Assignments

Nhat-Nam Nguyen

1 CP decomposition with Alternating least squares algorithm

1.1 Overview

Canonical Polyadic Decomposition (CP) expresses a tensor as a sum of rank-one tensors $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$. CP-ALS iteratively refines factor matrices by solving alternating least squares(ALS) subproblems, minimizing the approximation error.

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

where $\mathbf{a}_r, \mathbf{b}_r, \mathbf{c}_r$ are vectors and $\circ$ represents the outer product and $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$ are factor matrices and λ_r are scaling weights for each rank-one component.
Now problem back to solving alternating least squares below:

$$\min_{\hat{\mathcal{X}}} \|\mathcal{X} - \hat{\mathcal{X}}\|$$

Based on Matricization refer to the combination of the vectors from the rank-one components, i.e., $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_R \end{bmatrix}$ and likewise for $\mathbf{B}$ and $\mathbf{C}$. We have a transformation:

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^\top, \\ \mathbf{X}_{(2)} &\approx \mathbf{B}(\mathbf{C} \odot \mathbf{A})^\top \\ \mathbf{X}_{(3)} &\approx \mathbf{C}(\mathbf{B} \odot \mathbf{A})^\top \end{aligned}$$

The ALS algorithm proceeds by alternately fixing two factor matrices and solving for the third in $\hat{\mathcal{X}}$. If we fix $\mathbf{B}$ and $\mathbf{C}$, we solve for $\mathbf{A}$ by minimizing the following least squares problem:

$$\|\mathcal{X}_{(1)} - \hat{\mathbf{A}}(\mathbf{C} \odot \mathbf{B})^\top\|_F$$

where $\mathcal{X}_{(1)}$ is the mode-1 matricization of the tensor, and $\odot$ denotes the Khatri-Rao product. The problem is equivalent to a linear least squares problem we can rewrite the solution as:

$$\hat{\mathbf{A}} = \mathbf{X}_{(1)} \left[(\mathbf{C} \odot \mathbf{B})^\top\right]^\dagger$$

where $\dagger$ denotes the Moore-Penrose pseudo-inverse. To reduce computational cost, we use the property of the Khatri-Rao product:

$$\left[(\mathbf{C} \odot \mathbf{B})^\top\right]^\dagger = (\mathbf{C} \odot \mathbf{B}) \left(\mathbf{C}^\top \mathbf{C} * \mathbf{B}^\top \mathbf{B}\right)^\dagger$$

Thus, the final update for $\mathbf{A}$ is:

$$\hat{\mathbf{A}} = \mathbf{X}_{(1)} (\mathbf{C} \odot \mathbf{B}) \left(\mathbf{C}^\top \mathbf{C} * \mathbf{B}^\top \mathbf{B}\right)^\dagger$$

Complete pseudocode of CP-ALS:

procedure CP-ALS($\mathcal{X}, R$)
  Initialize $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$ for $n = 1, \ldots, N$
  repeat
    for $n = 1, \ldots, N$ **do**
      $\mathbf{V} \leftarrow \mathbf{A}^{(1)\top}\mathbf{A}^{(1)} * \cdots * \mathbf{A}^{(n-1)\top}\mathbf{A}^{(n-1)}$
      $\mathbf{V} \leftarrow \mathbf{V} * \mathbf{A}^{(n+1)\top}\mathbf{A}^{(n+1)} * \cdots * \mathbf{A}^{(N)\top}\mathbf{A}^{(N)}$

$\mathbf{A}^{(n)} \leftarrow \mathbf{X}^{(n)} \left(\mathbf{A}^{(N)} \odot \ldots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \ldots \odot \mathbf{A}^{(1)}\right) \mathbf{V}^{\dagger}$

Normalize columns of $\mathbf{A}^{(n)}$ and store norms as $\boldsymbol{\lambda}$

end for

until fit ceases to improve or maximum iterations are exhausted

return $\boldsymbol{\lambda}, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}$

end procedure

Input R is the rank of the CP decomposition which we can specify as an input to determine the number of rank-one components used in the approximation in (1). And λ is the scaling coefficients and can be compute by $\lambda_r = \|\hat{\mathbf{a}}_r\|$ with $\hat{\mathbf{a}}_r$ is column r of $\hat{\mathbf{A}}$.

1.2 CP-ALS Algorithm Implementation

In this section, I will introduce four helper functions that will be useful in implementing the CP-ALS algorithm in MATLAB: `khatrirao_prod`, `ndim_unfold`, `reconstruct_tensor`, and `normalize_columns`.

1.2.1 Khatri-Rao Product

The `khatrirao_prod` function computes the Khatri-Rao product of two matrices. This operation is essential for CP-ALS as it computes a column-wise Kronecker product, used in the least squares updates of the factor matrices.

```
function KR = khatrirao_prod(A, B)
    % Computes the Khatri-Rao product of matrices A and B
    % A is of size I x R, B is of size J x R same number of columns
    [I, R1] = size(A);
    [J, R2] = size(B);
    if R1 ~= R2
        error('Matrices must have the same number of columns.');
    end
    KR = zeros(I * J, R1);
    for r = 1:R1
        KR(:, r) = kron(A(:, r), B(:, r));
    end
end
```

Func 1: Khatri-Rao Product

1.2.2 Tensor Unfolding (Matricization)

The `ndim_unfold` function performs the unfolding (or matricization) of a tensor along a specified mode. This is necessary for transforming a tensor into a matrix form so that we can apply matrix operations like least squares in the CP-ALS algorithm.

```
function Xn = ndim_unfold(X, mode)
    % Unfold tensor X along a specified mode (1, 2, or 3)
    switch mode
        case 1
            Xn = reshape(permute(X, [1 2 3]), size(X, 1), []);
        case 2
            Xn = reshape(permute(X, [2 1 3]), size(X, 2), []);
        case 3
            Xn = reshape(permute(X, [3 1 2]), size(X, 3), []);
        otherwise
            error('Invalid mode.');
    end
end
```

Func 2: Tensor Unfolding (Matricization)

1.2.3 Tensor Reconstruction

The `reconstruct_tensor` function reconstructs the tensor from the factor matrices A, B, and C, and the scaling vector λ. This is important in the CP-ALS algorithm to reconstruct the original tensor after factorization for validation or testing purposes.

```
function X_reconstructed = reconstruct_tensor(A, B, C, lambda)
    % Get the sizes of the dimensions
    I = size(A, 1);  % Number of rows in A
    J = size(B, 1);  % Number of rows in B
    K = size(C, 1);  % Number of rows in C
    R = length(lambda);  % Rank of the decomposition (length of lambda)

    % Initialize the reconstructed tensor
    X_reconstructed = zeros(I, J, K);

    % Accumulate contributions from each rank R
    for r = 1:R
        % Loop over each slice (third dimension of the tensor)
        for k = 1:K
            X_reconstructed(:,:,k) = X_reconstructed(:,:,k) + lambda(r) * (A(:,r) * B(:,r)')
   * C(k,r);
        end
    end

end
```

Func 3: reconstruct_tensor

1.2.4 Normalization of Factor Matrix Columns

The `normalize_columns` function normalizes the columns of a factor matrix, ensuring that each column has unit norm. This step is necessary in the CP-ALS algorithm to maintain the scaling properties of the factor matrices, with the scaling factors stored in λ.

```
function [A, lambda_r] = normalize_columns(A)
    % Normalize the columns of matrix A and store the norms in lambda
    % A: factor matrix to normalize
    % lambda_r: vector to store column norms

    [m, n] = size(A);
    lambda_r = zeros(1, n); % Initialize the lambda vector for storing norms

    % Loop over each column and compute the 2-norm using the built-in norm function
    for i = 1:n
        lambda_r(i) = norm(A(:, i), 2);  % Compute the 2-norm of the i-th column
        A(:, i) = A(:, i) / lambda_r(i); % Normalize the i-th column
    end
end
```

Func 4: Normalization of Factor Matrix Columns

1.2.5 Main Function: CP-ALS Algorithm

The `CP_ALS` function is the core of the Canonical Polyadic Decomposition using the Alternating Least Squares (ALS) method.

```
function [lambda, A,B,C] = CP_ALS(X, R, max_iter, tol)
    [I, J, K] = size(X);

    % Initialize A, B, C and lambda
    A = rand(I, R);
    B = rand(J, R);
    C = rand(K, R);
    lambda = ones(R, 1);

    N = ndims(X);
    prev_norm = inf;

    for iter = 1:max_iter
        % Update A
        V1 = (C' * C) .* (B' * B);
        X1 = ndim_unfold(X, 1);
        A_hat = X1 * khatrirao_prod(C, B) * pinv(V1);
        [A, lambda_A] = normalize_columns(A_hat);
```

```
        % Update B
        V2 = (C' * C) .* (A' * A);
        X2 = ndim_unfold(X, 2);
        B_hat = X2 * khatrirao_prod(C, A) * pinv(V2);
        [B, lambda_B] = normalize_columns(B_hat);

        % Update C
        V3 = (B' * B) .* (A' * A);
        X3 = ndim_unfold(X, 3);
        C_hat = X3 * khatrirao_prod(B, A) * pinv(V3);
        [C, lambda_C] = normalize_columns(C_hat);

        % Reconstruct the tensor and compute the difference norm
        X_reconstructed = reconstruct_tensor(A, B, C, lambda);
        Difference_norm = norm(X(:) - X_reconstructed(:));

        % Calculate the relative norm
        Relative_norm = Difference_norm / norm(X(:));
        fprintf('Iteration %d: Relative_norm = %.4f\n', iter, Relative_norm);

        % Check for convergence
        if abs(prev_norm - Relative_norm) < tol
            fprintf('Convergence achieved within tolerance at iteration %d.\n', iter);
            break;
        end

        % Update prev_norm and lambda for the next iteration
        prev_norm = Relative_norm;
        lambda = lambda_A;
    end
end
```

Func 5: CP-ALS Function

The function begins by initializing the factor matrices $A \in \mathbb{R}^{I \times R}$, $B \in \mathbb{R}^{J \times R}$, $C \in \mathbb{R}^{K \times R}$ randomly. The vector λ, which stores scaling factors for the rank-one tensors, is initialized to ones. The algorithm proceeds by iteratively updating each of the factor matrices A, B, and C while keeping the others fixed. This is achieved by solving alternating least squares problems. After that, the algorithm will update each factor matrix A, B, and C, and normalize them using the helper function `normalize_columns`. The scaling factors are stored in λ. The tensor is reconstructed using the updated factor matrices by calling the `reconstruct_tensor` function. The error between the original tensor X and the reconstructed tensor $X_{reconstructed}$ is computed using the Frobenius norm. The relative norm (error divided by the norm of the original tensor) is checked to determine if the change in error is below a specified tolerance level. If so, the algorithm terminates early, signaling convergence. If the algorithm does not converge within the specified tolerance, it will continue until the maximum number of iterations is reached.

1.3 Example of CP-ALS Implementation

In this example, we apply the `CP_ALS` function to a randomly generated tensor X of size $5 \times 4 \times 3$. We specify the rank $R = 3$ for the decomposition, set a tolerance level of 0.000001, and allow a maximum of 1000 iterations. The goal is to decompose the tensor into three factor matrices A, B, and C, along with the scaling vector λ, by minimizing the approximation error.

```
X = rand([5,4,3]);      % Randomly generated tensor
R = 3;                  % Rank of the decomposition
tol = 0.000001;         % Tolerance for convergence
max_iter = 1000;        % Maximum number of iterations

[lambda, A, B, C] = CP_ALS(X, R, max_iter, tol);  % Apply CP-ALS algorithm
```

Func 6: Example of CP-ALS

Output Log:

```
Iteration 1: Relative_norm = 0.7981
```

```
Iteration 2: Relative_norm = 0.4619
Iteration 3: Relative_norm = 0.3983
Iteration 4: Relative_norm = 0.3905
Iteration 5: Relative_norm = 0.3820
Iteration 6: Relative_norm = 0.3761
...
Iteration 133: Relative_norm = 0.2606
Iteration 134: Relative_norm = 0.2606
Iteration 135: Relative_norm = 0.2606
Convergence achieved within tolerance at iteration 135.
```

The initial relative error starts at 0.7981, reflecting the large difference between the initial random factorization and the original tensor. Over successive iterations, the relative error reduces as the factor matrices A, B, and C are updated, gradually improving the approximation of the original tensor. After 135 iterations, the algorithm reaches convergence, as the change in the relative error becomes smaller than the specified tolerance 0.000001. The final relative error is 0.2606, indicating a reasonable approximation of the original tensor.

2 Higher-Order Singular Value Decomposition (HOSVD) Algorithm

2.1 Overview

The HOSVD algorithm was introduced by Lathauwer, De Moor, and Vandewalle [**?**], and it is also known as the Tucker decomposition when truncated. HOSVD aims to decompose an N-way tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ into a core tensor $\mathcal{G}$ and a set of orthogonal matrices $\mathbf{A}^{(n)}$, where each matrix captures the principal components along one mode of the tensor. The main goal of HOSVD is to approximate the original tensor by capturing its most significant structure through singular values and vectors across all modes.

$$\boldsymbol{X} \approx \mathcal{G} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} = [\![\mathcal{G}; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$$

With $\mathbf{A} \in \mathbb{R}^{I \times P}, \mathbf{B} \in \mathbb{R}^{J \times Q}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$ are the factor matrices and core tensor $\mathcal{G} \in \mathbb{R}^{P \times Q \times R}$.
For each mode n ($n = 1, \ldots, N$), we begin by unfolding the tensor $\mathcal{X}$ into a matrix $\mathbf{X}_{(n)}$, where $\mathbf{X}_{(n)} \in \mathbb{R}^{I_n \times \prod_{k \neq n} I_k}$. Then, we perform singular value decomposition (SVD) on each mode-n unfolding $\mathbf{X}_{(n)}$ as follows:

$$\mathbf{X}_{(n)} = \mathbf{U}^{(n)} \mathbf{\Sigma}^{(n)} \mathbf{V}^{(n)T}$$

where $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times I_n}$ is an orthogonal matrix, and $\mathbf{\Sigma}^{(n)}$ is the diagonal matrix of singular values.
Once we have obtained the leading R_n left singular vectors and stored them in $\mathbf{A}^{(n)}$, we compute the core tensor $\mathcal{G}$ by multiplying the original tensor $\mathcal{X}$ with the transpose of each orthogonal matrix $\mathbf{A}^{(n)}$ along the corresponding mode n:

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

where $\times_n$ denotes the n-mode product of a tensor with a matrix.
Finally, the original tensor $\mathcal{X}$ can be reconstructed from the core tensor $\mathcal{G}$ and the factor matrices $\mathbf{A}^{(n)}$ by reversing the process:

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

2.2 HOSVD Algorithm Implementation

I will two additional helper functions: `ndim_fold` and `tprod`. These functions are using HOSVD.

2.2.1 Matrix folding

Function `ndim_fold` restores a matrix based on the unfolded previous specific mode, back into its original multidimensional tensor form. This is crucial when we want to reconstruct the original tensor after the unfolding and transformation steps in HOSVD.

```
function X = ndim_fold(Ai, n, sizeX)
    % NDIM_FOLD Restore a matrix into a multidimensional array with the given size
    %
    % X = NDIM_FOLD(Ai, n, sizeX)
    % Ai    - matrix to be folded
    % n     - dimension along which the tensor was unfolded
    % sizeX - size of the original tensor

    % Calculate total elements in the tensor
    num_elements = prod(sizeX);

    % Ensure the product of dimensions
    if prod(size(Ai)) ~= num_elements
        error('The product of dimensions does not match the number of elements in the original tensor.');
    end

    % Reshape Ai into the tensor of the correct size
    X = reshape(Ai, sizeX([n, 1:n-1, n+1:end]));

    % Permute to swap the dimensions back to their original order
    order = [2:n, 1, n+1:length(sizeX)];
    X = permute(X, order);
end
```

Func 7: ndim_fold: Fold a matrix into a tensor

2.2.2 Tensor product

The `tprod` function performs a series of matrix multiplications between a tensor and transformation matrices along each mode. This operation computing the core tensor $\mathcal{G}$ by multiplying the original tensor $\mathcal{X}$ with the left singular vectors for each mode.

```
function G = tprod(X, A)
    % TPROD Tensor product of a multidimensional array and a set of transformation matrices
    %   G = TPROD(X, A)
    %
    %   X  - tensor (multidimensional array)
    %   A  - cell array containing matrices for each dimension of X
    %
    %   G  - result of the product, effectively the core tensor G when A{n} contains
    %        the left singular vectors for each mode of the tensor X

    G = X;  % Initialize the resulting tensor G to be the input tensor X
    sizeX = size(X);

    for i = 1:length(A)
        if ~isempty(A{i})
            sizeX(i) = size(A{i},1); % Update the size of the n-th dimension as per the transformation matrix A{n}
            Xn = ndim_unfold(G, i); % Unfold tensor X along the n-th dimension
            G = ndim_fold(A{i}*Xn, i, sizeX); % Multiply the unfolded tensor by matrix A{n} and fold it back
        end
    end
end
```

Func 8: tprod: Tensor product with transformation matrices

2.3 Main Function: HOSVD Algorithm Implementation

Higher-Order Singular Value Decomposition (HOSVD) on a given input tensor $\mathcal{X}$. It uses the helper functions `ndim_unfold` and `tprod` that were introduced earlier to compute the core tensor $\mathcal{G}$ and the factor matrices $\mathbf{A}^{(n)}$.

```
function [G, A] = HOSVD(X, R)
% HOSVD Higher Order Singular Value Decomposition
%   [G, A] = HOSVD(X, R)
%
%   X   - Input tensor, a multidimensional array
```

```
%   R    - Rank in each mode specifying the number of singular values
%          and vectors to retain
%
%   G    - Core tensor such that X approximates tprod(G, A)
%   A    - Cell array of matrices containing the left singular vectors
%          for each dimension
%
%   Example:
%   [G, A] = HOSVD(rand(10,10,10), [3, 3, 3])

    M = size(X);
    P = length(M);  % Number of modes of the tensor X

    if nargin < 2
        R = ones(1,P);  % Default rank if not specified
    end

    A = cell(1,P);    % Cell array to store factor matrices A^{(n)}
    AT = cell(1,P);   % Cell array to store the transpose of A^{(n)}

    for i = 1:P
        if R(i)
            Xi = ndim_unfold(X, i);  % Unfold tensor X along mode i
            [U, ~] = svd(Xi, 'econ');  % Perform SVD on the unfolded tensor
            Ai = U(:, 1:R(i));  % Retain the first R(i) singular vectors
            A{i} = Ai;          % Store the singular vectors in A^{(n)}
            AT{i} = Ai';        % Store the transpose of A^{(n)}
        else
            A{i} = [];
            AT{i} = [];
        end
    end

    G = tprod(X, AT);  % Compute the core tensor G using tprod
end
```

Func 9: HOSVD: Higher Order Singular Value Decomposition

For each mode n, the tensor $\mathcal{X}$ is unfolded along that mode using the `ndim_unfold` function. Singular value decomposition (SVD) is applied to the unfolded matrix $\mathbf{X}_{(n)}$ to compute the leading singular vectors. These vectors are stored as the columns of $\mathbf{A}^{(n)}$, the factor matrix corresponding to mode n. After computing the factor matrices for all modes, the core tensor $\mathcal{G}$ is computed using the `tprod` function. This involves multiplying the original tensor $\mathcal{X}$ by the transposed factor matrices along each mode.

2.4 Example of HOSVD Implementation

In this example, we demonstrate the application of the `HOSVD` function on a randomly generated 3-order tensor. The rank of the decomposition is specified to be $[3, 3, 3]$ for each mode. The following MATLAB code showcases the procedure:

```
% Generate a random 3D tensor of size 10x10x10
X = rand(10,10,10);

% Apply the HOSVD algorithm with rank [3, 3, 3]
[G, A] = HOSVD(X, [3, 3, 3]);

% Ensure the tprod function and other related functions are correctly implemented
X_reconstructed = tprod(G, A);

% Calculate the norm of the difference between the original tensor and the reconstructed
    tensor
difference_norm = norm(X(:) - X_reconstructed(:));
Relative_norm = difference_norm / norm(X(:));

% Display the results using fprintf
fprintf('Difference norm: %f\n', difference_norm);
fprintf('Relative norm: %f\n', Relative_norm);
```

Func 10: Example of HOSVD Implementation

Output: The output below shows the difference between the original tensor $\mathcal{X}$ and the reconstructed tensor using the core tensor $\mathcal{G}$ and factor matrices $\mathbf{A}^{(n)}$. The difference is computed using the Frobenius norm, and the relative norm is the ratio of the difference norm to the norm of the original tensor.

```
>> test_HOSVD
Difference norm: 8.639821
Relative norm: 0.477368
```

3 Higher-order Orthogonal Iteration (HOOI) Algorithm

3.1 Overview

The Higher-Order Orthogonal Iteration (HOOI) can be seen as an improvement over the Higher-Order Singular Value Decomposition (HOSVD). Unlike HOSVD, which directly uses the leading singular vectors, HOOI iteratively refines the factor matrices to provide a better approximation of the original tensor. This is achieved using the Alternating Least Squares (ALS) technique. The primary advantage of HOOI is that it converges to a local minimum by iteratively updating each factor matrix while keeping the others fixed.
In HOOI, the objective is to approximate a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ as:

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

where $\mathcal{G}$ is the core tensor, and $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R_n}$ are the factor matrices for each mode. The goal of HOOI is to minimize the error in this approximation by iteratively refining the factor matrices.
The process begins by initializing the factor matrices $\mathbf{A}^{(n)}$ using the leading R_n left singular vectors from the HOSVD of the mode-n unfolding $\mathbf{X}_{(n)}$.
After initialization, the algorithm proceeds iteratively to update each factor matrix $\mathbf{A}^{(n)}$ by solving a linear least squares problem. The optimization problem solved by HOOI is:

$$\min \left\| \mathcal{X} - \mathcal{G} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)} \right\|$$

subject to the constraint that the factor matrices $\mathbf{A}^{(n)}$ are orthogonal for each mode n. The factor matrices $\mathbf{A}^{(n)}$ is updated iteratively until convergence is achieved, i.e., until the fit ceases to improve or a maximum number of iterations is reached.
For each matrix factor $\mathbf{A}^{(n)}$, the algorithm computes an intermediate tensor $\mathbf{Y}^{(n)}$ by fixing all other factor matrices and performing a multilinear product of the tensor $\mathcal{X}$ with the current estimates of the other factor matrices:

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

Once $\mathbf{Y}^{(n)}$ is computed, the matrix $\mathbf{A}^{(n)}$ is updated by solving a linear least squares problem using the mode-n unfolding of $\mathbf{Y}^{(n)}$ and extracting the leading R_n left singular vectors of the resulting matrix:

$$\mathbf{A}^{(n)} \leftarrow \text{leading } R_n \text{ left singular vectors of } \mathbf{Y}_{(n)}$$

This process is repeated iteratively for each mode n, updating one factor matrix at a time while keeping the others fixed.
The iterative process continues until the approximation error no longer improves or the maximum number of iterations is reached. The final result is an approximation of the tensor $\mathcal{X}$ with the core tensor $\mathcal{G}$ and the updated factor matrices $\mathbf{A}^{(n)}$.

3.2 HOOI Algorithm Implementation

The following MATLAB function implements the Higher-Order Orthogonal Iteration (HOOI) for tensor decomposition.

```
function [ G_final, A ] = HOOI( X, R ,max_iter, tol)
% Higher Order Orthogonal Iteration for Tensor Decomposition
% X        - input tensor
% R        - vector of target ranks for each mode
% max_iter- maximum number of iterations
% tol      - convergence tolerance

```

```
dims = size(X);
N = ndims(X);
A = cell(1, N);
G = X;
[~,A] = HOSVD( X, R ); % Initialize A using HOSVD

AT = cellfun(@(x) x', A, 'UniformOutput', false); % Transpose all A from the previous step
for iter = 1:max_iter % Repeat until convergence or maximum iterations
    for n = 1:N  % Iterate through each factor matrix A{n}
        AT_except_n = AT;
        AT_except_n{n} = eye(size(A{n}, 1)); % Replace n-th matrix with identity matrix
        Y = tprod(X, AT_except_n);  % Project tensor X onto the subspace defined by A except
    A{n}
        Yn = ndim_unfold(Y,n);  % Unfold Y with respect to the n-th mode
        [U, ~] = svdtrunc(Yn); % Compute the left singular vectors of Y(n)
        A{n} = U(:, 1:R(n)); % Keep Rn leading left singular vectors

        AT{n} = A{n}'; % Update the transpose of A{n}
    end

    % Reconstruct the core tensor using the updated A
    G_new = tprod(X, AT);
    X_reconstructed = tprod(G_new, A);
    Difference_norm = norm(X(:) - X_reconstructed(:));
    Relative_norm = Difference_norm/norm(X(:));
    fprintf('Iteration %d: Relative_norm = %.4f\n', iter, Relative_norm );

    if Relative_norm < tol
        fprintf('Convergence achieved within tolerance at iteration %d.\n', iter);
        break;
    elseif iter > 1 && abs(prev_norm - Relative_norm) < 1e-6  % Stop if improvement ceases
        fprintf('No significant improvement, stopping at iteration %d.\n', iter);
        break;
    end
    prev_norm = Relative_norm;  % Update the previous norm for the next iteration
end
G_final = G_new;
end
```

Func 11: HOOI: Higher Order Orthogonal Iteration

The algorithm begins by initializing the factor matrices $\mathbf{A}^{(n)}$ using the leading left singular vectors obtained from the Higher-Order Singular Value Decomposition (HOSVD). The initial core tensor $\mathcal{G}$ is then computed using the current factor matrices.
The algorithm iterates through the factor matrices $\mathbf{A}^{(n)}$ for each mode n. For each iteration, the current factor matrix $\mathbf{A}^{(n)}$ is updated by solving a linear least squares problem on the unfolded tensor $\mathbf{Y}^{(n)}$, which is obtained by projecting the tensor onto the subspace defined by the other factor matrices. The left singular vectors are computed using SVD, and the first R_n singular vectors are used to update $\mathbf{A}^{(n)}$.
After each iteration, the core tensor $\mathcal{G}$ is reconstructed using the updated factor matrices, and the approximation error (relative norm) between the original tensor and the reconstructed tensor is computed. If the error is smaller than the tolerance or no significant improvement occurs, the algorithm terminates.
The function returns the final core tensor G_{final} and the updated factor matrices A.
The algorithm stops when one of the following conditions is met:

- The relative approximation error falls below the specified tolerance (`tol`).
- The change in the relative error between iterations becomes insignificant (less than 10^{-6}).
- The maximum number of iterations (`max_iter`) is reached.

This implementation demonstrates the practical application of the HOOI algorithm for tensor decomposition, allowing for more accurate approximation of the input tensor through iterative refinement of the factor matrices.

3.3 Example HOOI Implementation

In this example, we demonstrate the application of the `HOOI` algorithm on a randomly generated 3th-order tensor. The rank for the decomposition is specified as $[2, 2, 2]$ for each mode, with a maximum of 1000 iterations and a convergence tolerance of 10^{-4}. The following MATLAB code illustrates the process:

```
% Generate a random 3D tensor of size 10x10x10
X = rand(10,10,10);

% Set the desired rank for each mode
R = [2, 2, 2];

% Set the maximum number of iterations and the tolerance for convergence
max_iter = 1000;
tol = 0.0001;

% Apply the HOOI algorithm to the tensor
[G_final, A] = HOOI(X, R, max_iter, tol);
```

Func 12: Example of HOOI Implementation

Output: The output below shows the relative norm of the difference between the original tensor $\mathcal{X}$ and the reconstructed tensor at each iteration. After 18 iterations, the algorithm stops because the improvement in the approximation error is no longer significant.

```
>> test_HOOI
Iteration 1: Relative_norm = 0.4894
Iteration 2: Relative_norm = 0.4877
Iteration 3: Relative_norm = 0.4870
Iteration 4: Relative_norm = 0.4866
Iteration 5: Relative_norm = 0.4865
..
Iteration 17: Relative_norm = 0.4855
Iteration 18: Relative_norm = 0.4855
No significant improvement, stopping at iteration 18.
```

The relative norm measures the approximation error between the original tensor $\mathcal{X}$ and the reconstructed tensor after each iteration. The error gradually decreases as the factor matrices $\mathbf{A}^{(n)}$ are refined. The algorithm stops at iteration 18 because the improvement in the relative norm becomes negligible, indicating that the solution has converged.

File Structure

The project consists of the following MATLAB files, each serving a specific purpose in the implementation of tensor decomposition methods, including CP-ALS, HOOI, and HOSVD. Below is the list of the files and a brief description of their functionality:

- `CP_ALS.m`: Implements the Canonical Polyadic Decomposition.
- `HOOI.m`: Contains the implementation of the Higher-Order Orthogonal Iteration (HOOI) algorithm for Tucker decomposition.
- `HOSVD.m`: Implements the Higher-Order Singular Value Decomposition (HOSVD) algorithm for tensor decomposition.
- `khatrirao_prod.m`: A helper function that computes the Khatri-Rao product.
- `ndim_fold.m`: A helper function that folds a matrix back into its original multidimensional tensor form after it has been unfolded along a specific mode.
- `ndim_unfold.m`: A helper function that unfolds a tensor along a specified mode into a matrix.
- `normalize_columns.m`: Normalizes the columns of a factor matrix and stores the norms as scaling factors.
- `reconstruct_tensor.m`: Reconstructs the original tensor from the factor matrices and the core tensor.
- `tprod.m`: Computes the tensor product of a multidimensional array with a set of transformation matrices for each mode.

- `test_CP_ALS.m`: A test script that runs the CP-ALS algorithm on a sample tensor and outputs the results.
- `test_HOOI.m`: A test script that applies the HOOI algorithm to a sample tensor and checks the convergence and approximation quality.
- `test_HOSVD.m`: A test script that applies the HOSVD algorithm to a sample tensor to evaluate the decomposition.