tensap—A Python Tensor Approximation Package

Anthony Nouy[*] Erwan Grelier[*]

July 29, 2020

Abstract

This article provides an introduction to tensap (Tensor Approximation Package), which is a Python package for the approximation of functions and tensors, available on GitHub at `https://github.com/anthony-nouy/tensap`, or through its GitHub page `https://anthony-nouy.github.io/tensap/`. The package tensap features low-rank tensors (including canonical, tensor train and tree-based tensor formats or tree tensor networks), sparse tensors, polynomials, and allows the plug-in of other approximation tools. It provides different approximation methods based on interpolation, least-squares projection or statistical learning.

[*]Centrale Nantes, Laboratoire de Mathématiques Jean Leray, CNRS UMR 6629

Contents

Introduction

tensap (Tensor Approximation Package) is a Python package for the approximation of functions and tensors, available on GitHub at `https://github.com/anthony-nouy/tensap`, or through its GitHub page `https://anthony-nouy.github.io/tensap/`.

To install from PyPi, run `pip install tensap`. Alternatively, you can install tensap directly from github by running
`pip install git+git://github.com/anthony-nouy/tensap@master`.

The package tensap features low-rank tensors (including canonical, tensor-train and tree-based tensor formats or tree tensor networks), sparse tensors, polynomials, and allows the plug-in of other approximation tools. It provides different approximation methods based on interpolation, least-squares projection or statistical learning.

The package is shipped with tutorials showing its main applications. A documentation is also available.

At minimum, tensap requires the packages numpy and scipy. The packages tensorflow and sklearn are required for some applications.

In this document, all Python commands are written using a `typewriter font`. The quantities in `typewriter font` are Python objects, whereas the quantities in *math font* are mathematical objects. For better readability, we use the same letter for an object, with different fonts whether we refer to its mathematical definition (X) or to its Python implementation (`X`).

1 FullTensor

A `FullTensor X` represents an order d tensor $X \in \mathbb{R}^{N_1\times\cdots\times N_d}$, or multidimensional array of size $N_1 \times \ldots \times N_d$. The entries of X are $X_{i_1,\ldots,i_d}$, with $(i_1,\ldots,i_d)$ a tuple of indices, where $i_\nu \in \{0,\ldots,N_\nu - 1\}$ is related to the ν-th mode of the tensor.

We present in this section how to create a `FullTensor` using tensap, and several possible operations with such an object. For an introduction to tensor calculus, we refer to the monograph [6].

For examples of use, see the tutorial file `tutorials\tensor_algebra\tutorial_FullTensor.py`.

1.1 Creating a `FullTensor`

Provided with an array `data` of shape `[N_1, ..., N_d]`, the command `X = tensap.FullTensor(data)` returns a tensor $X \in \mathbb{R}^{N_1 \times \cdots \times N_d}$, with order `X.order = d` and shape `X.shape = (N_1, ..., N_d)`. The number of entries of `X` is given by `X.size = X.storage()` $= \prod_{i=1}^{d} N_i$. The number of nonzero entries of `X` is given by `X.sparse_storage()`.

It is also possible to generate a `FullTensor` with entries:

- equal to 0 with `tensap.FullTensor.zeros([N_1, ..., N_d])`,
- equal to 1 with `tensap.FullTensor.ones([N_1, ..., N_d])`,
- drawn randomly according to the uniform distribution on $[0, 1]$ with `tensap.FullTensor.rand([N_1, ..., N_d])`,
- drawn randomly according to the standard gaussian distribution with `tensap.FullTensor.randn([N_1, ..., N_d])`,
- different from 0 only on the diagonal, provided in `diag_data`, with `tensap.FullTensor.diag(diag_data, d)` (generating a tensor of order d and shape $[N, \ldots, N]$, with $N =$ `len(diag_data)`),
- generated using a provided `generator` with `tensap.FullTensor.create(generator, [N_1, ..., N_d])`.

1.2 Accessing the entries of a `FullTensor`

The entries of a tensor `X` can be accessed with the method `eval_at_indices`: `X.eval_at_indices(ind)` returns the entries of X indexed by the list `ind` containing the indices to access in each dimension.

Extracting diagonal entries. For a tensor $X \in \mathbb{R}^{N,\ldots,N}$, the command `X.eval_diag()` returns the diagonal entries $X_{i,\ldots,i}$, $i = 1, \ldots, N$, of the tensor. The command `X.eval_diag(dims)` returns the entries $X_{i,\ldots,i}$, with i in `ind`.

Extracting a sub-tensor. A sub-tensor can be extracted from `X` with the method `sub_tensor`: for an order-3 `FullTensor X` of size $N_1 \times N_2 \times N_3$, `X.sub_tensor([0, 1], ':', 2)` returns a sub-tensor of size $2 \times N_2 \times 1$ containing the entries X_{i_1,i_2,i_3} with $i_1 \in \{0, 1\}$, $0 \leq i_2 \leq N_2 - 1$ and $i_3 = 2$.

1.3 Permuting the modes of a `FullTensor`

The methods `transpose` and `itranspose` permute the dimensions of a tensor `X`, given a permutation `dims` of $\{1, \dots, d\}$. They are such that `X = X.transpose(dims).itranspose(dims)`.

1.4 Reshaping a `FullTensor`.

The command `X.reshape(shape)` reshapes a `FullTensor` using a column-major order (e.g. used in Fortran, Matlab, R). It relies on the numpy's reshape function with Fortran-like index (argument `order='F'`). For a tuple $(i_1, \dots, i_d)$, we define

$$\overline{i_1, \dots, i_d} = i_1 + N_1(i_{2-1} - 1) + N_1 N_2 (i_{3-1} - 1) + \dots + N_1 \dots N_{d-1}(i_d - 1).$$

A tensor X is be identified with a vector $\text{vec}(X)$ whose entries are $\text{vec}(X)_{\overline{i_1,\dots,i_d}}$. This vector can be obtained with the command `X.reshape(N)` with `N=numpy.prod(X.shape)`.

α-Matricization. For $\alpha \subset \{1, \dots, d\}$ an its complementary subset α^c in $\{1, \dots, d\}$, an α-matricization of a tensor X is a matrix M of size $(\prod_{i \in \alpha} N_i) \times (\prod_{i \in \alpha^c} N_i)$, such that $X_{i_1,\dots,i_d} = M_{\overline{i_\alpha}, \overline{i_{\alpha^c}}}$ with $i_\alpha = (i_\nu)_{\nu \in \alpha}$. It can be obtained with `X.matricize(alpha)`, which returns a `FullTensor` or order 2. The matricization relies on the method `reshape`.

Orthogonalization. It is possible to obtain a representation of a tensor X such that its α-matricization is an orthogonal matrix (i.e. with orthogonal columns) using the method `X.orth(alpha)`.

1.5 Norms and singular-values

Computing the Frobenius norm of a FullTensor. The command `X.norm()` returns the Frobenius norm $\|X\|_F$ of X, defined by

$$\|X\|_F^2 = \sum_{i_1}^{N_1} \cdots \sum_{i_d}^{N_d} X_{i_1,\dots,i_d}^2.$$

Computing the α-singular values and α-principal components of a FullTensor. For a subset $\alpha \subset \{1, \dots, d\}$ and its complementary subset α^c, the α-matricization M of X admits a singular value decomposition

$$M_{i_\alpha, i_{\alpha^c}} = \sum_k \sigma^k v_{i_\alpha}^k w_{i_\alpha^c}^k$$

where the σ^k are the singular values of M and the v^k the corresponding left singular vectors, or principal components of M. They are respectively called the α-singular values and α-principal components of X. The α-singular values are obtained with `X.singular_values()`. The α-principal components (and α-singular values) are obtained with `X.alpha_principal_components(alpha)`, which is equivalent to `X.matricize(alpha).principal_components()`.

1.6 Operations with `FullTensor`

1.6.1 Outer product.

The outer product $X \circ Y$ of two tensors $X \in \mathbb{R}^{N_1 \times \cdots \times N_d}$ and $Y \in \mathbb{R}^{\hat{N}_1 \times \cdots \times \hat{N}_{\hat{d}}}$ is a tensor $Z \in \mathbb{R}^{N_1 \times \ldots \times N_d \times \hat{N}_1 \times \cdots \times \hat{N}_{\hat{d}}}$ of order $d + \hat{d}$ with entries

$$Z_{i_1,\ldots,i_d,j_1,\ldots,j_{\hat{d}}} = X_{i_1,\ldots,i_d} Y_{j_1,\ldots,j_{\hat{d}}}$$

It is provided by `X.tensordot(Y, 0)`, similarly to numpy's tensordot function.

1.6.2 Kronecker product.

The Kronecker product $X \otimes Y$ of two tensors X and Y of the same order $d = \hat{d}$ is a tensor Z of size $N_1\hat{N}_1 \times \ldots \times N_d\hat{N}_{\hat{d}}$ with entries

$$Z_{\overline{i_1 j_1},\ldots,\overline{i_d j_d}} = X_{i_1,\ldots,i_d} Y_{j_1,\ldots,j_d}.$$

It is given by the command `kron`, which is similar to numpy's kron function, but for arbitrary tensors.

1.6.3 Hadamard product.

The Hadamard (elementwise) product $X \circledast Y$ of two tensors X and Y of the same order and size is obtained through the command `__mul__(X,Y)`, which returns a tensor Z with entries

$$Z_{i_1,\ldots,i_d} = X_{i_1,\ldots,i_d} Y_{i_1,\ldots,i_d}$$

1.6.4 Contracted product.

For $I \subset \{1,\ldots,d\}$ and $J \subset \{1,\ldots,\hat{d}\}$ with $\#I = \#J$, `Z = X.tensordot(Y, I, J)` performs the mode (I,J)-contracted product of X and Z which is a tensor Z of order $d + \hat{d} - \#I - \#J$ with entries

$$Z_{(i_\nu)_{\nu \notin I},(j_\mu)_{\mu \notin J}} = \sum_{\substack{i_\nu=1\\ \nu\in I}}^{N_\nu} \sum_{\substack{j_\mu=1\\ \mu\in J}}^{N_\mu} \prod_{\nu\in I} \prod_{\mu\in J} \delta_{i_\nu,j_\mu} X_{i_1,\ldots,i_d} Y_{j_1,\ldots,j_{\hat{d}}}$$

with $\delta_{i,j}$ the Kronecker delta, that is a contraction of tensors X and Y along dimensions I of X and J of Y. For example, for order-4 tensors X and Y, `Z = X.tensordot(Y, [0,1], [1,2])` returns a tensor Z or order 4 such that

$$Z_{i_3,i_4,j_1,j_4} = \sum_{i_1,i_2} X_{i_1,i_2,i_3,i_4} Y_{j_1,i_1,i_2,j_4}.$$

The method `tensordot_eval_diag` provides the diagonal (or entries with equal pairs of indices) of the result of the method `tensor_dot`, but at a cost lower than when using `X.tensordot(Y, I, J).eval_diag()`.
For example, for order-4 tensors X and Y,
`X.tensordot_eval_diag(Y,[0,1],[1,2],[2,3],[0,3])` returns the diagonal of Z, i.e. an order-one tensor M with entries

$$M_k = Z_{k,k,k,k} = \sum_{i_1,i_2} X_{i_1,i_2,k,k} Y_{k,i_1,i_2,k}$$

`X.tensordot_eval_diag(Y,[0,1],[1,2],[2,3],[0,3],diag = True)` returns a tensor M of order 2 with entries

$$M_{k_1,k_2} = Z_{k_1,k_2,k_1,k_2} = \sum_{i_1,i_2} X_{i_1,i_2,k_1,k_3} Y_{k,i_1,i_2,k}$$

`X.tensordot_eval_diag(Y,[0,1],[1,2],[2],[0])` returns the diagonal of Z, i.e. a tensor M of order 3 v with entries

$$M_{k,i_4,j_4} = Z_{k,i_4,k,j_4} = \sum_{i_1,i_2} X_{i_1,i_2,k,i_4} Y_{k,i_1,i_2,j_4}$$

1.6.5 Dot product.

The dot product of two tensors X and Y with same shape $[N_1, \dots, N_d]$, defined by

$$(X, Y) = \sum_{\substack{i_\nu=1 \\ \nu=1,\dots,d}}^{N_\nu} X_{i_1,\dots,i_d} Y_{i_1,\dots,i_d}, \tag{1}$$

can be obtained with `X.dot(Y)`. It is equivalent to `X.tensordot(Y, range(X.order), range(Y.order))`.

1.6.6 Contractions with matrices or vectors

Given a tensor X and a list of matrices $M = [M^1, ..., M^d]$, the command `Z = X.tensor_matrix_product(M)` returns an order-d tensor Z whose entries are

$$Z_{i_1,\dots,i_d} = \sum_{\substack{k_\nu=1 \\ \nu=1,\dots,d}}^{N_\nu} X_{k_1,\dots,k_d} \prod_{\nu=1}^{d} M^\nu_{i_\nu,k_\nu}$$

The same method exists for vectors instead of matrices: `tensor_vector_product`. Similarly to `tensordot_eval_diag`, the method `tensor_matrix_product_eval_diag` evaluates the diagonal of the result of `tensor_matrix_product`, with a lower cost.

2 Tensor formats

Here we present tensor formats available in tensap, which are structured formats of tensors in $\mathbb{R}^{N_1\times\dots\times N_d}$. For a detailed description of methods, see the description of the corresponding methods for `FullTensor` in Section 1. For an introduction to tensor formats, we refer to the monograph [6] and the survey [8].

2.1 `CanonicalTensor`

The entries of an order-d tensor $X \in \mathbb{R}^{N_1\times\cdots\times N_d}$ in canonical format can be written

$$X_{i_1,\dots,i_d} = \sum_{k=1}^{r} C_k U^1_{i_1,k} \cdots U^d_{i_d,k},$$

with r the canonical rank, and where the $U_\nu = (U^\nu_{i_\nu,k})_{1\le i_\nu\le N_\nu, 1\le k\le r}$ are order-two tensors.

<u>**Creating a CanonicalTensor.**</u> To create a canonical tensor in tensap, one can use the command `tensap.CanonicalTensor(C, U)`, where `C` contains the $(C_k)_{k=1}^d$, and `U` is a list containing the U^ν, $1 \le \nu \le d$.

The storage complexity of such a tensor, obtained with `X.storage()`, is equal to $r(1 + N_1 + \cdots + N_d)$.
It is also possible to generate a `CanonicalTensor` with entries

- equal to 0 with `tensap.CanonicalTensor.zeros(r, [N_1, ..., N_d])`,
- equal to 1 with `tensap.CanonicalTensor.ones(r, [N_1, ..., N_d])`,
- drawn randomly according to the uniform distribution on $[0, 1]$ with `tensap.CanonicalTensor.rand(r, [N_1, ..., N_d])`,
- drawn randomly according to the standard gaussian distribution with `tensap.CanonicalTensor.randn(r, [N_1, ..., N_d])`,
- generated using a provided `generator` with `tensap.CanonicalTensor.create (generator, r, [N_1, ..., N_d])`.

<u>**Converting a CanonicalTensor to a FullTensor.**</u> A `CanonicalTensor` `X` can be converted to a `FullTensor` (introduced in Section 1) with the command `X.full()`.

Converting a CanonicalTensor to a TreeBasedTensor. A `CanonicalTensor X` can be converted to a `TreeBasedTensor` (introduced in Section 2.4) with the command `X.tree_based_tensor(tree, is_active_node)`, with `tree` a `DimensionTree` object, and `is_active_node` a list or array of booleans indicating if each node of the tree is active.

Accessing the diagonal of a CanonicalTensor. For a canonical tensor $X \in \mathbb{R}^{N\times\dots\times N}$, the command `X.eval_diag()` returns the diagonal $X_{i,\dots,i}$, $i = 1,\dots,N$, of the tensor. The method `eval_diag` can also be used to evaluate the diagonal in a subset of dimensions `dims` of the tensor with `X.eval_diag(dims)`, which returns a `CanonicalTensor`.

Computing the Frobenius norm of a CanonicalTensor. The command `X.norm()` returns the Frobenius norm of X. The Frobenius norm of X is equal to the Frobenius norm of its core C if `X.is_orth` is `True`.

Computing the derivative of CanonicalTensor with respect to one of its parameters. Given an order-d canonical tensor X in $\mathbb{R}^{N\times\dots\times N}$, the command `X.parameter_gradient_eval_diag(k)`, for $1 \le k \le d$, returns the derivative

$$\left.\frac{\partial X_{i_1,\dots,i_d}}{\partial U^k}\right|_{i_1=\dots=i_d=i}, \; i = 1,\dots,N.$$

The derivative of X with respect to its core C, that writes

$$\left.\frac{\partial X_{i_1,\dots,i_d}}{\partial C}\right|_{i_1=\dots=i_d=i}, \; i = 1,\dots,N,$$

is obtained with `X.parameter_gradient_eval_diag(d+1)`.

The method `parameter_gradient_eval_diag` is used in the statistical learning algorithms presented in Section 5.4.

Performing operations with CanonicalTensor. Some operations between tensors are implemented for `DiagonalTensor` (see Section 1.6 for a detailed description of the operations): the Kronecker product with `kron`, the contraction with matrices with `tensor_matrix_product`, the evaluation of the diagonal of a contraction with matrices with `tensor_matrix_product_eval_diag`, the dot product with `dot`.

Given a tensor X and a list of matrices $M = [M^1, ..., M^d]$, the command `Z = X.tensor_matrix_product(M)` returns an order-d tensor Z whose entries are

$$Z_{i_1,\dots,i_d} = \sum_{k=1}^{r} \sum_{\substack{k_\nu=1\\ \nu=1,\dots,d}}^{N_\nu} C_k U^1_{k_1,k} \cdots U^d_{k_d,k} \prod_{\nu=1}^{d} M^\nu_{i_\nu,k_\nu}$$

The method `tensor_matrix_product_eval_diag` evaluates the diagonal of the result of `tensor_matrix_product`.

The dot product of two canonical tensors X and Y with same shape $[N_1, \dots, N_d]$ can be obtained with `X.dot(Y)`.

2.2 `DiagonalTensor`

A diagonal tensor $X \in \mathbb{R}^{N_1 \times \cdots \times N_d}$ is a tensor whose entries $X_{i_1,\dots,i_d}$ are non-zero only if $i_1 = \cdots = i_d$.

Creating a DiagonalTensor. To create a diagonal tensor in tensap, one can use the command `tensap.DiagonalTensor(D, d)`, where `D` (of length r) contains the diagonal of the tensor, and `d` is the order of the tensor. The result if an order d tensor in $\mathbb{R}^{r \times \cdots \times r} = \mathbb{R}^{r^d}$.

The sparse storage complexity of such a tensor, obtained with `X.sparse_storage()`, is equal to `r = len(D)`. Its storage complexity, not taking into account the fact that only the diagonal is non-zero, is equal to r^d and obtained with `X.storage()`.

It is also possible to generate a `DiagonalTensor` with entries

- equal to 0 with `tensap.DiagonalTensor.zeros(r, d)`,
- equal to 1 with `tensap.DiagonalTensor.ones(r, d)`,
- drawn randomly according to the uniform distribution on $[0, 1]$ with `tensap.DiagonalTensor.rand(r, d)`,
- drawn randomly according to the standard gaussian distribution with `tensap.DiagonalTensor.randn(r, d)`,
- generated using a provided `generator` with `tensap.DiagonalTensor.create (generator, r, d)`.

Converting a DiagonalTensor to a FullTensor. A `DiagonalTensor X` can be converted to a `FullTensor` (introduced in Section 1) with the command `X.full()`.

Converting a DiagonalTensor to a SparseTensor. A `DiagonalTensor X` can be converted to a `SparseTensor` (introduced in Section 2.3) with the command `X.sparse()`.

Converting a DiagonalTensor to a TreeBasedTensor. A `DiagonalTensor X` can be converted to a `TreeBasedTensor` (introduced in Section 2.4) with the command `X.tree_based_tensor(tree, is_active_node)`, with `tree` a `DimensionTree` object, and `is_active_node` a list or array of booleans indicating if each node of the tree is active.

Accessing the entries of a DiagonalTensor. The entries of the tensor `X` can be accessed with the method `eval_at_indices`: `X.eval_at_indices(ind)` returns the entries of X indexed by the list `ind` containing the indices to access in each dimension.
A sub-tensor can be extracted from `X` with the method `sub_tensor`.

For a tensor $X \in \mathbb{R}^{N\times\cdots\times N}$, the command `X.eval_diag()` returns the diagonal $X_{i,\ldots,i}$, $i = 1, \ldots, N$, of the tensor. The method `eval_diag` can also be used to evaluate the diagonal in some dimensions `dims` of the tensor with `X.eval_diag(dims)`.

Computing the Frobenius norm of a DiagonalTensor. The command `X.norm()` returns the Frobenius norm of X.

Performing operations with DiagonalTensor. Some operations between tensors are implemented for `DiagonalTensor` (see Section 1.6 for a detailed description of the operations): the outer product with `tensordot`, the evaluation of the diagonal (or subtensors) of an outer product with `tensordot_eval_diag`, the Kronecker product with `kron`, the contraction with matrices or vectors with `tensor_matrix_product` or `tensor_vector_product` respectively, the evaluation of the diagonal of a contraction with matrices with `tensor_matrix_product_eval_diag`, the dot product with `dot`.

2.3 `SparseTensor`

A sparse tensor $X \in \mathbb{R}^{N_1\times\cdots\times N_d}$ is a tensor whose entries $X_{i_1,\ldots,i_d}$ are non-zero only for $(i_1, \ldots, i_d) \in I$, with I a set of multi-indices.

Creating a SparseTensor. To create a sparse tensor `X` in tensap, one can use the command `tensap.SparseTensor(D, I, [N_1, ..., N_d])`, where `D` contains the non-zero entries of X, `I` is a `tensap.MultiIndices` containing the indices of its non-zero enties, and where $N_1, \ldots, N_d$ is its shape.

The sparse storage complexity of such a tensor, obtained with `X.sparse_storage()`, is equal to $\mathrm{card}(I)$. Its storage complexity, not taking into account the sparsity, is equal to $N_1 \cdots N_d$ and can be accessed with `X.storage()`.

Converting a SparseTensor to a FullTensor. A `SparseTensor X` can be converted to a `FullTensor` (introduced in Section 1) with the command `X.full()`.

Converting a FullTensor to a SparseTensor. A `FullTensor X` can be converted to a `SparseTensor` (introduced in Section 2.3) with the command `X.sparse()`.

Accessing the entries of a SparseTensor. The entries of the tensor `X` can be accessed with the method `eval_at_indices`: `X.eval_at_indices(ind)` returns the entries of X indexed by the list `ind` containing the indices to access in each dimension.

A sub-tensor can be extracted from `X` with the method `sub_tensor`.

For a tensor $X \in \mathbb{R}^{N,\dots,N}$, the command `X.eval_diag()` returns the diagonal $X_{i,\dots,i}$, $i = 1, \dots, N$, of the tensor. The method `eval_diag` can also be used to evaluate the diagonal in some dimensions `dims` of the tensor with `X.eval_diag(dims)`.

Reshaping a SparseTensor. The method `reshape` reshapes a `SparseTensor` using the Fortran-like index order of numpy's reshape function.

The methods `transpose` and `itranspose` permute the dimensions of a tensor `X`, given a permutation `dims` of $\{1, \dots, d\}$. They are such that `X = X.transpose(dims).itranspose(dims)`.

Computing the Frobenius norm of a SparseTensor. The command `X.norm()` returns the Frobenius norm of X.

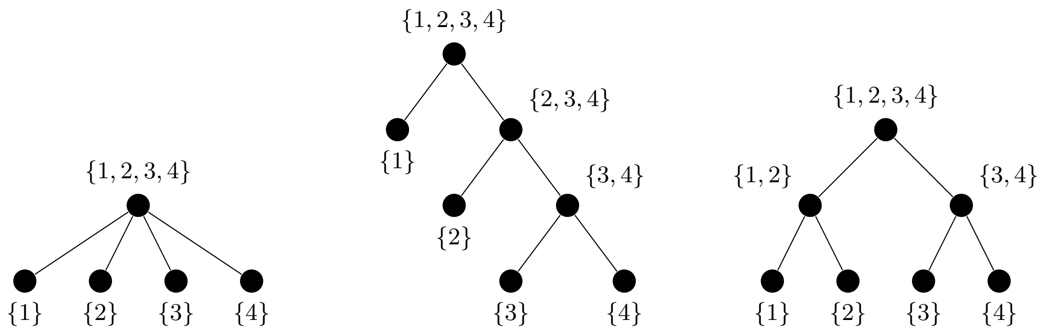
Performing operations with SparseTensor. Some operations between tensors are implemented for `SparseTensor` (see Section 1.6 for a detailed description of the operations): the Kronecker product with `kron`, the contraction with matrices or vectors with `tensor_matrix_product` or `tensor_vector_product` respectively, the evaluation of the diagonal of a contraction with matrices with `tensor_matrix_product_eval_diag`, the dot product with `dot`.

2.4 `TreeBasedTensor` and `DimensionTree`

We present in this section the `DimensionTree` and `TreeBasedTensor` objects. For examples of use, see the tutorial file `tutorials\tensor_algebra\tutorial_DimensionTree.py` and `tutorials\tensor_algebra\tutorial_TreeBasedTensor.py`.

2.4.1 `DimensionTree`

A dimension tree T is a collection of non-empty subsets of $D = \{1, \dots, d\}$ which is such that (i) all nodes $\alpha \in T$ are non-empty subsets of D, (ii) D is the root of T, (iii) every node $\alpha \in T$ with $\#\alpha \geq 2$ has at least two children and the set of children of α, denoted by $S(\alpha)$, is a non-trivial partition of α, and (iv) every node α with $\#\alpha = 1$ has no child and is called a leaf (see for example Figure 1).
We let $\text{depth}(T) = \max_{\alpha \in T} \text{level}(\alpha)$ be the depth of T, and $\mathcal{L}(T)$ be the set of leaves of T, which are such that $S(\alpha) = \emptyset$ for all $\alpha \in \mathcal{L}(T)$.



(a) A trivial tree. (b) A linear binary tree. (c) A balanced binary tree.

Figure 1: Examples of dimension partition trees over $D = \{1, \dots, 4\}$.

Creating a DimensionTree. A `DimensionTree` is characterized by its adjacency matrix and the dimension associated with each leaf node: `T = tensap.DimensionTree(dims, adjacency_matrix)`. The adjacency matrix of a dimension tree T can be accessed with `T.adjacency_matrix`. The dimension associated with each leaf node can be accessed with `T.dim2ind`.

Denoting by `order` the number of leaf nodes, it is possible to create

- a trivial tree with `tensap.DimensionTree.trivial(order)` (Figure 1a),
- a linear tree with `tensap.DimensionTree.linear(order)` (Figure 1b),
- a balanced tree with `tensap.DimensionTree.balanced(order)`(Figure 1c),
- a random tree with `tensap.DimensionTree.random(order, arity)`, with `arity` the arity of the tree, equal to the maximum number of children per node (randomly selected in an interval if provided).

Finally, a dimension tree can be created by extracting a sub-tree from an existing tree T with `T.sub_dimension_tree(root)` where `root` is the node in T that will become the root node of the extracted tree.

Displaying a DimensionTree. A `DimensionTree` can be displayed with the command `T.plot()`. The dimension associated with each leaf node can be plotted on the tree with `T.plot_dims()`. Finally, the tree can be plotted with some quantity displayed at each node with `T.plot_with_labels_at_nodes(labels)`.

Accessing properties of the tree. The number of nodes of a dimension tree T is given by `T.nb_nodes`.

The parent of α, denoted by $P(\alpha)$, can be obtained with `T.parent(alpha)`, and its ascendants $A(\alpha)$ and descendants $D(\alpha)$ by `T.ascendants(alpha)` and `T.descendants(alpha)`, respectively. The children of α are given by `T.children(alpha)`. The command `T.child_number(alpha)` returns i_α^γ, for $\alpha \in T \setminus \{D\}$ and $\gamma = P(\alpha)$, which is such that α is the i_α^γ-th child of γ. For instance, in the tree of Figure 1b, the node $\alpha = \{3, 4\}$ is the second child of $\gamma = \{2, 3, 4\}$.

The level of a node α is denoted by level(α). The levels are defined such that level(D) = 0 and level(β) = level(α) + 1 for $\beta \in S(\alpha)$. The nodes of T with level l are returned by `T.nodes_with_level(l)`.

The leaf nodes $\alpha \in \mathcal{L}(T)$ are such that `T.is_leaf[alpha-1]` is `True`.

2.4.2 `TreeBasedTensor`

Given a dimension tree T, a `TreeBasedTensor X` is a tensor in *tree-based format* (see [3, 6]). It represents an order d tensor $X \in \mathbb{R}^{N_1 \times \cdots \times N_d}$ in the set of tensors with α-ranks bounded by some integer r_α, $\alpha \in T$. Such a tensor admits a representation

$$X_{i_1,\ldots,i_d} = \sum_{\substack{1 \le k_\beta \le r_\beta \\ \beta \in T \setminus \{D\}}} \prod_{\alpha \in T \setminus \mathcal{L}(T)} C^\alpha_{(k_\beta)_{\beta \in S(\alpha)}, k_\alpha} \prod_{\alpha \in \mathcal{L}(T)} C^\alpha_{i_\alpha, k_\alpha},$$

with C^α, $\alpha \in T$, some tensors that parameterize the representation of X. When T is a binary tree, the corresponding format is the so-called hierarchical Tucker (HT) format. The particular case of a linear binary tree is the tensor train Tucker format.

The *Tucker format* corresponds to a trivial tree $T = \{\{1\}, \ldots, \{d\}, \{1, \ldots, d\}\}$ and admits the representation

$$X_{i_1,\ldots,i_d} = \sum_{k_1=1}^{r_1} \cdots \sum_{k_d=1}^{r_d} C^{1,\ldots,d}_{k_1,\ldots,k_d} C^1_{i_1,k_1} \cdots C^d_{i_d,k_d}.$$

A *degenerate tree-based format* is defined as the set of tensors with α-ranks bounded by some integer r_α, for all α in a subset A of T. The set A corresponds to active nodes, which should contain all interior nodes $T \setminus \mathcal{L}(T)$. A `TreeBasedTensor X` with active nodes A admits a representation.

$$X_{i_1,\ldots,i_d} = \sum_{\substack{1 \le k_\beta \le r_\beta \\ \beta \in A \setminus \{D\}}} \prod_{\alpha \in A \setminus \mathcal{L}(T)} C^\alpha_{(k_\beta)_{\beta \in S(\alpha)}, k_\alpha} \prod_{\alpha \in \mathcal{L}(T) \cap A} C^\alpha_{i_\alpha, k_\alpha},$$

with C^α, $\alpha \in A$, some tensors that parameterize the representation of X.

The *tensor train format* is a degenerate tree-based format with a linear tree T and all leaf nodes inactive except the first one, that means $A = \{\{1\}, \{1, 2\}, \ldots, \{1, \ldots, d\}\}$. A tensor X in tensor train format admits a representation

$$X_{i_1,\dots,i_d} = \sum_{k_1=1}^{r_1} \dots \sum_{k_{d-1}=1}^{r_{d-1}} C^1_{1,i_1,k_1} C^2_{k_1,i_2,k_2} \dots C^{d-1}_{k_{d-2},i_{d-1},k_{d-1}} C^d_{k_{d-1},i_d,1}$$

with tensor C^ν and rank r_ν associated with the node $\alpha = \{1, \dots, \nu\}$.

For a more detailed presentation of tree-based formats (possibly degenerate) and more examples, see [9, Section 4].

If the rank r_D associated with the root node is different from 1, a `TreeBasedTensor X` represents a tensor of order $d+1$ with entries $X_{i_1,\dots,i_d,k_D}$, $1 \leq k_D \leq r_D$. I can be used to defined vector-valued functional tensors (see Section 3.7).

Creating a TreeBasedTensor.

A `TreeBasedTensor` is created with the command `X = tensap.TreeBasedTensor(C, T)`, with `C` the list of `FullTensor` objects representing the C^α, $\alpha \in T$, and `T` a `DimensionTree`. If some entries of the list `C` corresponding to leaf nodes are empty, it creates a degenerate tensor format, with $T \setminus A$ corresponding to the empty entries of `C`.

It is possible to create a `TreeBasedTensor` in tensor-train format with the command `tensap.TreeBasedTensor.tensor_train(C)`, with `C` a list containing the tensors $C^1, \dots, C^d$.

Given a `DimensionTree T`, it is also possible to generate a `TreeBasedTensor` with entries

- equal to 0 with `tensap.TreeBasedTensor.zeros(T, r, s, I)`,
- equal to 1 with `tensap.TreeBasedTensor.ones(T, r, s, I)`,
- drawn randomly according to the uniform distribution on $[0, 1]$ with `tensap.TreeBasedTensor.rand(T, r, s, I)`,
- drawn randomly according to the standard gaussian distribution with `tensap.TreeBasedTensor.randn(T, r, s, I)`,
- generated using a provided `generator` with `tensap.TreeBasedTensor.create (generator, T, r, s, I)`,

where `r` is a list containing the α-ranks, $\alpha \in T$, or `'random'`, `s` is a list containing the sizes $N_1, \dots, N_d$, or `'random'`, and `I` is a list of booleans indicating if the node α is active, $\alpha \in T$, or `'random'`.

Storage complexity. The storage complexity of `X` is given by `X.size = X.storage()` and returns the number of entries in tensors C^α, $\alpha \in A$.
The storage complexity of `X` taking into account the sparsity in the C^α, $\alpha \in T$, is given by `X.sparse_storage()`. It returns the number of non-zero entries in tensors C^α, $\alpha \in A$.
The storage complexity of `X` taking into account the sparsity only in the leaf nodes is given by `X.sparse_leaves_storage()`.

Displaying a TreeBasedTensor. A graphical representation of a `TreeBasedTensor X` can be obtained with the command `X.plot()`. Labels can be added to the nodes of the tree, as well as a title, with `X.plot(labels, title)`.

Converting a TreeBasedTensor to a FullTensor. A `TreeBasedTensor X` can be converted to a `FullTensor` (introduced in Section 1) with the command `X.full()`.

Converting a FullTensor to a TreeBasedTensor. A `FullTensor X` can be converted to a `TreeBasedTensor` with the command `X.tree_based_tensor()`. The associated dimension tree is a trivial tree with active nodes.

Accessing the entries of a TreeBasedTensor. The entries of the tensor `X` can be accessed with the method `eval_at_indices`: `X.eval_at_indices(ind)` returns the entries of X indexed by the list `ind` containing the indices to access in each dimension.

A sub-tensor can be extracted from `X` with the method `sub_tensor` (see Section 1.2)

For a tensor $X \in \mathbb{R}^{N,\dots,N}$, the command `X.eval_diag()` returns the diagonal $X_{i,\dots,i}$, $i = 1,\dots,N$, of the tensor. The method `eval_diag` can also be used to evaluate the diagonal in some dimensions `dims` of the tensor with `X.eval_diag(dims)`.

Obtaining an orthonormal representation of a TreeBasedTensor. The command `X.orth()` returns a representation of `X` where all the core tensors except the root core represent orthonormal bases of principal subspaces.

The command `X.orth_at_node(alpha)` returns a representation of `X` where all the core tensors except the one of node α represent orthonormal bases of principal subspaces. The core tensor C^α of the node α is such that the tensor writes

$$X_{i_\alpha,i_{\alpha^c}} = \sum_k \sum_l C^\alpha_{k,l} u_l(i_\alpha) w_k(i_{\alpha^c}),$$

where the u_l are orthonormal tensors and the w_k are orthonormal tensors. This orthonormality of the representation can be checked by computing the Gram matrices of the bases of minimal subspaces associated with the nodes of the tree with `X.gramians()`.

Modifying the tree structure of a TreeBasedTensor. It is possible to modify the tree of a `TreeBasedTensor X` by permuting two of its nodes α and β given a relative tolerance `tol` with `X.permute_nodes([alpha, beta], tol)`.

The leaves of the tree can also be permuted with the command `X.permute_leaves(perm, tol)`, where `perm` is a permutation of $(1, \dots, d)$.

The method `optimize_dimension_tree` tries random permutations of nodes to minimize the storage complexity of a tree-based tensor X: `X.optimize_dimension_tree(tol, n)` tries n random permutations and returns a `TreeBasedTensor Y` which is such that `Y.storage()` is less or equal than `X.storage()`. The nodes to permute are drawn according to probability measures favoring high decreases of the ranks while maintaining a permutation cost as low as possible (see [5, Section 4.2.1]).

The similar method `optimize_leaves_permutations` focuses on the permutation of the leaf nodes to try to reduce the storage complexity of a `TreeBasedTensor`.

Computing the Frobenius norm of a TreeBasedTensor. The command `X.norm()` returns the Frobenius norm of X.

Computing the α-singular values of a TreeBasedTensor. For all $\alpha \in T$, the α-singular values of X can be obtained with `X.singular_values()`.

The method `rank` uses the method `singular_values` to compute the α-ranks, $\alpha \in T$, of a `TreeBasedTensor`.

Computing the derivative of TreeBasedTensor with respect to one of its parameters. For an order-d tree-based tensor `X` in $\mathbb{R}^{N\times\cdots\times N}$, `X.parameter_gradient_eval_diag(alpha)`, for $\alpha \in T$, returns the derivative

$$\left.\frac{\partial X_{i_1,\dots,i_d}}{\partial C^\alpha}\right|_{i_1=\cdots=i_d=i}, \ i = 1, \dots, N.$$

The method `parameter_gradient_eval_diag` is used in the statistical learning algorithms presented in Section 5.4.

Performing operations with TreeBasedTensor. Some operations between tensors are implemented for `TreeBasedTensor` (see Section 1.6 for a detailed description of the operations): the Kronecker product with `kron`, the contraction with matrices or vectors with `tensor_matrix_product` or `tensor_vector_product` respectively, the evaluation of the diagonal of a contraction with matrices with `tensor_matrix_product_eval_diag`, the dot product with `dot`.

`Z = X.tensor_matrix_product(M)` `tensor_vector_product` `tensor_matrix_product_eval_diag` `X.kron(Y)` `X.dot(Y)`

2.5 Tensor truncation with `Truncator`

The object `Truncator` embeds several methods of truncation of tensors in different formats. Given a tolerance `tol` and a maximum rank or tuple of ranks `r`, a `Truncator` object can be created with `t = tensap.Truncator(tol, r)`. The thresholding type (`'hard'` or `'soft'`) can also be specified as a third argument.

For examples of use, see the tutorial file `tutorials\tensor_algebra\tutorial_tensor_truncation.py`.

Truncation. The generic method `truncate` calls one of the methods presented below, based on the type and order of its input, to obtain a truncation of the provided tensor satisfying the relative prevision and maximal rank requirements.

For an order 2 tensor, the method `svd` is called. For a tensor of order greater than 2, the method `hosvd` is called for a `FullTensor`, and `hsvd` for a `TreeBasedTensor`.

Truncated singular value decomposition. The method `svd` computes the truncated singular value decomposition of an order 2 tensor. The input tensor can be a `numpy.ndarray`, a `tensorflow.Tensor`, a `FullTensor` or a `CanonicalTensor`, in which case the method `trunc_svd` is called, or a `TreeBasedTensor`, in which case the method `hsvd` is called.

The method `trunc_svd` computes the truncated singular value decomposition of a matrix, with a given relative precision in Schatten p-norm (with a specified value for p) and given maximal rank. The returned truncation is a `CanonicalTensor`.

Truncated higher-order singular value decomposition. A truncated higher-order singular value decomposition of a `numpy.ndarray`, a `FullTensor` or a `TreeBasedTensor` can be computed with the method `hosvd`. The output is either a `CanonicalTensor` for an order 2 tensor, or a `TreeBasedTensor` with a trivial tree for a tensor of order greater than 2.

Truncation in tree-based tensor format. The method `hsvd` computes, given a `TreeBasedTensor` or a `FullTensor` with a tree and a set of active nodes, a truncation in tree-based tensor format.

Truncation in tensor train format. The method `ttsvd`, given a `FullTensor`, calls the method `hsvd` with a linear tree and all the leaf nodes inactive except the first one, resulting in a truncation in tensor-train format.

3 Measures, bases and functions

3.1 `RandomVariable`

A random variable X can be created by calling its name: for instance, `X = tensap.UniformRandomVariable(a, b)` creates a random variable with a uniform distribution on the interval $[a, b]$. The random variables currently implemented in tensap are:

- `tensap.DiscreteRandomVariable(v, p)`: a random variable with discrete values v and associated probabilities p,
- `tensap.UniformRandomVariable(a, b)`: a uniform random variable on $[a, b]$,
- `tensap.NormalRandomVariable(m, s)`: a normal random variable with mean m and standard deviation s,
- `tensap.EmpiricalRandomVariable(S)`: a random variable created from a sample S using kernel density estimation with Scott's rule of thumb to determine the bandwidth.

A new random variable can easily be implemented in tensap by making its class inheriting from `RandomVariable` and implementing the few methods necessary for its creation.

Once a random variable X is created, one can for instance generate n random numbers according to its distribution with `X.random(n)`, create the orthonormal polynomials associated with its measure with `X.orthonormal_polynomials()` (as presented in Section 3.3), or evaluate its probability density function (`X.pdf(x)`), cumulative distribution function (`X.cdf(x)`) or inverse cumulative distribution function (`X.icdf(x)`).

3.2 `RandomVector`

A random vector X if defined in tensap by a list of `RandomVariable` objects and a `Copula`, characterizing the dependencies between the random variables. Currently, only the independent copula `IndependentCopula` is implemented.

Given a list of `RandomVariable` `random_variables` and a `Copula C`, a random vector can be created with `X = tensap.RandomVector(random_variables, copula=C)`.

Once a random vector X is created, one can for instance generate n random numbers according to its distribution with `X.random(n)`, create the orthonormal polynomials associated with its measure with `X.orthonormal_polynomials()` (as presented in Section 3.3), or evaluate its probability density function (`X.pdf(x)`) or cumulative distribution function (`X.cdf(x)`).

3.3 `Polynomials`

Families of univariate polynomials $(p_i)_{i\geq 0}$ are represented in tensap with classes inheriting from `UnivariatePolynomials`. The i-th polynomial p_i represented by a `UnivariatePolynomials` object `P` can be evaluated with `P.polyval(x, i)`, as well as its first order derivative (`P.d_polyval(x, i)`) and its n-th order derivative (`P.dn_polyval(x, n, i)`).

Given a measure μ, the moments $\int p_{i_1}(x)...p_{i_k}(x)d\mu(x)$ for $(i_1, ..., i_k) \in \mathbb{N}^k$ can be obtained with `P.moment(I, mu)`, with `I` a n-by-k array representing n tuples $(i_1, ..., i_k)$. `P.moment(I, X)` with `X` a random variable considers for μ the probability distribution of X.

`CanonicalPolynomials.` The family of canonical polynomials is implemented in the class `CanonicalPolynomials`. It is such that its i-th polynomial is $p_i(x) = x^i$.

`OrthonormalPolynomials.` Orthonormal polynomials are families of polynomials $(p_i)_{i\geq 0}$ that satisfy

$$\int p_i(x)p_j(x)d\mu(x) = \delta_{ij}$$

with δ_{ij} the Kronecker delta, and with μ some measure.

In tensap, the orthonormal polynomials p_i, $i \geq 0$, are defined using the three-term recurrence relation

$$\begin{aligned}
&\tilde{p}_{-1}(x) = 0, \quad \tilde{p}_0(x) = 1,\\
&\tilde{p}_{i+1}(x) = (x - a_i)\tilde{p}_i(x) - b_i\tilde{p}_{i-1}(x), \quad i \geq 0,\\
&p_i(x) = \frac{\tilde{p}_i(x)}{n_i}, \quad i \geq 0
\end{aligned}$$

with a_i and b_i the recurrence coefficients, and n_i the norm of $\tilde{p}_i$, defined by

$$a_i = \frac{\int \tilde{p}_i(x)x\tilde{p}_i(x)d\mu(x)}{\int p_i(x)\tilde{p}_i(x)d\mu(x)}, \quad b_i = \frac{\tilde{p}_i(x)\tilde{p}_i(x)d\mu(x)}{\int \tilde{p}_{i-1}(x)\tilde{p}_{i-1}(x)d\mu(x)}, \quad n_i = \sqrt{\int \tilde{p}_i(x)\tilde{p}_i(x)d\mu(x)}.$$

Implementing a new family of orthonormal polynomials in tensap is easy: one only needs to create a class with a method providing the recurrence coefficients a_i, b_i and the norms n_i, $\forall i \geq 0$.

Are currently implemented in tensap:

- `DiscretePolynomials`: discrete polynomials orthonormal with respect to the measure of a `DiscreteRandomVariable`;

- `LegendrePolynomials`: polynomials defined on $[-1,1]$ and orthonormal with respect to the uniform measure on $[-1,1]$ with density $\frac{1}{2}\mathbf{1}_{[-1,1]}(x)$;
- `HermitePolynomials`: polynomials defined on $\mathbb{R}$ and orthonormal with respect to the standard gaussian measure with density $\exp(-x^2/2)/\sqrt{2\pi}$;
- `EmpiricalPolynomials`: polynomials orthonormal with respect to the measure of an `EmpiricalRandomVariable`.

If `mu` is a `LebesgueMeasure` on $[-1,1]$, `mu.orthonormal_polynomials()` returns a `LegendrePolynomials` with suitably normalized coefficients. If `mu` is a `LebesgueMeasure` on $[a,b]$ different from $[-1,1]$, `mu.orthonormal_polynomials()` returns a `ShiftedOrthonormalPolynomials`.

If `X` is a `DiscreteRandomVariable`, a `UniformRandomVariable`, a `NormalRandomVariable`, or a `EmpiricalRandomVariable`, the corresponding family of orthonormal polynomials can be created with the command `X.orthonormal_polynomials()`. If `X` does not correspond to a default measure but can be obtained as the push-forward measure of a default measure by an affine transformation (e.g. a uniform measure on $[a,b] \neq [-1,1]$, or a gaussian measure with mean a and standard deviation σ with $(a,\sigma) \neq (0,1)$.), the returned object is a `ShiftedOrthonormalPolynomials`.

3.4 `FunctionalBasis`

Bases of functions can be implemented in tensap by inheriting from `FunctionalBasis`. The basis functions of a `FunctionalBasis` object `H` can be evaluated with `H.eval(x)`, as well as their i-th order derivative with `H.eval_derivative(i, x)`.

We present below some specific bases implemented in tensap. New bases can easily be implemented by making their class inherit from `FunctionalBasis`.

`PolynomialFunctionalBasis.` The command `tensap.PolynomialFunctionalBasis (basis, indices)`, with `basis` a `UnivariatePolynomials` and `indices` a list, returns the basis of polynomials $(p_i)_{i \in I}$ with I given by `indices`.

`UserDefinedFunctionalBasis.` Given a list of functions `fun`, taking each as inputs d variables, and a `Measure mu`, the command `tensap.UserDefinedFunctionalBases(fun, mu, d)` returns a basis whose functions are the ones given in `fun`, with a domain equipped with the measure *mu*.

FullTensorProductFunctionalBasis. A `FullTensorProductFunctionalBasis` object represents a basis of multivariate functions $\{\phi^1_{i_1}(x_1)\cdots\phi^d_{i_d}(x_d)\}_{i_1\in I^1,\ldots,i_d\in I^d}$. It is obtained with the command `tensap.FullTensorProductFunctionalBasis(bases)`, where `bases` is a list of `FunctionalBasis` or a `FunctionalBases`, containing the different bases $\{\phi^\nu_{i_\nu}\}_{i_\nu\in I^\nu}$, $\nu = 1,\ldots,d$.

SparseTensorProductFunctionalBasis. A `SparseTensorProductFunctionalBasis` object represents a basis of multivariate functions $\{\phi^1_{i_1}(x_1)\cdots\phi^d_{i_d}(x_d)\}_{(i_1,\ldots,i_d)\in\Lambda}$, with $\Lambda \subset I^1\times\cdots\times I^d$ a set of multi-indices. It is obtained with the command `tensap.SparseTensorProductFunctionalBasis(bases, indices)`, where `bases` is a list of `FunctionalBasis` or a `FunctionalBases`, containing the different bases $\{\phi^\nu_{i_\nu}\}_{i_\nu\in I^\nu}$, $\nu = 1,\ldots,d$, and `indices` is a `MultiIndices` representing the set of multi-indices Λ.

3.5 FunctionalBases

The command `tensap.FunctionalBases(bases)`, with `bases` a list of `FunctionalBasis`, returns an object representing a collections of bases. To obtain a collection of d identical bases, one can use `tensap.FunctionalBases.duplicate(basis, d)`.

Similarly to `FunctionalBasis`, the basis functions of a `FunctionalBases` object `H` can be evaluated with `H.eval(x)`, as well as their i-th order derivative with `H.eval_derivative(i, x)`.

3.6 FunctionalBasisArray

Given a basis of functions $\{\phi_i\}_{i\in I}$, a `FunctionalBasisArray` object represents a function f that writes

$$f(x) = \sum_{i\in I} a_i\phi_i(x),$$

with some coefficients a_i, $i \in I$, and can be created with the command `f = tensap.FunctionalBasisArray(a, basis, shape)`, with `shape` the output shape of f.

A `FunctionalBasisArray` is a `Function`. It can be evaluated with the command `f.eval(x)`, and one can obtain its derivatives with `f.eval_derivative(n, x)`.

3.7 FunctionalTensor

Given d bases of functions $\{\phi^\nu_{i_\nu}\}_{i_\nu\in I^\nu}$, $\nu = 1,\ldots,d$, and a tensor $a \in \mathbb{R}^{I^1\times\cdots\times I^d}$, a `FunctionalTensor` object represents a function f that writes

$$f(x) = \sum_{i_1\in I^1}\cdots\sum_{i_d\in I^d} a_{i_1,\ldots,i_d}\phi^1_{i_1}(x_1)\cdots\phi^d_{i_d}(x_d).$$

The tensor a can be in different tensor formats (`FullTensor`, `TreeBasedTensor`, ...).

A `FunctionalTensor` is a `Function`. It can be evaluated with the command `f.eval(x)`, and one can obtain its derivatives with `f.eval_derivative(n, x)`.

3.8 `Tensorizer` and `TensorizedFunction`

For an introduction to tensorization of functions, see [1, 2].

We consider functions defined on the interval $I = [0, 1)$. For a given $b \in \{2, 3, \dots, \}$ and $d \in \mathbb{N}$, an element $x \in I$ can be identified with the tuple $(i_1, \dots, i_d, y)$, such that

$$x = t_{b,d}(i_1, \dots, i_d, y) = \sum_{k=1}^{d} i_k b^{-k} + b^{-d} y \tag{2}$$

with $i_k \in I_b = \{0, \dots, b-1\}$, $k = 1, \dots, d$, and $y = b^d x - \lfloor b^d x \rfloor \in [0, 1)$. The tuple $(i_1, \dots, i_d)$ is the representation in base b of $\lfloor b^d x \rfloor$. This defines a bijective map $t_{b,d}$ from $\{0, \dots, b-1\}^d \times [0, 1)$ to $[0, 1)$.

Such a mapping is represented in tensap by the object `Tensorizer`: `t = tensap.Tensorizer(b, d)`. For a given x in $[0, 1)$, on obtains the corresponding tuple $(i_1, ..., i_d, y)$ with the command `= t.map(x)`. For a given tuple $(i_1, ..., i_d, y)$, on obtains the corresponding x with `t.inverse_map([i_1, ..., i_d,y])`.

This identification is generalized to functions of D variables with `t = tensap.Tensorizer(b, d, D)`.

The map $t_{b,d}$ allows to define a tensorization map $T_{b,d}$, which associates to a univariate function F defined on $[0, 1)$ the multivariate function $f = F \circ t_{b,d}$ defined on $I_b^d \times I$, such that

$$f(i_1, \dots, i_d, y) = F(t_{b,d}(i_1, \dots, i_d, y)).$$

Such a function is represented in tensap by a `TensorizedFunction`, and can be created with `f = tensap.TensorizedFunction(fun, t)`, with `fun` a `function` or `Function` and `t` a `Tensorizer`. The `TensorizedFunction` `f` is a function of $d+1$ variables that can be evaluated with `f.eval(x)`, with `x` a list or `numpy.ndarray` with $d+1$ columns.

See the tutorial file `tutorials\functions\tutorial_TensorizedFunction.py`.

4 Tools

4.1 `MultiIndices`

A multi-index is a tuple $(i_1, \dots, i_d) \in \mathbb{N}_0^d$. A set $I \subset \mathbb{N}_0^d$ of multi-indices is represented with an object `MultiIndices`.

To create a multi-index set I, we use the command `tensap.MultiIndices(I)` with `I` a numpy array of size $\#I \times d$.

A product set $I = I_1 \times \ldots \times I_d$ can be obtained with `tensap.MultiIndices.product_set([I1,...,Id])`.

The set of multi-indices

$$I = \{i \in \mathbb{N}_0^d : \|i\|_{\ell^p} \leq m\}$$

can be obtained with `tensap.MultiIndices.with_bounded_norm(d, p, m)`

The set of multi-indices

$$I = \{i \in \mathbb{N}_0^d : i_\nu \leq m_\nu, 1 \leq \nu \leq d\}$$

can be obtained with `tensap.MultiIndices.bounded_by(d, p, m)`. If m is of length 1, it uses $m_\nu = m$ for all ν.

For obtaining the margin or reduced margin of an multi-index set I, we can use

For other operations of `MultiIndices`, see the tutorial file `tutorials\tools\tutorial_MultiIndices.py`.

4.2 `TensorGrid`, `FullTensorGrid` and `SparseTensorGrid`

Tensor product grids or sparse grids are represented with classes `FullTensorGrid` and `SparseTensorGrid`, that inherit from `TensorGrid`.

See the tutorial file `tutorials\functions\tutorial_functions_bases_grids.py`.

5 Learning

We present in this section some objects implemented in tensap for learning functions or tensors.

5.1 `(Functional)TensorPrincipalComponentAnalysis`

The objects `TensorPrincipalComponentAnalysis` (resp. `FunctionalTensorPrincipalComponentAnalysis`) implements approximation methods for algebraic (resp. functional) tensors based on principal component analysis, using an adaptive sampling of the entries of the tensor (or the function). See [9] for a description of the algorithms, and for examples of use, see the tutorial files `tutorials\approximation\tutorial_TensorPrincipalComponentAnalysis.py` and `tutorials\approximation\tutorial_FunctionalTensorPrincipalComponentAnalysis.py`.

The difference between the two objects if that `TensorPrincipalComponentAnalysis`' methods take as first input a function returning components of the algebraic tensor to learn, whereas the methods of `FunctionalTensorPrincipalComponentAnalysis` take as first input the functional tensor to learn.

Both objects are parameterized by the attributes:

- `pca_sampling_factor`: a factor to determine the number of samples N for the estimation of the principal components (1 by default): if the precision is prescribed, $N =$ `pca_sampling_factor` $\times N_\alpha$, if the rank is prescribed, $N =$ `pca_sampling_factor` $\times t$;
- `pca_adaptive_sampling`: a boolean indicating if adaptive sampling is used to determine the principal components with prescribed precision;
- `tol`: an array containing the prescribed relative precision; set `tol = inf` for prescribing the rank;
- `max_rank`: an array containing the maximum alpha-ranks (the length depends on the format). If `len(max_rank) == 1`, uses the same value for all alpha; setting `max_rank = inf` prescribes the precision.

Furthermore, a `FunctionalTensorPrincipalComponentAnalysis` is parameterized by the attributes:

- `bases`: the functional bases used for the projection of the function;
- `grid`: the `FullTensorGrid` used for the projection of the function on the functional bases;
- `projection_type`: the type of projection, the default being 'interpolation'.

Both objects implement four main methods:

- `hopca`: returns the set of $\{\nu\}$-principal components of an order d tensor, for all $\nu \in \{1, \ldots, d\}$;
- `tucker_approximation`: returns an approximation of a tensor of order d or a function of d variables in Tucker format;
- `tree_based_approximation`: provided with a tree and a list of active nodes, returns an approximation of a tensor of order d or a function of d variables in tree-based tensor format;
- `tt_approximation`: returns an approximation of a tensor of order d or a function of d variables in tensor-train format.

5.2 `LossFunction`

In tensap, a loss function is an object inheriting from `LossFunction`. Given a function `fun` and a sample as a list used to evaluate the loss function, a `LossFunction` object ℓ can be evaluated with `l.eval(fun, sample)`. The risk associated with `fun` can be evaluated using the sample with `l.risk_estimation(fun, sample)`. Finally, the test error and relative test error (if defined) can be evaluated with `l.test_error(fun, sample)` and `l.relative_test_error(fun, sample)`, respectively.

Currently, three loss functions are implemented in tensap:

- `SquareLossFunction`: $\ell(g,(x,y)) = (y - g(x))^2$, used for least-squares regression in supervised learning, to construct an approximation of a random variable Y as a function of a random vector X (a predictive model);
- `DensityL2LossFunction`: $\ell(g,x) = \|g\|^2 - 2g(x)$, used for least-squares density estimation, to approximate the distribution of a random variable X from samples of X;
- `CustomLossFunction`: defined by the user as any function defining a loss. If the loss is defined using tensorflow operations, then the empirical risk can be minimized using tensorflow's automatic differentiation capability with a `LinearModelLearningCustomLoss` object, presented in the next section.

5.3 `LinearModelLearning`

Objects inheriting from `LinearModelLearning` implement the empirical risk minimization associated with a linear model that writes

$$g(x) = \sum_{i\in I} a_i\phi_i(x),$$

with $\{\phi_i\}_{i\in I}$ a given basis (or a set of features) and $(a_i)_{i\in I}$ some coefficients, and a loss function, introduced in the previous section.

In order to perform empirical risk minimization, a `LinearModelLearning` object `s` must be provided with a training sample in `s.training_sample`. In supervised learning, for the approximation of a random variable Y as a function of X, the training sample is a list `[x, y]`, with `y` represents n samples $\{y_k\}_{k=1}^n$ of Y and x the n corresponding samples $\{x_k\}_{k=1}^n$ of X. In density estimation, the training sample is an array `x` containing samples $\{x_k\}_{k=1}^n$ from the distribution to estimate.

One must also provide a basis (in `s.basis`) or evaluations of the basis on the training set (in `s.basis_eval`, in which case the x are not mandatory in `s.training_sample`). The latter option allows for providing features $\phi_i(x_k)$ associated with samples x^k, without providing the feature maps ϕ_i.

One can also provide the `LinearModelLearning` `s` with a test sample in `s.test_data` to compute a test error.

Currently in tensap, three different `LinearModelLearning` objects are implemented:

- `LinearModelLearningSquareLoss`, to minimize the risk associated with a `SquareLossFunction`;
- `LinearModelLearningDensityL2`, to minimize the risk associated with a `DensityL2LossFunction`;
- `LinearModelLearningCustomLoss`, to minimize the risk associated with a `CustomLossFunction`.

<u>`LinearModelLearningSquareLoss`.</u> A `LinearModelLearningSquareLoss` object `s` implements three ways of solving the empirical risk minimization associated with a `SquareLossFunction`:

- by default, `s.solve()` solves the ordinary least-squares problem
$$\min_{(a_i)_{i\in I}} \frac{1}{n}\sum_{k=1}^{n}(y_k - \sum_{i\in I} a_i\phi_i(x_k))^2;$$
- with the attribute `s.regularization = True`, `s.solve()` solves the regularized problem
$$\min_{(a_i)_{i\in I}} \frac{1}{n}\sum_{k=1}^{n}(y_k - \sum_{i\in I} a_i\phi_i(x_k))^2 + \lambda\|a\|_p$$
with λ a regularization hyper-parameter, selected with a cross-validation estimate of the error and p specified by `s.regularization_type` which can be `'l0'` ($p = 0$), `'l1'` ($p = 1$) or `'l2'` ($p = 2$);
- let us suppose that we have a collection of candidate sparsity patterns K_λ, $\lambda \in \Lambda$, for the parameter a: with the attribute `s.basis_adaptation = True`, `s.solve()` solves, for all $\lambda \in \Lambda$, the problem
$$\min_{(a_i)_{i\in I}} \frac{1}{n}\sum_{k=1}^{n}(y_k - \sum_{i\in I} a_i\phi_i(x_k))^2 \quad \text{subject to support}(a) \subset K_\lambda,$$
where $\text{support}(a) = \{k \in K : a_k \neq 0\}$, and selects the optimal sparsity pattern using a cross-validation estimate of the error.

<u>`LinearModelLearningDensityL2`.</u> A `LinearModelLearningDensityL2` object `s` implements three ways of solving the empirical risk minimization associated with a `DensityL2LossFunction`:

- by default, `s.solve()` solves the minimization problem

$$\min_{(a_i)_{i\in I}} \| \sum_{i\in I} a_i\phi_i\|_{L^2}^2 - \frac{2}{n}\sum_{k=1}^{n}\sum_{i\in I} a_i\phi_i(x_k);$$

- with the attribute `s.regularization = True`, `s.solve()` solves the constrained problem

$$\min_{(a_i)_{i\in I}} \| \sum_{i\in I} a_i\phi_i\|_{L^2}^2 - \frac{2}{n}\sum_{k=1}^{n}\sum_{i\in I} a_i\phi_i(x_k) \quad \text{subject to support}(a) \subset K_\lambda,$$

  with K_λ, $\lambda \in \Lambda$, a sequence of sets of indices that introduce the coefficients solution of the minimization problem without regularization in decreasing order of magnitude. The optimal sparsity pattern is determined using a cross-validation estimate of the error;

- let us suppose that we have a collection of candidate patterns K_λ, $\lambda \in \Lambda$, for the parameter a: with the attribute `s.basis_adaptation = True`, `s.solve()` solves, for all $\lambda \in \Lambda$, the problem

$$\min_{(a_i)_{i\in I}} \| \sum_{i\in I} a_i\phi_i\|^2 - \frac{2}{n}\sum_{k=1}^{n}\sum_{i\in I} a_i\phi_i(x_k) \quad \text{subject to support}(a) \subset K_\lambda,$$

  and selects the optimal sparsity pattern using a cross-validation estimate of the error.

<u>`LinearModelLearningCustomLoss.`</u> A `LinearModelLearningCustomLoss` object `s` implements a way of solving the empirical risk minimization associated with a `CustomLossFunction` using tensorflow's automatic differentiation capabilities.

By default, the optimizer used is keras' Adam algorithm, which is a "stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments" (per tensorflow's documentation).

The algorithm requires a starting point, provided in `s.initial_guess`, and several options can be set:

- `s.options['max_iter']` sets the maximum number of iterations used in the optimization algorithm,

- `s.options['stagnation']` sets the stopping tolerance on the stagnation between two iterates,

- for the Adam algorithm (and other minimization algorithms provided by tensorflow/keras), the learning rate can be provided in `s.optimizer.learning_rate`.

5.4 `TensorLearning`

The package tensap implements algorithms to perform statistical learning with canonical and tree-based tensor formats. See [4, 5, 7] for a detailed presentation of algorithms and related theory.

For examples, see the tutorial files `tutorials\approximation\tutorial_tensor_learning_CanonicalTen`
`tutorials\approximation\tutorial_tensor_learning_TreeBasedTensorLearning.py`,
`tutorials\approximation\tutorial_tensor_learning_TreeBasedTensorDensityLearning.py`,
`tutorials\approximation\tutorial_tensor_learning_tensorized_function_learning.py`.

These algorithms are implemented in the core object `TensorLearning`, common to all the tensor formats, so that implementing such a learning algorithm for a new tensor format is simple. In tensap are currently implemented `CanonicalTensorLearning` for the learning in canonical tensor format and `TreeBasedTensorLearning` for the learning in tree-based tensor format.

Two algorithms are proposed: the standard one, which minimizes an empirical risk over the set of tensors in a given format thanks to an alternating minimization over the parameters of the tensors, and the adaptive one, which returns a sequence of empirical risk minimizers with adapted rank (for the canonical and tree-based tensor formats) and adapted tree (for the tree-based tensor format).

In order to perform empirical risk minimization, a `TensorLearning` object `s` must be provided with a training sample in `s.training_sample`. In supervised learning, for the approximation of a random variable Y as a function of X, the training sample is a list `[x, y]`, with `y` represents n samples $\{y_k\}_{k=1}^n$ of Y and x the n corresponding samples $\{x_k = (x_{k,1}, \dots, x_{k,d})\}_{k=1}^n$ of X. In density estimation, the training sample is an array `x` containing samples $\{x_k = (x_{k,1}, \dots, x_{k,d})\}_{k=1}^n$ from the distribution to estimate.

One must also provide bases (in `s.bases`) or evaluations of the bases on the training set (in `s.bases_eval`, in which case the x are not mandatory in `s.training_sample`). The latter option allows for providing features $\phi_i^\nu(x_{\nu,k})$, $1 \leq \nu \leq d$, associated with samples $x_k = (x_{k,1}, \dots, x_{k,d})$, without providing the feature maps ϕ_i^ν.

One can also provide the `TensorLearning s` with a test sample in `s.test_data` to compute a test error.

Rank adaptation. (See [4, Section 4.1]) The rank adaptation is enabled by setting `s.rank_adaptation` to `True`.

For tensors in canonical format, the algorithm returns a sequence of rank-r approximations, with $r = 1, \dots, r_{\max}$, $r_{\max}$ being given by `s.rank_adaptation_options ['max_iterations']`.

For tensors in tree-based format, the algorithm returns a sequence of tensors with non-decreasing tree-based rank, obtained by increasing, at each iterations, the ranks associated with a subset of nodes of the tree T. The number of nodes in this subset is influenced by a parameter `s.rank_adaptation_options['theta']` in $[0, 1]$, which is such that the larger it is, the more ranks are increased at each iteration. The default value of 0.8.

Tree adaptation. (See [4, Section 4.2]) For tree-based tensor formats, the tree can be adapted at each iteration using the algorithm mentioned in Section 2.4, by setting `s.tree_adaptation` to `True`. The tolerance for the tree adaptation is provided by `s.tree_adaptation_options['tolerance']` and the maximal number of tried trees by `s.tree_adaptation_options['max_iterations']`.

Model selection. (See [7]) At the end of the adaptive procedure, a model can be selected by setting `s.model_selection` to `True`, using either a test error (specified by `s.model_selection_options['type'] = 'test_error'`) or a cross-validation estimate of the error (specified by `s.model_selection_options['type'] = 'cv_error'`).

5.5 Example: character classification in tree-based tensor format.

We present below a part of the tutorial file `tutorial\tensor\learning\digits\recognition.py` shipped with the package tensap. Its aim is to create a classifier in tree-based tensor format, able to recognize hand written digits from 0 to 9.

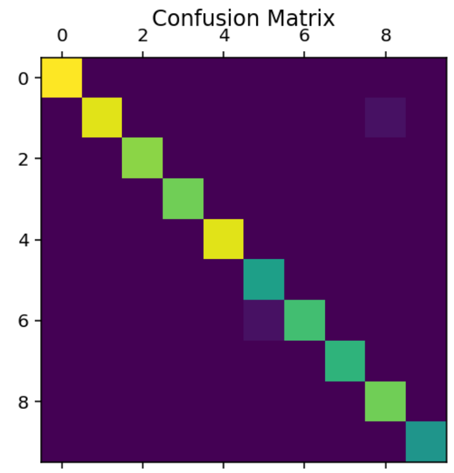
The output of the algorithm is displayed below the Python script, as well as in Figure 2, which shows the confusion matrix on the test sample as well as a visual comparison on some test samples. We see that, using a training sample of size 1617, it returns a classifier that obtains a score of 98.89% of correct classification on a test sample of size 180.

Tutorial file tutorial_tensor_learning_digits_recognition.py

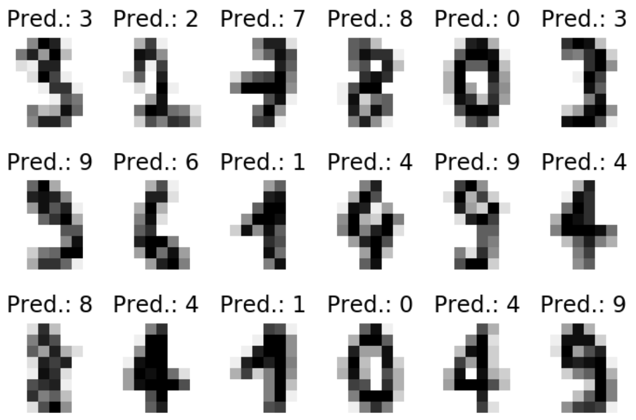
```python
from sklearn import datasets, metrics
import random
import numpy as np
import tensorflow as tf
import time
import matplotlib.pyplot as plt
import tensap

# %% Data import and preparation
DIGITS = datasets.load_digits()
DATA = DIGITS.images.reshape((len(DIGITS.images), -1))
DATA = DATA / np.max(DATA)  # Scaling of the data

# %% Patch reshape of the data: the patches are consecutive entries
    ↪ of the data
PS = [4, 4]  # Patch size
DATA = np.array([np.concatenate(
    [np.ravel(np.reshape(DATA[k, :], [8]*2)[PS[0]*i:PS[0]*i+PS[0],
                                                 PS[1]*j:PS[1]*j+PS[1]]) for
     i in range(int(8/PS[0])) for j in range(int(8/PS[1]))]) for
    k in range(DATA.shape[0])])
DIM = int(int(DATA.shape[1]/np.prod(PS)))

# %% Probability measure
print('Dimension %i' % DIM)
X = tensap.RandomVector(tensap.DiscreteRandomVariable(np.unique(DATA)), DIM)

# %% Training and test samples
P_TRAIN = 0.9  # Proportion of the sample used for the training

N = DATA.shape[0]
TRAIN = random.sample(range(N), int(np.round(P_TRAIN*N)))
TEST = np.setdiff1d(range(N), TRAIN)
X_TRAIN = DATA[TRAIN, :]
X_TEST = DATA[TEST, :]
Y_TRAIN = DIGITS.target[TRAIN]
Y_TEST = DIGITS.target[TEST]

# One hot encoding (vector-valued function)
Y_TRAIN = tf.one_hot(Y_TRAIN.astype(int), 10, dtype=tf.float64)
Y_TEST = tf.one_hot(Y_TEST.astype(int), 10, dtype=tf.float64)

# %% Approximation bases: 1, cos and sin for each pixel of the patch
FUN = [lambda x: np.ones((np.shape(x)[0], 1))]
for i in range(np.prod(PS)):
    FUN.append(lambda x, j=i: np.cos(np.pi / 2*x[:, j]))
    FUN.append(lambda x, j=i: np.sin(np.pi / 2*x[:, j]))

BASES = [tensap.UserDefinedFunctionalBasis(FUN, X.random_variables[0],
                                                np.prod(PS)) for _ in range(DIM)]
BASES = tensap.FunctionalBases(BASES)

# %% Loss function: cross-entropy custom loss function
LOSS = tensap.CustomLossFunction(
        lambda y_true, y_pred: tf.nn.sigmoid_cross_entropy_with_logits(
            logits=y_pred, labels=y_true))


def error_function(y_pred, sample):
    '''
    Return the error associated with a set of predictions using a sample, equal
    to the number of misclassifications divided by the number of samples.

    Parameters
    ----------
    y_pred : numpy.ndarray
    The predictions.
    sample : list
    The sample used to compute the error. sample[0] contains the inputs,
    and sample[1] the corresponding outputs.

    Returns
    -------
    int
    The error.

    '''
    try:
        y_pred = y_pred(sample[0])
    except Exception:
        pass
    return np.count_nonzero(np.argmax(y_pred, 1) - np.argmax(sample[1], 1)) / \
        sample[1].numpy().shape[0]


LOSS.error_function = error_function

# %% Learning in tree-based tensor format
TREE = tensap.DimensionTree.balanced(DIM)
IS_ACTIVE_NODE = np.full(TREE.nb_nodes, True)
SOLVER = tensap.TreeBasedTensorLearning(TREE, IS_ACTIVE_NODE, LOSS)

SOLVER.tolerance['on_stagnation'] = 1e-10
SOLVER.initialization_type = 'random'
SOLVER.bases = BASES
SOLVER.training_data = [X_TRAIN, Y_TRAIN]
SOLVER.test_error = True
SOLVER.test_data = [X_TEST, Y_TEST]

SOLVER.rank_adaptation = True
SOLVER.rank_adaptation_options['max_iterations'] = 15
SOLVER.model_selection = True
SOLVER.display = True

SOLVER.alternating_minimization_parameters['display'] = False
SOLVER.alternating_minimization_parameters['max_iterations'] = 10
SOLVER.alternating_minimization_parameters['stagnation'] = 1e-10

# Options dedicated to the LinearModelCustomLoss object
SOLVER.linear_model_learning.options['max_iterations'] = 10
SOLVER.linear_model_learning.options['stagnation'] = 1e-10
SOLVER.linear_model_learning.optimizer.learning_rate = 1e3

SOLVER.rank_adaptation_options['early_stopping'] = True
SOLVER.rank_adaptation_options['early_stopping_factor'] = 10

T0 = time.time()
F, OUTPUT = SOLVER.solve()
T1 = time.time()
print(T1-T0)

# %% Display of the results
F_X_TEST = np.argmax(F(X_TEST), 1)
Y_TEST_NP = np.argmax(Y_TEST.numpy(), 1)

print('\nAccuracy = %2.5e\n' % (1 - np.count_nonzero(F_X_TEST -
    ↪ Y_TEST_NP) /
                                Y_TEST_NP.shape[0]))

IMAGES_AND_PREDICTIONS = list(zip(DIGITS.images[TEST], F_X_TEST))
for i in np.arange(1, 19):
    plt.subplot(3, 6, i)
    plt.imshow(IMAGES_AND_PREDICTIONS[i][0],
               cmap=plt.cm.gray_r, interpolation='nearest')
    plt.axis('off')
    plt.title('Pred.: %i' % IMAGES_AND_PREDICTIONS[i][1])

print('Classification report:\n%s\n'
      % (metrics.classification_report(Y_TEST_NP, F_X_TEST)))
MATRIX = metrics.confusion_matrix(Y_TEST_NP, F_X_TEST)
plt.matshow(MATRIX)
plt.title('Confusion Matrix')
plt.show()
print('Confusion matrix:\n%s' % MATRIX)
```

Output of the algorithm

```
Dimension 4

The implemented learning algorithms are designed for orthonormal
   ↪ bases. These algorithms work with non-orthonormal bases, but
   ↪ without some guarantees on their results.


Rank adaptation, iteration 0:
    Enriched nodes: []
    Ranks = [10, 1, 1, 1, 1, 1, 1]
    Storage complexity = 144
    Test error = 9.38889e-01

Rank adaptation, iteration 1:
    Enriched nodes: [2, 4, 3, 5, 6, 7]
    Ranks = [10, 2, 2, 2, 2, 2, 2]
    Storage complexity = 320
    Test error = 8.44444e-01

Rank adaptation, iteration 2:
    Enriched nodes: [2, 3, 4, 5, 7]
    Ranks = [10, 3, 3, 3, 3, 2, 3]
    Storage complexity = 498
    Test error = 7.00000e-01

Rank adaptation, iteration 3:
    Enriched nodes: [2, 3, 4, 6, 7]
    Ranks = [10, 4, 4, 4, 3, 3, 4]
    Storage complexity = 718
    Test error = 5.61111e-01

Rank adaptation, iteration 4:
    Enriched nodes: [2, 5, 3]
    Ranks = [10, 5, 5, 4, 4, 3, 4]
    Storage complexity = 885
    Test error = 1.22222e-01

Rank adaptation, iteration 5:
    Enriched nodes: [2, 3, 4, 5, 6, 7]
    Ranks = [10, 6, 6, 5, 5, 4, 5]
    Storage complexity = 1257
    Test error = 5.55556e-02

Rank adaptation, iteration 6:
    Enriched nodes: [2, 3, 5, 6]
    Ranks = [10, 7, 7, 5, 6, 5, 5]
    Storage complexity = 1568
    Test error = 2.22222e-02

Rank adaptation, iteration 7:
    Enriched nodes: [2, 3]
    Ranks = [10, 8, 8, 5, 6, 5, 5]
    Storage complexity = 1773
    Test error = 3.33333e-02

Rank adaptation, iteration 8:
    Enriched nodes: [3, 7, 2, 6]
    Ranks = [10, 9, 9, 5, 6, 6, 6]
    Storage complexity = 2163
    Test error = 2.22222e-02

Rank adaptation, iteration 9:
    Enriched nodes: [4, 5]
    Ranks = [10, 9, 9, 6, 7, 6, 6]
    Storage complexity = 2337
    Test error = 2.22222e-02

Rank adaptation, iteration 10:
    Enriched nodes: [3, 4, 2, 6]
    Ranks = [10, 10, 10, 7, 7, 7, 6]
    Storage complexity = 2801
    Test error = 1.66667e-02

Rank adaptation, iteration 11:
    Enriched nodes: [2, 3, 5]
    Ranks = [10, 11, 11, 7, 8, 7, 6]
    Storage complexity = 3212
    Test error = 2.22222e-02

Rank adaptation, iteration 12:
    Enriched nodes: [2, 4, 6, 7, 3]
    Ranks = [10, 12, 12, 8, 8, 8, 7]
    Storage complexity = 3903
    Test error = 1.66667e-02

Rank adaptation, iteration 13:
    Enriched nodes: [2, 3, 4, 6, 7]
    Ranks = [10, 13, 13, 9, 8, 9, 8]
    Storage complexity = 4684
    Test error = 1.66667e-02

Rank adaptation, iteration 14:
    Enriched nodes: [5]
    Ranks = [10, 13, 13, 9, 9, 9, 8]
    Storage complexity = 4834
    Test error = 1.11111e-02

Model selection using the test error: model #14 selected
Ranks = [10, 13, 13, 9, 9, 9, 8], test error = 1.11111e-02
615.6790609359741

Accuracy = 9.88889e-01

Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        23
           1       1.00      0.96      0.98        23
           2       1.00      1.00      1.00        19
           3       1.00      1.00      1.00        18
           4       1.00      1.00      1.00        22
           5       0.93      1.00      0.96        13
           6       1.00      0.94      0.97        17
           7       1.00      1.00      1.00        15
           8       0.95      1.00      0.97        18
           9       1.00      1.00      1.00        12

    accuracy                           0.99       180
   macro avg       0.99      0.99      0.99       180
weighted avg       0.99      0.99      0.99       180


Confusion matrix:
[[23  0  0  0  0  0  0  0  0  0]
 [ 0 22  0  0  0  0  0  0  1  0]
 [ 0  0 19  0  0  0  0  0  0  0]
 [ 0  0  0 18  0  0  0  0  0  0]
 [ 0  0  0  0 22  0  0  0  0  0]
 [ 0  0  0  0  0 13  0  0  0  0]
 [ 0  0  0  0  0  1 16  0  0  0]
 [ 0  0  0  0  0  0  0 15  0  0]
 [ 0  0  0  0  0  0  0  0 18  0]
 [ 0  0  0  0  0  0  0  0  0 12]]
```



(a) Confusion matrix on the test sample.



(b) Comparison on some test samples: the prediction associated with each test sample image is displayed on top.

Figure 2: Obtained results for the classification tutorial.

References

[1] M. Ali and A. Nouy. Approximation with tensor networks. part I: Approximation spaces. *ArXiv*, abs/2007.00118, 2020.

[2] M. Ali and A. Nouy. Approximation with tensor networks. part II: Approximation rates for smoothness classes. *ArXiv*, abs/2007.00128, 2020.

[3] A. Falcó, W. Hackbusch, and A. Nouy. Tree-based tensor formats. *SeMA Journal*, Oct 2018.

[4] E. Grelier, A. Nouy, and M. Chevreuil. Learning with tree-based tensor formats. *arXiv e-prints*, page arXiv:1811.04455, November 2018.

[5] E. Grelier, A. Nouy, and R. Lebrun. Learning high-dimensional probability distributions using tree tensor networks. *arXiv preprint arXiv:1912.07913*, 2019.

[6] W. Hackbusch. *Tensor Spaces and Numerical Tensor Calculus*, volume 56. Springer Nature, 2019.

[7] B. Michel and A. Nouy. Learning with tree tensor networks: complexity estimates and model selection. *arXiv e-prints*, page arXiv:2007.01165, July 2020.

[8] A. Nouy. *Low-Rank Methods for High-Dimensional Approximation and Model Order Reduction*, chapter 4. SIAM, Philadelphia, PA, 2017.

[9] A. Nouy. Higher-order principal component analysis for the approximation of tensors in tree-based low-rank formats. *ArXiv e-prints*, 2017.