Exercise Sheet Tensor Methods

CEMRACS 2021

By Mazen Ali[*] & Anthony Nouy

July 21st, 2021

For this exercise, download the folder

`https://github.com/MazenAli/CEMRACS2021/exercise`

There, you will find the file `exercise.pdf` (this document), a Jupyter notebook `exercise.ipynb` containing the programming exercises and a Jupyter notebook `exercise_solution.ipynb` containing the solutions to the programming exercises. We strongly recommend you work the problems before consulting the solutions.

For part I of this exercise, you will need `python`, the software package `numpy` and the `Jupyter Notebook`. For part III, you will additionally need the software package `tensap` that you can download from

`https://anthony-nouy.github.io/tensap`

or install via `pip install tensap`. You will also need the tutorial file

`tutorials/approximation/tutorial_PCA_FunctionalTensorPrincipalComponentAnalysis.py`

contained in the download folder. Part III is based on the paper

Nouy, A. *Higher-order principal component analysis for the approximation of tensors in tree-based low-rank formats.* Numer. Math. **141**, 743–789 (2019).

Abbreviations:

- MPS = Matrix Product State
- PCA = Principal Component Analysis
- SVD = Singular Value Decomposition
- TT = Tensor Train
- TTN = Tree Tensor Network
- TN = Tensor Network

Part I: Basic TN Algebra (`python`)

Tensor networks come with an intuitive graphical notation known as *tensor diagrams* or *Penrose graphical notation.* It is convenient and often necessary to use diagrams to understand, reason and compute with TNs.

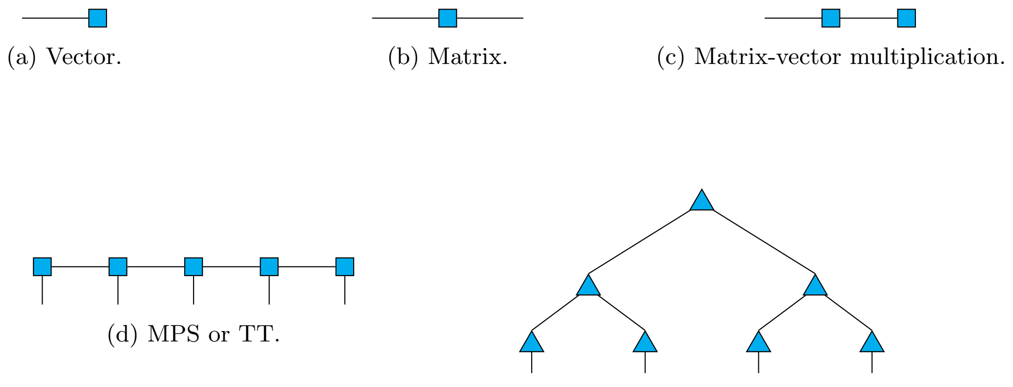
In its simplest form, a TN diagram is an undirected graph with vertices and edges. A vertex corresponds to a tensor – sometimes referred to as a *core* – and an edge corresponds to an index of that tensor. Connected edges between vertices represent *contractions* of tensors – multiplying and summing over the corresponding index. The number of summands corresponding to an edge is referred to as *rank*, and the number of outgoing edges from a vertex is referred to as the *order* of the tensor corresponding to that vertex. A word of caution, however: in physics and geometry it is common to refer to the order as the *rank, degree, dimension, total order, valence* or *type* (in the case of co- and contravariant indices) of the tensor and the rank as the *bond dimension* or *dimension.*

Elementary tensor products $A \otimes B$ can be represented by either putting two vertices next to each other without a connecting edge or, equivalently, a (trivial) connecting edge corresponding to rank one. *Free, dangling* or *unconnected* edges of a tensor correspond to unspecified (input) indices, i.e., their overall number determines the order of the tensor object represented by the network. For instance, an order-d tensor object $X \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ as a TN must have d free edges and can have any number of *internal* or *connected* edges.

[*]`mazen.ali@ec-nantes.fr`

In particular, this means a TN can have any number of internal vertices – without free edges – that do not affect the overall order of the tensor object.

TN representations are not unique which is sometimes referred to as *gauge freedom* in physics. For reasons of numerical stability and certain theoretical considerations, some TN representations are favored over others – e.g., TN representations with orthonormal cores. In general, TNs can have loops – most common examples include *matrix product states with periodic boundary conditions (MPS with periodic BC)* or *tensor rings (TR)*, *projected entangled pair states (PEPS)* and the *multiscale entanglement renormalization ansatz (MERA)* – and these are essential for certain applications. However, due to multiple theoretical and numerical difficulties, loop free *tree tensor networks* are more common and better studied. In Figure 1, you will find some examples of tensor diagrams. Familiarize yourself with this notation as you will need it for the following exercises.


(a) Vector. (b) Matrix. (c) Matrix-vector multiplication.

(d) MPS or TT. (e) Hierarchical Tucker.

Figure 1: Examples of TNs.

Exercise 1: TN to Tensor Conversion

1. Implement a function that converts a TN representation from Figure 2 into an explicitly entry-wise stored tensor $X \in \mathbb{R}^{n_1 \times n_2 \times n_3}$. We added labels to vertices and edges for your convenience, but you are free to use any labeling you like.

   Hint: the `numpy` function `tensordot` might be helpful.

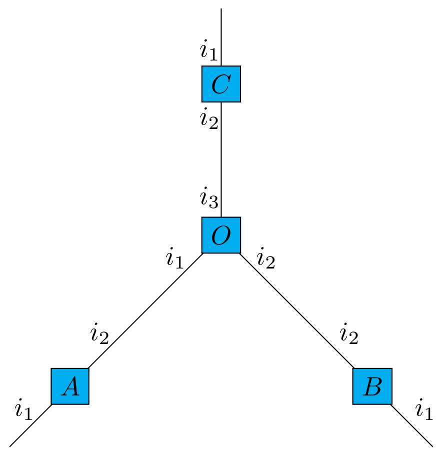


Figure 2

2. Implement a function that converts a TN representation from Figure 3 into an explicitly entry-wise stored tensor $X \in \mathbb{R}^{n_1 \times n_2 \times n_3}$.

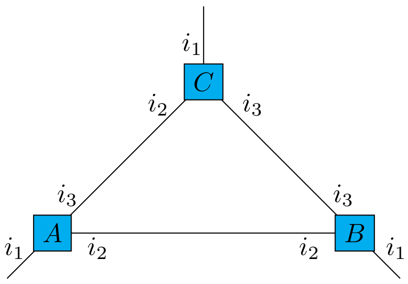


Figure 3

Exercise 2: Tensor to TN Conversion

1. Implement a function that converts an explicitly entry-wise stored tensor $X \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ into a TN representation corresponding to Figure 2.

   Hint: use a matrix factorization method.

2. Implement a function that converts an explicitly entry-wise stored tensor $X \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ into a TN representation corresponding to Figure 3.

   Hint: any index/edge can be split in two by reshaping. You have some freedom in deciding how to reshape.

Exercise 3: TT Truncation

The most commonly used TNs are *matrix product states* or *tensor trains*, see Figure 1d. The nodes correspond to order-2 and order-3 tensors (*cores*) $U_1 \in \mathbb{R}^{n_1 \times r_1}$, $U_j \in \mathbb{R}^{r_{j-1} \times n_j \times r_j}$, $j = 2, \ldots, d-1$, $U_d \in \mathbb{R}^{n_d \times r_d}$. The TT ranks $\{r_j\}_{j=1}^d$ corresponding to the edges in Figure 1d determine the complexity of a TN representation. An important TN operation is *truncation*: for a TT $X = \mathrm{TT}(U_1, \ldots, U_d)$, find a representation $\tilde{X} = \mathrm{TT}(\tilde{U}_1, \ldots, \tilde{U}_d)$ with smaller ranks $\tilde{r}_j \leq r_j$ with error $\|X - \tilde{X}\|_2$ as small as possible. This is commonly implemented via a *higher-order singular value decomposition.*

Suppose we want to truncate r_3 in Figure 1d – of course, preferably without computing the full tensor $X \in \mathbb{R}^{n_1 \times \cdots \times n_5}$ explicitly. If $d = 2$ and $X \in \mathbb{R}^{n_1 \times n_2}$ is a rank-r matrix, an SVD of X is the factorization

$$X = U \Sigma V^*,$$

where U is *left orthonormal* – $U^* U = I_{r \times r}$ – and V^* is *right orthonormal* – $V^* V = I_{r \times r}$, and Σ is the diagonal matrix of singular values. For $d = 5$ and X as in Figure 1d, we can write

$$X(i_1, \ldots, i_5) = U_1(i_1) U_2(i_2) U_3(i_3) U_4(i_4) U_5(i_5). \tag{1}$$

Suppose U_1, U_2 are left orthonormal, i.e., the unfoldings $U_j([m, i_j]; k)$ – where $[m, i_j]$ is reshaped into a single row index and k is the column index – are left orthonormal, i.e., the columns $k = 1, 2, \ldots$ of the tensor U_j reshaped into the matrix $U_j([m, i_j]; k)$ are orthonormal.

Similarly, suppose U_4, U_5 are right orthonormal – the unfoldings $U_j(m; [i_j, k])$ are right orthonormal – then an SVD of the unfolding $U_3([m, i_3]; k)$ also provides an SVD of the TT from Figure 1d w.r.t. the 3rd connected edge corresponding to r_3.

1. Implement a function that, for a given TT representation and a given fixed mode number μ, computes a TT representation of the same tensor such that all cores to the left of μ are left orthnormal, and all cores to the right of μ are right orthonormal.

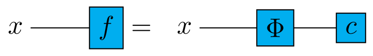
2. Implement a function that, for a given fixed mode μ, a fixed rank r_μ and a TT representation with orthogonal cores, computes a TT representation truncated in the μ-th core. Estimate the resulting ℓ^2-error via the truncated singular values. Note that, for a truncation in a single core, the ℓ^2-error should correspond *exactly* to the sum of the squared truncated singular values.

Part II: TNs for Functions (pen & paper)

There are several possibilites for adapting the TN framework to functions. In principle, an abstract TN representation does not require a basis of functions. However, for numerical computations, continuous input variables such as $x \in \mathbb{R}$ have to be discretized by, e.g., introducing a basis $\Phi := \{\varphi_i\}_i$ – see below for an illustration. In the multidimensional case, one can use a tensor product basis. TNs can be applied to one-dimensional functions as well by introducing artificial variables through a process known as *tensorization* or *quantization*.

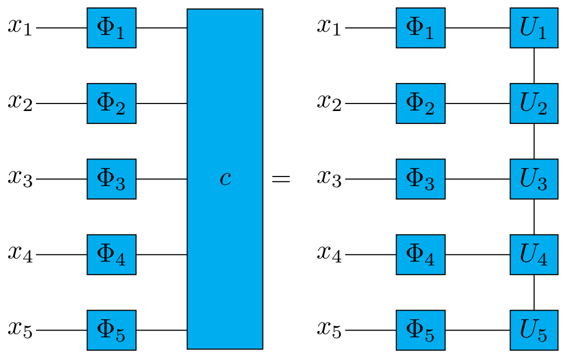
A one-dimensional function f expanded in a basis Φ:

$$f(x) = \sum_i c_i \varphi_i(x) =$$



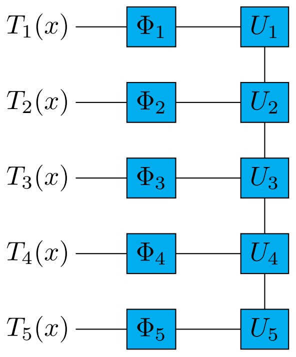
A 5-dimensional function f expanded in a tensor product basis and as a TT:

$$f(x_1, \ldots, x_5) =$$



Tensorization for a one-dimensional function, applying a general (possibly nonlinear) transformation $T(x) = (T_1(x), \ldots, T_5(x))$:

$$x \longmapsto$$



Exercise 4: Explicit TN Function Representations

Write down explicit TTN representations of the following functions. Think about the TN structure and required ranks.

1.

$$u(x_1, x_2, x_3, x_4) = a_1(x_1)b_1(x_2)c_1(x_3)d(x_4) + a_2(x_1)b_2(x_2)c_2(x_3)d(x_4)$$

2.

$$u(x_1, x_2, x_3, x_4) = a(x_1)c_1(x_3) + c_2(x_3)d_1(x_4) + d_2(x_4)b(x_2)$$

3.

$$\begin{aligned} u(x_1, x_2, x_3, x_4) = {} & b_1(x_2)c_1(x_3) + b_2(x_2)c_2(x_3) \\ & + b_1(x_2)c_2(x_3)a_2(x_1)d_1(x_4) \\ & + a_1(x_1)d_1(x_4) + a_2(x_1)d_2(x_4) \end{aligned}$$

4. We can tensorize one-dimensional functions by introducing the transformation

$$x = T^{-1}(i_1, i_2, i_3, y) := 2^{-1}i_1 + 2^{-2}i_2 + 2^{-3}i_3 + 2^{-3}y,$$

for binary i_1, i_2, $i_3 \in \{0, 1\}$ and $y \in [0, 1)$. Any $u : [0, 1) \to \mathbb{R}$ is then tensorized via

$$\boldsymbol{u}(i_1, i_2, i_3, y) := u(T^{-1}(i_1, i_2, i_3, y)).$$

With the above tensorization, find an explicit TT representation of the linear function $u(x) \equiv x$.

Part III: Advanced Applications (`python`)

Tensor networks provide a powerful tool for analysis, approximation and simulation of high-dimensional problems. Some common applications include eigenvalue problems in quantum mechanics, PDEs (both high- and low-dimensional with quantized/tensorized formats), parameter dependent problems in model order reduction, uncertainty quantification, compression, belief networks, neural networks, learning and many more. In this exercise, we will examine one particular use case implemented in the `tensap` library that you can download from

`https://anthony-nouy.github.io/tensap/`

We seek to approximate a function $u : \Omega \to \mathbb{R}$ based on samples $\{u(x) : \; x \in S \subset \Omega\}$. Within a tree tensor network, this can be accomplished with a *higher-order principal component analysis* (PCA) detailed in the paper

Nouy, A. *Higher-order principal component analysis for the approximation of tensors in tree-based low-rank formats.* Numer. Math. **141**, 743–789 (2019).

This procedure uses *partial evaluations* of the function u in some of the variables $(x_1, \ldots, x_d)$, while interpolating u on low-dimensional nested subspaces in the complementary variables. We briefly elaborate.

Let $\alpha \subset \{1, \ldots, d\}$ be a node in a TTN, $\alpha^c := \{1, \ldots, d\} \setminus \alpha$ and consider the variables $x_\alpha := (x_i)_{i \in \alpha}$ and x_{α^c} defined accordingly. Furthermore, assume α has exactly two children in the TTN, α_1 and α_2. If we are given two low-dimensional subspaces U_{α_1} and U_{α_2}, and partial evaluations $u(\cdot; x_{\alpha^c})$ for some randomly sampled x_{α^c}, then we can interpolate each of these partial evaluations in the tensor product space $U_{\alpha_1} \otimes U_{\alpha_2}$. This interpolation is of low complexity provided both U_{α_1} and U_{α_2} are low-dimensional.

We thus obtain an interpolation of the partial evaluations $u(\cdot; x_{\alpha^c})$ in the variables x_α for each sampled x_{α^c}, and their linear span over all sampled x_{α^c} defines a subspace $V_\alpha \subset U_{\alpha_1} \otimes U_{\alpha_2}$ in the variables x_α. Finally, we apply SVD to determine the r_α-dominant vectors of this subspace which defines the new interpolation space $U_\alpha \subset V_\alpha$. We then proceed to the next node in the TTN. The complexity of the overall procedure depends on the ranks r_α of the TTN and the number of samples for the PCA/partial evaluations.

Exercise 5: Higher-Order PCA

1. Edit the source code of the file

   `tutorials/approximation/tutorial_PCA_FunctionalTensorPrincipalComponentAnalysis.py`

   such that it uses the Henon-Heiles potential as the test case and run it. What do you observe for the required ranks and tree structure?

2. Run the same example with a random tree. What do you observe? Experiment with the library.