DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Canonicalization of Loop-free Tensor Networks

David A. Tellenbach

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Canonicalization of Loop-free Tensor Networks

Kanonisierung Schleifenfreier Tensornetzwerke

Author:	David A. Tellenbach
Supervisor:	Prof. Dr. rer. nat. Christian Mendl
Advisor:	M.Sc. Qunsheng Huang
Submission Date:	15.03.2022

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 12.03.2022 David A. Tellenbach

Abstract

Many important tensor network algorithms can benefit from orthogonality constraints on the tensors of the network or even require that tensors form isometries when considered as matrices. We explore different methods to orthogonalize tensor networks and present a canonical form that can be used to flexibly shift centers of orthogonality.

After methods to orthogonalize tensor networks have been established, we clarify how a network can be considered as an element of a product of Riemannian manifolds to eventually introduce a modified line-search method based on gradient descent that can be used to minimize complex-valued functions on tensor networks.

Contents

List of Figures

List of Tables

List of Algorithms

List of Source Codes

List of Abbreviations

CG	Conjugate gradient
DMRG	Density-matrix renormalization group
MPS	Matrix product state
RGD	Riemannian gradient descent
SVD	Singular value decomposition
TT	Tensor train

CHAPTER 1

Introduction

A tensor network is a graph with tensors as nodes and tensor contractions as edges. Although being a relatively simple representation of tensors and contractions between them, tensor networks have proven to be an efficient tool for a whole range of interesting applications, such as the simulation of quantum computers [Che+18; Hua+20], the description of quantum many-body systems [CV09; VCM09] or big data processing [Cic14].

Besides being an expressive tool for understanding large systems of tensors, tensor networks have further reaching practical advantages. A canonical example to see the strength of tensor network ansätze is the description of a quantum spin state on a one-dimensional lattice with $N \in \mathbb{N}$ sites. Instead of describing the entire system as one large state, each lattice site can be described as a three-dimensional tensor where two of the dimensions are connected to adjacent sites. The resulting tensor network is called a *matrix product state* or MPS. For a relevant family of systems, it can be shown that a truncated MPS can express the state of the whole systems with a lesser number of coefficients which leads to a real computational advantage [BC17; Has07].

Many interesting algorithms in all of the aforementioned areas can benefit from orthogonality constraints on the tensors in a network, rising the necessity for transformations that enforce such constraints on general tensor networks.

This thesis introduces procedures that transform arbitrary loop-free tensor networks into a canonical form, enforcing certain properties on the network tensors. We then subsequently present two applications that benefit from networks in canonical form.

In chapter 2 we introduce the necessary theory of linear algebra and fix the notation of tensor networks to present different orthogonalization procedures in chapter 3. The third chapter also introduces canonical forms for loop-free tensor networks. Chapter 4 then introduces the application of canonical forms to optimal truncation of edges in a tensor network and the optimization of functions on Riemannian manifolds.

We implemented all methods introduced in this work in the programming language Python and finish the thesis with an overview of the implementation and a final discussion of some numerical results.

CHAPTER 2

Mathematical Formalism

This chapter gives a brief introduction to the mathematical formalism we will need in the further course of this thesis. We first give a recap of linear algebra basics and subsequently introduce the notation of tensor networks.

2.1. Linear Algebra

Linear algebra is a branch of mathematics that deals with finite dimensional vector spaces and linear mappings between them. The topic is covered in a wide range of introductory textbooks such as [GL13] or [Str13] to name just two.

This section gives a brief recap of linear algebra basics and introduces a few matrix decompositions that will be useful later. For our purpose, all vector spaces we consider have finite dimension and are defined over the field of complex numbers $\mathbb{C}$. Since any vector space over $\mathbb{C}$ of dimension n is isomorphic to $\mathbb{C}^n$, we consider spaces of the latter form only.

2.1.1. Basic concepts

Vectors and matrices An element of a vector space $\mathbb{C}^n$ is called a *vector* and is denoted by a small letter such as v or u. A *homomorphism* is a linear mapping $f : \mathbb{C}^n \to \mathbb{C}^m$ between two vector spaces that preserves the structure of the spaces and is fully specified by its action on some basis vectors. We can therefore consider f as a matrix $M \in \mathbb{C}^{m \times n}$ where each

column of M is the image of f on the standard basis $e_i \in \mathbb{C}^n$, $i = 1, \ldots, n$. On the other hand, each matrix gives rise to a homomorphism such that we uniquely identify matrices with homomorphisms.

Special matrices Let $M \in \mathbb{C}^{m \times n}$ be a matrix with coefficients $m_{ij} \in \mathbb{C}$ for $i = 1, \ldots m$ and $j = 1, \ldots n$. The *transposed* matrix $M^T = (m'_{ij})$ has coefficients $m'_{ij} = m_{ji}$. The *complex conjugated* matrix $M^* = (m'_{ij})$ is the matrix given by $m'_{ij} = m^*_{ij}$ where $*$ denotes complex conjugation and the *adjoint matrix* $M^\dagger$ is defined as $M^\dagger = (M^*)^T$.

A square matrix $H \in \mathbb{C}^{n \times n}$ is *Hermitian* if $H^\dagger = H$ and *unitary* if $H^\dagger H = I_n$. Equivalently a square matrix H is unitary, if $H^{-1} = H^\dagger$.

Inner product and norm The vector spaces $\mathbb{C}^n$ can be equipped with the usual dot product

$$\langle , \rangle \colon \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}, (x, y) \mapsto xy^*$$

as an inner product which induces the 2-norm

$$||x||_2^2 = \langle x, x \rangle.$$

For matrices $A \in \mathbb{C}^{m \times n}$ we define the *Frobenius norm*, which is the 2-norm of the vectorization of A, in other words

$$||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}}.$$

In the future we will omit the subscripts for both, the 2-norm for vectors and the Frobenius norm for matrices.

Isometry We call a matrix $A \in \mathbb{C}^{m \times n}$ an *isometry* if it preserves the above norm, i.e., if

$$||Av|| = ||v||, \quad \forall v \in \mathbb{C}^m.$$

This gives rise to the equivalent definition of an isometry that we are using as our primary one during the course of this thesis: A matrix $A \in \mathbb{C}^{m \times n}$ is an isometry if $A^\dagger A = I_n$. Note that an isometric matrix is unitary if it is square, i.e., if $n = m$.

2.1.2. Matrix Decompositions

Matrix decompositions are factorizations of matrices into a product of other matrices and are fundamental for a lot of different applications, ranging from approximation of matrices to solving linear equations or finding eigenvalues. We discuss three decompositions, the spectral decomposition, the QR decomposition and the singular value decomposition which we will use later to transform tensor networks.

Spectral decomposition The spectral or eigen decomposition of a square matrix $A \in \mathbb{C}^{n\times n}$ is applicable if A has n linearly independent eigenvectors.

Theorem 2.1 (Spectral decomposition)**.** *Let $A \in \mathbb{C}^{n\times n}$ be a square matrix with n linearly independent eigenvectors. Then*

$$A = U\Delta U^{\dagger},$$

where $U \in \mathbb{C}^{n\times n}$ is a unitary matrix which $i-$th column is the $i-$th eigenvector of A and $\Delta \in \mathbb{C}^{n\times n}$ is a diagonal matrix with the $i-$th eigenvalue on the $i-$th diagonal entry.

Proof. See [GL13, p. 67]. □

QR decomposition A QR decomposition is a decomposition of matrices that always exists, regardless of the structure of the matrix to be decomposed. It factorizes a matrix into a product of a unitary matrix Q and an upper triangular matrix R, thus the name.

Theorem 2.2 (QR decomposition)**.** *If $A \in \mathbb{C}^{m\times n}$, then there exists a unitary matrix $Q \in \mathbb{C}^{m\times m}$ and an upper triangular matrix $R \in \mathbb{C}^{m\times n}$ such that*

$$A = QR. \tag{2.1}$$

Proof. See [GL13, p. 247]. □

In general, the QR decomposition of a matrix is not unique as R might contain rows of 0. In this case one can consider the *thin* QR decomposition given by

$$A = Q_1 R_1 \quad \text{with} \quad A = \underbrace{\begin{pmatrix} Q_1 & Q_2 \end{pmatrix}}_{=Q} \underbrace{\begin{pmatrix} R_1 \\ 0 \end{pmatrix}}_{=R}.$$

If A has full rank and the diagonal elements of R_1 are fixed to be positive, then R_1 and Q_1 are unique.

Similarly to the QR decomposition, it is also possible to decompose a matrix into a product RQ where R is again upper triangular and Q is again unitary. In fact, the implementation of both decompositions are just slightly different [And+99]. We call this modified QR decomposition the *RQ decomposition*.

Singular value decomposition (SVD) SVD is one of the most widely used matrix decompositions and has some remarkable properties. As for the QR decomposition, there are no requirements on the matrix structure.

Theorem 2.3 (Singular value decomposition)**.** *Let $A \in \mathbb{C}^{m\times n}$ be a complex matrix. Then there exist unitary matrices*

$$U \in \mathbb{C}^{m\times m} \quad \textit{and} \quad V \in \mathbb{C}^{n\times n}$$

and a matrix

$$\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m\times n}, \quad p = \min(m, n)$$

with $\sigma_1 \geqslant \sigma_2 \geqslant \ldots \geqslant \sigma_p$ such that

$$A = U\Sigma V^\dagger.$$

Proof. See [GL13, p. 76]. □

We call the σ_i for $i = 1, \ldots, p$ the *singular values* of A, the columns of U *left-singular vectors* of A and the columns of V *right-singular vectors* of A, respectively.

The singular value decomposition of a matrix is *rank-revealing*, that means we can read of the rank of the decomposed matrix A. It is given as the number of non-zero singular values. This leads to the interpretation of theorem 2.3, that the SVD decomposes an arbitrary matrix into a sum of r rank-1 pieces, given by

$$A = \sum_{k=1}^{r} u_k \sigma_k v_k^\dagger,$$

where $r \in \mathbb{N}$ is the rank of the matrix A.

SVD has a bunch of important properties. Here we focus on its usage to optimally approximate a matrix using *low-rank approximation* which can be formulated as

Defintion 2.1 (Low-rank approximation)**.** Let $A \in \mathbb{C}^{m\times n}$ be a matrix and $A = U\Sigma V^\dagger$ its singular value decomposition. Let σ_i be the i-th singular value of A, u_i the i-th left-singular vector and v_i the i-th right-singular vector, respectively. Then

$$A^k = \sum_{i=1}^{k} \sigma_i u_i v_i^\dagger \tag{2.2}$$

is a matrix of rank k, the *k-rank approximation* of A.

The importance of low-rank approximations is due to the following theorem by Eckard and Young [EY36]:

Theorem 2.4 (Eckart–Young)**.** *For a matrix $A \in \mathbb{C}^{m \times n}$, the k-rank approximation A^k is the best approximation of rank k with respect to the Frobenius norm, i.e.,*

$$\min_{\substack{B \in \mathbb{C}^{m \times n} \\ \text{rank(B)}=k}} ||A - B|| = ||A - A^k||. \tag{2.3}$$

Proof. See [GL13, p. 79]. □

Theorem 2.4 states that the best approximation of a matrix of rank k is given by the first k rank-1 pieces resulting from its SVD.

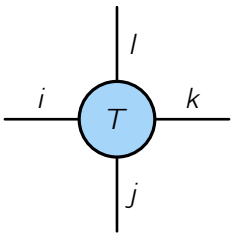
2.2. Tensor Networks

The first ideas to represent tensors and operations between them in the form of graphical diagrams date back to Roger Penrose who developed first ideas in the article "Applications of negative dimensional tensors" [Pen71]. This section gives an introduction to modern graphical notation for tensor networks which turns out to be surprisingly helpful in many cases.

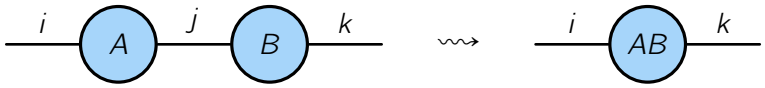
Besides many resources on the web, comprehensive introductions to the topic can be found in [Bia19], [BB17] or [BC17]. The following chapter gives a summary of the key results.

2.2.1. Graphical Diagrams

Given a tensor T of degree N, we denote T as a circle with N outgoing legs, each leg representing one dimension of the tensor. E.g., for $N = 4$

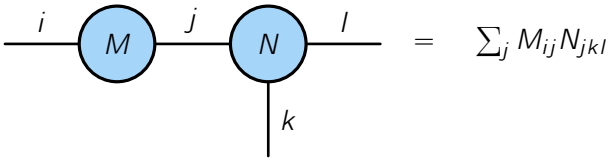


Contraction of tensors is denoted by connecting edges representing indices we want to contract along. A standard matrix multiplication of two matrices $A \in \mathbb{C}^{m \times n}$ and $B \in \mathbb{C}^{n \times p}$ can thus be denoted as



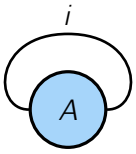
Here we denoted the index corresponding to dimension m as i, to dimension n as j and to dimension p as k. Using such a simple graphical representation immediately enforces that the second dimension of A and the first dimension of B must match.

The above notation is consistent for arbitrary tensor contractions as the following example illustrates:

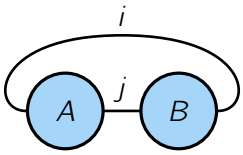


The identity matrix is denoted as a simple *line* since contracting with the identity matrix has no effect. One can imagine our networks to be free of unnecessary identity matrices in the sense that those have already been absorbed into other tensors.

The trace of a matrix $\mathrm{tr}[A]$ is denoted by connecting legs that we want to *trace out*. In the simplest case of a matrix the standard matrix trace is given by

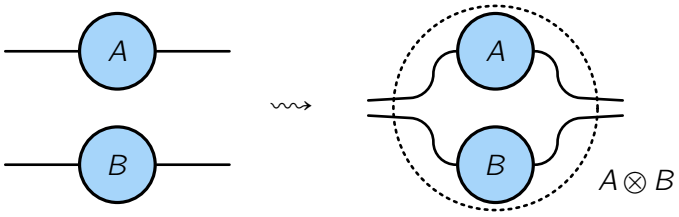


Again, we immediately see that taking the trace of a matrix requires it to be square since both dimensions need to match. Using this notation, we can denote the trace of arbitrary complex networks:



Here we *traced out* the first dimension of A and the second dimension of B to arrive at the so called *partial trace* of both tensors.

Neighboring tensors that are not necessarily connected, form an implicit Kronecker product by grouping legs together:



2.2.2. Matricization

Many interesting operations are defined for matrices only, while our usual objects of interest are tensors of arbitrary finite dimension. To be able to apply matrix methods, we often have to group tensor legs together to form a two-dimensional object. This process is known as *matricization* which we can identify with a procedure, that takes as inputs a tensor we want to consider as a matrix, a list of legs we want to identify as the matrix rows and a list of legs we want to identify as the matrix columns (see algorithm 1).

Algorithm 1: Matricization of a tensor

Input: Tensor $T \in \mathbb{C}^{n_1 \times \ldots \times n_k}$, row-legs $\{r_1, \ldots, r_l\} \subsetneq \{1, \ldots, k\}$, col-legs $\{c_1, \ldots, c_m\} \subsetneq \{1, \ldots, k\}$

Output: Matrix $M \in \mathbb{C}^{\prod_{i=1}^{l} n_{r_i} \times \prod_{i=1}^{m} n_{c_i}}$

1 $\sigma \leftarrow \{r_1, \ldots, r_l, c_1, \ldots, c_m\}$;
2 $T \leftarrow$ transpose legs of T according to σ;
3 $M \leftarrow$ reshape T to $\left(\prod_{i=1}^{l} n_{r_i}, \prod_{i=1}^{m} n_{c_i}\right)$;
4 **return** M;

We call the reverse procedure of reshaping a matrix back to a known tensor as *tensorization*. In contrast to matricization we also need to receive the original tensor's dimensions as an input. Algorithm 2 sketches the tensorization procedure where the permutation σ^{-1} is the inverse of the permutation σ in the symmetric group.

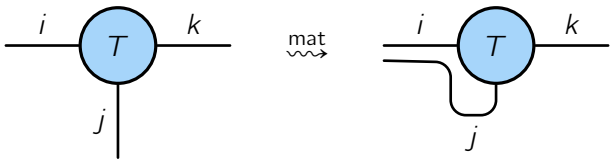
Algorithm 2: Tensorization of a matrix

Input: Matrix $M \in \mathbb{C}^{m \times n}$, row-legs $\{r_1, \ldots, r_l\} \subsetneq \{1, \ldots, k\}$, col-legs $\{c_1, \ldots, c_m\} \subsetneq \{1, \ldots, k\}$, tensor dimensions $\{n_1, \ldots, n_k\}$

Output: Tensor $T \in \mathbb{C}^{n_1, \ldots, n_k}$

1 $\sigma \leftarrow \{r_1, \ldots, r_l, c_1, \ldots, c_m\}$;
2 $d \leftarrow$ permute $\{n_1, \ldots, n_k\}$ according to σ;
3 $T \leftarrow$ reshape M to d;
4 $T \leftarrow$ transpose legs of T according to σ^{-1};
5 **return** T;

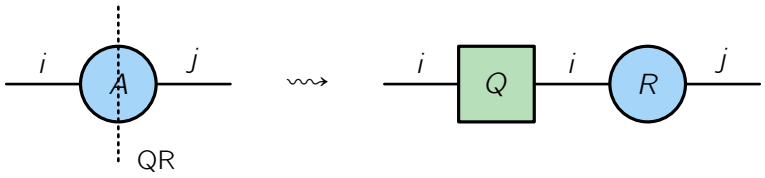
Using graphical tensor network notation, matricization of a tensor is denoted as graphically grouping tensor legs together, e.g. as



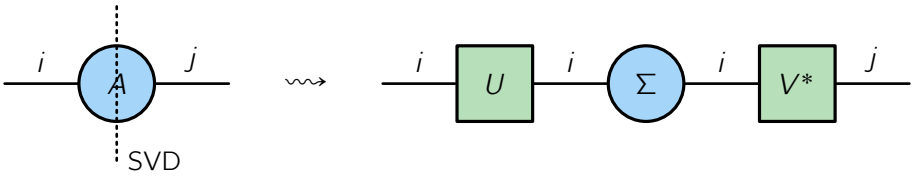
2.2.3. Decompositions

Section 2.1.2 discussed how to decompose a matrix $A \in \mathbb{C}^{m \times n}$ into a unitary matrix $Q \in \mathbb{C}^{m \times m}$ and an upper triangular matrix $R \in \mathbb{C}^{m \times n}$. We now want to apply the same decomposition to the now established graphical representation in form of tensor networks. To make clear that the matrix Q from the QR decomposition is unitary, we'll represent it as a green rectangle in contrast to our representation of general tensors as blue circles. In the future we will denote general isometries as green rectangles.

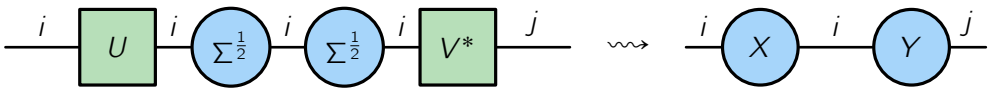
Given a matrix A we can form its QR decomposition graphically as



Similarly we can graphically apply a SVD to a tensor in a network:



SVD can also be used to split a tensor in a network by first performing a SVD as shown above and then multiplying both resulting unitary matrices with $\Sigma^{1/2}$. Since Σ is a diagonal matrix by construction, taking its square-root is as easy as taking the coefficient-wise square root of its diagonal entries. Graphically we get

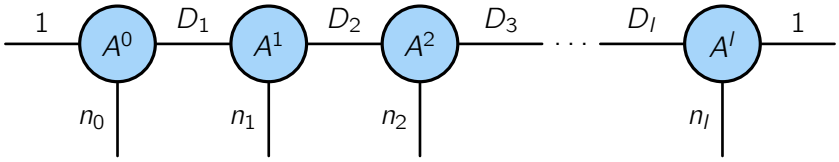


It should be noted that the resulting tensors are in general not unitary anymore after $\Sigma^{\frac{1}{2}}$ has been absorbed.

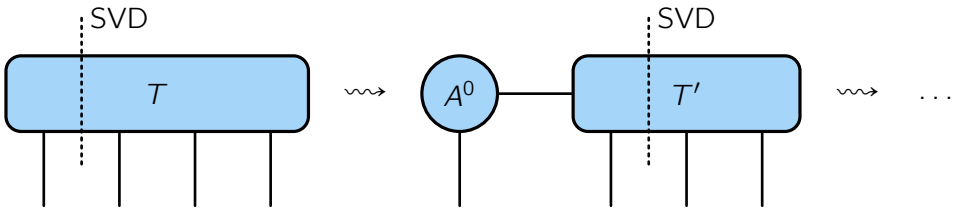
Both, QR decomposition and SVD are defined for matrices only. In the case of arbitrary tensors, these have to be reshaped into matrices first, as shown in subsection 2.2.2. The decision which tensor legs are grouped together to form a matrix that can be decomposed is crucial, since properties of the individual factors are only given under the grouping of legs used to perform the decomposition. In particular, isometries are only preserved under a certain grouping of legs.

2.2.4. Matrix Product States

Matrix product states (MPS) or tensor trains (TT) are a form of tensor network that arises when factorizing a tensor of dimension N into a chain of tensors, each of them being three-dimensional.
A (finite) MPS consists of l rank 3 tensors $A^l \in \mathbb{C}^{D_l \times n_l \times D_{l+1}}$ with $D_0 = 1 = D_{l+1}$. The legs n_i for $i = 0, \ldots, l$ are called *physical bonds*, the legs D_i for $i = 0, \ldots, l+1$ connecting the individual MPS tensors are called *virtual bonds*.



One can interpret an MPS to originate in subsequent decomposition of a larger tensor, e.g.,



CHAPTER 3

Canonical Tensor Networks

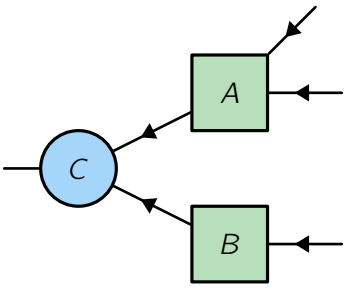
In this chapter we develop the theory of canonical forms for loop-free tensor networks, which is a certain form of a network that is well-suited for further usage in different tensor network algorithms. Canonical forms have been described in [Oru13], [Vid07] and [Vid03]. While we focus on loop-free tensor networks, the canonicalization procedures can be generalized to networks containing closed loops as shown in [Eve18].

We start by fixing some definitions and continue to present different methods to orthogonalize tensor networks, in particular using QR decompositions in section 3.1 and branch densities in section 3.2, to then subsequently introduce general canonical forms in section 3.4.

Consider a loop-free tensor network with tensors $T_1, \ldots, T_k$. We want to fix the notation of tensor being a *center of orthogonality*.

Defintion 3.1 (Center of orthogonality)**.** A tensor T_k in a loop-free tensor network is a *center of orthogonality* if every branch of T_k is an isometry after matricization with edges attached to T_k forming the column legs and all other edges forming the row legs.

Having a fixed center of orthogonality in a network is a crucial property and in some sense gives the network orientation if we consider all edges being oriented towards the center of orthogonality. For any tensor T_k that is not the center, we can then group incoming and outgoing edges together to arrive at an isometry. The following network illustrates this fact:



Tensor C is the center of orthogonality, the tensors A and B are isometries, when reshaped according to the arrows on the edges.

We call the process of transforming a network in a way that results in the existence of a center of orthogonality, *orthogonalization*.

3.1. Orthogonalization via QR Decomposition

As discussed in section 2.1.2 and section 2.2.3, the QR decomposition of a matrix factorizes it into a unitary matrix Q and an upper triangular matrix R. Since unitary matrices are closed under multiplication, using the unitary factor from a QR decomposition is a promising approach for orthogonalization of a network.

Orthogonalization via QR or RQ decomposition works by repeatedly decomposing tensors and absorbing the resulting upper triangular factor into neighboring tensors until the proposed center of orthogonality is reached. All tensors but the center of orthogonality are the unitary factors Q from the QR or RQ decomposition and thus isometries.

The choice of using a QR or a RQ decomposition depends on the position of a tensor relative to the new center of orthogonality.

However, this approach has some limitations over other orthogonalization procedures we will discuss in the next sections. Namely, it requires to fix the center of orthogonality before starting the orthogonalization procedure and is computationally costly since a QR or RQ decomposition has to be performed for every tensor but the center of orthogonality.

3.2. Direct Orthogonalization

Direct orthogonalization is another orthogonalization approach that forms the basis for transforming a tensor network into its canonical form. It relies on transforming the network by working with branch densities.

Defintion 3.2 (Branch density)**.** Given a tensor network and a tensor T in the network. Let P be a branch in the network, starting at T and M_P the matricization of P with the leg connecting P to T being the column legs and all other open legs being the row legs. The *branch density* ρ_P of P is the product

$$\rho_P = M_P^\dagger M_P.$$

The branch density ρ_P is a matrix since the open legs after forming the product of the branch are exactly the two legs connected to the beginning of the branch. Furthermore

Lemma 3.1. *For any branch P in a tensor network, the branch density ρ_P is a Hermitian matrix.*

Proof. The lemma holds for all matrices $M \in \mathbb{C}^{m\times n}$ and thus in particular for the branch density. Let $m_{ij} \in \mathbb{C}$ denote the $ij-$th coefficient of M and ρ_{ij} the $ij-$th coefficient of $\rho = M^\dagger M$, then $M^\dagger M$ is given by

$$\begin{aligned}
\rho_{ij} &= \sum_{k=1}^{m} m_{ki}^* m_{kj} \\
&= \left(\sum_{k=1}^{m} m_{ki} m_{kj}^* \right)^* \\
&= \left(\sum_{k=1}^{m} m_{kj}^* m_{ki} \right)^* \\
&= \rho_{ji}^*
\end{aligned}$$

and ρ is Hermitian. □

The direct orthogonalization procedure now works as follows: Consider a loop-free tensor network and a tensor T_0. For all branches $b_1, \ldots, b_l$ that originate in T_0, compute the branch density $\rho_{b_i} = b_i^\dagger b_i, i = 0, \ldots, l$. By lemma 3.1 ρ_{b_i} is Hermitian and thus has a spectral decomposition with real eigenvalues only, i.e.,

$$b_i^\dagger b_i = \rho_{b_i} = U_{b_i} \Delta_{b_i} U_{b_i}^\dagger,$$

where U_{b_i} is unitary. We now want to compute the *principal square root* $\Delta_{b_i}^{\frac{1}{2}}$ of Δ_{b_i}. Since Δ_{b_i} is a diagonal matrix with real coefficients, we can form its square root by simply taking the square root of its diagonal entries, i.e., the coefficients are given by

$$\left(\Delta_{b_i}^{\frac{1}{2}}\right)_{ij} = \begin{cases} \sqrt{\left(\Delta_{b_i}\right)_{ij}}, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$$

Define

$$X_{b_i} := \Delta_{b_i}^{\frac{1}{2}}.$$

Again, since Δ_{b_i} is a diagonal matrix, we have

$$X_{b_i}^{-\frac{1}{2}} = \Delta_{b_i}^{-\frac{1}{2}}.$$

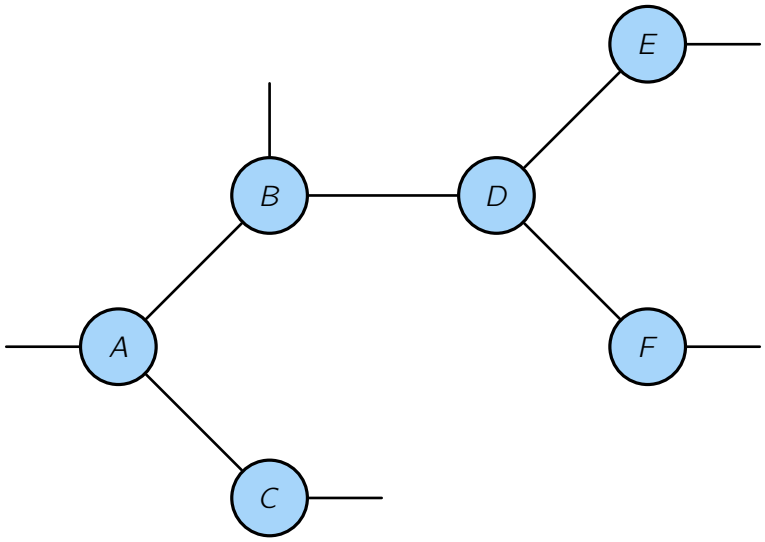
The orthogonalization procedure is then given by algorithm 3.

Algorithm 3: Direct orthogonalization

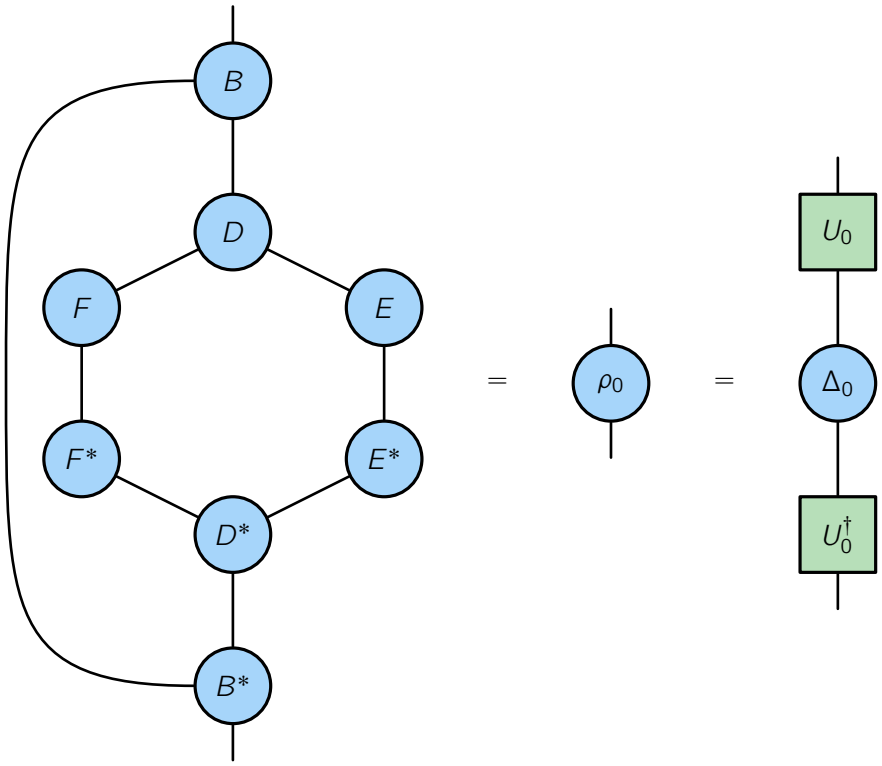
Input: Tensor network with tensors $\{T_1, \ldots, C, \ldots, T_n\}$, proposed center of orthogonality C

1 **for** *branches* b_i *from* C **do**
2 $\quad \rho_{b_i} \leftarrow$ branch density of b_i ;
3 $\quad$ Find spectral decomposition $\rho_{b_i} = U_{b_i} \Delta_{b_i} U_{b_i}^\dagger$;
4 $\quad X_{b_i} \leftarrow U_{b_i} \Delta_{b_i}^{\frac{1}{2}} U_{b_i}^\dagger$;
5 $\quad X_{b_i}^{-1} \leftarrow U_{b_i} \Delta_{b_i}^{-\frac{1}{2}} U_{b_i}^\dagger$;
6 **for** *branches* b_i *from* C **do**
7 $\quad C \leftarrow$absorb X_{b_i};
8 $\quad b_i \leftarrow$absorb $X_{b_i}^{-1}$ into first tensor of b_i;

The following example shows the direct orthogonalization procedure in detail. Consider the following tensor network.



First we compute the branch density for the branch $\rightarrow B \rightarrow D \rightarrow (E, F)$ as



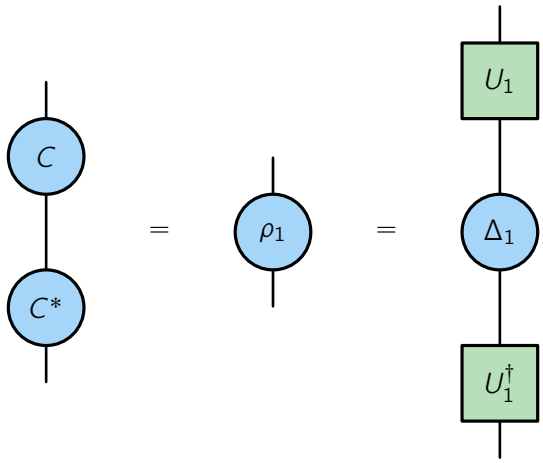
As discussed above, ρ_0 is Hermitian and we can compute its spectral decomposition as

$$\rho_0 = U_0 \Delta_0 U_0^\dagger = U_0 \Delta_0^{\frac{1}{2}} \Delta_0^{\frac{1}{2}} U_0^\dagger$$

which leads to

$$X_0 = U_0 \Delta_0^{\frac{1}{2}} U_0^\dagger \quad \text{and} \quad X_0^{-1} = U_0 \Delta_0^{-\frac{1}{2}} U_0^\dagger.$$

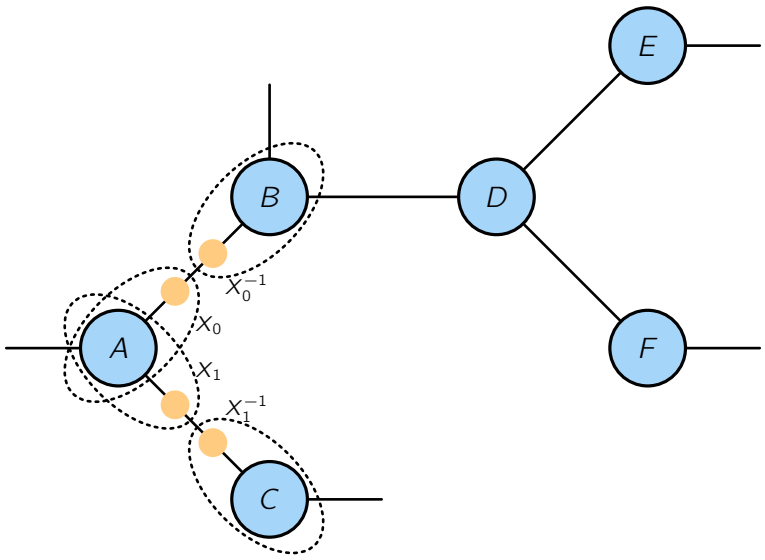
Next we compute the branch density for the branch $\rightarrow C$, as



and likewise get X_1 and X_1^{-1} as

$$X_1 = U_1 \Delta_1^{\frac{1}{2}} U_1^\dagger \quad \text{and} \quad X_1^{-1} = U_1 \Delta_1^{-\frac{1}{2}} U_1^\dagger.$$

Next we absorb X_0, X_1 into tensor A, X_0^{-1} into tensor B and X_1^{-1} into tensor C:



Now the network is orthogonalized with tensor A being the center of orthogonality and both, branch $\rightarrow B \rightarrow D \rightarrow (E, F)$ and $\rightarrow E$ being isometries under the condition that the edges connecting the branches with A form the columns and all other free edges form the rows.

3.3. Orthogonalization with respect to Edges

We now want to extend the orthogonalization procedure described in section 3.2 to shift the center of orthogonality not towards a tensor in the network but towards an edge connecting two tensors. Obviously, we cannot absorb square roots into edges between tensors, therefore we introduce a new tensor that we call *link-tensor* on the edge we want to be the center of orthogonality. To keep the network invariant under such changes, we initially set the link-tensor to be the identity matrix. Algorithm 4 describes the procedure in detail.

Algorithm 4: Orthogonalization with respect to edges

Input: Tensor network with tensors $\{T_1, \dots T_n\}$, edge (T_i, T_j)

Output: Tensor network with tensors $\{T_1, \dots T_i', \dots, T_j', \dots, T_k, \sigma_{T_i,Tj}\}$

1 Initialize link-tensor $\sigma_{T_i,Tj} \leftarrow I$;

2 Insert $\sigma_{T_i,Tj}$ on edge (T_i, TJ);

// Handle branch over T_i

3 $\rho_{T_i} \leftarrow$ branch density for branch from $\sigma_{T_i,Tj}$ over T_i;

4 Find spectral decomposition $\rho_{T_i} = U_{T_i} \Delta_{T_i} U_{T_i}^\dagger$;

5 $X_{T_i} \leftarrow U_{T_i} \Delta_{T_i}^{\frac{1}{2}} U_{T_i}^\dagger$;

6 $X_{T_i}^{-1} \leftarrow U_{T_i} \Delta_{T_i}^{-\frac{1}{2}} U_{T_i}^\dagger$;

// Handle branch over T_j

7 $\rho_{T_j} \leftarrow$ branch density for branch from $\sigma_{T_i,Tj}$ over T_j;

8 Find spectral decomposition $\rho_{T_j} = U_{T_j} \Delta_{T_j} U_{T_j}^\dagger$;

9 $X_{T_j} \leftarrow U_{T_j} \Delta_{T_j}^{\frac{1}{2}} U_{T_j}^\dagger$;

10 $X_{T_j}^{-1} \leftarrow U_{T_j} \Delta_{T_j}^{-\frac{1}{2}} U_{T_j}^\dagger$;

// Absorb principal square roots and their inverses

11 $T_i' \leftarrow$ absorb $X_{T_i}^{-1}$ into T_i;

12 $T_j' \leftarrow$ absorb $X_{T_j}^{-1}$ into T_j;

13 $\sigma_{T_i,Tj} \leftarrow$ absorb X_{T_i}, X_{T_j} into $\sigma_{T_i,Tj}$;

14 **return** $\{T_1, \dots T_i', \dots, T_j', \dots, T_k, \sigma_{T_i,Tj}\}$;

3.4. Canonical Forms

We now have all the pieces together to introduce general canonical forms for loop-free tensor networks. The orthogonalization procedure described in this section has been discussed in [Oru13], [Vid07] or [Vid03]. The idea is to use the direct orthogonalization procedure from section 3.2 to simultaneously orthogonalize a network with respect to all of its edges, similar

to what we presented in section 3.3. The resulting network is said to be in *canonical form* and can be used to flexibly shift centers of orthogonality as we will show in section 3.5.

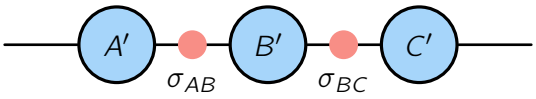
Algorithm 5: Canonicalization of a loop-free tensor network

Input: Tensor network with tensors $\{T_1, \ldots, T_n\}$
Output: Tensor network in canonical form
// Collect square roots of branch densities
1 **for** *edges* (T_i, T_j) *in the network* **do**
2 Initialize link-tensor $\sigma_{T_i,T_j} \leftarrow I$;
3 Insert σ_{T_i,T_j} on edge (T_i, T_j);
// Handle branch over T_i
4 $\rho_{T_i} \leftarrow$ branch density for branch from σ_{T_i,T_j} over T_i;
5 Find spectral decomposition $\rho_{T_i} = U_{T_i} \Delta_{T_i} U_{T_i}^\dagger$;
6 $X_{T_i} \leftarrow U_{T_i} \Delta_{T_i}^{\frac{1}{2}} U_{T_i}^\dagger$;
7 $X_{T_i}^{-1} \leftarrow U_{T_i} \Delta_{T_i}^{-\frac{1}{2}} U_{T_i}^\dagger$;
// Handle branch over T_j
8 $\rho_{T_j} \leftarrow$ branch density for branch from σ_{T_i,T_j} over T_j;
9 Find spectral decomposition $\rho_{T_j} = U_{T_j} \Delta_{T_j} U_{T_j}^\dagger$;
10 $X_{T_j} \leftarrow U_{T_j} \Delta_{T_j}^{\frac{1}{2}} U_{T_j}^\dagger$;
11 $X_{T_j}^{-1} \leftarrow U_{T_j} \Delta_{T_j}^{-\frac{1}{2}} U_{T_j}^\dagger$;
// Absorb into tensors
12 **for** *tensors* T_i **do**
13 $T_i' \leftarrow$absorb all $X_{T_i}^{-1}$ into T_i;
// Absorb into link-tensors
14 **for** *link-tensors* σ_{T_i,T_j} **do**
15 $\sigma'_{T_i,T_j} \leftarrow$absorb X_{T_i}, $X_{T_j}^{-1}$ into σ_{T_i,T_j};

Algorithm 5 almost looks like a repeated application of algorithm 4 for all edges in the network, but there is a subtle yet important difference: While a repeated application of algorithm 5 would absorb the principal square roots for each branch density right after computing it, algorithm 4 postpones the absorption until after all branch densities have been found. This guarantees that each principal square root originates from a branch density of the original network.

One could assume that repeated absorption into the network tensors prevents individual branches from being isometries, however, exactly this is the case. To see this, we consider

the following small example of a network in canonical form:



We show that the path $\rightarrow B' \rightarrow \sigma_{AB} \rightarrow A'$ forms an isometry: Consider the product $A'\sigma_{AB}B'$. We denote X_1 as the principal square root of the branch density for the path $\rightarrow A$, X_2 for the path $\rightarrow B \rightarrow C$ and X_3 for the path $\rightarrow B \rightarrow A$. Reverse engineering the orthogonalization procedure we arrive at

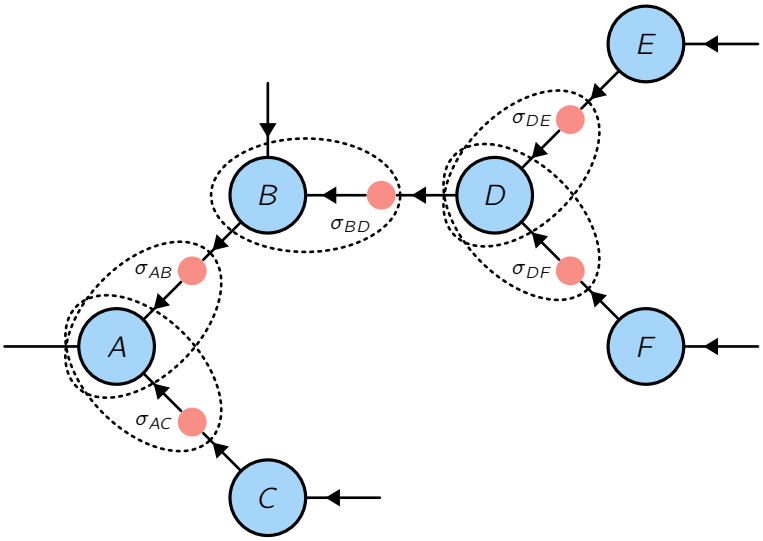
$$\begin{aligned}
(A'\sigma_{AB}B')^\dagger(A'\sigma_{AB}B') &= (AX_1^{-1}X_1X_2X_2^{-1}BX_3^{-1})^\dagger(AX_1^{-1}X_1X_2X_2^{-1}BX_3^{-1}) \\
&= (ABX_3^{-1})^\dagger(ABX_3^{-1}) \\
&= (X_3^{-1})^\dagger(AB)^\dagger(AB)X_3^{-1} \\
&= \rho_3^{-\frac{1}{2}}(AB)^\dagger(AB)\rho_3^{-\frac{1}{2}} \\
&= \rho_3^{-\frac{1}{2}}\rho_3\rho_3^{-\frac{1}{2}} \\
&= \rho_3^{-\frac{1}{2}}\rho_3^{\frac{1}{2}}\rho_3^{\frac{1}{2}}\rho_3^{-\frac{1}{2}} \\
&= I.
\end{aligned}$$

Here we used that $(X_3^{-1})^\dagger = X_3^{-1}$ because

$$\left(X_3^{-1}\right)^\dagger = \left(U_3\Delta^{-\frac{1}{2}}U_3^\dagger\right)^\dagger = U_3\Delta^{-\frac{1}{2}}U_3^\dagger = X_3^{-1}.$$

3.5. Shifting Centers of Orthogonality

Having a tensor network in canonical form as described in section 3.4 allows for flexibly shifting the center of orthogonality in the network as we will show next. Consider the following network in canonical form



To shift the center of orthogonality to e.g. the tensor A, we absorb each link-tensor into on of the network tensors, according to its direction towards A as shown in the image where dotted lines show how link-tensors should be absorbed. As discussed previously, this gives the network orientation in the sense that each tensors forms an isometric matrix, when legs are grouped together accordingly.

The above absorption of link-tensors can be efficiently implemented using a standard graph traversal of the network, beginning at the proposed center of orthogonality (see algorithm 6).

Now consider any tensor $T \in \mathbb{C}^{n_1 \times \ldots \times n_k}$ in the network that is not the center of orthogonality. We know that T is an isometry T_M if legs are grouped together correctly. To obtain this isometric matrix, we have to know the *isometry leg* of T, that is the leg of T that, when followed, leads to the center of orthogonality. Since we consider loop-free tensor networks only, we know that there will be at most one such leg. The isometric matrix can be obtained by using the matricization procedure described earlier with the isometry leg as column legs and all other legs as row legs.

Algorithm 6: Shift center of orthogonality

Input: Tensor network with tensors $\{T_1, \ldots, T_n, C\}$, proposed center of orthogonality C'

Result: Tensor network $\{T'_1, \ldots, T'_n, C'\}$ with C as the center of orthogonality

```
1  Initialize empty queue Q;
2  Q.add(C);
3  while Q is not empty do
4      T ← Q.pop();
5      if T has not been visited then
6          Mark T as visited;
       // Perform the absorption of link-tensors
7      if T is a link-tensor then
8          p(T)' ←absorb T into p(T);
       // Add neighboring tensors and set parents
9      for K in neigbors of T do
10         if neighbor has not been visited then
11             Q.add(K);
12             p(K) ← T;
```

CHAPTER 4

Application

4.1. Optimal Truncation

Given a tensor network, we are often interested in an approximation of the same network with smaller dimensions of the edges. We obviously want this approximation to be as good as possible and have seen in theorem 2.4 that the SVD of a matrix is the best possible approximation of a given rank that is available. We could therefore just perform a low-rank approximation of all the network tensors to get a good approximation. However, having a tensor network in canonical form yields an easier approach shown in algorithm 7. This optimal truncation routine has been described in [Eve18].

Algorithm 7: Optimal truncation

Input: Tensor network with tensors $T_0, \ldots, T_n$, edge (T_i, T_j)

Output: Tensor network with tensors $T_0, \ldots, T_i', \ldots, T_j', \ldots, T_n$

1 Orthogonalize the edge (T_i, T_j) by applying algorithm 4;

2 $\sigma_{T_iT_j} \leftarrow$ link-tensor of the above orthogonalization ;

3 $U\Sigma V^\dagger \leftarrow \text{SVD}\,(\sigma_{T_iT_j})$;

4 $U'\Sigma'V'^\dagger \leftarrow$ low-rank approximation as shown in definition 2.1;

5 $T_i' \leftarrow$ absorb U' into T_i;

6 $T_j' \leftarrow$ absorb $V'^\dagger$ into T_j;

7 $\sigma_{T_iT_j} \leftarrow \Sigma'$;

The term *optimal truncation* refers to the Eckard-Young theorem and is meant with respect to the Frobenius norm. Theorem 4.1 states that it is not necessary to approximate the tensors themselves but that this approximation can be performed on the link-tensor instead. We claim that this is the best approximation possible for the edge between two tensors in a network:

Theorem 4.1 (Optimal truncation)**.** *Given a loop-free tensor network and T_i, T_j two connected tensors in the network. Then the optimal truncation of the edge between T_i and T_j is given by first shifting the center of orthogonality to the edge, performing a low-rank approximation of the resulting link-tensor on the edge and absorbing the unitary factors of the low-rank approximation into T_i and T_j, respectively.*

Proof. See [Eve18]. □

The intuition behind theorem 4.1 is that the edge-orthogonalization procedure relies on the spectral decomposition of the branch density $\rho_P = P^\dagger P$ with P being the branches leading to the edge to be truncated. This spectral decomposition is closely tied to the SVD of the paths themselves: Let $P = U\Sigma V^\dagger$ be the SVD of P and $P^\dagger P = W\Delta W^\dagger$ the spectral decomposition of the branch density, then

$$A^T A = \left(U\Sigma V^\dagger\right)^\dagger \left(U\Sigma V^\dagger\right) = V\Sigma^2 V^\dagger = W\Delta W^\dagger \quad \Rightarrow \quad U = V \text{ and } \Delta = \Sigma^2.$$

A low-rank approximation of P can thus be mapped to a low-rank approximation of the spectral decomposition of $\rho_P = P^\dagger P$.

4.2. Optimization on Riemannian Manifolds

When we interpret the tensors of a tensor network as isometries under a certain grouping of legs, we can consider the network as elements of a geometric structure called a *Riemannian manifold* $\mathcal{M}$. Given a function $f\colon \mathcal{M} \to \mathbb{R}$, we can try to minimize it, i.e., we try to solve the optimization problem

$$\min_{x \in \mathcal{M}} f(x).$$

While there are different techniques to solve the above optimization problem, we will focus on a modified line-search method, well-known from optimization on $\mathbb{R}^n$.

Methods to perform optimization on Riemannian manifolds have been extensively studied in [AMS09], [Boo03], [Bou22] or [Smi14]. Introductions to general and Riemannian manifolds can be found in [BD13] and [Lee13]. Applying these optimization techniques to tensor networks is a relatively young idea and has previously been done in e.g. [Hae+12], [HDH20], [LKF20] and [VHV18].

We give a brief introduction into the language and tools of Riemannian manifolds and will then show how the optimization procedure can be applied to tensor networks in canonical form.

4.2.1. Manifolds

A manifold can be thought of as a structure that *locally looks like* $\mathbb{R}^d$. The canonical example for a manifold is the sphere S^2 which can be locally mapped onto the two-dimension space $\mathbb{R}^2$.

We omit the most general definition of manifolds and restrict ourselves to the definition of a (smooth, embedded sub-) manifold as follows:

Defintion 4.1 (Smooth, embedded sub-manifold)**.** A subset $\mathcal{M} \subseteq \mathcal{E}$ of a $d-$dimensional vector space $\mathcal{E}$ is a smooth, embedded sub-manifold (or just manifold), if for every $x \in \mathcal{M}$ there exists a neighborhood $U \subseteq \mathcal{E}$, an open set $V \in \mathbb{R}^d$ and a smooth bijection with smooth inverse (a diffeomorphism) $\varphi \colon U \to V$ such that $\varphi(\mathcal{M} \cap U) = E \cap V$, with $E \subseteq \mathbb{R}^d$ being a linear subspace of $\mathbb{R}^d$.

Although definition 4.1 is already a restriction of the most general definition of a manifold, it reads very technically but basically just says that the manifold looks *locally*, in a neighborhood of every point, like $\mathbb{R}^d$. Our definition of a manifold is *smooth* because the mapping of the local look-alike neighborhoods are smooth and *embedded* because the manifold lives in a larger vector space $\mathcal{E}$.

Our objects of interest are tensors that we can consider as matrices after appropriate matricization. Furthermore, using tools from chapter 3, we can enforce orthogonality constraints on the tensors. The following lemma covers this fact:

Lemma 4.1 (Stiefel manifold)**.** *The set of isometric matrices in $\mathbb{C}^{n \times p}$ with $p \leqslant n$ is a manifold, called the* Stiefel manifold $\mathrm{St}(n, p)$*:*

$$\mathrm{St}(n, p) = \{M \in \mathbb{C}^{n \times p} \mid X^\dagger X = I_p\},$$

where I_p is the identity matrix of size p.

Proof. See [AMS09]. □

For each point $x \in \mathcal{M}$ we can consider vectors being tangent to $\mathcal{M}$ at x. The set of all tangent vectors of x has the structure of a vector space and is called the *tangent space* of x, denoted as $T_x\mathcal{M}$. More formally

Defintion 4.2 (Tangent space)**.** Let $\mathcal{M}$ be a sub-manifold of a vector space $\mathcal{E}$ and $\gamma\colon \mathbb{R} \to \mathcal{M}$ a curve with $\gamma(0) = x$. The derivative $\gamma'(0) \in \mathcal{E}$ is a *tangent vector* at x. The set of all tangent vectors at x is the *tangent space* $T_x\mathcal{M} \subseteq \mathcal{E}$ at x.

The disjoint union of the tangent spaces of all points $x \in \mathcal{M}$ is the *tangent bundle* $T\mathcal{M}$ of $\mathcal{M}$:

$$T\mathcal{M} = \bigsqcup_{x \in \mathcal{M}} T_x\mathcal{M}.$$

Let's draw an example for a tangent space of $\mathrm{St}(n, p)$: Given a curve

$$\gamma(t) = M + tN + \mathcal{O}(t^2),$$

with $M \in \mathrm{St}(n, p)$. Then, $\gamma(t) \in \mathrm{St}(n, p)$ if $\gamma(t)^\dagger\gamma(t) = I$, i.e.,

$$I = \gamma(t)^\dagger\gamma(t) = (M + tN)^\dagger(M + tN) + \mathcal{O}(t^2) = I + t(M^\dagger N + N^\dagger M) + \mathcal{O}(t^2),$$

so $M^\dagger N + N^\dagger M = 0$. Thus the tangent space $T_M \mathrm{St}(n, p)$ can be characterized as

$$T_M \mathrm{St}(n, p) = \{N \in \mathbb{C}^{n \times p} \mid M^\dagger N = -N^\dagger M\}.$$

Note that the tangent space at a point $x \in \mathcal{M}$ is in general no subspace of $\mathcal{M}$. However, our optimizations introduced later will require to land on the manifold again, so we need a way to map points from tangent spaces to the manifold. Such mappings are called *retractions* and are defined as

Defintion 4.3 (Retraction, [AMS09, p. 55])**.** A retraction on a manifold $\mathcal{M}$ is a smooth mapping

$$R\colon T\mathcal{M} \to \mathcal{M}$$

with the following properties. We denote the restriction $R|_{T_X\mathcal{M}}$ as R_x.

1. $R_x(0_x) = x$, with 0_x being the zero-element of $T_x\mathcal{M}$ and
2. With the identification $T_{0_x}T_x\mathcal{M} \simeq T_x\mathcal{M}$, R_x satisfies

   $$DR_x(0_x) = \mathrm{id}_{T_x\mathcal{M}},$$

   where $\mathrm{id}_{T_x\mathcal{M}}$ denotes the identity on $T_x\mathcal{M}$.

The second condition in the above definition is called *local rigidity* [AMS09, p. 55] and makes sure a valid retraction in some sense preserves gradients (see fig. 4.1).

For our purposes we want to use a retraction on the Stiefel manifold $\mathrm{St}(n, p)$. Let $X \in \mathrm{St}(n, p)$ be an isometry, and $A \in \mathbb{C}^{n \times p}$ a matrix in the tangent space $T_X\mathrm{St}(n, p)$. We claim that the mapping of A to the first factor of the polar decomposition of $A + X$ is a retraction R_X.

An alternative choice for a valid retraction is a mapping of A to unitary factor Q of the QR decomposition of $A + X$.

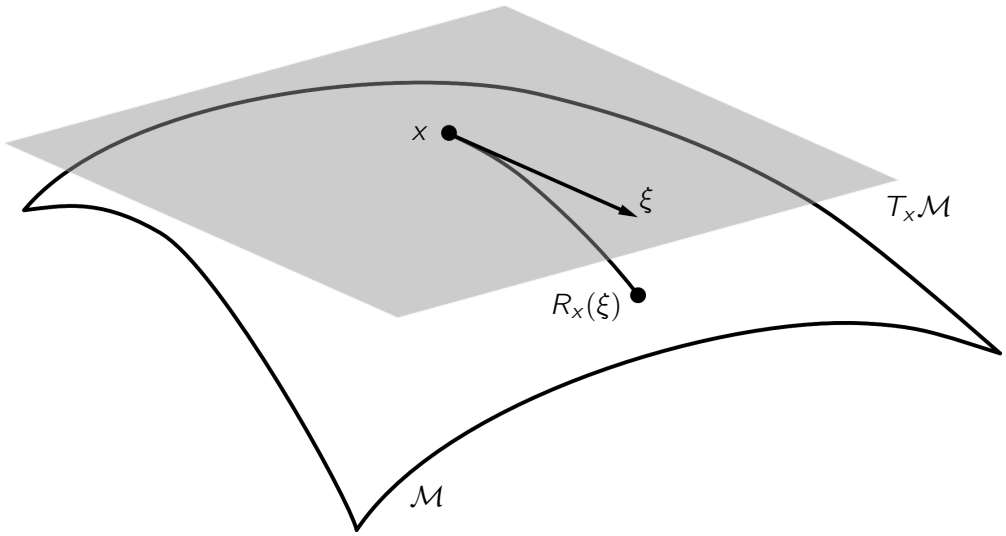


Figure 4.1.: Retraction R_x of point $\xi \in T_x\mathcal{M}$ onto manifold $\mathcal{M}$.

4.2.2. Riemannian Manifolds

The modified line-search we will be using to optimize functions on tensor networks is a modified gradient-descent. For the definition of gradients we need to equip the tangent spaces of manifolds with an inner product to arrive at a *Riemannian manifold*, but not every inner product will do. We additionally require some smoothness properties on the inner product, that we are not going to discuss here in detail. It's enough to know that the inner product we presented in chapter 2 fulfills this smoothness property.

Without giving a formal definition, we define a *Riemannian manifold* to be a manifold $\mathcal{M}$ with an inner product on the tangent space $T_x\mathcal{M}$ at every point $x \in \mathcal{M}$ that fulfills the mentioned smoothness properties.

We are now ready to give a rule to compute the gradient on Riemannian manifolds:

Theorem 4.2 (Gradient on Riemannian manifolds)**.** *Let $\mathcal{M}$ be a sub-manifold of a vector space $\mathcal{E}$, $\mathrm{Proj}_x \colon \mathcal{E} \to T\mathcal{M}$ an orthogonal projector, $f : \mathcal{M} \to \mathbb{R}$ a smooth function and $\operatorname{grad} \bar{f}$ the gradient of f in $\mathcal{E}$. Then*

$$\operatorname{grad} f(x) = \mathrm{Proj}_x \left(\operatorname{grad} \bar{f}(x) \right).$$

Proof. See [Bou22, p. 58]. □

Theorem 4.2 states that computing gradients on the manifold $\mathcal{M}$ is as easy as computing the regular gradient and projecting it onto the tangent bundle of $\mathcal{M}$.

The Stiefel manifold $\mathrm{St}(n,p)$ is a sub-manifold of the linear space $\mathbb{C}^{n\times p}$. As an orthogonal projector we can thus choose

$$\mathrm{Proj}_X \colon \mathbb{C}^{n\times p} \to T_X \,\mathrm{St}(n,p), \quad \mathrm{Proj}_X(Z) = Z - X\mathrm{sym}\left(X^\dagger Z\right),$$

where

$$\mathrm{sym}(M) = \frac{1}{2}\left(M + M^\dagger\right)$$

is the symmetric part of M [AMS09, p. 81].

4.2.3. Riemannian gradient descent

Line-search methods are based on the formula [AMS09, p. 54]

$$x_{k+1} = x_k + t_k \eta_k,$$

where η_k is the direction of the search and t_k is the step-size. One of the most prominent procedures to perform a line-search is gradient descent. For Riemannian manifolds it can be formulated as shown in algorithm 8.

Algorithm 8: Riemannian gradient descent

Input: Riemannian manifold $\mathcal{M}$, smooth function $f \colon \mathcal{M} \to \mathbb{R}$, initial point $x_1 \in \mathcal{M}$

1 **for** $k = 1, 2, \ldots$ **do**
2 $\quad t_k \leftarrow$ step-size;
3 $\quad \eta_k \leftarrow -t_k \,\mathrm{grad}\, f(x_k)$;
4 $\quad x_{k+1} \leftarrow R_{x_k}(\eta_k)$;

Riemannian gradient descent works by finding the direction for the line-search as the gradient of the function to be optimized on the manifold. As shown in theorem 4.2 this can be done by computing the gradient in the larger vector space and projecting it onto the manifold's tangent bundle. Since we require the updated point to live on the manifold again, a retraction R_x from the tangent bundle to the manifold is applied.

We now want to use algorithm 8 to optimize functions on tensor networks.

4.2.4. Riemannian Optimization on Tensor Networks

As already explained, the Stiefel manifold is a natural choice to embed isometric tensors. However, a whole network consists not of a single tensor, but of a list of tensors. We therefore need a Riemannian manifold that contains multiple tensors as elements. Furthermore, if a network is in canonical form, all but the center of orthogonality can be considered as isometries and we need another manifold containing the center. For the latter we are going

to use the fact that the set of unconstrained matrices in $\mathbb{C}^{m\times n}$ is a usual Euclidean space and where we can just compute gradients as usual.

To handle multiple tensors we consider the product of manifolds:

Lemma 4.2 (Product of manifolds)**.** *Let $\mathcal{M}_1, \dots, \mathcal{M}_k$ be Riemannian manifolds, then the product*

$$\bigtimes_i \mathcal{M}_i$$

is a Riemannian manifold on which the inner product, retractions and projections are defined point-wise.

For a product of Stiefel manifolds lemma 4.2 states that for $(X_1, \dots, X_n) \in \bigtimes_i \operatorname{St}(n_i, p_i)$ an orthogonal projector

$$\operatorname{Proj}_{(X_1,\dots,X_n)} \colon \bigtimes_i \mathbb{C}^{n_i\times p_i} \to T_{(X_1,\dots,X_n)} \bigtimes_i \operatorname{St}(n_i, p_i)$$

is given by

$$\operatorname{Proj}_{(X_1,\dots,X_n)}(Z_1, \dots, Z_n) = \left(Z_1 - X_1 \operatorname{sym}\left(X_1^\dagger Z_1^\dagger\right), \dots, Z_n - X_n \operatorname{sym}\left(X_n^\dagger Z_n^\dagger\right)\right)$$

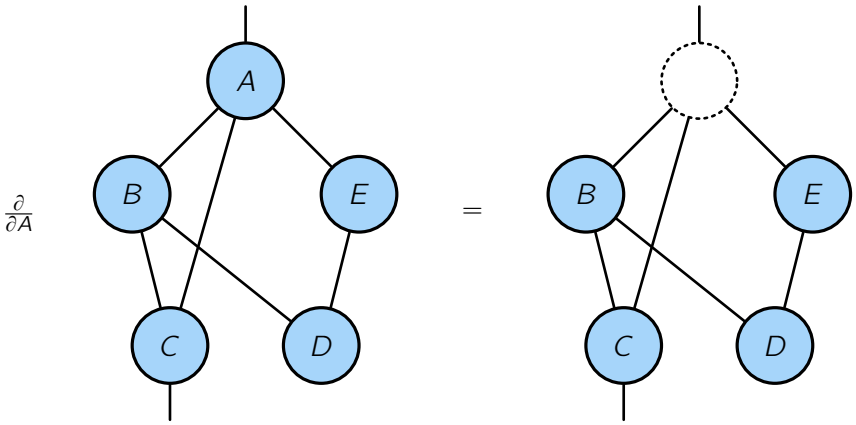
and a retraction

$$R_{(X_1,\dots,X_n)} \colon T_{(X_1,\dots,X_n)} \bigtimes_i \operatorname{St}(n_i, p_i) \to \bigtimes \operatorname{St}(n_i, p_i)$$

by mapping each component of the tangent vector to the first factor if its polar decomposition.

One key ingredient is missing to optimize functions on product manifolds of tensors: The function to be optimized will take a list of tensors as input but the gradient descent procedure operates point-wise. We therefore have to clarify what a partial derivative of a function on a tensor network is.

Since the contraction of tensors is linear in its arguments, a tensor network is a linear function of its tensors. The partial derivative of a tensor network with respect to a tensor T is thus defined by *cutting out* T:



Due to lemma 4.2 we can define Riemannian gradient decent for tensors networks as shown in algorithm 9. In other words, algorithm 9 works by individually performing Riemannian gradient descent on all tensors of a network while taking the partial derivatives as defined above as the gradient in the embedding vector space.

Algorithm 9: Riemannian gradient descent on orthogonalized tensor networks

Input: Tensors network with initial tensors $\{C^1, T_1^1, \ldots, T_n^1\}$, center of orthogonality C, real-valued smooth function f on tensors

1 **for** $k = 1, 2, \ldots$ **do**
2 $\quad t^k \leftarrow$ step-size;
$\quad$ *// Gradient descent for center of orthogonality*
3 $\quad \eta_C^k \leftarrow -t^k \partial_C^k f(C^k, T_1^k, \ldots, T_n^k)$;
$\quad$ *// Gradient descent for isometric tensors*
4 $\quad$ **for** $i = 1, \ldots, n$ **do**
$\quad\quad$ *// Find direction by projecting gradient onto tangent space*
5 $\quad\quad \eta_{T_i}^k \leftarrow -t^k \mathrm{Proj}_{T_i^k}\left(\partial_{T_i^k} f(C^k, T_1^k, \ldots, T_n^k)\right)$;
$\quad\quad$ *// Retract direction onto manifold and update tensor*
6 $\quad\quad T_i^{k+1} \leftarrow R_{T_i^k}(\eta_{T_i}^k)$;

Table 4.1 recaps the properties of the Stiefel manifold $\mathrm{St}(n, p)$ as needed for a line-search with gradient descent.

tangent space	$Z \in \mathbb{C}^{n\times p} :\ \operatorname{sym}(X^\dagger Z) = 0$
projection onto tangent space	$\operatorname{Proj}_x Z = Z - X \operatorname{sym}(X^T Z)$
gradient	$\operatorname{grad} f(X) = \operatorname{Proj}_X \operatorname{grad} \bar{f}(X)$

Table 4.1.: Properties of Stiefel manifold $\operatorname{St}(n, p)$.

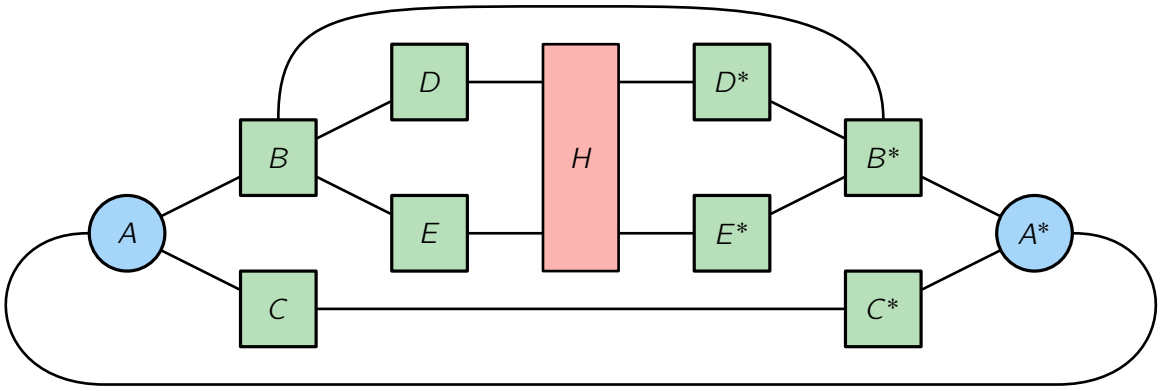
4.2.5. Finding ground states

A recurring objective in physics is to minimize functions of the form

$$f(T) = \frac{\langle T|H|T\rangle}{\langle T|T\rangle}$$

where T can be a tensor network and H is a Hermitian operator. Functions of the above form, can be interpreted as expectation values of the energy in a system and states minimizing it are called *ground states*.

The following figure shows an example for $\langle T|H|T\rangle$ where T is a network of five tensors, with the center of orthogonality shifted to A.



Since every tensor but A is an isometry, evaluating f can be simplified to

$$\begin{aligned} f(T) = \frac{\langle T|H|T\rangle}{\langle T|T\rangle} &= \frac{\langle T|H|T\rangle}{\langle A|A\rangle} \\ &= \frac{\langle A, B, C, D, E, F|H|A, B, C, D, E, F\rangle}{\langle A|A\rangle} \\ &= \frac{\langle A, B, D, E, F|H|A, B, D, E, F\rangle}{\langle A|A\rangle}. \end{aligned}$$

Since f is complex-valued we treat a tensor and its conjugated-transposed as two different variables. As shown in [VHV18, p. 25] each component of the gradient of f can be computed as

$$\operatorname{grad} f = 2\frac{\partial_{M^\dagger}\langle T|H|T\rangle - \frac{\langle T|H|T\rangle}{\langle T|T\rangle}\partial_{M^\dagger}\langle T|T\rangle}{\langle T|T\rangle}.$$

If T would consist of isometries only, the gradient would reduce to

$$\operatorname{grad} f = 2\partial_{M^\dagger} \langle T|H|T\rangle ,$$

but we consider general networks.

As already explained, each of the individual partial derivatives amounts of cutting out the respective tensor from the network. Due to the simplified form of $f(T)$ above, this leads to simple terms in many cases.

Given the gradient of f, the Riemannian gradient descent machinery developed previously can be used to find ground states. The Riemannian manifold will be the product manifold of an Euclidean space of unconstrained matrices for the center of orthogonality and Stiefel manifolds for each isometric tensor. In the Stiefel manifold cases $\operatorname{grad} f$ has to be projected from the embedding vector space onto the tangent bundle and retracted from the tanget bundle back onto the manifold, in the case of the unconstrained center nothing else has to be done.

To simplify both, f and $\operatorname{grad} f$ it is not necessary to have a center of orthogonality. In fact, a canonical form as shown in section 3.4 is enough and shifting the center as shown in section 3.5 is not necessary. It is sufficient to choose a center of orthogonality and drop all paths in the network connected to this center since it is known that these paths would be isometries when the actual shift would be performed. This holds for both, simplifying terms of the form $\langle T|H|T\rangle$ as well as $\langle T|T\rangle$. In the latter case it is always enough to just form the inner product $\langle A\sigma_1 \dots \sigma_n | A\sigma_1 \dots \sigma_n\rangle$ with A being the center and σ_i being link-matrices, adjacent to A. All other paths in the network, including all other link-matrices, can be dropped.

4.2.6. Implementation and Numerical Results

The Riemannian gradient descent methods introduced above have been implemented using the programming language Python. Besides explicitly computing gradients by cutting out tensors from networks we also implemented a version where gradients are computed using automatic differentiation provided by the Python library Jax [Bra+18]. Both versions produce the same gradients which we consider a sanity check for our implementation.

To be able to use more sophisticated line-search methods we experimented with the Python library Pymanopt [Bou+14; TKW16], which provides methods for optimizations on Riemannian manifolds.

Figure 4.2 shows the convergence of the norm of the gradient for the standard Riemannian gradient descent as presented in this thesis, a modified version of Riemannian gradient descent that dynamically adapts the step-size [AMS09, p. 76; Bou22, p. 76], called *backtracking line-search*, and a conjugated gradient method, also accelerated using backtracking line-search [AMS09, p. 180; Bou22, p. 140]. While more advanced methods can produce faster

convergence, the simple approach we developed in the course of this thesis successfully converges and is able to solve the posed problem.

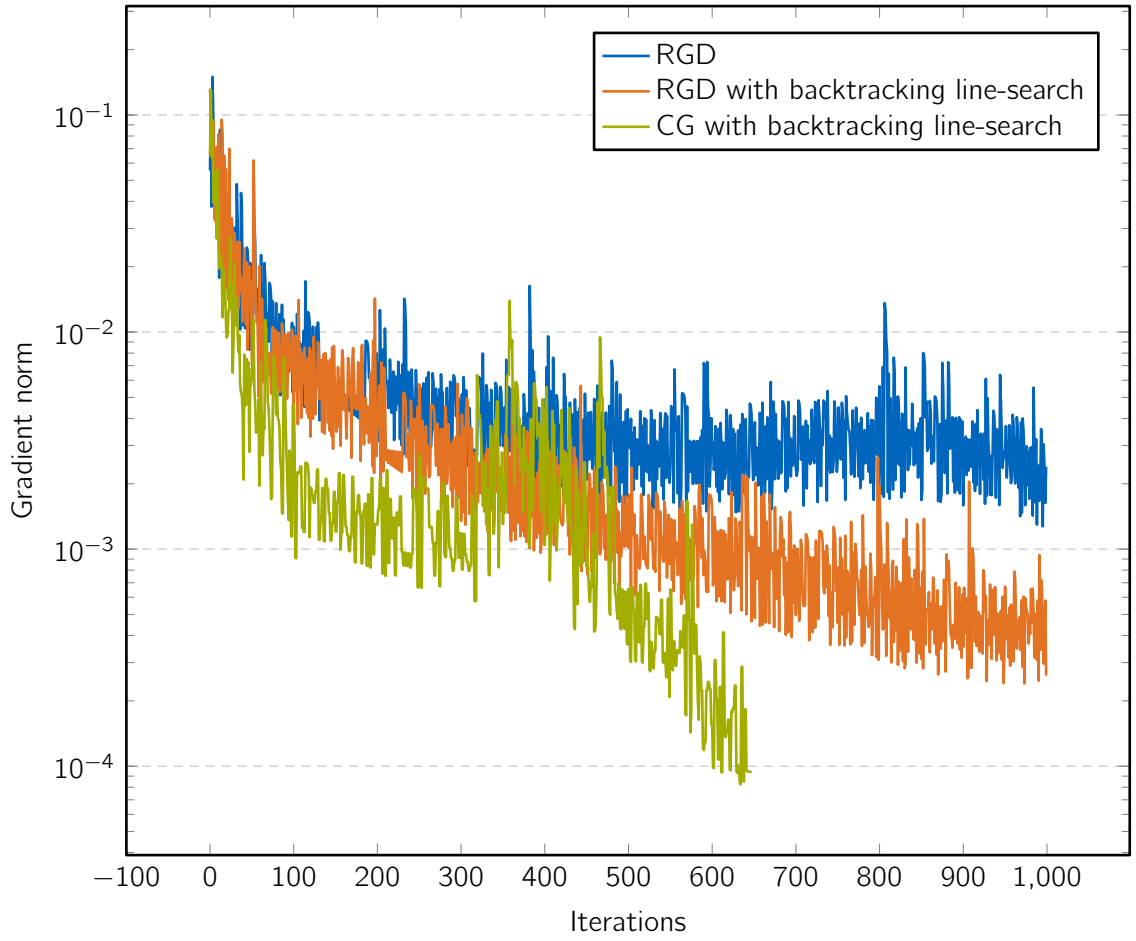


Figure 4.2.: Convergence of gradient norms for Riemannian gradient descent (RGD), Riemannian gradient descent with backtracking line-search and conjugated gradient (CG) with backtracking line-search.

CHAPTER 5

Conclusion and Future Work

In the course of this thesis we presented several methods to enforce orthogonality constraints on tensor networks and to transform networks into canonical forms that can simplify further usage in different algorithms.

We explored how tensor networks can be considered as elements on Riemannian manifolds and how line-search methods can be used to optimize functions taking network tensors as inputs. Gradient-based optimization procedures seem to be a promising approach to solve optimization problems involving tensor networks and are an interesting addition to existing algorithms such as the density-matrix renormalization group, DMRG [McC08; Sch11] or the Evenbly-Vidal algorithm [EV07], especially since the Riemannian optimization machinery is independent of the underlying tensor network.

The toolbox of Riemannian optimization techniques doesn't end with first-order line-search methods and it would be interesting to see how second-order methods such as Newton's methods [AMS09, p. 111; Bou22, p. 137] or Riemannian trust regions [AMS09, p. 136; Bou22, p. 148] and their usually superlinear convergence [AMS09, p. 91] could be utilized.

During the implementation of the concepts presented here, we noticed a lack of fully functional, high performing and flexible frameworks for tensor networks. While more or less specialized frameworks exist [FWS20; HP18; Rob+19; Lya+22], a general and widely accepted solution is still missing.

APPENDIX A

Code Examples

Construction of Tensor Networks

```
class Network:
  def __init__(self, tensors=None, shapes=None, edges=None):
    # Construct graph to hold tensors and edges
    self.graph = Graph()

    # Add tensors specified as Numpy ndarrays
    if tensors is not None:
      for tensor in tensors:
        self.graph.add_node(Node(tensor[0], tensor[1]))

    # Add tensors specified by shape and initialized randomly
    if shapes is not None:
      for shape in shapes:
        self.graph.add_node(Node(shape[0], np.random.rand(*shape[1:])))

    # Add edges
    if edges is not None:
      for edge in edges:
        self.graph.add_edge(Edge(edge[0], edge[2], edge[1], edge[3]))
```

Listing A.1: Constructing a tensor network.

Inner products

```
def contract_conjugate_transposed(self,
                                  nodes,
                                  free_axes=None,
                                  output_match_topologyFalse):
  # Insert conjugated nodes
  conjugated_nodes, conjugated_edges = self._insert_conjugated_nodes(
      nodes, free_axes)

  # Perform actual contraction
  res = self.contract(nodes + conjugated_nodes,
                      output_match_topology=output_match_topology)

  # Remove conjugated nodes
  for node in conjugated_nodes:
    self.graph.remove_node(node)

  # Remove newly introduced edges
  for edge in conjugated_edges:
    self.graph.remove_edge(edge)

  return res
```

Listing A.2: Computing the inner product of tensors.

Tensor decomposition

```
def split_node(self,
               node: str,
               left_axes: list[int],
               right_axes: list[int],
               decomposition: str = 'svd'):
  if decomposition not in ['svd', 'qr']:
    raise RuntimeError(f"Unknown decomposition '{decomposition}'")

  T = self.graph.nodes[node].tensor

  # All decompositions we can use to split a node work on matrices rather than
  # general tensors. We therefore reshape the tensor behind the node to 2D,
  # according to left_nodes (rows) and right_nodes(cols)
  left_axes.sort()
  left_dims = [T.shape[i] for i in left_axes]
  right_axes.sort()
  right_dims = [T.shape[i] for i in right_axes]
  T = Network._matricization(T, left_axes, right_axes)

  # Split the node using svd
  if decomposition == 'svd':
```

```
    u, s, vh = np.linalg.svd(T)
    sigma = np.diag(s)
    sigma_sqrt = np.sqrt(sigma)
    k = len(s)
    u = u[:, :k]
    vh = vh[:k, :]
    left_tensor = (u @ sigma_sqrt).reshape((*left_dims, vh.shape[0]))
    right_tensor = (sigma_sqrt @ vh).reshape((u.shape[1], *right_dims))

  if decomposition == 'qr':
    q, r = np.linalg.qr(T)
    left_tensor = q.reshape((*left_dims, r.shape[0]))
    right_tensor = r.reshape((q.shape[1], *right_dims))

  # Get all edges of the original node
  old_edges = self.graph.get_connecting_edges(node)
  old_edges.sort(
      key=lambda edge: edge.axis1 if edge.node1 == node else edge.axis2)

  old_shape = self.graph.nodes[node].shape

  # Remove original node
  self.graph.remove_node(node)

  # Add new nodes
  left_node_name = f"{node}_splitted_left"
  self.graph.add_node(Node(left_node_name, left_tensor))

  right_node_name = f"{node}_splitted_right"
  self.graph.add_node(Node(right_node_name, right_tensor))

  free_axes = [-1 for s in old_shape]
  for node1, node2, axis1, axis2 in old_edges:
    if node1 == node:
      free_axes[axis1] = (node2, axis2)
    if node2 == node:
      free_axes[axis2] = (node1, axis1)

  left_axes_count = 0
  right_axes_count = 1
  for i, axis in enumerate(free_axes):
    if i in left_axes:
      if axis != -1:
        self.graph.add_edge(
            Edge(left_node_name, axis[0], left_axes_count, axis[1]))
      left_axes_count += 1
    if i in right_axes:
      if axis != -1:
        self.graph.add_edge(
            Edge(right_node_name, axis[0], right_axes_count, axis[1]))
      right_axes_count += 1
```

```
73
74    # Remove old edges
75    for edge in old_edges:
76      self.graph.remove_edge(edge)
77
78    # Insert edge between left and right nodes
79    self.graph.add_edge(
80        Edge(left_node_name, right_node_name,
81             len(left_tensor.shape) - 1, 0))
82
83    return (left_node_name, right_node_name)
```

Listing A.3: Decomposition of tensors using either SVD or QR.

Orthogonalization with respect to Edges

```
1  def orthogonalize_with_respect_to_edge(self, edge: Edge):
2    a = edge.node1
3    b = edge.node2
4    free_axis_a = edge.axis1
5    free_axis_b = edge.axis2
6
7    # insert link tensor
8    link_tensor = self.add_link_tensor(edge)
9    paths = self.graph.find_paths(link_tensor)
10   assert len(paths) <= 2
11
12   # Get principal sqrts
13   for path in paths:
14     if path[0] == a:
15       rho = self.contract_conjugate_transposed(path, [(a, free_axis_a)],
16                                                output_match_topology=True)
17       x_a, x_a_inv = self._principal_sqrt(rho)
18     if path[0] == b:
19       rho = self.contract_conjugate_transposed(path, [(b, free_axis_b)],
20                                                output_match_topology=True)
21       x_b, x_b_inv = self._principal_sqrt(rho)
22
23   # Absorb sqrts into link-tensor
24   self.graph.nodes[link_tensor].tensor = self._perform_contraction_on_tensors(
25       self.graph.nodes[link_tensor].tensor,
26       x_a,
27       x_b,
28       contraction_axes=[[1, 2], [1, -3], [2, -4]])
29
30   # Absorb inverse sqrt into a
31   a_node = self.graph.nodes[a]
32   a_axes = list(range(-1, -1 * len(a_node.shape) - 1, -1))
33   a_axes[free_axis_a] = 1
34   contraction_axes = [a_axes, [1, min(a_axes) - 1]]
```

```
  a_node.tensor = self._perform_contraction_on_tensors(
      a_node.tensor,
      x_a_inv,
      contraction_axes=contraction_axes,
      output_match_topology=True)

  # Absorb inverse sqrt into b
  b_node = self.graph.nodes[b]
  b_axes = list(range(-1, -1 * len(b_node.shape) - 1, -1))
  b_axes[free_axis_b] = 1
  contraction_axes = [b_axes, [min(b_axes) - 1, 1]]
  b_node.tensor = self._perform_contraction_on_tensors(
      b_node.tensor,
      x_b_inv,
      contraction_axes=contraction_axes,
      output_match_topology=True)
```

Listing A.4: Orthogonalization with respect to edges.

Canonical Forms

```
def canonicalization(self):
  sqrt_dict = {}
  for edge in self.graph.edges:
    # Collect principal sqrts that will be absorbed later
    sqrt_dict[edge] = []

    a = edge.node1
    free_axis_a = edge.axis1

    b = edge.node2
    free_axis_b = edge.axis2

    # Get principal sqrts for path from b starting with a
    a_path = [path for path in self.graph.find_paths(b) if path[0] == a][0]
    rho = self.contract_conjugate_transposed(a_path, [(a, free_axis_a)],
                                             output_match_topology=True)
    x_a, x_a_inv = self._principal_sqrt(rho)

    # Get principal sqrts for path from a starting with b
    b_path = [path for path in self.graph.find_paths(a) if path[0] == b][0]
    rho = self.contract_conjugate_transposed(b_path, [(b, free_axis_b)],
                                             output_match_topology=True)
    x_b, x_b_inv = self._principal_sqrt(rho)

    sqrt_dict[edge] = (x_a, x_a_inv, x_b, x_b_inv)

    for edge in list(self.graph.edges):
      # Insert link tensor
      link_tensor = self.add_link_tensor(edge)
```

```
30
31      # Absorb principal sqrts into link tensors
32      T = self.graph.nodes[link_tensor].tensor
33
34      x_a = sqrt_dict[edge][0]
35      x_b = sqrt_dict[edge][2]
36
37      self.graph.nodes[
38        link_tensor].tensor = self._perform_contraction_on_tensors(
39          T, x_a, x_b, contraction_axes=[[1, 2], [1, -3], [2, -4]])
40
41      # Absorb inverse sqrt into a
42      x_a_inv = sqrt_dict[edge][1]
43      a_node = self.graph.nodes[edge.node1]
44      a_axes = list(range(-1, -1 * len(a_node.shape) - 1, -1))
45      a_axes[edge.axis1] = 1
46      contraction_axes = [a_axes, [1, min(a_axes) - 1]]
47      a_node.tensor = self._perform_contraction_on_tensors(
48        a_node.tensor,
49        x_a_inv,
50        contraction_axes=contraction_axes,
51        output_match_topology=True)
52
53      # Absorb inverse sqrt into b
54      x_b_inv = sqrt_dict[edge][3]
55      b_node = self.graph.nodes[edge.node2]
56      b_axes = list(range(-1, -1 * len(b_node.shape) - 1, -1))
57      b_axes[edge.axis2] = 1
58      contraction_axes = [b_axes, [1, min(b_axes) - 1]]
59      b_node.tensor = self._perform_contraction_on_tensors(
60        b_node.tensor,
61        x_b_inv,
62        contraction_axes=contraction_axes,
63        output_match_topology=True)
```

Listing A.5: Transforming a tensor network into canonical form.

Shifting Centers of Orthogonality

```
1  def shift_center(self, center: str):
2    # Keep track of parents during the traversal
3    parents = {center: None}
4
5    # Keep track of link tensors we need to absorb
6    absorption_dict = {}
7
8    # Keep track of axes along that a tensor is an isometry after absorption
9    # of link tensors
10   self._isometric_dict = {}
11
```

```
12  # Mark center as visited
13  visited = [center]
14
15  queue = [center]
16
17  while len(queue) != 0:
18    # Pop the next node from the queue
19    current_node = queue.pop(0)
20
21    if current_node not in visited:
22      visited.append(current_node)
23
24      # If the current node is a link-tensor we add it to its parent's
25      # absorption dict
26      if current_node in self._link_tensors:
27        if current_node not in parents or parents[current_node] is None:
28          raise RuntimeError(
29            "Cannot␣shift␣orthogonality␣center␣to␣link-tensor")
30        parent_node = parents[current_node]
31        if parent_node not in absorption_dict:
32          absorption_dict[parent_node] = []
33        absorption_dict[parent_node].append(current_node)
34
35    # Find neighbors of current node and add them to the queue
36    neighbors = self.graph.neighbors(current_node, include_axes=True)
37    for neighbor in neighbors:
38      if neighbor[0] not in visited:
39        queue.append(neighbor[0])
40        parents[neighbor[0]] = current_node
41        self._isometric_dict[neighbor[0]] = neighbor[1][0]
42
43  # Absorb link-tensors according to absorption_dict
44  for node, links in absorption_dict.items():
45    self.contract([node] + links,
46      absorb=True,
47      new_name=str(hash(node)),
48      output_match_topology=True)
49
50  # Rename all nodes to match original names
51  for node in absorption_dict:
52    self.graph.rename_node(str(hash(node)), node)
53
54  self._center = center
```

Listing A.6: Shifting the center of orthogonality.

Gradient computation

```
def inner_product_gradient(self, node, nodes):
  # Insert conjugated nodes and conjugated edges
  conjugated_nodes, conjugated_edges = self._insert_conjugated_nodes(
    nodes, [])

  # Cut out node and edges
  removed_edges = []
  for edge in list(self.graph.edges):
    if edge.node1 == node or edge.node2 == node:
      removed_edges.append(edge)
      self.graph.edges.remove(edge)

  removed_node = self.graph.nodes[node]

  self.graph.remove_node(node)

  contraction_nodes = [n for n in nodes if n != node
    ] + [n for n in conjugated_nodes if n != node]

  # get actual gradient
  grad = self.contract(contraction_nodes)

  # Re-insert node and edges
  self.graph.add_node(removed_node)
  for edge in removed_edges:
    self.graph.add_edge(edge)

  for edge in conjugated_edges:
    self.graph.remove_edge(edge)

  for n in conjugated_nodes:
    self.graph.remove_node(n)

  return grad
```

Listing A.7: Computing the gradient of an inner product.

Riemannian Gradient Descent

```
def _tangent_space_projection(X, Z):
  sym = lambda W: 0.5 * (W + W.conj().T)
  return Z - X @ sym(X.conj().T @ Z)


def _stiefel_manifold_retraction(X, A):
  return scipy.linalg.polar(X + A)[0]

```

```

def riemannian_gradient_descent(network,
                                nodes,
                                grad,
                                step_size=0.001,
                                max_iterations=1000,
                                eps=1e-7):
  iteration = 0
  while max_iterations == -1 or iteration < max_iterations:
    iteration += 1
    norm = 0

    for i, node in enumerate(nodes):
      # Get tensor at node as 2d isometry
      U = network.Nd_to_isometric_2d(node)
      assert np.allclose(U.T @ U, np.eye(U.shape[1]))

      # Get euclidean gradient w.r.t. node
      euclidean_gradient = grad(node).reshape(U.shape)

      # Project gradient onto tanget space
      G = _tangent_space_projection(U, euclidean_gradient)

      # Update norm
      norm += np.linalg.norm(G)

      # Update tensor by using retraction from tangent space onto manifold
      network.graph.nodes[node].tensor = network.isometric_2d_to_Nd(
          node, _stiefel_manifold_retraction(U, -1 * step_size * G))

    if norm < eps:
      break
```

Listing A.8: Riemannian gradient descent on tensor network.

Conjugate Gradient with backtracking line-search

```
# Construct tensor network
network = tn.Network(tensors=[("C", np.linalg.qr(np.random.rand(4, 6))[0])],
                     shapes=[("A", 4, 3), ("B", 3, 5, 7), ("D", 5, 7, 10),
                             ("F", 10, 11), ("E", 7, 9)],
                     edges=[("A", 1, "B", 0), ("B", 1, "D", 0),
                            ("F", 0, "D", 2), ("D", 1, "E", 0),
                            ("C", 0, "A", 0)])

# Canonicalize and shift center to "A"
network.canonicalize()
network.shift_center("A")

manifolds = []
```

```
# "A" is embedded in a simple Euclidean manifold
a_shape = network.tensor("A").shape
manifolds.append(pymanopt.manifolds.Euclidean(a_shape[0], a_shape[1]))

# All nodes but "A" are isometries. Construct Stiefel manifolds for them
isometric_nodes = ["B", "C", "D", "E", "F"]

for node in isometric_nodes:
  shape = network.Nd_to_isometric_2d(node).shape
  manifolds.append(pymanopt.manifolds.Stiefel(shape[0], shape[1]))

# Construct product manifold
product_manifold = pymanopt.manifolds.Product(manifolds)

H = np.load("hermitian_operator_9_11_9_11.npy")


# Cost function:
#
# <T|H|T>
# -------
# <A|A>
def contract(a, b, c, d, e, f, a_adj, b_adj, c_adj, d_adj, e_adj, f_adj):
  return opt_einsum.contract(
      a, [1, 2], b, [2, 5, 8], c, [1, 9], d, [5, 4, 3], e, [4, 6], f, [3, 7], H,
      [6, 7, 15, 16], a_adj, [10, 11], b_adj, [11, 12, 8], c_adj, [10, 9],
      d_adj, [12, 14, 13], e_adj, [14, 15], f_adj, [13, 16],
      []) / opt_einsum.contract(a, [1, 2], a_adj, [1, 2], [])


@pymanopt.function.Callable
def cost(a, b, c, d, e, f):
  return contract(
      a,
      network.isometric_2d_to_Nd("B", b),
      network.isometric_2d_to_Nd("C", c),
      network.isometric_2d_to_Nd("D", d),
      network.isometric_2d_to_Nd("E", e),
      network.isometric_2d_to_Nd("F", f),
      a,
      network.isometric_2d_to_Nd("B", b),
      network.isometric_2d_to_Nd("C", c),
      network.isometric_2d_to_Nd("D", d),
      network.isometric_2d_to_Nd("E", e),
      network.isometric_2d_to_Nd("F", f),
  )


# Gradients in Euclidean space
tensor_grads = [jax.jit(jax.grad(contract, i + 6)) for i in range(6)]

```

```
65
66  @pymanopt.function.Callable
67  def egrad(u0, u1, u2, u3, u4, u5):
68    a = u0
69    b = network.isometric_2d_to_Nd("B", u1)
70    c = network.isometric_2d_to_Nd("C", u2)
71    d = network.isometric_2d_to_Nd("D", u3)
72    e = network.isometric_2d_to_Nd("E", u4)
73    f = network.isometric_2d_to_Nd("F", u5)
74
75    return [2 * tensor_grads[0](a, b, c, d, e, f, a, b, c, d, e, f)] + [
76        network.Nd_to_isometric_2d(node,
77                                   2 * grad(a, b, c, d, e, f, a, b, c, d, e, f))
78        for node, grad in zip(isometric_nodes, tensor_grads[1:])
79    ]
80
81
82  # Optimization problem
83  problem = pymanopt.Problem(product_manifold,
84                             cost=cost,
85                             egrad=egrad,
86                             verbosity=0)
87
88  # Conjugate gradient with backtracking line-search
89  linesearch = pymanopt.solvers.linesearch.LineSearchBackTracking()
90  solver = pymanopt.solvers.ConjugateGradient(linesearch=linesearch)
91
92  # Solve the optimization problem
93  solver.solve(problem, verbosity=2)
```

Listing A.9: Conjugate Gradient with backtracking line-search.

Bibliography

[AMS09] P.-A. Absil, R. Mahony, and R. Sepulchre. *Optimization Algorithms on Matrix Manifolds* -. Kassel: Princeton University Press, 2009. ISBN: 140-0-830-249-.

[And+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback).

[BB17] J. Biamonte and V. Bergholm. "Tensor Networks in a Nutshell." In: (July 31, 2017). arXiv: 1708.00006 [quant-ph].

[BC17] J. C. Bridgeman and C. T. Chubb. "Hand-waving and interpretive dance: an introductory course on tensor networks." In: *Journal of Physics A: Mathematical and Theoretical* 50.22 (2017), p. 223001.

[BD13] T. Bröcker and T. Dieck. *Representations of Compact Lie Groups* -. Berlin Heidelberg: Springer Science & Business Media, 2013. ISBN: 978-3-662-12918-0.

[Bia19] J. Biamonte. "Lectures on Quantum Tensor Networks." In: (Dec. 20, 2019). arXiv: 1912.10049 [quant-ph].

[Boo03] W. M. Boothby. *An Introduction to Differentiable Manifolds and Riemannian Geometry, Revised*. Orlando, Florida: Gulf Professional Publishing, 2003. ISBN: 978-0-121-16051-7.

[Bou+14] N. Boumal, B. Mishra, P.-A. Absil, and R. Sepulchre. "Manopt, a Matlab Toolbox for Optimization on Manifolds." In: *Journal of Machine Learning Research* 15.42 (2014), pp. 1455–1459.

[Bou22] N. Boumal. *An introduction to optimization on smooth manifolds*. To appear with Cambridge University Press. Jan. 2022.

[Bra+18] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018.

[Che+18] J. Chen, F. Zhang, C. Huang, M. Newman, and Y. Shi. "Classical Simulation of Intermediate-Size Quantum Circuits." In: (May 3, 2018). arXiv: 1805.01450 [quant-ph].

[Cic14] A. Cichocki. "Era of Big Data Processing: A New Approach via Tensor Networks and Tensor Decompositions." In: (Mar. 9, 2014). arXiv: 1403.2048 [cs.ET].

[CV09] J. I. Cirac and F. Verstraete. "Renormalization and tensor product states in spin chains and lattices." In: *Journal of Physics A: Mathematical and Theoretical* 42.50 (Dec. 2009), p. 504004. DOI: 10.1088/1751-8113/42/50/504004.

[EV07] G. Evenbly and G. Vidal. "Algorithms for entanglement renormalization." In: *Phys. Rev. B 79, 144108 (2009)* (July 10, 2007). DOI: 10.1103/PhysRevB.79.144108. arXiv: 0707.1454 [cond-mat.str-el].

[Eve18] G. Evenbly. "Gauge fixing, canonical forms and optimal truncations in tensor networks with closed loops." In: *Phys. Rev. B 98, 085155 (2018)* (Jan. 16, 2018). DOI: 10.1103/PhysRevB.98.085155. arXiv: 1801.05390 [quant-ph].

[EY36] C. Eckart and G. Young. "The approximation of one matrix by another of lower rank." In: *Psychometrika* 1.3 (1936), pp. 211–218.

[FWS20] M. Fishman, S. R. White, and E. M. Stoudenmire. *The ITensor Software Library for Tensor Network Calculations*. 2020. arXiv: 2007.14822.

[GL13] G. H. Golub and C. F. V. Loan. *Matrix Computations*. London: JHU Press, 2013. ISBN: 978-1-421-40859-0.

[Hae+12] J. Haegeman, M. Mariën, T. J. Osborne, and F. Verstraete. "Geometry of Matrix Product States: metric, parallel transport and curvature." In: *J. Math. Phys. 55, 021902 (2014)* (Oct. 29, 2012). DOI: 10.1063/1.4862851. arXiv: 1210.7710 [quant-ph].

[Has07] M. B. Hastings. "An Area Law for One Dimensional Quantum Systems." In: *JSTAT, P08024 (2007)* (May 14, 2007). DOI: 10.1088/1742-5468/2007/08/P08024. arXiv: 0705.2024 [quant-ph].

[HDH20] M. Hauru, M. V. Damme, and J. Haegeman. "Riemannian optimization of isometric tensor networks." In: *SciPost Phys. 10, 040 (2021)* (July 7, 2020). DOI: 10.21468/SciPostPhys.10.2.040. arXiv: 2007.03638 [quant-ph].

[HP18] J. Hauschild and F. Pollmann. "Efficient numerical simulations with Tensor Networks: Tensor Network Python (TeNPy)." In: *SciPost Phys. Lect. Notes* (2018). Code available from https://github.com/tenpy/tenpy, p. 5. DOI: 10.21468/SciPostPhysLectNotes.5. arXiv: 1805.00055.

[Hua+20] C. Huang, F. Zhang, M. Newman, J. Cai, X. Gao, Z. Tian, J. Wu, H. Xu, H. Yu, B. Yuan, M. Szegedy, Y. Shi, and J. Chen. "Classical Simulation of Quantum Supremacy Circuits." In: (May 14, 2020). arXiv: 2005.06787 [quant-ph].

[Lee13] J. M. Lee. *Introduction to Smooth Manifolds* -. Berlin Heidelberg: Springer Science & Business Media, 2013. ISBN: 978-0-387-21752-9.

[LKF20] I. A. Luchnikov, M. E. Krechetov, and S. N. Filippov. "Riemannian geometry and automatic differentiation for optimization problems of quantum physics and quantum technologies." In: *New J. Phys. 23, 073006 (2021)* (July 2, 2020). DOI: 10.1088/1367-2630/ac0b02. arXiv: 2007.01287 [quant-ph].

[Lya+22] D. I. Lyakh, A. McCaskey, J. Osborn, and T. Nguyen. *exatn*. https://github.com/ORNL-QCI/exatn. 2022.

[McC08] I. P. McCulloch. "Infinite size density matrix renormalization group, revisited." In: (Apr. 16, 2008). arXiv: 0804.2509 [cond-mat.str-el].

[Oru13] R. Orus. "A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States." In: *Annals of Physics 349 (2014) 117-158* (June 10, 2013). DOI: 10.1016/j.aop.2014.06.013. arXiv: 1306.2164 [cond-mat.str-el].

[Pen71] R. Penrose. "Applications of negative dimensional tensors." In: *Combinatorial mathematics and its applications* 1 (1971), pp. 221–244.

[Rob+19] C. Roberts, A. Milsted, M. Ganahl, A. Zalcman, B. Fontaine, Y. Zou, J. Hidary, G. Vidal, and S. Leichenauer. "TensorNetwork: A Library for Physics and Machine Learning." In: (May 3, 2019). arXiv: 1905.01330 [physics.comp-ph].

[Sch11] U. Schollwöck. "The density-matrix renormalization group in the age of matrix product states." In: *Annals of physics* 326.1 (2011), pp. 96–192.

[Smi14] S. T. Smith. "Optimization Techniques on Riemannian Manifolds." In: *Fields Institute Communications; Volume: 3; 1994* (July 22, 2014). arXiv: 1407.5965 [math.OC].

[Str13] G. Strang. *Lineare Algebra* -. Berlin Heidelberg New York: Springer-Verlag, 2013. ISBN: 978-3-642-55631-9.

[TKW16] J. Townsend, N. Koep, and S. Weichwald. "Pymanopt: A Python Toolbox for Optimization on Manifolds using Automatic Differentiation." In: *Journal of Machine Learning Research* 17.137 (2016), pp. 1–5.

[VCM09] F. Verstraete, J. I. Cirac, and V. Murg. "Matrix Product States, Projected Entangled Pair States, and variational renormalization group methods for quantum spin systems." In: *Adv. Phys. 57,143 (2008)* (July 16, 2009). DOI: 10.1080/14789940801912366. arXiv: 0907.2796 [quant-ph].

[VHV18] L. Vanderstraeten, J. Haegeman, and F. Verstraete. "Tangent-space methods for uniform matrix product states." In: *SciPost Phys. Lect. Notes 7 (2019)* (Oct. 16, 2018). DOI: 10.21468/SciPostPhysLectNotes.7. arXiv: 1810.07006 [cond-mat.str-el].

[Vid03] G. Vidal. "Efficient Classical Simulation of Slightly Entangled Quantum Computations." In: *Phys. Rev. Lett.* 91 (14 Oct. 2003), p. 147902. DOI: 10.1103/PhysRevLett.91.147902.

[Vid07] G. Vidal. "Classical Simulation of Infinite-Size Quantum Lattice Systems in One Spatial Dimension." In: *Phys. Rev. Lett.* 98 (7 Feb. 2007), p. 070201. DOI: 10.1103/PhysRevLett.98.070201.