ptens

Permutation equivariant message passing library, v0.0

Last update: August 2023

Risi Kondor
Department of Computer Science, Statistics,
and the Computational and Applied Math Initiative
The University of Chicago

Contents

Overview

ptens is a C++/CUDA library for permutation equivariant message passing, including primitives for higher order message passing in graph neural networks. ptens is optimized for sending messages between a hetereogeneous group of many permutation co-variant tensors (P-tensors) on GPUs in parallel.

Most operations in ptens are differentiable, facilitating backprogation in neural networks. In addition to the native C++ API, the library also has a PyTorch interface. This document describes the library's overall design and its C++ API. The PyTorch interface is documented separately at https://risi-kondor.github.io/ptens-doc/.

ptens uses the cnine tensor library, which can be found at https://github.com/risi-kondor/cnine.

ptens is under continuous development by Risi Kondor's research group and is released for non-commercial use only under a custom license bundled with the source code in the file `LICENSE.txt`.

Installation and Usage

ptens is mostly a header library, so it does not need to be installed in a specific location on your system. However, if the library is to be used with GPU functionality, certain CUDA object files must be separately compiled.

ptens builds on cnine, which must be separately downloaded from https://github.com/risi-kondor/cnine. Please refer to cnine's documentation for information on the basic scalar and tensor classes used in ptens. ptens also requires:

- An appropriate `C++11` compiler together with the STL standard template library.
- CUDA and CUBLAS if the library is to be used with GPU functionality.

Compilation options

Before attempting to compile the CUDA objects or compile your own code against ptens, please copy `config_template.txt` to a file named `config.txt`, and set the following compilation parameters.

Variable	Default value	Description
`CC`	`clang`	Name of C++ compiler.
`CNINE_ROOT`	`$(ROOTDIR)/../cnine/`	Root directory of cnine relative to the root of ptens.
`WITH_CUDA`	`t`	If this variable is defined, the library will link to CUDA. Otherwise, GPU functionality is disabled.
`CUDA_DIR`	`/usr/local/cuda`	Root directory of the CUDA installation on your system.

Usage

To call ptens from C++ code you must do the following:

- `#include` the relevant header files in your soure files.
- `#include` the file `include/ptens_base.cpp` in your top level source file (the one that contains your `main` function).
- Link in the appropriate CUDA object files from the `cuda` directory, as required.

Background

Let $\mathbf{v}$ be an n-element vector. Permutations act naturally on $\mathbf{v}$, be permuting its elements. Formally, letting $\mathbb{S}_n$ denote the **symmetric group** of order n, regarded as the group of bijections of the set $\{1, 2, \ldots, n\}$, permutations act on $\mathbf{v}$ as

$$\mathbf{v} \mapsto \sigma \circ \mathbf{v} \qquad [\sigma \circ \mathbf{v}]_i = \mathbf{v}_{\sigma^{-1}(i)}, \qquad \sigma \in \mathbb{S}_n. \tag{1}$$

Any vector $\mathbf{v}$ that transforms in this way we will call a **permutation covariant** vector. On the other hand, the natural action of $\mathbb{S}_n$ on $n \times n$ matrices is

$$A \mapsto (\sigma \circ A) \qquad [\sigma \circ A]_{i,j} = \mathbf{v}_{\sigma^{-1}(i),\sigma^{-1}(j)}, \qquad \sigma \in \mathbb{S}_n, \tag{2}$$

so we say that A is permutation covariant if it transforms in this way. Generalizing to tensors of arbitrary order, we say that $T \in \mathbb{R}^{n \times n \times \ldots \times n}$ is a k'th order permutation covariant tensor if it transforms as

$$T \mapsto (\sigma \circ T) \qquad [\sigma \circ T]_{i_1,\ldots,i_k} = T_{\sigma^{-1}(i_1),\ldots,\sigma^{-1}(i_k)} \qquad \sigma \in \mathbb{S}_n.$$

Permutation covariant vectors, matrices and tensors appear frequently in different branches of mathematics, physics and computer science.

Often, it is of particular interest to construct **equivariant linear maps** between such objects. Here, a linear map ϕ from one tensor T_1 to another tensor T_2 (which may be of the same or different order as T_1) being equivariant just means that it preserves the behavior of tensors under permutations in the sense if

$$[T']_{i_1,\ldots,i_{k_1}} = [T]_{\sigma^{-1}(i_1),\ldots,\sigma^{-1}(i_k)}$$

then

$$[\phi(T')]_{i_1,\ldots,i_{k_1}} = [\phi(T)]_{\sigma^{-1}(i_1),\ldots,\sigma^{-1}(i_k)}.$$

In other words, ϕ commutes with the action of $\mathbb{S}_n$:

$$\phi(\sigma \circ T) = \sigma \circ (\phi(T)).$$

Permutation equivariant maps are particularly important in the field of graph neural networks, since they effectively define all possible linear message passing operations between vertices, edges and subgraphs, ecompassing a wide range of neural network architectures. Hence there is considerable interest in implementing all possible permutation equivariant maps in software in a way that can scale to meet the demands of present day neural network architectures.

Enumerating all equivariant maps

In the first-order to first-order case (i.e. vector-to-vector maps) [Zaheer et al.] showed that the most general permutation equivariant map is of the form

$$\phi \colon \mathbf{v} \mapsto \mathbf{w} \qquad \mathbf{w} = (\lambda_1 I + \lambda_2 \mathbf{1}^\top \mathbf{1})\mathbf{v}.$$

In other words, the space of equivariant linear maps is spanned by

$$\phi_{\{1,2\}}(\mathbf{v}) = \mathbf{v}$$
$$[\phi_{\{1\},\{2\}}(\mathbf{v})]_i = \sum_j \mathbf{v}_j.$$

Any equivariant linear map $\mathbf{v} \to \mathbf{w}$ can be expressed as a linear combination of these two maps.

The number of independent equivariant maps grows sharply with the order of the tensors involved. For example, in the second-order to second-order, i.e., matrix-to-matrix case, we already have 15 linearly independent maps:

$$
\begin{aligned}
[\phi_{\{1,3\},\{2,4\}}(A)]_{i,j} &= A_{i,j} \\
[\phi_{\{1,4\},\{2,3\}}(A)]_{i,j} &= A_{j,i} \\
\\
[\phi_{\{1,2,3,4\}}(A)]_{i,i} &= A_{i,i} \\
[\phi_{\{1,2,3\},\{4\}}(A)]_{i,i} &= \textstyle\sum_k A_{i,k} \\
[\phi_{\{1,2,4\},\{3\}}(A)]_{i,i} &= \textstyle\sum_k A_{k,i} \\
[\phi_{\{1,2\},\{3,4\}}(A)]_{i,i} &= \textstyle\sum_k A_{k,k} \\
[\phi_{\{1,2\},\{3\},\{4\}}(A)]_{i,i} &= \textstyle\sum_{k,\ell} A_{k,\ell}
\end{aligned}
\qquad
\begin{aligned}
[\phi_{\{1,3,4\},\{2\}}(A)]_{i,j} &= A_{i,i} \\
[\phi_{\{1\},\{2,3,4\}}(A)]_{i,j} &= A_{j,j} \\
[\phi_{\{1,3\},\{2\},\{4\}}(A)]_{i,j} &= \textstyle\sum_k A_{i,k} \\
[\phi_{\{1,4\},\{2\},\{3\}}(A)]_{i,j} &= \textstyle\sum_k A_{k,i} \\
[\phi_{\{1\},\{2,3\},\{4\}}(A)]_{i,j} &= \textstyle\sum_k A_{j,k} \\
[\phi_{\{1\},\{2,4\},\{3\}}(A)]_{i,j} &= \textstyle\sum_k A_{k,j} \\
[\phi_{\{1\},\{2\},\{3,4\}}(A)]_{i,j} &= \textstyle\sum_k A_{k,k} \\
[\phi_{\{1\},\{2\},\{3\},\{4\}}(A)]_{i,j} &= \textstyle\sum_{k,\ell} A_{k,\ell}
\end{aligned}
$$

Here we use the convention that any entries of the output matrix that are not explicitly referenced are assumed to be zero. For example, $[\phi_{\{1,2\},\{3\},\{4\}}(A)]_{i,i} = \sum_{k,\ell} A_{k,\ell}$ would be more explicitly written as

$$
[\phi_{\{1,2\},\{3\},\{4\}}(A)]_{i,j} = \begin{cases} \sum_{k,\ell} A_{k,\ell} & \text{if } \; i = j \\ 0 & \text{otherwise,} \end{cases}
$$

so this particular map just transfers the sum of all the matrix elements to the diagonal of the output matrix.

In a seminal paper [Maron et al] showed that in the general case, the total number of linearly independent equivariant maps from a k_1'th order tensor to a k_2'th order tensor is given by the so-called Bell number $B(k_1 + k_2)$, which enumerates all possible partitions of the set $\{1, 2, \ldots, k_1 + k_2\}$. Moreover, the paper also gives an an explicit way to construct the map corresponding to each partition.

To understand this connection between partitions and equivariant linear maps, we denote the indices appearing on the left hand side of the equation defining the map $i_1, \ldots, i_{k_2}$ and those on the right hand side $i_{k_2+1}, \ldots, i_{k_1+k_2}$. The partition $\mathcal{P}$ indexing a given map ϕ just indicates which of these indices are tied together. To be specific, we consider three different types of partitions:

1. **Summation variables.** Partitions that only involve $\{k_2 + 1, \ldots, k_1 + k_2\}$ correspond to summation variables, i.e., indices of the input tensor that we sum over. These operations are also called **tensor reductions**.
2. **Transfer variables.** Partitions that involve indices both from $\{1, 2, \ldots, k_2\}$ and $\{k_2+1, \ldots, k_1+k_2\}$ correspond to **transfer variables**, i.e., variables that indicate which indices of the ouput tensor correspond to which indices of the input tensor.
3. **Broadcast variables.** Partitions that only involve $\{1, 2, \ldots, k_2\}$ correspond to broadcast variables, i.e., dimensions of the output tensor along which the result will just be repeated.

An example can serve to illustrate these concepts. Let us take $k_1 = k_2 = 3$ case, and consider the map cprrespoonding to $\mathcal{P} = \{\{1, 3\}, \{2, 5, 6\}, \{4\}\}$. This partition prescribes (a) summing T_1 along its first dimension (corresponding to $\{4\}$) (b) transferring the diagonal along the second and third dimensions of T_1 to the second dimension of T_2 (corresponding to $\{2, 5, 6\}$), (c) broadcasting the result along the diagonal of the first and third dimensions (corresponding to $\{1, 3\}$). Explicitly, this gives the equivariant map

$$
[\phi(T)]_{a,b,a} = \sum_c T_{c,b,b}. \tag{3}
$$

In actual fact the indices used in enumerating the vector-to-vector and matrix-to-matrix cases above already followed [Maron et al.]'s scheme.

Partially overlapping reference domains

When used in graph neural networks, the indices of permutation covariant tensors either correspond to all the vertices of the graph G or just of a subgraphs. This complicates matters because it implies that we have to construct maps between tensors that are equivariant to just a subset of their indices. The technical formalism required to deal with this case was introduced in [Ptensors] and involves talking about a "universe" of atomic objects, reference domains, and so-called P-tensors. Formally, P-tensors are defined as follows.

Definition 1 (**P-tensors**). *Let $\mathcal{U}$ be a finite or countably infinite set of atoms and $\mathcal{D} = (x_1, \ldots, x_d)$ an ordered subset of $\mathcal{U}$. We say that a k'th order tensor $T \in \mathbb{R}^{d \times d \times \ldots \times d}$ is a k'th order* permutation covariant tensor *(or **P-tensor** for short) with **reference domain** $\mathcal{D}$ if under the action of a permutation $\tau \in \mathbb{S}_d$ acting on $\mathcal{D}$ it transforms as*

$$\tau(T)_{i_1,i_2,\ldots,i_k} = T_{\tau^{-1}(i_1),\ldots\tau^{-1}(i_k)}. \tag{4}$$

P-tensor appearing in graph neural networks usually correspond to subgraphs of a larger graph G. $\mathcal{U}$ consists of all the vertices of G, while the reference domain $\mathcal{D}$ of a P-tensor corresponding to a subgraph consists of all the vertices featured in the subgraph.

The reason that enumerating all possible equivariant maps between an input P-tensors T^{in} and an output P-tensor T^{out} is more complicated is that their repsctive reference domains $\mathcal{D}^{\text{in}}$ and $\mathcal{D}^{\text{out}}$ might only partially overlap. Specifically, this implies that in a given equivariant map

(a). The domain of the transfer variables can only be $\mathcal{D}^{\text{in}} \cap \mathcal{D}^{\text{out}}$.

(b). The domain of the reduction variables can be *either* $\mathcal{D}^{\text{in}} \cap \mathcal{D}^{\text{out}}$ or the entirety of $\mathcal{D}^{\text{in}}$.

(c). The domain of the broadcast variables can be *either* $\mathcal{D}^{\text{in}} \cap \mathcal{D}^{\text{out}}$ or the entirety of $\mathcal{D}^{\text{out}}$.

Correspondingly, in the general case, the space of equivariant linear maps between P-tensors is larger. For a given partition $\mathcal{P}$ of $\{1, 2, \ldots, k_1 + k_2\}$ that has p_{in} parts that fall entirely within $\{k_2 + 1, \ldots, k_1 + k_2\}$ and p_{out} parts that fall entirely within $\{1, 2, \ldots, k_2\}$, we will have not just one, but $2^{p_{\text{in}} + p_{\text{out}}}$ separate linerarly independent equivariant maps.

The ptens library

The fundamental data objects in ptens are P-tensors, as defined above, with the small caveat that to accomodate usual neural network architectures we also give P-tensors an extra "channel" dimension. Thus, a k'th order P-tensor is actually implemented as a $(k+1)$'th order tensor object. Integral to each P-tensor is its reference domain $\mathcal{D}$.

The core functionality of the ptens library consists of what we call **transfer** operations, which amount to efficiently computing all possible equivariant maps between two P-tensors, automatically taking into account how their reference domains are related to each other. Naturally, to enable integrration into neural network architectures, all the operations in ptens support automatic differentiation.

Design

ptens implements the Ptensors formalism in a generic way that makes it applicable to both graph neural networks and other domains. In principle, each equivariant linear map between Ptensors introduced in the Background section can be reduced to standard tensor operations supported by a number of existing tensor and neural network libraries such as `TensorFlow`, `PyTorch`, and so on. However, there are three factors that would make efficient message passing between Ptensors difficult to implement in standard libraries:

1. Each map $\phi_{\mathcal{P}}: T^{\rm in} \mapsto T^{\rm out}$ involves matching the reference domains of the input and output tensors. In practical terms, this involves forming subtensors of a given P-tensor corresponding to just the indices in $\mathcal{D}^{\rm in} \cap \mathcal{D}^{\rm out}$ along specific dimensions and permuting indices in such a way that the corresponding indices in $T^{\rm in}$ and $T^{\rm out}$ match up. While many tensor libraries support taking subtensors along specific combinations of slices, as well as permuting indices, when these processes is implemented in a high level language such as Pythoon, they carries a large overhead. Moreover, making an explicit copy of the subtensor and explicitly permuting the vertices is wasteful, but for these operations most libraries do not offfer tensor views.
2. Typically we are interested in not just computing a single linear map or a small collection of linear maps, but all possible linearly independent equivariant maps from $T^{\rm in}$ to $T^{\rm out}$, so that in a neural network architecture, after being multiplied by appropriate weight matrices, the network can "learn" their ideal linear combinations. As explained in the Background section, the number of such maps grows sharply with the order of the tensors involved. However their computation can be accelerated by taking advantage of shared intermediate results.
3. Most crucially, typical applications of the Ptensors formalism involve computing message passing operations between many pairs of P-tensors. Acceptable levels of performance in neural network applications in particular, can typically only be achieved if these operations are performed on graphics accelerators (GPUs), and they can be parallelized over the $(T^{\rm in}, T^{\rm out})$ pairs. However, for each $(T^{\rm in}, T^{\rm out})$ pair (a) the size of the input and output tensors, (b) the number of overlapping atoms (c) the matching between the atoms in $\mathcal{D}^{\rm in}$ and $\mathcal{D}^{\rm out}$ are all different. This would make it impossible to compute all the linear maps in parallel, in a single GPU kernel call.

Parallelizing message passing

ptens addresses the above issues by using specialized CUDA kernels for computing equivariant maps between P-tensors. Specifically, it assumes that the P=tensors are organized in layers and to compute the maps from layer $\mathcal{L}_1$ to layer $\mathcal{L}_2$,

1. ptens first computes internal datastructures on the CPU side encoding information about all $(T^{\rm in}, T^{\rm out})$ pairs that messages need to be passed between, including the index matching process;
2. Then it copies these datatstructures to the GPU and computes all linear maps between all pairs of tensors in a single kernel call.

An additional factor contributing to the efficiency of message passing in ptens is that the library separates the process into two distince phases: 1. first computing all possible reductions of the input tensors 2. then broadcasting the reductions to the output tensors in all possible ways. For example, in the matrix-to-matrix case for which we listed all 15 equivariant linear maps, this amounts to computing two zeroth order "messages"

$$M^0_{\{1\},\{2\}} = \sum_{k,\ell} T^{\rm in}_{k,\ell}$$
$$M^0_{\{1,2\}} = \sum_{k} T^{\rm in}_{k,k}$$

three first order messages

$$[M^1_{\{*,1\}}]_i = \sum_k T^{\text{in}}_{i,k}$$
$$[M^1_{\{*,2\}}]_i = \sum_k T^{\text{in}}_{k,i}$$
$$[M^1_{\{*,2\}}]_i = T^{\text{in}}_{i,i}$$

and a single second order message

$$[M^2_{\{*,1\},\{2\}}]_{i,j} = T^{\text{in}}_{i,j}$$

Each zeroth order message can be broadcast in two different ways:

$$T^{\text{out}}_{a,b} = M^0$$
$$T^{\text{out}}_{a,a} = M^0,$$

each first order message can be broadcast in three different ways:

$$T^{\text{out}}_{i,j} = M^1_i$$
$$T^{\text{out}}_{i,j} = M^1_j$$
$$T^{\text{out}}_{i,i} = M^1_j,$$

whereas the second order massage can be broadcase in two different ways:

$$T^{\text{out}}_{i,j} = M^2_{i,j}$$
$$T^{\text{out}}_{i,j} = M^2_{j,i}.$$

Combining the above reductions with broadcast maps gives all $2 \times 2 + 3 \times 3 + 1 \times 2 = 15$ equivariant maps.

GPU utilization strategy

Normally a single CUDA kernel call performs exactly the same series of operations in parallel on many data instances. In contrast, passing messages from one layer of P-tensors to another layer, as described above, requires performing a slightly different sequence of instructions for each $(T^{\text{in}}, T^{\text{out}})$ pair, because the exact size of the intersection of their reference domains, and the matching between their atoms varies.

To address this issue, ptens uses thread-block level parallelism for the different $(T^{\text{in}}, T^{\text{out}})$ pairs, i.e., the reduction and broadcast operations for each tensor are dispatched to different streaming multiprocessors (SM's). Today's NVIDIA GPU architectures typically have on the order of a few dozen SM's, so that determines the number of messages that can be processed concurrently. A single kernel call can of course still initiate a far larger number of reduction and broadcast operations, which the GPU's internal scheduler then dispatches to the available SM's according to its own policy. However, the total number of reduction and broadcast operations running at any given point in time will be limited by the number of SM's.

Each SM can run multiple threads internally, but these threads are tied together in the sense that (with some caveats) they essentially need to be performing exactly the same sequence of operation at the same time. While the number of logical threads on a SM can be up to 1024, in reality only 32 or 64 of them can be truly concurrent. ptens primarily uses this thread-level parallelism to process to parallelize over channels.

An added advantage of using different thread blocks for different $(T^{\text{in}}, T^{\text{out}})$ pairs is that threads running on the same SM can share information with each other through the SM's internal so-called shared memory. While the size of shared memory is small (typically 48k), accessing it can be an order of magnitude faster than accessing the GPU's global memory. Thus, the sharing of information between different reductions and broadcast maps of the same tensor can be extremely efficient.

The thread layout described above has very concrete implications as to what types of computations ptens can perform efficiently. The library is designed to handle message passing between a relatively large number of heterogeneous $(T^{\text{in}}, T^{\text{out}})$ pairs, where the individual tensors are of moderate size. The specific consdiderations for optimal GPU utilization are the following:

1. The number of channels is ideally between 32 and 1024 and it is best if it is an exact multiple of 32 or just less than an exact multiple. If the number of channels is much less than 32, then GPU utilization will be poor, because on each SM many threads will be sitting idle.
2. Similarly, if the number of $(T^{\text{in}}, T^{\text{out}})$ pairs is very small, then GPU utilization will be poor because ptens will effectively only use a very small number of SMs.
3. Ideally the size of the tensors is small enough that all the intermediate computations involved in computing the messages and broadcasting them fits in shared memory.
4. Presently, 1024 is a hard limit on the number channels in each P-tensor.

Reference

Th following pages documents the APIs of the most important classes in ptens. Not all member functions are listed in the documentation.

In addition to the explicitly listed member functions, for debugging purposes, each class has a `str` method with signature

```
string str(const indent="") const
```

that prints the object to string in human readable format. Most classes also support serialization directly to `cout`.

For brevity, when referring to classes from `cnine`, the `cnine::` prefix is generally omitted.

P-tensors & P-tensor layers

Ptensor0

A `Ptensor0` object stores a single zeroth order P-tensor T.

Derived from: `cnine::RtensorA`

VARIABLES

`int nc`
: Number of channels

`Atoms atoms`
: The reference domain of this P-tensor

CONSTRUCTORS

`Ptensor0(const Atoms& atoms, const int nc, const FILLTYPE& fill, const int dev=0)`
: Create a new `Ptensor0` object with reference domain given by `atoms` and `nc` channels. The fill type can be `cnine::fill_zero()`, `cnine::fill_gaussian()` or `cnine::fill_sequential()`. Depending on `dev`, the `Ptensor0` will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

NAMED CONSTRUCTORS

`Ptensor0 Ptensor0::raw(const Atoms& atoms, const int nc, const int dev=0)`
`Ptensor0 Ptensor0::zero(const Atoms& atoms, const int nc, const int dev=0)`
`Ptensor0 Ptensor0::gaussian(const Atoms& atoms, const int nc, const int dev=0)`
`Ptensor0 Ptensor0::sequential(const Atoms& atoms, const int nc, const int dev=0)`
: Create a new `Ptensor0` object with reference domain given by `atoms` and `nc` channels. Depending on `dev`, the `Ptensor0` will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

REDUCTIONS

`RtensorA reduce0() const`
`RtensorA reduce0(const int offs, const int n) const`
`RtensorA reduce0(const vector<int>& ix) const`
`RtensorA reduce0(const vector<int>& ix, const int offs, const int n) const`
: Compute a zeroth order message M from T. In the first case we just have $M = T$. In the second case $M = T^{\text{offs:offs}+n}$. The second case is usually used for backpropagation.

BROADCASTING

`void broadcast0(const RtensorA& M) const`
`void broadcast0(const RtensorA& M, const int offs) const`
`void broadcast0(const RtensorA& M, const vector<int>& ix) const`
`void broadcast0(const RtensorA& M, const vector<int>& ix, const int offs) const`
: Broadcast the zeroth order message M into T. In the first case we just have $T \mathrel{+}= M$. In the second case $T^{\text{offs:offs}+n} \mathrel{+}= M$.

Ptensor1

A `Ptensor1` object stores a single first order P-tensor T.

Derived from: `cnine::RtensorA`

VARIABLES

`int nc`
: Number of channels

`Atoms atoms`
: The reference domain of this P-tensor

CONSTRUCTORS

`Ptensor1(const Atoms& atoms, const int nc, const FILLTYPE& fill, const int dev=0)`
: Create a new `Ptensor1` object with reference domain given by `atoms` and `nc` channels. The fill type can be `cnine::fill_zero()`, `cnine::fill_gaussian()` or `cnine::fill_sequential()`. Depending on `dev`, the `Ptensor1` will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

NAMED CONSTRUCTORS

```
Ptensor0 Ptensor1::raw(const Atoms& atoms, const int nc, const int dev=0)
Ptensor0 Ptensor1::zero(const Atoms& atoms, const int nc, const int dev=0)
Ptensor0 Ptensor1::gaussian(const Atoms& atoms, const int nc, const int dev=0)
Ptensor0 Ptensor1::sequential(const Atoms& atoms, const int nc, const int dev=0)
```
: Create a new `Ptensor1` object with reference domain given by `atoms` and `nc` channels. Depending on `dev`, the `Ptensor1` will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

REDUCTIONS

```
RtensorA reduce0() const
RtensorA reduce0(const int offs, const int n) const
RtensorA reduce0(const vector<int>& ix) const
RtensorA reduce0(const vector<int>& ix, const int offs, const int n) const
```
: Compute a zeroth order message `M` from `T`. In the first case $M = \sum_i T_i$, in the second case $M = \sum_i T_i^{\text{offs:offs}+n}$.

```
RtensorA reduce1() const
RtensorA reduce1(const int offs, const int n) const
RtensorA reduce1(const vector<int>& ix) const
RtensorA reduce1(const vector<int>& ix, const int offs, const int n) const
```
: Compute a first order message `M` from `T`. In the first case we just have $M_i = T_i$, in the second case $M = T_i^{\text{offs:offs}+n}$.

BROADCASTING

```
void broadcast0(const RtensorA& M) const
void broadcast0(const RtensorA& M, const int offs) const
void broadcast0(const RtensorA& M, const vector<int>& ix) const
void broadcast0(const RtensorA& M, const vector<int>& ix, const int offs) const
```

Broadcast the zeroth order message M into T. In the first case we have $T_i^c \mathrel{+}= M^c$. In the second case $T_i^{\mathrm{offs}+c} \mathrel{+}= M^c$.

```
void broadcast1(const RtensorA& M) const
void broadcast1(const RtensorA& M, const int offs) const
void broadcast1(const RtensorA& M, const vector<int>& ix) const
void broadcast1(const RtensorA& M, const vector<int>& ix, const int offs) const
```

Broadcast the first order message M into T. In the first case we just have $T_i^c \mathrel{+}= M_i^c$. In the second case $T_i^{\mathrm{offs}+c} \mathrel{+}= M_i^c$.

Ptensor2

A `Ptensor2` object stores a single second order P-tensor T.

Derived from: `cnine::RtensorA`

VARIABLES

`int nc`
: Number of channels

`Atoms atoms`
: The reference domain of this P-tensor

CONSTRUCTORS

`Ptensor2(const Atoms& atoms, const int nc, const FILLTYPE& fill, const int dev=0)`
: Create a new `Ptensor2` object with reference domain given by `atoms` and `nc` channels. The fill type can be `cnine::fill_zero()`, `cnine::fill_gaussian()` or `cnine::fill_sequential()`. Depending on `dev`, the `Ptensor2` will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

NAMED CONSTRUCTORS

`Ptensor0 Ptensor2::raw(const Atoms& atoms, const int nc, const int dev=0)`
`Ptensor0 Ptensor2::zero(const Atoms& atoms, const int nc, const int dev=0)`
`Ptensor0 Ptensor2::gaussian(const Atoms& atoms, const int nc, const int dev=0)`
`Ptensor0 Ptensor2::sequential(const Atoms& atoms, const int nc, const int dev=0)`
: Create a new `Ptensor2` object with reference domain given by `atoms` and `nc` channels. Depending on `dev`, the `Ptensor2` will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

REDUCTIONS

`RtensorA reduce0() const`
`RtensorA reduce0(const int offs, const int n) const`
`RtensorA reduce0(const vector<int>& ix) const`
`RtensorA reduce0(const vector<int>& ix, const int offs, const int n) const`
: Compute the zeroth order messages `M` from `T`. `M` will have twice as many channels as `T`, with

$$M^c = \sum_{i,j} T^c_{i,j}$$
$$M^{n_c+c} = \sum_i T^c_{i,i}.$$

`RtensorA reduce1() const`
`RtensorA reduce1(const int offs, const int n) const`
`RtensorA reduce1(const vector<int>& ix) const`
`RtensorA reduce1(const vector<int>& ix, const int offs, const int n) const`
: Compute a first order messages `M` from `T`. M will have three times as many channels as `T` with

$$M^c_i = \sum_j T^c_{j,i}$$
$$M^{n_c+c}_i = \sum_j T^c_{i,j}$$
$$M^{2n_c+c}_i = T^c_{i,i}.$$

```
RtensorA reduce2() const
RtensorA reduce2(const int offs, const int n) const
RtensorA reduce2(const vector<int>& ix) const
RtensorA reduce2(const vector<int>& ix, const int offs, const int n) const
```
Compute the second order messages `M` from `T`, which in this case are just $M^c_{i,j} = T^c_{i,j}$.

BROADCASTING

```
void broadcast0(const RtensorA& M) const
void broadcast0(const RtensorA& M, const int offs) const
void broadcast0(const RtensorA& M, const vector<int>& ix) const
void broadcast0(const RtensorA& M, const vector<int>& ix, const int offs) const
```
Broadcast the zeroth order message M into T. Letting $\tilde{n}_c$ be the number of channels in M, the broadcast rule is

$$T^c_{i,j} = \begin{cases} M^c & 1 \le c \le \tilde{n}_c \\ \delta_{i,j} M^c & \tilde{n}_c + 1 \le c \le 2\tilde{n}_c. \end{cases}$$

```
void broadcast1(const RtensorA& M) const
void broadcast1(const RtensorA& M, const int offs) const
void broadcast1(const RtensorA& M, const vector<int>& ix) const
void broadcast1(const RtensorA& M, const vector<int>& ix, const int offs) const
```
Broadcast the first order message M into T. Letting $\tilde{n}_c$ be the number of channels in M, the broadcast rule is

$$T^c_{i,j} = \begin{cases} M^c_j & 1 \le c \le \tilde{n}_c \\ M^c_i & \tilde{n}_c + 1 \le c \le 2\tilde{n}_c \\ \delta_{i,j} M^c_i & 2\tilde{n}_c + 1 \le c \le 3\tilde{n}_c. \end{cases}$$

```
void broadcast2(const RtensorA& M) const
void broadcast2(const RtensorA& M, const int offs) const
void broadcast2(const RtensorA& M, const vector<int>& ix) const
void broadcast2(const RtensorA& M, const vector<int>& ix, const int offs) const
```
Broadcast the second order message M into T. Letting $\tilde{n}_c$ be the number of channels in M, the broadcast rule is

$$T^c_{i,j} = \begin{cases} M^c_{i,j} & 1 \le c \le \tilde{n}_c \\ M^c_{j,i} & \tilde{n}_c + 1 \le c \le 2\tilde{n}_c. \end{cases}$$

Ptensors0

A `Ptensors0` object $\mathbb{T}$ stores a zeroth order "P-tensors layer", i.e., a collection of several zeroth order P-tensors. For efficiency reasons the P-tensors are stored in a fused `cnine::RtensorPackB` datastructure rather than as individual `Ptensor0` objects.

Derived from: `cnine::RtensorPackB`

VARIABLES

`AtomsPack atoms`
: The reference domain of each P-tensor in $\mathbb{T}$

CONSTRUCTORS

`Ptensors0(const AtomsPack& apack, const int nc, const FILLTYPE& fill, const int dev=0)`
: Create a new zeroth order P-tensor layer $\mathbb{T}$. Each P-tensor in $\mathbb{T}$ will have `nc` channels, and the reference domain of the i'th P-tensor is given by the i'th element of `apack`. The filltype can be `cnine::fill_raw()`, `cnine::fill_zero()`, `cnine::fill_gaussian()` or `cnine::fill_sequential()`. Depending on `dev`, the `Ptensors0` will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

NAMED CONSTRUCTORS

```
Ptensors0 Ptensors0::raw(const AtomsPack& apack, const int nc, const int dev=0)
Ptensors0 Ptensors0::zero(const AtomsPack& apack, const int nc, const int dev=0)
Ptensors0 Ptensors0::gaussian(const AtomsPack& apack, const int nc, const int dev=0)
Ptensors0 Ptensors0::sequential(const AtomsPack& apack, const int nc, const int dev=0)
```
: Create a new layer of zeroth order P-tensors with reference domains given by `apack` and `nc` channels each. Depending on `dev`, the layer will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

REDUCTIONS

```
RtensorPackB reduce0() const
RtensorPackB reduce0(const int offs, const int n) const
RtensorPackB reduce0(const AindexPack& list) const
RtensorPackB reduce0(const AindexPack& list, const int offs, const int n) const
```
: Aggregate the zeroth order messages sent by each P-tensor in an `RtensorPackB`. The individual messages are computed as in the `Ptensor0::reduce0` functions.

BROADCASTING

```
void broadcast0(const RtensorPackB& M) const
void broadcast0(const RtensorPackB& M, const int offs, const int n) const
void broadcast0(const RtensorPackB& M, const AindexPack& list) const
void broadcast0(const RtensorPackB& M, const AindexPack& list, const int offs) const
```
: Broadcast each tensor in `M`, regarded as a zeroth order message, into the corresponding Ptensor in $\mathbb{T}$.

Ptensors1

A `Ptensors1` object $\mathbb{T}$ stores a first order "P-tensors layer", i.e., a collection of several first order P-tensors. For efficiency reasons the P-tensors are stored in a fused `cnine::RtensorPackB` datastructure rather than as individual `Ptensor1` objects.

Derived from: `cnine::RtensorPackB`

VARIABLES

`AtomsPack atoms`
: The reference domain of each P-tensor in $\mathbb{T}$

CONSTRUCTORS

`Ptensors1(const AtomsPack& apack, const int nc, const FILLTYPE& fill, const int dev=0)`
: Create a new first order P-tensor layer $\mathbb{T}$. Each P-tensor in $\mathbb{T}$ will have `nc` channels, and the reference domain of the i'th P-tensor is given by the i'th element of `apack`. The filltype can be `cnine::fill_raw()`, `cnine::fill_zero()`, `cnine::fill_gaussian()` or `cnine::fill_sequential()`. Depending on `dev`, the `Ptensors1` will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

NAMED CONSTRUCTORS

`Ptensors1 Ptensors1::raw(const AtomsPack& apack, const int nc, const int dev=0)`
`Ptensors1 Ptensors1::zero(const AtomsPack& apack, const int nc, const int dev=0)`
`Ptensors1 Ptensors1::gaussian(const AtomsPack& apack, const int nc, const int dev=0)`
`Ptensors1 Ptensors1::sequential(const AtomsPack& apack, const int nc, const int dev=0)`
: Create a new layer of first order P-tensors with reference domains given by `apack` and `nc` channels each. Depending on `dev`, the layer will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

REDUCTIONS

`RtensorPackB reduce0() const`
`RtensorPackB reduce0(const int offs, const int n) const`
`RtensorPackB reduce0(const AindexPack& list) const`
`RtensorPackB reduce0(const AindexPack& list, const int offs, const int n) const`
: Aggregate the zeroth order messages sent by each P-tensor in an `RtensorPackB`. The individual messages are computed as in `Ptensor1::reduce0`.

`RtensorPackB reduce1() const`
`RtensorPackB reduce1(const int offs, const int n) const`
`RtensorPackB reduce1(const AindexPack& list) const`
`RtensorPackB reduce1(const AindexPack& list, const int offs, const int n) const`
: Aggregate the first order messages sent by each P-tensor in an `RtensorPackB`. The individual messages are computed as in `Ptensor1::reduce1`.

BROADCASTING

```
void broadcast0(const RtensorPackB& M) const
void broadcast0(const RtensorPackB& M, const int offs) const
void broadcast0(const RtensorPackB& M, const AindexPack& list) const
void broadcast0(const RtensorPackB& M, const AindexPack& list, const int offs) const
```

Broadcast each first order message in `M` into the corresponding Ptensor in $\mathbb{T}$.

```
void broadcast1(const RtensorPackB& M) const
void broadcast1(const RtensorPackB& M, const int offs) const
void broadcast1(const RtensorPackB& M, const AindexPack& list) const
void broadcast1(const RtensorPackB& M, const AindexPack& list, const int offs) const
```

Broadcast each first order messagge in `M` into the corresponding Ptensor in $\mathbb{T}$.

Ptensors2

A `Ptensors2` object $\mathbb{T}$ stores a second order "P-tensors layer", i.e., a collection of several second order P-tensors. For efficiency reasons the P-tensors are stored in a fused `cnine::RtensorPackB` datastructure rather than as individual `Ptensor1` objects.

Derived from: `cnine::RtensorPackB`

VARIABLES

`AtomsPack atoms`
: The reference domain of each P-tensor in $\mathbb{T}$

CONSTRUCTORS

`Ptensors2(const AtomsPack& apack, const int nc, const FILLTYPE& fill, const int dev=0)`
: Create a new second order P-tensor layer $\mathbb{T}$. Each P-tensor in $\mathbb{T}$ will have `nc` channels, and the reference domain of the i'th P-tensor is given by the i'th element of `apack`. The filltype can be `cnine::fill_raw()`, `cnine::fill_zero()`, `cnine::fill_gaussian()` or `cnine::fill_sequential()`. Depending on `dev`, the `Ptensors2` will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

NAMED CONSTRUCTORS

```
Ptensors2 Ptensors2::raw(const AtomsPack& apack, const int nc, const int dev=0)
Ptensors2 Ptensors2::zero(const AtomsPack& apack, const int nc, const int dev=0)
Ptensors2 Ptensors2::gaussian(const AtomsPack& apack, const int nc, const int dev=0)
Ptensors2 Ptensors2::sequential(const AtomsPack& apack, const int nc, const int dev=0)
```
: Create a new layer of second order P-tensors with reference domains given by `apack` and `nc` channels each. Depending on `dev`, the layer will either be stored on the CPU (`dev=0`) or the GPU (`dev=1`).

REDUCTIONS

```
RtensorPackB reduce0() const
RtensorPackB reduce0(const int offs, const int n) const
RtensorPackB reduce0(const AindexPack& list) const
RtensorPackB reduce0(const AindexPack& list, const int offs, const int n) const
```
: Aggregate the zeroth order messages sent by each P-tensor in an `RtensorPackB`. The individual messages are computed as in `Ptensor2::reduce0`.

```
RtensorPackB reduce1() const
RtensorPackB reduce1(const int offs, const int n) const
RtensorPackB reduce1(const AindexPack& list) const
RtensorPackB reduce1(const AindexPack& list, const int offs, const int n) const
```
: Aggregate the first order messages sent by each P-tensor in an `RtensorPackB`. The individual messages are computed as in `Ptensor2::reduce1`.

```
RtensorPackB reduce2() const
RtensorPackB reduce2(const int offs, const int n) const
RtensorPackB reduce2(const AindexPack& list) const
RtensorPackB reduce2(const AindexPack& list, const int offs, const int n) const
```
: Aggregate the second order messages sent by each P-tensor in an `RtensorPackB`. The individual messages are computed as in `Ptensor2::reduce2`.

BROADCASTING

```
void broadcast0(const RtensorPackB& M) const
void broadcast0(const RtensorPackB& M, const int offs) const
void broadcast0(const RtensorPackB& M, const AindexPack& list) const
void broadcast0(const RtensorPackB& M, const AindexPack& list, const int offs) const
```

Broadcast each first order message in `M` into the corresponding Ptensor in $\mathbb{T}$.

```
void broadcast1(const RtensorPackB& M) const
void broadcast1(const RtensorPackB& M, const int offs) const
void broadcast1(const RtensorPackB& M, const AindexPack& list) const
void broadcast1(const RtensorPackB& M, const AindexPack& list, const int offs) const
```

Broadcast each first order messagge in `M` into the corresponding Ptensor in $\mathbb{T}$.

```
void broadcast2(const RtensorPackB& M) const
void broadcast2(const RtensorPackB& M, const int offs) const
void broadcast2(const RtensorPackB& M, const AindexPack& list) const
void broadcast2(const RtensorPackB& M, const AindexPack& list, const int offs) const
```

Broadcast each second order messagge in `M` into the corresponding Ptensor in $\mathbb{T}$.

Helper classes

Atoms

An `Atoms` object $\mathcal{D}$ stores an ordered set of atoms (positive integers). In particular, it is used to define the reference domain of P-tensors.

Derived from: `std::vector<int>`

CONSTRUCTORS

`Atoms(const initializer_list<int>& list)`
`Atoms(const vector<int>& list)`
Construct $\mathcal{D}$ from `list`.

NAMED CONSTRUCTORS

`Atoms Atoms::sequential(const int n)`
Set $\mathcal{D} = \{1, 2, \ldots, n-1\}$.

ACCESS

`int operator[](const int i) const`
Return the i'th atom.

`int operator()(const int x) const`
Return the index of the atom x.

`bool includes(const int x) const`
Return `true` if x is amongst the atoms otherwise return `false`.

`void foreach(const std::function<void(const int)> lambda) const`
Call `lambda` for each atom.

OPERATIONS

`Atoms intersect(const Atoms y) const`
Return the intersection of this set of atoms with `y`.

AtomsPack

An `AtomsPack` object stores a collection of ordered sets of atoms (positive integers). In particular, it is used to collectively define the reference domains of all the P-tensors in a P-tensor layer.

Derived from: `cnine::array_pool<int>`

CONSTRUCTORS

`AtomsPack(const initializer_list<initializer_list<int>>& list_of_lists)`
`AtomsPack(const vector<vector<<int>>& list_of_lists)`
Construct the `AtomsPack` from a list of lists of atoms.

ACCESS

`Atoms operator[](const int i) const`
Return the i'th ordered set as an `Atoms` objects.

AindexPack

An `AindexPack` object stores a collection of objects $\{(t^j, (i^j_1, \ldots, i^j_{k^j}))\}_j$ where t^j is the index of a P-tensor in a P-tensor layer and $(i^j_1, \ldots, i^j_{k^j})$ is an ordered set of indices of the given tensor. `AindexPack`s are used to specify which tensors which tensors to extract messages from and which subset of indices of those tensors the messages are to be defined with respect to. `AindexPack`s are also used to specify which tensors in a given layer a pack of messages is to be broadcast to, and with respect to which subset of indices.

Derived from: `cnine::array_pool<int>`

ACCESS

`int tix(const int j) const`
: Return the tensor index t^j in the j'th entry of the pack.

`int nix(const int j) const`
: Return the number of indices k^j in the j'th entry of the pack.

`vector<int> ix(const int j) const`
: Return the vector $(i^j_1, \ldots, i^j_{k^j})$ of indices in the j'th entry.

`vector<int> ix(const int j, const int p) const`
: Return the p'th index i^j_p in the j'th entry.

`void push_back(const int t, vector<int> indices)`
: Add a new entry $(t, (i_1, \ldots, i_k))$ to the pack.

Hgraph

An `Hgraph` object stores an unlabeled or labeled, undirected graph in sparse format.

Derived from: `cnine::SparseRmatrix`

CONSTRUCTORS

`Hgraph(const int n, const initializer_list<pair<int,int> >& list)`
Define an unlabeled graph with n vertices and edges defined by `list`.

`Hgraph(const int n, const initializer_list<pair<int,int> >& list, const RtensorA& labels)`
Define a labeled graph with n vertices, edges defined by `list` and labels defined by `labels`.

NAMED CONSTRUCTORS

`static Hgraph Hgraph::edge_index(const cnine::RtensorA& E, int n=-1)`
Create an `Hgraph` from an edge-index matrix E.

ACCESS

`vector<int> neighbors(const int i) const`
Return the list of vectors of vertex i.

OPERATIONS

`AtomsPack nhoods(const int r) const`
Return an `AtomsPack` object, the i'th element of which consists of the vertices in the r-neighborhood of vertex i.

`AtomsPack merge(const AtomsPack& x) const`

```
pair<AindexPack,AindexPack>intersects(const AtomsPack& inputs,
          const AtomsPack& outputs, const bool self=0) const
```