of neurons, while we can reuse the partial derivatives of the previous computed parameters.

Once we have all the partial derivatives computed, we can get the gradient of all the parameters and then do gradient descent onto the direction of the gradients, i.e, we update the parameters as

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \pi_D}{\partial W^{(l)}}$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial \pi_D}{\partial b^{(l)}}$$

Where η is the learning rate. The compression that we will do is that we will reshape the matrices $W^{(l)}$ as tensors of higher order (say for example of order d_l) and then represent these reshaped matrices as the element of a tensor network state $\mathrm{TNS}(G; R)$. Since obtaining the representation consists of evaluating the core tensors through the contraction mapping $\mathcal{C}(\mathcal{G}_1, \ldots, \mathcal{G}_n)$, we can apply the chain rule and compute the partial derivative of the loss function respect to each parameter of each core. In this way we can find the cores of the tensor network while training the neural network.

Another approach on compressing the neural network is to compress the weight matrices $W^{(l)}$ once the neural network is already trained using the TN-ALS algorithm. On (TODO: Citar resultado apendice) we see the results of these two different methods.

Chapter 5

Conclusions

In this thesis we have formally constructed the tensor product space, we reviewed the basics of tensor algebra, we have presented tensor networks and tensor network states and we have seen that it is posible for some tensor networks to express a tensor with less rank than its usual tensor rank. Then, we have given some algorithms for computing the cores of the tensor networks and we have also presented algorithms that search over the space of possible tensor networks for a optimal structure. Finally, we have applied these algorithms for compressing images and neural networks.

Further research on this topic would include implementing other algorithms that improve the tensor network structure search, such as finding the tensor structures using program synthesis which Zhang et. al. in [21] claim that is superior to TnALE. Finding how the structure found by this algorithm improves compression on neural networks would be a good experiment.

Another interesting topic that could be also researched is about representing big tensors by splitting its parameters into multiple low-dimension tensors and then applying the tensor network low-rank approximation with these, since we have found that computing the core tensors for big tensors is very computationally expensive.

Finally, I would like to add some words about how hard it has been as a total non initiated to write this thesis since all of it is based on recent research papers, and they are quite often not very explanatory and are very summarized. However, I liked researching on the topic of tensor networks, and I would like to do some more research on the topic of this thesis.

Bibliography

[1] Yong Qing et al. "Compressing Neural Networks Using Tensor Networks with Exponentially Fewer Variational Parameters". In: *Intelligent Computing* 4 (Jan. 2025), p. 0123. ISSN: 2771-5892. DOI: 10.34133/icomputing.0123. arXiv: 2305.06058 [cs]. URL: http://arxiv.org/abs/2305.06058 (visited on 06/06/2025).

[2] Cheng-Yao Lyu et al. "Multi-Dimensional Image Recovery via Fully-Connected Tensor Network Decomposition Under the Learnable Transforms". In: *Journal of Scientific Computing* 93.2 (Sept. 29, 2022), p. 49. ISSN: 1573-7691. DOI: 10.1007/s10915-022-02009-0. URL: https://doi.org/10.1007/s10915-022-02009-0 (visited on 06/06/2025).

[3] Chao Li et al. *Permutation Search of Tensor Network Structures via Local Sampling*. June 14, 2022. DOI: 10.48550/arXiv.2206.06597. arXiv: 2206.06597 [cs]. URL: http://arxiv.org/abs/2206.06597 (visited on 03/24/2025). Pre-published.

[4] Chao Li et al. *Alternating Local Enumeration (TnALE): Solving Tensor Network Structure Search with Fewer Evaluations*. May 29, 2023. DOI: 10.48550/arXiv.2304.12875. arXiv: 2304.12875 [cs]. URL: http://arxiv.org/abs/2304.12875 (visited on 03/24/2025). Pre-published.

[5] Román Orús. "Tensor Networks for Complex Quantum Systems". In: *Nature Reviews Physics* 1.9 (Sept. 2019), pp. 538–550. ISSN: 2522-5820. DOI: 10.1038/s42254-019-0086-7. URL: https://www.nature.com/articles/s42254-019-0086-7 (visited on 05/14/2025).

[6] Jacob Biamonte. *Lectures on Quantum Tensor Networks*. Jan. 4, 2020. DOI: 10.48550/arXiv.1912.10049. arXiv: 1912.10049 [quant-ph]. URL: http://arxiv.org/abs/1912.10049 (visited on 06/06/2025). Pre-published.

[7] *[1812.04011] Tensor Networks for Complex Quantum Systems*. URL: https://arxiv.org/abs/1812.04011 (visited on 06/06/2025).

[8] Ke Ye and Lek-Heng Lim. *Tensor Network Ranks*. Feb. 9, 2019. DOI: 10.48550/arXiv.1801.02662. arXiv: 1801.02662 [math]. URL: http://arxiv.org/abs/1801.02662 (visited on 03/24/2025). Pre-published.

[9] Christopher Hillar and Lek-Heng Lim. *Most Tensor Problems Are NP-hard*. July 1, 2013. DOI: 10.48550/arXiv.0911.1393. arXiv: 0911.1393 [cs]. URL: http://arxiv.org/abs/0911.1393 (visited on 03/31/2025). Pre-published.

[10] Matthias Christandl, Asger Kjærulff Jensen, and Jeroen Zuiddam. "Tensor Rank Is Not Multiplicative under the Tensor Product". In: *Linear Algebra and its Applications* 543 (Apr. 2018), pp. 125–139. ISSN: 00243795. DOI: 10.1016/j.laa.2017.12.020. arXiv: 1705.09379 [math]. URL: http://arxiv.org/abs/1705.09379 (visited on 05/28/2025).

[11] Roger Penrose. "Applications of Negative Dimensional Tensors". In: *Combinatorial Mathematics and its Applications* (1971), pp. 221–244. URL: https://homepages.math.uic.edu/~kauffman/Penrose.pdf (visited on 03/04/2025).

[12] Steven R. White. "Density Matrix Formulation for Quantum Renormalization Groups". In: *Physical Review Letters* 69.19 (Nov. 9, 1992), pp. 2863–2866. DOI: 10.1103/PhysRevLett.69.2863. URL: https://link.aps.org/doi/10.1103/PhysRevLett.69.2863 (visited on 03/27/2025).

[13] Melven Röhrig-Zöllner, Jonas Thies, and Achim Basermann. "Performance of the Low-Rank Tensor-Train SVD (TT-SVD) for Large Dense Tensors on Modern Multi-Core CPUs". In: *SIAM Journal on Scientific Computing* 44.4 (Aug. 2022), pp. C287–C309. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/21M1395545. arXiv: 2102.00104 [math]. URL: http://arxiv.org/abs/2102.00104 (visited on 05/28/2025).

[14] Qibin Zhao et al. *Tensor Ring Decomposition*. June 17, 2016. DOI: 10.48550/arXiv.1606.05535. arXiv: 1606.05535 [cs]. URL: http://arxiv.org/abs/1606.05535 (visited on 03/04/2025). Pre-published.

[15] Osman Asif Malik, Vivek Bharadwaj, and Riley Murray. *Sampling-Based Decomposition Algorithms for Arbitrary Tensor Networks*. Oct. 7, 2022. DOI: 10.48550/arXiv.2210.03828. arXiv: 2210.03828 [math]. URL: http://arxiv.org/abs/2210.03828 (visited on 04/30/2025). Pre-published.

[16] *PyTorch Documentation — PyTorch 2.7 Documentation*. URL: https://docs.pytorch.org/docs/stable/index.html (visited on 06/04/2025).

[17] R. Penrose. "A Generalized Inverse for Matrices". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 51.3 (July 1955), pp. 406–413. ISSN: 1469-8064, 0305-0041. DOI: 10.1017/S0305004100030401. URL: https://www.cambridge.org/core/journals/mathematical-proceedings-of-the-cambridge-philosophical-society/article/generalized-inverse-for-matrices/5F4516D6B9989BB6563A4B267CC7D615 (visited on 06/03/2025).

[18] *Automatic Differentiation with Torch.Autograd — PyTorch Tutorials 2.7.0+cu126 Documentation*. URL: https://docs.pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html (visited on 06/03/2025).

[19] Chao Li et al. *Alternating Local Enumeration (TnALE): Solving Tensor Network Structure Search with Fewer Evaluations*. May 29, 2023. DOI: 10.48550/arXiv.2304.12875. arXiv: 2304.12875 [cs]. URL: http://arxiv.org/abs/2304.12875 (visited on 05/15/2025). Pre-published.

[20] Daniel Golovin et al. "Gradientless Descent: High-Dimensional Zeroth-Order Optimization". In: International Conference on Learning Representations. Sept. 25, 2019. URL: https://openreview.net/forum?id=Skep6TVYDB (visited on 06/03/2025).

[21] Zheng Guo et al. *Tensor Network Structure Search Using Program Synthesis*. Feb. 22, 2025. DOI: 10.48550/arXiv.2502.02711. arXiv: 2502.02711 [cs]. URL: http://arxiv.org/abs/2502.02711 (visited on 05/12/2025). Pre-published.

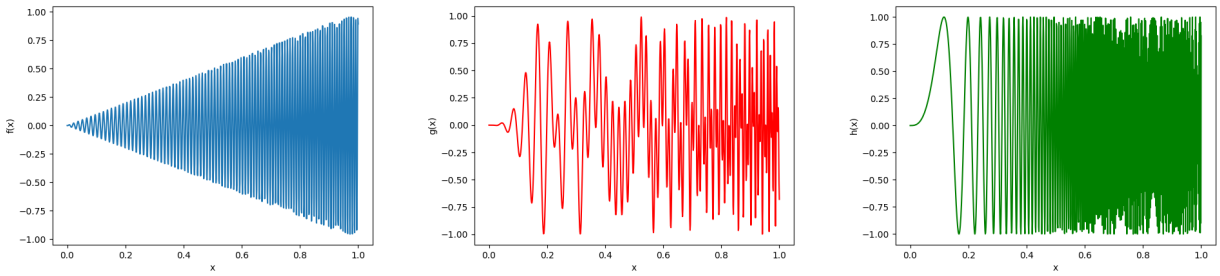
Appendix A

Experiments and results

The following experiments were performed on PyTorch running on CUDA on a system equipped with a AMD Ryzen 7 3700X 3.6GHz CPU, an NVIDIA GeForce RTX 2080 Super with 8GB of VRAM and 32GB of RAM.

A.1 Core finding algorithms

In this section we will compare the TN-ALS algorithm and the backpropagation algorithm in finding the cores that best represent a pseudorandom tensor T across different tensor network structures. To make the experiment reproducible, we will sample high oscilatory functions to form the objective tensors



$f(x) = x\sin(200(x+1)^2)$ $\quad$ $g(x) = 250x^3\cos(150x)$ $\quad$ $h(x) = \sin(1000x^3)$

Figure A.1: The three high oscilatory functions that we will use to represent objective tensors

We will pick each function and we will evaluate it over the interval $[0, 1]$ by uniformly picking N samples. We will construct the vectors

$$v_F = \left(f\left(\frac{i}{N}\right)\right)_{i=1}^{N} \qquad v_G = \left(g\left(\frac{i}{N}\right)\right)_{i=1}^{N} \qquad v_H = \left(h\left(\frac{i}{N}\right)\right)_{i=1}^{N}$$

And we will reshape this tensors onto a shape $N_1 \times \cdots \times N_n$ such that $\prod_{i=1}^{n} N_i = N$

$$T_F = \text{reshape}(v_F, (1), (N_1, \ldots, N_n)) \qquad T_G = \text{reshape}(v_G, (1), (N_1, \ldots, N_n))$$

$$T_H = \text{reshape}(v_H, (1), (N_1, \ldots, N_n))$$

We tested 6 different tensor network structures. The graphs $G_1, \ldots, G_6$ are as defined in A.2 the ranks of each graph are $R_1 = (5, 5, 5)$, $R_2 = (10, 10, 10)$, $R_3 = (7, 7, 7, 7, 7)$, $R_4 = (7, 7, 7, 7, 7, 7)$, $R_5 = (12, 12, 12, 12)$ and $R_6 = (3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$.

The results of the experiment are shown in fig. A.3.

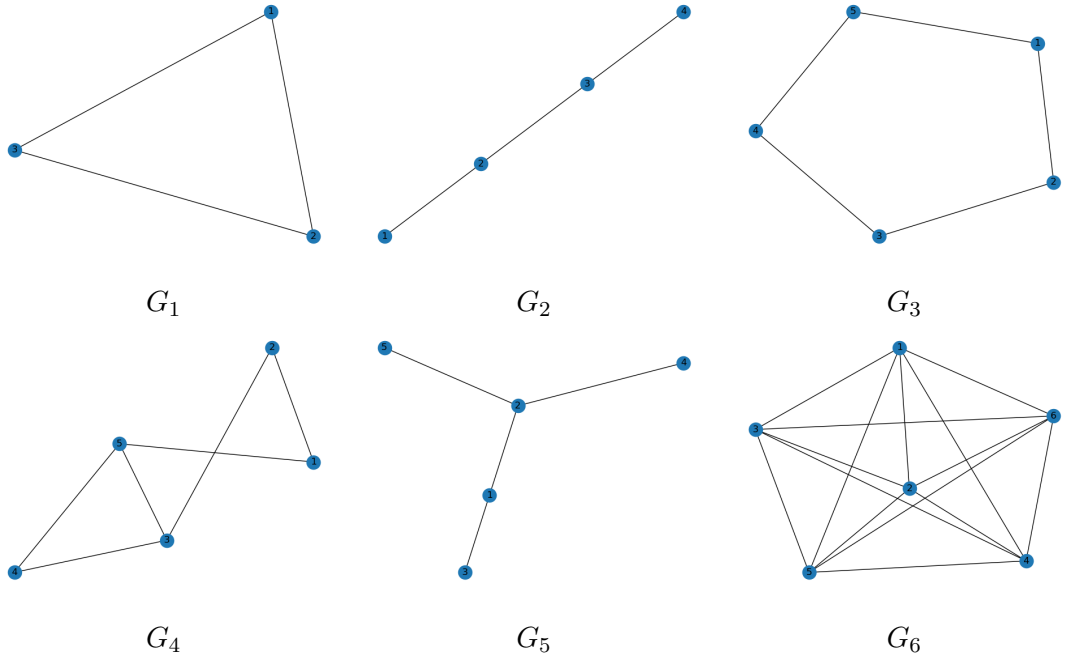


Figure A.2: The graphs $G_1, \ldots, G_6$ used for each test. Source: own elaboration.

A.2 Structure search and image compression

On the next experiment we evaluated the performance of the TnALE algorithm and the optimal structures that it finds. The objective tensor that we will approximate will be the intensity values of the pixels of an image.

An RGB image is a set of $n \times m$ pixels. Each pixel contains three channels (red, green and blue) in which a color is represented. For simplicity, in this experiment we will only encode grayscale images since grayscale images only need to store the intensity value of each pixel. Therefore, we can encode a channel as as a tensor $T \in \mathbb{R}^{n \times m}$ with each of its values being on the interval $[0, 1] \subset \mathbb{R}$.

First, we could make the order of the tensor T higher by reshaping accordingly the divisors of n or m. For example, if we have a 128×128 image, we could reshape T as tensor of $\mathbb{R}^{8 \times 8 \times 4 \times 8 \times 8}$

Then reshaping, for each channel we could pick a graph $G^{(0)}$ with the number of nodes being equal as the order of the tensors. Following our example, G should have 6 nodes. We can also pick some initial ranks $R^{(0)} = (R_1, \ldots, R_c)$. Then we can apply the ALS algorithm for finding some cores $\mathcal{G}_1, \ldots, \mathcal{G}_6$ that once contracted following $G^{(0)}$ will yield to a good approximation of T

		Time	CR	T_F Rel. Error	T_G Rel. Error	T_H Rel. Error
(G_1, R_1)	TN-ALS	5s	1.34	$\mathbf{6.9456 \times 10^{-3}}$	$\mathbf{1.0652 \times 10^{-2}}$	7.9986×10^{-2}
	Backprop			2.1491×10^{-2}	5.0026×10^{-2}	$\mathbf{6.1857 \times 10^{-2}}$
(G_2, R_2)	TN-ALS	5s	4.55	$\mathbf{1.9582 \times 10^{-4}}$	$\mathbf{8.9858 \times 10^{-5}}$	**0.113449**
	Backprop			2.8724×10^{-2}	2.8724×10^{-2}	0.115987
(G_3, R_3)	TN-ALS	5s	40.82	$\mathbf{4.7541 \times 10^{-3}}$	$\mathbf{9.5917 \times 10^{-3}}$	$\mathbf{4.4717 \times 10^{-2}}$
	Backprop			3.1899×10^{-2}	4.9388×10^{-2}	4.5730×10^{-2}
(G_4, R_4)	TN-ALS	5s	12	$\mathbf{7.1427 \times 10^{-4}}$	$\mathbf{1.6699 \times 10^{-3}}$	$\mathbf{1.1705 \times 10^{-2}}$
	Backprop			8.3855×10^{-2}	8.7846×10^{-2}	4.5468×10^{-2}
(G_5, R_5)	TN-ALS	5s	5.24	$\mathbf{4.5774 \times 10^{-2}}$	$\mathbf{6.8181 \times 10^{-2}}$	0.203061
	Backprop			5.3277×10^{-2}	7.4776×10^{-2}	**0.200155**
(G_6, R_6)	TN-ALS	10s	68.59	$\mathbf{1.9251 \times 10^{-3}}$	$\mathbf{2.4820 \times 10^{-3}}$	$\mathbf{2.0864 \times 10^{-2}}$
	Backprop			0.880255	0.751659	0.572296

Figure A.3: Relative error obtained by running the TN-ALS algorithm and the back-propagation algorithm for approximating the tensors T_F, T_G and T_H fot the same fixed amount of time. The CR column denotes the compression ratio of each structure. In each test the result with less relative error is marked in bold test. Source: own elaboration

Now, we apply the TnALE algorithm for finding if there is a more optimal structure than $(G^{(0)}, R^{(0)})$. Once we found a good structure candidate (G^*, R^*) we then perform the TnALS algorithm using this structure.

Then, to recover the channel back we contract all the core tensors and then undo the reshape operation to recover the matrix that encodes the image channel. Following our example, we would reshape the tensor $T \in \mathbb{R}^{8\times8\times4\times8\times8}$ obtained by contracting the cores found by TnALS into a matrix of $\mathbb{R}^{128\times128}$.

The main objective of this experiment is to demonstrate the compression capabilities of the optimal structure of the TnALE algorithm by doing image compression. For measuring the error we used the relative error of the objective tensor representing the grayscale channel of the original tensor respect to the tensor that the contraction of the cores yields.

In this experiment we compressed the image `files/fruits.png` grayscaled and resized as a 128×128 image. We fixed 5 initial pairs of initial structures $(G^{(1)}, R^{(1)}), \ldots, (G^{(5)}, R^{(5)})$. For each structure we first computed the core tensors using the TnALS algorithms directly, and then we made the TnALE algorithm find a better structure by varying $\lambda \in \{0.25, 0.5, 1, 2, 4, 8, 12\}$. We use the TnALE algorithm two times: first, we make broad search by doing 3 iterations of it with radius 5 that are able to modify the topology of the original graph G_0. We made that if the algorithm decided to add an edge, it will try doing so by putting initial ranks $\{1, 2, 5\}$. Then, we do again the TnALE algorithm with 3 iterations with radius 2 and we let G fixed so that the structure can be relaxed to adjust its ranks. On both TnALE algorithms, for evaluating the structure we use the TnALS algorithm with 5 iterations applying the Tikhonov factor for faster computation. Since we can't be sure if the Tikhonov factor has converged in 5 iterations, we make an approximation of the real relative error that would yield the structure by picking the

minimum value of the relative error in these 5 iterations. After we have found an optimal structure, we finally perform the TnALS algorithm with 200 iterations.

The starting graphs $G^{(1)}, \ldots, G^{(5)}$ that we have picked can be seen in the first row of fig. A.5. We will also describe explicitly the order of the starting edges $E^{(1)}, \ldots, E^{(5)}$ and the starting ranks $R^{(1)}, \ldots, R^{(5)}$:

- $E^{(1)} = ((1,2),(2,3),(3,4),(4,5))$
  $R^{(1)} = (8,8,8,8)$
- $E^{(2)} = ((1,2),(2,3),(3,4),(4,5),(5,1))$
  $R^{(2)} = (8,8,8,8,8)$
- $E^{(3)} = ((1,2),(2,3),(3,4),(4,5),(5,1),(3,5),(1,4))$
  $R^{(3)} = (6,6,6,6,6,6,6)$
- $E^{(4)} = ((1,2),(2,3),(3,4),(4,5),(5,1),(1,4),(5,2),(3,5))$
  $R^{(4)} = (4,4,4,4,4,4,4,4)$
- $E^{(5)} = ((1,2),(2,3),(3,4),(4,5),(5,1),(5,2),(5,3),(4,1),(4,2),(3,1))$
  $R^{(5)} = (3,3,4,3,3,4,3,3,4,3)$

The final ranks that produced each execution can be found inside the text files of the folder `files/img-compression/`

A.3 Neural Network compression

		CR	Rel. Error	TN-ALS	TN-ALE First Pass	TN-ALE Second Pass
$(G^{(1)}, R^{(1)})$	Original	11.63	0.13159	3.08s		
	$\lambda = 0.25$	455.11	0.19703	3.08s	4.84s	0.23s
	$\lambda = 0.5$	105.02	0.16981	1.95s	6.65s	1.56s
	$\lambda = 1$	34.13	0.13397	2.45s	7.49s	2.97s
	$\lambda = 2$	23.81	0.12605	2.40s	9.81s	2.99s
	$\lambda = 4$	10.06	0.10416	3.24s	22.85s	4.66s
	$\lambda = 8$	4.61	0.07901	6.43s	55.36s	10.77s
	$\lambda = 12$	2.16	0.05304	14.10s	122.91s	20.13s
$(G^{(2)}, R^{(2)})$	Original	7.11	0.09592	3.56s		
	$\lambda = 0.25$	455.11	0.19703	1.39s	5.65s	0.64s
	$\lambda = 0.5$	50.56	0.14933	2.09s	6.07s	2.92s
	$\lambda = 1$	32.51	0.13736	2.29s	6.68s	3.63s
	$\lambda = 2$	18.62	0.12109	2.77s	8.55s	4.22s
	$\lambda = 4$	9.62	0.10570	3.45s	17.28s	4.98s
	$\lambda = 8$	3.52	0.07178	8.65s	190.7s	11.59s
	$\lambda = 12$	2.12	0.05678	15.39s	143.14s	29.29s
$(G^{(3)}, R^{(3)})$	Original	2.58	0.06481	9.81s		
	$\lambda = 0.25$	240.94	0.18556	1.81s	6.55s	3.20s
	$\lambda = 0.5$	102.4	0.16595	2.15s	6.51s	2.44s
	$\lambda = 1$	47.08	0.14801	2.40s	8.67s	4.29s
	$\lambda = 2$	18.12	0.12340	2.76s	17.66s	5.79s
	$\lambda = 4$	12.41	0.11887	3.18s	15.78s	5.15s
	$\lambda = 8$	3.53	0.07238	10.19s	61.14s	22.02s
	$\lambda = 12$	2.11	0.04997	21.57s	284.93s	29.51s
$(G^{(4)}, R^{(4)})$	Original	4.26	0.08985	5.46s		
	$\lambda = 0.25$	113.78	0.17616	2.03s	11.63s	2.39s
	$\lambda = 0.5$	89.04	0.16479	1.92s	11.59s	2.62s
	$\lambda = 1$	40.96	0.14096	2.26s	11.62s	2.78s
	$\lambda = 2$	24.38	0.13249	2.45s	11.99s	3.11s
	$\lambda = 4$	11.91	0.11154	3.11s	12.73s	6.07s
	$\lambda = 8$	3.28	0.06162	6.84s	28.71s	13.91s
	$\lambda = 12$	2.75	0.06293	8.81s	70.02s	17.61s
$(G^{(5)}, R^{(5)})$	Original	3.92	0.07800	6s		
	$\lambda = 0.25$	204.8	0.18553	1.95s	13.86s	3.58s
	$\lambda = 0.5$	70.62	0.15677	2.14s	13.89s	3.97s
	$\lambda = 1$	39.38	0.14342	2.38s	13.96s	4.92s
	$\lambda = 2$	16.38	0.12128	2.92s	14.62s	5.46s
	$\lambda = 4$	9.57	0.10488	3.39s	14.78s	6.38s
	$\lambda = 8$	6.82	0.09066	3.89s	21.7s	12.79s
	$\lambda = 12$	2.33	0.05597	10.43s	58.08s	22.48s

Figure A.4: Table containing the results of the TnALE evaluation compressing the pepper.png image. Source: own elaboration

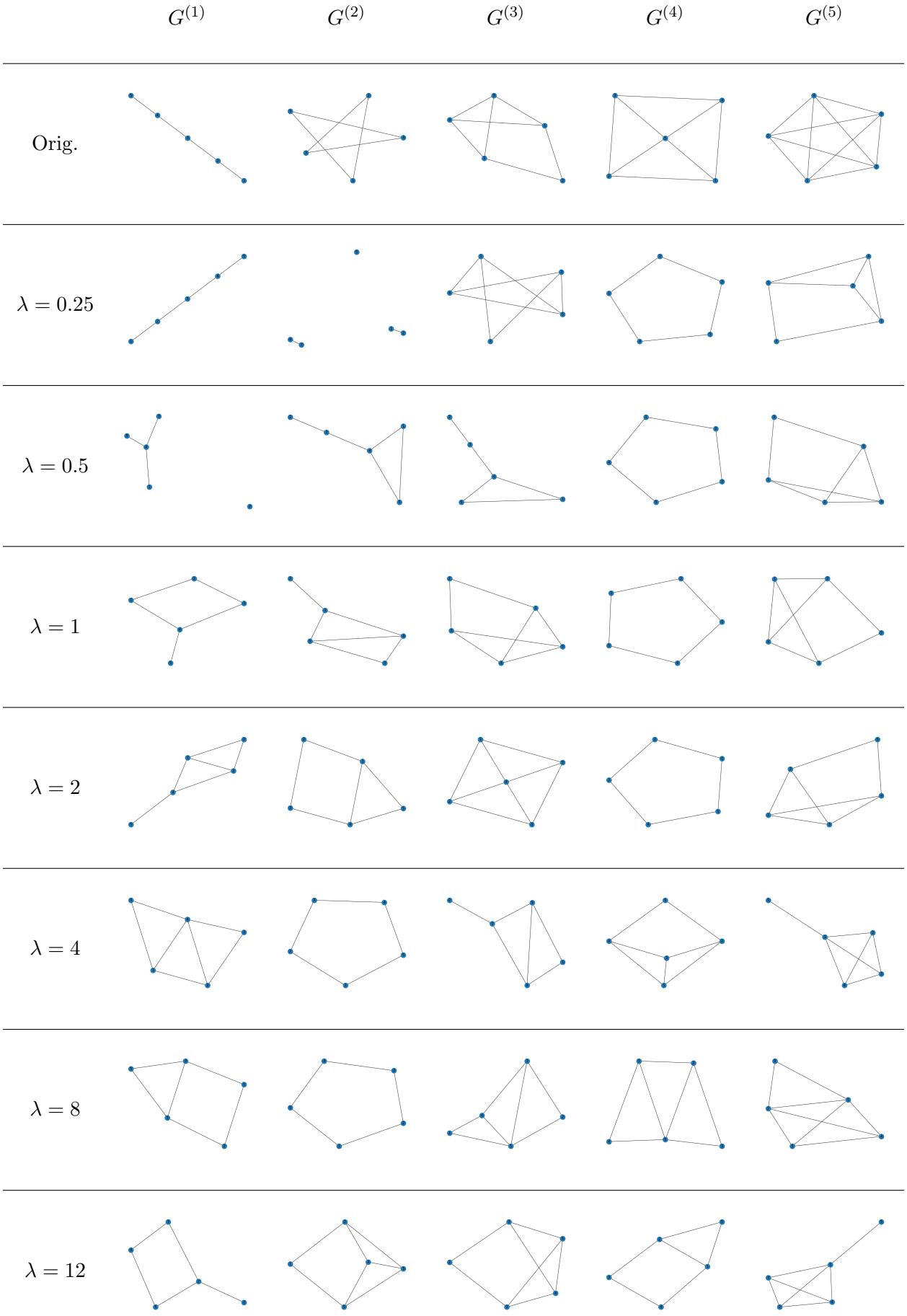


Figure A.5: The structures G^* found by the TnALE algorithm. Source: own elaboration.

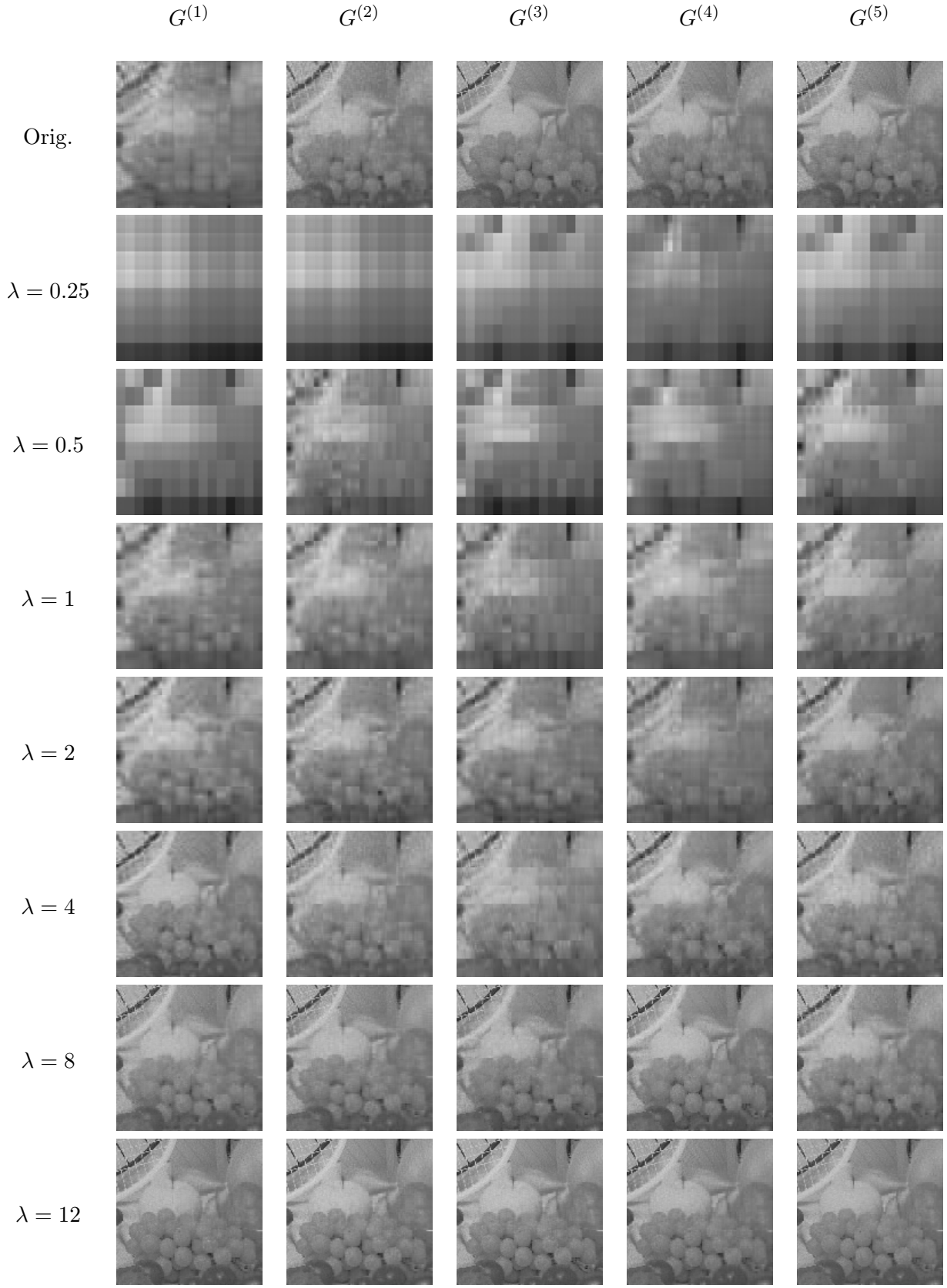


Figure A.6: The images produced by the contraction of the core tensors that the TnALS algorithm produced from the optimal structure that TnALE found for each value of λ for each structure on fig. A.5. Source: own elaboration.