tensorD Documentation

Release 0.2

Siqi Liang

May 06, 2019

Contents

TensorD is A tensor decomposition library in TensorFlow

CHAPTER 1

Introduction

TensorD: A Tensor Decomposition Library in Tensorflow

TensorD is a tensor decomposition library built on TensorFlow. It provides basic decomposition methods, such as Tucker decomposition and CANDECOMP/PARAFAC (CP) decomposition, as well as new decomposition methods developed recently, for example, Pairwise Interaction Tensor Decomposition. Except these utilities, other features of *TensorD}* include that its internal functions all use the interface provided by Tensorflow, and it can be embedded directly into Tensorflow-based codes when needed.

CHAPTER 2

User Guide

To start with this guidance, basic modules are needed:

```
>>> import tensorflow as tf
>>> import numpy as np
```

2.1 Tensor Types

A *tensor* is a multidimensional array. For the sake of different applications, we will introduce 4 different data types to store tensors.

2.1.1 Dense Tensor

`factorizer.base.DTensor` Class is used to store general high-order tensors, especially dense tensors. This data type accepts 2 kinds of tensor data, both `tf.Tensor` and `np.ndarray`.

Let's take for this example the tensor $\mathcal{X} \in \mathbb{R}^{3\times4\times2}$ defined by its frontal slices:

$$X_1 = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}, \quad X_2 = \begin{bmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{bmatrix}$$

To create `DTensor` with `np.ndarray`:

```
>>> from factorizer.base.type import DTensor
>>> tensor = np.array([[[1, 13], [4, 16], [7, 19], [10, 22]], [[2, 14], [5, 17], [8,␣
↪20], [11, 23]], [[3, 15], [6, 18], [9, 21], [12, 24]]])
>>> dense_tensor = DTensor(tensor)
```

To create `DTensor` with `tf.Tensor`:

```
>>> from factorizer.base.type import DTensor
>>> tensor = tf.constant([[[1, 13], [4, 16], [7, 19], [10, 22]], [[2, 14], [5, 17],␣
↪[8, 20], [11, 23]], [[3, 15], [6, 18], [9, 21], [12, 24]]])
>>> dense_tensor = DTensor(tensor)
```

Important: `DTensor.T` is the tensor data stored in `tf.Tensor` form, rather than the transpose of the original tensor.

2.1.2 Kruskal Tensor

`factorizer.base.KTensor` Class is designed for Kruskal tensors in CP model. Let's take a look at a 2-way tensor defined as below:

$$\mathcal{X} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

```
>>> X = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])    # the shape of tensor X is␣
↪(3, 4)
```

The CP decomposition can factorize $\mathcal{X}$ into 2 component rank-one tensors, and the CP model can be expressed as

$$\mathcal{X} \approx [\![\mathbf{A}, \mathbf{B}]\!] \equiv \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r.$$

If we assume the columns of $\mathbf{A}$ and $\mathbf{B}$ are normalized to length one with the weights absorbed into the vector $\boldsymbol{\lambda} \in \mathbb{R}^R$ so that

$$\mathcal{X} \approx [\![\boldsymbol{\lambda}; \mathbf{A}, \mathbf{B}]\!] \equiv \sum_{r=1}^{R} \lambda_r \, \mathbf{a}_r \circ \mathbf{b}_r$$

where $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_R]$, $\mathbf{B} = [\mathbf{b}_1, \cdots, \mathbf{b}_R]$.

Here we use singular value decomposition (SVD) to obtain the factor matrices (CP decomposition actually can be considered higher-order generation of matrix SVD):

```
>>> from factorizer.base.type import KTensor
>>> u,s,v = np.linalg.svd(X, full_matrices=False)    # X is equal to np.dot(u, np.
↪dot(np.diag(s), v)), that is X = u * diag(s) * v
```

Then we use 2 factor matrices and $\boldsymbol{\lambda}$ to create a `factorizer.base.KTensor` object:

```
>>> A = u    # the shape of A is (3, 3)
>>> B = v.T    # the shape of B is (4, 3)
>>> kruskal_tensor = KTensor([A, B], s)    # the shape of s is (3,)
```

Notice that the first argument `factors` is a list of `tf.Tensor` objects or `np.ndarray` objects representing factor matrices, and the order of these matrices must be fixed.

If you want to get the factor matrices with `KTensor` object:

```
>>> kruskal_tensor.U
[<tf.Tensor 'Const:0' shape=(3, 3) dtype=float64>,
 <tf.Tensor 'Const_1:0' shape=(4, 3) dtype=float64>]
```

If you want to get the vector $\boldsymbol{\lambda}$ with `KTensor` object:

```
>>> kruskal_tensor.lambdas
<tf.Tensor 'Reshape:0' shape=(3, 1) dtype=float64>
```

We also offer class method `KTensor.extract()` to retrieve original tensor with `KTensor` object:

```
>>> original_tensor = tf.Session().run(kruskal_tensor.extract())
>>> original_tensor
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])
```

To make sure `original_tensor` is equal to the tensor $\mathcal{X}$, you just need to run:

```
>>> np.testing.assert_array_almost_equal(X, original_tensor)
# no Traceback means these two np.ndarray objects are exactly the same
```

Following Kolda[1], for a general N th-order tensor, $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, the CP decomposition is

$$\mathcal{X} \approx [\![\boldsymbol{\lambda}; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!] \equiv \sum_{r=1}^{R} \lambda_r \, \mathbf{a}_r^{(1)} \circ \mathbf{a}_r^{(2)} \circ \cdots \circ \mathbf{a}_r^{(N)}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^R$ and $\mathbf{A}^{(n)} \in \mathbb{R}^{I_1 \times R}$ for $n = 1, \ldots, N$.

The following code can be used to create a N th-order Kruskal tensor object:

```
>>> lambdas = tf.constant([l1, l2, ..., lR],shape=(R,1))     # lambdas must be a␣
↪column vector
>>> A1 = np.random.rand(I1, R)
>>> A2 = np.random.rand(I2, R)
...
>>> AN = np.random.rand(IN, R)
>>> factors = [A1, A2, ..., AN]
>>> N_kruskal_tensor = KTensor(factors, lambdas)
```

2.1.3 Tucker Tensor

`factorizer.base.TTensor` Class is designed for Tucker tensors in Tucker decomposition.

Given an N -way tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, the Tucker model can be expressed as

$$\mathcal{X} = \mathcal{G} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)} = [\![\mathcal{G}; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots \mathbf{A}^{(N)}]\!],$$

where $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_N}$, and $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R_n}$.

To create the corresponding Tucker tensor, you just need to run:

```
>>> from factorizer.base.type import TTensor
>>> G = tf.constant(np.random.rand(R1, R2, ..., RN))
>>> A1 = np.random.rand(I1, R1)
>>> A2 = np.random.rand(I2, R2)
...
>>> AN = np.random.rand(IN, RN)
>>> factors = [A1, A2, ..., AN]
>>> tucker_tensor = TTensor(G, factors)
```

[1] Tamara G. Kolda and Brett W. Bader, "Tensor Decompositions and Applications", SIAM REVIEW, vol. 51, n. 3, pp. 455-500, 2009.

Important: All elements in `factors` as a whole should be either `tf.Tensor` objects or `np.ndarray` objects.

To get core tensor $\mathcal{G}$ given a `factorizer.base.TTensor` object:

```
>>> tucker_tensor.g
# <tf.Tensor 'Const_1:0' shape=(R1, R2, ..., RN) dtype=float64>
```

To get factor matrices given a `TTensor` object:

```
>>> tucker_tensor.U
#[<tf.Tensor 'Const_2:0' shape=(I1, R1) dtype=float64>,
# <tf.Tensor 'Const_3:0' shape=(I2, R2) dtype=float64>,
# ...
# <tf.Tensor 'Const_{N-1}:0' shape=(IN, RN) dtype=float64>]
```

To get the order of the tensor:

```
>>> tucker_tensor.order
# N
```

To retrieve original tensor, you just need to run:

```
>>> tf.Session().run(tucker_tensor.extract())
# an np.ndarray with shape (I1, I2, ..., IN)
```

2.1.4 References

2.2 Basic Operations

Most operations we offer return results with `tf.Tensor` form, except some build-in class methods in our module.

To begin with, load in operation module:

```
import factorizer.base.ops as ops
```

2.2.1 Basic Operations with Matrices

Hadamard Products

The *Hadamard product* is the elementwise matrix product. Given matrices $\mathbf{A}$ and $\mathbf{B}$, both of size $I \times J$, their Hadamard product is denoted by $\mathbf{A} * \mathbf{B}$. The result is defined by

$$\mathbf{A} * \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1J}b_{1J} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2J}b_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}b_{I1} & a_{I2}b_{I2} & \cdots & a_{IJ}b_{IJ} \end{bmatrix}$$

For instance, using matrices $\mathbf{A}$ and $\mathbf{B}$ defined as

$$\mathbf{A} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$$

Using `DTensor` to store matrices, $\mathbf{A} * \mathbf{B}$ can be performed as:

```
>>> A = DTensor(tf.constant([[1,4,7], [2,5,8],[3,6,9]]))
>>> B = DTensor(tf.constant([[2,3,4], [2,3,4],[2,3,4]]))
>>> result = A*B    # result is a DTensor with shape (3,3)
>>> tf.Session().run(result.T)
array([[ 2, 12, 28],
       [ 4, 15, 32],
       [ 6, 18, 36]], dtype=int32)
```

Using `tf.Tensor` to store matrices, $\mathbf{A} * \mathbf{B}$ can be performed as:

```
>>> A = tf.constant([[1,4,7], [2,5,8],[3,6,9]])
>>> B = tf.constant([[2,3,4], [2,3,4],[2,3,4]])
>>> tf.Session().run(ops.hadamard([A,B]))
array([[ 2, 12, 28],
       [ 4, 15, 32],
       [ 6, 18, 36]], dtype=int32)
```

`hadamard()` also supports the Hadamard products of more than two matrices:

```
>>> C = tf.constant(np.random.rand(3,3))
>>> D = tf.constant(np.random.rand(3,3))
>>> tf.Session().run(ops.hadamard([A, B, C, D], skip_matrices_index=[1]))
    # the result is equal to tf.Session().run(ops.hadamard([A, C, D]))
```

Kronecker Products

The *Kronecker product* of matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{K \times L}$ is denoted by $\mathbf{A} \otimes \mathbf{B}$. The result is a matrix of size $(IK) \times (JL)$ (See Kolda's[1] for more details).

For example, matrices $\mathbf{A}$ and $\mathbf{B}$ is defined as

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

To perform $\mathbf{A} \otimes \mathbf{B}$ with `tf.Tensor` objects:

```
>>> A = tf.constant([[1,2,3,4],[5,6,7,8],[9,10,11,12]])    # the shape of A is (3, 4)
>>> B = tf.constant([[1,1,1,1,1],[2,2,2,2,2]])    # the shape of B is (2, 5)
>>> tf.Session().run(ops.kron([A, B]))
# the shape of result is (6, 20)
array([[ 1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  4,  4,  4,  4,
↪4],
       [ 2,  2,  2,  2,  2,  4,  4,  4,  4,  4,  6,  6,  6,  6,  6,  8,  8,  8,  8,
↪8],
       [ 5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  8,  8,  8,  8,
↪8],
       [10, 10, 10, 10, 10, 12, 12, 12, 12, 12, 14, 14, 14, 14, 14, 16, 16, 16, 16,
↪16],
       [ 9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 12, 12, 12, 12,
↪12],
       [18, 18, 18, 18, 18, 20, 20, 20, 20, 20, 22, 22, 22, 22, 22, 24, 24, 24, 24,
↪24]], dtype=int32)
```

[1] Tamara G. Kolda and Brett W. Bader, "Tensor Decompositions and Applications", SIAM REVIEW, vol. 51, n. 3, pp. 455-500, 2009.

To perform $\mathbf{B} \otimes \mathbf{A}$:

```
>>> tf.Session().run(ops.kron([A, B], reverse=True))
# the shape of result is (6, 20)
array([[ 1,  2,  3,  4,  1,  2,  3,  4,  1,  2,  3,  4,  1,  2,  3,  4,  1,  2,  3,
↪4],
       [ 5,  6,  7,  8,  5,  6,  7,  8,  5,  6,  7,  8,  5,  6,  7,  8,  5,  6,  7,
↪8],
       [ 9, 10, 11, 12,  9, 10, 11, 12,  9, 10, 11, 12,  9, 10, 11, 12,  9, 10, 11,
↪12],
       [ 2,  4,  6,  8,  2,  4,  6,  8,  2,  4,  6,  8,  2,  4,  6,  8,  2,  4,  6,
↪8],
       [10, 12, 14, 16, 10, 12, 14, 16, 10, 12, 14, 16, 10, 12, 14, 16, 10, 12, 14,
↪16],
       [18, 20, 22, 24, 18, 20, 22, 24, 18, 20, 22, 24, 18, 20, 22, 24, 18, 20, 22,
↪24]], dtype=int32)
```

It might seem useless when using `reverse=True` to calculate the Kronecker product of two matrices, considering `ops.kron([B, A])` also do the same work, but it is considerable efficient to perform $X_1 \otimes X_2 \otimes \cdots \otimes X_N$ using `reverse=True` when given a list of `tf.Tensor` objects `matrices = [X_1, X_2, ..., X_N]`:

```
>>> tf.Session().run(ops.kron(matrices, reverse=True))
```

If the matrices are given in `DTensor` form:

```
>>> A = DTensor(tf.constant([[1,2,3,4],[5,6,7,8],[9,10,11,12]]))
```

Then $\mathbf{A} \otimes \mathbf{B}$ can be performed as:

```
>>> dtensor_B = DTensor(tf.constant([[1,1,1,1,1],[2,2,2,2,2]]))
>>> tf.Session().run(A.kron(dtensor_B).T)    # A.kron(dtensor_B) returns a DTensor
array([[ 1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  4,  4,  4,  4,
↪4],
       [ 2,  2,  2,  2,  2,  4,  4,  4,  4,  4,  6,  6,  6,  6,  6,  8,  8,  8,  8,
↪8],
       [ 5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  8,  8,  8,  8,
↪8],
       [10, 10, 10, 10, 10, 12, 12, 12, 12, 12, 14, 14, 14, 14, 14, 16, 16, 16, 16,
↪16],
       [ 9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 12, 12, 12, 12,
↪12],
       [18, 18, 18, 18, 18, 20, 20, 20, 20, 20, 22, 22, 22, 22, 22, 24, 24, 24, 24,
↪24]], dtype=int32)
```

or

```
>>> tf_B = tf.constant([[1,1,1,1,1],[2,2,2,2,2]])
>>> tf.Session().run(A.kron(tf_B).T)    # A.kron(tf_B) returns a DTensor
array([[ 1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  4,  4,  4,  4,
↪4],
       [ 2,  2,  2,  2,  2,  4,  4,  4,  4,  4,  6,  6,  6,  6,  6,  8,  8,  8,  8,
↪8],
       [ 5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  8,  8,  8,  8,
↪8],
       [10, 10, 10, 10, 10, 12, 12, 12, 12, 12, 14, 14, 14, 14, 14, 16, 16, 16, 16,
↪16],
       [ 9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 12, 12, 12, 12,
↪12],
```

(continues on next page)

(continued from previous page)

```
       [18, 18, 18, 18, 18, 20, 20, 20, 20, 20, 22, 22, 22, 22, 22, 24, 24, 24, 24,␣
↪24]], dtype=int32)
```

Khatri-Rao Products

The *Khatri-Rao product* can be expressed in Kronecker product form. Given matrices $\mathbf{A} \in \mathbb{R}^{I \times K}$ and $\mathbf{B} \in \mathbb{R}^{J \times K}$, their Khatri-Rao product is denoted by $\mathbf{A} \odot \mathbf{B}$. The result is a matrix of size $(IJ) \times (K)$ and defined by

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b}_1 & \mathbf{a}_2 \otimes \mathbf{b}_2 & \cdots & \mathbf{a}_K \otimes \mathbf{b}_K \end{bmatrix}$$

Let's take a look at matrices $\mathbf{A}$ and $\mathbf{B}$ defined as

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \end{bmatrix}$$

To perform $\mathbf{A} \odot \mathbf{B}$:

```
>>> A = tf.constant([[1,2,3,4],[5,6,7,8],[9,10,11,12]])    # the shape of A is (3, 4)
>>> B = tf.constant([[1,1,1,1],[2,2,2,2]])    # the shape of B is (2, 4)
>>> tf.Session().run(ops.khatri([A, B]))
# the shape of the result is (6, 4)
array([[ 1,  2,  3,  4],
       [ 2,  4,  6,  8],
       [ 5,  6,  7,  8],
       [10, 12, 14, 16],
       [ 9, 10, 11, 12],
       [18, 20, 22, 24]], dtype=int32)
```

`khatri()` function also offers `skip_matrices_index` to ignore specific matrices in the computation. For example, given `matrices = [A, B, C, D]` to calculate $\mathbf{A} \odot \mathbf{B} \odot \mathbf{D}$:

```
>>> C = tf.constant(np.random.rand(4,4))
>>> D = tf.constant(np.random.rand(5,4))
>>> matrices = [A, B, C, D]
>>> tf.Session().run(ops.khatri(matrices, skip_matrices_index=[2]))
# the shape of the result is (30, 4)
```

To obtain the result of $\mathbf{D} \odot \mathbf{C} \odot \mathbf{B} \odot \mathbf{A}$:

```
>>> tf.Session().run(ops.khatri(matrices, reverse=True))
# the shape of the result is (120, 4)
```

`DTensor` class also offers class method `DTensor.khatri()` which accepts only one single `DTensor` object or `tf.Tensor` object:

```
>>> A = DTensor(tf.constant([[1,2,3,4],[5,6,7,8],[9,10,11,12]]))
>>> B = tf.constant([[1,1,1,1],[2,2,2,2]])
>>> tf.Session().run(A.khatri(B).T)
# the shape of the result is (6, 4)
array([[ 1,  2,  3,  4],
       [ 2,  4,  6,  8],
       [ 5,  6,  7,  8],
       [10, 12, 14, 16],
       [ 9, 10, 11, 12],
       [18, 20, 22, 24]], dtype=int32)
```

2.2.2 Basic Operations with Tensors

Addition & Subtraction

Given a `DTensor` object, it is easy to perform addition and subtraction.

```
>>> X = DTensor(tf.constant([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]]))    # the shape␣
↪of tensor X is (2, 2, 3)
>>> Y = DTensor(tf.constant([[[-1,-2,-3],[-4,-5,-6]],[[-7,-8,-9],[-10,-11,-12]]]))
↪# the shape of tensor Y is (2, 2, 3)
>>> sum_X_Y = X + Y    # sum_X_Y is a DTensor
>>> sub_X_Y = X - Y    # sub_X_Y is a DTensor
>>> tf.Session().run(sum_X_Y.T)
array([[[0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0]]], dtype=int32)
>>> tf.Session().run(sub_X_Y.T)
array([[[ 2,  4,  6],
        [ 8, 10, 12]],

       [[14, 16, 18],
        [20, 22, 24]]], dtype=int32)
```

The second operand can also be a `tf.Tensor` object:

```
>>> Z = tf.constant([[[-1,-2,-3],[-4,-5,-6]],[[-7,-8,-9],[-10,-11,-12]]])    # the␣
↪shape of tensor Z is (2, 2, 3)
>>> sum_X_Z = X + Z    # sum_X_Z is a DTensor
>>> sub_X_Z = X - Z    # sub_X_Z is a DTensor
>>> tf.Session().run(sum_X_Z.T)
array([[[0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0]]], dtype=int32)
>>> tf.Session().run(sub_X_Z.T)
array([[[ 2,  4,  6],
        [ 8, 10, 12]],

       [[14, 16, 18],
        [20, 22, 24]]], dtype=int32)
```

Inner Products

The *inner product* of two same-sized tensor $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is the sum of products of their entries, which can be denoted as $\langle \mathcal{X}, \mathcal{Y} \rangle$.

Given tensor $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{3 \times 3 \times 2}$ defined by their frontal slices:

$$X_1 = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}, \quad X_2 = \begin{bmatrix} 10 & 13 & 16 \\ 11 & 14 & 17 \\ 12 & 15 & 18 \end{bmatrix}$$

$$Y_1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad Y_2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

```
>>> X = tf.constant(np.array([[[1,10],[4,13],[7,16]], [[2,11],[5,14],[8,17]], [[3,12],
↪[6,15],[9,18]]]))    # the shape of X is (3, 3, 2)
>>> Y = tf.constant(np.array([[[1,1],[1,1],[1,1]], [[1,1],[1,1],[1,1]], [[1,1],[1,1],
↪[1,1]]]))    # the shape of Y is (3, 3, 2)
```

To calculate $\langle \mathcal{X}, \mathcal{Y} \rangle$:

```
>>> tf.Session().run(ops.inner(X, Y))
171
```

Warning: Notice that `ops.inner()` function does not support implicit type-casting, so be careful when using tensors of different `dtype` !

Vectorization & Reconstruction

The *vectorization* of a tensor is ordering the tensor into a vector. And the process transforming the vector back to the tensor is called *reconstruction* or reshaping.

Take the tensor $\mathcal{X} \in \mathbb{R}^{3\times3\times2}$ defined before as example.

```
>>> X = tf.constant(np.array([[[1,10],[4,13],[7,16]], [[2,11],[5,14],[8,17]], [[3,12],
↪[6,15],[9,18]]]))    # the shape of X is (3, 3, 2)
>>> vec = ops.vectorize(X)
>>> tf.Session().run(vec)
array([ 1, 10,  4, 13,  7, 16,  2, 11,  5, 14,  8, 17,  3, 12,  6, 15,  9, 18])
```

To reconstruct the vector:

```
>>> tf.Session().run(ops.vec_to_tensor(vec,(3,3,2)))
array([[[ 1, 10],
        [ 4, 13],
        [ 7, 16]],

       [[ 2, 11],
        [ 5, 14],
        [ 8, 17]],

       [[ 3, 12],
        [ 6, 15],
        [ 9, 18]]])
```

Unfolding & Folding

Unfolding, also known as *matricization*, is the process of reordering the elements of an N -way array into a matrix. Here we call operation **mode-n matricization** as **unfolding** in default.

Let the frontal slices of $\mathcal{X} \in \mathbb{R}^{3\times4\times2}$ be

$$X_1 = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}, \quad X_2 = \begin{bmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{bmatrix}$$

```
>>> X = tf.constant([[[1, 13], [4, 16], [7, 19], [10, 22]], [[2, 14], [5, 17], [8,␣
↪20], [11, 23]], [[3, 15], [6, 18], [9, 21], [12, 24]]])    # the shape of X is (3,␣
↪4, 2)
```

To get the mode-1 matricization of tensor $\mathcal{X}$:

```
>>> tf.Session().run(ops.unfold(X, 0))
array([[ 1,  4,  7, 10, 13, 16, 19, 22],
       [ 2,  5,  8, 11, 14, 17, 20, 23],
       [ 3,  6,  9, 12, 15, 18, 21, 24]], dtype=int32)
```

To get the mode-2 matricization of tensor $\mathcal{X}$:

```
>>> tf.Session().run(ops.unfold(X, 1))
array([[ 1,  2,  3, 13, 14, 15],
       [ 4,  5,  6, 16, 17, 18],
       [ 7,  8,  9, 19, 20, 21],
       [10, 11, 12, 22, 23, 24]], dtype=int32)
```

To get the mode-3 matricization of tensor $\mathcal{X}$:

```
>>> tf.Session().run(ops.unfold(X, 2))
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]], dtype=int32)
```

For a `DTensor` object, class method `DTensor.unfold()` is available:

```
>>> X = DTensor(tf.constant([[[1, 13], [4, 16], [7, 19], [10, 22]], [[2, 14], [5, 17],
↪ [8, 20], [11, 23]], [[3, 15], [6, 18], [9, 21], [12, 24]]]))
>>> tf.Session().run(X.unfold(mode=0).T)    # mode-1 matricization, X.unfold(mode=0)␣
↪return a DTensor
array([[ 1,  4,  7, 10, 13, 16, 19, 22],
       [ 2,  5,  8, 11, 14, 17, 20, 23],
       [ 3,  6,  9, 12, 15, 18, 21, 24]], dtype=int32)
```

General Matricization

According to Kolda's[2], *general matricization* can flatten a high-order tensor into a matrix with size defined with row indices and column indices.

Given tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, if we want to rearrange it into a matrix with size $J_1 \times J_2$,

$$where \quad J_1 = \prod_{k=1}^{K} I_{r_k} \quad and \quad J_2 = \prod_{\ell=1}^{L} I_{c_\ell}.$$

The set $\{r_1, \cdots, r_K\}$ defines those indices that will mapped to the row indices of the resulting matrix and the set $\{c_1, \cdots, c_L\}$ defines those indices that will mapped to the column indices.

Note: The order of $\{r_1, \cdots, r_K\}$ or $\{c_1, \cdots, c_L\}$ is not necessarily ascending or descending.

Take a look at tensor $\mathcal{X}$ defined as:

[2] Tamara G. Kolda and Brett W. Bader, "Algorithm 862: MATLAB tensor classes for fast algorithm prototyping", ACM Trans. Math. Softw, 32 (4): 635-653 (2006)

```
>>> X = tf.constant([[[1, 13], [4, 16], [7, 19], [10, 22]], [[2, 14], [5, 17], [8,␣
↪20], [11, 23]], [[3, 15], [6, 18], [9, 21], [12, 24]]])    # the shape of X is (3,␣
↪4, 2)
```

To mapped $\mathcal{X}$ into matrix of size $(4 \times 3) \times 2 = 12 \times 2$:

```
>>> r_axis = [1,0]    # indices of row
>>> c_axis = 2    # indices of column
>>> mat = ops.t2mat(X, r_axis, c_axis)    # mat is a tf.Tensor
>>> tf.Session().run(mat)
array([[ 1, 13],
       [ 2, 14],
       [ 3, 15],
       [ 4, 16],
       [ 5, 17],
       [ 6, 18],
       [ 7, 19],
       [ 8, 20],
       [ 9, 21],
       [10, 22],
       [11, 23],
       [12, 24]], dtype=int32)
```

function `ops.t2mat()` can also perform *mode-n matricization* mapping indices appropriately:

To perform Kolda-type mode-2 unfolding:

```
>>> mat1 = ops.t2mat(X, 1, [2,0])
```

To perform LMV-type mode-2 unfolding:

```
>>> mat2 = ops.t2mat(X, 1, [0,2])
```

`DTensor` also offers class method:

```
>>> X = DTensor(tf.constant([[[1, 13], [4, 16], [7, 19], [10, 22]], [[2, 14], [5, 17],
↪ [8, 20], [11, 23]], [[3, 15], [6, 18], [9, 21], [12, 24]]]))    # the shape of X␣
↪is (3, 4, 2)
>>> mat3 = X.t2mat([1,0], 2)    # mat3 is a DTensor
>>> tf.Session().run(mat3.T)
array([[ 1, 13],
       [ 2, 14],
       [ 3, 15],
       [ 4, 16],
       [ 5, 17],
       [ 6, 18],
       [ 7, 19],
       [ 8, 20],
       [ 9, 21],
       [10, 22],
       [11, 23],
       [12, 24]], dtype=int32)
```

The *n* -mode Products

The *n-mode product* of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ **with** a matrix $\mathbf{A} \in \mathbb{R}^{J \times I_n}$ is denoted by $\mathcal{X} \times_n \mathbf{A}$ and

is of size $I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N$.

Let the frontal slices of $\mathcal{X} \in \mathbb{R}^{3\times4\times2}$ be

$$X_1 = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}, \quad X_2 = \begin{bmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{bmatrix}$$

```
>>> X = tf.constant([[[1, 13], [4, 16], [7, 19], [10, 22]], [[2, 14], [5, 17], [8,␣
↪20], [11, 23]], [[3, 15], [6, 18], [9, 21], [12, 24]]])    # the shape of X is (3,␣
↪4, 2)
```

And Let $\mathbf{A}$ be

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}.$$

```
>>> A = tf.constant([[1,3,5], [2,4,6]])
```

Then the product $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{A} \in \mathbb{R}^{2\times4\times2}$ is

$$Y_1 = \begin{bmatrix} 22 & 49 & 76 & 103 \\ 28 & 64 & 100 & 136 \end{bmatrix}, \quad Y_2 = \begin{bmatrix} 130 & 157 & 184 & 211 \\ 172 & 208 & 244 & 280 \end{bmatrix}$$

Now run code below to perform the calculation:

```
>>> Y = tf.Session().run(ops.ttm(X,[A],[0]))
>>> Y[:,:,0]    # the first frontal slice of Y
array([[ 22,  49,  76, 103],
       [ 28,  64, 100, 136]], dtype=int32)
>>> Y[:,:,1]    # the second frontal slice of Y
array([[130, 157, 184, 211],
       [172, 208, 244, 280]], dtype=int32)
```

It is often desirable to calculate the prduct of a tensor and a sequence of matrices. Let $\mathcal{X}$ be an $\mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$ tensor, and let $\mathbf{A}^{(n)} \in \mathbb{R}^{J_n\times I_n}$ for $n = 1, 2, \cdots, N$. The the sequence of products

$$\mathcal{Y} = \mathcal{X} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)}$$

To perform this calculation:

```
>>> A1 = tf.constant(np.random.rand(J1,I1))
>>> A2 = tf.constant(np.random.rand(J2,I2))
...
>>> AN = tf.constant(np.random.rand(JN,IN))
>>> X = tf.constant(np.random.rand(I1, I2, ..., IN))
>>> seq_A = [A1, A2, ..., AN]    # map all matrices into a list
>>> B = ops.ttm(X, seq_A, axis=range(N))
>>> tf.Session().run(B)
```

If needed, arguments `transpose` and `skip_matrices_index` are also available.

Tensor Contraction

The *tensor contraction* multiplies two tensors along the given axis. Let tensor $\mathcal{X} \in \mathbb{R}^{I_1\times\cdots\times I_M\times J_1\times\cdots\times J_N}$, and tensor $\mathcal{Y} \in \mathbb{R}^{I_1\times\cdots\times I_M\times K_1\times\cdots\times K_P}$, then multiplying both tensors along the first M modes can be denoted by $\mathcal{Z} = \langle\mathcal{X},\mathcal{Y}\rangle_{\{1,\ldots,M;1,\ldots,M\}}$. And the size of $\mathcal{Z}$ is $J_1 \times \cdots \times J_N \times K_1 \times \cdots \times K_P$. See Cichocki's[3] for more details.

[3] Cichocki, Andrzej. "Era of big data processing: A new approach via tensor networks and tensor decompositions." arXiv preprint arXiv:1403.2048 (2014).

To perform tensor contraction:

```
>>> X = tf.constant(np.random.rand(I1, ..., IM, J1, ..., JN))
>>> Y = tf.constant(np.random.rand(I1, ..., IM, K1, ..., KP))
>>> Z = ops.mul(X, Y,  a_axis=[0,1,...,M-1], b_axis=[0,1,...,M-1])     # either a_axis␣
↪or b_axis can also be tuple or a single integer
>>> tf.Session().run(Z)     # Z is a tf.Tensor object
```

The arguments `a_axis` and `b_axis` specifying the modes of $\mathcal{X}$ and $\mathcal{Y}$ for contraction are not consecutive necessarily, but the sizes of corresponding dimensions must be equal.

Classic matrix multiplication can also be performed with `mul()`:

```
>>> A = tf.constant(np.random.rand(4,5))     # matrix A with shape (4, 5)
>>> B = tf.constant(np.random.rand(5,4))     # matrix B with shape (5, 4)
>>> C = ops.mul(A, B, 1, 0)     # same as tf.matmul(A, B, transpose_a=False, transpose_
↪b=False)
>>> D = ops.mul(A, B, 0, 1)     # same as tf.matmul(A, B, transpose_a=True, transpose_
↪b=True)
```

Class `DTensor` also provides class method `DTensor.mul()`:

```
>>> X_dtensor = DTensor(np.random.rand(4,5))
>>> Y_dtensor = DTensor(np.random.rand(5,4))
>>> Z_dtensor = X_dtensor.mul(Y_dtensor, a_axis=1, b_axis=0)
# same as DTensor( tf.matmul(X_dtensor.T, Y_dtensor.T, transpose_a=False, transpose_
↪b=False) )
```

The argument `tensor` of `DTensor.mul()` only accepts `DTensor` object.

2.2.3 References

2.3 Tensor Decomposition

In this section, we will show how to perform tensor decomposition. Refer to[1][2] for more mathematical details.

In the following subsections, we present examples of four basic decomposition and pairwise interaction tensor decomposition, and the data flow graph generated by using TensorBoard, which is a visual tool offered by TensorFlow. We also provide example for multiple decompositions in one script.

Before using each decomposition model, import `Enviroment` and `Provider` for decomposition model, and `DataGenerator` for synthetic tensor generation:

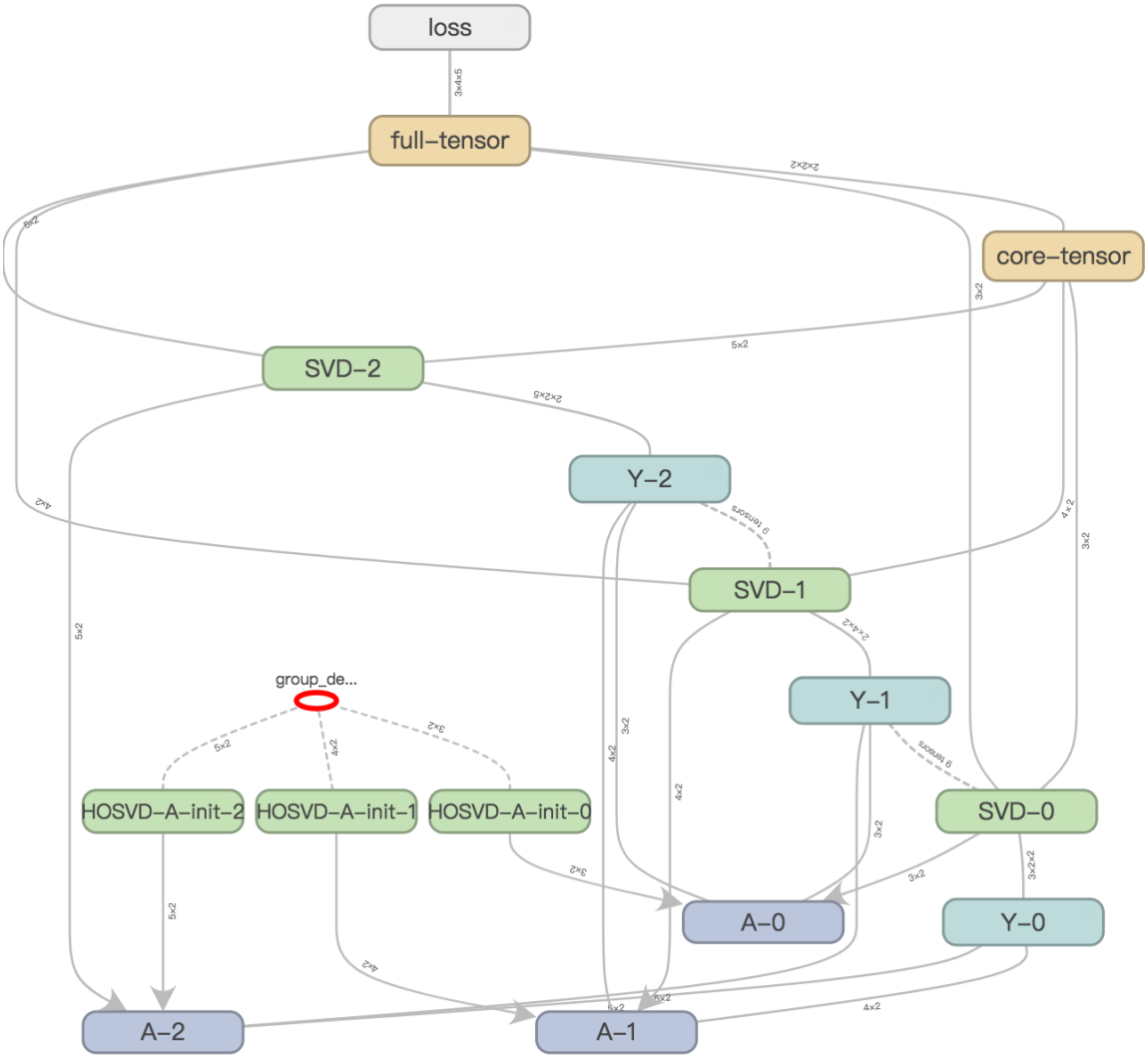
```
from tensorD.factorization.env import Environment
from tensorD.dataproc.provider import Provider
from tensorD.demo.DataGenerator import *
```

2.3.1 The Tucker decomposition

$$\chi = g \times_1 \mathbf{U} \times_2 \mathbf{V} \times_3 \mathbf{W} = \sum_r \sum_s \sum_t g_{rst}\, \mathbf{u}_r \circ \mathbf{v}_s \circ \mathbf{w}_t \equiv [g; \mathbf{U}, \mathbf{V}, \mathbf{W}]$$

[1] Tamara G. Kolda and Brett W. Bader, Tensor Decompositions and Applications, SIAM REVIEW, vol. 51, n. 3, pp. 455-500, 2009.

[2] Y. Xu, W. Yin, A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion, SIAM J. Imaging Sci. 6 (3) (2013) 1758–1789.




```
# generate a random tensor with shape 20x20x20
X = synthetic_data_tucker([20, 20, 20], [10, 10, 10])
data_provider = Provider()
data_provider.full_tensor = lambda: X

# HOSVD
from tensorD.factorization.tucker import HOSVD
env = Environment(data_provider, summary_path='/tmp/hosvd_' + '20')
hosvd = HOSVD(env)
args = HOSVD.HOSVD_Args(ranks=[10, 10, 10], validation_internal=1, tol=1.0e-4)
hosvd.build_model(args)
hosvd.train()
# obtain factor matrices from trained model
factor_matrices = hosvd.factors
# obtain core tensor from trained model
core_tensor = hosvd.core
```

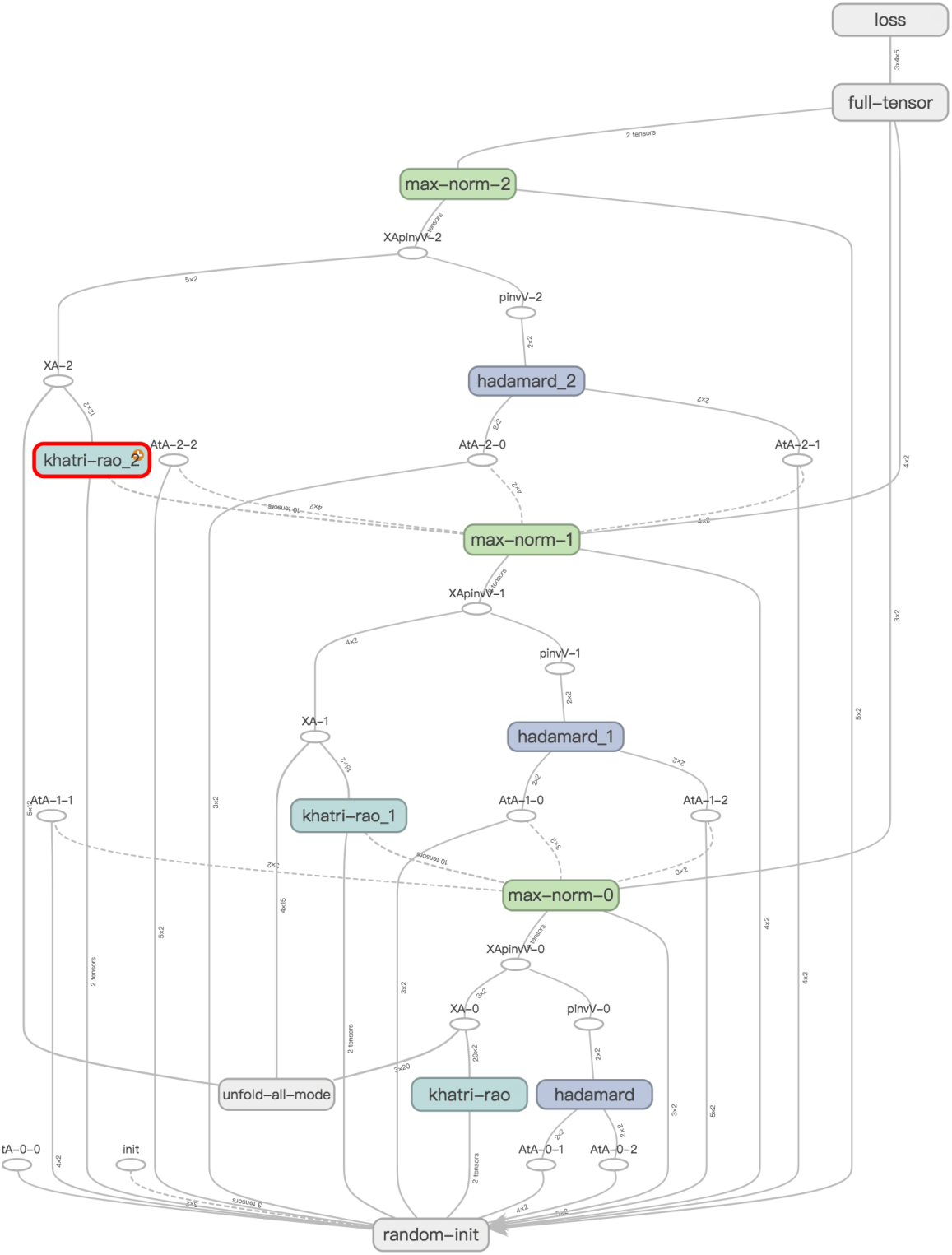
(continues on next page)

(continued from previous page)

```
# HOOI
from tensorD.factorization.tucker import HOOI
env = Environment(data_provider, summary_path='/tmp/hooi_' + '20')
hooi = HOOI(env)
args = HOOI.HOOI_Args(ranks=[10, 10, 10], validation_internal=1, tol=1.0e-4)
# build HOOI model
hooi.build_model(args)
# train HOOI model with the maximum iteration of 100
hooi.train(100)
# obtain factor matrices from trained model
factor_matrices = hooi.factors
# obtain core tensor from trained model
core_tensor = hooi.core
```

2.3.2 The CANDECOMP/PARAFAC (CP) decomposition

$$\mathcal{X} = \sum_r \lambda_r \mathbf{u}_r \circ \mathbf{v}_r \circ \mathbf{w}_r \equiv [\lambda; \mathbf{U}, \mathbf{V}, \mathbf{W}]$$



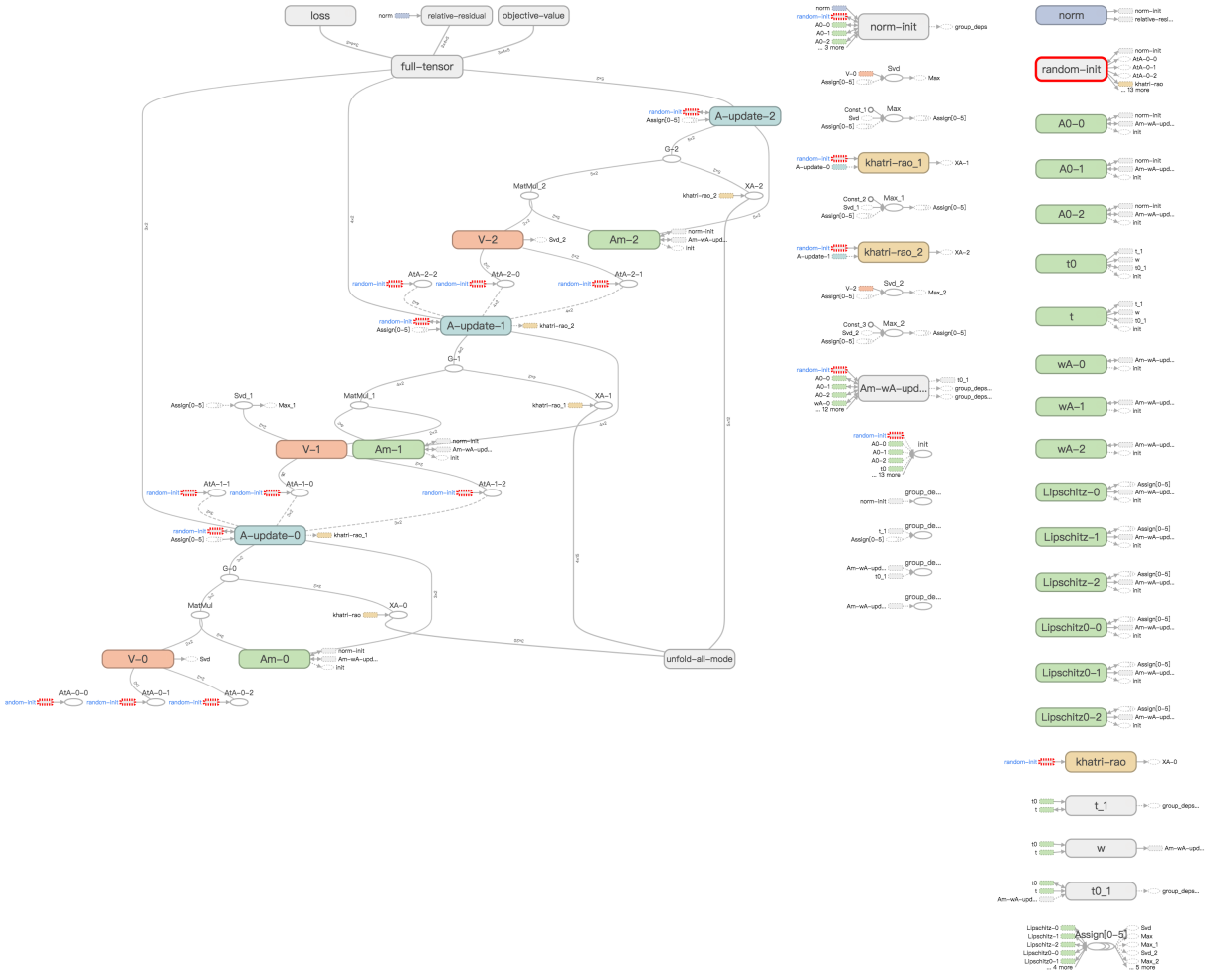

```
from tensorD.factorization.cp import CP_ALS
# generate a random tensor with shape 20x20x20
```

(continues on next page)

(continued from previous page)

```
X = synthetic_data_cp([20, 20, 20], 10)
data_provider = Provider()
data_provider.full_tensor = lambda: X
env = Environment(data_provider, summary_path='/tmp/cp_' + '20')
cp = CP_ALS(env)
args = CP_ALS.CP_Args(rank=10, validation_internal=1)
# build CP model with arguments
cp.build_model(args)
# train CP model with the maximum iteration of 100
cp.train(100)
# obtain factor matrices from trained model
factor_matrices = cp.factors
# obtain scaling vector from trained model
lambdas = cp.lambdas
```

2.3.3 The non-negative CANDECOMP/PARAFAC (NCP) decomposition



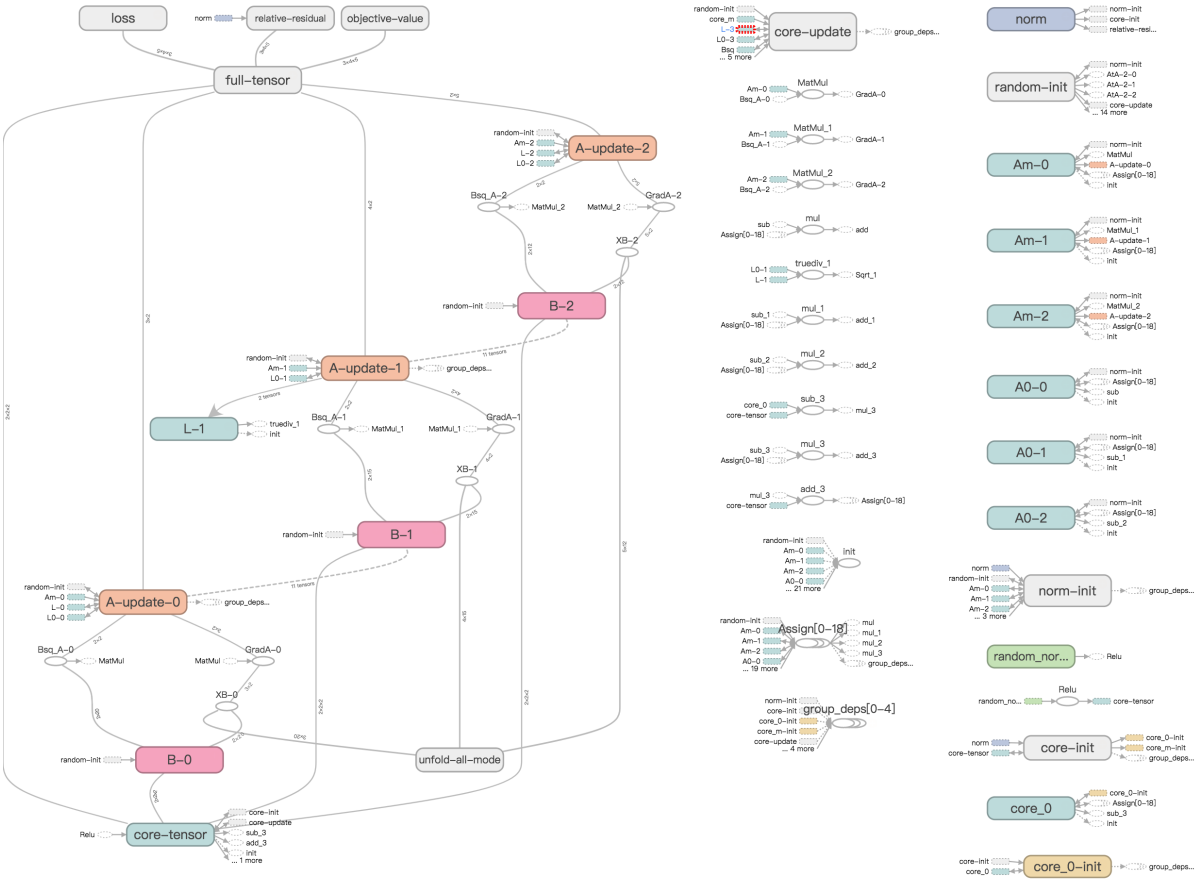

```
from tensorD.factorization.ncp import NCP_BCU
# generate a random tensor with shape 20x20x20
```

(continues on next page)

(continued from previous page)

```
X = synthetic_data_cp([20, 20, 20], 10)
data_provider = Provider()
data_provider.full_tensor = lambda: X
env = Environment(data_provider, summary_path='/tmp/ncp_' + '20')
ncp = NCP_BCU(env)
args = NCP_BCU.NCP_Args(rank=10, validation_internal=1)
# build NCP model
ncp.build_model(args)
# train NCP model with the maximum iteration of 100
ncp.train(100)
# obtain factor matrices from trained model
factor_matrices = ncp.factors
# obtain scaling vector from trained model
lambdas = ncp.lambdas
```

2.3.4 The non-negative Tucker (NTucker) decomposition


loss
norm
relative_residual
objective_value
full_tensor
A_update_2
B_2
A_update_1
L_1
B_1
A_update_0
B_0
unfold_all_mode
core_tensor
core_update
norm
random_init
Am_0
Am_1
Am_2
A0_0
A0_1
A0_2
L_0
L_2
L_3
L0_0
L0_1
L0_2
L0_3
Bsq
Grad_core
t0
t
wA_0
wA_1
wA_2
wA_3
core_0
core_m
core_0_init
core_m_init
norm_init
random_nor...
core_init
Assign[0_18]
group_deps[0_4]
group_deps...

```
from tensorD.factorization.ntucker import NTUCKER_BCU
# generate a random tensor with shape 20x20x20
X = synthetic_data_tucker([20, 20, 20], [10, 10, 10])
data_provider = Provider()
data_provider.full_tensor = lambda: X
env = Environment(data_provider, summary_path='/tmp/ntucker_demo_' + '30')
ntucker = NTUCKER_BCU(env)
args = NTUCKER_BCU.NTUCKER_Args(ranks=[10, 10, 10], validation_internal=1)
# build NTucker model
ntucker.build_model(args)
# train NCP model with the maximum iteration of 500
ntucker.train(500)
# obtain factor matrices from trained model
factor_matrices = ntucker.factors
# obtain core tensor from trained model
core_tensor = ntucker.core
```

2.3.5 The example: Pairwise Interaction Tensor Decomposition

Formally, pairwise interaction tensor assumes that each entry T_{ijk} of a tensor $\mathcal{T}$ of size $n_1 \times n_2 \times n_3$ is given by following:

$$T_{ijk} = \left\langle \mathbf{u}_i^{(a)}, \mathbf{v}_j^{(a)} \right\rangle + \left\langle \mathbf{u}_j^{(b)}, \mathbf{v}_k^{(b)} \right\rangle + \left\langle \mathbf{u}_k^{(c)}, \mathbf{v}_i^{(c)} \right\rangle, \text{for all } (i,j,k) \in [n_1] \times [n_2] \times [n_3]$$

The pairwise vectors in this formula are r_1, r_2, r_3 dimensions:

$$\left\{\mathbf{u}_i^{\{a\}}\right\}_{i\in[n_1]}, \left\{\mathbf{v}_j^{\{a\}}\right\}_{j\in[n_2]}$$
$$\left\{\mathbf{u}_j^{\{b\}}\right\}_{j\in[n_2]}, \left\{\mathbf{v}_k^{\{b\}}\right\}_{k\in[n_3]}$$
$$\left\{\mathbf{u}_k^{\{c\}}\right\}_{k\in[n_3]}, \left\{\mathbf{v}_i^{\{c\}}\right\}_{i\in[n_1]}$$

```
from tensorD.factorization.pitf_numpy import PITF_np
X = synthetic_data_tucker([20, 20, 20], [10, 10, 10])
data_provider = Provider()
data_provider.full_tensor = lambda: X
pitf_np_env = Environment(data_provider, summary_path='/tmp/pitf')
pitf_np = PITF_np(pitf_np_env)
sess_t = pitf_np_env.sess
init_op = tf.global_variables_initializer()
sess_t.run(init_op)
tensor = pitf_np_env.full_data().eval(session=sess_t)
args = PITF_np.PITF_np_Args(rank=5, delt=0.8, tao=12, sample_num=100, validation_
↪internal=1, verbose=False, steps=500)
y, X_t, Y_t, Z_t, Ef_t, If_t, Rf_t = pitf_np.exact_recovery(args, tensor)
y = tf.convert_to_tensor(y)
X = tf.convert_to_tensor(X_t)
Y = tf.convert_to_tensor(Y_t)
Z = tf.convert_to_tensor(Z_t)
Ef = tf.convert_to_tensor(Ef_t)
If = tf.convert_to_tensor(If_t)
Rf = tf.convert_to_tensor(Rf_t)
```

Specific details can refer to the paper "Exact and Stable Recovery of Pairwise Interaction Tensors, NIPS 2013"[3] .

[3] Chen, S., Lyu, M. R., King, I., & Xu, Z. (2013). Exact and stable recovery of pairwise interaction tensors. In Advances in Neural Information

2.3.6 Multiple decompositions

To perform multiple decompositions or one decomposition algorithm on several different tensors, we can use `tf.Graph()` to build several graphs and perform decompositions on different graphs. Take performing HOOI decomposition on 3 tensors as example:

```
for i in range(3):
    g1 = tf.Graph()
    data_provider = Provider()
    X = np.arange(60).reshape(3, 4, 5) + i
    data_provider.full_tensor = lambda: X
    hooi_env = Environment(data_provider, summary_path='/tmp/tensord')
    hooi = HOOI(hooi_env)
    args = hooi.HOOI_Args(ranks=[2, 2, 2], validation_internal=5)
    with g1.as_default() as g:
        hooi.build_model(args)
        hooi.train(100)
    print(np.sum(hooi.full - X))
    tf.reset_default_graph()
```

2.3.7 Tips

The test files include in the project.

The images shown above can clearly see the decomposition process and relationship between each step in decomposition algorithm.

2.3.8 References

Processing Systems (pp. 1691-1699).

CHAPTER 3

tensorD

3.1 tensorD package

3.1.1 Subpackages

tensorD.base package

Subpackages

tensorD.base.test package

Submodules

tensorD.base.test.dtensor_test module

tensorD.base.test.ktensor_test module

tensorD.base.test.ops_test module

tensorD.base.test.ttensor_test module

tensorD.base.test.pitf_ops_test module

Module contents

Submodules

3.1.2 Submodules

3.1.3 tensorD.loss module

3.1.4 tensorD.test_bench module

3.1.5 Module contents