tntorch Documentation

Release 0.1

Rafael Ballester-Ripoll

Mar 20, 2019

Contents

This is a PyTorch-powered library for tensor modeling and learning that features transparent support for the tensor train (TT) model, CANDECOMP/PARAFAC (CP), the Tucker model, and more. Supported operations (CPU and GPU) include:

- Basic and fancy indexing of tensors, broadcasting, assignment, etc.
- Tensor decomposition and reconstruction
- Element-wise and tensor-tensor arithmetics
- Building tensors from black-box functions using cross-approximation
- Statistics and sensitivity analysis
- Optimization using autodifferentiation, useful for e.g. regression or classification
- Misc. operations on tensors: stacking, unfolding, sampling, derivating, etc.

CHAPTER 1

Get the Code

You can clone the project from tntorch's GitHub page:

```
git clone https://github.com/rballester/tntorch.git
```

or get it as a zip file.

CHAPTER 2

Installation

The main dependencies are NumPy and PyTorch (we recommend to install those with Conda or Miniconda). To install *tntorch*, run:

```
cd tntorch
pip install .
```

CHAPTER 3

First Steps

Some basic tensor manipulation:

```
import tntorch as tn

t = tn.ones(64, 64)  # 64 x 64 tensor, filled with ones
t = t[:, :, None] + 2*t[:, None, :]  # Singleton dimensions, broadcasting, and
↪arithmetics
print(tn.mean(t))  # Result: 3
```

Decomposing a tensor:

```
import tntorch as tn

data = ...  # A NumPy or PyTorch tensor
t1 = tn.Tensor(data, ranks_cp=5)  # A CP decomposition
t2 = tn.Tensor(data, ranks_tucker=5)  # A Tucker decomposition
t3 = tn.Tensor(data, ranks_tt=5)  # A tensor train decomposition
```

To get fully on board, check out the complete documentation:

3.1 Project Goals

This package was born to bring together some of the most popular tensor decomposition models (including CP, Tucker, and the tensor train) under a common interface. Thus, we use *one class* for all those models. They are all particular cases of tensor networks, and the idea is that decomposing, manipulating, and reconstructing tensors can be (to some extent) abstracted away from the particular decomposition format.

Building on top of PyTorch's flexibility and built-in automatic differentiation, the overall goal is to exploit those features and allow users to quickly develop, model, and fit various tensor decompositions in a range of data science applications.

3.2 API Documentation

3.2.1 anova

anova.**anova_decomposition**(*t*, *marginals=None*)

Compute an extended tensor that contains all terms of the ANOVA decomposition for a given tensor.

Reference: R. Ballester-Ripoll, E. G. Paredes, and R. Pajarola: "Sobol Tensor Trains for Global Sensitivity Analysis" (2017)

Parameters

- **t** – ND input tensor
- **marginals** – list of N vectors, each containing the PMF for each variable (use None for uniform distributions)

Returns a `Tensor`

anova.**dimension_distribution**(*t*, *mask=None*, *order=None*, *marginals=None*)

Computes the dimension distribution of an ND tensor.

Parameters

- **t** – ND input `Tensor`
- **mask** – an optional mask `Tensor` to restrict to
- **order** – int, compute only this many order contributions. By default, all N are returned
- **marginals** – PMFs for input variables. By default, uniform distributions

Returns a PyTorch vector containing N elements

anova.**mean_dimension**(*t*, *mask=None*, *marginals=None*)

Computes the mean dimension of a given tensor with given marginal distributions. This quantity measures how well the represented function can be expressed as a sum of low-parametric functions. For example, mean dimension 1 (the lowest possible value) means that it is a purely additive function: $f(x_1, ..., x_N) = f_1(x_1) + ... + f_N(x_N)$.

Assumption: the input variables x_n are independently distributed.

References:

- R. E. Caflisch, W. J. Morokoff, and A. B. Owen: "Valuation of Mortgage Backed Securities Using Brownian Bridges to Reduce Effective Dimension" (1997)
- R. Ballester-Ripoll, E. G. Paredes, and R. Pajarola: "Tensor Algorithms for Advanced Sensitivity Metrics" (2017)

Parameters

- **t** – an N-dimensional `Tensor`
- **marginals** – a list of N vectors (will be normalized if not summing to 1). If None (default), uniform distributions are assumed for all variables

Returns a scalar >= 1

anova.**sobol**(*t*, *mask*, *marginals=None*, *normalize=True*)

Compute Sobol indices (as given by a certain mask) for a tensor and independently distributed input variables.

Reference: R. Ballester-Ripoll, E. G. Paredes, and R. Pajarola: "Sobol Tensor Trains for Global Sensitivity Analysis" (2017)

Parameters

- **t** – an N-dimensional `Tensor`
- **mask** – an N-dimensional mask
- **marginals** – a list of N vectors (will be normalized if not summing to 1). If None (default), uniform distributions are assumed for all variables
- **normalize** – whether to normalize indices by the total variance of the model (True by default)

Returns a scalar >= 0

`anova.`**truncate_anova**(*t*, *mask*, *keepdim=False*, *marginals=None*)

Given a tensor and a mask, return the function that results after deleting all ANOVA terms that do not satisfy the mask.

Example

```
>>> t = ...  # an ND tensor
>>> x = tn.symbols(t.dim())[0]
>>> t2 = tn.truncate_anova(t, mask=tn.only(x), keepdim=False)  # This tensor will␣
↪depend on one variable only
```

Parameters

- **t** –
- **mask** –
- **keepdim** – if True, all dummy dimensions will be preserved, otherwise they will disappear. Default is False
- **marginals** – see *anova_decomposition()*

Returns a `Tensor`

`anova.`**undo_anova_decomposition**(*a*)

Undo the transformation done by *anova_decomposition()*.

Parameters **a** – a `Tensor` obtained with *anova_decomposition()*

Returns a `Tensor` t that has *a* as its ANOVA tensor

3.2.2 autodiff

`autodiff.`**dof**(*t*)

Compute the number of degrees of freedom of a tensor network.

It is the sum of sizes of all its tensor nodes that have the requires_grad=True flag.

Parameters **t** – input tensor

Returns an integer

`autodiff.`**optimize**(*tensors*, *loss_function*, *optimizer=<class 'torch.optim.adam.Adam'>*, *tol=0.0001*, *max_iter=10000.0*, *print_freq=500*, *verbose=True*)

High-level wrapper for iterative learning.

Default stopping criterion: either the absolute (or relative) loss improvement must fall below *tol*. In addition, the rate loss improvement must be slowing down.

Parameters

- **tensors** – one or several tensors; will be fed to *loss_function* and optimized in place
- **loss_function** – must take *tensors* and return a scalar (or tuple thereof)
- **optimizer** – one from https://pytorch.org/docs/stable/optim.html. Default is torch.optim.Adam
- **tol** – stopping criterion
- **max_iter** – default is 1e4
- **print_freq** – progress will be printed every this many iterations
- **verbose** –

3.2.3 automata

automata.**accepted_inputs**(*t*)

Returns all strings accepted by an automaton, in alphabetical order.

Note: each string s will appear as many times as the value t[s]

Parameters **t** – a Tensor

Return Xs a Torch matrix, each row is one string

automata.**length**(*N*)

Todo

Parameters **N** –

Returns

automata.**weight**(*N*, *nsymbols=2*)

For any string, counts how many 1's it has

Parameters

- **N** – number of dimensions
- **nsymbols** – slices per core (default is 2)

Returns a mask tensor

automata.**weight_mask**(*N*, *weight*, *nsymbols=2*)

Accepts a string iff its number of 1's equals (or is in) *weight*

Parameters

- **N** – number of dimensions
- **weight** – an integer (or list thereof): recognized weight(s)
- **nsymbols** – slices per core (default is 2)

Returns a mask tensor

automata.**weight_one_hot**(*N*, *r=None*, *nsymbols=2*)

Given a string with k 1's, it produces a vector that represents k in one hot encoding

Parameters

- **N** – number of dimensions
- **r** –
- **nsymbols** –

Returns a vector of N zeros, except its k-th element which is a 1

3.2.4 create

`create.`**`arange`**(**args*, ***kwargs*)
Creates a 1D `Tensor` (see PyTorch's *arange*).

Parameters

- **args** –
- **kwargs** –

Returns a 1D `Tensor`

`create.`**`eye`**(*n*, *m=None*, *device=None*, *requires_grad=None*)
Generates identity matrix like PyTorch's *eye()*.

Parameters

- **n** – number of rows
- **m** – number of columns (default is n)

Returns a 2D `Tensor`

`create.`**`full`**(*shape*, *fill_value*, ***kwargs*)
Generate a `Tensor` filled with a constant.

Parameters

- **shape** – list of ints
- **fill_value** – constant to fill the tensor with
- **requires_grad** –
- **device** –

Returns a TT `Tensor` of rank 1

`create.`**`full_like`**(*t*, *fill_value*, ***kwargs*)
Calls *full()* with the shape of a given tensor.

Parameters

- **t** – a tensor
- **kwargs** –

Returns a `Tensor`

`create.`**`gaussian`**(*shape*, *sigma_factor=0.2*)
Create a multivariate Gaussian that is axis-aligned (i.e. with diagonal covariance matrix).

Parameters

- **shape** – list of ints
- **sigma_factor** – a real (or list of reals) encoding the ratio sigma / shape. Default is 0.2, i.e. one fifth along each dimension

Returns a `Tensor` that sums to 1

`create.`**`gaussian_like`**(*tensor*, ***kwargs*)
Calls *gaussian()* with the shape of a given tensor.

Parameters

- **`t`** – a tensor
- **`kwargs`** –

Returns a `Tensor`

`create.`**`linspace`**(**args*, ***kwargs*)
Creates a 1D `Tensor` with evenly spaced values (see PyTorch's *linspace*).

Parameters

- **`args`** –
- **`kwargs`** –

Returns a 1D `Tensor`

`create.`**`logspace`**(**args*, ***kwargs*)
Creates a 1D `Tensor` with logarithmically spaced values (see PyTorch's *logspace*).

Parameters

- **`args`** –
- **`kwargs`** –

Returns a 1D `Tensor`

`create.`**`ones`**(**shape*, ***kwargs*)
Generate a `Tensor` filled with ones.

Example

```
>>> tn.ones(10)  # Vector of ones
```

Parameters

- **`shape`** – N ints (or a list of ints)
- **`requires_grad`** –
- **`device`** –

Returns a TT `Tensor` of rank 1

`create.`**`ones_like`**(*t*, ***kwargs*)
Calls *ones()* with the shape of a given tensor.

Parameters

- **`t`** – a tensor
- **`kwargs`** –

Returns a `Tensor`

`create.`**`rand`**(**shape*, ***kwargs*)
Generate a `Tensor` with random cores (and optionally factors), whose entries are uniform in $[0, 1]$.

Example

```
>>> tn.rand([10, 10], ranks_tt=3)  # Rank-3 TT tensor of shape 10x10
```

Parameters

- **shape** – N ints (or a list of ints)
- **ranks_tt** – an integer or list of N-1 ints
- **ranks_cp** – an int or list. If a list, will be interleaved with ranks_tt
- **ranks_tucker** – an int or list
- **requires_grad** – default is False
- **device** –

Returns a random tensor

`create.rand_like(t, **kwargs)`
Calls *rand()* with the shape of a given tensor.

Parameters

- **t** – a tensor
- **kwargs** –

Returns a `Tensor`

`create.randn(*shape, **kwargs)`
Like *rand()*, but entries are normally distributed with $\mu = 0, \sigma = 1$.

`create.randn_like(t, **kwargs)`
Calls *randn()* with the shape of a given tensor.

Parameters

- **t** – a tensor
- **kwargs** –

Returns a `Tensor`

`create.zeros(*shape, **kwargs)`
Generate a `Tensor` filled with zeros.

Parameters

- **shape** – N ints (or a list of ints)
- **requires_grad** –
- **device** –

Returns a TT `Tensor` of rank 1

`create.zeros_like(t, **kwargs)`
Calls *zeros()* with the shape of a given tensor.

Parameters

- **t** – a tensor
- **kwargs** –

Returns a `Tensor`

3.2.5 cross

cross.**cross**(*function*, *domain=None*, *tensors=None*, *function_arg='vectors'*, *ranks_tt=None*, *kickrank=3*, *rmax=100*, *eps=1e-06*, *max_iter=25*, *val_size=1000*, *verbose=True*, *return_info=False*)

Cross-approximation routine that samples a black-box function and returns an N-dimensional tensor train approximating it. It accepts either:

- A domain (tensor product of N given arrays) and a function $\mathbb{R}^N \to \mathbb{R}$
- A list of K tensors of dimension N and equal shape and a function $\mathbb{R}^K \to \mathbb{R}$

Examples

```
>>> tn.cross(function=lambda x: x**2, tensors=[t])  # Compute the element-wise␣
↪square of `t` using 5 TT-ranks
```

```
>>> domain = [torch.linspace(-1, 1, 32)]*5
>>> tn.cross(function=lambda x, y, z, t, w: x**2 + y*z + torch.cos(t + w),␣
↪domain=domain)  # Approximate a function over the rectangle :math:`[-1, 1]^5`
```

```
>>> tn.cross(function=lambda x: torch.sum(x**2, dim=1), domain=domain, function_
↪arg='matrix')  # An example where the function accepts a matrix
```

References:

- I. Oseledets, E. Tyrtyshnikov: "TT-cross Approximation for Multidimensional Arrays" (2009)
- D. Savostyanov, I. Oseledets: "Fast Adaptive Interpolation of Multi-dimensional Arrays in Tensor Train Format" (2011)
- S. Dolgov, R. Scheichl: "A Hybrid Alternating Least Squares - TT Cross Algorithm for Parametric PDEs" (2018)
- A. Mikhalev's maxvolpy package
- I. Oseledets (and others)'s ttpy package

Parameters

- **function** – should produce a vector of P elements. Accepts either N comma-separated vectors, or a matrix (see *function_arg*)
- **domain** – a list of N vectors (incompatible with *tensors*)
- **tensors** – a Tensor or list thereof (incompatible with *domain*)
- **function_arg** – if 'vectors', *function* accepts N vectors of length P each. If 'matrix', a matrix of shape $P \times N$.
- **ranks_tt** – int or list of $N-1$ ints. If None, will be determined adaptively
- **kickrank** – when adaptively found, ranks will be increased by this amount after every iteration (full sweep left-to-right and right-to-left)
- **rmax** – this rank will not be surpassed
- **eps** – the procedure will stop after this validation error is met (as measured after each iteration)
- **max_iter** – int
- **val_size** – size of the validation set

- **verbose** – default is True
- **return_info** – if True, will also return a dictionary with informative metrics about the algorithm's outcome

Returns an N-dimensional TT `Tensor` (if 'return_info'=True, also a dictionary)

3.2.6 derivatives

`derivatives.`**`active_subspace`**(*t*)
Compute the main variational directions of a tensor.

Reference: P. Constantine et al. "Discovering an Active Subspace in a Single-Diode Solar Cell Model" (2017)

See also P. Constantine's data set repository.

Parameters **t** – input tensor

Returns (eigvals, eigvecs): an array and a matrix, encoding the eigenpairs in descending order

`derivatives.`**`curl`**(*ts*, *bounds=None*)
Compute the curl of a 3D vector field.

Parameters

- **ts** – three 3D tensors encoding the x, y, z vector coordinates respectively
- **bounds** –

Returns three tensors of the same shape

`derivatives.`**`divergence`**(*ts*, *bounds=None*)
Computes the divergence (scalar field) out of a vector field encoded in a tensor.

Parameters

- **ts** – an ND vector field, encoded as a list of N ND tensors
- **bounds** –

Returns a scalar field

`derivatives.`**`gradient`**(*t*, *dim='all'*, *bounds=None*)
Compute the gradient of a tensor.

Parameters

- **t** – a `Tensor`
- **dim** – an integer (or list of integers). Default is all
- **bounds** – a pair (or list of pairs) of reals, or None. The bounds for each variable

Returns a `Tensor` (or a list thereof)

`derivatives.`**`laplacian`**(*t*, *bounds=None*)
Computes the Laplacian of a scalar field.

Parameters

- **t** – a `Tensor`
- **bounds** –

Returns a `Tensor`

derivatives.**partial**(*t*, *dim*, *order=1*, *bounds=None*, *periodic=False*, *pad='top'*)
Compute a single partial derivative.

Parameters

- **t** – a Tensor
- **dim** – int or list of ints
- **order** – how many times to derive. Default is 1
- **bounds** – variable(s) range bounds (to compute the derivative step). If None (default), step 1 will be assumed
- **periodic** – int or list of ints (same as *dim*), mark dimensions with periodicity
- **pad** – string or list of strings indicating dimension zero-padding after differentiation. If 'top' (default) or 'bottom', the tensor will retain the same shape after the derivative. If 'none' it will lose one slice

Returns a Tensor

derivatives.**partialset**(*t*, *order=1*, *mask=None*, *bounds=None*)
Given a tensor, compute another one that contains all partial derivatives of certain order(s) and according to some optional mask.

Examples

```
>>> t = tn.rand([10, 10, 10])  # A 3D tensor
>>> x, y, z = tn.symbols(3)
>>> partialset(t, 1, x)  # x
>>> partialset(t, 2, x)  # xx, xy, xz
>>> partialset(t, 2, tn.only(y | z))  # yy, yz, zz
```

Parameters

- **t** – a Tensor
- **order** – an int or list of ints. Default is 1
- **mask** – an optional mask to select only a subset of partials
- **bounds** – a list of pairs [lower bound, upper bound] specifying parameter ranges (used to compute derivative steps). If None (default), all steps will be 1

Returns a Tensor

3.2.7 logic

logic.**absence**(*N*, *which*)
True iff all symbols in *which* are absent.

Parameters

- **N** – int
- **which** – a list of ints

Returns a masked Tensor

logic.**all**(*N*, *which=None*)
Create a formula (N-dimensional tensor) that is satisfied iff all symbols are true.

Parameters

- **N** – an integer
- **which** – list of integers to consider (default: all)

Returns a 2^N Tensor

logic.**any**(*N*, *which=None*)
Create a formula (N-dimensional tensor) that is satisfied iff at least one symbol is true.

Parameters

- **N** – an integer
- **which** – list of integers to consider (default: all)

Returns a 2^N Tensor

logic.**equiv**(*t1*, *t2*)
Checks if two formulas are logically equivalent.

Parameters

- **t1** – a 2^N Tensor
- **t2** – a 2^N Tensor

Returns True if *t1* implies *t2* and vice versa; False otherwise

logic.**false**(*N*)
Create a formula (N-dimensional tensor) that is always false.

Parameters **N** – an integer

Returns a 2^N Tensor

logic.**implies**(*t1*, *t2*)
Checks if a formula implies another one (i.e. is a sufficient condition).

Parameters

- **t1** – a 2^N Tensor
- **t2** – a 2^N Tensor

Returns True if *t1* implies *t2*; False otherwise

logic.**irrelevant_symbols**(*t*)
Finds all variables whose values never affect the formula's output.

Parameters **t** – a 2^N Tensor

Returns a list of integers

logic.**is_contradiction**(*t*)
Checks if a formula is never satisfied.

Parameters **t** – a 2^N tensor

Returns True if *t* is a contradiction; False otherwise

logic.**is_satisfiable**(*t*)
Checks if a formula can be satisfied.

Parameters **t** – a 2^N Tensor

Returns True if *t* is satisfiable; False otherwise

logic.**is_tautology**(*t*)
Checks if a formula is always satisfied.

Parameters **t** – a 2^N Tensor

Returns True if *t* is a tautology; False otherwise

logic.**none**(*N*, *which=None*)
Create a formula (N-dimensional tensor) that is satisfied iff all symbols are false.

Parameters

- **N** – an integer
- **which** – list of integers to consider (default: all)

Returns a 2^N Tensor

logic.**one**(*N*, *which=None*)
Create a formula (N-dimensional tensor) that is satisfied iff one and only one input is true.

Also known as "n-ary exclusive or".

Parameters

- **N** – an integer
- **which** – list of integers to consider (default: all)

Returns a 2^N Tensor

logic.**only**(*t*)
Forces all irrelevant symbols to be zero.

Example

```
>>> x, y = tn.symbols(2)
>>> tn.sum(x)  # Result: 2 (x = True, y = False, and x = True, y = True)
>>> tn.sum(tn.only(x))  # Result: 1 (x = True, y = False)
```

Param a 2^N Tensor

Returns a masked Tensor

logic.**presence**(*N*, *which*)
True iff all symbols in *which* are present.

Parameters

- **N** – int
- **which** – a list of ints

Returns a masked Tensor

logic.**relevant_symbols**(*t*)
Finds all variables whose values affect the formula's output in at least one case.

Parameters **t** – a 2^N Tensor

Returns a list of integers

logic.**symbols**(*N*)
Generate N Boolean symbols (each represented as an N-dimensional tensor).

Parameters **N** – an integer

Returns a list of N 2^N Tensor

`logic.`**`true`**(*N*)
Create a formula (N-dimensional tensor) that is always true.

Parameters **N** – an integer

Returns a 2^N `Tensor`

3.2.8 metrics

`metrics.`**`dist`**(*t1*, *t2*)
Computes the Euclidean distance between two tensors. Generally faster than *tn.norm(t1-t2)*.

Parameters

- **t1** – a `Tensor` (or a PyTorch tensor)
- **t2** – a `Tensor` (or a PyTorch tensor)

Returns a scalar ≥ 0

`metrics.`**`dot`**(*t1*, *t2*, *k=None*)
Generalized tensor dot product: contracts the k leading dimensions of two tensors of dimension N1 and N2.

- **If k is None:**
  - If N1 == N2, returns a scalar (dot product between the two tensors)
  - If N1 < N2, the result will have dimension N2 - N1
  - If N2 < N1, the result will have dimension N1 - N2

  Example: suppose t1 has shape 3 x 4 and t2 has shape 3 x 4 x 5 x 6. Then, tn.dot(t1, t2) will have shape 5 x 6.
- **If k is given:** The trailing (N1-k) dimensions from the 1st tensor will be sorted backwards, and then the trailing (N2-k) dimensions from the 2nd tensor will be appended to them.

  Example: suppose t1 has shape 3 x 4 x 5 x 6 and t2 has shape 3 x 4 x 10 x 11. Then, tn.dot(t1, t2, k=2) will have shape 6 x 5 x 10 x 11.

Parameters

- **t1** – a `Tensor` (or a PyTorch tensor)
- **t2** – a `Tensor` (or a PyTorch tensor)
- **k** – an int (default: None)

Returns a scalar (if k is None and t1.dim() == t2.dim()), a tensor otherwise

`metrics.`**`kurtosis`**(*t*, *fisher=True*)
Computes the kurtosis of a `Tensor`. Note: this function uses cross-approximation (`tntorch.cross()`).

Parameters

- **t** – a `Tensor`
- **fisher** – if True (default) Fisher's definition is used, otherwise Pearson's (aka excess)

Returns a scalar

`metrics.`**`mean`**(*t*, *dim=None*, *keepdim=False*)
Computes the mean of a `Tensor` along all or some of its dimensions.

Parameters

- **t** – a `Tensor`
- **dim** – an int or list of ints (default: all)
- **keepdim** – whether to keep the same number of dimensions

Returns a scalar

`metrics.`**norm**(*t*)
Computes the L^2 (Frobenius) norm of a tensor.

Parameters **t** – a `Tensor`

Returns a scalar ≥ 0

`metrics.`**normsq**(*t*)
Computes the squared norm of a `Tensor`.

Parameters **t** – a `Tensor`

Returns a scalar ≥ 0

`metrics.`**r_squared**(*gt*, *approx*)
Computes the R^2 score between two tensors (torch or tntorch).

Parameters

- **gt** – a torch or tntorch tensor
- **approx** – a torch or tntorch tensor

Returns a scalar <= 1

`metrics.`**relative_error**(*gt*, *approx*)
Computes the relative error between two tensors (torch or tntorch).

Parameters

- **gt** – a torch or tntorch tensor
- **approx** – a torch or tntorch tensor

Returns a scalar ≥ 0

`metrics.`**rmse**(*gt*, *approx*)
Computes the RMSE between two tensors (torch or tntorch).

Parameters

- **gt** – a torch or tntorch tensor
- **approx** – a torch or tntorch tensor

Returns a scalar ≥ 0

`metrics.`**skew**(*t*)
Computes the skewness of a `Tensor`. Note: this function uses cross-approximation (`tntorch.cross()`).

Parameters **t** – a `Tensor`

Returns a scalar

`metrics.`**std**(*t*)
Computes the standard deviation of a `Tensor`.

Parameters **t** – a `Tensor`

Returns a scalar ≥ 0

metrics.**sum**(*t*, *dim=None*, *keepdim=False*, *_normalize=False*)
Compute the sum of a tensor along all (or some) of its dimensions.

Parameters

- **t** – input Tensor
- **dim** – an int or list of ints. By default, all dims will be summed
- **keepdim** – if True, summed dimensions will be kept as singletons. Default is False

Returns a scalar (if keepdim is False and all dims were chosen) or Tensor otherwise

metrics.**var**(*t*)
Computes the variance of a Tensor.

Parameters **t** – a Tensor

Returns a scalar ≥ 0

3.2.9 ops

ops.**abs**(*t*)
Element-wise absolute value computed using cross-approximation; see PyTorch's *abs()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**acos**(*t*)
Element-wise arccosine computed using cross-approximation; see PyTorch's *acos()*.

Parameters **t** – input :class:'Tensor's

Returns a Tensor

ops.**add**(*t1*, *t2*)
Element-wise addition computed using cross-approximation; see PyTorch's *add()*.

Parameters

- **t1** – input Tensor
- **t2** – input Tensor

Returns a Tensor

ops.**asin**(*t*)
Element-wise arcsine computed using cross-approximation; see PyTorch's *asin()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**atan2**(*t1*, *t2*)
Element-wise arctangent computed using cross-approximation; see PyTorch's *atan2()*.

Parameters

- **t1** – input Tensor
- **t2** – input Tensor

Returns a Tensor

ops.**cos**(*t*)
Element-wise cosine computed using cross-approximation; see PyTorch's *cos()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**cosh**(*t*)
Element-wise hyperbolic cosine computed using cross-approximation; see PyTorch's *cosh()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**cumprod**(*t*, *dim=None*)
Computes the cumulative sum of a tensor along one or several dims, similarly to PyTorch's *cumprod()*.

Note: this function is approximate and uses cross-approximation (tntorch.cross())

Parameters

- **t** – input Tensor
- **dim** – an int or list of ints (default: all)

Returns a Tensor of the same shape

ops.**cumsum**(*t*, *dim=None*)
Computes the cumulative sum of a tensor along one or several dims, similarly to PyTorch's *cumsum()*.

Parameters

- **t** – input Tensor
- **dim** – an int or list of ints (default: all)

Returns a Tensor of the same shape

ops.**div**(*t1*, *t2*)
Element-wise division computed using cross-approximation; see PyTorch's *div()*.

Parameters

- **t1** – input Tensor
- **t2** – input Tensor

Returns a Tensor

ops.**erf**(*t*)
Element-wise error function computed using cross-approximation; see PyTorch's *erf()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**erfinv**(*t*)
Element-wise inverse error function computed using cross-approximation; see PyTorch's *erfinv()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**exp**(*t*)
Element-wise exponentiation computed using cross-approximation; see PyTorch's *exp()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**log**(*t*)
Element-wise natural logarithm computed using cross-approximation; see PyTorch's *log()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**log10**(*t*)
Element-wise base-10 logarithm computed using cross-approximation; see PyTorch's *log10()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**log2**(*t*)
Element-wise base-2 logarithm computed using cross-approximation; see PyTorch's *log2()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**mul**(*t1*, *t2*)
Element-wise product computed using cross-approximation; see PyTorch's *mul()*.

Parameters

- **t1** – input Tensor
- **t2** – input Tensor

Returns a Tensor

ops.**pow**(*t1*, *t2*)
Element-wise power operation computed using cross-approximation; see PyTorch's *pow()*.

Parameters

- **t1** – input Tensor
- **t2** – input Tensor

Returns a Tensor

ops.**reciprocal**(*t*)
Element-wise reciprocal computed using cross-approximation; see PyTorch's *reciprocal()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**rsqrt**(*t*)
Element-wise square-root reciprocal computed using cross-approximation; see PyTorch's *rsqrt()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**sigmoid**(*t*)
Element-wise sigmoid computed using cross-approximation; see PyTorch's *igmoid()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**sin**(*t*)
Element-wise sine computed using cross-approximation; see PyTorch's *in()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**sinh**(*t*)
Element-wise hyperbolic sine computed using cross-approximation; see PyTorch's *inh()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**sqrt**(*t*)
Element-wise square root computed using cross-approximation; see PyTorch's *qrt()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**tan**(*t*)
Element-wise tangent computed using cross-approximation; see PyTorch's *tan()*.

Parameters **t** – input Tensor

Returns a Tensor

ops.**tanh**(*t*)
Element-wise hyperbolic tangent computed using cross-approximation; see PyTorch's *tanh()*.

Parameters **t** – input Tensor

Returns a Tensor

3.2.10 round

round.**round**(*t*, ***kwargs*)
Copies and rounds a tensor (see *tensor.Tensor.round()*.

Parameters

- **t** – input Tensor
- **kwargs** –

Returns a rounded copy of *t*

round.**round_tt**(*t*, ***kwargs*)
Copies and rounds a tensor (see *tensor.Tensor.round_tt()*.

Parameters

- **t** – input Tensor
- **kwargs** –

Returns a rounded copy of *t*

round.**round_tucker**(*t*, ***kwargs*)
Copies and rounds a tensor (see *tensor.Tensor.round_tucker()*.

Parameters

- **t** – input Tensor
- **kwargs** –

Returns a rounded copy of *t*

round.**truncated_svd**(*M*, *delta=None*, *eps=None*, *rmax=None*, *left_ortho=True*, *algorithm='svd'*, *verbose=False*)

Decomposes a matrix M (size (m x n) in two factors U and V (sizes m x r and r x n) with bounded error (or given r).

Parameters

- **M** – a matrix
- **delta** – if provided, maximum error norm
- **eps** – if provided, maximum relative error
- **rmax** – optionally, maximum r
- **left_ortho** – if True (default), U will be orthonormal. If False, V will
- **algorithm** – 'svd' (default) or 'eig'. The latter is often faster, but less accurate
- **verbose** – Boolean

Returns U, V

3.2.11 tensor

class tensor.**Tensor**(*data*, *Us=None*, *idxs=None*, *device=None*, *requires_grad=None*, *ranks_cp=None*, *ranks_tucker=None*, *ranks_tt=None*, *eps=None*, *max_iter=25*, *tol=0.0001*, *verbose=False*)

Bases: object

Class for all tensor networks. Currently supported: tensor train (TT), CANDECOMP/PARAFAC (CP), Tucker, and hybrid formats.

Internal representation: an ND tensor has N cores, with each core following one of four options:

- Size $R_{n-1} \times I_n \times R_n$ (standard TT core)
- Size $R_{n-1} \times S_n \times R_n$ (TT-Tucker core), accompanied by an $I_n \times S_n$ factor matrix
- Size $I_n \times R$ (CP factor matrix)
- Size $S_n \times R_n$ (CP-Tucker core), accompanied by an $I_n \times S_n$ factor matrix

The constructor can either:

- Decompose an uncompressed tensor
- Use an explicit list of tensor cores (and optionally, factors)

See this notebook for examples of use.

Parameters

- **data** – a NumPy ndarray, PyTorch tensor, or a list of cores (which can represent either CP factors or TT cores)
- **Us** – optional list of Tucker factors
- **idxs** – annotate maskable tensors (*advanced users*)
- **device** – PyTorch device
- **requires_grad** – Boolean
- **ranks_cp** – an integer (or list)
- **ranks_tucker** – an integer (or list)

- **ranks_tt** – an integer (or list)
- **eps** – maximal error
- **max_iter** – maximum number of iterations when computing a CP decomposition using ALS
- **tol** – stopping criterion (change in relative error) when computing a CP decomposition using ALS
- **verbose** – Boolean

Returns a *Tensor*

as_leaf()
Makes this tensor a leaf (optimizable) tensor, thus forgetting the operations from which it arose.

Example

```
>>> t = tn.rand([10]*3, requires_grad=True)  # Is a leaf
>>> t *= 2  # Is not a leaf
>>> t.as_leaf()  # Is a leaf again
```

clone()
Creates a copy of this tensor (calls PyTorch's *clone()* on all internal tensor network nodes)

Returns another compressed tensor

decompress_tucker_factors(*dim='all'*, *_clone=True*)
Decompresses this tensor along the Tucker factors only.

Parameters **dim** – int, list, or 'all' (default)

Returns a *Tensor* in CP/TT format, without Tucker factors

dim()
Returns the number of dimensions of this tensor.

Returns an int

dot(*other*, ***kwargs*)
See *metrics.dot()*.

factor_orthogonalize(*mu*)
Pushes the factor's non-orthogonal part to its corresponding core.

This method works in place.

Parameters **mu** – an int between 0 and N-1

left_orthogonalize(*mu*)
Makes the mu-th core left-orthogonal and pushes the R factor to its right core. This may change the ranks of the cores.

This method works in place.

Note: internally, this method will turn CP (or CP-Tucker) cores into TT (or TT-Tucker) ones.

Parameters **mu** – an int between 0 and N-1

Returns the R factor

mean(***kwargs*)
See *metrics.mean()*.

norm(***kwargs*)
See *metrics.norm()*.

normsq(***kwargs*)
See *metrics.normsq()*.

numcoef()
Counts the total number of compressed coefficients of this tensor.

Returns an integer

numel()
Counts the total number of uncompressed elements of this tensor.

Returns an integer

numpy()
Decompresses this tensor into a NumPy ndarray.

Returns a NumPy tensor

orthogonalize(*mu*)
Apply all left and right orthogonalizations needed to make the tensor mu-orthogonal.

This method works in place.

Note: internally, this method will turn CP (or CP-Tucker) cores into TT (or TT-Tucker) ones.

Parameters **mu** – an int between 0 and N-1

Returns L, R: left and right factors

ranks_tt
Returns the TT ranks of this tensor.

Returns a vector of integers

ranks_tucker
Returns the Tucker ranks of this tensor.

Returns a vector of integers

repeat(**rep*)
Returns another tensor repeated along one or more axes; works like PyTorch's *repeat()*.

Parameters **rep** – a list, possibly longer than the tensor's number of dimensions

Returns another tensor

right_orthogonalize(*mu*)

Makes the mu-th core right-orthogonal and pushes the L factor to its left core. Note: this may change the ranks of the tensor.

This method works in place.

Note: internally, this method will turn CP (or CP-Tucker) cores into TT (or TT-Tucker) ones.

Parameters **mu** – an int between 0 and N-1

Returns the L factor

round(*eps=1e-14*, ***kwargs*)
General recompression. Attempts to reduce TT ranks first; then does Tucker rounding with the remaining error budget.

Parameters

- **eps** – this relative error will not be exceeded
- **kwargs** – passed to *round_tt()* and *round_tucker()*

round_tt(*eps=1e-14*, *rmax=None*, *algorithm='svd'*, *verbose=False*)
Tries to recompress this tensor in place by reducing its TT ranks.

Note: this method will turn CP (or CP-Tucker) cores into TT (or TT-Tucker) ones.

Parameters

- **eps** – this relative error will not be exceeded
- **rmax** – all ranks should be rmax at most (default: no limit)
- **algorithm** – 'svd' (default) or 'eig'. The latter can be faster, but less accurate
- **verbose** –

round_tucker(*eps=1e-14*, *rmax=None*, *dim='all'*, *algorithm='svd'*)
Tries to recompress this tensor in place by reducing its Tucker ranks.

Note: this method will turn CP (or CP-Tucker) cores into TT (or TT-Tucker) ones.

Parameters

- **eps** – this relative error will not be exceeded
- **rmax** – all ranks should be rmax at most (default: no limit)
- **algorithm** – 'svd' (default) or 'eig'. The latter can be faster, but less accurate
- **verbose** –

set_factors(*name*, *dim='all'*, *requires_grad=False*)
Sets factors Us of this tensor to be of a certain family.

Parameters

- **name** – See *tools.generate_basis()*
- **dim** – list of factors to set; default is 'all'
- **requires_grad** – whether the new factors should be optimizable. Default is False

shape
Returns the shape of this tensor.

Returns a PyTorch shape object

size()
Alias for *shape()* (as PyTorch does)

std(***kwargs*)
See *metrics.std()*.

sum(***kwargs*)
See *metrics.sum()*.

torch()
Decompresses this tensor into a PyTorch tensor.

Returns a PyTorch tensor

tt()
Casts this tensor as a pure TT format.

Returns a *Tensor* in the TT format

tucker_core()
If this is a Tucker-like tensor, returns its Tucker core as an explicit PyTorch tensor.

If this tensor does not have Tucker factors, then it returns the full decompressed tensor.

Returns a PyTorch tensor

var(***kwargs*)
See *metrics.var()*.

3.2.12 tools

tools.**cat**(**ts*, *dim*)
Concatenate two or more tensors along a given dim, similarly to PyTorch's *cat()*.

Parameters

- **ts** – a list of Tensor
- **dim** – an int

Returns a Tensor of the same shape as all tensors in the list, except along *dim* where it has the sum of shapes

tools.**flip**(*t*, *dim*)
Reverses the order of a tensor along one or several dimensions; see NumPy's or PyTorch's *flip()*.

Parameters

- **t** – input Tensor
- **dims** – an int or list of ints

Returns another Tensor of the same shape

tools.**generate_basis**(*name*, *shape*, *orthonormal=False*)
Generate a factor matrix whose columns are functions of a truncated basis.

Parameters

- **name** – 'dct', 'legendre', 'chebyshev' or 'hermite'
- **shape** – two integers
- **orthonormal** – whether to orthonormalize the basis

Returns a PyTorch matrix of *shape*

tools.**hash**(*t*)
Computes an integer number that depends on the tensor entries (not on its internal compressed representation).

We obtain it as $\langle T, W \rangle$, where W is a rank-1 tensor of weights selected at random (always the same seed).

Returns an integer

tools.**left_unfolding**(*core*)
Computes the left unfolding of a 3D PyTorch tensor.

Parameters **core** – a PyTorch tensor of shape $I_1 \times I_2 \times I_3$

Returns a PyTorch matrix of shape $I_1 I_2 \times I_3$

tools.**mask**(*t*, *mask*)
Masks a tensor. Basically an element-wise product, but this function makes sure slices are matched according to their "meaning" (as annotated by the tensor's *idx* field, if available)

Parameters

- **t** – input `Tensor`
- **mask** – a mask `Tensor`

Returns masked `Tensor`

`tools.`**meshgrid**(**axes*)
See NumPy's or PyTorch's *meshgrid()*.

Parameters **axes** – a list of N ints or torch vectors

Returns a list of N `Tensor`, of N dimensions each

`tools.`**reduce**(*ts*, *function*, *eps=0*, *rmax=2147483647*, *algorithm='svd'*, *verbose=False*, ***kwargs*)
Compute a tensor as a function to all tensors in a sequence.

Example 1 (addition)

```
>>> import operator
>>> tn.reduce([t1, t2], operator.add)
```

Example 2 (cat with bounded rank)

```
>>> tn.reduce([t1, t2], tn.cat, rmax=10)
```

Parameters

- **ts** – A generator (or list) of `Tensor`
- **eps** – intermediate tensors will be rounded at this error when climbing up the hierarchy
- **rmax** – no node should exceed this number of ranks
- **algorithm** – passed to *round.round()*
- **verbose** – Boolean

Returns the reduced result

`tools.`**right_unfolding**(*core*)
Computes the right unfolding of a 3D PyTorch tensor.

Parameters **core** – a PyTorch tensor of shape $I_1 \times I_2 \times I_3$

Returns a PyTorch matrix of shape $I_1 \times I_2 I_3$

`tools.`**sample**(*t*, *P=1*)
Generate P points (with replacement) from a joint PDF distribution represented by a tensor.

The tensor does not have to sum 1 (will be handled in a normalized form).

Parameters

- **t** – a `Tensor`
- **P** – how many samples to draw (default: 1)

Return Xs an integer matrix of size $P \times N$

`tools.`**squeeze**(*t*, *dim=None*)
Removes singleton dimensions.

Parameters

- **t** – input Tensor
- **dim** – which dim to delete. By default, all that have size 1

Returns another Tensor, without dummy (singleton) indices

tools.**transpose**(*t*)
Inverts the dimension order of a tensor, e.g. $I_1 \times I_2 \times I_3$ becomes $I_3 \times I_2 \times I_1$.

Parameters **t** – input tensor

Returns another Tensor, indexed by dimensions in inverse order

tools.**ttm**(*t*, *U*, *dim=None*, *transpose=False*)
Tensor-times-matrix (TTM) along one or several dimensions.

Parameters

- **t** – input Tensor
- **U** – one or several factors
- **dim** – one or several dimensions (may be vectors or matrices). If None, the first len(U) dims are assumed
- **transpose** – if False (default) the contraction is performed along U's rows, else along its columns

Returns transformed Tensor

tools.**unbind**(*t*, *dim*)
Slices a tensor along a dimension and returns the slices as a sequence, like PyTorch's *unbind()*.

Parameters

- **t** – input Tensor
- **dim** – an int

Returns a list of Tensor, as many as *t.shape[dim]*

tools.**unfolding**(*data*, *n*)
Computes the n-th mode unfolding of a PyTorch tensor.

Parameters

- **data** – a PyTorch tensor
- **n** – unfolding mode

Returns a PyTorch matrix

tools.**unsqueeze**(*t*, *dim*)
Inserts singleton dimensions at specified positions.

Parameters

- **t** – input Tensor
- **dim** – int or list of int

Returns a Tensor with dummy (singleton) dimensions inserted at the positions given by *dim*

3.3 Tutorial Notebooks

3.3.1 Introduction

Welcome to *tntorch*! This notebook illustrates a few basic tensor manipulation and optimization use cases.

First Example

[1]:
```
import torch
import tntorch as tn
```

Let's create a 2D random tensor of size 128×128 and tensor train rank 10 (equivalent to a rank-10 matrix). The `requires_grad` flag tells PyTorch that this tensor should be optimizable:

[2]:
```
t = tn.randn(128, 128, ranks_tt=10, requires_grad=True)
t
```

[2]:
```
2D TT tensor:

 128 128
  |   |
 (0) (1)
 / \ / \
1   10  1
```

The spatial dimensions are shown above, the TT ranks are below, and the cores are listed as `(0)` and `(1)` in the middle.

In many ways, compressed tensors can be treated as if they were normal, uncompressed tensors:

[3]:
```
print(tn.mean(t))
print(tn.var(t))
print(tn.norm(t))
print(tn.norm(t+t))
print(tn.norm(2*t))
```

```
tensor(0.0153, grad_fn=<DivBackward1>)
tensor(10.3761, grad_fn=<DivBackward1>)
tensor(412.3184, grad_fn=<SqrtBackward>)
tensor(824.6368, grad_fn=<SqrtBackward>)
tensor(824.6368, grad_fn=<SqrtBackward>)
```

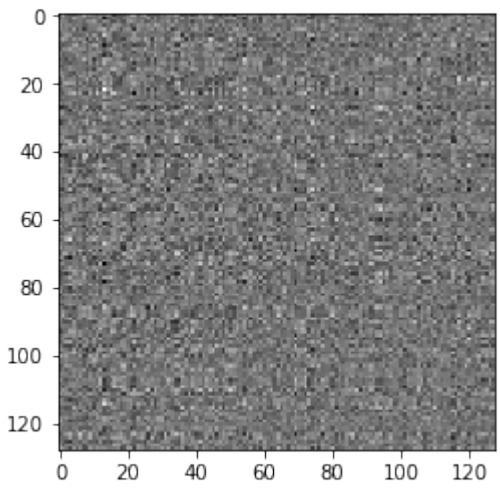
To decompress a tensor into a `torch.Tensor`, we can use the function `torch()`:

[4]:
```
print(t.torch().shape)
```

```
torch.Size([128, 128])
```

The function `numpy()` returns the same, just as a NumPy tensor. Let's use it to visualize the contents of our random tensor in *matplotlib*:

[5]:
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.imshow(t.numpy(), cmap='gray')
plt.show()
```

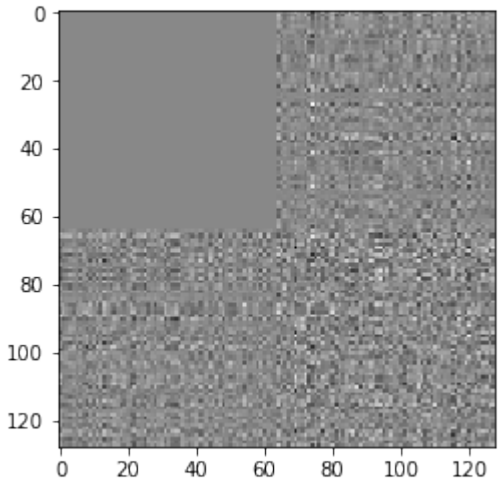


To optimize tensors we can use *tntorch*'s helper function `optimize()`.

Now, we will make our tensor zero over the top left quadrant by minimizing its norm:

[6]:
```
def loss(t):
    return tn.norm(t[:64, :64])

tn.optimize(t, loss)
plt.imshow(t.numpy(), cmap='gray')
plt.show()
```

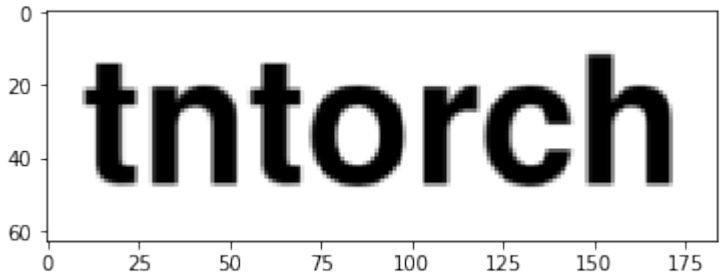
```
iter: 0     | loss: 216.324445 | total time:    0.0011
iter: 500   | loss:  93.631189 | total time:    0.4329
iter: 1000  | loss:  19.823667 | total time:    0.9703
iter: 1322  | loss:   0.082656 | total time:    1.2697 <- converged (tol=0.0001)
```



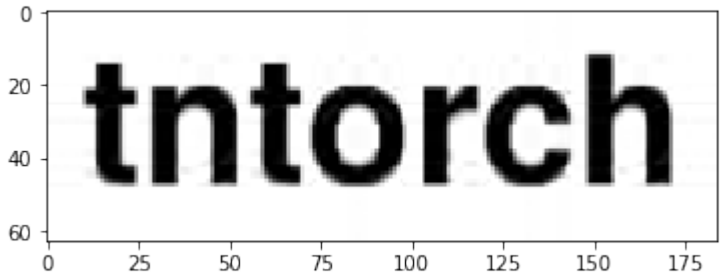
Second Example

Next, we will fit our tensor to be a rank-10 approximation of a grayscale image:

[7]:
```
im = torch.DoubleTensor(plt.imread('../../images/text.png'))
plt.imshow(im.numpy(), cmap='gray')
plt.show()

t = tn.rand(im.shape, ranks_tt=10, requires_grad=True)

def loss(t):
    return tn.dist(t, im)  # Equivalent to torch.norm(t.torch() - im)

tn.optimize(t, loss)
plt.imshow(t.numpy(), cmap='gray', vmin=im.min(), vmax=im.max())
plt.show()
```




```
iter: 0     | loss: 197.608333 | total time:    0.0005
iter: 500   | loss:  29.738809 | total time:    0.6222
iter: 1000  | loss:  21.908585 | total time:    1.2766
iter: 1500  | loss:   9.171426 | total time:    1.8568
iter: 2000  | loss:   4.394665 | total time:    2.4462
iter: 2500  | loss:   2.824173 | total time:    3.0283
iter: 2688  | loss:   2.709813 | total time:    3.2575 <- converged (tol=0.0001)
```



For other available tensor formats beyond plain TT, see *this notebook*.

3.3.2 Active Subspaces

Sometimes, the behavior of an N-dimensional model f can be explained best by a *linear reparameterization* of its inputs variables, i.e. we can write $f(\mathbf{x}) = g(\mathbf{y}) = g(\mathbf{M} \cdot \mathbf{x})$ where $\mathbf{M}$ has size $M \times N$ and $M < N$. When this happens, we say that f admits an M-dimensional *active subspace* with basis given by $\mathbf{M}$'s rows. Those basis vectors are the main directions of variance of the function f.

The main directions are the eigenvectors of the matrix

$$\mathbb{E}[\nabla f^T \cdot \nabla f] = \begin{pmatrix} \mathbb{E}[f_{x_1} \cdot f_{x_1}] & \dots & \mathbb{E}[f_{x_1} \cdot f_{x_N}] \\ \dots & \dots & \dots \\ \mathbb{E}[f_{x_N} \cdot f_{x_1}] & \dots & \mathbb{E}[f_{x_N} \cdot f_{x_N}] \end{pmatrix}$$

whereas the eigenvalues reveal the subspace's dimensionality –that is, a large gap between the M-th and $(M+1)$-th eigenvalue indicates that an M-dimensional active subspace is present.

The necessary expected values are easy to compute from a tensor decomposition: they are just dot products between tensors. We will show a small demonstration of that in this notebook using a 4D function.

Reference: see e.g. "Discovering an Active Subspace in a Single-Diode Solar Cell Model", P. Constantine et al. (2015).

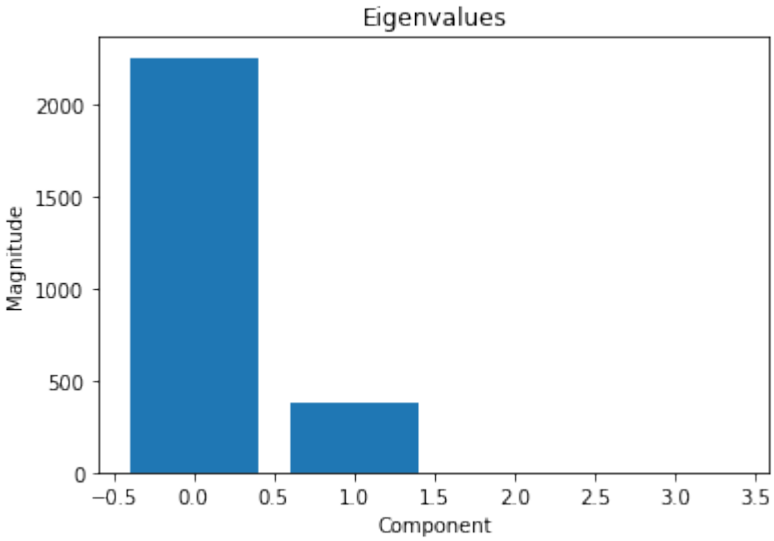
[1]:
```
import tntorch as tn
import torch

def f(X):
    return X[:, 0] * X[:, 1] + X[:, 2]

ticks = 64
P = 100
N = 4

X = torch.rand((P, N))
X *= (ticks-1)
X = torch.round(X)
y = f(X)
```

We will fit this function `f` using a low-degree expansion in terms of *Legendre polynomials*.

[2]:
```
t = tn.rand(shape=[ticks]*N, ranks_tt=2, ranks_tucker=2, requires_grad=True)
t.set_factors('legendre')

def loss(t):
    return torch.norm(t[X].torch()-y) / torch.norm(y)
tn.optimize(t, loss)
```

```
iter: 0      | loss:   0.999513 | total time:    0.0020
iter: 500    | loss:   0.977497 | total time:    0.8296
iter: 1000   | loss:   0.763221 | total time:    1.6445
iter: 1500   | loss:   0.044802 | total time:    2.5523
iter: 2000   | loss:   0.008546 | total time:    3.4807
iter: 2266   | loss:   0.008208 | total time:    3.9928 <- converged (tol=0.0001)
```

[3]:
```
eigvals, eigvecs = tn.active_subspace(t)

import matplotlib.pyplot as plt
%matplotlib inline
plt.figure()
plt.bar(range(N), eigvals.detach().numpy())
plt.title('Eigenvalues')
plt.xlabel('Component')
plt.ylabel('Magnitude')
plt.show()
```



In view of those eigenvalues, we can conclude that the learned model can be written (almost) perfectly in terms of 2 linearly reparameterized variables.

3.3.3 ANOVA Decomposition

The *analysis of variances (ANOVA) decomposition* is well-defined for any square-integrable multidimensional function $f : \mathbb{R}^N \to \mathbb{R}$. If the input variables $\{x_0, \dots, x_{N-1}\}$ are independently distributed random variables, the ANOVA decomposition partitions the total variance of the model, $\mathrm{Var}[f]$, as a sum of variances of orthogonal functions $\mathrm{Var}[f_\alpha]$ for all possible subsets α of the input variables. Each f_α depends effectively on the variables contained in α only, and is constant with respect to the rest.

[1]:
```
import torch
import tntorch as tn

N = 4
t = tn.rand([32]*N, ranks_tt=5)
```

Let's compute all ANOVA terms in one single tensor network:

[2]:
```
anova = tn.anova_decomposition(t)
anova
```

[2]:
```
4D TT-Tucker tensor:

 33  33  33  33
  |   |   |   |
 32  32  32  32
 (0) (1) (2) (3)
 / \ / \ / \ / \
1   5   5   5   1
```

This tensor `anova` indexes *all* 2^N functions f_α of the ANOVA decomposition of f, and we can access it using our tensor masks.

Reference: *"Sobol Tensor Trains for Global Sensitivity Analysis"*, R. Ballester-Ripoll, E. G. Paredes, R. Pajarola (2017).

Manipulating the Decomposition

For example, let's keep all terms that *do not* interact with w:

[3]:
```
x, y, z, w = tn.symbols(N)
anova_cut = tn.mask(anova, ~w)
```

We can undo the decomposition to obtain a regular tensor again:

[4]:
```
t_cut = tn.undo_anova_decomposition(anova_cut)
```

As expected, our truncated tensor `t_cut` has become constant with respect to the fourth variable w:

[5]:
```
t_cut[0, 0, 0, :].torch()
```

[5]:
```
tensor([6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559,
        6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559,
        6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559, 6.1559,
        6.1559, 6.1559, 6.1559, 6.1559, 6.1559])
```

How much did we lose by making that variable unimportant?

[6]:
```
print('The truncated tensor accounts for {:g}% of the original variance.'.format(tn.
↪var(t_cut) / tn.var(t) * 100))
```

```
The truncated tensor accounts for 53.3676% of the original variance.
```

... which is also what *Sobol's method* gives us:

[7]:
```
tn.sobol(t, ~w) * 100
```

[7]:
```
tensor(53.3676)
```

or, equivalently,

[8]:
```
tn.sobol(t, tn.only(x | y | z)) * 100
```

[8]:
```
tensor(53.3676)
```

Note that the empty subfunction $f_\emptyset$ is constant, and its value everywhere coincides with the original function's mean:

[9]:
```
empty = tn.undo_anova_decomposition(tn.mask(anova, tn.none(N)))
print(tn.var(empty))  # It's a constant function
print(empty[0, 0, 0, 0], tn.mean(t))  # Coincides with the global mean
```

```
tensor(6.5831e-14)
tensor(8.3357) tensor(8.3357)
```

Also note that summing all ANOVA subfunctions results in the original function:

[10]:
```
all_summed = tn.undo_anova_decomposition(tn.mask(anova, tn.true(N)))
tn.relative_error(t, all_summed)
```

[10]:
```
tensor(1.8894e-08)
```

Truncating Dimensionality

The low-dimensional terms of the ANOVA decomposition (i.e. terms that depend on a few variables only) are usually the ones that play the most important role in many analytical and real-world models.

We will now approximate our original function as a sum of (at most) bivariate functions. To that end we will use a *weight mask tensor*.

[11]:
```
m = tn.weight_mask(N, [0, 1, 2])  # Keep tuples with zero, one or two '1's
print('We will keep only {:g} of the original ANOVA terms'.format(tn.sum(m)))
t_cut = tn.undo_anova_decomposition(tn.mask(tn.anova_decomposition(t), m))
print('Relative error after truncation: {}'.format(tn.relative_error(t, t_cut)))
```

```
We will keep only 11 of the original ANOVA terms
Relative error after truncation: 0.028525682742228935
```

Visualizing the ANOVA Decomposition

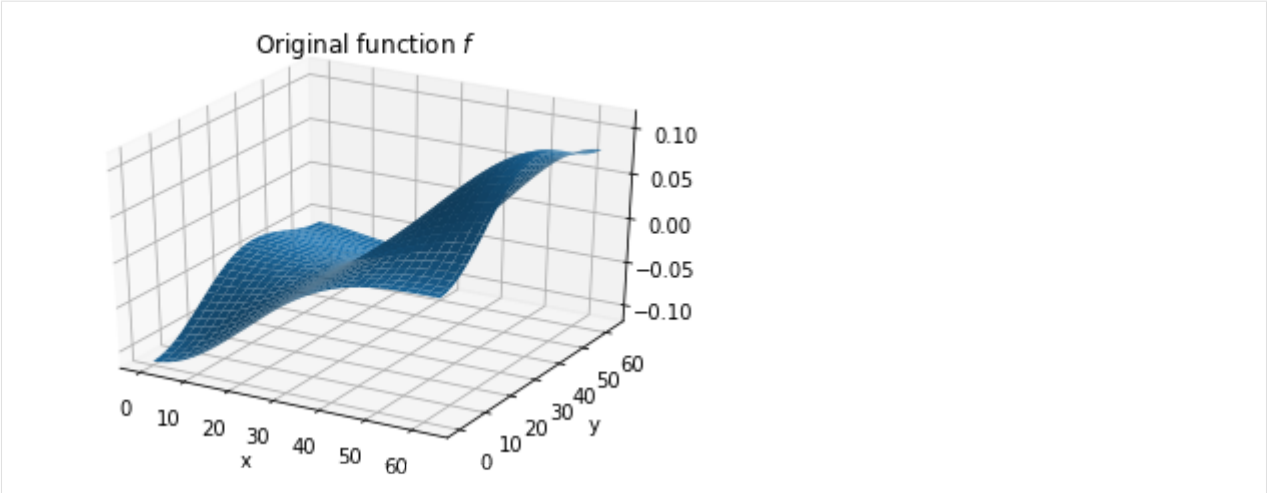
Let's now restrict ourselves to $N = 2$ so that we can easily plot the ANOVA subfunctions and get an idea how they look like.

[12]:
```
# We set up a smooth 2D tensor, sum of a few cosine wavefunctions
N = 2
t = tn.randn([64]*N, ranks_tt=4, ranks_tucker=3)
t.set_factors('dct')
t
```

[12]:
```
2D TT-Tucker tensor:

 64  64
  |   |
  3   3
 (0) (1)
 / \ / \
1   4   1
```

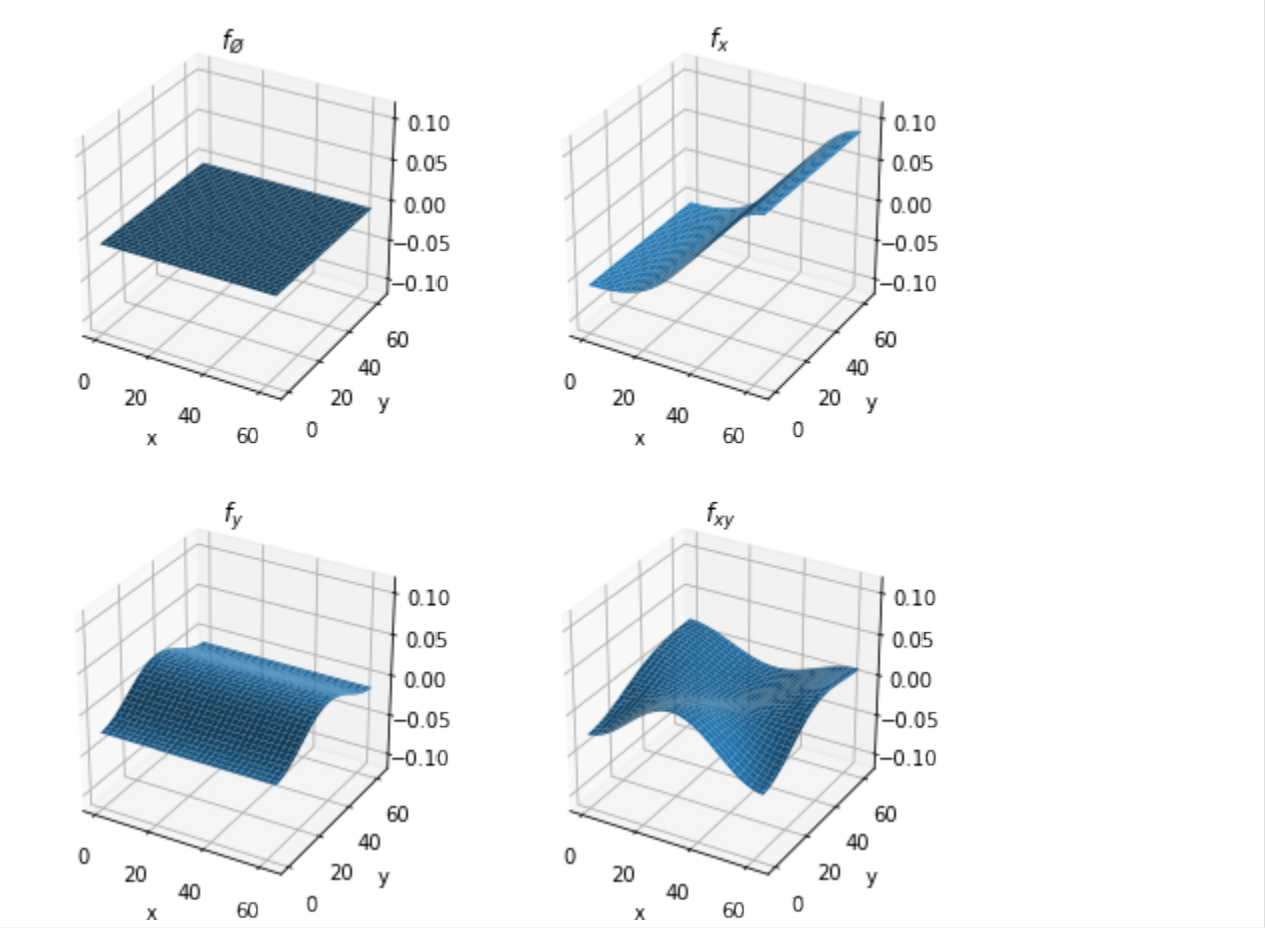
The full function looks like this:

[13]:
```
import matplotlib.pyplot as plt
%matplotlib inline
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y = np.meshgrid(np.arange(t.shape[0]), np.arange(t.shape[1]), indexing='ij')
surf = ax.plot_surface(X, Y, t.numpy())
plt.title('Original function $f$')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Next we will show the $2^N = 4$ ANOVA subfunctions:

[14]:
```
x, y = tn.symbols(N)
anova = tn.anova_decomposition(t)
zlim = [t.numpy().min(), t.numpy().max()]

fig = plt.figure(figsize=(8, 8))

ax = fig.add_subplot(2, 2, 1, projection='3d')
ax.plot_surface(X, Y, tn.undo_anova_decomposition(tn.mask(anova, tn.none(N))).
↪numpy())  # Equivalent to ~x & ~y
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlim3d(zlim[0], zlim[1])
ax.set_title('$f_{\O}$')

ax = fig.add_subplot(2, 2, 2, projection='3d')
ax.plot_surface(X, Y, tn.undo_anova_decomposition(tn.mask(anova, tn.only(x))).numpy())
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlim3d(zlim[0], zlim[1])
ax.set_title('$f_x$')

ax = fig.add_subplot(2, 2, 3, projection='3d')
ax.plot_surface(X, Y, tn.undo_anova_decomposition(tn.mask(anova, tn.only(y))).numpy())
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlim3d(zlim[0], zlim[1])
ax.set_title('$f_y$')

ax = fig.add_subplot(2, 2, 4, projection='3d')
ax.plot_surface(X, Y, tn.undo_anova_decomposition(tn.mask(anova, tn.only(x & y))).
↪numpy())
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlim3d(zlim[0], zlim[1])
ax.set_title('$f_{xy}$')

plt.show()
```



3.3.4 Arithmetics

Basic Arithmetics

The most basic tensor operations (addition +, subtraction −, and product ⋆ with either a scalar or with another tensor) can be accomplished via direct manipulation of tensor cores (see e.g. the original tensor train paper).

[1]:
```
import tntorch as tn
import torch
import numpy as np

t1 = tn.ones([32]*4)
t2 = tn.ones([32]*4)

t = tn.round((t1+t2)*(t2-2))
print(t)
```

```
4D TT tensor:

 32  32  32  32
  |   |   |   |
 (0) (1) (2) (3)
 / \ / \ / \ / \
```

(continues on next page)

(continued from previous page)

```
1   1   1   1   1
```

You can also *assign* values to parts of a tensor:

[2]:
```
t = tn.ones(5, 5)
t[:3, :] = 2
t[:, :2] *= 3
print(t.torch())
```

```
tensor([[6., 6., 2., 2., 2.],
        [6., 6., 2., 2., 2.],
        [6., 6., 2., 2., 2.],
        [3., 3., 1., 1., 1.],
        [3., 3., 1., 1., 1.]])
```

Advanced Operations

Thanks to *cross-approximation*, *tntorch* supports many other more advanced operations on tensors, including element-wise division `/`, `exp()`, `log()`, `sin()`, etc.

[3]:
```
domain = [torch.linspace(0, np.pi, 32)]*4
x, y, z, w = tn.meshgrid(domain)

t = tn.round(1 / (1+x+y+z+w))
print(t)
```

```
4D TT-Tucker tensor:

 32  32  32  32
  |   |   |   |
  7  13  13   7
 (0) (1) (2) (3)
 / \ / \ / \ / \
1   7   7   7   1
```

We will now try the trigonometric identity $\sin^2(x) + \cos^2(x) = 1$:

[4]:
```
t = tn.round(tn.sin(t)**2 + tn.cos(t)**2)
print(t)
```

```
4D TT tensor:

 32  32  32  32
  |   |   |   |
 (0) (1) (2) (3)
 / \ / \ / \ / \
1   13  17  13  1
```

The tensor `t` should be 1 everywhere. Indeed:

[5]:
```
print(tn.mean(t))
print(tn.var(t))
```

```
tensor(1.0000)
tensor(1.8159e-15)
```

3.3.5 Automata

Tensor trains can represent compactly *deterministic finite automata* and *weighted finite automata* that read a fixed number of symbols.

[1]:
```
import torch
import tntorch as tn
```

For instance, `weight_mask` produces an automaton that accepts a string iff it has a certain amount of 1's:

[2]:
```
m = tn.weight_mask(N=4, weight=2)
m
```

[2]:
```
4D TT tensor:

  2   2   2   2
  |   |   |   |
 (0) (1) (2) (3)
 / \ / \ / \ / \
1   2   3   2   1
```

All accepted input strings can be retrieved alphabetically via `accepted_inputs()`:

[3]:
```
tn.accepted_inputs(m)
```

[3]:
```
tensor([[0, 0, 1, 1],
        [0, 1, 0, 1],
        [0, 1, 1, 0],
        [1, 0, 0, 1],
        [1, 0, 1, 0],
        [1, 1, 0, 0]])
```

On the other hand, `weight()` produces an automaton that is a little different. Instead of accepting or rejecting strings, it just counts how many 1's the string has:

[4]:
```
m = tn.weight(N=4)
print(m[0, 0, 0, 0])
print(m[0, 1, 0, 0])
print(m[1, 0, 1, 1])
```

```
tensor(0.)
tensor(1.)
tensor(3.)
```

Applications

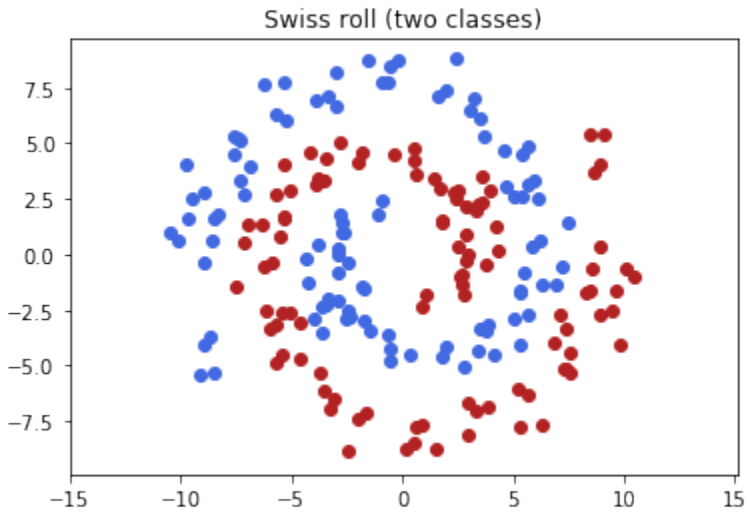
TT automata come in handy to group and sum tensor entries, which is important to obtain advanced *metrics for sensitivity analysis*. See also the tutorial on *Boolean logic with *tntorch**.

3.3.6 Classification

We can model a classifier for C classes with N features using an $(N+1)$-dimensional compressed tensor: the first N dimensions capture all possible feature values, whereas the last one has size C and is used to compute class probabilities.

Here we will try a simple 2-class example in $N = 2$, the Swiss roll classification problem.

[1]:
```
import tntorch as tn
import torch
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

N = 2
C = 2  # Number of classes
P = 100  # Points per class
c0 = torch.rand(P)*8+2
c0 = c0[:, None]
c0 = torch.cat([c0*torch.cos(c0), c0*torch.sin(c0)], dim=1)
c0 += torch.randn(*c0.shape)/1.5
c1 = -c0

plt.figure()
plt.scatter(c0[:, 0], c0[:, 1], color='royalblue')
plt.scatter(c1[:, 0], c1[:, 1], color='firebrick')
plt.gca().set_aspect('equal', 'datalim')
plt.title('Swiss roll (two classes)')
plt.show()
```



[2]:
```
# Assemble (X, y) data set
X = torch.cat([c0, c1], dim=0)
y = torch.cat([torch.zeros(len(c0)), torch.ones(len(c1))])

# Shuffle data
idx = np.random.permutation(len(X))
X = X[idx]
y = y[idx]
```

(continues on next page)

(continued from previous page)

```
# Discretize features into [0, 1, ..., nticks-1]
nticks = 128
X = (X-X.min()) / (X.max()-X.min())
X = X*(nticks-1)

# Split into 75% train / 25% test
ntrain = int(len(X)*0.75)
X_train = X[:ntrain, :].long()
y_train = y[:ntrain].long()
X_test = X[ntrain:, :].long()
y_test = y[ntrain:].long()
```

Let's set up the $128 \times 128 \times 2$ tensor that will be optimized. We will use an expansion using low-frequency cosine wavefunctions:

[3]:
```
t = tn.rand(shape=[nticks]*N + [C], ranks_tt=10, ranks_tucker=6, requires_grad=True)
t.set_factors('dct', dim=range(N))
t
```

[3]:
```
3D TT-Tucker tensor:

 128 128  2
  |   |   |
  6   6   6
 (0) (1) (2)
 / \ / \ / \
1   10  10  1
```

Our tensor's last dimension is 2: for each feature (x_1, x_2) it produces 2 numbers, one per class. For classification we will transform these weights into probabilities using the softmax function:

[4]:
```
def softmax(x):
    expx = torch.exp(x-x.max())
    return expx / torch.sum(expx, dim=-1, keepdim=True)
```

To assess the goodness of a matrix of probabilities (rows are instances, columns are classes) we use the cross-entropy loss:

[5]:
```
def cross_entropy_loss(probs, y):
    return torch.mean(-torch.log(probs[np.arange(len(probs)), y]))
```

We are now ready to fit our tensor network:

[6]:
```
def loss(t):
    return cross_entropy_loss(softmax(t[X_train].torch()), y_train)
tn.optimize(t, loss)
```

```
iter: 0      | loss:   0.707212 | total time:    0.0022
iter: 500    | loss:   0.056675 | total time:    1.3520
iter: 1000   | loss:   0.006464 | total time:    2.7943
iter: 1500   | loss:   0.001936 | total time:    4.1302
iter: 2000   | loss:   0.000841 | total time:    5.5035
iter: 2500   | loss:   0.000438 | total time:    6.8394
iter: 3000   | loss:   0.000254 | total time:    8.1114
iter: 3500   | loss:   0.000157 | total time:    9.4423
iter: 4000   | loss:   0.000102 | total time:   10.7657
iter: 4026   | loss:   0.000100 | total time:   10.8177 <- converged (tol=0.0001)
```

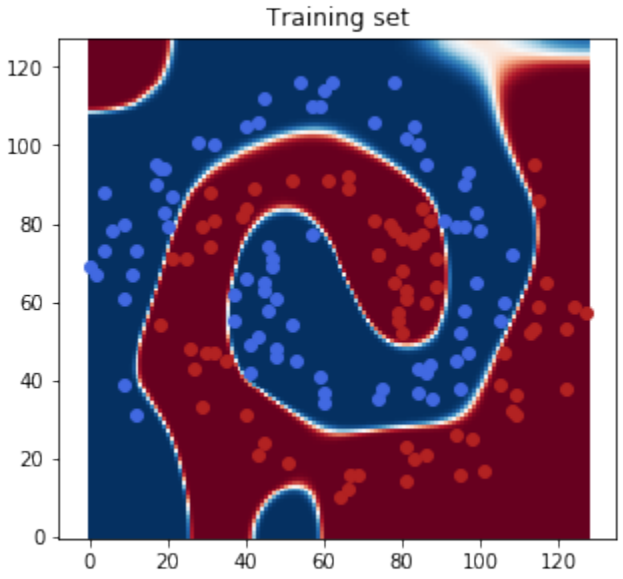
We now predict classes for the test instances and compute the *score* (#correctly classified / number of test instances):

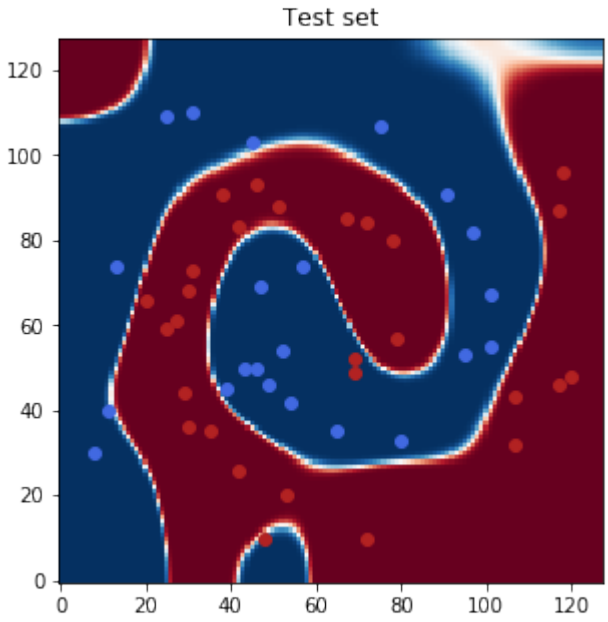
[7]:
```
prediction = torch.max(t[X_test].torch(), dim=1)[1]
score = torch.sum(prediction == y_test).double() / len(y_test)
print('Score:', score)
```

```
Score: tensor(0.9400)
```

Finally, we will show the class probabilities for the whole feature space (blue is class `0`, red is class `1`):

[8]:
```
fig = plt.figure(figsize=(5, 5))
plt.title('Training set')
plt.imshow(softmax(t.torch())[..., 0].detach().numpy().T, origin='lower', cmap='RdBu')
plt.scatter(X_train[y_train==0, 0], X_train[y_train==0, 1], color='royalblue')
plt.scatter(X_train[y_train==1, 0], X_train[y_train==1, 1], color='firebrick')
plt.show()

fig = plt.figure(figsize=(5, 5))
plt.title('Test set')
plt.imshow(softmax(t.torch())[..., 0].detach().numpy().T, origin='lower', cmap='RdBu')
plt.scatter(X_test[y_test==0, 0], X_test[y_test==0, 1], color='royalblue')
plt.scatter(X_test[y_test==1, 0], X_test[y_test==1, 1], color='firebrick')
plt.show()
```





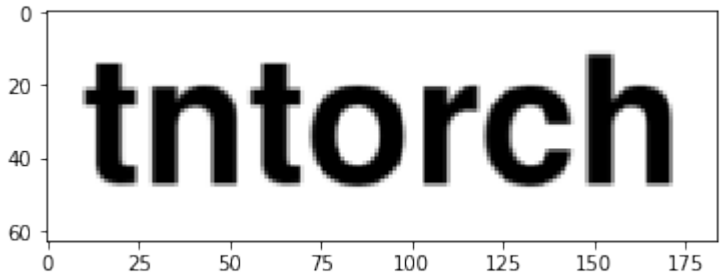
3.3.7 Tensor Completion

Completing a tensor means filling out its missing values. It's the equivalent of interpolation for the case of discretized tensor grids.

[9]:
```
import torch
import tntorch as tn
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
```

We will start by reading a text image and masking out 90% of its pixels:

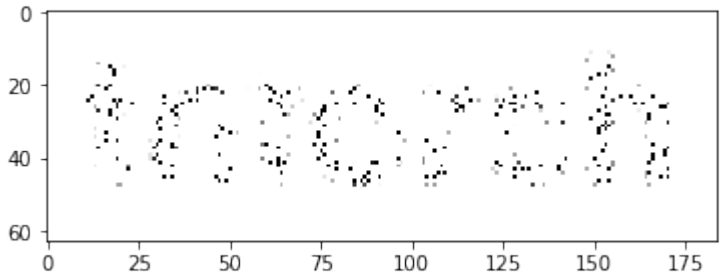
[10]:
```
im = torch.DoubleTensor(plt.imread('../images/text.png'))
plt.imshow(im, cmap='gray', vmin=im.min(), vmax=im.max())
plt.show()

P = im.shape[0]*im.shape[1]
Q = int(P/10)
print('We will keep {} out of {} pixels'.format(Q, P))
X = np.unravel_index(np.random.choice(P, Q), im.shape)  # Coordinates of surviving␣
↪pixels
y = torch.Tensor(im[X])  # Grayscale values of surviving pixels
```




```
We will keep 1159 out of 11592 pixels
```

The masked image looks like this:

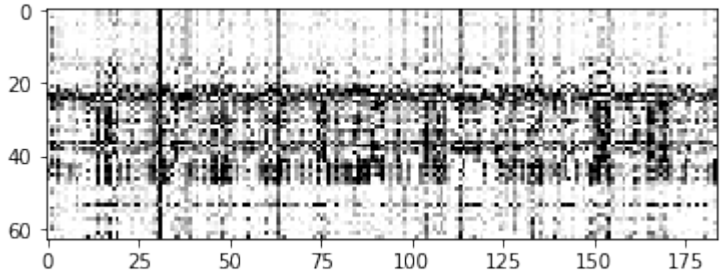
[11]:
```
mask = np.ones([im.shape[0], im.shape[1]])
mask[X] = y
plt.imshow(mask, cmap='gray', vmin=im.min(), vmax=im.max())
plt.show()
```



Now, we will try to recover the image by completing a rank-6 tensor with those samples:

[12]:
```
t = tn.rand(im.shape, ranks_tt=6, requires_grad=True)

def loss(t):
    return tn.relative_error(y, t[X])
tn.optimize(t, loss)

plt.imshow(t.numpy(), cmap='gray', vmin=im.min(), vmax=im.max())
plt.show()
```

```
iter: 0      | loss:   1.178050 | total time:    0.0009
iter: 500    | loss:   0.200007 | total time:    0.5207
iter: 1000   | loss:   0.084521 | total time:    1.0735
iter: 1500   | loss:   0.021333 | total time:    1.6547
iter: 2000   | loss:   0.003651 | total time:    2.2765
iter: 2187   | loss:   0.001562 | total time:    2.5357 <- converged (tol=0.0001)
```



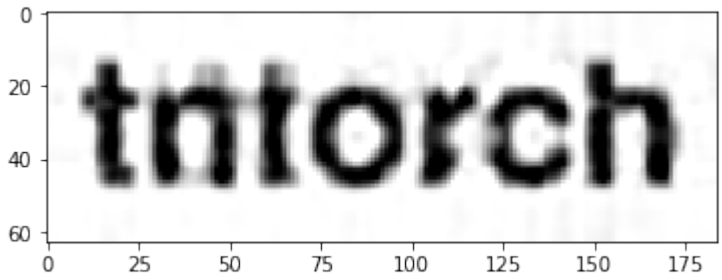
The result is not convincing since we are not using any notion of spatial correlation: the real world signal is smooth, but our tensor does not know about that.

Smoothness Priors

This time we will add a penalization term in the hope of getting a smoother reconstruction. We will use the norm of the tensor's 2nd-order derivatives, for which we can use *the function *partialset()**. To combine (add) both losses, we need to return them as a tuple from the `loss()` function:

[13]:
```
t = tn.rand(im.shape, ranks_tt=6, requires_grad=True)

def loss(t):
    return tn.relative_error(y, t[X])**2, tn.normsq(tn.partialset(t, order=2))*1e-4
tn.optimize(t, loss)

plt.imshow(t.numpy(), cmap='gray', vmin=im.min(), vmax=im.max())
plt.show()
```

```
iter: 0       | loss:   1.172050 +   2.569117 =      3.741 | total time:    0.0037
iter: 500     | loss:   0.120569 +   0.085111 =     0.2057 | total time:    2.4681
iter: 1000    | loss:   0.063130 +   0.021324 =    0.08445 | total time:    4.9900
iter: 1500    | loss:   0.053652 +   0.010877 =    0.06453 | total time:    7.6926
iter: 2000    | loss:   0.046964 +   0.008748 =    0.05571 | total time:   10.2740
iter: 2500    | loss:   0.037407 +   0.008513 =    0.04592 | total time:   12.8340
iter: 3000    | loss:   0.025241 +   0.007971 =    0.03321 | total time:   15.2129
iter: 3500    | loss:   0.015514 +   0.006649 =    0.02216 | total time:   17.7566
iter: 4000    | loss:   0.010046 +   0.005578 =    0.01562 | total time:   19.9821
iter: 4500    | loss:   0.006938 +   0.005028 =    0.01197 | total time:   22.7111
iter: 5000    | loss:   0.005096 +   0.004723 =   0.009818 | total time:   25.3216
iter: 5500    | loss:   0.004118 +   0.004510 =   0.008628 | total time:   28.6819
iter: 6000    | loss:   0.003574 +   0.004364 =   0.007938 | total time:   33.3063
iter: 6500    | loss:   0.003203 +   0.004258 =   0.007461 | total time:   37.6611
iter: 6542    | loss:   0.003178 +   0.004251 =   0.007429 | total time:   38.1314 <-
↪converged (tol=0.0001)
```



3.3.8 Cross-approximation

Often, we would like to build a N-dimensional tensor from a **black-box function** $f : \Omega \subset \mathbb{R}^N \to \mathbb{R}$, where Ω is a tensor product grid. That is, we are free to sample *whatever entries we want* within our domain Ω, but we cannot afford to sample the entire domain (as it contains an **exponentially large number of points**). One way to build such a tensor is using **cross-approximation** (I. Oseledets, E. Tyrtyshnikov: "TT-cross Approximation for Multidimensional Arrays") from well-chosen fibers in the domain.

We support two major use cases of cross-approximation in the TT format.

Approximating a Function over a Domain

This is the more basic setting. We just need to specify:

- Our function of interest
- The tensor product domain $\Omega = u_1 \otimes \cdots \otimes u_N$

[1]:
```
import tntorch as tn
import torch

def function(x, y, z, t, w):  # Input arguments are vectors
    return 1 / (x + y + z + t + w)  # Hilbert tensor

domain = [torch.arange(1, 33) for n in range(5)]
t = tn.cross(function=function, domain=domain)

print(t)
```

```
Cross-approximation over a 5D domain containing 3.35544e+07 grid points:
iter: 0  | eps: 9.221e-01 | total time:   0.0110 | largest rank:   1
iter: 1  | eps: 4.867e-03 | total time:   0.0350 | largest rank:   4
iter: 2  | eps: 4.295e-06 | total time:   0.0609 | largest rank:   7
iter: 3  | eps: 8.606e-09 | total time:   0.1027 | largest rank:  10 <- converged:␣
↪eps < 1e-06
Did 33984 function evaluations, which took 0.001594s (2.133e+07 evals/s)

5D TT tensor:

 32  32  32  32  32
  |   |   |   |   |
 (0) (1) (2) (3) (4)
 / \ / \ / \ / \ / \
1   10  10  10  10  1
```

Sometimes it's more convenient to work with functions that accept matrices (instead of a list of vectors) as input. We can do this with the `function_arg='matrix'` flag:

[2]:
```
def function(Xs):  # Matrix (one row per sample, one column per input variable) and␣
↪return a vector with one result per sample
    return 1/torch.sum(Xs, dim=1)

t = tn.cross(function=function, domain=domain, function_arg='matrix')
```

```
Cross-approximation over a 5D domain containing 3.35544e+07 grid points:
iter: 0  | eps: 9.355e-01 | total time:   0.0138 | largest rank:   1
iter: 1  | eps: 4.148e-03 | total time:   0.0341 | largest rank:   4
iter: 2  | eps: 5.244e-06 | total time:   0.0610 | largest rank:   7
iter: 3  | eps: 7.581e-09 | total time:   0.0961 | largest rank:  10 <- converged:␣
↪eps < 1e-06
Did 33984 function evaluations, which took 0.00437s (7.777e+06 evals/s)
```

Element-wise Operations on Tensors

Here we have one (or several) N-dimensional tensors that we want to transform element-wise. For instance, we may want to square each element of our tensor:

[3]:
```
t2 = tn.cross(function=lambda x: x**2, tensors=t)
```

```
Cross-approximation over a 5D domain containing 3.35544e+07 grid points:
iter: 0  | eps: 9.539e-01 | total time:   0.0062 | largest rank:   1
iter: 1  | eps: 2.066e-02 | total time:   0.0174 | largest rank:   4
iter: 2  | eps: 5.644e-05 | total time:   0.0338 | largest rank:   7
iter: 3  | eps: 6.255e-08 | total time:   0.0627 | largest rank:  10 <- converged:␣
↪eps < 1e-06
Did 33984 function evaluations, which took 0.0005157s (6.59e+07 evals/s)
```

Just for practice, let's do this now in a slightly different way by passing two tensors:

[4]:
```
t2 = tn.cross(function=lambda x, y: x*y, tensors=[t, t])
```

```
Cross-approximation over a 5D domain containing 3.35544e+07 grid points:
iter: 0  | eps: 9.757e-01 | total time:   0.0081 | largest rank:   1
iter: 1  | eps: 2.939e-02 | total time:   0.0228 | largest rank:   4
iter: 2  | eps: 1.086e-04 | total time:   0.0440 | largest rank:   7
iter: 3  | eps: 8.331e-08 | total time:   0.0675 | largest rank:  10 <- converged:␣
↪eps < 1e-06
Did 33984 function evaluations, which took 0.0005171s (6.572e+07 evals/s)
```

Let's check the accuracy of our cross-approximated squaring operation, compared to the groundtruth `t*t`:

[5]:
```
tn.relative_error(t*t, t2)
```

[5]:
```
tensor(8.6986e-08)
```

See *this notebook* for more examples on element-wise tensor operations.

3.3.9 Tensor Decompositions

The philosophy of *tntorch* is simple: **one class for all formats**. *Different decompositions* (CP, Tucker, TT, hybrids) all use the same interface.

Note: sometimes the internal format will change automatically. For example, no recompression algorithm is known for the CP format, and running ''round()'' on a CP tensor will convert it to the TT format.

We will give a few examples of how to compress a full tensor into different tensor formats.

[1]:
```
import tntorch as tn
import torch
import time

import numpy as np
X, Y, Z = np.meshgrid(range(128), range(128), range(128))
full = torch.Tensor(np.sqrt(np.sqrt(X)*(Y+Z) + Y*Z**2)*(X + np.sin(Y)*np.cos(Z)))  #␣
↪Some analytical 3D function
print(full.shape)
```

```
torch.Size([128, 128, 128])
```

TT

To compress as a low-rank tensor train (TT), use the `ranks_tt` argument:

[2]:
```
t = tn.Tensor(full, ranks_tt=3)  # You can also pass a list of ranks

def metrics():
    print(t)
    print('Compression ratio: {}/{} = {:g}'.format(full.numel(), t.numel(), full.
↪numel() / t.numel()))
    print('Relative error:', tn.relative_error(full, t))
    print('RMSE:', tn.rmse(full, t))
    print('R^2:', tn.r_squared(full, t))

metrics()
```

```
3D TT tensor:

 128 128 128
  |   |   |
 (0) (1) (2)
 / \ / \ / \
1   3   3   1

Compression ratio: 2097152/1920 = 1092.27
Relative error: tensor(0.0005)
RMSE: tensor(22.0745)
R^2: tensor(1.0000)
```

The TT cores are available as `t.cores`.

Tucker

Use the `ranks_tucker` argument:

[3]:
```
t = tn.Tensor(full, ranks_tucker=3)
metrics()
```

```
3D TT-Tucker tensor:

 128 128 128
  |   |   |
  3   3   3
 (0) (1) (2)
 / \ / \ / \
1   9   3   1

Compression ratio: 2097152/1269 = 1652.6
Relative error: tensor(0.0005)
RMSE: tensor(22.0752)
R^2: tensor(1.0000)
```

Even though technically a TT-Tucker tensor, it has the *exact same expressive power* as a low-rank Tucker decomposition.

The Tucker factors are `t.Us`. To retrieve the full Tucker core, use `tucker_core()`:

[4]:
```
t.tucker_core().shape
```

[4]:
```
torch.Size([3, 3, 3])
```

CP

Use the `ranks_cp` argument:

[5]:
```
t = tn.Tensor(full, ranks_cp=3, verbose=True)  # CP is computed using alternating␣
↪least squares (ALS)
metrics()
```

```
ALS - initialization time = 0.045638084411621094
iter: 0  | eps: 0.00098631 | total time:    0.0682
iter: 1  | eps: 0.00092816 | total time:    0.0896 <- converged (tol=0.0001)
3D CP tensor:

 128 128 128
  |   |   |
 <0> <1> <2>
 / \ / \ / \
3   3   3   3

Compression ratio: 2097152/1152 = 1820.44
Relative error: tensor(0.0009)
RMSE: tensor(39.9936)
R^2: tensor(1.0000)
```

The CP factors are `t.cores` (they are all 2D tensors).

Hybrid Formats

`ranks_tucker` can be combined with the other arguments to produce hybrid decompositions:

[6]:
```
t = tn.Tensor(full, ranks_cp=3, ranks_tucker=3)
metrics()
t = tn.Tensor(full, ranks_tt=2, ranks_tucker=4)
metrics()
```

```
3D CP-Tucker tensor:

 128 128 128
  |   |   |
  3   3   3
 <0> <1> <2>
 / \ / \ / \
3   3   3   3

Compression ratio: 2097152/1179 = 1778.75
Relative error: tensor(0.0035)
RMSE: tensor(149.4028)
R^2: tensor(1.0000)
3D TT-Tucker tensor:

 128 128 128
  |   |   |
  4   4   4
 (0) (1) (2)
 / \ / \ / \
1   2   2   1

Compression ratio: 2097152/1568 = 1337.47
```

(continues on next page)

(continued from previous page)

```
Relative error: tensor(0.0012)
RMSE: tensor(51.8083)
R^2: tensor(1.0000)
```

Error-bounded Decompositions

If you instead pass the argument `eps`, a decomposition will be computed that will not exceed that relative error:

[7]:
```
t = tn.Tensor(full, eps=1e-5)
metrics()
```

```
3D TT-Tucker tensor:

 128 128 128
  |   |   |
  4   5   6
 (0) (1) (2)
 / \ / \ / \
1   4   6   1

Compression ratio: 2097152/2092 = 1002.46
Relative error: tensor(8.3402e-06)
RMSE: tensor(0.3594)
R^2: tensor(1.0000)
```

That will always try to compress in both Tucker and TT senses, and therefore will always produce a TT-Tucker tensor. If you only want to compress, say, in the Tucker sense, you can do:

[8]:
```
t = tn.Tensor(full)
t.round_tucker(eps=1e-5)
metrics()
```

```
3D TT-Tucker tensor:

 128 128 128
  |   |   |
  5   4   7
 (0) (1) (2)
 / \ / \ / \
1   28  7   1

Compression ratio: 2097152/3021 = 694.191
Relative error: tensor(4.0447e-06)
RMSE: tensor(0.1743)
R^2: tensor(1.0000)
```

And conversely, for a TT-only compression:

[9]:
```
t = tn.Tensor(full)
t.round_tt(eps=1e-5)
metrics()
```

```
3D TT tensor:

 128 128 128
  |   |   |
```

(continues on next page)

(continued from previous page)

```
 (0) (1) (2)
 / \ / \ / \
1   4   6   1

Compression ratio: 2097152/4352 = 481.882
Relative error: tensor(8.3358e-06)
RMSE: tensor(0.3592)
R^2: tensor(1.0000)
```

3.3.10 Differentiation

To derive a tensor network one just needs to derive each core along its spatial dimension (if it has one).

[1]:
```
import torch
import tntorch as tn

t = tn.rand([32]*3, ranks_tt=3, requires_grad=True)
t
```

[1]:
```
3D TT tensor:

 32  32  32
  |   |   |
 (0) (1) (2)
 / \ / \ / \
1   3   3   1
```

Basic Derivatives

To derive w.r.t. one or several variables, use `partial()`:

[2]:
```
tn.partial(t, dim=[0, 1], order=2)
```

[2]:
```
3D TT tensor:

 32  32  32
  |   |   |
 (0) (1) (2)
 / \ / \ / \
1   3   3   1
```

Many Derivatives at Once

Thanks to *mask tensors* we can specify and consider groups of many derivatives at once using the function `partialset()`. For example, the following tensor encodes *all* 2nd-order derivatives that contain x:

[3]:
```
x, y, z = tn.symbols(t.dim())
d = tn.partialset(t, order=2, mask=x)
print(d)
```

```
3D TT tensor:

 96  96  96
```

(continues on next page)

(continued from previous page)

```
  |   |   |
 (0) (1) (2)
 / \ / \ / \
1   9   9   1
```

We can check by summing squared norms:

[4]:
```
print(tn.normsq(d))
print(tn.normsq(tn.partial(t, 0, order=2)) + tn.normsq(tn.partial(t, [0, 1],␣
↪order=1)) + tn.normsq(tn.partial(t, [0, 2], order=1)))
```

```
tensor(48342.2888, grad_fn=<SumBackward0>)
tensor(48342.2888, grad_fn=<ThAddBackward>)
```

The method with masks is attractive because its cost scales linearly with dimensionality N. Computing all order-O derivatives costs $O(NO^3R^2)$ with `partialset()` vs. $O(N^{(O+1)}R^2)$ with the naive `partial()`.

Applications

See *this notebook* for an example of tensor optimization that tries to maximize an interpolator's smoothness. Tensor derivatives are also used for some *vector field* computations and in the *active subspace method*.

3.3.11 Exponential Machines

Exponential machines (Novikov et al., 2017) are predictors that are able to model all 2^N interactions between N features. Those interactions are limited to being of degree 1, i.e. the regressor is of the form

$$f(x_1, \ldots, x_N) = \sum_{\alpha \subseteq \{1,\ldots,N\}} \left(w_\alpha \cdot \prod_{n \in \alpha} x_n \right)$$

and fitting the regressor means finding the best w_α for all 2^N possible α's. Besides the L^2 loss on the training data, the original paper also adds an L^2 regularization term ("ridge") on the set of weights w.

We note that an exponential machine is equivalent to a TT-Tucker model with Tucker rank equal to 2: the first column of the Tucker factors should be a constant (1), and the second column should be linear (x). To this end, we can use for example a Legendre expansion of polynomials truncated to 2 basis elements. Note that this can be seen as a particular case of a *polynomial chaos expansion*.

We will try here a synthetic model with noise: $f^{\text{true}}(x_1, \ldots, x_5) = x_1x_2x_3 + x_1x_2x_3x_4x_5 + \varepsilon$:

[1]:
```
import torch
import tntorch as tn

P = 100
ntrain = int(P*0.75)
N = 5
ticks = 32  # We will use a 32^5 tensor

X = torch.rand(P, N)*2 - 1  # Features between -1 and 1
y = torch.prod(X, dim=1) + torch.prod(X[:, :3], dim=1)
y += torch.randn(y.shape)*torch.std(y)/10  # Gaussian noise: 1/5th of the clean signal
↪'s sigma
X = (X+1)/2*(ticks-1)  # Make feature between 0 and ticks-1, i.e. indexable by the␣
↪tensor
```

(continues on next page)

(continued from previous page)

```
# Split into train/test
X_train = X[:ntrain]
y_train = y[:ntrain]
X_test = X[ntrain:]
y_test = y[ntrain:]
```

A TT-rank of 2 should be enough to fit this data set, since it arises from the sum of exactly two interactions.

[2]:
```
t = tn.rand(shape=[ticks]*N, ranks_tt=2, ranks_tucker=2, requires_grad=True)
t.set_factors('legendre', requires_grad=False)  # We set the factors to Legendre␣
↪polynomials, and fix them (won't be changed during optimization)
```

[3]:
```
def loss(t):
    return tn.relative_error(y_train, t[X_train])**2
tn.optimize(t, loss)
```

```
iter: 0      | loss:   7.956270 | total time:    0.0022
iter: 500    | loss:   0.475733 | total time:    0.9688
iter: 1000   | loss:   0.071103 | total time:    1.8799
iter: 1500   | loss:   0.024970 | total time:    2.8605
iter: 2000   | loss:   0.013503 | total time:    3.9351
iter: 2500   | loss:   0.009449 | total time:    5.0269
iter: 3000   | loss:   0.007776 | total time:    6.1072
iter: 3500   | loss:   0.007118 | total time:    7.1863
iter: 3532   | loss:   0.007094 | total time:    7.2544 <- converged (tol=0.0001)
```

[4]:
```
print('Test relative error:', tn.relative_error(y_test, t[X_test]))
```

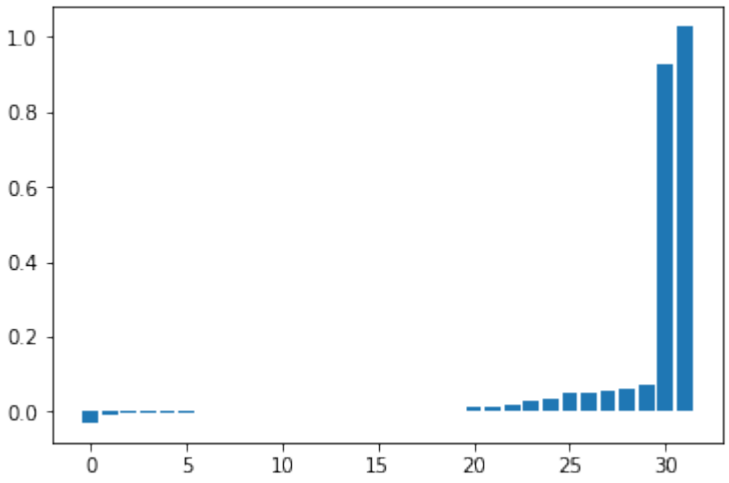
```
Test relative error: tensor(0.1051, grad_fn=<DivBackward1>)
```

The tensor of weights ($\mathcal{W}$ in the original paper) can be retrieved as the TT part of our TT-Tucker tensor:

[5]:
```
core = tn.Tensor(t.cores)
core
```

[5]:
```
5D TT tensor:

  2   2   2   2   2
  |   |   |   |   |
 (0) (1) (2) (3) (4)
 / \ / \ / \ / \ / \
1   2   2   2   2   1
```

[6]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

plt.figure()
plt.bar(np.arange(core.size), np.sort(core.numpy().flatten()))
plt.show()
```



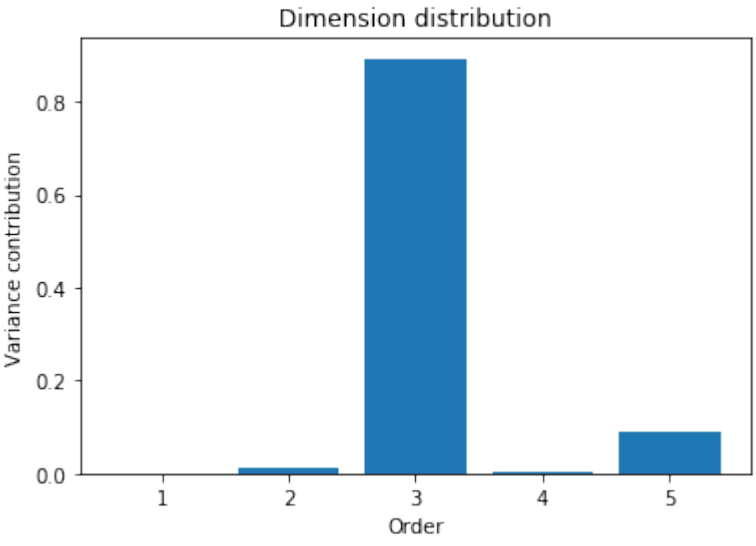
Despite the noise, the model correctly retrieved the two interacting components. Those indeed correspond to $x_1x_2x_3$ and $x_1x_2x_3x_4x_5$:

[7]:
```
print(core[1, 1, 1, 0, 0])
print(core[1, 1, 1, 1, 1])
```

```
tensor(1.0280, grad_fn=<SqueezeBackward0>)
tensor(0.9253, grad_fn=<SqueezeBackward0>)
```

The orders of participating interactions (namely, 3 and 5) can be revealed as well by the *dimension distribution*:

[8]:
```
plt.figure()
plt.bar(np.arange(1, N+1), tn.dimension_distribution(t).numpy())
plt.xlabel('Order')
plt.ylabel('Variance contribution')
plt.title('Dimension distribution')
plt.show()
```



3.3.12 Boolean Logic

Tensor networks make for a great representation of Boolean expressions. Let's have a look at how to build tensor Boolean formulas and what we can do with them.

[1]:
```
import tntorch as tn
import torch

p, q, r = tn.symbols(3)
```

These symbols are plain *tntorch* tensors (in this case, 3D ones) and have rank 1 each:

[2]:
```
p
```

[2]:
```
3D TT tensor:

  2   2   2
  |   |   |
 (0) (1) (2)
 / \ / \ / \
1   1   1   1
```

The fun starts with overloaded Boolean operators:

[3]:
```
tn.is_tautology(p | ~p)  # Formula is always true
```

[3]:
```
True
```

[4]:
```
tn.is_contradiction(p & ~p)  # Formula is always false
```

[4]:
```
True
```

[5]:
```
tn.is_satisfiable(p ^ q | r)  # Formula is true for some values
```

[5]:
```
True
```

[6]:
```
tn.implies(p & q | q & r, q)  # First formula implies the second
```

[6]:
```
True
```

[7]:
```
tn.equiv(p & q, ~(~p | ~q))  # A double implication (De Morgan's law)
```

[7]:
```
True
```

Check out the quantifiers too:

[8]:
```
tn.equiv(tn.all(3), p & q & r)  # "for all"
```

[8]:
```
True
```

[9]:
```
tn.equiv(tn.any(3), p | q | r)  # "exists"
```

[9]:
```
True
```

[10]:
```
tn.equiv(tn.none(3), ~tn.any(3))  # "does not exist"
```

[10]:
```
True
```

[11]:
```
tn.equiv(tn.one(3), p&~q&~r | ~p&q&~r | ~p&~q&r)  # "exists and is unique"
```

[11]:
```
True
```

To find out the relevant symbols of a formula (those whose values can affect its result), do the following:

[12]:
```
tn.relevant_symbols(p & q | ~p & q)
```

[12]:
```
[1]
```

The ranks grow exponentially after each binary operation. It is recommended to run `round()` to try to reduce those ranks, often saving up memory and computational costs:

[13]:
```
formula = ~p | (p & q) | (q & r)
print(formula)
formula.round()
print(formula)
```

```
3D TT tensor:

  2   2   2
  |   |   |
 (0) (1) (2)
 / \ / \ / \
1   11  11  1

3D TT tensor:

  2   2   2
  |   |   |
 (0) (1) (2)
 / \ / \ / \
1   2   2   1
```

How many ways are there to satisfy a given Boolean formula? This is known as the #SAT problem. For us, it's just the sum of elements of the tensor:

[14]:
```
tn.sum(formula)
```

[14]:
```
tensor(6.0000)
```

We can look at all the inputs that satisfy this formula with the function `accepted_inputs()`:

[15]:
```
tn.accepted_inputs(formula)  # One row per accepted input
```

[15]:
```
tensor([[0, 0, 0],
        [0, 0, 1],
        [0, 1, 0],
        [0, 1, 1],
        [1, 1, 0],
        [1, 1, 1]])
```

The function `only()` is useful to enforce that *only* certain variables play a role:

[16]:
```
tn.accepted_inputs(tn.only(p) | tn.only(r))
```

[16]:
```
tensor([[0, 0, 1],
        [1, 0, 0]])
```

Applications

Boolean tensors can be applied in *ANOVA* and *variance-based sensitivity analysis* as well as for compact *derivative computation*.

3.3.13 Main Tensor Formats

Three of the most popular tensor decompositions are supported in *tntorch*:

- CANDECOMP/PARAFAC (CP)
- Tucker
- Tensor Train (TT)

Those formats are all represented using N *tensor cores* (one per tensor dimension, used for CP/TT) and, optionally, up to N *factor matrices* (needed for Tucker).

In an N-D tensor of shape $I_1 \times \cdots \times I_N$, each n-th core can come in four flavors:

- $R_n^{\text{TT}} \times I_n \times R_{n+1}^{\text{TT}}$: a TT core.
- $R_n^{\text{TT}} \times S_n^{\text{Tucker}} \times R_{n+1}^{\text{TT}}$: a TT-Tucker core, accompanied by an $I_n \times S_n^{\text{Tucker}}$ factor matrix.
- $I_n \times R_n^{\text{CP}}$: a CP core. Conceptually, it works as if it were a 3D TT core of shape $R_n^{\text{CP}} \times I_n \times R_n^{\text{CP}}$ whose slices along the 2nd mode are all diagonal matrices.
- $S_n^{\text{Tucker}} \times R_n^{\text{CP}}$: a CP-Tucker core, accompanied by an $I_n \times S_n^{\text{Tucker}}$ factor matrix. Conceptually, it works as a 3D TT-Tucker core.

One tensor network can combine cores of different kinds. So all in all one may have TT, TT-Tucker, Tucker, CP, TT-CP, CP-Tucker, and TT-CP-Tucker tensors. We will show examples of all.

(see 'this notebook <decompositions.ipynb>'__ to decompose full tensors into those main formats)

(see 'this notebook <other_formats.ipynb>'__ for other structured and custom decompositions)

TT

Tensor train cores are represented in parentheses `( )`:

[1]:
```
import tntorch as tn

tn.rand([32]*5, ranks_tt=5)
```

[1]:
```
5D TT tensor:

 32  32  32  32  32
  |   |   |   |   |
 (0) (1) (2) (3) (4)
 / \ / \ / \ / \ / \
1   5   5   5   5   1
```

TT-Tucker

In this format, TT cores are compressed along their spatial dimension (2nd mode) using an accompanying Tucker factor. This was considered e.g. in the original TT paper.

[2]:
```
tn.rand([32]*5, ranks_tt=5, ranks_tucker=6)
```

[2]:
```
5D TT-Tucker tensor:

 32  32  32  32  32
  |   |   |   |   |
  6   6   6   6   6
 (0) (1) (2) (3) (4)
 / \ / \ / \ / \ / \
1   5   5   5   5   1
```

Here is an example where only *some* cores have Tucker factors:

[3]:
```
tn.rand([32]*5, ranks_tt=5, ranks_tucker=[None, 6, None, None, 7])
```

[3]:
```
5D TT-Tucker tensor:

     32          32
 32   |  32  32   |
  |   6   |   |   7
 (0) (1) (2) (3) (4)
 / \ / \ / \ / \ / \
1   5   5   5   5   1
```

Note: if you want to leave some factors fixed during gradient descent, simply set them to some PyTorch tensor that has requires_grad=False.

Tucker

"Pure" Tucker is technically speaking not supported, but is equivalent to a TT-Tucker tensor with full TT-ranks. The minimal necessary ranks are automatically computed and set up for you:

[4]:
```
tn.rand([32]*5, ranks_tucker=3)  # Actually a TT-Tucker network, but just as␣
↪expressive as a pure Tucker decomposition
```

[4]:
```
5D TT-Tucker tensor:

 32  32  32  32  32
  |   |   |   |   |
  3   3   3   3   3
 (0) (1) (2) (3) (4)
 / \ / \ / \ / \ / \
1   3   9   9   3   1
```

In other words, all 32^5 tensors of Tucker rank 3 can be represented by a tensor that has the shape shown above, and vice versa.

CP

CP factors are shown as cores in brackets < >:

[5]:
```
tn.rand([32]*5, ranks_cp=4)
```

[5]:
```
5D CP tensor:

 32  32  32  32  32
```

(continues on next page)

(continued from previous page)

```
  |   |   |   |   |
 <0> <1> <2> <3> <4>
 / \ / \ / \ / \ / \
4   4   4   4   4   4
```

Even though those factors work conceptually as 3D cores in a tensor train (every CP tensor is a particular case of the TT format), they are stored in 2D as in a standard CP decomposition. In this case all cores have shape 32×4.

TT-CP

TT and CP cores can be combined by specifying lists of ranks for each format. You should provide $N-1$ TT ranks and N CP ranks and use `None` so that they do not collide anywhere. Also note that consecutive CP ranks must coincide. Here is a tensor with 3 TT cores and 2 CP cores:

[6]:
```
tn.rand([32]*5, ranks_tt=[2, 3, None, None], ranks_cp=[None, None, None, 4, 4])
```

[6]:
```
5D TT-CP tensor:

 32  32  32  32  32
  |   |   |   |   |
 (0) (1) (2) <3> <4>
 / \ / \ / \ / \ / \
1   2   3   4   4   4
```

Here is another example with 2 TT cores and 3 CP cores:

[7]:
```
tn.rand([32]*5, ranks_tt=[None, 2, None, None], ranks_cp=[4, None, None, 5, 5])
```

[7]:
```
5D TT-CP tensor:

 32  32  32  32  32
  |   |   |   |   |
 <0> (1) (2) <3> <4>
 / \ / \ / \ / \ / \
4   4   2   5   5   5
```

CP-Tucker

Similarly to TT-Tucker, this model restricts the columns along CP factors to live in a low-dimensional subspace. It is also known as a canonical decomposition with linear constraints (*CANDELINC*). Compressing a Tucker core via CP leads to an equivalent format.

[8]:
```
tn.rand([32]*5, ranks_cp=2, ranks_tucker=4)
```

[8]:
```
5D CP-Tucker tensor:

 32  32  32  32  32
  |   |   |   |   |
  4   4   4   4   4
 <0> <1> <2> <3> <4>
 / \ / \ / \ / \ / \
2   2   2   2   2   2
```

TT-CP-Tucker

Finally, we can combine all sorts of cores to get a hybrid of all 3 models:

[9]:
```
tn.rand([32]*5, ranks_tt=[2, 3, None, None], ranks_cp=[None, None, None, 10, 10],␣
↪ranks_tucker=[5, None, 5, 5, None])
```

[9]:
```
5D TT-CP-Tucker tensor:

 32       32  32
  |  32    |   |  32
  5   |    5   5   |
 (0) (1) (2) <3> <4>
 / \ / \ / \ / \ / \
1   2   3   10  10  10
```

3.3.14 Other Tensor Formats

Besides the *natively supported formats*, you can use *tntorch* to emulate other structured tensor decompositions (or at least, some of their functionality).

Reference: all the following models are surveyed in *"Tensor Decompositions and Applications"*, by Kolda and Bader (2009).

INDSCAL

Individual differences in scaling (INDSCAL) is just a 3D CP decomposition with two shared factors, say the first two.

[1]:
```
import tntorch as tn
import torch


def INDSCAL(shape, rank):

    assert len(shape) == 3
    assert shape[0] == shape[1]

    A = torch.randn(shape[0], rank, requires_grad=True)
    B = A  # The first two cores share the same memory
    C = torch.randn(shape[2], rank, requires_grad=True)

    return tn.Tensor([A, B, C])

t = INDSCAL([10, 10, 64], 8)
print(t)
print(tn.mean(t))
```

```
3D CP tensor:

 10  10  64
  |   |   |
 <0> <1> <2>
 / \ / \ / \
8   8   8   8

tensor(0.0559, grad_fn=<DivBackward1>)
```

This tensor's two first factors are the same PyTorch tensor in memory. So if we optimize (fit) the tensor they will stay the same, as is desirable.

CANDELINC

CANDELINC (*canonical decomposition with linear constraints*) is a CP decomposition such that each factor is compressed along its columns by an additional given matrix (the *linear constraints*). In other words, it is a CP-Tucker format with fixed Tucker factors.

[2]:
```
def CANDELINC(rank, constraints):  # `constraints` are N In x Sn matrices encoding␣
↪the linear constraints for the N CP factors
    cores = [torch.randn(c.shape[1], rank, requires_grad=True) for c in constraints]
    return tn.Tensor(cores, constraints)

N = 3
rank = 3
constraints = [torch.randn(10, 5), torch.randn(11, 6), torch.randn(12, 7)]
CANDELINC(rank, constraints)
```

[2]:
```
3D CP-Tucker tensor:

 10  11  12
  |   |   |
  5   6   7
 <0> <1> <2>
 / \ / \ / \
3   3   3   3
```

DEDICOM

In three-way *decomposition into directional components* (DEDICOM), 5 factors interact to encode a 3D tensor (2 of those factors are repeated). All factors use the same rank.

[3]:
```
def DEDICOM(shape, rank):

    assert len(shape) == 3
    assert shape[0] == shape[2]

    A = torch.randn(shape[0], rank, requires_grad=True)
    D = torch.randn(shape[1], rank, requires_grad=True)
    R = torch.randn(rank, 1, rank, requires_grad=True)

    return tn.Tensor([A, D, R, D, A])

DEDICOM([10, 64, 10], 8)
```

[3]:
```
5D TT-CP tensor:

 10  64   1  64  10
  |   |   |   |   |
 <0> <1> (2) <3> <4>
 / \ / \ / \ / \ / \
8   8   8   8   8   8
```

Note that this tensor is to be accessed via a special pattern (`t[i, j, k]` should be written as `t[i, j, 0, j, k]`). Some routines (e.g. `numel()`, `torch()`, `norm()`, etc.) that are unaware of this special structure will not work properly.

PARATUCK2

PARATUCK2 is a variant of DEDICOM in which no factors are repeated, and two different ranks intervene.

[4]:
```
def PARATUCK2(shape, ranks):

    assert len(shape) == 3
    assert shape[0] == shape[2]
    assert len(ranks) == 2

    A = torch.randn(shape[0], ranks[0], requires_grad=True)
    DA = torch.randn(shape[1], ranks[0], requires_grad=True)
    R = torch.randn(ranks[0], 1, ranks[1], requires_grad=True)
    DB = torch.randn(shape[1], ranks[1], requires_grad=True)
    B = torch.randn(shape[2], ranks[1], requires_grad=True)

    return tn.Tensor([A, DA, R, DB, B])

PARATUCK2([10, 64, 10], [7, 8])
```

[4]:
```
5D TT-CP tensor:

 10  64   1  64  10
  |   |   |   |   |
 <0> <1> (2) <3> <4>
 / \ / \ / \ / \ / \
7   7   7   8   8   8
```

3.3.15 Polynomial Chaos Expansions

The main idea behind PCE is to search an interpolator that lives in the subspace of low-degree polynomials (or rather, in the tensor product of such subspaces).

In this notebook we will tackle a regression problem: we have N features that are used to train an N-dimensional TT model. Each feature x_n is mapped to the n-th entry of the tensor; in other words, we learn a function as a discretized tensor:

$$f(x) = f(x_0, \dots, x_{N-1}) \approx \mathcal{T}[i_0, \dots, i_{N-1}]$$

Here we will use a noisy 5D synthetic function: $f(x) := \sum w_n x_n^2 + \epsilon$ where the w_n are random weights and ϵ is Gaussian noise added to every observation.

[1]:
```
import torch
import tntorch as tn

P = 200
ntrain = int(P*0.75)
N = 5
ticks = 32  # We will use a 32^5 tensor

X = torch.randint(0, ticks, (P, N))  # Make features be between 0 and ticks-1 (will␣
↪be used directly as tensor indices)
ws = torch.rand(N)
y = torch.matmul(X**2, ws)
y += torch.randn(y.shape)*torch.std(y)/10  # Gaussian noise: 1/10th of the clean␣
↪signal's sigma
```

(continues on next page)

(continued from previous page)

```
# Split into train/test
X_train = X[:ntrain]
y_train = y[:ntrain]
X_test = X[ntrain:]
y_test = y[ntrain:]
```

Our first attempt will be to learn this problem using only the low rank assumption, i.e. plain *tensor completion*:

[2]:
```
t = tn.rand(shape=[ticks]*N, ranks_tt=2, requires_grad=True)

def loss(t):
    return tn.relative_error(y_train, t[X_train])**2
tn.optimize(t, loss)
```

```
iter: 0      | loss:   0.999345 | total time:    0.0020
iter: 500    | loss:   0.884514 | total time:    0.8227
iter: 1000   | loss:   0.130079 | total time:    1.7812
iter: 1500   | loss:   0.019054 | total time:    2.7306
iter: 2000   | loss:   0.009741 | total time:    3.7072
iter: 2500   | loss:   0.005941 | total time:    4.7091
iter: 3000   | loss:   0.003692 | total time:    5.7308
iter: 3500   | loss:   0.002271 | total time:    6.8586
iter: 4000   | loss:   0.001340 | total time:    7.8531
iter: 4500   | loss:   0.000724 | total time:    8.8624
iter: 5000   | loss:   0.000347 | total time:    9.8511
iter: 5500   | loss:   0.000153 | total time:   10.8660
iter: 5743   | loss:   0.000100 | total time:   11.3829 <- converged (tol=0.0001)
```

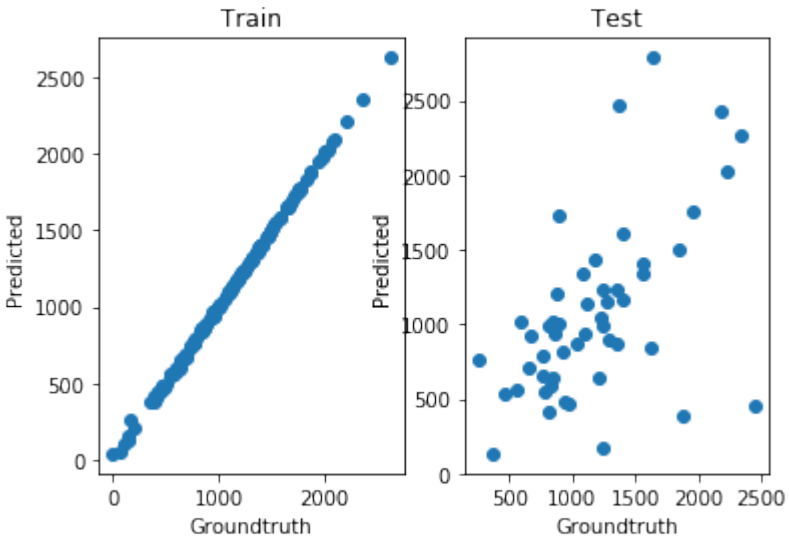
This TT did very well for the training data, but clearly overfitted:

[3]:
```
print('Test relative error:', tn.relative_error(y_test, t[X_test]))
print('The model overfitted: it has {} degrees of freedom, and there are {} training␣
↪instances'.format(tn.dof(t), len(X_train)))
```

```
Test relative error: tensor(0.4168, grad_fn=<DivBackward1>)
The model overfitted: it has 512 degrees of freedom, and there are 150 training␣
↪instances
```

We can look at the groundtruth vs. prediction for the training and test splits:

[4]:
```
import matplotlib.pyplot as plt
%matplotlib inline

def show():
    fig = plt.figure()
    fig.add_subplot(121)
    plt.scatter(y_train, t[X_train].torch().detach().numpy())
    plt.xlabel('Groundtruth')
    plt.ylabel('Predicted')
    plt.title('Train')
    fig.add_subplot(122)
    plt.scatter(y_test, t[X_test].torch().detach().numpy())
    plt.xlabel('Groundtruth')
    plt.ylabel('Predicted')
    plt.title('Test')
    plt.show()

show()
```



Seeking a Low-degree Polynomial Solution

Next we will try a PCE-like solution. The good news is that PCE is essentially a Tucker decomposition with certain custom factors, namely polynomial, and we can emulate this in *tntorch* via the TT-Tucker model. Essentially we will fix polynomial bases and impose low TT-rank structure on the learnable coefficients.

[5]:
```
t = tn.rand(shape=[ticks]*N, ranks_tt=2, ranks_tucker=3, requires_grad=True)  # There␣
↪are both TT-ranks *and* Tucker-ranks
t
```

[5]:
```
5D TT-Tucker tensor:

 32  32  32  32  32
  |   |   |   |   |
  3   3   3   3   3
 (0) (1) (2) (3) (4)
 / \ / \ / \ / \ / \
1   2   2   2   2   1
```

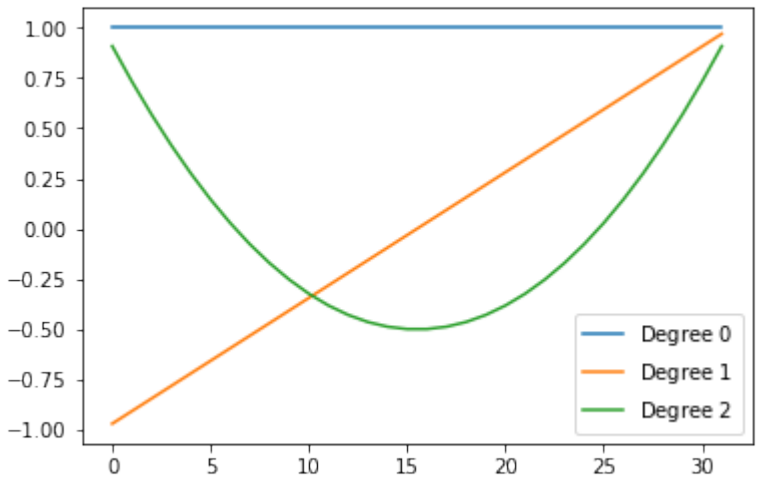
Note the shape of that tensor network: cores are no longer $2 \times 32 \times 2$, but $2 \times 3 \times 2$. Their middle dimension is now *compressed* using a so-called *factor matrix* of size 32×3. In other words, we are expressing each slice of a TT model as a linear combination of three "master" slices only.

Furthermore, we want those factors to be fixed bases and contain nicely chosen 1D polynomials along their columns:

[6]:
```
t.set_factors('legendre', requires_grad=False)  # We set the factors to Legendre␣
↪polynomials, and fix them (won't be changed during optimization)
```

As expected, the factors' columns are Legendre polynomials of increasing order:

[7]:
```
for i in range(t.ranks_tucker[0]):
    plt.plot(t.Us[0][:, i].numpy(), label='Degree ${}$'.format(i))
plt.legend()
plt.show()
```



We are now ready to proceed with our usual optimization target, this time just with this "smarter" tensor:

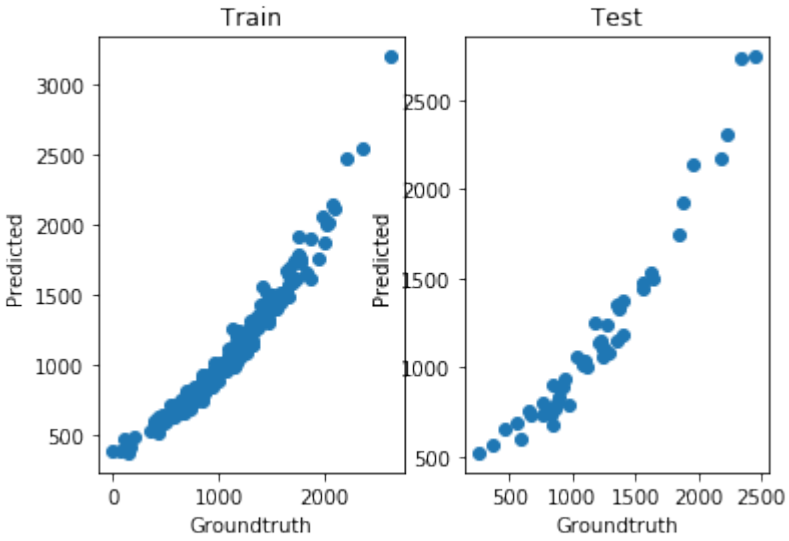
[8]:
```
tn.optimize(t, loss)
```

```
iter: 0      | loss:   0.998845 | total time:    0.0026
iter: 500    | loss:   0.816849 | total time:    1.5710
iter: 1000   | loss:   0.087476 | total time:    3.0443
iter: 1500   | loss:   0.011566 | total time:    4.4006
iter: 1737   | loss:   0.011055 | total time:    5.0428 <- converged (tol=0.0001)
```

[9]:
```
print('Test relative error:', tn.relative_error(y_test, t[X_test]))
print('This model is more restrictive: it only has {} degrees of freedom'.format(tn.
↪dof(t)))
```

```
Test relative error: tensor(0.1027, grad_fn=<DivBackward1>)
This model is more restrictive: it only has 48 degrees of freedom
```

[10]:
```
show()
```



3.3.16 Sobol Indices

Sobol's method is one of the most popular for global sensitivity analysis. It builds on the *ANOVA decomposition*.

[1]:
```
import tntorch as tn
import torch
import time

N = 20
t = tn.rand([32]*N, ranks_tt=10)
t
```

[1]:
```
20D TT tensor:

 32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9) (10)(11)(12)(13)(14)(15)(16)(17)(18)(19)
 / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \
1   10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  1
```

With *tntorch* we can handle *all* Sobol indices (i.e. for all subsets $\alpha \subseteq \{0, \dots, N-1\}$) at once. We can access and aggregate them using the function `sobol()` and whatever *mask* is appropriate.

Single Variables

Variance Components

The relative influence (proportion of the overall model variance) attributable to one variable n only, without interactions with others, is known as its *variance component* and denoted as S_n. Let's compute it for the first variable x:

[2]:
```
x, y, z = tn.symbols(N)[:3]
start = time.time()
print(tn.sobol(t, mask=tn.only(x)))
print('This compressed tensor has {} parameters; computing this index took only {:g}s
↪'.format(t.numel(), time.time()-start))
```

```
tensor(0.1882)
This compressed tensor has 58240 parameters; computing this index took only 0.0380359s
```

Input parameters $x, y, \dots$ are assumed independently distributed. By default, uniform marginal distributions are used, but you can specify others with the `marginals` argument (list of vectors). For instance, if the first variable can take one value only, then its sensitivity indices will be 0 (no matter how strong its effect on the multidimensional model is!):

[3]:
```
marginals = [None]*N  # By default, None means uniform
marginals[0] = torch.zeros(t.shape[0])
marginals[0][0] = 1  # This marginal's PMF is all zeros but the first value
tn.sobol(t, tn.only(x), marginals=marginals)
```

[3]:
```
tensor(0.)
```

Total Sobol Indices

The effect that also includes x's interaction with other variables is called *total Sobol index* (it's always larger or equal than the corresponding variance component):

```
[4]: tn.sobol(t, x)
[4]: tensor(0.2150)
```

Tuples of variables

What are the indices for the first and third variables x and z?

```
[5]: tn.sobol(t, tn.only(x & z))  # Variance component
[5]: tensor(0.0005)
```

```
[6]: tn.sobol(t, x | z)  # Total index
[6]: tensor(0.2401)
```

What's the relative importance of x with respect to the group $\{y, z\}$?

```
[7]: tn.sobol(t, x & (y|z)) / tn.sobol(t, y|z)
[7]: tensor(0.1638)
```

Closed Sobol Indices

For tuples of variables two additional kinds of indices exist. The *closed index* aggregates all components for tuples *included* in α, and for tuple $\{x, z\}$ it can be computed as follows:

```
[8]: tn.sobol(t, tn.only(x | z))
[8]: tensor(0.2100)
```

Superset Indices

The *superset index* aggregates all components for tuples *that include* α:

```
[9]: tn.sobol(t, x & z)
[9]: tensor(0.0009)
```

Counting k-plets of Variables

We can also easily count the influence of all k-plets of variables combined:

```
[10]: tn.sobol(t, tn.weight_mask(N, weight=[1]))
[10]: tensor(0.9222)
```

Often, there are different ways to express the same mask. For example, these three are equivalent:

```
[11]: print(tn.sobol(t, x | z))
      print(tn.sobol(t, x & ~z) + tn.sobol(t, ~x & z) + tn.sobol(t, x & z))
      print(tn.sobol(t, x) + tn.sobol(t, z) - tn.sobol(t, x & z))
```

```
tensor(0.2401)
tensor(0.2401)
tensor(0.2401)
```

The Mean Dimension

Variance components are the basis for an important advanced sensitivity metric, the *mean dimension*. It's defined as $D_S := \sum_\alpha |\alpha| \cdot S_\alpha$ and computed as:

[12]:
```
tn.mean_dimension(t)
```

[12]:
```
tensor(1.0831)
```

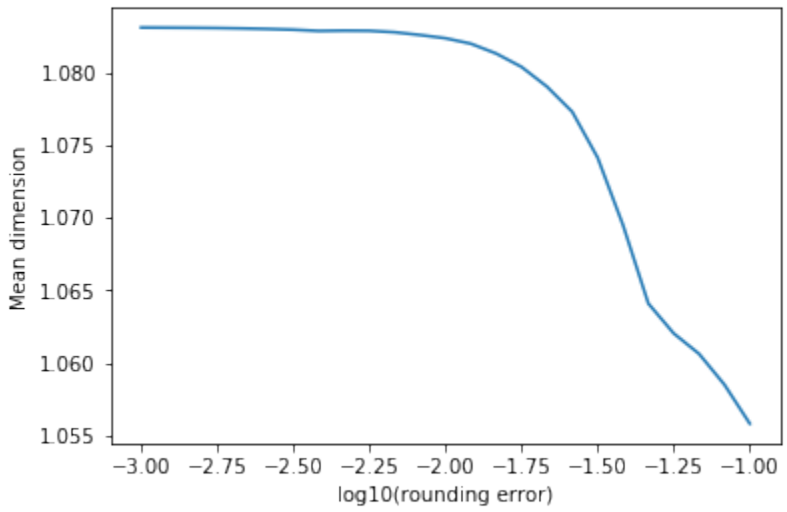
We can also compute it in one line by weighting the Sobol indices by their tuple weight (according to the definition of mean dimension):

[13]:
```
tn.sobol(t, tn.weight(N))
```

[13]:
```
tensor(1.0831)
```

The mean dimension is always greater or equal than 1. It gives a notion of *complexity* of a multidimensional function (the lower the mean dimension, the simpler it is). For example, rounding a tensor usually results in a lower mean dimension:

[14]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

errors = 10**np.linspace(-1, -3, 25)
mean_dimensions = []
for eps in errors:
    mean_dimensions.append(tn.mean_dimension(tn.round(t, eps=eps)))

plt.figure()
plt.plot(np.log10(errors), mean_dimensions)
plt.xlabel('log10(rounding error)')
plt.ylabel('Mean dimension')
plt.show()
```



We can also compute the *restricted* mean dimension, i.e. impose certain conditions on the set of tuples that intervene. For example, we can see which of two variables tends to show up more with higher-order terms:

[15]:
```
print(tn.mean_dimension(t, mask=x))
print(tn.mean_dimension(t, mask=y))
```

```
tensor(1.1355)
tensor(1.3664)
```

The Dimension Distribution

Last, the *dimension distribution* gathers the relevance of k-tuples of variables for each $k = 1, \dots, N$:

[16]:
```
start = time.time()
dimdist = tn.dimension_distribution(t)
print(dimdist)
print('Time:', time.time() - start)
```

```
tensor([9.2223e-01, 7.2741e-02, 4.7444e-03, 2.6979e-04, 1.3797e-05, 6.4638e-07,
        2.8097e-08, 1.1428e-09, 4.3612e-11, 1.5722e-12, 5.3506e-14, 1.7216e-15,
        5.2287e-17, 1.4966e-18, 4.0144e-20, 1.0023e-21, 2.3034e-23, 4.7723e-25,
        8.6303e-27, 1.3940e-28])
Time: 0.11500120162963867
```

It can be viewed as a probability mass function, namely the probability of choosing a k-variable tuple, if tuples are chosen according to their variance components. The expected value of this random variable is the mean dimension. Naturally, the dimension distribution must sum to 1:

[17]:
```
sum(dimdist)
```

[17]:
```
tensor(1.0000)
```

Again, we can extract the dimension distribution with respect to any mask:

[18]:
```
tn.dimension_distribution(t, mask=y&z)
```

[18]:
```
tensor([0.0000, 0.6815, 0.2870, 0.0293, 0.0021, 0.0001, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000])
```

We just imposed that two variables (y and z) appear. Note how, accordingly, the relevance of 1-tuples has become zero since they have all been discarded.

3.3.17 Vector Fields

The standard way to represent N-dimensional vector fields in *tntorch* is via a list of N tensors, each of which has N dimensions. Functions that accept or return vector fields do so in that form. For example, `gradient()`:

[11]:
```
import torch
import tntorch as tn

t = tn.rand([64]*3, ranks_tt=10)
grad = tn.gradient(t)
print(grad)
```

```
[3D TT tensor:

 64  64  64
  |   |   |
 (0) (1) (2)
 / \ / \ / \
1   10  10  1
, 3D TT tensor:

 64  64  64
  |   |   |
 (0) (1) (2)
 / \ / \ / \
1   10  10  1
, 3D TT tensor:

 64  64  64
  |   |   |
 (0) (1) (2)
 / \ / \ / \
1   10  10  1
]
```

We can check that the curl of any gradient is 0:

[12]:
```
curl = tn.curl(tn.gradient(t))  # List of 3 3D tensors
print(tn.norm(curl[0]))
print(tn.norm(curl[1]))
print(tn.norm(curl[2]))
```

```
tensor(1.0344e-06)
tensor(1.4569e-06)
tensor(2.5995e-06)
```

Let's also check that the divergence of any curl is zero (we'll use a random, non-gradient vector field here):

[3]:
```
vf = [tn.rand([64]*3, ranks_tt=1) for n in range(3)]
tn.norm(tn.divergence(tn.curl(vf)))
```

[3]:
```
tensor(4.8832e-08)
```

... and that the Laplacian of a scalar field `t` equals the divergence of `t`'s gradient:

[5]:
```
tn.norm(tn.laplacian(t) - tn.divergence(tn.gradient(t)))
```

[5]:
```
tensor(0.)
```

3.4 Contact/Contributing

This project is mainly developed by Rafael Ballester-Ripoll (Visualization and MultiMedia Lab, University of Zurich). Feel free to contact me at rballester@ifi.uzh.ch for comments, ideas, or issues (consider using GitHub's issue tracker as well).

Pull requests are welcome anytime!

Python Module Index

Index

M

N

O

P

R

S

T

U

V

W

Z