Tensor Decomposition: Fast CNN in your pocket

Kerem Turgutlu Jul 5, 2018 · 6 min read

Why do we even care about tensor decomposition ?

Even though it's an active area of research and there are many applications of tensor decomposition, such as signal processing, numerical algebra, computer vision, numerical analysis, machine learning, graph analysis and many more. We will mainly learn about a technique that will help us to train our image classifiers faster and which will compress our models so that they can fit devices in our pockets and run efficiently in them.

Speeding up the training and reducing the memory footprint is not only an interest for big tech companies who want to run AI as mobile services but also useful for practitioners with low resources. DAWNBench is a great example of what you can achieve with limited resources as long as you know the right techniques.

Back to Basics

Before getting started, this post will assume that the reader is already familiar with basic deep learning concepts, convolutional neural networks and basic linear algebra.

If you are already familiar with matrix decomposition then you can think tensor decomposition as a higher order matrix decomposition. If you are not familiar with it or need a refresher here we will briefly go over some of the basic ideas. In general we decompose matrices to solve systems of linear equations, to reduce the data dimension, to fill missing values, to find approximations for eigenvalues and for many other cool applications.

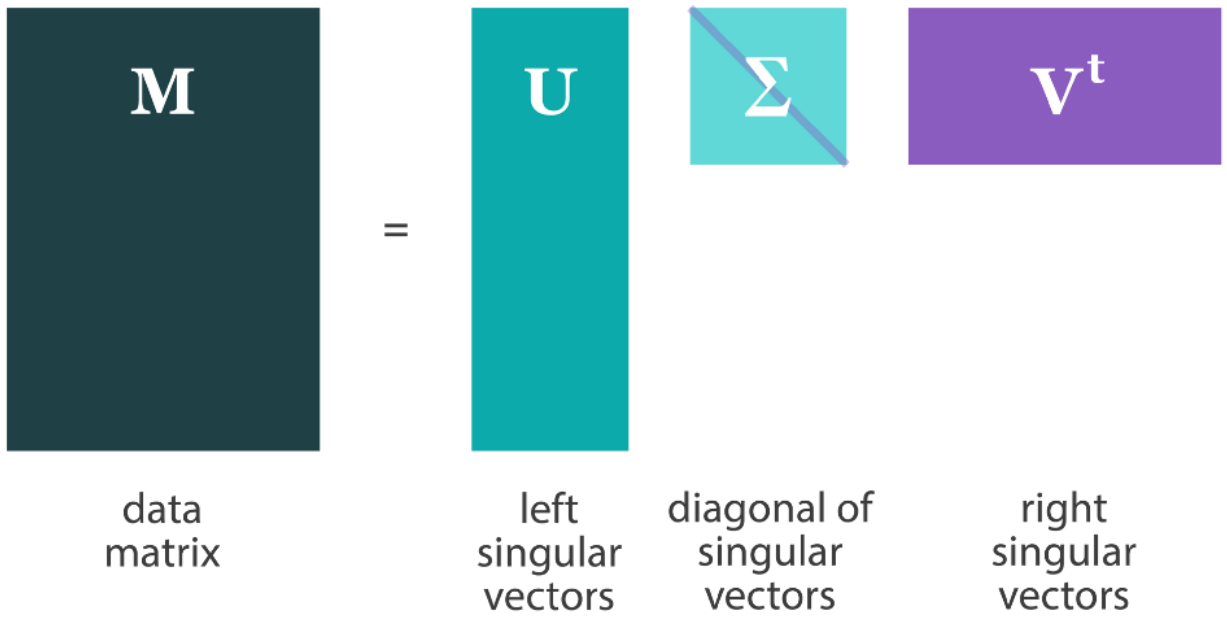
Some of the popular matrix decomposition techniques are:

SVD

Singular Value Decomposition (SVD) is a very fundamental algorithm to decompose any given matrix into **UΣV***. Here U and V* are orthogonal matrices.

It is the generalization of the eigendecomposition of a positive semidefinite normal matrix. We can use QR shift factorization to find singular values

and solve for SVD faster.



PCA

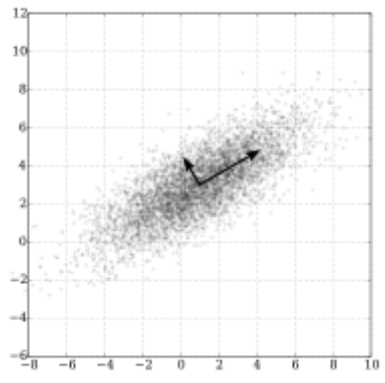
Principal component analysis is a very popular practical method used in statistics and data science and it can be thought as a by product of SVD. In SVD we decompose our matrix into orthogonal and diagonal components and PCA tries to identify orthogonal vectors which maximizes the variance explained in data.

Let X = UΣV* then principal components after ordering singular values can be written as:

X1 = u1σ1v1*

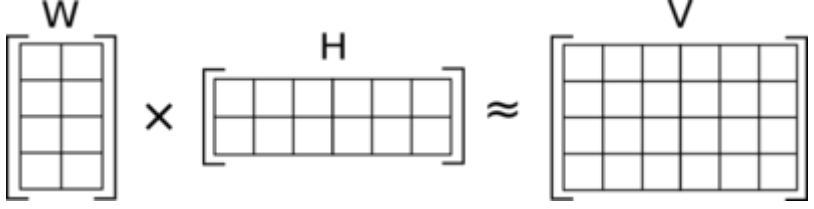
X2 = u2σ2v2*

…



NMF

Non-negative matrix factorization (NMF) is widely used in recommender systems and topic modelling. NMF will decompose a non-negative data matrix into two components containing non-negative values by minimizing the reconstruction error, e.g. Frobenius norm. In terms of optimization, constraining ourselves into non-negativity is not fundamental but it increases the interpretability of the results obtained by the factorization.



LU

LU decomposition uses Gaussian elimination to decompose a matrix into following two matrices: L for lower triangular and U for upper triangular.

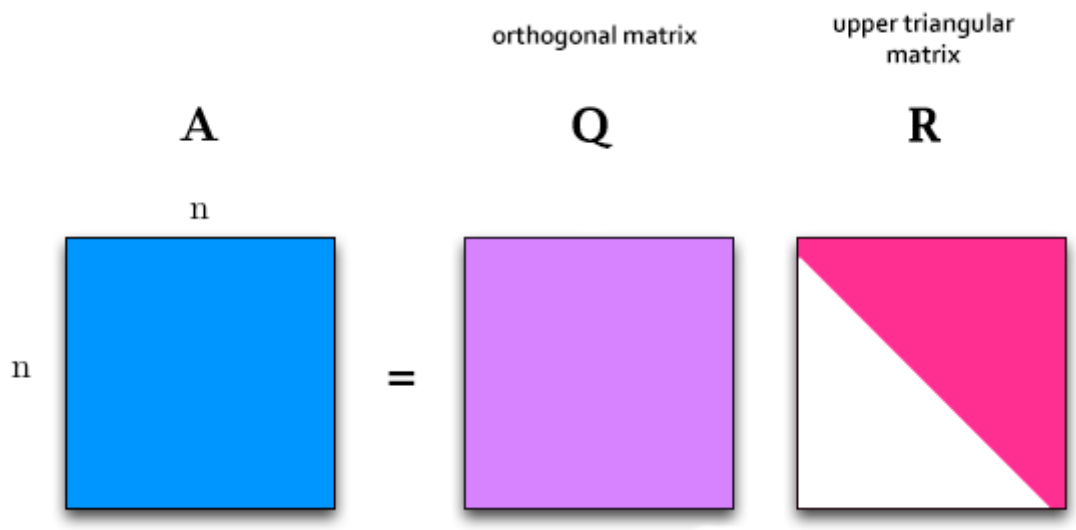
$$A = L \cdot U$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{pmatrix}$$

QR

QR is a very fundamental algorithm in linear algebra it is used to answer 3 out of 4 main problems that linear algebra tries to solve. It is used in **Least Squares Problem, Eigenvalue Problem** and **Finding SVD Problem.** For all these problems QR factorization is the optimal algorithm.

There are mainly three ways of computing QR decomposition, Gram-Schmidt process, householder reflections and Givens rotations. QR decomposes a matrix into orthogonal and upper triangular matrices.



CHOLESKY

We can use cholesky decomposition to solve for **Ax = b**, **Least Squares Problem** though still QR is more optimal compared to Cholesky. Cholesky factorization is a special case of LU where A is positive definite and algorithm is stable. It will decompose into a lower triangular matrix and conjugate transpose of that matrix.

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix}$$

$$\begin{pmatrix} L_{11}^2 & & \text{(symmetric)} \end{pmatrix}$$

$$= \begin{pmatrix} L_{11}L_{11} & & \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix},$$

Tensor Decomposition

Both CP (CANDECOMP/PARAFAC) and Tucker decompositions can be considered to be higher order generalizations of the matrix singular value decomposition (SVD) and principal component analysis (PCA).

Canonical Rank: Minimal possible R in CP decomposition.

In 2d low-rank approximation can be computed in a stable way by using SVD or if the matrix is large by rank revealing algorithms. This is not the case for the CP decomposition when d $>$ 2, as there is no finite algorithm for determining canonical rank of a tensor. Therefore, most algorithms approximate a tensor with different values of R until the approximation error becomes small enough.

Terminology

Here is some terminology about tensors.

NLS: Non-linear least square which minimizes L2-norm of the approximation residual for a user defined fixed R.

a: vector

A: matrix

X: tensor

fiber: Fixing every index but one in a tensor.

row: mode-2 fiber

column: mode-1 fiber

slice: 2 dimensional sections of a tensor

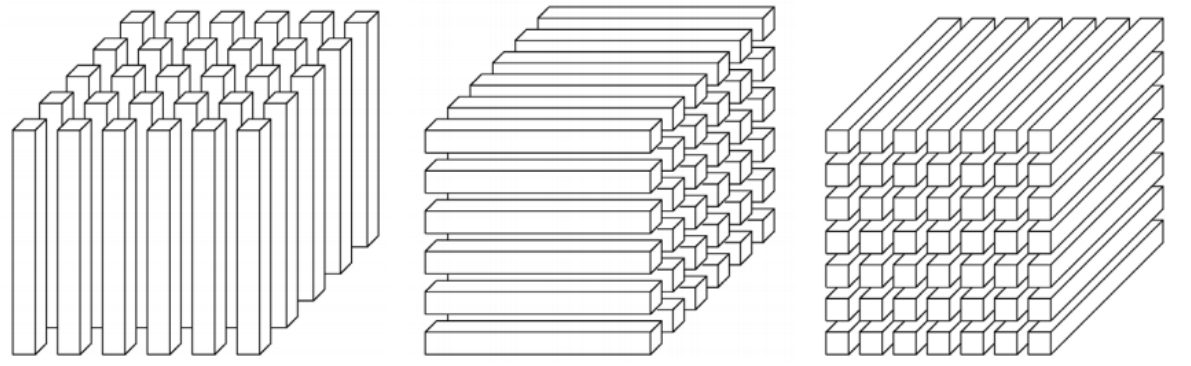
Third order tensors have column, row, and tube fibers denoted as x:jk, xi:k, xij:

horizontal view: Xi::

lateral view: X:j:

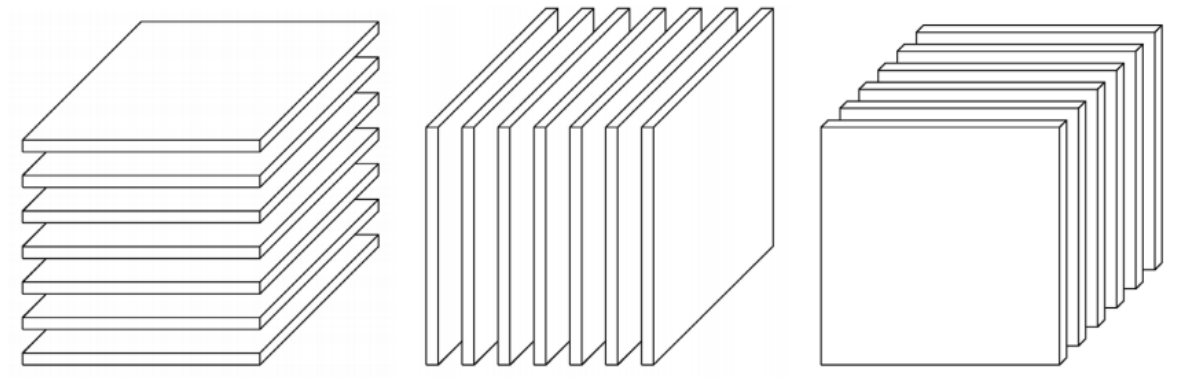
frontal view: X::k

||X|| = norm of tensor is square root of sum of square of all of it's elements



(a) Mode-1 (column) fibers: $\mathbf{x}_{:jk}$ (b) Mode-2 (row) fibers: $\mathbf{x}_{i:k}$ (c) Mode-3 (tube) fibers: $\mathbf{x}_{ij:}$

Fig. 2.1 *Fibers of a 3rd-order tensor.*



(a) Horizontal slices: $\mathbf{X}_{i::}$ (b) Lateral slices: $\mathbf{X}_{:j:}$ (c) Frontal slices: $\mathbf{X}_{::k}$ (or $\mathbf{X}_k$)

Fig. 2.2 *Slices of a 3rd-order tensor.*

Inner Product of two same sized tensors is the sum of the product of their entries:

```
import torch
import tensorly as tl # using pytorch >= 0.4 backend


A = torch.randint(0, 100, (3,3,3))
B = torch.randint(0, 100, (3,3,3))
tl.tenalg.inner(A, B) == torch.sum((A*B))
```

$<X, X> = X^2$

An N way tensor or array is **rank one** if it can be written as the outer product of N vectors:

X = a o b o c

o: denotes outer product

If a tensor has same size along every dimension (every mode is same size) then it's called **cubical.**

Matricization: Unfolding in a given dimension(s) to make a tensor a matrix.

Important Matrix Operations

Kronecker Product: Generalization of outer product from vectors to matrices.

$$\begin{bmatrix} 1 \cdot \begin{bmatrix} 0 & 5 \end{bmatrix} & 2 \cdot \begin{bmatrix} 0 & 5 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1 \cdot 0 & 1 \cdot 5 & 2 \cdot 0 & 2 \cdot 5 \end{bmatrix} = \begin{bmatrix} 0 & 5 & 0 & 10 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1 \cdot \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 2 \cdot \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \\ 3 \cdot \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 4 \cdot \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1 \cdot 6 & 1 \cdot 7 & 2 \cdot 6 & 2 \cdot 7 \\ 3 \cdot 0 & 3 \cdot 5 & 4 \cdot 0 & 4 \cdot 5 \\ 3 \cdot 6 & 3 \cdot 7 & 4 \cdot 6 & 4 \cdot 7 \end{bmatrix} = \begin{bmatrix} 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{bmatrix}.$$

Khatri-Rao: Kronecker product using columns of each matrix.

$$\mathbf{C} = \left[\begin{array}{c|c|c} \mathbf{C}_1 & \mathbf{C}_2 & \mathbf{C}_3 \end{array} \right] = \left[\begin{array}{c|c|c} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \right], \quad \mathbf{D} = \left[\begin{array}{c|c|c} \mathbf{D}_1 & \mathbf{D}_2 & \mathbf{D}_3 \end{array} \right] = \left[\begin{array}{c|c|c} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{array} \right],$$

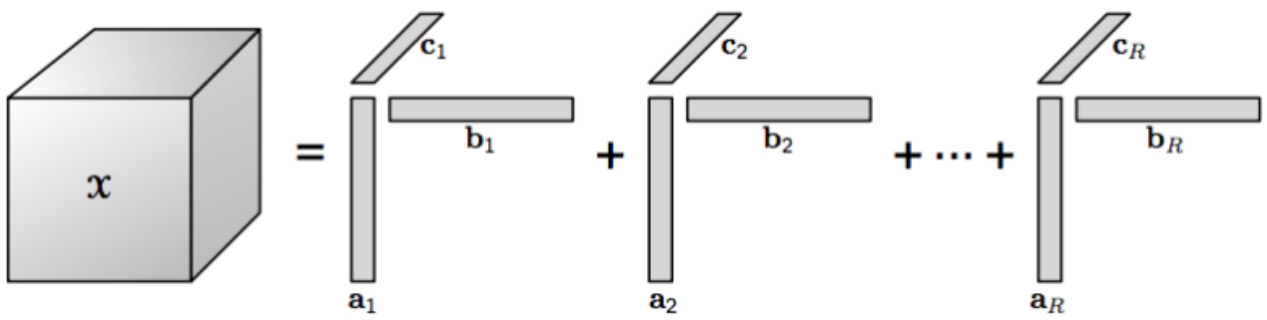
so that:

$$\mathbf{C} * \mathbf{D} = \left[\begin{array}{c|c|c} \mathbf{C}_1 \otimes \mathbf{D}_1 & \mathbf{C}_2 \otimes \mathbf{D}_2 & \mathbf{C}_3 \otimes \mathbf{D}_3 \end{array} \right] = \left[\begin{array}{c|c|c} 1 & 8 & 21 \\ 2 & 10 & 24 \\ 3 & 12 & 27 \\ 4 & 20 & 42 \\ 8 & 25 & 48 \\ 12 & 30 & 54 \\ 7 & 32 & 63 \\ 14 & 40 & 72 \\ 21 & 48 & 81 \end{array} \right].$$

Now, we will look into a particular tensor decomposition algorithm called CP Decomposition.

CP Decomposition

The basic idea behind CP decomposition is that you define a rank R, then sum of outer product of rank 1 tensors is your approximation. Still, approximating rank of a tensor is a hard problem:



`tensorly` is a great library when working with tensors and it provides off the shelf functions for tensor decomposition. In this library CP decomposition optimizes Frobenius norm of difference of **constructed tensor** and the **original tensor** using **ALS (Alternating Least Squares).**

Here is our objective function:

$$\min_{\mathbf{a}_i^k \in F^{m_k}} \|\mathcal{A} - \sum_{i=1}^{r} \mathbf{a}_i^1 \otimes \mathbf{a}_i^2 \otimes \cdots \otimes \mathbf{a}_i^d\|_F,$$

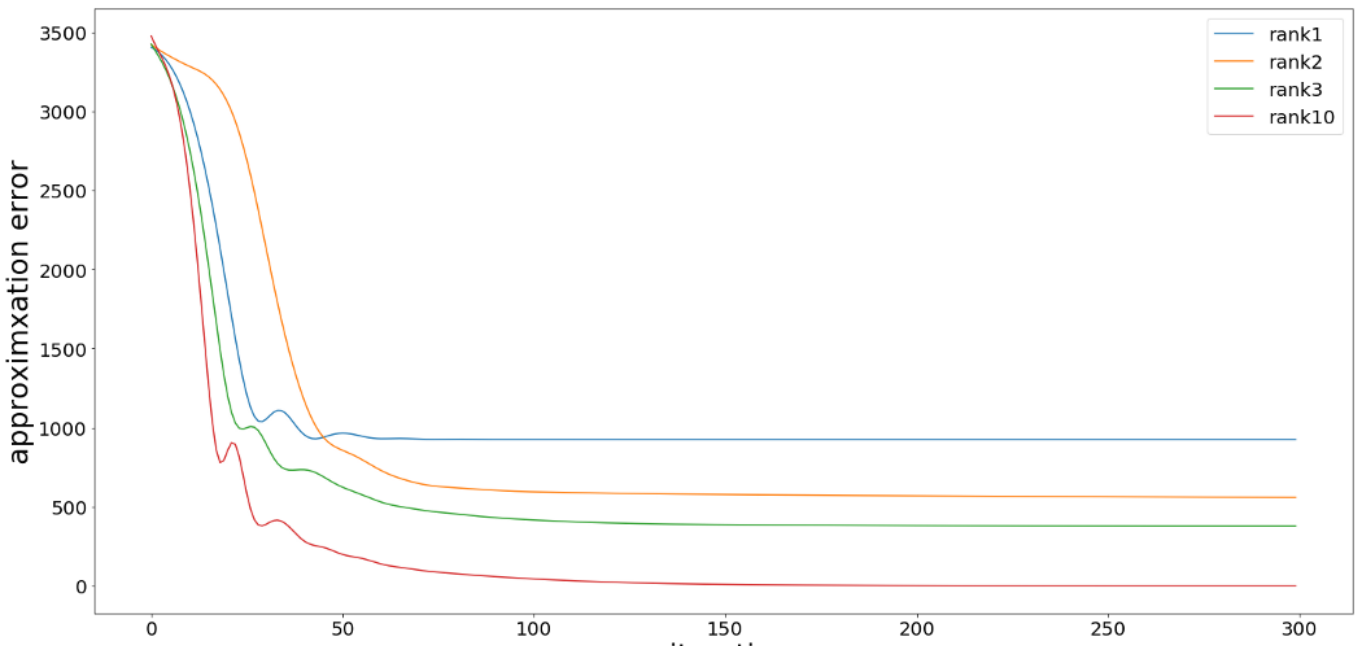
Minimizing Frobenius Norm

But we can also use PyTorch and Adam optimizer or any other optimizer to implement CP decomposition ourselves. Here is a code snippet for

decomposing a rank-3 tensor using CP decomposition via Adam optimizer, learning rate is adjusted by trial and error so it may change for your own case. I've also compared construction errors for different ranks between tensorly and my implementation, difference was negligible. So, we can say that below is a PyTorch version of the same optimization problem.

```
def construct(A,B,C):
    """
    Given Matrices A, B, C construct 3D Tensor
        A : i, r
        B: j, r
        C : k, r
    """
    X_tilde = 0
    r = A.shape[1]
    for i in range(r):
        X_tilde += torch.ger(A[:,i], B[:,i]).unsqueeze(2)*C[:,i].unsqueeze(0).unsqueeze(0)
    return X_tilde

def CP_decomposition(factors=[A,B,C], max_iter=10000, lr=0.1):
    """
    Minimize Frobenius Norm |X-X_tilde|
    Update decomposition factors
    """
    opt = Adam(factors, lr=lr)
    losses = []
    for i in range(max_iter):
        X_tilde = construct(*factors)

        opt.zero_grad()
        loss = torch.mean((X - X_tilde)**2)
        #print(loss)
        losses.append(loss.item())

        loss.backward(retain_graph=True)
        opt.step()
    return losses

r = 1
A = torch.randn((3,r), requires_grad=True)
B = torch.randn((4,r), requires_grad=True)
C = torch.randn((5,r), requires_grad=True)

rank1_loss = CP_decomposition([A,B,C])

```

Here is a plot of approximation errors for different rank decomposition. As you will notice as we increase the rank reconstruction error decreases, which is the expected behavior.



n iterations

In part 2 of this blog post, I will explain how we can use CP decomposition to decompose convolutional layers of a CNN and reduce the memory footprint of a image classifier like VGG, so stay tuned !

Thanks for reading the blog, hope you enjoyed it !

References:

- tensorly library: talk bit about tensorly too
- https://epubs.siam.org/doi/pdf/10.1137/07070111X
- https://arxiv.org/pdf/1412.6553.pdf
- http://www.dsrc.rpi.edu/presentations/Acar_RPI11.pdf
- https://arxiv.org/abs/1802.10410
- https://jacobgil.github.io/deeplearning/tensor-decompositions-deep-learning
- https://medium.com/@nicolas_19145/state-of-the-art-in-compressing-deep-convolutional-neural-networks-cfd8c5404f22
- https://arxiv.org/pdf/1412.6553.pdf

Machine Learning