Tensorlab Demos

Release 3.0

Otto Debals

Frederik Van Eeghem

Nico Vervliet

Lieven De Lathauwer

This is a demo-based introduction to tensor computations using Tensorlab. The demonstrations range from basic tensor operations, over multidimensional harmonic retrieval and ICA, to structured data fusion for machine learning using a GPS dataset.

March 28, 2016

CONTENTS

This collection of demos has as goal to introduce Tensorlab to users as a tool to solve tensor problems. We begin with a demo about the *basic use of Tensorlab* followed by a demo about the *multilinear singular value decomposition and the low multilinear rank approximation*. In the other demos, the usage of Tensorlab is illustrated for several applications. The list is far from complete, but should give readers from various domains examples of how the software can be used. The second demo discusses the *multidimensional harmonic retrieval* problem. The demo *Using basic Tensorlab features for ICA* uses basic tensor tools to solve the independent component analysis (ICA) problem. ICA is also discussed in the *Using advanced Tensorlab features for ICA* demo, which applies the structured data fusion (SDF) framework from Tensorlab to implement constrained decompositions. Furthermore, SDF is also illustrated for *independent vector analysis* (IVA), and for *user involvement prediction* based on a GPS dataset.

Matlab-scripts for the demos can be downloaded here[1], or from their respective pages. A PDF version of the demos can be found here[2].

[1] http://www.tensorlab.net/demos/allcode.zip
[2] http://www.tensorlab.net/demos/demos.pdf

CHAPTER

ONE

BASIC USE OF TENSORLAB

Tags: `tensor construction`, `tensor visualization`, `cpd`, `lmlra`, `mlsvd`, `initialization`, `compression`

Tensorlab is a Matlab package for complex optimization and tensor computations. This demo will discuss the basics of Tensorlab. It consists of three consecutive parts. A first section *Tensor construction and visualization* will explain how a tensor can be defined and visualized. The second section *CPD for beginners* handles the elementary use of the `cpd` command for the computation of the canonical polyadic decomposition, while the third section *CPD for pros* discusses more advanced items such as customized initialization methods and algorithms.

A Matlab implementation of this demo is given in the `demo_basic.m` file, which can be found in the demo folder of Tensorlab or downloaded here[1].

1.1 Tensor construction and visualization

Construction of a tensor with random entries

A random tensor can be constructed by using the basic `randn` and `rand` commands in Matlab. For example, we can construct a tensor $\mathcal{T} \in \mathbb{R}^{10\times 20\times 30}$ with entries randomly sampled from a Gaussian distribution as follows:

```
size_tens = [10 20 30];
T = randn(size_tens);
```

Construction of a tensor as a sum of R rank-1 terms

To construct a third-order tensor $\mathcal{T} \in \mathbb{R}^{I\times J\times K}$ which can be written as a sum of R rank-1 terms, we start with the construction of the three factor matrices $\mathbf{U}^{(1)} \in \mathbb{R}^{I\times R}$, $\mathbf{U}^{(2)} \in \mathbb{R}^{J\times R}$ and $\mathbf{U}^{(3)} \in \mathbb{R}^{K\times R}$, corresponding to the three different modes (see Canonical polyadic decomposition). This can be done with the command `cpd_rnd`, returning a cell which collects these matrices:

```
size_tens = [10 20 30];
R = 4;
U = cpd_rnd(size_tens,R); % U is a cell containing the factor matrices
```

The tensor can then be obtained from the factor matrices using the command `cpdgen`:

```
T = cpdgen(U);
```

For a graphical representation, we refer to Fig. 1.1.

Note that it is possible to construct complex factor matrices and tensors too, through the options field of the `cpd_rnd` command:

[1] http://www.tensorlab.net/demos/demo_basic.zip

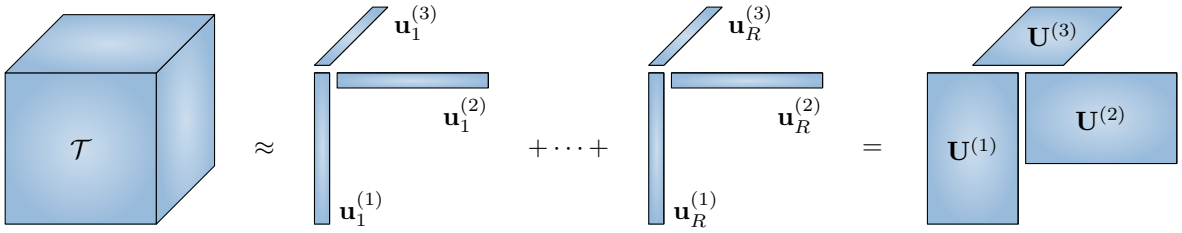

Fig. 1.1: Visualization of a polyadic decomposition.

```
size_tens = [10 20 30];
R = 4;
options = struct;
options.Real = @randn;
options.Imag = @randn;
U = cpd_rnd(size_tens,R,options);
T = cpdgen(U);
```

Alternatively, instead of constructing a structure array, the options can be passed as follows:

```
U = cpd_rnd(size,tens,R,'Real',@randn,'Imag',@randn);
```

We will use this method in this demo when the number of options to be passed is small. Note that the option names are case insensitive.

Construction of a tensor obtained by a multilinear transformation of a core tensor

We start with the construction of the core tensor and the transformation matrices using the `lmlra_rnd` command. The tensor can be obtained through the tensor-matrix product of the core tensor with the different matrices, for which one can use the `lmlragen` command.

```
size_tens = [10 20 30];
size_core = [4 5 6];
[F,S] = lmlra_rnd(size_tens,size_core);
% F is a cell collecting the transformation matrices U, V and W
Tlmlra = lmlragen(F,S);
```

For a graphical representation of the multilinear product, we refer to Fig. 1.2.

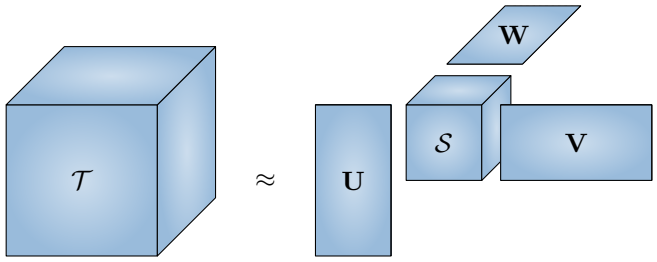

Fig. 1.2: Visualization of the multilinear product of a core tensor $\mathcal{S}$ with factor matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$.

Perturbation of a tensor by noise

A tensor can be perturbed by Gaussian noise with a user-defined signal-to-noise-ratio using the command `noisy`:

```
SNR = 20;
Tnoisy = noisy(T,SNR);
```

Visualization

Tensorlab offers four commands for tensor visualization: `slice3`, `surf3`, `voxel3` and `visualize`. The plots of the first two commands contain sliders to go through the different slices. The `voxel3` command plots the elements of a tensor as voxels whose color and opacity are proportial to the value of the elements. The plot has two sliders for the threshold parameter and the degree parameter. Figures 1.3, 1.4 and 1.5 visualize the tensor constructed in the following block of code:

```
t = linspace(0,1,60).';
T = cpdgen({sin(2*pi*t+pi/4),sin(4*pi*t+pi/4),sin(6*pi*t+pi/4)});
figure(); slice3(T);
figure(); surf3(T);
figure(); voxel3(T);
```

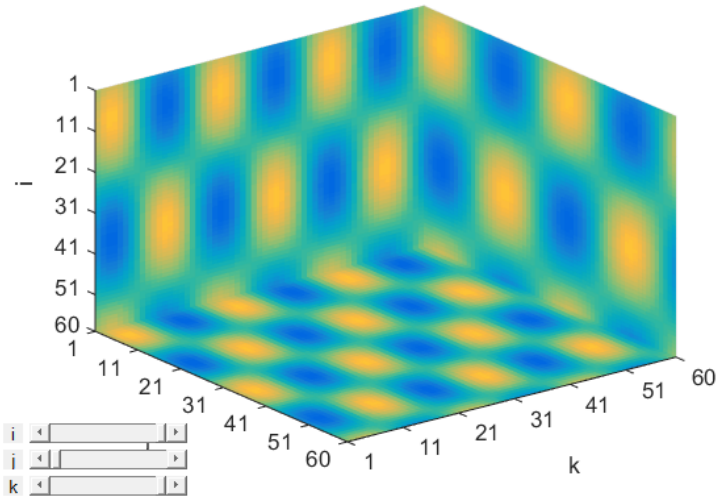

Fig. 1.3: Visualization of a tensor using `slice3`.

Finally, the visualize method can be used to 'walk through' the data and to compare tensor a decomposed tensor with its decomposition. For example:

```
t = linspace(0,1,60).';
U = {sin(2*pi*t+pi/4),sin(4*pi*t+pi/4),sin(6*pi*t+pi/4)};
T = noisy(cpdgen(U), 30);
visualize(U, 'original', T);
```

The result can be seen in Fig. 1.6. The check boxes can be used to select the dimensions to plot. The sliders or the text fields allow a user to select which slices are plotted. More examples involving the `visualize` method can be found in the user guide.

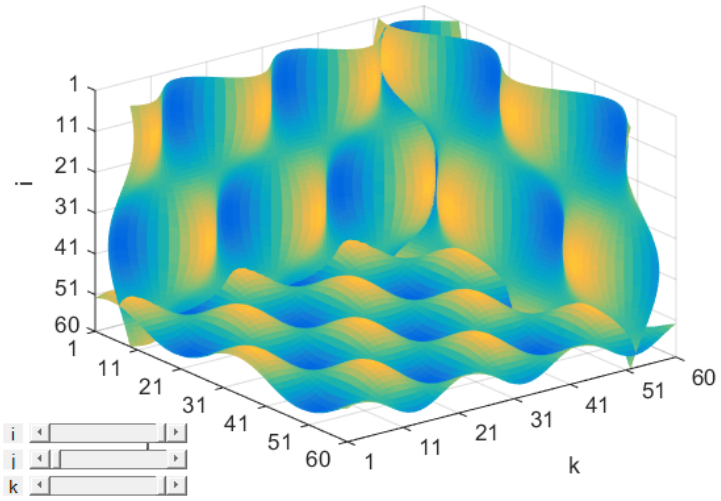

Fig. 1.4: Visualization of a tensor using `surf3`.

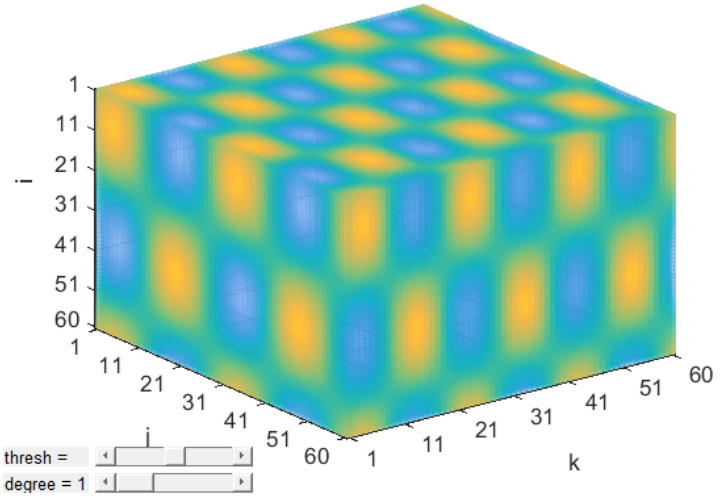

Fig. 1.5: Visualization of a tensor using `voxel3`.

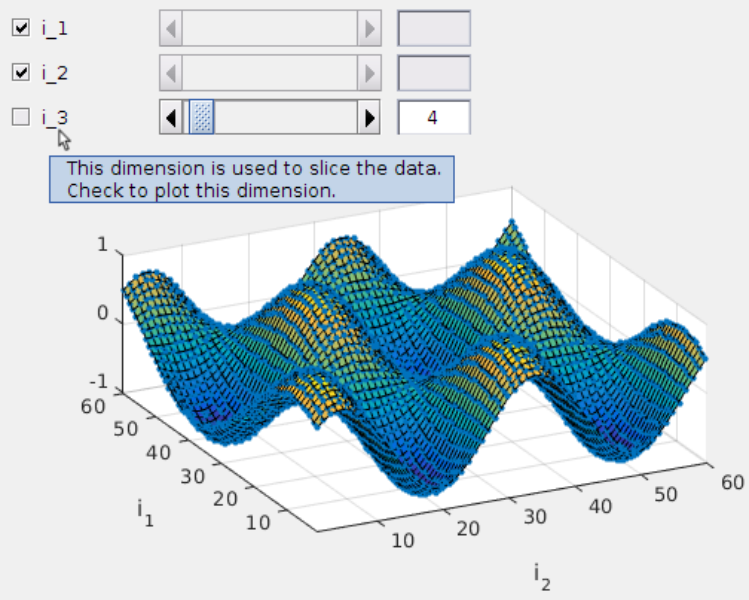

Fig. 1.6: Visualization of a tensor using `visualize`.

1.2 CPD for beginners

Let us consider a tensor $\mathcal{T}$ that approximately has low rank, constructed for example using one of the techniques from the previous section. We discuss the estimation of the rank, the use of the basic `cpd` command and the calculation of two different error measures.

Rank estimation

If the rank of a tensor is not known or cannot be obtained from prior knowledge, the command `rankest` from Tensorlab may be useful. It estimates the corner of the L-curve obtained by sweeping over a number of rank-1 terms. This corner is taken as an estimate of the rank of the tensor. If there is no output argument, an L-curve as in Fig. 1.7 is plotted.

```
figure();
U = cpd_rnd([10 20 30],4);
T = cpdgen(U);
rankest(T);
```

For noisy tensors, it can be worthwhile to edit the relative error tolerance `MinRelErr` in the `options` structure of the `rankest` command.

The basic CPD command

The basic `cpd` algorithm requires only a tensor and a value for the rank:

```
R = 4;
Uest = cpd(T,R)
```

An extra `output` argument can be provided with `[U,output] = cpd(T,R)` to obtain information about the convergence of the algorithm, among other things.

Error measures

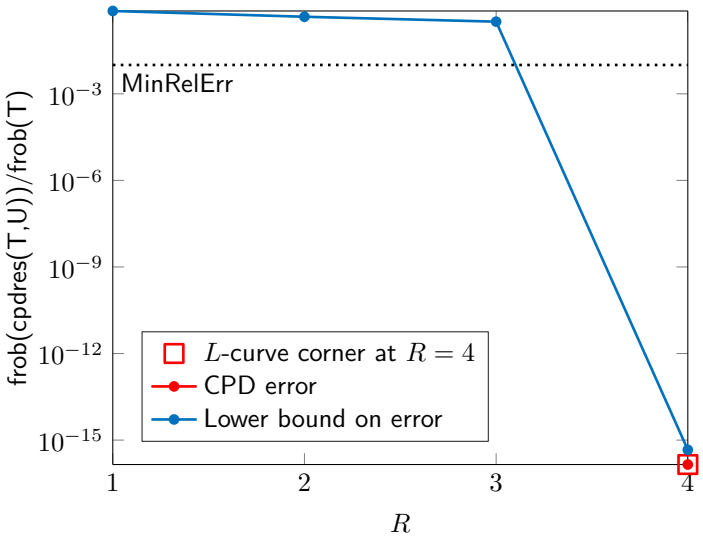

Fig. 1.7: The plot generated by the `rankest` command.

The relative error on the estimates of the factor matrices can be calculated with the command `cpderr`. It takes into account the permutation and scaling indeterminacies. The command returns the relative errors (for each factor matrix), the permutation matrix, the scaling matrices and the permuted and scaled factor matrices:

```
[relerrU,P,D,Uest] = cpderr(U,Uest)
% D is a cell collecting the different scaling matrices
```

The relative error on the estimate of the tensor can be determined using `frob` and `cpdres`, with the latter calculating the residuals:

```
relerrT = frob(cpdres(T,Uest))/frob(T)
```

1.3 CPD for pros

In general, a `cpd` computation consists of a compression step, a step in which the CPD of the compressed tensor is computed, and a step in which the estimates are refined using the uncompressed tensor. The user has control over what happens in each step. In this section, we discuss the different steps in more detail.

Note that an overview of the output of each step can be obtained if the `Display` option is set to true:

```
Uest = cpd(T,R,'Display',true);
```

Compression

In the compression step, the tensor is compressed to a smaller tensor using the `mlsvd_rsi` command by default, which applies a randomized SVD algorithm in combination with randomized subspace iterations. Compression is only performed if it is expected to lead to a significant reduction in computational complexity. The user can alter the compression method with the `Compression` option by providing a compression technique:

```
Uest = cpd(T,R,'Compression',@lmlra_aca);
```

In this example, a low multilinear rank approximation with adaptive cross-approximation is used. The user can also disable the compression step (note that there will not be a refinement step then):

```
Uest = cpd(T,R,'Compression',false);
```

Initialization

To initialize the computation of the CPD of the compressed tensor, `cpd` uses a method based on the generalized eigenvalue decomposition (implemented in `cpd_gevd`) if the rank does not exceed the second largest dimension of the tensor. Otherwise, a random initialization is used. The initialization method can be altered with the `Initialization` option:

```
Uest = cpd(T,R,'Initialization',@cpd_rnd);
```

In this example, a random initialization is chosen. The user can also provide particular factor matrices `Uinit` for the initialization:

```
Uest = cpd(T,Uinit);
```

When specific factor matrices are provided, the compression step is skipped.

Core algorithms

Tensorlab provides several core algorithms for the computation of a CPD. First, there are optimization-based methods such as `cpd_als` (alternating least squares), `cpd_minf` (unconstrained nonlinear optimization) and `cpd_nls` (nonlinear least squares). Second, there are (semi-)algebraic methods such as `cpd3_sd` and `cpd3_sgsd`. The former reduces the computation to the simultaneous diagonalization of symmetric matrices, also in cases where only one factor matrix has full column rank, while the latter reduces the computation to the estimation of orthogonal factors in a simultaneous generalized Schur decomposition.

By default, `cpd_nls` is used for the CPD of the compressed tensor and for the refinement. The user can alter the choice of algorithm with the `Algorithm` and `Refinement` options. By default, `Refinement` is equal to `Algorithm`. The following example uses an alternating least squares approach for the CPD of the compressed tensor and for the refinement:

```
Uest = cpd(T,R,'Algorithm',@cpd_als);
```

Core algorithm parameters

Parameters can be passed to the core algorithm with the `AlgorithmOptions` option for `cpd`. Tensorlab provides default choices for all parameters.

A first parameter is `Display`, which determines after how many iterations intermediate results are printed. The first column of the printed results shows the objective function values. The second and third column give the relative change in the objective function values and the step sizes, respectively.

There are three parameters that affect the stopping criterion. The parameter `Maxiter` determines the maximum number of iterations. The parameters `TolX` and `TolFun` are tolerances for the step size and for the relative change in objective function value, respectively. For the NLS algorithm, the `CGMaxIter` option determines the maximum number of conjugate gradient iterations in each NLS iteration. The following piece of code illustrates how these parameters can be changed for the `cpd_nls` and `cpd_als` algorithms:

```
R = 4;
T = cpdgen(cpd_rnd([10 20 30],R));
Uinit = cpd_rnd([10 20 30],R);

options = struct;
options.Compression = false;
options.Algorithm = @cpd_nls;
```

```
options.AlgorithmOptions.Display = 1;
options.AlgorithmOptions.MaxIter = 100;      % Default 200
options.AlgorithmOptions.TolFun = eps^2;     % Default 1e-12
options.AlgorithmOptions.TolX = eps;         % Default 1e-6
options.AlgorithmOptions.CGMaxIter = 20;     % Default 15
[Uest_nls,output_nls] = cpd(T,Uinit,options);

options = struct;
options.Compression = false;
options.Algorithm = @cpd_als;
options.AlgorithmOptions.Display = 1;
options.AlgorithmOptions.MaxIter = 100;      % Default 200
options.AlgorithmOptions.TolFun = eps^2;     % Default 1e-12
options.AlgorithmOptions.TolX = eps;         % Default 1e-6
[Uest_als,output_als] = cpd(T,Uinit,options);
```

A convergence plot can be obtained from the link in the command terminal output, or by plotting the objective function values from `output.Algorithm` (and `output.Refinement`):

```
figure();
semilogy(output_nls.Algorithm.fval); hold all;
semilogy(output_als.Algorithm.fval);
ylabel('Objective function'); xlabel('Iteration');
title('Convergence plot'); legend('cpd\_nls','cpd\_als')
```

Fig. 1.8 shows the output of the code block above and illustrates first-order convergence for the ALS algorithm and second-order convergence for the NLS algorithm.

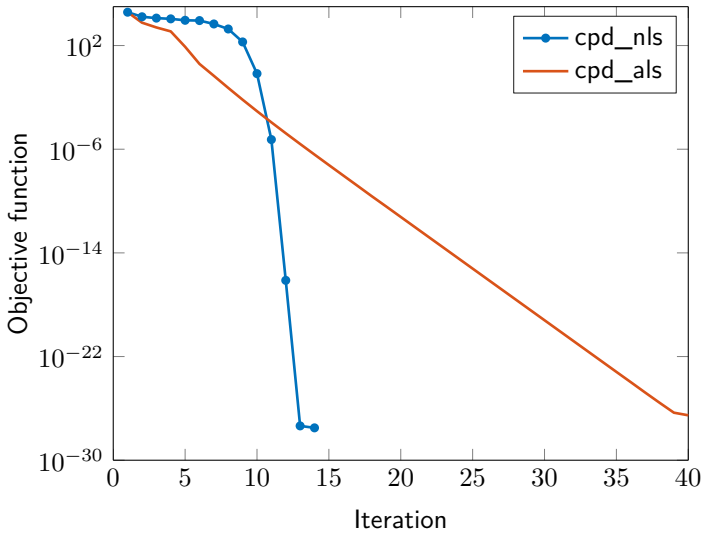

Fig. 1.8: Convergence plot for the `cpd_nls` and `cpd_als` algorithm.

1.4 Multilinear singular value decomposition and low multilinear rank approximation

Let us consider a tensor $\mathcal{T} \in \mathbb{R}^{10\times 20\times 30}$. The multilinear singular value decomposition (MLSVD) of $\mathcal{T}$ is a decomposition of the form

$$\mathcal{T} = \mathcal{S} \cdot_1 \mathbf{U}^{(1)} \cdot_2 \mathbf{U}^{(2)} \cdot_3 \mathbf{U}^{(3)},$$

with $\mathcal{S} \in \mathbb{R}^{10\times 20\times 30}$ an all-orthogonal and ordered tensor. The factor matrices $\mathbf{U}^{(1)} \in \mathbb{R}^{10\times 10}$, $\mathbf{U}^{(2)} \in \mathbb{R}^{20\times 20}$ and $\mathbf{U}^{(3)} \in \mathbb{R}^{30\times 30}$ have orthonormal columns. This decomposition can be calculated using the `mlsvd` command. In the following example, a tensor with random entries is decomposed:

```
size_tens = [10 20 30];
T = randn(size_tens);
[U,S,sv] = mlsvd(T);
```

The mode-n singular values are stored in `sv{n}`. A truncated MLSVD can be obtained by providing the desired core size as second input argument:

```
size_core = [4 5 6];
[U,S,sv] = mlsvd(T,size_core);
```

Note that, due to the truncation, the core tensor may no longer be all-orthogonal and ordered. The truncated MLSVD is just one, not necessarily optimal, way of obtaining a low multilinear rank approximation (LMLRA). A better approximation with the same multilinear rank can be obtained with the `lmlra` command:

```
[U,S] = lmlra(T,size_core);
```

Similar to the `cpd` command as explained above, the initialization and core algorithm can be altered using the `Initialization` and `Algorithm` options. Various LMLRA core algorithms are available; we refer to the user guide and the documentation of the `lmlra` command for more information. Factor matrices and a core tensor to initialize the algorithm can be provided with `lmlra(T,U0,S0)`. By default, `lmlra` provides orthogonal factor matrices and an all-orthogonal and ordered core tensor. This step can be skipped by setting the `Normalize` option to false.

For a more in-depth discussion of MLSVD and LMLRA, we refer to the demo *Multilinear singular value decomposition and low multilinear rank approximation*.

CHAPTER

TWO

MULTILINEAR SINGULAR VALUE DECOMPOSITION AND LOW MULTILINEAR RANK APPROXIMATION

Tags: `mlsvd`, `lmlra`

The multilinear singular value decomposition is, as the term indicates, a multilinear generalization of the matrix singular value decomposition. In this demo, we provide some insight in this decomposition and the low multilinear rank approximation. A Matlab implementation of this demo is given in the `demo_mlsvd.m` file, which can be found in the demo folder of Tensorlab or downloaded here[1].

For matrices, the singular value decomposition $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}$ is well known. Using tensor-matrix products, this decomposition can be written as

$$\mathbf{M} = \mathbf{\Sigma} \cdot_1 \mathbf{U} \cdot_2 \mathbf{V}.$$

The matrix $\mathbf{\Sigma}$ is a diagonal matrix, and the matrices $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices. A generalization of this SVD is the multilinear singular value decomposition (MLSVD). In the literature, one can also find the names higher-order SVD (HOSVD) and Tucker decomposition. The term Tucker decomposition has evolved over the years and is now often used in a more general sense. The MLSVD of a third-order tensor can be written as

$$\mathcal{T} = \mathcal{S} \cdot_1 \mathbf{U}^{(1)} \cdot_2 \mathbf{U}^{(2)} \cdot_3 \mathbf{U}^{(3)}.$$

Like in the matrix case where $\mathbf{U}$ and $\mathbf{V}$ are orthonormal bases for the column and row space, respectively, the MLSVD computes N orthonormal bases $\mathbf{U}^{(n)}$, $n = 1, \ldots, N$ for the N different subspaces of mode-n vectors. As is illustrated in Fig. 2.1, these orthonormal bases have exactly the same interpretation in as the matrix case. Essentially, the MLSVD considers a given tensor as N sets of mode-n vectors, and computes the matrix SVD of these sets. The multilinear rank of the tensor $\mathcal{T}$ corresponds to the $N-$tuplet consisting of the dimensions of the different subspaces.

2.1 The noiseless case

First we study the properties of the MLSVD for the exact case. Random factor matrices and a random core tensor are used to construct a multilinear rank-$(2, 2, 3)$ tensor $\mathcal{T}$. Next the MLSVD is computed using `mlsvd`.

```
S = randn(2,2,3);
U = {rand(4,2), rand(5,2), rand(6,3)};
T = lmlragen(U,S);
[Ue,Se,sve] = mlsvd(T)
```

We can check the properties of the MLSVD:

[1] http://www.tensorlab.net/demos/demo_mlsvd.zip

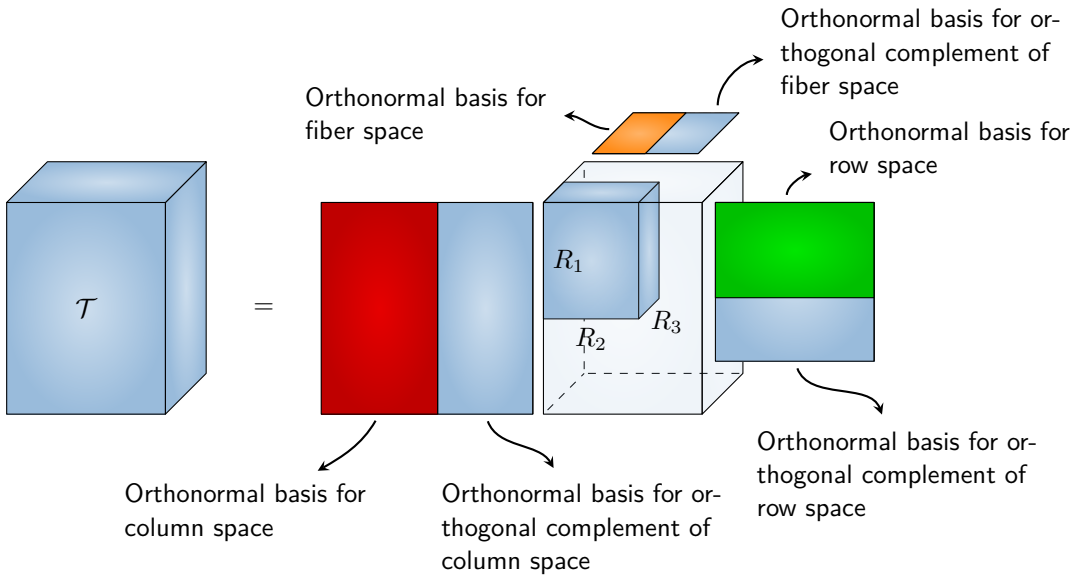

Fig. 2.1: Illustration of the multilinear singular value decomposition of a multilinear rank-(R_1, R_2, R_3) tensor and the different spaces.

- The first two, two and three columns of respectively `Ue{1}`, `Ue{2}`, and `Ue{3}` are a basis for the column, row and fiber subspaces of $\mathcal{T}$, respectively. (`subspace` computes the angle between subspaces: a zero angle indicates that the spaces are identical.)

```
subspace(Ue{1}(:,1:2),U{1})
subspace(Ue{2}(:,1:2),U{2})
subspace(Ue{3}(:,1:3),U{3})
```

```
 4.3356e-16
 4.9257e-16
 6.2976e-16
```

- The remaining two, three and three columns of `Ue{1}`, `Ue{2}`, and `Ue{3}` are orthogonal bases for the complements of the column, row and fiber subspaces of $\mathcal{T}$. These subspaces are orthogonal to the column, row, and fiber subspaces, respectively:

```
U{1}'*Ue{1}(:, 3:end)
U{2}'*Ue{2}(:, 3:end)
U{3}'*Ue{3}(:, 4:end)
```

```
ans =
   1.0e-15 *
    0.2220    0.0555
    0.0694    0.4580
ans =
   1.0e-15 *
   -0.0139    0.0230    0.0173
    0.1110    0.0607    0.1110
ans =
   1.0e-15 *
```

```
    -0.1110    0.2637         0
    -0.0971    0.2463    0.0902
     0.0555   -0.2220   -0.1527
```

- Similar to the matrix SVD, the multilinear singular values `sve` indicate the importance of a basis vector in a particular mode. In contrast to the SVD, the core tensor $\mathcal{S}$ is not diagonal. The multilinear singular values are defined as the Frobenius norms of the slices of order $N-1$ for the different modes, i.e. the norm of the subtensor created by fixing one index:

```
nrm = zeros(size(Se,1),1);
for i = 1:size(Se,1)
    nrm(i) = frob(Se(i,:,:));
end
[sve{1} nrm] % mode-1 singular values
```

```
 ans =
    2.0722    2.0722
    1.1272    1.1272
    0.0000    0.0000
    0.0000    0.0000
```

The multilinear rank of a tensor can be determined by looking at the multilinear singular values. As shown in Fig. 2.2, the multilinear singular values $\sigma_r^{(n)}$ become zero for $r > R_n$, in which R_n is the mode-n rank.

```
for n = 1:3
    subplot(1,3,n);
    semilogy(sve{n},'.-');
    xlim([1 length(sve{n})])
end
```

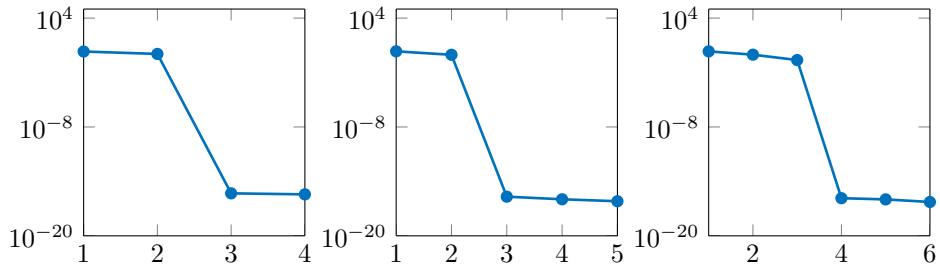

Fig. 2.2: Multilinear singular values in the noiseless case.

- Finally, the all-orthogonality of the core tensor is investigated. This property means that all slices of order $N-1$ are mutually orthogonal to each other. We test this for the second mode by computing the inner product between the slices. To do this efficiently, we unfold the tensor in the second mode.

```
M = tens2mat(Se, 2);
M*M'
```

```
  4.7448         0    0.0000   -0.0000   -0.0000
       0    0.8197    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    0.0000    0.0000    0.0000    0.0000
```

We indeed see that all the off-diagonal entries are zero. Note that the diagonal contains the squared

multilinear singular values for the second mode.

```
[sve{2} sve{2}.^2]
```

```
 2.1782    4.7448
 0.9054    0.8197
 0.0000    0.0000
 0.0000    0.0000
 0.0000    0.0000
```

2.2 Noisy case

The equality in Equation holds only for the exact case. When noise is added, the multilinear rank-$(2,2,3)$ decomposition only approximates the tensor. Here, the influence of the noise is investigated. In this example, noise is added such that the signal-to-noise ratio is 30dB.

```
Tn = noisy(T, 30);
```

Again, a MLSVD can be computed from this noisy tensor. The multilinear singular values are plotted in Fig. 2.3. Estimating the multilinear rank is now a more difficult task.

```
[Uen,Sen,sven] = mlsvd(Tn);
for n = 1:3
    subplot(1,3,n);
    semilogy(sven{n},'.-');
    xlim([1 length(sve{n})])
end
```

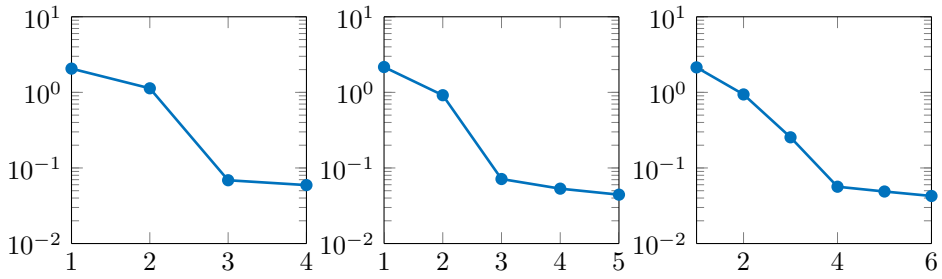

Fig. 2.3: Multilinear singular values in the noisy case.

From the construction, we know that the multilinear rank is $(2,2,3)$. We can only keep the respective vectors and part of the core tensor:

```
Uentrunc{1} = Uen{1}(:,1:2);
Uentrunc{2} = Uen{2}(:,1:2);
Uentrunc{3} = Uen{3}(:,1:3);
Sentrunc = Sen(1:2, 1:2, 1:3);
```

The approximation error will not be zero now. The relative error is:

```
format long
frob(lmlragen(Uentrunc,Sentrunc)-Tn)/frob(Tn)
```

```
  0.049896088936784
```

In contrast to the matrix case, this approximation may not be optimal in terms of relative Frobenius norm error, however. The `lmlra` method attempts to find the tensor of multilinear rank $(2, 2, 3)$ that best approximates $\mathcal{T}$ in a least squares sense.

```
[Uopt, Sopt] = lmlra(Tn, [2 2 3]);
```

We again compute the error:

```
frob(lmlragen(Uopt,Sopt)-Tn)/frob(Tn)
```

```
  0.049884982735855
```

The error has indeed improved. (For other tensors the difference may be more substantial.) The computational cost has increased, however. This cost can be somewhat reduced by taking the MLSVD solution as initialization:

```
[Uopt, Sopt] = lmlra(Tn, Uentrunc, Sentrunc);
```

The relative approximation error can also be improved by keeping more multilinear singular values, and hence by taking larger subspaces.

```
Uentrunc2{1} = Uen{1}(:,1:3);
Uentrunc2{2} = Uen{2}(:,1:3);
Uentrunc2{3} = Uen{3}(:,1:3);
Sentrunc2 = Sen(1:3, 1:3, 1:3);

frob(lmlragen(Uentrunc2,Sentrunc2)-Tn)/frob(Tn)
```

```
  0.042189469973458
```

While the error indeed decreases and the MLSVD is computationally cheaper than `lmlra`, taking larger subspaces may not be desirable in particular applications, e.g., in multidimensional harmonic retrieval.

CHAPTER

THREE

MULTIDIMENSIONAL HARMONIC RETRIEVAL

Tags: `cpd` , `missing elements`

The goal of this demo is to illustrate the use of the basic `cpd` command in an array processing application. We consider binary phase-shift keying (BPSK) signals impinging on a 10x10 uniform rectangular array (URA). The goal is to separate these signals and identify the directions of arrival. The harmonic structure due to the array geometry is not exploited in this demo; this can be done in the structured data fusion framework, as illustrated in other demos such as *Using advanced Tensorlab features for ICA*. This demo also illustrates the use of `cpd` in the case where values are missing due to sensor malfunctioning.

A Matlab implementation of this demo is given in `demo_mdhr.m`, which can be found in the demo folder of Tensorlab or downloaded here[1].

3.1 Data

Three sources impinge on a URA with azimuth angles of 10°, 30° and 70°, respectively, and with elevation angles of 20°, 30° and 40°, respectively. We observe 15 time samples, such that a tensor $\mathcal{T} \in \mathbb{C}^{10\times10\times15}$ is obtained with t_{ijk} the observed signal from antenna (i, j) sampled at time instance k. Each source contributes a rank-1 term to the tensor. The vectors in the first and second mode are Vandermonde and the third mode contains the respective source signals multiplied by attenuation factors. Hence, the factor matrices in the first and second mode, denoted as $\mathbf{A}$ and $\mathbf{E}$, are Vandermonde matrices:

```
N = 15; R = 3; I = 10;
azimuth_angles = [10 30 70];     % azimuth angles in degrees
elevation_angles = [20 30 40];   % elevation angles in degrees
lambda = 0.25;                   % wavelength of signal carriers
dx = 0.1;                        % sensor distance
for r = 1:R
  zar = exp(2*pi/lambda*sin(azimuth_angles(r)*pi/180)*dx*1i);
  for i = 1:I, A(i,r) = zar^(i-1); end
  zer = exp(2*pi/lambda*sin(elevation_angles(r)*pi/180)*dx*1i);
  for i = 1:I, E(i,r) = zer^(i-1); end
end
```

The factor matrix in the third mode is the matrix containing the attenuated sources, denoted by $\mathbf{S}$:

```
S = round(rand(N,R))*2-1;
attenuation_vector = [1 .5 .25];
for r = 1:R
```

[1] http://www.tensorlab.net/demos/demo_mdhr.zip

```
  S(:,r) = attenuation_vector(r)*S(:,r);
end
```

The source signals are shown in Fig. 3.1. The generated tensor is perturbed by Gaussian noise with a signal-to-noise-ratio of 0 dB:

```
T = cpdgen({A,E,S});
SNR = 0;
Tnoisy = noisy(T,SNR);
```

Fig. 3.2 shows two observed signals with and without noise.

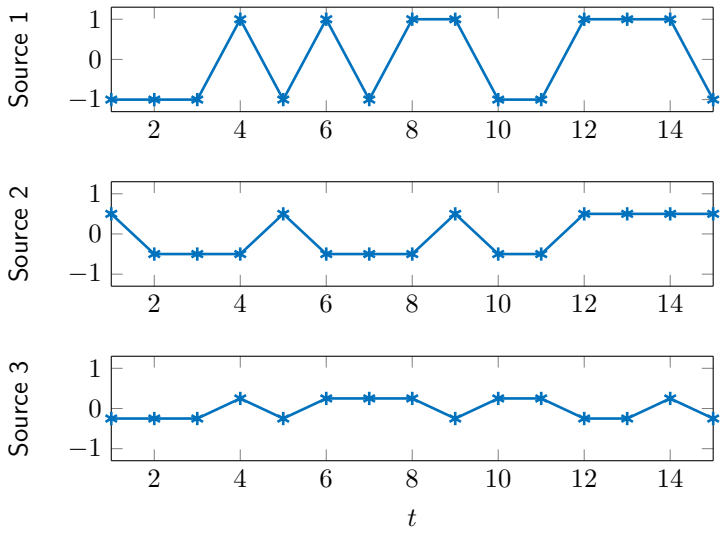

Fig. 3.1: The three BPSK sources.

3.2 Signal separation and direction-of-arrival estimation

The sources are separated by means of a basic CPD, without using the Vandermonde structure.

```
Uest = cpd(Tnoisy,R);
Aest = Uest{1};
Eest = Uest{2};
Sest = Uest{3};
```

The relative errors on the estimates of the factor matrices can be calculated with `cpderr`. They are 0.3127, 0.2540 and 0.1698, respectively. The `cpderr` command also returns estimates of the permutation matrix and scaling matrices, which can be used to fix the indeterminacies:

```
[err,P,D,Uestfix] = cpderr({A,E,S},Uest);
Aestfix = Uestfix{1}; errA = err(1)
Eestfix = Uestfix{2}; errE = err(2)
Sestfix = Uestfix{3}; errS = err(3)
```

The source signals and their recovered versions are compared in Fig. 3.3.

The direction-of-arrival angles can be determined using the shift invariance property of the individual Vandermonde vectors. This gives relative errors for the azimuth angles of 0.0390, 0.0362 and 0.1148, and for the elevation

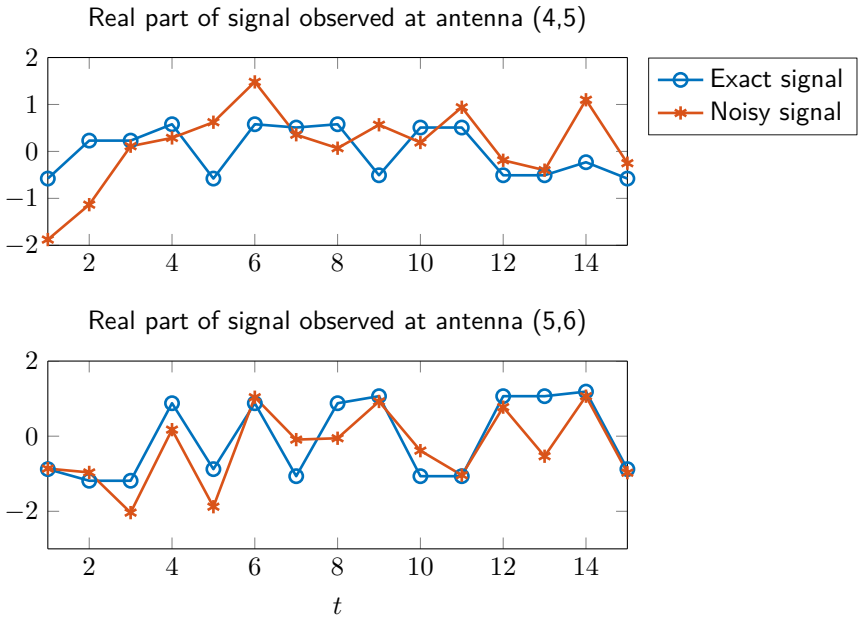

Fig. 3.2: Two observed signals with and without noise, for antennas (4,5) and (5,6).

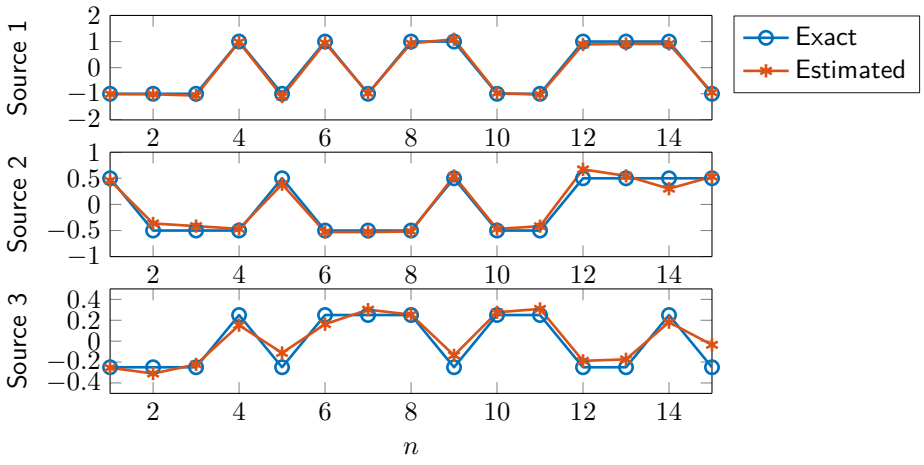

Fig. 3.3: The original and recovered source signals.

angles of 0.0073, 0.0067 and 0.0307:

```
for r = 1:R
  azimuth_z_est = Aestfix(1:end-1,i)\Aestfix(2:end,i);
  azimuth_angles_est(i) = asin(angle(azimuth_z_est)*lambda/(2*pi*dx))*(180/pi);
  elevation_z_est = Eestfix(1:end-1,i)\Eestfix(2:end,i);
  elevation_angles_est(i) = asin(angle(elevation_z_est)*lambda/(2*pi*dx))*(180/pi);
end

azimuth_err = abs(sort(azimuth_angles)-sort(azimuth_angles_est))./ ...
abs(sort(azimuth_angles))
elevation_err = abs(sort(elevation_angles)-sort(elevation_angles_est))./ ...
abs(sort(elevation_angles))
```

Note that one can additionally impose the Vandermonde structure in the structured data fusion framework, as illustrated in other demos such as the demo *Using advanced Tensorlab features for ICA*.

3.3 Missing values due to broken sensors

If one or more antennas are broken, the tensor $\mathcal{T}$ is incomplete. Tensorlab is able to process full, sparse and incomplete tensors. The missing entries can be indicated by NaN values. We consider the equivalent of a deactivated sensor, a sensor that breaks half way the experiment, and a sensor that starts to work half way the experiment:

```
Tnoisy(2,3,:) = NaN;
Tnoisy(5,1,1:floor(end/2)) = NaN;
Tnoisy(9,7,ceil(end/2):end) = NaN;
```

The incomplete tensor is visualized in Fig. 3.4. When computing the CPD of the incomplete tensor, relative errors of 0.3180, 0.2490 and 0.1736 on the azimuth factor matrix, the elevation factor matrix and the attenuated source matrix are obtained, respectively.

```
Uest = cpd(Tnoisy,R);
[err,P,D] = cpderr({A,E,S},Uest);
errA = err(1)
errE = err(2)
errS = err(3)
```

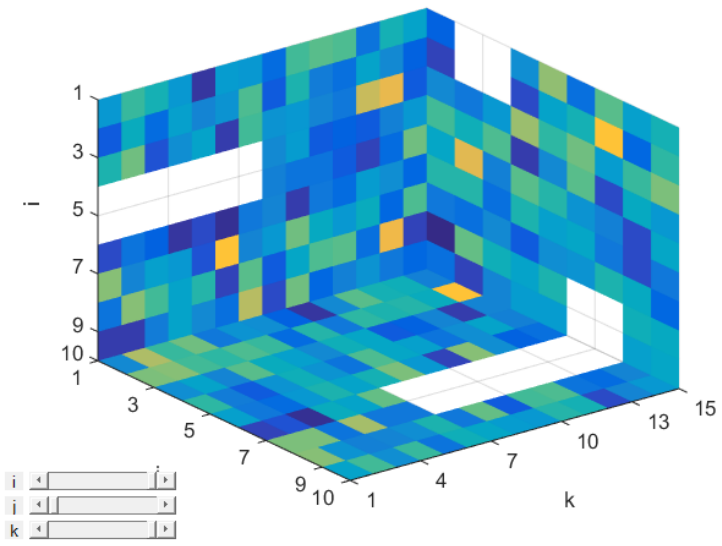

Fig. 3.4: Visualization of the data tensor in the case of broken sensors.

CHAPTER

FOUR

USING BASIC TENSORLAB FEATURES FOR ICA

Tags: `lagged covariance matrices`, `fourth-order cumulant`, `missing elements`

This demo illustrates the use of basic tensor tools in the context of independent component analysis (ICA). We consider an ICA variant that uses lagged covariance matrices and a variant that relies on a fourth-order cumulant. The demo also illustrates the decomposition of incomplete tensors. In this demo, we do not exploit structure such as symmetry, or orthogonality after prewhitening. The demo *Using advanced Tensorlab features for ICA* goes a step further and illustrates the use of the structured data fusion framework to impose constraints. More information on methods for ICA can for instance be found in [3].

A Matlab implementation of this demo is given in the `demo_basicica.m` file, which can be found in the demo folder of Tensorlab or downloaded here[1].

4.1 Instantaneous mixtures

An instantaneous mixture $\mathbf{x}(t) \in \mathbb{R}^Q$ of R independent components $\mathbf{s}(t) \in \mathbb{R}^R$ can be written as

$$\mathbf{x}(t) = \mathbf{M} \cdot \mathbf{s}(t) + \mathbf{n}(t),$$

in which $\mathbf{M}$ contains the mixing coefficients and temporally i.i.d. noise is captured by the additive noise term $\mathbf{n}(t)$. We consider two auxiliary signals that are identically uniformly distributed over $[-1, 1]$. Eeach signal has 10^4 samples. The two actual source signals are generated by passing these auxiliary signals through finite impulse response filters with coefficients $[-1, 1, 1]$ and $[1, 1, 1]$, respectively:

```
N = 10000; R = 2;
Saux = rand(N,R)*2-1;
S(:,1) = filter([-1 1 1],1,Saux(:,1));
S(:,2) = filter([1 1 1],1,Saux(:,2));
```

Fig. 4.1 shows the source histograms. The samples are confined to the interval $[-3, 3]$. The mixing matrix is taken equal to

$$\mathbf{M} = \begin{bmatrix} 1/\sqrt{2} & 2/\sqrt{5} \\ -1/\sqrt{2} & 1/\sqrt{5} \end{bmatrix}.$$

Independent identically distributed Gaussian noise is added with a signal-to-noise-ratio of 10 dB:

```
M = [1/sqrt(2) 2/sqrt(5); -1/sqrt(2) 1/sqrt(5)];
X = S*M.';
```

[1] http://www.tensorlab.net/demos/demo_basicica.zip

```
SNR = 10;
Xnoisy = noisy(X,SNR);
```

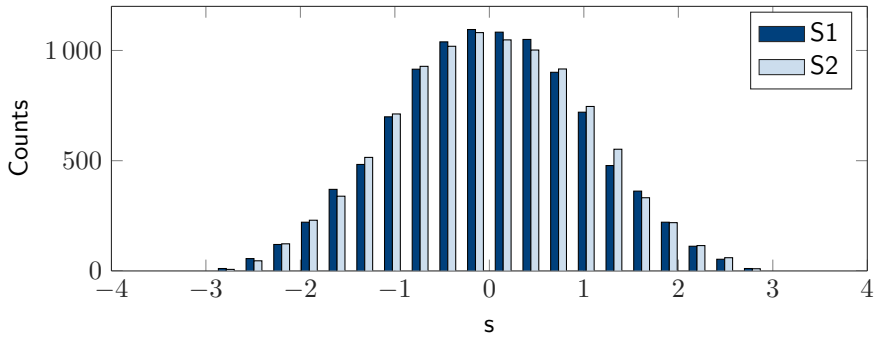

Fig. 4.1: Histogram of the source samples.

4.2 Second-order statistics

Consider a tensor $\mathcal{C}_x^{(2)}$ of which the slices are lagged covariance matrices. This tensor can be generated with the `scov` command:

```
lags = [0 1 2];
C2x = scov(Xnoisy,lags);
```

Each observed lagged covariance matrix equals the corresponding lagged covariance matrix of the source signals, multiplied by the mixing matrix in the first and second mode. As the lagged covariance matrices of the source signals are approximately diagonal, the tensor is approximately low-rank, as illustrated in Fig. 4.2.

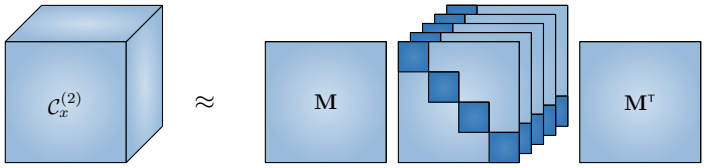

Fig. 4.2: Tensorizing the data by stacking lagged covariance matrices.

The mixing matrix can be recovered by decomposing $\mathcal{C}_x^{(2)}$ in a minimal number of rank-1 terms. We use the basic `cpd` command without exploiting the symmetry; hence, two estimates of the mixing matrix are obtained:

```
Uest = cpd(C2x,R);
Mest_mode1 = Uest{1};
Mest_mode2 = Uest{2};
```

Since the mixing matrix is square non-singular, the source signals can subsequently be estimated as

$$\hat{\mathbf{s}}(t) = \mathbf{M}^{-1} \cdot \mathbf{y}(t).$$

The `cpderr` command can be used to compute the relative error on the estimates of the mixing matrix, and to fix the indeterminacies for visualization purposes:

```
[err_scov_mode1,P1,D1,Mestscaled] = cpderr(M,Mest_mode1)
[err_scov_mode2,P2,D2] = cpderr(M,Mest_mode2)
Srec = Xnoisy*pinv(Mestscaled.');
```

The joint distributions of the signals are visualized in Fig. 4.3. The joint distribution of the source signals is confined to the square $[-3,3]\times[-3,3]$, and the components vary independently along the horizontal and vertical axis according to their distribution function (cf. Fig. 4.1). The edges of the figure in the middle illustrate that the observed signals cannot take values independent of one another. As a matter of fact, the observed joint distribution is equal to the joint distribution of the source signals, linearly transformed by the mixing matrix $\mathbf{M}$. The axes correspond to the columns of the identity matrix (left), the mixing matrix $\mathbf{M}$ (middle), and the matrix $\tilde{\mathbf{M}}^{-1}\cdot\mathbf{M}$ (right), with $\tilde{\mathbf{M}}$ the estimate of $\mathbf{M}$. The figure on the right shows that the estimated source signals are close to independent.

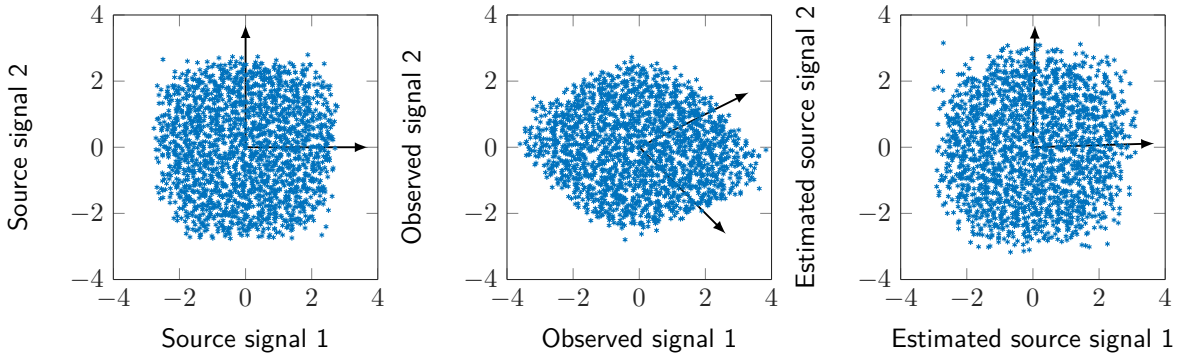

Fig. 4.3: Visualization of the joint distributions of the source signals (left), the observed signals (middle) and the estimated source signals (right). The axes correspond to the columns of the identity matrix (left), the mixing matrix $\mathbf{M}$ (middle), and the matrix $\tilde{\mathbf{M}}^{-1}\cdot\mathbf{M}$ (right), with $\tilde{\mathbf{M}}$ the estimate of $\mathbf{M}$.

4.3 Fourth-order statistics

Consider the fourth-order cumulant $\mathcal{C}_x^{(4)}$ of the observed signals, which can be computed using the `cum4` command:

```
C4x = cum4(Xnoisy);
```

The fourth-order cumulant of the sources is approximately diagonal because of the mutual statistical independence; hence, the tensor $\mathcal{C}_x^{(4)}$ can be decomposed in rank-1 terms as follows:

$$\mathcal{C}_x^{(4)} \approx \mathcal{C}_s^{(4)} \cdot_1 \mathbf{M} \cdot_2 \mathbf{M} \cdot_3 \mathbf{M} \cdot_4 \mathbf{M} = \left[\!\left[\mathcal{C}_s^{(4)}; \mathbf{M}, \mathbf{M}, \mathbf{M}, \mathbf{M}\right]\!\right],$$

with $\mathcal{C}_s^{(4)} = \text{diag}_4(c_{s_1}^{(4)}, c_{s_2}^{(4)}, \ldots, c_{s_R}^{(4)})$. All four factor matrices are equal to the mixing matrix. We only use the first factor matrix as an estimate:

```
Uest = cpd(C4x,R);
Mest_mode1 = Uest{1};
[err_cum,P,D] = cpderr(M,Mest_mode1)
```

4.4 Second-order statistics and missing values

If infinitely many samples are available, the i.i.d. Gaussian noise only affects the values on the diagonal of the covariance matrix for lag zero. These entries can be omitted from the tensor $\mathcal{C}_x^{(2)}$ as follows:

```
C2xi = C2x;
for i = 1:size(C2x,1)
  C2xi(i,i,1) = NaN;
end
```

The `cpd` command can be used to decompose the incomplete tensor `C2xi`:

```
Uest = cpd(C2xi,R);
Mest_mode1 = Uest{1};
[err_scovi,P,D] = cpderr(M,Mest_mode1)
```

The demo *Using advanced Tensorlab features for ICA* elaborates on noise modeling using advanced Tensorlab features.

CHAPTER

FIVE

USING ADVANCED TENSORLAB FEATURES FOR ICA

Tags: `lagged covariance matrices`, `sdf`, `nonnegativity`, `symmetry`, `missing elements`, `Toeplitz structure`, `imposing structure`

Many signals can be modeled as a mixture of independent components. In its most basic form, independent component analysis (ICA) tries to estimate the independent components from such mixtures without prior knowledge of the signals or mixing coefficients. Though many methods for ICA use higher-order statistics to solve this problem, this demo will focus on approaches using second-order statistics. More information on methods for ICA can for instance be found in [3].

For basic ICA techniques, we refer to the demo *Using basic Tensorlab features for ICA*. In this demo, we will consider variants of second-order statistics based ICA that illustrate more advanced Tensorlab features. A Matlab implementation of this demo is given in the `demo_advancedica.m` file, which can be found in the demo folder of Tensorlab or downloaded here[1].

5.1 Instantaneous mixtures

An instantaneous mixture $\mathbf{x}(t) \in \mathbb{R}^Q$ of R independent components $\mathbf{s}(t) \in \mathbb{R}^R$ can be written as

$$\mathbf{x}(t) = \mathbf{M} \cdot \mathbf{s}(t) + \mathbf{n}(t),$$

in which $\mathbf{M}$ contains the mixing coefficients. Temporally i.i.d. noise is captured by the additive noise term $\mathbf{n}(t)$. Here, the sources are assumed to be temporally coherent. All results below are computed for a mixture of three sources with five observed signals. Each signal has 10^4 samples and the signal-to-noise ratio is 10 dB.

In our examples, the temporally coherent sources are generated by passing 10^4 i.i.d. samples from a standard normal distribution through a finite impulse response filter. The filter coefficients, the entries of the mixing matrix M and the additive noise are taken from a standard normal distribution as well. The noise is scaled to obtain the desired signal-to-noise ratio. In Matlab, this is done as follows:

```
% Parameters
Q = 5;              % nb outputs
R = 3;              % nb sources
K = 20;             % nb slices
N = 1e4;            % nb samples
SNR = 10;           % Signal to noise ratio in dB
filterlength = 21;  % Filter length for sources

% Generate signals
```

[1] http://www.tensorlab.net/demos/demo_advancedica.zip

```
s = randn(N,R);
filt = randn(filterlength,R);
for r = 1:R
    s(:,r) = conv(s(:,r),filt(:,r),'same'); % temporally coherent sources
end
M = randn(Q,R);
x = s*M.'; % create mixture
x = noisy(x,SNR); % Additive Gaussian noise
```

By stacking lagged covariance matrices of the observations $E\{\mathbf{x}(t)\mathbf{x}(t+\tau)^{\mathsf{T}}\}$, the tensor $\mathcal{C}_{\mathbf{x}}^{(2)}$ shown in Fig. 5.1 is obtained. Because the source covariance matrices are close to diagonal, the figure actually shows the approximate canonical polyadic decomposition (CPD) of $\mathcal{C}_{\mathbf{x}}^{(2)}$. If this CPD is unique, its computation will return an estimate of the mixing matrix $\mathbf{M}$. Since the mixture is overdetermined, the mixing matrix estimate can subsequently be used to estimate the original source signals as

$$\hat{\mathbf{s}}(t) = \mathbf{M}^{\dagger} \cdot \mathbf{y}(t).$$

The rest of this section will show how this CPD can be computed using advanced Tensorlab features. We focus on the implementation of the different types of structure, for which we use Tensorlab's structured data fusion framework.

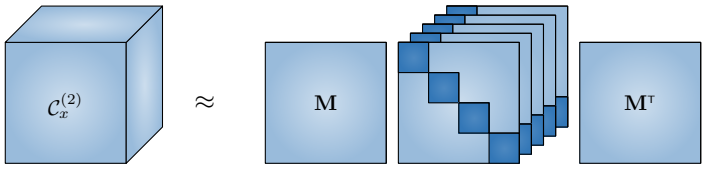

Fig. 5.1: Tensorizing the data by stacking lagged covariance matrices.

5.1.1 Without first slice

If infinitely many samples are available, the noise will only influence the diagonal of the covariance matrix with shift $\tau = 0$, which is the diagonal of the first frontal slice. We could simply set this noisy diagonal to missing, as explained in the *second-order statistics and missing values* section in the *basic ICA* demo. Here, we will gradually build up an approach that uses more advanced Tensorlab features.

Our first approach is to drop the entire first slice and compute the CPD of the remaining tensor with lags $\tau = \{1, \ldots, K-1\}$ instead of $\tau = \{0, \ldots, K-1\}$. First we compute this CPD without imposing any structure:

```
C2x = scov(x,1:K-1);
U = cpd(C2x,R);
cpderr(M,U{1})
```

which yields a relative error on $\mathbf{M}$ of `2.347691e-02`.

Fig. 5.1 shows that the factor matrices in the first and second mode are both equal to $\mathbf{M}$. This symmetry can be imposed in the structured data fusion framework.

First, the variables of the SDF algorithm need to be properly initialized. This can be done using the result of the unconstrained CPD, which has already been computed:

```
model.variables.M = U{1};
model.variables.C = U{3};
```

These variables can now be used to compose the factors that will appear in the decomposition:

```
model.factors.M = 'M';
model.factors.C = 'C';
```

In a next step, the tensor and the type of decomposition have to be specified. At this point, the symmetry is exploited by imposing that the factor matrices in the first two modes are both equal to `M`.

```
model.factorizations.example_cpdsym.data = C2x;
model.factorizations.example_cpdsym.cpd = {'M','M','C'};
```

It may be useful to verify the syntax and consistency of the model. This can be done using `sdf_check`, which indicates the correctness. The `print` option can be used to see the conversions made between variables, factors and factorizations:

```
sdf_check(model,'print')
```

```
Factor                    Size                 Comment
-------------------------------------------------------------------------
M                         [5x3]
C                         [19x3]

Factorization             Type       Factors   Comments
-------------------------------------------------------------------------
example_cpdsym (id = 1)   cpd        M,M,C
```

Finally, the model is computed using numerical optimization (we choose a nonlinear least squares method) and the relative estimation error of the result is determined:

```
sol = sdf_nls(model);
cpderr(M,sol.factors.M)
```

This approach leads to a relative error of `1.594028e-02`, which is a slight improvement on the previous result. Note that, by default, the syntax and consistency of the model is always verified automatically when using the SDF algorithms such as `sdf_nls` and `sdf_minf`.

5.1.2 With first slice

In the previous approaches, the first tensor slice was dropped to get rid of the noisy values on its diagonal. Here, the first slice will be kept and the noise effects will be explicitly modeled. In this case, computation of a simple unconstrained CPD leads to suboptimal results:

```
C2x = scov(x,0:K-1);
U = cpd(C2x,R);
cpderr(M,U{1})
```

This yields a relative error of `2.095566e-02`. Since only the diagonal of the first slice is perturbed, and actually not much, the effect on the result is limited.

Explicit modeling of the noise contributions may perhaps improve this result. Such additional modeling allows us to further illustrate Tensorlab's SDF framework.

First, the variables need to be initialized. Prior information can be used for this purpose, if it is available. Here, the results of the unconstrained CPD are re-used. To capture the standard deviations of the noise, an extra variable `noise` is added, initialized with random positive values.

```
model.variables.M = U{1};
model.variables.C = U{3};
model.variables.noise = rand(Q,1);
```

Construction of the factors requires some mathematical insight into the problem. The goal is to add a diagonal matrix holding noise variances to the first frontal slice of the tensor. Mathematically, we would like to construct

$$\mathbf{C}_{\mathbf{x}}^{(2)} = \mathbf{M}\mathbf{C}_{\mathbf{s}}^{(2)}\mathbf{M}^T + \Sigma_n^2$$

as the first frontal slice of $\mathcal{C}_{\mathbf{x}}^{(2)}$.

One can start by placing the variable `noise`, containing the noise standard deviations σ_{n_q}, in a diagonal matrix:

$$\mathbf{\Sigma}_n = \begin{bmatrix} \sigma_{n_1} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_{n_Q} \end{bmatrix}.$$

To obtain a matrix with variances on the diagonal, the matrix above can be used as factor matrix in the first and second mode, leading to

$$\begin{bmatrix} \sigma_{n_1} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_{n_Q} \end{bmatrix} \cdot \begin{bmatrix} \sigma_{n_1} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_{n_Q} \end{bmatrix}^{\mathsf{T}} = \begin{bmatrix} \sigma_{n_1}^2 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_{n_Q}^2 \end{bmatrix}$$

To integrate this diagonal perturbation in the decomposition, the factors in the first two modes can be concatenated, i.e., instead of the factor matrix $\mathbf{M}$, the matrix $[\mathbf{M}, \mathbf{\Sigma}_n]$ is used. However, the perturbation must only appear in the first frontal slice. This can be achieved by appending a matrix with ones in the first row and zeros elsewhere to the factor matrix in the third mode, yielding

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1R} & 1 & 1 & \cdots & 1 \\ c_{21} & c_{22} & \cdots & c_{2R} & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ c_{K1} & c_{K2} & \cdots & c_{KR} & 0 & 0 & \cdots & 0 \end{bmatrix}$$

as factor matrix in the third mode.

The Matlab implementation of the explanation above is quite straightforward. To construct a diagonal factor (sub)matrix from the variable `noise`, the `struct_diag`-function can be used:

```
diagmat = @(z,task)struct_diag(z,task);
model.factors.M = {'M',{'noise',diagmat}};
model.factors.C = {'C',[ones(1,Q); zeros(K-1,Q)]};
```

Note that concatenating in SDF is similar as in standard Matlab notation: a comma represents horizontal concatenation and a semicolon represents vertical concatenation. The rest of the procedure is the same as above: the data and the decomposition are specified, and the model is fitted using a nonlinear least squares method.

```
model.factorizations.example_cpdnoisemodel.data = C2x;
model.factorizations.example_cpdnoisemodel.cpd = {'M','M','C'};
sol = sdf_nls(model,'MaxIter',500);
cpderr(M,sol.factors.M)
```

At the cost of a more complicated model, a more accurate estimate with a relative error of `1.044207e-02` is obtained in this case.

An alternative approach is to denote possibly unreliable values as missing. As mentioned earlier, the i.i.d. noise only affects the diagonal of the first frontal slice of the tensor $\mathcal{C}_{\mathbf{x}}^{(2)}$ for long enough data sets. It thus suffices to denote these elements on the first diagonal as missing:

```
C2xmiss = C2x;
for q = 1:Q
    C2xmiss(q,q,1) = NaN;
end
```

Constrained CPDs of incomplete tensors can be computed in the same way as those of full tensors. For instance, symmetry can be imposed as follows:

```
model.variables.M = U{1};
model.variables.C = U{3};
model.factors.M = {'M'};
model.factors.C = {'C'};
model.factorizations.example_cpdmissing.data = C2xmiss;
model.factorizations.example_cpdmissing.cpd = {'M','M','C'};
sol = sdf_minf(model,'MaxIter',500);
cpderr(M,sol.factors.M)
```

This results in a relative estimation error of `2.021610e-02`.

5.1.3 With prewhitening

In the original SOBI-method, the covariance of the observed signals without shift is used to prewhiten the data [5].

```
[E,D] = eig(cov(x));
z = (E*abs(D)^(-1/2)*E.'*(x.')).';
```

After this transformation, the tensor with covariance slices has orthogonal factor matrices in the first and second mode. In the SDF-framework, this orthogonality can be imposed as well. First, the variables are initialized:

```
model.variables.M = randn(R*(Q - (R-1)/2),1);
model.variables.C = randn(K,R);
```

Columnwise orthonormal matrices of size $Q \times R$ are fully determined by $R(Q-(R-1)/2)$ variables, hence the size of the initialization vector. In a next step, the variables are mapped to a columnwise orthonormal factor matrix, using the `struct_orth` function:

```
ortho = @(z,task)struct_orth(z,task,[Q R]);
model.factors.M = {'M',ortho};
model.factors.C = {'C'};
```

Next, the data and factorization are specified and the model is computed:

```
model.factorizations.example_cpdprewhit.data = C2x;
model.factorizations.example_cpdprewhit.cpd = {'M','M','C'};
sol = sdf_nls(model,'MaxIter',500);
```

Taking into account the prewhitening, it is found that the relative estimation error is `3.009932e-02`.

```
Mest = E*abs(D)^(1/2)*E.'*sol.factors.M;
cpderr(M,Mest)
```

5.2 Nonstationary sources

We now consider an instantaneous mixture of nonstationary independent source signals

$$\mathbf{x}(t) = \mathbf{M} \cdot \mathbf{s}(t),$$

in which $\mathbf{x}(t)$ are the observed mixtures, $\mathbf{M}$ contains the mixing coefficients, and $\mathbf{s}(t)$ represent the nonstationary independent sources. For simplicity, we assume that there is no additive noise. Here, we do not assume that the sources are temporally coherent, but that they are nonstationary.

To identify mixtures of such nonstationary source signals, the nonstationarity can be exploited in a similar way as the temporal coherence in the example before. To do this, a tensor-based version of the method presented in [6] will be used.

First, the observed signals are divided into segments, assuming that the source statistics can approximately be considered constant over each segment. For the k th segment, computing the covariance yields

$$\mathbf{C}_{\mathbf{x}_k} = \mathbf{M} \cdot \mathbf{C}_{\mathbf{s}_k} \cdot \mathbf{M}^{\mathrm{T}}.$$

The matrices $\mathbf{C}_{\mathbf{s}_k}$ are diagonal due to the independence of the inputs. By stacking the observed covariance matrices, a third-order tensor is obtained.

```
% Segment signals and compute covariances
Xs = segmentize(x,'segsize',N/K);
C2x = covd(Xs,'dims',[1 3],'perm',true);
```

The factor matrices of the CPD in the first and second mode are again both equal to $\mathbf{M}$. The factor matrix in the third mode contains the source variances. Since these are always positive, it makes sense to impose nonnegativity on this factor matrix. In Tensorlab, nonnegativity can be imposed on a variable using the `struct_nonneg` structure:

```
model.variables.M = randn(Q,R);
model.variables.C = rand(K,R);
model.factors.M = 'M';
model.factors.C = {'C',@struct_nonneg};
model.factorizations.example_cpdnonneg.data = C2x;
model.factorizations.example_cpdnonneg.cpd = {'M','M','C'};
sol = sdf_nls(model,'MaxIter',500,'TolFun',1e-8);
cpderr(M,sol.factors.M)
```

This procedure leads to an estimation error of `1.186503e-01` for a mixture of three sources with five observed signals and with signals of length 10^4 samples and a signal-to-noise ratio of 10 dB.

5.3 Convolutive mixtures

A convolutive mixture of R sources $s_r(t)$ in M measurements $x_m(t)$ can be written as

$$x_m(t) = \sum_{r=1}^{R} \sum_{l=0}^{L} h_{mr}(l) s_r(t-l) \qquad \text{for } m = 1, \ldots, M,$$

in which $h_{mr}(l)$ represents the convolutive mixing filter from input r to output m and L is the maximum filter delay (which implies that the filter length is $L+1$). We assume for simplicity that there is no additive noise. To

write this relation as a matrix product, first construct

$$\mathbf{s}(n) = [s_1(n), \ldots, s_1(n-(L+L')+1), \ldots, s_R(n-(L+L')+1)]^{\mathsf{T}}$$
$$\mathbf{x}(n) = [x_1(n), \ldots, x_1(n-L'+1), \ldots, x_M(n), \ldots, x_M(n-L'+1)]^{\mathsf{T}},$$

with L' chosen so that $ML' \geq R(L+L')$. The mixture can now be written as $\mathbf{x}(n) = \mathbf{H} \cdot \mathbf{s}(n)$, in which $\mathbf{H}$ is a Toeplitz-block matrix given by

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_{11} & \cdots & \mathbf{H}_{1R} \\ \vdots & \ddots & \vdots \\ \mathbf{H}_{M1} & \cdots & \mathbf{H}_{MR} \end{bmatrix} \in \mathbb{R}^{ML' \times R(L+L')}$$

with blocks

$$\mathbf{H}_{ij} = \begin{bmatrix} h_{ij}(0) & \cdots & h_{ij}(L) & \cdots & 0 \\ & \ddots & \ddots & \ddots & \\ 0 & \cdots & h_{ij}(0) & \cdots & h_{ij}(L) \end{bmatrix} \in \mathbb{R}^{L' \times (L+L')}.$$

Note that the choice of L' implies that $\mathbf{H}$ will always have at least as many rows as columns.

Now, a similar approach as in the instantaneous case can be used. By stacking lagged covariance matrices of the outputs $E\{\mathbf{x}(t)\mathbf{x}(t+\tau)^{\mathsf{T}}\}$, the tensor shown in Fig. 5.2 is obtained. This tensor can be decomposed using a block term decomposition in multilinear rank-$(L+L', L+L', \cdot)$ terms. In Fig. 5.2, the matrix $\mathbf{H}_k$ represents the k th block column of $\mathbf{H}$ and the core tensors are formed by the block diagonals of the covariances of $\mathbf{s}(n)$. This decomposition can be computed using the SDF-framework of Tensorlab.

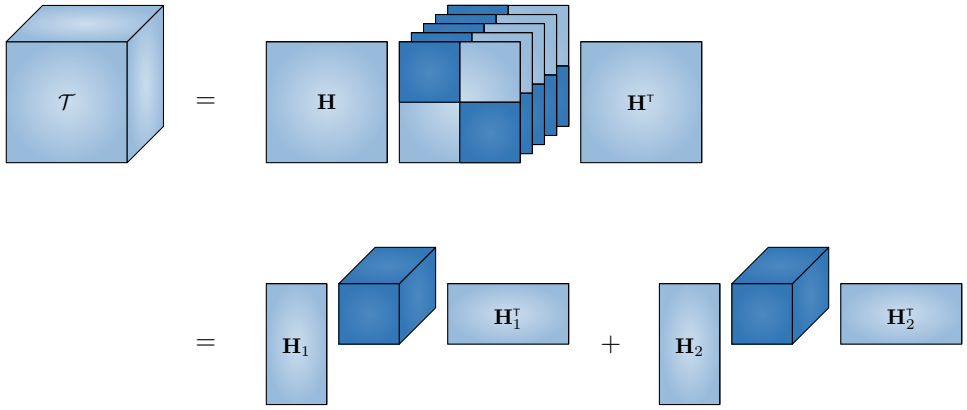

Fig. 5.2: Stacking the lagged covariance matrices for convolutive mixtures leads to a block-term decomposition in rank-$(L+L', L+L', \cdot)$ terms.

First, the filter coefficients for each submatrix of $\mathbf{H}$ and the core tensors have to be initialized. For a mixture of two sources having three outputs, this is done by:

```
model.variables.H11 = randn(L+1,1);
model.variables.H21 = randn(L+1,1);
model.variables.H31 = randn(L+1,1);
model.variables.H12 = randn(L+1,1);
model.variables.H22 = randn(L+1,1);
model.variables.H32 = randn(L+1,1);
model.variables.Core1 = randn(L+Lacc,L+Lacc,K);
```

```
model.variables.Core2 = randn(L+Lacc,L+Lacc,K);
```

Note that the parameter L' is represented by `Lacc`. Next, the Toeplitz structure of the submatrices $\mathbf{H}_{ij}$ is implemented. As explained in the documentation of `struct_toeplitz`, it is possible to pass along known diagonals under or above the sought ones. Here, the fact that the first and last $L'-1$ diagonals of $\mathbf{H}_{ij}$ are zero, is taken into account in:

```
toepl = @(z,task)struct_toeplitz(z,task,[Lacc L+Lacc], ...
zeros(Lacc-1,1),zeros(Lacc-1,1));
```

This code will provide a structure that pads the variable `z` with $L'-1$ zeros at both ends and then uses these elements for the diagonals of a Toeplitz matrix. The rest of the code is similar to the previous examples: the structure is imposed when creating the factors, the data and decomposition are specified, and the model is computed:

```
model.factors.H1 = {{'H11',toepl};{'H21',toepl};{'H31',toepl}};
model.factors.H2 = {{'H12',toepl};{'H22',toepl};{'H32',toepl}};
model.factors.Core1 = {'Core1'};
model.factors.Core2 = {'Core2'};
model.factors.C = eye(K);
model.factorizations.example_btdconv.data = T;
model.factorizations.example_btdconv.btd = {{'H1','H1','C','Core1'},
                                            {'H2','H2','C','Core2'}};
options = struct;
options.Display = 50;
options.MaxIter = 500;
options.TolFun = 1e-8;
options.TolX = 1e-8;
sol = sdf_minf(model,options);

Pest = [sol.variables.H11, sol.variables.H12; sol.variables.H21,...
    sol.variables.H22; sol.variables.H31, sol.variables.H32];

cpderr(P,Pest)
```

Because there is no factor matrix in the third mode, an identity matrix is used in that mode. The matrices `Pest` and `P` are constructed to easily compare the computed solution to the real one. The obtained accuracy is `1.033602e-02` when the signals have 10^4 samples and the signal-to-noise ratio is 20 dB in a mixture of two sources having three outputs and $L = 1, L' = 2$.

CHAPTER

SIX

INDEPENDENT VECTOR ANALYSIS

Tags: `sdf` , `coupled decompositions`

In independent vector analysis (IVA), the goal is mixture identification or signal separation for a collection of disjoint but coupled data sets. Within the same data set, the source signals are assumed to be statistically independent, while between data sets, source signals may be correlated. IVA can thus be seen as a multi-set generalization of independent component analysis (ICA). More information can for instance be found in [4].

A Matlab implementation of this demo is given in the `demo_iva.m` file, which can be found in the demo folder of Tensorlab or downloaded here[1].

We consider two data sets A and B, which we take of the same size for convenience. In each data set, the measured signals $\mathbf{x}(t) \in \mathbb{R}^K$ can be written as an instantaneous mixture of R independent components $\mathbf{s}(t) \in \mathbb{R}^R$:

$$\mathbf{x}_A(t) = \mathbf{M}_A \cdot \mathbf{s}_A(t) + \mathbf{n}_A(t),$$
$$\mathbf{x}_B(t) = \mathbf{M}_B \cdot \mathbf{s}_B(t) + \mathbf{n}_B(t),$$

in which $\mathbf{M}_A$ and $\mathbf{M}_B \in \mathbb{R}^{K \times R}$ are the mixing matrices, $\mathbf{n}_A(t)$ and $\mathbf{n}_B(t) \in \mathbb{R}^K$ represent additive Gaussian noise, and the subscripts denote the data set.

We use the sinusoidal sources depicted in Fig. 6.1, which are constructed as

```
L = 2;                                % number of data sets
frequencies = (1:R)*2*pi;
phases = (1:L)*pi/L/2;

S = cell(R,L);
for l = 1:L
    for r = 1:R
        S{r,l} = cos(frequencies(r)*t+phases(l));
    end
end
```

The empirical cross-correlation within each data set is set to zero by using sinusoids with the same phase but with frequencies that are integer multiples of each other. Over the different data sets, only the phase is different, which implies that sources having the same frequency will be empirically correlated.

The measured signals $\mathbf{x}_A(t)$ and $\mathbf{x}_B(t)$ can then be constructed as

```
SNR = 10;       % Signal to noise ratio (in dB)

for l = 1:L
```

[1] http://www.tensorlab.net/demos/demo_iva.zip

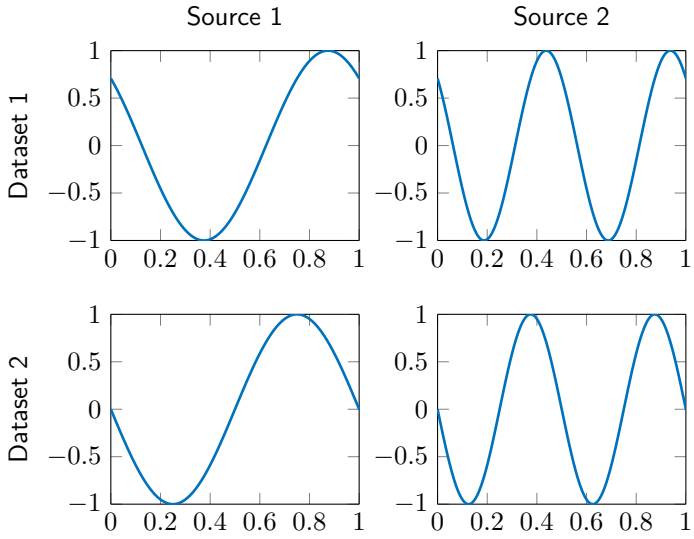

Fig. 6.1: The sources in the two data sets.

```
    M{l} = randn(K,R);
    X{l} = M{l}*cell2mat(S(:,l));    % create mixture
    X{l} = noisy(X{l},SNR);          % add noise
end
```

If the source signals are temporally coherent, one can compute lagged covariances of the measured signals, both in and between the data sets, and stack them in tensors. For instance, the lagged covariance matrices of the signals from data set A can be stacked into a third-order tensor

$$\begin{aligned}(\mathcal{C}_{\mathbf{x}}^{A})_{:,:,i} &= \mathbf{C}_{\mathbf{x}}^{A}(\tau_i) \\ &= E\left\{\mathbf{x}_A(t)\mathbf{x}_A(t+\tau_i)^{\mathsf{T}}\right\} \\ &= \mathbf{M}_A \cdot \mathbf{C}_{\mathbf{s}}^{A}(\tau_i) \cdot \mathbf{M}_A^{\mathsf{T}}.\end{aligned}$$

Because the source signals within each data set are statistically independent, the matrices $\mathbf{C}_{\mathbf{s}}^{A}(\tau_i)$ are diagonal, as illustrated in Fig. 6.2. The figure actually shows the canonical polyadic decomposition (CPD) of $\mathcal{C}_{\mathbf{x}}^{A}$. For the second data set, a tensor $\mathcal{C}_{\mathbf{x}}^{B}$ is obtained, which displays a similar low-rank CPD structure.

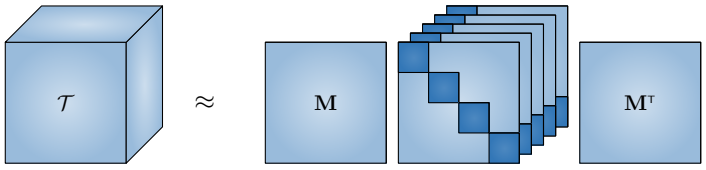

Fig. 6.2: Tensorizing the data by stacking lagged covariance matrices.

We also compute lagged covariances across data sets:

$$\begin{aligned}(\mathcal{C}_{\mathbf{x}}^{AB})_{:,:,i} &= \mathbf{C}_{\mathbf{x}}^{AB}(\tau_i) \\ &= E\left\{\mathbf{x}_A(t)\mathbf{x}_B(t+\tau_i)^{\mathsf{T}}\right\} \\ &= \mathbf{M}_A \cdot \mathbf{C}_{\mathbf{s}}^{AB}(\tau_i) \cdot \mathbf{M}_B^{\mathsf{T}}.\end{aligned}$$

Since there are only pair-wise correlations between the sources of data sets A and B, the matrix $\mathbf{C}_{\mathbf{s}}^{AB}(\tau_i)$ is diagonal as well. This implies that the tensor $\mathcal{C}_{\mathbf{x}}^{AB}$ has a low-rank CPD structure with factor matrices that also appear in the decompositions of the tensors $\mathcal{C}_{\mathbf{x}}^{A}$ and $\mathcal{C}_{\mathbf{x}}^{B}$.

The three tensors $\mathcal{C}_{\mathbf{x}}^{A}$, $\mathcal{C}_{\mathbf{x}}^{B}$ and $\mathcal{C}_{\mathbf{x}}^{AB}$ can now be jointly decomposed. Note that the number of tensors can be higher if more data sets are combined, but the principle remains the same. In the structured data fusion-framework of Tensorlab, the coupled decomposition can be computed as follows. First, the variables are initialized by

```
model = struct;
model.variables.M_A = randn(K,R);              % Mixing matrix data set A
model.variables.M_B = randn(K,R);              % Mixing matrix data set B
model.variables.D_AA = randn(numel(shifts),R); % Factor in third mode for xcovA
model.variables.D_AB = randn(numel(shifts),R); % Factor in third mode for xcovAB
model.variables.D_BB = randn(numel(shifts),R); % Factor in third mode for xcovB
```

Next, the variables are mapped to the factors that appear in the decompositions. No specific transformation is needed in this example.

```
model.factors.M_A = 'M_A';
model.factors.M_B = 'M_B';
model.factors.D_AA = 'D_AA';
model.factors.D_AB = 'D_AB';
model.factors.D_BB = 'D_BB';
```

Then the different data tensors are specified and the type of decomposition is chosen. At this stage, the coupling between the decompositions is implemented by specifying the same factor matrix in several decompositions:

```
model.factorizations.T11.data = xcovA;
model.factorizations.T11.cpd = {'M_A','M_A','D_AA'};
model.factorizations.T12.data = xcovAB;
model.factorizations.T12.cpd = {'M_A','M_B','D_AB'};
model.factorizations.T22.data = xcovB;
model.factorizations.T22.cpd = {'M_B','M_B','D_BB'};
```

The user can verify the syntax and the consistency of the model by using `sdf_check`, possibly with the `print` option:

```
sdf_check(model,'print')
```

Finally, the decompositions are computed using numerical optimization, for which we choose a nonlinear least squares method:

```
sol = sdf_nls(model,'Display',100,'MaxIter',200);
```

To verify the result, we can use the `cpderr` function, which takes care of the scaling and permutation ambiguities of the CPD:

```
disp('Error on first mixing matrix:')
cpderr(M{1},sol.factors.M_A)
disp('Error on second mixing matrix:')
cpderr(M{2},sol.factors.M_B)
```

This leads to relative errors of `0.0315` and `0.0576` on the first and second mixing matrix, respectively.

CHAPTER

SEVEN

GPS DEMO: PREDICTING USER INVOLVEMENT

Tags: `sdf` , `nonnegativity` , `missing elements` , `imposing structure`

Using real life data, we try to predict if a user participates in an activity at a certain location using Tensorlab's structured data fusion framework. The GPS dataset consists of a tensor and some additional matrices [1]:

- `UserLocAct` : a third order tensor containing how many times a user has participated in an activity at a certain location;
- `UserUser` : similarity between users;
- `LocFea` : each location is described by features. Each feature counts certain point of interest;
- `ActAct` : correlation between activities;
- `UserLoc` : how many times a user has visited a location.

We try to approximate these datasets using a low-rank model with shared components as shown in Fig. 7.1. Following [2], we formulate the problem mathematically as

$$\begin{aligned}\underset{\mathbf{U},\mathbf{L},\mathbf{A},\mathbf{F},\mathbf{d}_{\mathsf{uu}},\mathbf{d}_{\mathsf{aa}},\mathbf{d}_{\mathsf{ul}}}{\text{minimize}} \quad & \frac{\omega_1}{2}\,||\mathcal{T}-[\![\mathbf{U},\mathbf{L},\mathbf{A}]\!]||_{\mathsf{F}}^2+\frac{\omega_2}{2}\,||\mathbf{M}_{\mathsf{uu}}-\mathbf{U}\mathbf{D}_{\mathsf{uu}}\mathbf{U}^{\mathsf{T}}||_{\mathsf{F}}^2+\\ & \frac{\omega_3}{2}\,||\mathbf{M}_{\mathsf{ul}}-\mathbf{U}\mathbf{D}_{\mathsf{ul}}\mathbf{L}^{\mathsf{T}}||_{\mathsf{F}}^2+\frac{\omega_4}{2}\,||\mathbf{M}_{\mathsf{lf}}-\mathbf{L}\mathbf{F}^{\mathsf{T}}||_{\mathsf{F}}^2+\\ & \frac{\omega_5}{2}\,||\mathbf{M}_{\mathsf{aa}}-\mathbf{A}\mathbf{D}_{\mathsf{aa}}\mathbf{A}^{\mathsf{T}}||_{\mathsf{F}}^2\,.\end{aligned}$$

The matrices $\mathbf{U}$, $\mathbf{A}$, $\mathbf{L}$ and $\mathbf{F}$ are the variables which are used to predict the user involvement. The extra variables $\mathbf{d}_{\mathsf{uu}}$, $\mathbf{d}_{\mathsf{ul}}$, and $\mathbf{d}_{\mathsf{aa}}$ are needed to capture scaling differences between datasets. The matrices $\mathbf{D}_{\mathsf{uu}}$, $\mathbf{D}_{\mathsf{ul}}$, and $\mathbf{D}_{\mathsf{aa}}$ are diagonal matrices with the respective vectors on their diagonal. The parameters ω_i are the weights for each term.

The `demo_gps.m` file contains a Matlab implementation of this demo and can be downloaded here[1].

7.1 Obtaining the data

The GPS dataset can be obtained from Microsoft Research[2]. The demo file tries to download the files automatically.

[1] http://www.tensorlab.net/demos/demo_gps.zip
[2] http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/

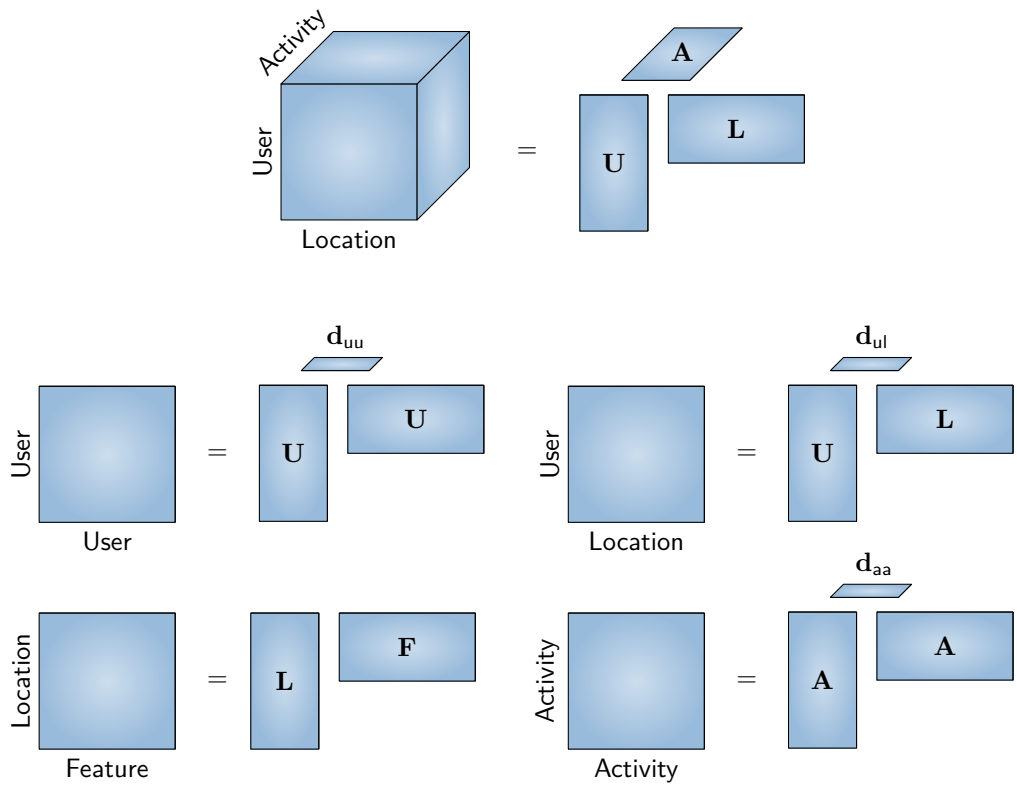

Fig. 7.1: The different datasets and their relations.

7.2 Preprocessing

Here, we will only predict whether or not a user participates in an activity at a certain location. Therefore, the number of participations is ignored:

```
UserLocAct(UserLocAct > 0) = 1;
```

Users without any user-location-activity data are ignored as we are unable to evaluate how good the algorithms predicts whether a user participates in an activity at a certain location. This way 18 users are removed from `UserLocAct`, `UserLoc` and `UserUser`.

```
u = any(any(UserLocAct,2),3);
UserLocAct = UserLocAct(u,:,:);
UserLoc = UserLoc(u,:);
UserUser = UserUser(u,u);
```

Finally, we normalize the location-feature data by normalizing the columns such that they sum to one:

```
LocFea = bsxfun(@rdivide,LocFea,sum(LocFea));
LocFea(~isfinite(LocFea)) = 0;
```

7.3 Predicting random events

First, we try to predict if a user will perform an activity at a certain location by randomly removing entries of `UserLocAct` (see Fig. 7.2, left) . This can be achieved easily by setting the missing entries to `nan` . The `fmt` function reformats the tensor as an incomplete tensor. (This last step is automatically performed by all algorithms if needed.)

```
missing = 0.80;
sel = randperm(numel(UserLocAct), round(missing*numel(UserLocAct)));
UserLocAct_missing = UserLocAct;
UserLocAct_missing(sel) = nan;
UserLocAct_missing = fmt(UserLocAct_missing);
```

Fig. 7.2: The two scenarios: 20% of known entries (left) and 50 missing users (right).

7.3.1 Without data fusion

As a baseline test, only the `UserLocAct_missing` tensor will be used. We model the tensor by a rank $R = 2$ CPD and add regularization to prevent overfitting. Mathematically, we solve:

$$\underset{\mathbf{U},\mathbf{L},\mathbf{A}}{\text{minimize}} \quad \frac{\omega_1}{2}\,||\mathcal{T} - [\![\mathbf{U},\mathbf{L},\mathbf{A}]\!]||_{\text{F}}^2 + \frac{\omega_2}{2}(||\mathbf{U}||_{\text{F}}^2 + ||\mathbf{L}||_{\text{F}}^2 + ||\mathbf{A}||_{\text{F}}^2).$$

In Tensorlab we use the `sdf_nls` method to solve this model. The `cpd_nls` method can be used as well if regularization is not required. As usual, the SDF model is defined in three steps: variable definition, factor matrix definition and model choice for each dataset.

First three variables are defined: `u` for the users, `l` for the location and `a` for the activities. For each variable, we define a random initial guess. A better guess can be used if available from prior knowledge, or from previously computed models.

```
R = 2;
model.variables.u   = rand(nbUsers,R);
model.variables.l   = rand(nbLocations,R);
model.variables.a   = rand(nbActivities,R);
```

Next, we define the factor matrices. To illustrate the use of constraints, we impose nonnegativity constraints on all factor matrices.

```
model.factors.U   = {'u', @struct_nonneg};
model.factors.L   = {'l', @struct_nonneg};
model.factors.A   = {'v', @struct_nonneg};
```

Finally, the tensor is modeled. A CPD will be used here. We also add L2 regularization by defining a second factorization. In the case of regularization, the `data` part can be omitted if it is zero. Note that `regL2` is implemented to add a term $||\cdot||_{\text{F}}^2$ for each factor matrix separately.

```
model.factorizations.ula.data = UserLocAct_missing;
model.factorizations.ula.cpd  = {'U','L','A'};

model.factorizations.reg.regL2 = {'U','L','A'};
```

We can investigate the resulting model using the language parser `sdf_check`:

```
sdf_check(model, 'print');
```

If no errors have been made, the following overview is printed:

```
Factor                Size                  Comment
-------------------------------------------------------------------------
U                     [146x2]
L                     [168x2]
A                     [5x2]

Factorization         Type        Factors              Comments
-------------------------------------------------------------------------
ula (id = 1)          cpd         U,L,A
reg (id = 2)          regL2       U,L,A
```

In the Matlab Command Window, the factors are links. These links can be used to investigate the model further. For example, when clicking on the factor `U`, the steps in its construction are revealed:

```
Expansion of factor U (id = 1) (view):
  (1,1)  u               [146x2]    struct_nonneg        Factor          [146x2]
```

We can now call the `sdf_nls` routine to solve the problem using extra optimization options. The one that most affects the results here is the `RelWeights` option, which gives a weight to each factorization, in casu the tensor and the regularization. The `lambdareg` parameter has experimentally been chosen equal to `0.01`:

```
options = struct;
options.Display    = 10;
options.MaxIter    = 500;
options.TolFun     = 1e-8;
options.TolX       = 1e-4;
options.CGMaxIter  = 150;
options.RelWeights = [1 lambdareg];

% call the solver
[sol,out] = sdf_nls(model,options);
```

To measure the quality, we use performance curves and the area-under-curve (AUC) value. Here we compare the predicted values `pred` with the removed values. First the predicted values are computed by reconstructing the tensor using the computed factor matrices. The result can be seen in Fig. 7.3.

```
pred = cpdgen({sol.factors.U,sol.factors.L,sol.factors.A});
lab  = UserLocAct(sel);
pred = pred(sel);

[X,Y,~,AUC] = perfcurve(lab(:), pred(:),1);
plot(X,Y);
```

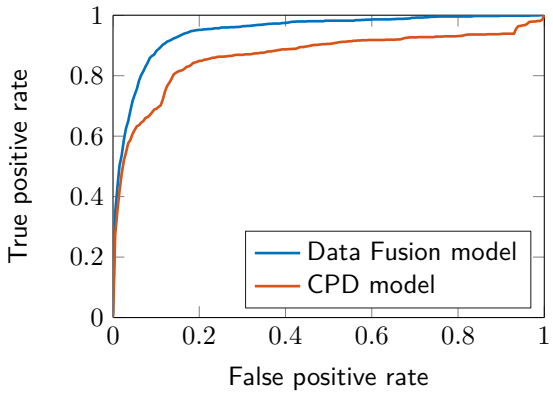

Fig. 7.3: Performance curves for the regularized CPD model and the model using data fusion. The respective AUC values are 0.865 and 0.948.

7.3.2 With data fusion

Now, we enrich the model by using the other datasets as well. We will need an extra variable `f` to model the features in the `LocFea` tensor.

```
model.variables.f = rand(size(LocFea, 2), R);
```

Also, because the data in the tensor and the matrices may be scaled differently, we introduce extra variables `duu`, `daa` and `dul` to capture the scaling differences in the `UserUser`, the `ActAct` and the `UserLoc` matrices. We do not need an extra variable for the `LocFea` matrix as the scaling differences are captured in `F`.

```
model.variables.duu = rand(1,R);
model.variables.daa = rand(1,R);
model.variables.dul = rand(1,R);
```

Like before, we create the corresponding factor matrices. We use a more dynamic approach to apply constraints here. If `constr = {}`, `cat(2, 'u', constr)` is equivalent to `{'u'}`; if `constr = {@struct_nonneg}`, `cat(2, 'u', constr)` becomes `{'u', @struct_nonneg}` like we used above. This more dynamic notation becomes useful when experimenting with different types of constraints.

```
constr = {};
model.factors.u   = cat(2,'u',constr);
model.factors.l   = cat(2,'l',constr);
model.factors.a   = cat(2,'a',constr);
model.factors.f   = cat(2,'f',constr);
model.factors.duu = cat(2,'duu',constr);
model.factors.daa = cat(2,'daa',constr);
model.factors.dul = cat(2,'dul',constr);
```

Alternatively, the `transform` field can be used. Instead of defining the factors, we can write:

```
model.transform = {{'u', 'l', 'a', 'f', 'duu', 'daa', 'dul'}, ...
                   {'U', 'L', 'A', 'F', 'Duu', 'Daa', 'Dul'}, ...
                   @struct_nonneg};
```

We define the models used for the different matrices. Here we choose a CP model for all matrices and apply L2 regularization to all factor matrices. Note that the factor matrices `Duu`, `Daa` and `Dul` can be added without

any problem, because a matrix is a tensor with the third dimension equal to 1.

```
model.factorizations.ula.data = UserLocAct_missing;
model.factorizations.ula.cpdi = {'U','L','A'};

Model.factorizations.uu.data = UserUser;
model.factorizations.uu.cpd  = {'U','U','Duu'};

model.factorizations.lf.data = LocFea;
model.factorizations.lf.cpd  = {'L','F'};

model.factorizations.aa.data = ActAct;
model.factorizations.aa.cpd  = {'A','A','Daa'};

model.factorizations.ul.data = UserLoc;
model.factorizations.ul.cpd  = {'U','L','Dul'};

model.factorizations.reg.regL2 = {'U','L','A','F','Duu','Daa','Dul'};
```

Note that we used `cpdi` instead of `cpd` for the `ula` factorization. The `cpdi` factorization type activates a specialized kernel for incomplete tensors, which usually improves the convergence behavior compared to `cpd`.

We can again use the `sdf_check` method to get an overview of the model:

```
sdf_check(model, 'print');
```

```
Factor                Size                  Comment
-------------------------------------------------------------------------
U                     [146x2]
L                     [168x2]
A                     [5x2]
F                     [14x2]
Duu                   [1x2]
Daa                   [1x2]
Dul                   [1x2]

Factorization         Type       Factors              Comments
-------------------------------------------------------------------------
ula (id = 1)          cpdi       U,L,A
uu (id = 2)           cpd        U,U,Duu
lf (id = 3)           cpd        L,F
aa (id = 4)           cpd        A,A,Daa
ul (id = 5)           cpd        U,L,Dul
reg (id = 6)          regL2      U,L,A,F,Duu,Daa,Dul
```

The `sdf_nls` method is used to compute the results for the model. The relative weights have been determined using grid search.

```
options.RelWeights = [1 0.005 0.001 0.005 0.001 1e-6];
[sol,out] = sdf_nls(model,options);
```

The ROC curve of the result can be seen in Fig. 7.3. Using the extra information clearly improved the area-under-curve (AUC).

7.4 Predicting for new persons

In this second scenario 50 random persons are removed from the tensor to simulate a cold start scenario (see Fig. 7.2, right). We are unable to predict values for the deleted users when we only use the `UserLocAct` tensor. Therefore, we use the information from the other datasets to predict the missing values.

The strategy is the same as above. First the user-location-activity information for 50 persons is removed and the interactions between the datasets are modeled. Next the model is solved using the `sdf_nls` model for different parameters. Finally, we compute the performance for the different models.

```
nbmissingpersons = 50;
sel = randperm(size(UserLocAct,1), nbmissing);
UserLocAct_missing = UserLocAct;
UserLocAct_missing(sel,:,:) = nan; % Remove persons
UserLocAct_missing = fmt(UserLocAct_missing); % format as incomplete tensor
```

The model is the same as in the previous section, with the following parameters, found using grid search:

```
constr = {@struct_nonneg};
options.RelWeights = [1 1e-4 0.1 1e-4 1e-4 0.08];
```

The performance can be computed as follows and the ROC curve for the result is shown in Fig. 7.4. The area under curve (AUC) is 0.968.

```
pred = cpdgen({sol.factors.U,sol.factors.L,sol.factors.A});
lab = UserLocAct(sel,:,:);
pred = pred(sel,:,:);
[X,Y,T,AUC] = perfcurve(lab(:),pred(:),1);

plot(X,Y);
```

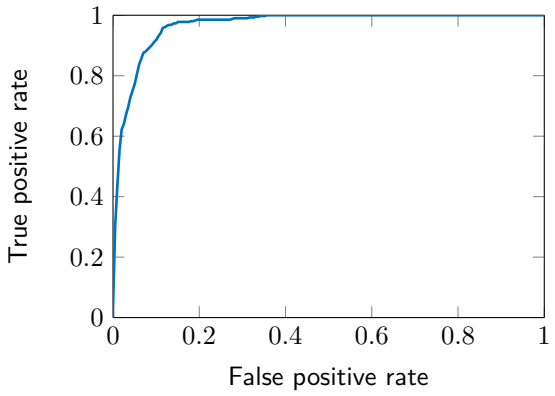

Fig. 7.4: Performance curves for the SDF model where 50 users are removed randomly. The AUC is 0.968.

7.5 Further explorations hints

The goal of this demo was to introduce some of the features of the modeling language used in the Structured Data Fusion framework. Many additions to these models can be explored, such as:

- changing the model for a dataset, e.g., change the model for the tensor to an LMLRA by defining an extra core tensor:

```
model.variables.s = randn(R,R,R);
model.factors.S = {'s'};
model.factorizations.ula.data  = UserLocAct_missing;
model.factorizations.ula.lmlra = {'U','L','A','S'};
```

- allowing extra rank-1 terms for some matrices

```
model.variables.u2 = randn(size(UserLocAct,1), 1);
model.factors.U2 = {'u', 'u2'};
model.UserUser.data = {'U2', 'U2', 'Duu'};
```

- using L0 or L1 regularization instead of L2;
- investigating the influence of the individual datasets;
- using proper validation of the models;
- changing constraints;
- changing the weights.

REFERENCES

[1] V. Zheng, B. Cao, Y. Zheng, X. Xie, and Q. Yang. Collaborative filtering meets mobile recommendation: a user-centered approach. In *AAAI*, volume 10, 236–241. 2010.

[2] L. Sorber, M. Van Barel, and L. De Lathauwer. Structured data fusion. *IEEE Journal of Selected Topics in Signal Processing*, 9(4):586–600, June 2015. 

[3] P. Comon and C. Jutten. *Handbook of Blind Source Separation: Independent component analysis and applications*. Academic press, 2010.

[4] Y. Li, T. Adali, W. Wang, and V. Calhoun. Joint blind source separation by multiset canonical correlation analysis. *IEEE Transactions on Signal Processing*, 57(10):3918–3929, 2009.

[5] A. Belouchrani, K. Abed-Meraim, J-F. Cardoso, and E. Moulines. A blind source separation technique using second-order statistics. *IEEE Transactions on Signal Processing*, 45(2):434–444, 1997.

[6] D-T. Pham and J-F. Cardoso. Blind separation of instantaneous mixtures of nonstationary sources. *IEEE Transactions on Signal Processing*, 49(9):1837–1848, 2001.