Alma Mater Studiorum
University of Bologna

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria Informatica

Master Thesis

Compression of Convolutional Neural Networks with tensor decomposition

Author:
Ali Alessio Salman

Thesis Supervisor:
Prof. Stefano Mattoccia

Thesis Advisor
Ph.D Matteo Poggi

DISI
Dipartimento di Informatica - Scienza e Ingegneria

III Session
2017/2018

Abstract

Quest'elaborato verte sullo studio delle reti neurali artificiali, e si prefigge principalmente due scopi: il primo consiste nel comprendere il funzionamento che sta alla base delle reti neurali ed il loro apprendimento; il secondo sta nell'affrontare studio ed implementazione dei modelli più all'avanguardia delle Convolutional Neural Networks e saggiare la loro nota efficacia nei compiti di visione artificiale, in particolare nella classificazione.
Nel *Capitolo 1* viene fornita un'introduzione alle reti neurali, che si conclude con la proposta di un problema da risolvere con un percettrone multi-strato; nel *Capitolo 2* si percorre passo passo l'implementazione da zero di quest'ultimo, fino a portare a termine l'effettivo addestramento; il *Capitolo 3* introduce le Convolutional Neural Networks, ne spiega l'architettura ed i loro ultimi successi; nel *Capitolo 4* vengono implementate queste reti e testate su un due dataset diversi; il *Capitolo 5* presenta un'architettura allo stato dell'arte che viene confrontata con quella del capitolo precedente; nel *Capitolo 6* viene analizzato un caso d'uso industriale della classificazione, affrontato sfruttando il transfer-learning; il *Capitolo 7*, infine, espone le riflessioni sul lavoro svolto e conclude l'elaborato.

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

List of Abbreviations

SGD **S**tochastic **G**radient **D**escent

List of Symbols

a	distance	m
P	power	W ($\mathrm{J\,s^{-1}}$)
ω	angular frequency	rad

For/Dedicated to/To my...

Chapter 1

4 Tensor Decomposition

In this chapter the main mathematical tools for the manipulation of tensors are introduced. Following, we will see how to apply these operators to decompose a convolutional layer through two different techniques, effectively exploring the state-of-the-art methods of low-rank approximation presented in Chapter 2.

1.1 Background

A tensor is a geometric object that generalizes all the structures usually defined in linear algebra to the n-dimensional space. As such, they can be defined as n-dimensional arrays.

In fact, recalling that a vectors basis is a set of linearly independent vectors with which we can represent every other vector in the correspondent vector space, a general vector v can be defined as n-dimensional array of length n. In the same way, we can define *order* (or way) of a tensor the dimensionality of the array needed to represent it with respect to this basis, or the number of indices needed to label a component of that array.

Thus, an *k-th* order tensor in an n-dimensional space is a mathematical object that has n indices and n^k components; each index ranges over the number of dimensions of the space.

A third-order tensor is showed in 1.1.

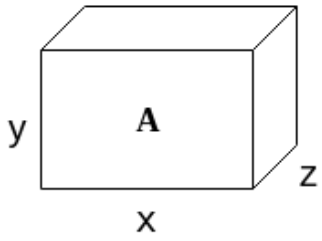

Figura 1.1: A third order tensor.

1.1.1 Tensor rank

Intuitevely, a scalar would be a tensor of *order 0*; a vector of *order 1*; a matric of *order 2* and so on.

The intuitive definition can also be used using "*rank*" instead of order, but that may be somewhat misleading since there is a subtle difference. To avoid confusion, the following definitions are introduced:

- a tensor of *rank*-1 (or a decomposable tensor [1]) is a tensor that can be written as a product of tensors of the form:

$$T = a \circ b \circ \ldots \circ d \tag{1.1}$$

- The *rank* of a tensor X is the minimum number of rank-1 tensor that sum to X.

The product used in 1.1 is an outer product for vectors and will be defined in details in the coming section.

1.1.2 Tensor fibers & slices

Tensor fibers are very intuitive to represent visually but not as easy to explain.

Let $A(i, j, k)$ be a third-order tensor. The mode-1 fiber of index s, t of $A(i, j, k)$ contains all the elements of A, which have the $j = s$ and $k = t$.

Intuitively, the i-fiber is the "column" that is obtained by fixing one index. Conversely, a tensor *slice* is instead obtained by fixing two indeces. It becomes immediately clear when observing figure 1.2. The three different ways in which it is possible to orient a fiber are called *tensor modes*. They are very important to define some basic tensor operation.

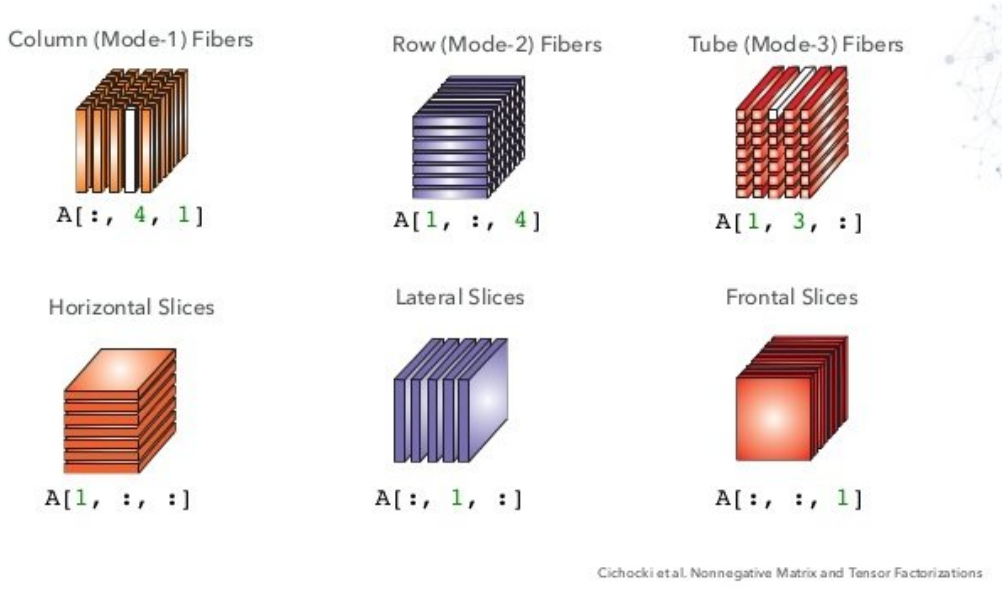

Figura 1.2: Fibers and slices of a tensor according to each mode.

To make things even clearer, here is a snippet code to index fibers and slices in Python:

```
In [13]: tensor = numpy.arange(24).reshape(2,3,4)
In [14]: tensor
Out[14]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])


In [18]: tensor[:,0,0]
Out[18]: array([ 0, 12])  #Fiber

In [16]: tensor[0,:,:]
Out[16]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]]) #Slice

In [17]:
```

Tensors are extensively used in many applications [2] to modelize multi-dimensional data. In the CNN scenario, a CONV layer with $\mathcal{W}$ weights is defined through a tensor of size:

$$dim(\mathcal{W}) = [T \times S \times D \times D] \tag{1.2}$$

where,

- T is the number of output filters
- D is the size of the kernel of the convolution
- S is the number of input filters

In pure mathematical terms, there exist a lot of different methods to decompose a tensor. In section 1.2 the formal tools to wield tensors are given and in section 1.3 the application of these methods to convolutional layers are presented.

1.1.3 Singular value decomposition

In order to understand better how a tensor decomposition work and its properties, it is necessary to introduce the *singular value decomposition* (SVD) for matrices.

Let M be a matrix $\in \mathbf{F}$ of size $m \times n$, the SVD is given by:

$$M = U\Sigma V^* \tag{1.3}$$

Where U and V are an $m \times m$ and $n \times n$ unitary matrix respectively. In the case of $\mathbf{F} = \mathbf{R}$, U and V are also orthogonal matrices. V is the conjugate transpose of V. Σ is a *diagonal* matrix with non-negative real numbers, which holds the singular values of M in its diagonal.

A thorough explanation of the above terms is required, so the following definitions must be kept in mind:

- A *diagonal* matrix is a matrix whose values are all zero except for those on the diagonal. A special case of diagonal matrix is an *identity* matrix, where all these diagonal elements are equal to 1.
- Given a matrix M, the *transpose* is an operator that flips M over its diagonal producing another matrix M^T as a result, whose column and rows indices are therefore also switched. Hence, the rows of M becomes the columns of M^T and viceversa.
- Given a matrix $M \in \mathbf{C}$, its *complex conjugate*, $\bar{M}$, is the conversion of each element $m_{i,j}$ to its conjugate i.e., the real part are the same while the imaginary part have opposite sign and same magnitude.
- A *conjugate transpose* is a matrix M^* who's been obtained by first transposing M and then taking the *complex conjugate* of each entry. Also, the following properties holds:
$$M^* = (\bar{M})^T = M \bar{*} T$$
- A quadratic matrix M is said to be *unitary* if $MM^* = M * M = I$, where I is the identity matrix. In the case $M \in \mathbb{R}$ the matrix is called *orthogonal* and it satisfies the equivalence $MM^T = M^T M = I$.
- An *orthogonal* matrix has rows and columns that are unitary or orthogonal between each other, respectively.
- A non-negative real number σ is a singular value for M of a space $F^{m \times n}$ if and only if there exist unit-length vectors $u \in \mathbf{F}^m$, $v \in \mathbf{F}^n$ such that $Mv = \sigma u$ and $M^* u = \sigma v$. The vectors u and v are called left-singular and right-singular vectors for σ respectively.

Recalling the SVD definition 1.3, the $\times n$ rectangular matrix Σ holds the *singular values* (the square roots of the non-zero *eigen-values*) σ_i $i = 1, \ldots, k$ of M on its diagonal. The first $k = min(m, n)$ columns of U and V are, respectively, left-singular vectors and right-singular vectors for the corresponding singular values. Consequently, the SVD theorem implies that:

- An $m \times n$ matrix M has at most k distinct singular values;
- It is always possible to find a unitary basis U for $\mathbf{F}^m$ with a subset of basis vectors spanning the left-singular vectors of each singular value of M;
- It is always possible to find a unitary basis V for $\mathbf{F}^n$ with a subset of basis vectors spanning the right-singular vectors of each singular value of M.

1.1.4 SVD Applications

The SVD factorization is useful in many fields of research. It can be used to solve the *linear least-squares* and the *total least-squares* problems; it is widely used in statistics where it is related to *principal component analysis* (PCA); it's successfully used in signal processing and pattern recognition. SVD is also fundamental in *recommender systems* to predict people's item ratings [3] [4]; for instance, Netflix has launched a global competition to find the best implementation of SVD for clusters to improve its collaborative filtering technique [5]. The main area of interest of SVD for convolutional neural networks is that of *low-rank matrix approximation* and image processing.

In fact, SVD can be thought of as decomposing a matrix into a *weighted, ordered* sum of separable matrices. Separable here means exactly the same concept introduced for tensors in section 1.1.1 i.e., a matrix A can be written as an outer product of two vectors $A = u \circ v$. More precisely, the matrix can be factorized as:

$$M = \sum_i A_i = \sum_i \sigma_i U_i \circ V_i^T \tag{1.4}$$

this turns out to be enable a fast convolution computation:

$$F = \sum_i^R \sigma_i (I * U_i) * V_i \tag{1.5}$$

where I is the input image, F is the resultant feature map and U_i and V_i are i-th column of U and V respectively.

Note that σ_i are the R largest singular values (the other singular values are replaced by zero). Since the number of non-zero σ_i correspond exactly to the rank of a matrix, the approximated matrix is thus of rank R.
For this purpose, given a matrix M, the best method to find a truncated matrix $\tilde{M}$ of rank R that approximates M, is to find the solution that minimizes the *frobenius norm* of their difference:

$$\left\| M - \hat{M} \right\|, \quad with\ \hat{M} = \sum_i U_i \circ V_i \tag{1.6}$$

The Eckart-Young theorem [6] states that the best solution to the problem is given by its SVD decomposition.

SVD on CNN

Given its application to speed up convolution filters, it is not surprising that SVD has been one of the methods used in recent developments on deep CNN compression, as portrayed in chapter **??**. In particular, the work of Cheng et al. [7] applies an original decomposition scheme which is based on top of SVD.

Given a conv layer with weights $\mathcal{W} = [T \times S \times d \times d]$, it consists in constructing a bigger matrix $\mathcal{B}$ built with the stack of feature maps, namely spreading the input channels kernels on rows and kernel composing the filter bank on column. Then, it computes the SVD of the so obtained matrix and ...

SVD on fully-connected layers

Beside convolution, there is another way to leverage on SVD in convnets and that is the fully-connected layers. In practice, fully-connected layers are just plain matrix multiplication plus a bias:

$$Y = W * X + B \tag{1.7}$$

where X are the input maps and W is an $m \times n$ matrix that holds the weights of the layer.

From this observation, a recipe to speed up fully-connected layers can be spotted:

1. compute the truncated SVD of the weight matrix W keeping only the first r singular values;

2. substitute the layer with 2 smaller fully-connected ones;

3. the first one will have a shape of $[m \times r]$ and no bias;

4. the second will have a shape of $[rn]$ and a bias equal to its original bias B.

The SVD decomposition of the layer is formalized as follow:

$$(U_{m\times r}\Sigma_{r\times r}V^T_{n\times t})X + B = U_{m\times r}(\Sigma_{r\times r}V^T_{n\times r}X) + B \tag{1.8}$$

This way, the total number of weights dropped from $n \times m$ to $r(n + m)$, which is dependent on the choice of the rank of the approximation r and can be significant when r is a much smaller than $min(n, m)$.
The first reference of this approach could be find in the *Fast R-CNN* paper fast-rcnn, an efficient implementation of RCNN [8], a very famous CNN built for region proposal and object detection. The implementation for Caffe is also available on the author github page, at [9].

1.2 Tensor mathematical tools

1.2.1 Basic operations

In multi-linear algebra it actually does not exist a method that satisfy all SVD properties for *m-way arrays*, i.e. tensors. However, by taking a closer look at SVD we can formulate two main requirements that a tensor decomposition algorithm should satisfy to be a feasible alternative to SVD in a tensor-world:

1. a Rank-R decomposition

2. the orthonormal row/column matrices.

SVD computes both of them simultaneously. Regarding tensors, these properties can be captured separately by two different family of decompositions.

The first property is extended to the multi-linear world by a class of decompositions that fall under the name of *CP decomposition* (named after the two most popular variants, CANDECOMP and PARAFAC). The latter is provided by the Tucker methods (and many other names). For each axes of a tensor, these methods compute the associated orthonormal space. Therefore, Tucker methods are also used in multilinear principal component analysis (PCA).

Historically, much of the interest in higher-order SVDs was driven by the need to analize empirical data, expecially in psychometrics, chemometrics and neuroscience [10]. As such, these techniques have been rediscovered many times with different names leading to a confused literature. Thus, these methods are often presented in a more practical goal-driven way, rather than through rigourous abstract general theorems, which are in fact rare.

Before diving into tensor decomposition algorithms, we need to introduce some fundamental tensor operations:

- **Tensor unfolding**: Tensor unfolding, also called *matrization*, is an operation that maps a tensor X into a matrix M. It is done by taking each mode-i fiber of a tensor and lay them down as columns in the resulting matrix. The other modes are handled cyclically. The specified mode will be the first mode of the matrix i.e., given X of size $I \times J \times K$, the first-mode unfolding will produce a matrix M of size $I \times JK$.

A simpler way to look at it is by taking the mode-i slices and put them one after the other. An example for the a third-order tensor is illustrated in 1.2.1.

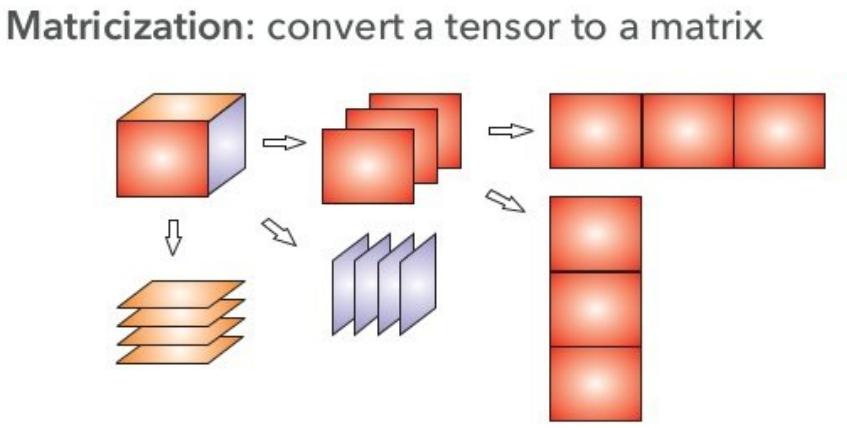

Figura 1.3: Tensor unfolding or matricization along different modes.

- **Tensor times matrix: *k*-mode product:** The k-mode product of a tensor $\mathrm{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ with a matrix $M \in \mathbb{R}^{J \times I_k}$ is written as:

$$\mathbf{Y} = \mathbf{X} \times_k M \tag{1.9}$$

The resulting tensor Y is of size $I_1 \times \ldots \times I_{k-1} \times J \times I_{k+1} \times \ldots \times I_N$, and contains the elements:

$$y_{i_1 \cdots i_{k-1} j i_{k+1} \cdots i_N} = \sum_{i_k=1}^{I_k} x_{i_1 i_2 \cdots i_N} a_{j i_k}.$$

It can be hard to visualize this operation at first, but effectively it boils down to multiply each mode-k fiber of X by the matrix M. Looking at 1.2.1, we can also represent the same operation with $\mathbf{Y}_{(i)} = \mathbf{X}_{(i)} \cdot M$, being $\mathbf{X}_{(i)}$ the mode-i unfolding of the tensor X.

To simplify things, let $\mathbf{X}$ be a 3-mode tensor $\in F^{I \times J \times K}$ and M a matrix $\in F^{N \times J}$, the k-mode product on axis 1 of X and M is:

$$\mathbf{Y} = \mathbf{X} \times_1 M, Y \in F^{I \times J \times K} \tag{1.10}$$

So each element (n, j, k) of Y is obtained by:

$$y_{n,j,k} = \sum_i x_{i,j,k} \cdot b_{n,j} \tag{1.11}$$

A visual example is depicted in 1.2.1.

Few interesting properties of the k-mode product are:

- $X \times_m A \times_n B = S \times_n B \times_m A$ if $n \neq m$
- $X \times_n A \times_n B = X \times_n (BA) \neq X \times_n B \times_n A$.

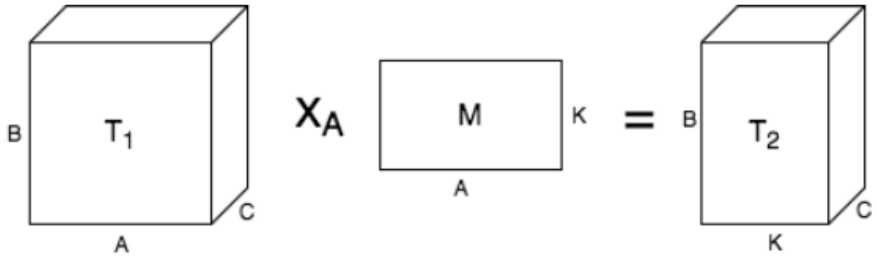

Figura 1.4: An example of a k-mode product i.e., a tensor-matrix multiplication of a 3-dimensional tensor.

- **Tensor times vector**: Given the same matrix X, the tensor-vector multiplication on the i-axis is defined as:

$$\mathbf{Y} = \mathbf{X} \times_1 v, \mathbf{Y} \in \tag{1.12}$$

with each element $y_{j,k}$:

$$y_{j,k} = \sum x_{i,j,k} \cdot a_i \tag{1.13}$$

An example is illustrated in 1.2.1.

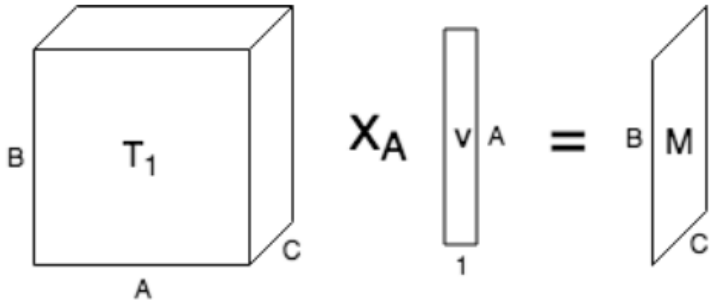

Figura 1.5: An example of a tensor-vector product.

- **Matrix Kronecker product**: A Kronecker product of two matrices $A \in \mathbf{R}^{M \times N}$ and $B \in \mathbf{R}^{P \times Q}$ is defined as:

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix} \tag{1.14}$$

and more explicitly:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}$$

- **Outer product**: If we take the *Kronecker product for matrices* definition and apply it to vectors, we obtain the outer product:

$$\mathbf{a} \circ \mathbf{b} = \mathbf{a}\mathbf{b}^{\mathrm{T}} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} = \begin{bmatrix} a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_1 & a_3b_2 & a_3b_3 \\ a_4b_1 & a_4b_2 & a_4b_3 \end{bmatrix}. \tag{1.15}$$

Let $a \in \mathbf{R}^I$, $b \in \mathbf{R}^J$ $c \in \mathbf{R}^K$ be three vectors. Computing the outer product $(a \circ b)$ of two of them will result in a matrix, as showed above. Proceeding in this way is easy to show that an outer product of 3-vectors will result in a 3-dimensional tensor, as illustrated in 1.6.

This comes in handy the other way around: a rank-1 tensor can be decomposed into 3 vectors. As we will see in the following sections, this operation is fundamental for tensor decomposition.
Another interesting way to look at it is that the outer product operation " $\circ$ " is a way of combining a tensor of $d1$-order and a tensor of $d2$-order to obtain a tensor of order-$(d1 + d2)$.

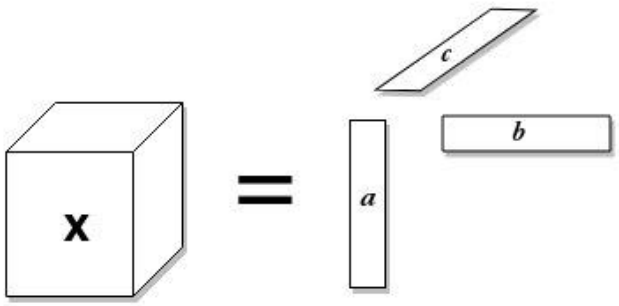

Figura 1.6: Representation of third-order tensor with an outer product of vectors.

- **Matrix Khatri-Rao product**: given two matrices $A \in \mathbf{R}^{M \times N}$ and $B \in \mathbf{R}^{P \times R}$ is defined as:

$$A \odot B = [a_1 \otimes b_1, a_2 \otimes b_2, \ldots, a_N \otimes b_R] \in \mathbf{R}^{MN \times R} \tag{1.16}$$

  Note that the Kronecker matrix operation returns the same number of elements of the Khatri-Rao product, but while the former produce a matrix, the latter is shaped into a vector.

1.2.2 Tucker Decomposition

The Tucker decomposition is a way to write a tensor of size $I_1 \times I_2 \times \ldots \times I_N$ as the multi-linear tensor-matrix product of a *core* tensor $\mathbf{G}$ of size $R_1 \times R_2 \times \cdots \times R_N$ with N factors $A^{(n)}$ of size $I_n \times R_n$:

$$\mathrm{X} = \mathbf{G} \times_1 A^{(1)} \times_2 A^{(2)} \times_3 \ldots \times_N A^{(N)} \tag{1.17}$$

where $\times_k$ is the k-mode product introduced before.

To make this definition a little bit more intuitive, let X be a third-order tensor $\in \mathbf{R} * I \times J \times K$), then the Tucker decomposition is defined as :

$$X = \mathbf{G} \times_1 A \times_2 B \times_3 C = \sum_r \sum_s \sum_t \sigma_{r,s,t} u_r \circ v_s \circ w_t \tag{1.18}$$

where A is a $I \times R$ matrix, B is a $J \times S$ matrix, C is a $K \times T$ matrix and $\mathbf{G}$ is a $R \times S \times T$-tensor with $R < I$, $S < J$ and $T < K$.

An example of third-order tensor Tucker decomposition is depicted in figure 1.7.

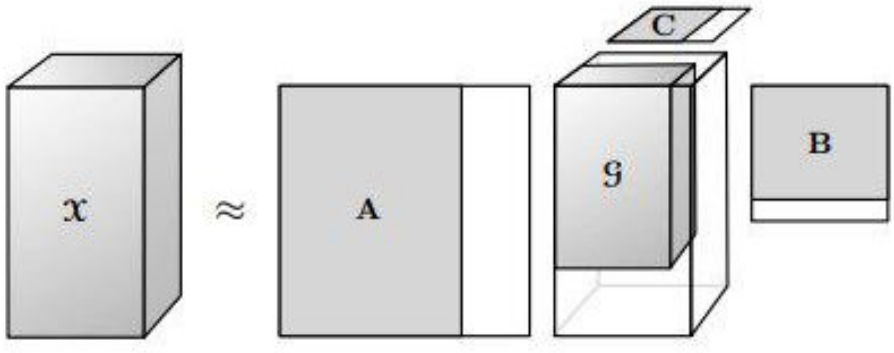

Figura 1.7: A Tucker decomposition of a three-modes tensor

A matrix representation of the Tucker tensor is given by the following equations:

$$X_{(1)} = A\mathbf{G}_{(1)}(C \otimes B)^T \tag{1.19}$$

$$X_{(2)} = B\mathbf{G}_{(2)}(C \otimes A)^T \tag{1.20}$$

$$X_{(3)} = A\mathbf{G}_{(3)}(B \otimes A)^T \tag{1.21}$$

and it is also possible to unfold it into a vector:

$$vec(X) = (C \otimes B \otimes A)vec(\mathbf{G}) \tag{1.22}$$

It is worth noting that we can see a Tucker decomposition as an SVD in each dimension, being $\mathcal{G}$ the singular value tensor and A, B, C the matrices with the singular vectors of each dimension.

We have already seen how SVD can be useful in reducing complexity in fully-connected layers 1.1.4; therefore it's legitimate to presume that the Tucker decomposition can have a meaningful impact on the filter maps in convolutional layers.

Moreover, from chapter 3 we can recall that convolutional filters expose a *parameter sharing* design. Hence it is intuibile that those filters will have some related parameters and thus feasible to a good factorization.

HO-SVD

As the *Higher-order singular value decomposition* was studied in many scientific fields, it is historically referred in different ways: multilinear singular value decomposition, m-mode SVD, or cube SVD, and it is often incorrectly identified with a Tucker decomposition.

HOSVD is actually a specific orthogonal version of the Tucker decomposition. To put it in other words, it is a specialized algorithm to compute the Tucker decomposition. HOSVD involves solving each k-mode matricized form of the specific tensor [11], relying on the following equivalence:

$$\begin{aligned} & Y &=& X \times_1 A^{(1)} \times_2 A^{(2)} \times_3 \ldots \times_N A^{(N)} \\ \Leftrightarrow \quad & Y_{(k)} &=& A^{(k)} X_{(k)} \left(A^{(N)} \otimes \cdots \otimes A^{(k+1)} \otimes A^{(k-1)} \otimes \cdots A^{(1)} \right)^T . \end{aligned}$$

The algorithm steps follow:

for $k = 1, 2, ..., N$ do
  $A^{(k)} \leftarrow$ left orthogonal matrix of SVD of $X_{(k)}$
end for
$G \leftarrow X \times_1 (A^{(1)})^T \times_2 (A^{(2)})^T \times_3 \cdots \times_N (A^{(N)})^T$

This approach may be regarded as one generalization of the matrix SVD, because:

- each matrix A^k is an ortoghonal matrix
- Two subtensors of the core tensor $\mathbf{G}$ are orthogonal, i.e. $\langle \mathbf{G}_p, \mathbf{G}_q \rangle \quad if \quad p \neq q$
- the subtensors in the core tensor $\mathbf{G}$ are ordered according to their Frobenius norm, i.e. $\|\mathbf{G}_1\| \geq \|\mathbf{G}_2\| \geq \ldots \geq \|\mathbf{G}_n\|_{\,for\, n=1,\ldots,N}$

A nice visualization of HOSVD is given in figure 1.8. The tensor $\mathbf{G}$ is said to be *"ordered"* and *"all-ortoghonal"*. For further explanation see [12].

Higher order orthogonal iteration

As for SVD, it is important to notice that HOSVD can be used to compress X by truncating the matrices $A(k)$. The problem with respect to matrices though, is that a truncated HOSVD is known not to give the best fit. However, as a logical consequence of two concepts that have been introduced earlier, an iterative algorithm to find the best solution can be proposed.
In fact, putting together the Eckart-Young theorem for the *Frobenius* norm and the HOSVD depicted above will result in an iterative optimization known as Higher-Order Orthogonal Iteration or *HOOI*.

HOOI finds the optimal approximation $\hat{X}$ with respect to the Frobenius norom loss by, essentially, iterating the alternating truncation and SVD. Thus, enforcing

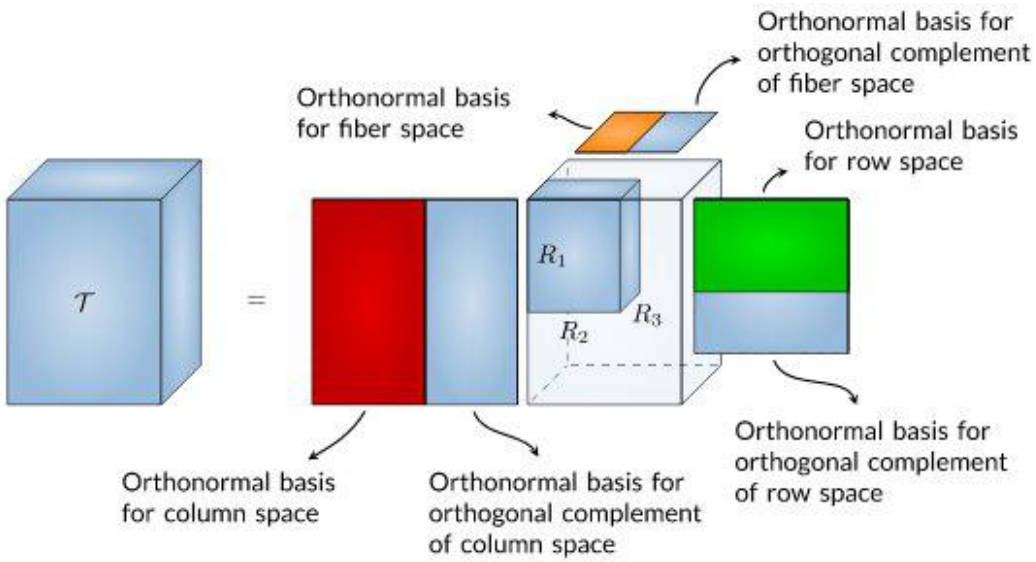

Figura 1.8: An Higher Order SVD of a third-order rank-(R1-R2-R3) tensor and the different spaces, from Tensorlab [13].

$A(k)$ to have r_k columns, the HOOI solution is defined by the following algorithm:

initialize via HOSVD
while not converged do
  for $k = 1, 2, ..., N$
    $Y \leftarrow X \times 1(A^{(1)})^T \times 2 \cdots \times k-1(A^{(k-1)})^T \times k+1(A^{(k+1)})^T \times k+2 \cdots \times N(A^{(N)})^T$
    $A^{(k)} \leftarrow rk$ leading left singular vectors of $Y(k)$
  end for
end while
$G \leftarrow X \times 1(A^{(1)})^T \times 2(A^{(2)})^T \times 3 \cdots \times N(A^{(N)})^T$ (1.23)

1.2.3 Canonical Polyadic Decomposition

The Polyadic Decomposition (PD) [10] approximates a tensor with a sum of R rank-one tensors. If the number of rank-one terms R is minimal, then R is called the rank of the tensor and the decomposition is called minimal or *canonical* (CPD).
For any other arbitrary rank-r, the decomposition is often referred to as CANDECOMP/PARAFAC (CP). As we will discover later, selecting the perfect rank is an *NP-Hard* problem. Hence, from now on we will refer to this decomposition as CP.

Recall the outer product between vectors introduced in section 1.2. Let $\vec{a}$, $\vec{b}$ and $\vec{c}$ be nonzero vectors in $\mathbf{R}^n$, then $\vec{a} \circ \vec{b} \equiv \vec{a} \cdot \vec{b}$ is a rank-one matrix and $\vec{a} \circ \vec{b} \circ \vec{c}$ is defined to be a rank-one tensor. Let T be a tensor of dimensions $I_1 \times I_2 \times \cdots \times I_N$, and let $U^{(n)}$ be matrices of size $I_n \times R$ and $\vec{u}_r^{(n)}$ the $r$þcolumn of $U^{(n)}$, then:

$$T \approx \sum_{r=1}^{R} \vec{u}_r^{(1)} \circ \vec{u}_r^{(1)} \circ \cdots \circ \vec{u}_r^{(N)}. \tag{1.24}$$

A visual representation of this decomposition in the third-order case is shown in **??**

It's interesting to notice that CPD can be regarded as a special case of a Tucker Decomposition in which the core tensor **G** is constrained to be a super-identity I, which is an extension of the identity matrix and has all one's on its superdiagonal and all zero's off the superdiagonal.

1.3 Application of tensor decompositon on CNN

1.3.1 Convolutional layer as 4-mode tensors

Convolutional layers units are organized as 3D tensors (*map stacks*) with two spatial dimensions and the third dimension corresponding to the different maps, or "channels". The most computational demanding operation within CNNs is the generalized - or standard - convolution that maps an input tensor $U(.,.,.)$ of size $X \times Y \times S$ to an output tensor $V(.,.,.)$ of size $(X - d + 1 \times (Y - d + 1) \times T$ using the following linear mapping:

$$V(x, y, t) = \sum_{i=x-\delta}^{x+\delta} \sum_{j=y-\delta}^{y+\delta} \sum_{s=1}^{S} K(i - x + \delta, j - y + \delta, s, t)U(i, j, s) \tag{1.25}$$

Thus it is possible to summarize a convolutional layer with the 4D kernel tensor $K(.,.,.,.)$ of size $d \times d \times S \times T$ where d is the spatial dimension of the 2D convolutional kernel, S and T are the input channels and the output channels dimension respectively. The "half-width" $\frac{d-1}{2}$ is denoted with δ, assuming square shaped kernels and even value of d as it is most often the case.

1.3.2 CP

Once we have modelized the convolutional kernel, we can actually grasp how the tensor decomposition can be practically applied on a CNN. A rank-R CP-decomposition of a 4D tensor will result in 4 factor matrices K^n with size equals to $I_n \times R$. Therefore, applying this decomposition on $K(.,.,.,.)$ will result in:

$$\sum_{r=1}^{R} K^x(i - x + \delta, r)K^y(j - y + \delta, r)K^s(s, r)K^t(t, r) \tag{1.26}$$

where K^x, K^y, K^s, K^t are the four components of the obtained decomposition of size $d \times R$ $d \times R$ $S \times R$ and $T \times R$ respectively.

Hence, plugging 1.26 into 1.25 and performing permutations and grouping of summands will result in the following mathematical receipt to compute a decomposed forward pass of a convolution in a CNN:

$$V(x, y, t) = \sum_{r} K^t(t, r) \left(\sum_{i} \sum_{j} K^x(i - x + \delta, r)K_r^y(i - y + \delta, r) \left(\sum_{s} K^s(s, r)U(i, j, s) \right) \right) \tag{1.27}$$

Equation 1.27 can be splitted into four steps that constitutes a sequence of four squeezed convolutions. The former convolutions will replace the single more costly one:

$$U^s(i,j,r) = \sum_{s=1}^{S} = K^s(s,r)U(i,j,s) \tag{1.28}$$

$$U^{sy}(i,y,r) = \sum_{j=y-\delta}^{y+\delta} K^y(j-x+\delta,r)U^s(i,j,r) \tag{1.29}$$

$$U^{syx}(x,y,r) = \sum_{i=x-\delta}^{x+\delta} K^x(i-x+\delta,r)U^{sy}(i,y,r) \tag{1.30}$$

$$V(x,y,t) = \sum_{r=1}^{R} K^t(t,r)U^{syx}(x,y,r), \tag{1.31}$$

where U^s, U^{sy} $Usyx$ are intermediate tensors i.e., map stacks.
This decomposition scheme is illustrated in figure 1.9.

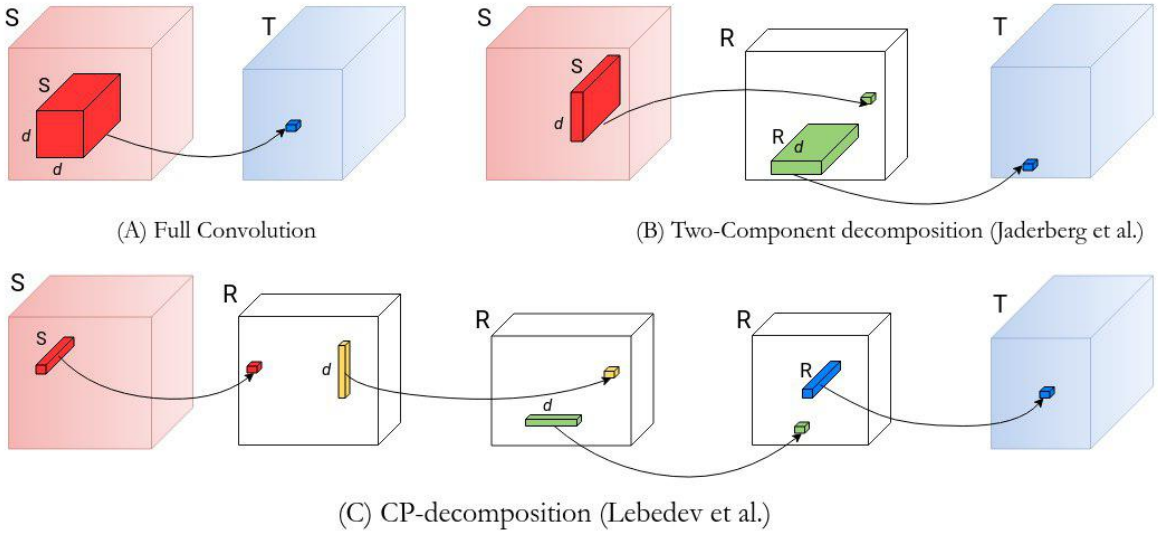

Figura 1.9: Tensor Decompositions for speeding up a generalized convolution. Each box correspond to a feature map stack within a CNN, (frontal sides are spatial dimensions). Arrows show linear mappings and demonstrate how scalar values on the right are computed. Initial full convolution (A) computes each element of the target tensor as a linear combination of the elements of a 3D subtensor that spans a spatial d × d window over all input maps. Jaderberg et al. (B) approximate the initial convolution as a composition of two linear mappings in which the intermediate mpa stack has R maps, being R the rank of the decomposition. Each of the two-components computes each target value with a convolution based on a spatial window of size dx1 or 1xd in all input maps. Finally, CP-decomposition (C) by Lebedev et al. approximates the convolution as a composition of four smaller convolutions: the first and the last components compute a standard 1x1 convolution that spans all input maps while the middle ones compute a 1D grouped convolution **only on one** input map. Each box is mathematically described in equation (1.32-1.31)

Note that the "actual" convolution step is performed in the two middle equations in a *separable* way i.e., the two spatial kernal are 1D with size $d \times 1$ and $1 \times d$ respectively. These are the steps in which *convolution properties (padding, stride, etc.) need to remain the same* as the original convolution in order to preserve the original output size (H' W'). These two steps alone could already replace the previous convolution and gain a speedup.

However here we can fully exploit the factorization provided by CP-decomposition by enclosing the convolution into two $[1 \times 1]$ convolutions that reduce the channels by a significant amount. The first one aims at squeezing the channel depth while the last one restore the original size expected from the subsequent layer, thus making the decomposition effectively embeddable in a pre-trained model.

Furthermore, as already mentioned in chapter 3 section **??**, the $[1 \times 1]$ convolution has the well known benefict of creating a "Network-in-Network" while being at the same time very cheap to compute.

Another peculiarity that is not trivial to capture at first is that the connections between the two separable convolutions are actually 1-to-1 (groups = number of input channels). This is very similar to what happens in depthwise convolutions **??** and have a significant impact on computational cost. If one doesn't want to group layer connections that way, he can always replicate U^{sy} and U^{syx} on each filter, thus having R filters with same weights; More details on complexity will follow in section 1.3.4.

CPD-3

There is another way to compute a convolutional layer with CP that has been suggested in[14] and consists in a three-way decomposition that preserves the standard spatial decomposition. Some might argue that spatial kernel in most modern architecture are small ($[3 \times 3]$ $[5 \times 5]$) and so the factorization gain is not significant compared to the much bigger channels size [15]; or simply the 4-way decomposition ends up being too aggressive and we need to preserve more original weights.

In these scenarios, a simple reshaping of the 4D tensor K of size $(d \times d \times S \times T)$ into a 3D tensor of size $d^2 \times S \times T$ could be helpful. In fact, composing the CP-decomposition with newly shaped tensor will result in three factor matrices of size $d^2 \times R$, $S \times R$ and $T \times R$. The two separable convolution can now be replaced by a single standard $[d \times d]$ convolution, mathematically:

$$U^s(i,j,r) = \sum_{s=1}^{S} = K^s(s,r)U(i,j,s) \tag{1.32}$$

$$U^{syx}(i,y,r) = \sum_{i=x-\delta}^{x+\delta} \sum_{j=y-\delta}^{y+\delta} K^{xy}(i-x+\delta, j-x+\delta, r)U^s(i,j,r) \tag{1.33}$$

$$V(x,y,t) = \sum_{r=1}^{R} K^t(t,r)U^{syx}(x,y,r), \tag{1.34}$$

This process is described in figure 1.10 in the Tucker decomposition section.

Summary

It is convenient to summarize all the properties of CP-decomposition on CNN:

(i) the *first layer* of the decomposition acts as a channel reduction;

(ii) the *last layer* of the decomposition restore channels to the original size;

(iii) both these enclosing layers perform $[1 \times 1]$ convolution i.e., they perform a linear recombination of input pixels and acts a "Network-in-Network";

(iv) the *middle layer(s)* perform the actual convolution and must possess the same convolution properties of the original convolution in order to obtain the output dimensions H' and W'

(v) the middle steps can be either separable or a regular convolution but with smaller channels;

(vi) if CP is applied on all dimensions, then the convolution will be performed with separable kernels; the connectivity between each of the spatial 1D convolution layers is of 1-by-1, thus saving another $\times R$ multiplications in each convolution;

(vii) if the decomposition is too aggressive we can replicate the d weights of each separable kernel on all the filter maps and fall back to a standard N-to-N connectivity between input and output channels;

(viii) CP-decomposition can be performed on only 3 dimensions with a simple tensor reshape; in that case the middle layers will collapse into one single $d \times d$ regular convolution step. This is an even less aggressive way of decomposing layers.

1.3.3 Tucker

The Tucker method is a more generalized way of decomposing a tensor with respect to CP. It factorizes a tensor K into a core tensor $\mathbf{G}$ and a list of factor matrices. If we apply a full Tucker decomposition on a convolutional forward pass, we end up with the following formula:

$$K(i,j,s,t) = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} G_{r_1,r_2,r_3,r_4} K^x_{r1}(i) K^y_{r2}(j) K^s_{r3}(s) K^t_{r4}(t) \tag{1.35}$$

However, this is not the only way to apply it. In fact, while CP-decomposition needs a "hack" to be applied on 3-dimensions only, the Tucker decomposition exposes a property that comes in handy for this specific purpose: it does not need to be applied along *all modes* (axis) of the tensors.
This is useful when we want to be more conservative and preserve the $d \times d$ convolution, going for what is known as a *Tucker-2* decomposition, from Kim et al.[15]:

$$K(i,j,s,t) = \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} \mathbf{G}_{i,j,r_3,r_4}(j) K^s_{r3}(s) K^t_{r4}(t) \tag{1.36}$$

Plugging equation 1.36 into 1.25 a new equation for the Tucker convolutional forward pass is obtained:

$$V(x,y,t) = \sum_{i=x-\delta}^{x+\delta} \sum_{j=y-\delta}^{y+\delta} \sum_{s} \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} \mathbf{G}(i-x+\delta)(j-y+\delta) r_3 r_4 K_{r3}^{s}(s) K_{r4}^{t}(t) X(i,j,s) \tag{1.37}$$

If we divide the former equation we can define a three steps Tucker convolutinal pass:

$$U^{s}(i,j,r) = \sum_{s=1}^{S} = K^{s}(s,r) U(i,j,s) \tag{1.38}$$

$$U^{syx}(i,y,r) = \sum_{i=x-\delta}^{x+\delta} \sum_{j=y-\delta}^{y+\delta} \mathbf{G}_{i,j,r_3,r_4} U^{s}(i,j,r) \tag{1.39}$$

$$V(x,y,t) = \sum_{r=1}^{R} K^{t}(t,r) U^{syx}(x,y,r), \tag{1.40}$$

From which we can deduce few properties:

(i) similarly to the standard CP, the first and the last layers act as a dimensionality reduction (S to R_3) and reshaping back (R_4 to T) respectively;

(ii) differently from CP, the decomposition results in three layers and the one in the middle performs a regular $[d \times d]$ spatial convolution;

(iii) diffrently from CP, the spatial convolution is *not* depthwise separable (i.e. 1-by-1 connectivity)

(iv) the compression comes only from channels dimensionality reduction: the spatial convolution counts $[d \times d \times R_3 \times R_4$ parameters; if $R_3 \ll S$ and $R_4 \ll T$, the impact is significant.

(v) the scheme is instead similar to CPD-3 with the difference of being more elastic thanks to two degrees of freedom i.e., the ranks R_3, R_4, instead of only one rank-R.

Figure 1.10 illustrate equation 1.38-1.40.

Full Tucker

As mentioned earlier, Tucker is more flexible compared to CP and there are other ways to explore for Tucker.

Instead of truncating the factors, we could compute a full Tucker decomposition and use all the resulting terms. Mathematically, this is described in the following equations:

First again is the dimensionality reduction from S to R_3.

$$U^{s}(i,j,r_3) = \sum_{s} K^{s}(s,r_3) U(i,j,s) \tag{1.41}$$

Then we apply 1D vertical convolution:

$$U^{sy}(i,y,r_3) = \sum_{j} K^{y}(j-y+\delta, r_3) U^{sx}(i,j,r_3) \tag{1.42}$$

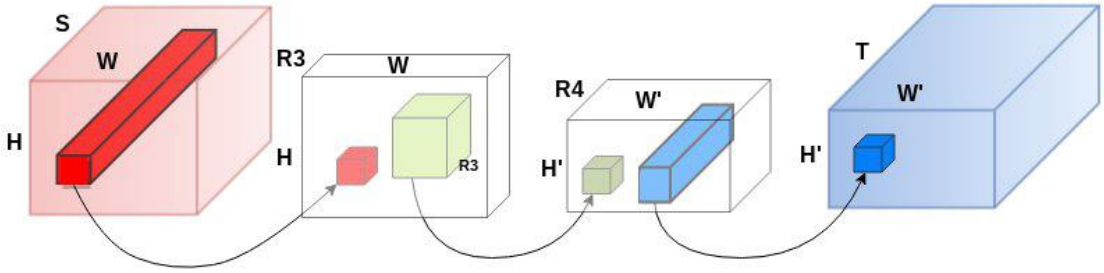

Figura 1.10: Tucker-2 Decompositions for speeding-up a generalized convolution. Each box corresponds to a 3-way tensor $X, Z, Z^{'andY}$ in equation (1.38-1.40). Arrows represent linear mappings and illustrate how each scalar value on the right is computed. Red tube, green cube and blue tube correspond to 1x1, dxd and 1x1 convolution respectively.

a 1D horizontal convolution:

$$U^{syx}(x, y, r_3) = \sum_i K^x(i - x + \delta, r_3) U^{sy}(i, y, r_3) \tag{1.43}$$

we compute the convolution with smaller core kernel $\mathbf{G}$:

$$U^{syx\sigma}(x, y, r_4) = \sum_{r_3} \mathbf{G}_{1,1,r_3,r_4} U^{syx}(x, y, r_3) \tag{1.44}$$

and finally reshape the output channel:

$$V(x, y, t) = \sum_{r_4} K^t(t, r_4) U^{syx\sigma}(x, y, r_4) \tag{1.45}$$

This method will enable a separable 1D convolution instead of a regular $[d \times d]$. However, it seems more complex to model than a 4-way CP-decomposition while also being more computationally expensive (for more details on complexity see the following section).
In a sense, it seems like CP-decomposition is naively best suited to obtain the spatially separable kernels and so be the more aggressive between the two, whilst Tucker is mathematically more flexible and can be more easily adapted to different designs. Hence, for this work the methods that have been employed are CPD and Tucker-2.

Moreover, it is possible to combine the two in a gradually more aggressive approach: at first the model can be decomposed using Tucker; if the central regular convolution still have many parameters, it can be decomposed again but with CP this time. This will probably lead to too many layers and would open a vanishing gradient scenario. However, deep residual nets[16] have already shown how to tackle this issue.

1.3.4 Complexity analysis

Standard convolution operation is defined by STd^2 parameters i.e, the number of parameters in the tensors. The computational cost is obtained by taking into account the height H and the width W of the input maps: $(STd^2)(HW)$.

CP complexity

With their approaches, both in Jaderberg et al.[17] and Zhang et al. [7] drop this complexity down to $Rd(S+T)$. Assuming the required rank is comparable or several times smaller than S and T, that is taking a rank $R \approx \frac{ST}{S+T}$, then the complexity would be reduced by a factor of d times.
Following the CP-decomposition proposed by Lebedev et al. [18], the complexity drops to $R(S + 2d + T)$. In modern architectures, almost always we can assume $d \ll T$, thus $R(S+2d+T) \approx R(S+T)$, which add another d factor of improvement over Jaderberg et al. and of d^2 over the initial convolution.

If we call $\mathbf{K}_{std}$ and $\mathbf{K}_{dec}$ the standard and decomposed convolution operation respectively, the *compression ratio* can be defined as

$$\mathcal{C}_r = \frac{O(\mathcal{K}_{std})}{O(\mathcal{K}_{dec})} \tag{1.46}$$

which in the case of CP-decomposition equals to:

$$\mathcal{C}_r = \frac{STd^2}{R(S+2d+T)} \tag{1.47}$$

Similarly, we can define the speed-up ratio S_r by considering the *multiplication-addition* operation on the input of height H and width W:

$$\mathcal{S}_r = \frac{STd^2HW}{RSHW + dRH'W + RdH'W' + RTH'W')} \tag{1.48}$$

Clearly both $\mathcal{C}_r$ and S_r depend on the rank R, which is the most important *hyper-parameter* of this approach. As described in the coming sections, the choice of the rank is non trivial; it is rather an ill-posed problem that boasts its very own research field [19].

However, after several experiments and careful parameter tuning, few patterns will come out and a *rule-of-thumb* can be spotted, at least on the same kind of architecture. After that, a trial-and-error strategy can be applied to enforce a more aggressive compression; more details will be described in section 1.4.5

CP-3

The compression factor for CP-decomposition on a reshaped 3D tensor K of size $S \times d^2 \times T$ is equal to:

$$mathcalC_r = \frac{d^2ST}{R(S+d^2+T)} \tag{1.49}$$

which has a d factor more than classical CP on the central convolution, but still having less parameters than Tucker-2.

Table 1.1: Summary of the parameters required by the different decomposition methods analyzed.

Method	Params
Low rank SVD (Zhang et al.)	$Rd(S+T)$
Full Tucker	$SR_3 + dR_1 + dR_2 + R_1R_2R_3R_4 + R_4T$
Tucker-2 (Kim et al.)	$SR_3 + R_3R_4d^2 + R_4T$
CPD (Lebedev et. al)	$R(S+2d+T)$
CPD-3 (Astrid)	$R(S+d^2*T)$

Tucker

Tucker-2 decomposition boasts a compression factor of:

$$\mathcal{C}_r = \frac{d^2ST}{SR_3 + R_3R_4d^2 + R_4T}$$

which is more conservative than the CP approaches but has a good impact when the ranks are much smaller than the channels dimensions.

The full Tucker decomposition is a bit more complex with its five steps; the compression ratio amount to

$$\mathcal{C}_r = \frac{d^2ST}{SR_3 + dR_1 + dR_2 + R_1R_2R_3R_4 + R_4T}$$

.

In all cases it is assumed

$$R_1, R_2, R_3, R_4 > 1, 1, 1, 1$$

. The speedup ratio can be computed from the quantities potrayed above by multipying them with H, W, H', W' which are the height and width before and after the spatial convolution respectively.

Complexity recap Taking into account also the approach of Zheng et al. [7], in table 1.1 can be seen a comparison of parameters reduction for the different methods explored.

1.4 In-depth discussion

1.4.1 Rank estimation

Ranks play key role in decomposition. If the rank is too high, the compression may not be significant at all and if it is too low, the accuracy would drop too much to be recovered by fine-tuning. Therefore a trade-off between compression and accuracy must always be kept in mind.
However, there is no straight solution to rank estimation. As mentioned earlier, determining the rank is *NP-Hard* [19][20][21].

iterative methods

One simple but costly way to predict a good rank is a trial-and-error approach, based on a threshold *th*:

1. start with a *lower-bound* rank based on a truncation error of the HOSVD explained in section 1.2.2
2. compute the relative error using the Frobenius norm:

$$Err = \frac{\|\mathbf{K}' - \mathbf{K}\|_F}{\|\mathbf{K}\|_F}$$

3. if the error is less than the threshold i.e., $Err \leq th$, the rank is chosen as the best rank; otherwise increment the rank and restart from 1.

In Tensorlab this is done by a utility function called `'rankest'`. A simple script to estimate how scalable this method is for large tensors is showed in B.

In figure 1.11 we can see how this approach does not scale so well with increasing input and output maps dimensions, which are also all not so large i.e., less than 256.

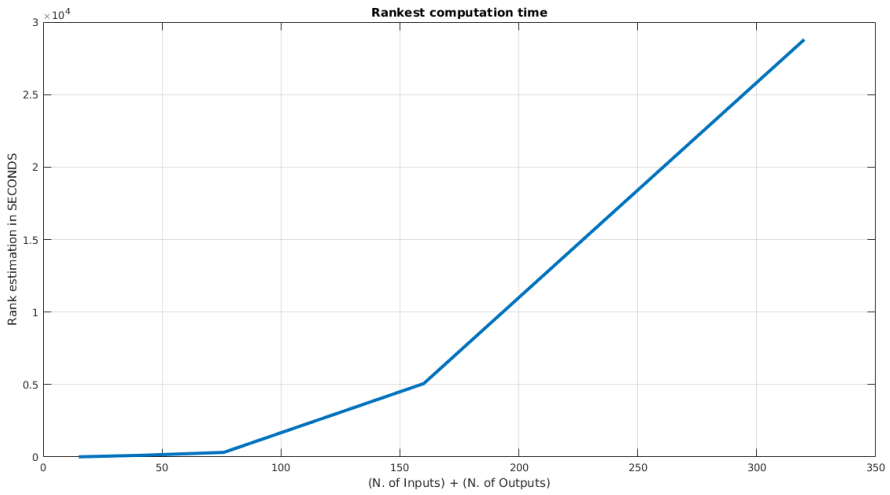

Figura 1.11: Evaluation of the 'rankest' method for different sizes of input and output maps. Note that the maps sizes can be even higher in CNNs. Clearly, an iterative rank estimation approach does not scale well.

iterative compression and fine-tuning A simple solution that has been applied by different authors is an iterative careful manual tuning [14] [18] i.e., choosing an arbitrary rank, calculating the accuracy and manually updating the rank according to an elementary rule: the higher is the drop in accuracy caused by its decomposition, the higher is the rank needed by the layer.
In Astrid and Lee [14] they define a metric to measure the *sensitivity* of a layer l as:

$$S_l = \frac{Loss_l}{Total \quad Loss} \tag{1.50}$$

that is how much the error of the approximation of one layer's tensor affects the whole network accuracy. The sensitivity is intended to be computed *after* the fine-tuning has been performed.

In Lebedev et al. [18] the authors compute the relative error

$$Err = \frac{\|\mathbf{K}' - \mathbf{K}\|_F}{\|\mathbf{K}\|_F}$$

(as shown in step 2 above) of the tensor approximation and how much it does impact accuracy before and after fine-tuning. Once a good rank for a specific layer is found, we can move to the next one. Following this slow but careful process, we will arguably end up with a fair enough compressed version of the original model.

Global Analytics VBMF

A work from 2013 by Nakajima et al. [22] has proven to find a way to compute *analytically* a solution to the *variational Bayesian matrix factorization* (VBMF).

More precisely, the global solutionis a reweighted SVD of the observed matrix, and each weight can be obtained by solving a quartic equation. This will result in a global optimum instead of a local one. VBMF is able to automatically find noise variance, de-noise the matrix under a low-rank assumption and thus compute the rank.

Since VBMF operates on matrices, the afore step to apply it on convolution tensors is a tensor-unfolding (or *matricization*).

Given the tensor K of size $[S \times T \times d \times d]$ we have to unfold it on mode=1 and mode=2 (with indices starts at 1), obtaining two matrices of size $[S \times d^2T]$ and $[T \times d^2S]$ respectively. An example of this process is depicted in figure 1.12.

Last but not least, comes the actual choice of the rank. If Tucker-2 decomposition is employed, the resulting ranks will be both selected as R_3 R_4 as in Kim et al. [15]. On the other hand, if we are using CPD we only need one rank R. Thus, we can take the highest rank to be more accurate; the lowest to apply a more aggressive decomposition or an average for a trade-off.

VBMF can also be used to tackle a subtle issue with the iterative approach of [14]: in fact when fine-tuning is performed on the whole network, the sensitivity on a layer is computed after previous layers have already been compressed and thus the previous rank change is ignored. Employing VBMF enable us to compute a rank estimation independently on each layer *before* actually doing the compressing-and-fine-tuning step.

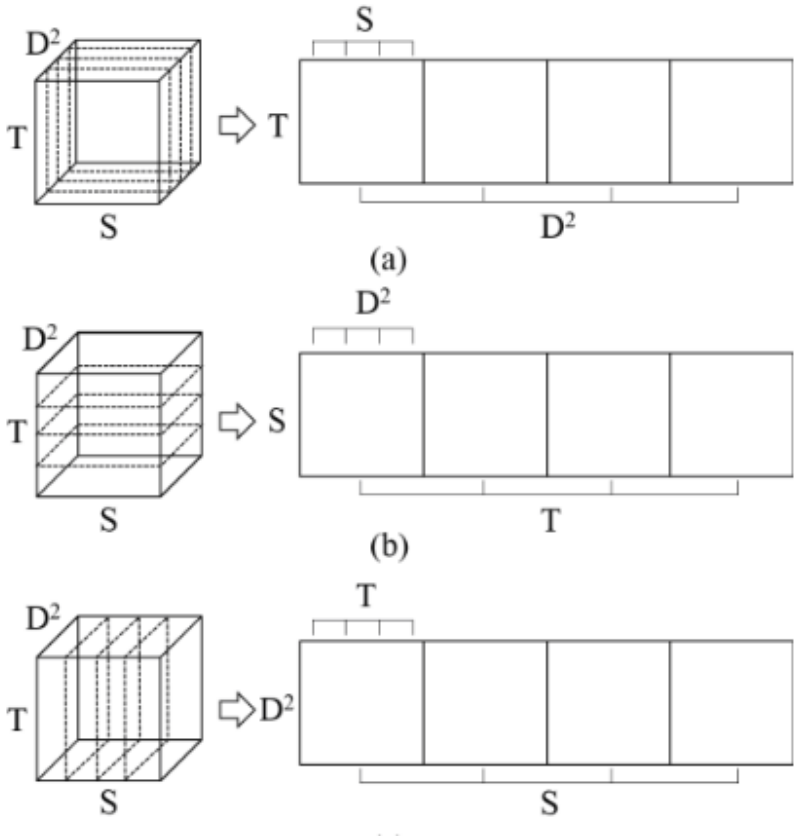

Figura 1.12: Mode-1 (top left), mode-2 (top-right), and mode-3 (bottom left) tensor unfolding of a third-order tensor. **Not shown**: after unfolding, the rank R of the resulting matrices is computed with VBMF. (Bottom right): Then, given the rank R, the Tucker decomposition produces a core tensor $\mathcal{S}$ and factor matrices $A^(1)$, $A^(2)$ and $A^(3)$ of size $I_1 \times R$ $I_2 \times R$ and $I_3 \times R$ respectively.

Variational autoencoders

Despite the advanteges of bayesian methods like VBMF over multi-linear ones, A recent work by Yingming et al, [**VAE**] have shown how Variational Auto-Encoders (VAE) can outperform state-of-the-art methods to tackle this issue. VAEs are actually a way to compute the CP-decomposition and not the rank of a tensor, but they are capable of doing so without depending heavily on a correct rank.
Their model leverage on a neural network to learn a complex non-linear function whose parameters can be learned from data. The authors explain that tensor entries can be generated via a very complex and intractable random process determined by some *latent* variables (as the rank). This complex generative process can be captured by training a neural network (VAE).

Their proposed model outperforms by a significant margin tradtional methods, as depicted in 1.13

The most captivating idea of this work is that it prove how tensor decomposition is a perfect task for neural networks. There are too many latent variables to be accounted to compute a perfect tensor decomposition with non-probabilistic models.

Therefore, it would be interesting to combine their variational auto-encoder model to optimize the accuracy of another network by finding the best tensor approximation of its layers. The outlined scenario could be tackled through the use of of Generative

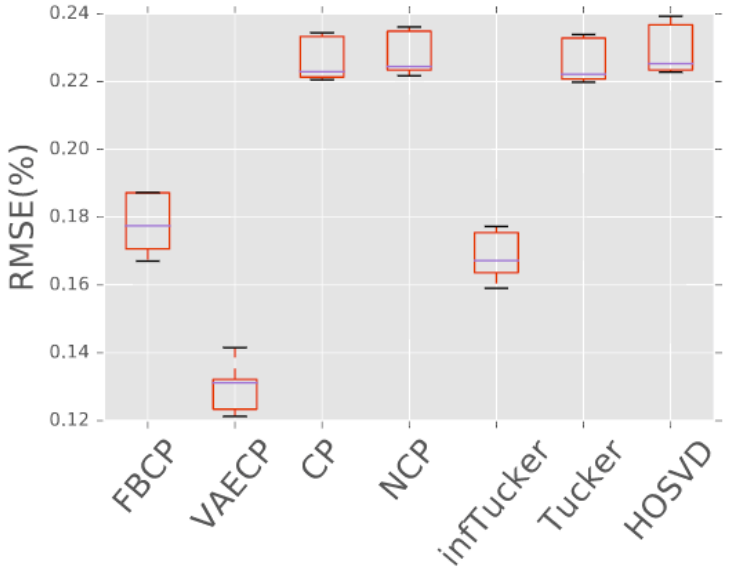

Figura 1.13: Comparison of different optimization algorithms to compute tensor decomposition on real datasets. The method proposed by Liu et al. outperforms other methods by a large margin.

Adversarial Networks (GANs)[**GAN**] that are actually related to VAEs. It would be interesting, as a future work, to understand if this is actually possible.

1.4.2 Decomposition algorithms overview

While the decomposition scenario for convolutional neural networks has been defined, it has not been clarified yet *how* to actually compute a decomposition. To this aim, this section provides a quick overview of the available algorithms in literature with few examples computed with the aim of powerful tensor toolbox for $MATLAB$, called Tensorlab [13].
A depth discussion of these algorithms is out of the scope of this work. For more details please check this rich reference page of the Tensorlab documentation [23].

CP

The CP-decomposition can be computed in a variety of different ways:

- *alternating least squares* (ALS)
- *nonlinear least squares* (NLS)
- *nonlinear unconstrained optimization*
- *Variational Auto-Encoder CP* (VAE-CP)
- *non-negative cpd* (NCP)
- *randomized block sampling for large-scale tensors*
- *generalized eigenvalue decomposition*
- *simultaneous diagonalization*

- *simultaneous generalized Schur decomposition*

Usually, the most common algorithms for CNN compression are ALS and NLS. Beside the principal algorithm itself, the convergence of the computation for large tensors is influenced - like many other optimization problems - by a good initialization. Thus, most algorithms works by starting with an initial guess; also the computational time is reduced by a significant factor when a pre-computed solution is used.

Basically, the computation of a good CPD is a multi-step approach that relies on a main algorithm (ALS, NLS, unconstrained opt) and a bunch of other methods that help the optimization. A simple test in Tensorlab to compare different algorithms is shown in appendix B, the results can be observed in figure 1.14.

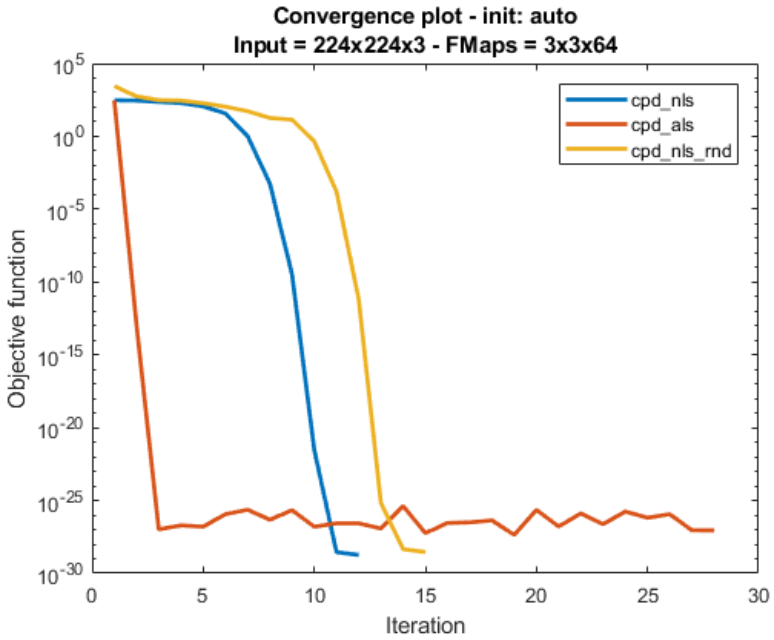

Figura 1.14: Comparison of different optimization algorithms to compute CPD; between yellow and blue line only the init method is diverse. It is evident how ALS is indeed faster but more unstable and less precise.

ALS is often faster while being less accurate than NLS. As it will be presented in the experiments chapter, they can both successfully be applied to CNN with minor difference. In fact, thanks to fine-tuning, we are able to minimize the reconstruction error of kernel tensors through SGD.

To dig deeper into the optimization details, an exhaustive comparison between four main algorithms is explained in [24]. Their results on accuracy and timing are reported in figure 1.16.

It is also interesting to see how these methods depend heavily on the chosen rank R. To this aim, a MATLAB script that computes CP-NLS of a relatively large matrix for different values of R is available in appendix B. Results are shown in 1.15.

Notably, the higest rank leads to a far better minimum of the NLS optimization. However, such values of R are incompatible with a good speeup in convolutions. Thus, between an accurate but costly decomposition and a fast inaccurate one plus retraining, the trade-off discussion seems to fall in favor of the latter.

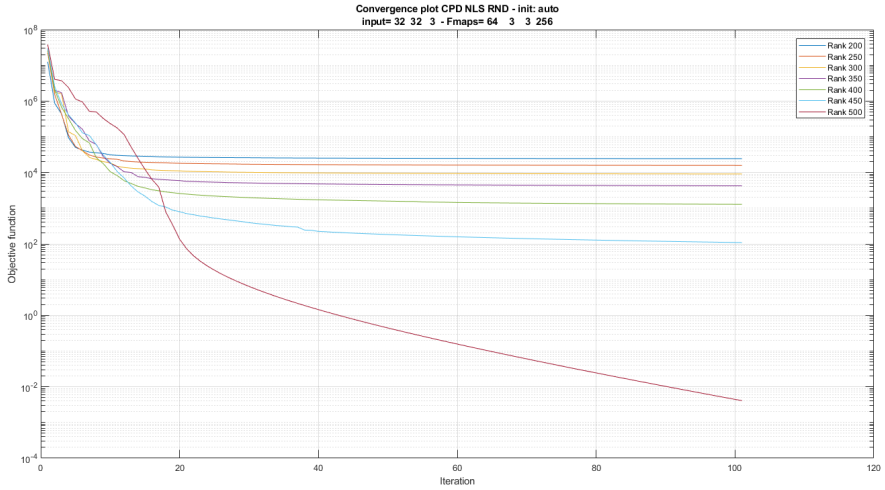

Figura 1.15: Comparison of CP-NLS decomposition for different ranks: the higher the R-terms, the higher is the accuracy. For R=500 the decomposition is much more accurate than other ranks.

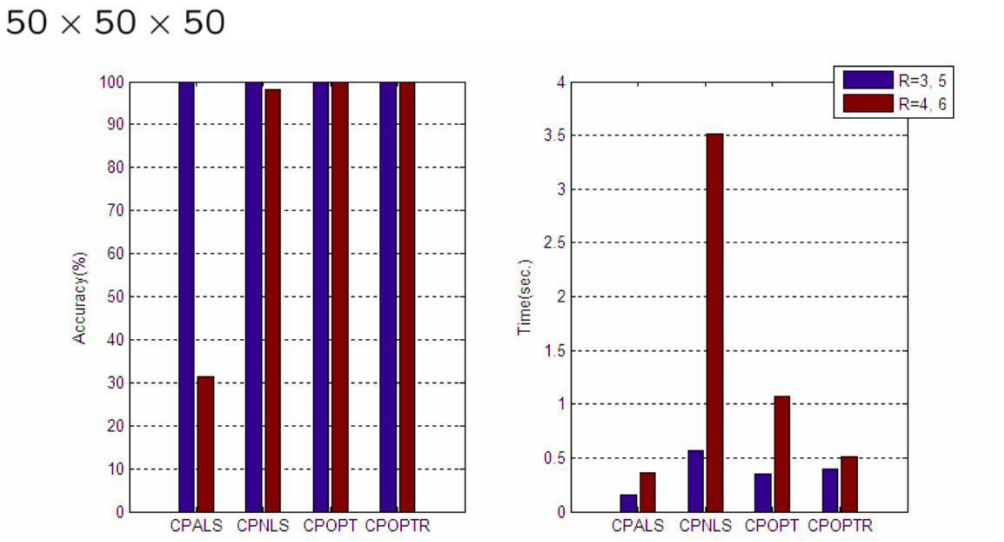

Figura 1.16: Accuracy and timing comparison between various CP-decomposition algorithms. ALS is always fast but much less accurate than CP-OPT. Image from [24].

Tucker

Like CP, the Tucker decomposition can also by computed in several different ways:

- NLS
- higher order orthogonal iteration (HOOI)
- nonlinear unconstrained optimization with initial guess
- adaptive cross approximation
- differential-Newton (only for dense third-order tensor)

- Riemannian trust region method (only for dense third-order tensor)

The *HOOI* was already introduced in section 1.2.2 as a logical consequence of the combination Eckart-Young theorem for SVD and the higher-order SVD, and have been extensively applied througout the experiments as the implementation was straightforward, being available in python libraries.

Interestingly, faster methods have been proposed in literature [25] [26] and coule be subject for further experiments in future work.

1.4.3 A micro-architectural view

The concepts of macro and micro-architecture of a CNN have been introduced in chapter 3. Taking a closer look to how the decomposed layers are assembled, we can lay out a common pattern that lead us to a generalised decomposition block.

As a matter of fact, regardless of the method (CP/Tucker), the first and the last layer of the decomposition always act as a dimensionality reduction whereas the layers in the middle perform the spatial convolution (more details are explained in section 1.3.2. This idea is very close to that of the *Inception-module* firstly introduced by Google in 2014 [**googlenet**]. and then widely applied in successive works [**squeeznet**], [**chollet**], [**mobilenets**].
The nice thing is that this design fits perfectly with the factor matrices of the decomposition. Therefore a general architecture of a *"tensor decomposition block"* (TP-block) can be finally introduced in figure 1.17

The type of convolution that is computed in the middle can be either separable or standard and can have an arbitrary number of channels depending on the choice of the rank(s) of the decomposition.
As a side note: it can also be pointed out that since this block is not a composition of several regular layers but of smaller parts of a single of them, we could coin the term *nano-architecture* of a convolutional neural network.

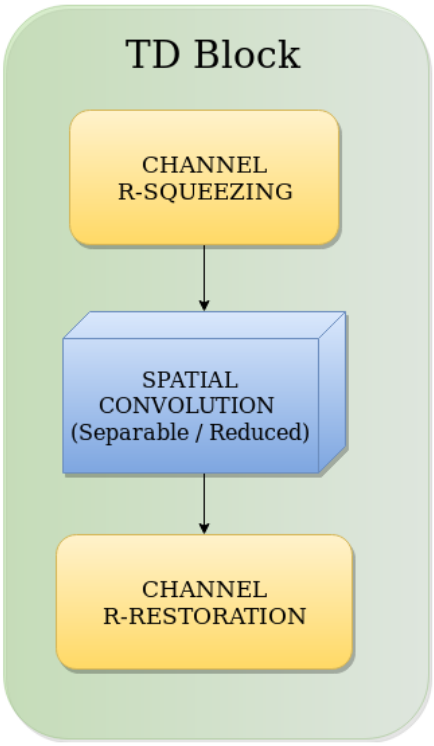

Figura 1.17: A generic Tensor Decomposition (TP) block micro-architecture. The first and last layers performs channels squeezing/restoration while the actual convolution is performed inside these two layers. This convolution can be either separable or standard; either way the cross-channel parameters redundancy is exploited.

1.4.4 TP-block model

The original goal of applying tensor decomposition on CNNs is the ability to shrink a pre-trained model without accuracy drop. However, one can easily guess that if a TP-block is a good fit to substitute a whole layer after training, then it will likely be as good if applied from the beginning.
Indeed, in the next chapter several experiments about this approach demonstrate how TP-blocks can perform well when training models from scratch.

This suggests *a new way* of building standard CNNs. To this aim, it is interesting how *Mobile Nets* [**mobilenets**], that builds upon a similar separable architecture, have been included in Tensorflow Lite, the mobile-optimized version of the most popular deep learning framework, becoming the go-to solution to deploy CNNs on mobile.

1.4.5 A framework for decomposition

In lights of all the concepts explained in this chapter, it is finally possible to outline a framework for CNN decomposition. The following constitute the main key points to address in order to apply tensor decompositions on CNNs:

1. choice of a model that can arguably be over-parametrized and thus suitable for decomposition;
2. selection of a desired range for compression ratio and speed-up;
3. selection of the number and connectivity of the layers that the TP-Block should enclose;
4. according to the first points, selection of the type of the decomposition between CP (more aggressive) and Tucker (more stable);
5. selection of a rank estimation technique (e.g. iterative, VBMF);
6. selection of the class of algorithms to actually compute the tensor decomposition (e.g. ALS, NLS, VAE, ...);
7. selection of a metric to measure the performance of the CNN according to the specific domain;
8. enforcing on the rank according to rule of thumbs and results.

Regarding the last point, it should be noted that after successfully trained a decomposed model, we may want to enforce a higher compression rate with an again arbitrary rank. This time though, the higher bound wold be given by techniques like VBMF, while the lower bound can be found with a "rule of thumb" dependent on the order of magnitude of the layer size.

As a final note: this chapter focused only on the 2D convolution scenario, but with due consideration every aspect that have been analyzed can be extended also to the 3D convolution (or convolutions between volumes) scheme. Indeed, 3D convolutions could expose an even bigger advantage in employing decomposition; it would be interesting to explore the best decomposition strategy for that scheme as a future work.

Appendix A

MLP: Codice addizionale

A.1 Classi in Lua

In Lua manca il costrutto delle classi. Si può tuttavia crearle utilizzando tables e meta-tables. Per realizzare il multi-layer perceptron si è utilizzato una piccola libreria, di seguito riportata.

```
-- class.lua
-- Compatible with Lua 5.1 (not 5.0).
function class(base, init)
   local c = {} -- a new class instance
   if not init and type(base) == 'function' then
      init = base
      base = nil
   elseif type(base) == 'table' then
      -- our new class is a shallow copy of the base class!
      for i,v in pairs(base) do
         c[i] = v
      end
      c._base = base
   end
   -- the class will be the metatable for all its objects,
   -- and they will look up their methods in it.
   c.__index = c

   -- expose a constructor which can be called by <classname>(<args>)
   local mt = {}
   mt.__call = function(class_tbl, ...)
      local obj = {}
      setmetatable(obj,c)
      if init then
         init(obj,...)
      else
         -- make sure that any stuff from the base class is initialized!
         if base and base.init then
            base.init(obj, ...)
         end
      end
      return obj
   end
   c.init = init
   c.is_a = function(self, klass)
      local m = getmetatable(self)
      while m do
         if m == klass then return true end
         m = m._base
      end
      return false
   end
   setmetatable(c, mt)
```

```
44    return c
45 end
```

A.2 La classe Neural_Network

Nel capitolo **??** si sono mostrati i vari snippet di codice man mano che si introducevano i concetti teorici che stanno alla base di questa implementazione. Di seguito è presentata l'intera classe `Neural_Network` :

```
1  --creating class NN in Lua, using a nice class utility
2  class = require 'class'
3
4  Neural_Network = class('Neural_Network')
5
6  --init NN
7  function Neural_Network:__init(inputs, hiddens, outputs)
8        self.inputLayerSize = inputs
9        self.hiddenLayerSize = hiddens
10       self.outputLayerSize = outputs
11       self.W1 = th.randn(net.inputLayerSize, self.hiddenLayerSize)
12       self.W2 = th.randn(net.hiddenLayerSize, self.outputLayerSize)
13 end
14
15 --define a forward method
16 function Neural_Network:forward(X)
17    --Propagate inputs though network
18    self.z2 = th.mm(X, self.W1)
19    self.a2 = th.sigmoid(self.z2)
20    self.z3 = th.mm(self.a2, self.W2)
21    yHat = th.sigmoid(self.z3)
22    return yHat
23 end
24
25 function Neural_Network:d_Sigmoid(z)
26    --derivative of the sigmoid function
27    return th.exp(-z):cdiv( (th.pow( (1+th.exp(-z)),2) ) )
28 end
29
30 function Neural_Network:costFunction(X, y)
31    --Compute the cost for given X,y, use weights already stored in class
32    self.yHat = self:forward(X)
33    --NB torch.sum() isn't equivalent to python sum() built-in method
34    --However, for 2D arrays whose one dimension is 1, it won't make any
       difference
35    J = 0.5 * th.sum(th.pow((y-yHat),2))
36    return J
37 end
38
39 function Neural_Network:d_CostFunction(X, y)
40    --Compute derivative wrt to W1 and W2 for a given X and y
41    self.yHat = self:forward(X)
42    delta3 = th.cmul(-(y-self.yHat), self:d_Sigmoid(self.z3))
43    dJdW2 = th.mm(self.a2:t(), delta3)
44
45    delta2 = th.mm(delta3, self.W2:t()):cmul(self:d_Sigmoid(self.z2))
46    dJdW1 = th.mm(X:t(), delta2)
47
48    return dJdW1, dJdW2
49 end
```

A.3 Metodi getter e setter

Nella sottosezione **??** del capitolo **??** si è dimostrato come calcolare numericamente il gradiente. Si è fatto cenno ai *getter e setter* per ottenere dei *flattened gradients*, ovvero "srotolare" i tensori dei gradienti in vettori monodimensionali. I metodi, qui mostrati, necessitano di una comprensione dei comandi di Torch. Data la maggiore popolarità di Python *Numpy*, nel caso il lettore fosse più familiare con quest'ultimo, nell'appendice B è mostrata anche una tabella di equivalenza dei metodi fra i due.

```
--Helper Functions for interacting with other classes:
function Neural_Network:getParams()
   --Get W1 and W2 unrolled into a vector
   params = th.cat((self.W1:view(self.W1:nElement())), (self.W2:view(
    self.W2:nElement())))
   return params
end

function Neural_Network:setParams(params)
   --Set W1 and W2 using single paramater vector.
   W1_start = 1 --index starts at 1 in Lua
   W1_end = self.hiddenLayerSize * self.inputLayerSize
   self.W1 = th.reshape(params[{ {W1_start, W1_end} }],
    self.inputLayerSize, self.hiddenLayerSize)
   W2_end = W1_end + self.hiddenLayerSize*self.outputLayerSize
   self.W2 = th.reshape(params[{ {W1_end+1, W2_end} }],
    self.hiddenLayerSize, self.outputLayerSize)
end

--this is like the getParameters(): method in the NN module of torch, i.e.
     compute the gradients and returns a flattened grads array
function Neural_Network:computeGradients(X, y)
   dJdW1, dJdW2 = self:d_CostFunction(X, y)
   return th.cat((dJdW1:view(dJdW1:nElement())), (dJdW2:view(dJdW2:
    nElement())))
end
```

Appendix B

Il framework Torch

«««< HEAD

B.1 Introduzione

```
%% Rankest performance evaluation
%{
Running rankest(T) on a dense, sparse or incomplete tensor T plots an L-curve which represents the balance
between the relative error of the CPD and the number of rank-one terms R.

The lower bound is based on the truncation error of the tensors multilinear singular values
For incomplete and sparse tensors, this lower bound is not available and the first value to be tried for R is 1.
The number of rank-one terms is increased until the relative error of the approximation is less than options.MinRelErr.
In a sense, the corner of the resulting L-curve makes an optimal trade-off between accuracy and
compression. The rankest tool computes the number of rank-one terms R corresponding to
the L-curve corner and marks it on the plot with a square. This optimal number of rank-one
terms is also rankests first output.
%}

A = {}
times = []
ranks = []
n_inputs = [3, 9, 12, 32, 64]
n_outputs = [12, 32, 64, 128, 256]
kernel_size = 3;

for i=1:5
    A{i} = abs(randn(n_inputs(i), kernel_size, kernel_size, n_outputs(i)))
    tic
    ranks(i) = rankest(A{i})
    times(i) = toc
end

disp('estimated rankest times:')
disp(times)

mul_dims = n_inputs + n_outputs;

h3 = figure(3);
h3 = plot(mul_dims(1:end), times);
set(h1, 'LineWidth',3);
grid on
```

```
38
39 title('Rankest computation time')
40 xlabel('(N. of Inputs) x (N. of Outputs)')
41 ylabel('Rank estimation in SECONDS')
42
43
44 %% plot
45 h = figure(3);
46 h = plot(mul_dims, times);
47 set(h, 'LineWidth',3);
48 grid on
49
50 title('Rankest computation time')
51 xlabel('(N. of INPUTS) + (N. of OUTPUTS)')
52 ylabel('Rank estimation in SECONDS')
```

Torch[27] è un framework per il calcolo numerico versatile che estende il linguaggio Lua. L'obiettivo è quello di fornire un ambiente flessibile per il progetto e addestramento di sistemi di machine learning anche su larga scala.
La flessibilità è ottenuta grazie a Lua stesso, un linguaggio di scripting estremamente leggero e efficiente. Le prestazioni sono garantite da backend compilati ed ottimizzati (C,C++,CUDA,OpenMP/SSE) per le routine di calcolo numerico di basso livello.

Gli obiettivi degli autori erano: (1) facilità di sviluppo di algoritmi numerici; (2) facilità di estensione, incluso il supporto ad altre librerie; (3) la velocità.
Gli obiettivi (2) e (3) sono stati soddisfatti tramite l'utilizzo di Lua poiché un linguaggio interpretato risulta conveniente per il testing rapido in modo interattivo; garantisce facilità di sviluppo e, grazie alle ottime C-API, unisce in modo eterogeneo le varie librerie, nascondendole sotto un'unica struttura di linguaggio di scripting. Infine, essendo ossessionati dalla velocità, è stato scelto Lua poiché è un veloce linguaggio di scripting e può inoltre contare su un efficiente compilatore JIT. Inoltre, Lua ha il grosso vantaggio di essere stato progettato per essere facilmente inserito nelle applicazioni scritte in C e consente quindi di "wrappare" le sottostanti implementazioni in C/C++ in maniera banale. Il binding C per il Lua è tra i più semplici e dona quindi grande estensibilità al progetto Torch.

Torch è un framework auto-contenuto e estremamente portabile su ogni piattaforma: iOS, Android, FPGA, processori DSP ecc. Gli scrpt che vengono scritti per Torch riescono ad essere eseguiti su queste piattaforme senza nessuna modifica.
Per soddisfare il requisito (1) hanno invece ideato un oggetto chiamato `'Tensor'`, il quale altro non è che una "vista" geometrica di una particolare area di memoria, e permette una efficiente e semplice gestione di vettori a N dimensioni, tensori appunto. L'oggetto Tensor fornisce anche un'efficiente gestione della memoria: ogni operazione fatta su di esso non alloca nuova memoria, ma trasforma il tensore esistente o ritorna un nuovo tensore che referenzia la stessa area di memoria. Torch fornisce un ricco set di routine di calcolo numerico: le comuni routine di Matlab, algebra lineare, convoluzioni, FFT, ecc. Ci sono molti package per diversi ambiti: Machine Learning, Visione Artificiale, Image & Video Processing, Speech Recognition, ecc.

I package più importanti per il machine learning sono:

- **nn**: Neural Network, fornisce ogni sorta di modulo per la costruzione di reti neurali, reti neural profonde (deep), regressione lineare, MLP, autoencoders ecc. È il package utilizzato nel progetto. Per topologie di rete bleeding-edge si suggerisce il package **'nnx'**;

- **optim**: package per l'ottimizzazione della discesa del gradiente Fondamentale per avere buone performance nel training della rete;
- **unsup**: è un toolbox per l'apprendimento non supervisionato;
- **Image**: contiene tutte le funzioni atte all'image processing;
- **cunn**: package per utilizzare le reti neurali sfruttando la potenza di calcolo parallelo delle GPU, mediante l'architettura CUDA.

L'architettura del framework è raffigurata in figura B.1.

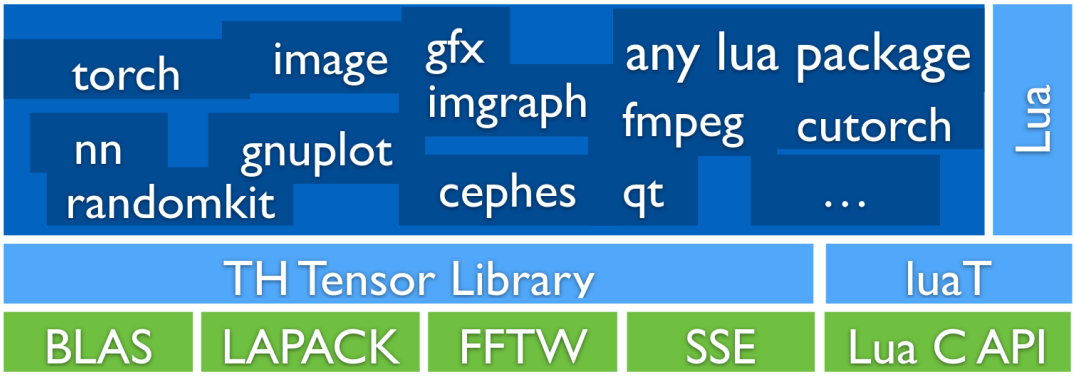

Figura B.1: Architettura del framework Torch

Torch è adottato da una fiorente comunità attiva di ricercatori in diverse università e importanti centri di ricerca come quelli di IBM; il dipartimento di IA di Facebook (FAIR); Google DeepMind prima di passare a TensorFlow nel 2016.

B.2 Utilizzo base per reti neurali

Costruire modelli di reti neurali è una procedura rapida con Torch, ecco alcuni esempi:

```
----------------------------------------------------------
-- simple linear model: logistic regression
----------------------------------------------------------
model:add(nn.Reshape(3*32*32))
model:add(nn.Linear(3*32*32,#classes))

----------------------------------------------------------
-- classic 2-layer fully-connected MLP
----------------------------------------------------------
model:add(nn.Reshape(3*32*32))
model:add(nn.Linear(3*32*32, 1*32*32))
model:add(nn.ReLU())
model:add(nn.Linear(1*32*32, #classes))

----------------------------------------------------------
-- convolutional layer
----------------------------------------------------------
--hyper-parameters
nfeats = 3 --3D input volume
nstates = {16, 64, 128} --output at each level
filtsize = 5 --filter size or kernel
poolsize = 2

```

```
--Here's only the first stage.
--The others look the same except for the nstates you're gonna use

-- filter bank -> squashing -> max pooling
model:add(nn.SpatialConvolutionMM(nfeats, nstates[1], filtsize, filtsize))
model:add(nn.ReLU())
model:add(nn.SpatialMaxPooling(poolsize, poolsize, poolsize, poolsize))
```

B.2.1 Supporto CUDA

CUDA (Compute Unified Device Architecture) è l'architettura di elaborazione in parallelo di NVIDIA che permette netti aumenti delle prestazioni di computing grazie allo sfruttamento della potenza di calcolo delle GPU per operazioni "general purpose". Torch offre un package chiamato `'cunn'` per usufruire di CUDA. Il package è basato su un tensore chiamato `'torch.CudaTensor()'` che altro non è che un normale Tensor che risiede ed utilizza la memoria della DRAM della GPU; tutte le operazioni definite per l'oggetto Tensor sono definite normalmente anche per il CudaTensor, il quale astrae completamente dall'utilizzo della GPU, offrendo un'interfaccia semplice e permettendo di sfruttare gli stessi script che si usano per l'elaborazione CPU. L'unica modifica da apportare, quindi, è cambiare il tipo di tensore.

```
tf = torch.FloatTensor(4,100,100) -- CPU's DRAM
tc = tf:cuda() -- GPU's DRAM
tc:mul() -- run on GPU
res = tc:float() -- res is instantiated on CPU's DRAM

--similarly, after we've built our model
--we can move it to the GPU by doing
model:cuda()
--we also need to compute our loss on GPU
criterion:cuda()

--now we're set, we can train our model on the GPU
--just by following the standard training procedure seen in (Capitolo 4)
```

B.3 ResNet

Una rappresentazione testuale del modello a 18 strati di Residual Network.

```
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> (7) -> (8) -> (9) ->
     output]
  (1): cudnn.SpatialConvolution(3 -> 16, 3x3, 1,1, 1,1) without bias
  (2): nn.SpatialBatchNormalization (4D) (16)
  (3): cudnn.ReLU
  (4): nn.Sequential {
    [input -> (1) -> (2) -> (3) -> output]
    (1): nn.Sequential {
      [input -> (1) -> (2) -> (3) -> output]
      (1): nn.ConcatTable {
        input
          |`-> (1): nn.Sequential {
          |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
          |      (1): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
    without bias
          |      (2): nn.SpatialBatchNormalization (4D) (16)
          |      (3): cudnn.ReLU
          |      (4): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
    without bias
```

```
         |      (5): nn.SpatialBatchNormalization (4D) (16)
         |    }
         `-> (2): nn.Identity
         ... -> output
     }
     (2): nn.CAddTable
     (3): cudnn.ReLU
   }
   (2): nn.Sequential {
     [input -> (1) -> (2) -> (3) -> output]
     (1): nn.ConcatTable {
       input
         |`-> (1): nn.Sequential {
         |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
         |      (1): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
  without bias
         |      (2): nn.SpatialBatchNormalization (4D) (16)
         |      (3): cudnn.ReLU
         |      (4): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
  without bias
         |      (5): nn.SpatialBatchNormalization (4D) (16)
         |    }
         `-> (2): nn.Identity
         ... -> output
     }
     (2): nn.CAddTable
     (3): cudnn.ReLU
   }
   (3): nn.Sequential {
     [input -> (1) -> (2) -> (3) -> output]
     (1): nn.ConcatTable {
       input
         |`-> (1): nn.Sequential {
         |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
         |      (1): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
  without bias
         |      (2): nn.SpatialBatchNormalization (4D) (16)
         |      (3): cudnn.ReLU
         |      (4): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
  without bias
         |      (5): nn.SpatialBatchNormalization (4D) (16)
         |    }
         `-> (2): nn.Identity
         ... -> output
     }
     (2): nn.CAddTable
     (3): cudnn.ReLU
   }
 }
 (5): nn.Sequential {
   [input -> (1) -> (2) -> (3) -> output]
   (1): nn.Sequential {
     [input -> (1) -> (2) -> (3) -> output]
     (1): nn.ConcatTable {
       input
         |`-> (1): nn.Sequential {
         |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
         |      (1): cudnn.SpatialConvolution(16 -> 32, 3x3, 2,2, 1,1)
  without bias
         |      (2): nn.SpatialBatchNormalization (4D) (32)
         |      (3): cudnn.ReLU
         |      (4): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
  without bias
```

```
         |      (5): nn.SpatialBatchNormalization (4D) (32)
         |    }
         `-> (2): nn.Sequential {
               [input -> (1) -> (2) -> output]
               (1): nn.SpatialAveragePooling(1x1, 2,2)
               (2): nn.Concat {
                 input
                   |`-> (1): nn.Identity
                   `-> (2): nn.MulConstant
                   ... -> output
               }
             }
         ... -> output
     }
     (2): nn.CAddTable
     (3): cudnn.ReLU
   }
   (2): nn.Sequential {
     [input -> (1) -> (2) -> (3) -> output]
     (1): nn.ConcatTable {
       input
         |`-> (1): nn.Sequential {
         |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
         |      (1): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
  without bias
         |      (2): nn.SpatialBatchNormalization (4D) (32)
         |      (3): cudnn.ReLU
         |      (4): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
  without bias
         |      (5): nn.SpatialBatchNormalization (4D) (32)
         |    }
         `-> (2): nn.Identity
         ... -> output
     }
     (2): nn.CAddTable
     (3): cudnn.ReLU
   }
   (3): nn.Sequential {
     [input -> (1) -> (2) -> (3) -> output]
     (1): nn.ConcatTable {
       input
         |`-> (1): nn.Sequential {
         |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
         |      (1): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
  without bias
         |      (2): nn.SpatialBatchNormalization (4D) (32)
         |      (3): cudnn.ReLU
         |      (4): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
  without bias
         |      (5): nn.SpatialBatchNormalization (4D) (32)
         |    }
         `-> (2): nn.Identity
         ... -> output
     }
     (2): nn.CAddTable
     (3): cudnn.ReLU
   }
 }
 (6): nn.Sequential {
   [input -> (1) -> (2) -> (3) -> output]
   (1): nn.Sequential {
     [input -> (1) -> (2) -> (3) -> output]
     (1): nn.ConcatTable {
```

```
        input
          |`-> (1): nn.Sequential {
          |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
          |      (1): cudnn.SpatialConvolution(32 -> 64, 3x3, 2,2, 1,1)
    without bias
          |      (2): nn.SpatialBatchNormalization (4D) (64)
          |      (3): cudnn.ReLU
          |      (4): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
    without bias
          |      (5): nn.SpatialBatchNormalization (4D) (64)
          |    }
           `-> (2): nn.Sequential {
                 [input -> (1) -> (2) -> output]
                 (1): nn.SpatialAveragePooling(1x1, 2,2)
                 (2): nn.Concat {
                   input
                     |`-> (1): nn.Identity
                      `-> (2): nn.MulConstant
                      ... -> output
                 }
               }
           ... -> output
      }
      (2): nn.CAddTable
      (3): cudnn.ReLU
    }
    (2): nn.Sequential {
      [input -> (1) -> (2) -> (3) -> output]
      (1): nn.ConcatTable {
        input
          |`-> (1): nn.Sequential {
          |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
          |      (1): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
    without bias
          |      (2): nn.SpatialBatchNormalization (4D) (64)
          |      (3): cudnn.ReLU
          |      (4): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
    without bias
          |      (5): nn.SpatialBatchNormalization (4D) (64)
          |    }
           `-> (2): nn.Identity
           ... -> output
      }
      (2): nn.CAddTable
      (3): cudnn.ReLU
    }
    (3): nn.Sequential {
      [input -> (1) -> (2) -> (3) -> output]
      (1): nn.ConcatTable {
        input
          |`-> (1): nn.Sequential {
          |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
          |      (1): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
    without bias
          |      (2): nn.SpatialBatchNormalization (4D) (64)
          |      (3): cudnn.ReLU
          |      (4): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
    without bias
          |      (5): nn.SpatialBatchNormalization (4D) (64)
          |    }
           `-> (2): nn.Identity
           ... -> output
      }
```

```
      (2): nn.CAddTable
      (3): cudnn.ReLU
    }
  }
  (7): cudnn.SpatialAveragePooling(8x8, 1,1)
  (8): nn.View(64)
  (9): nn.Linear(64 -> 10)
}

=======
\section{Introduzione}
\parencite{WTorch}

\section{Utilizzo base per reti neurali}

\subsection{Supporto CUDA}

\section{ResNet}
Una rappresentazione testuale del modello a 18 strati di Residual Network.

\begin{lstlisting}[language={[5.2]Lua}]
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> (7) -> (8) -> (9) ->
    output]
  (1): cudnn.SpatialConvolution(3 -> 16, 3x3, 1,1, 1,1) without bias
  (2): nn.SpatialBatchNormalization (4D) (16)
  (3): cudnn.ReLU
  (4): nn.Sequential {
    [input -> (1) -> (2) -> (3) -> output]
    (1): nn.Sequential {
      [input -> (1) -> (2) -> (3) -> output]
      (1): nn.ConcatTable {
        input
          |`-> (1): nn.Sequential {
          |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
          |      (1): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
    without bias
          |      (2): nn.SpatialBatchNormalization (4D) (16)
          |      (3): cudnn.ReLU
          |      (4): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
    without bias
          |      (5): nn.SpatialBatchNormalization (4D) (16)
          |    }
           `-> (2): nn.Identity
           ... -> output
      }
      (2): nn.CAddTable
      (3): cudnn.ReLU
    }
    (2): nn.Sequential {
      [input -> (1) -> (2) -> (3) -> output]
      (1): nn.ConcatTable {
        input
          |`-> (1): nn.Sequential {
          |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
          |      (1): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
    without bias
          |      (2): nn.SpatialBatchNormalization (4D) (16)
          |      (3): cudnn.ReLU
          |      (4): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
    without bias
          |      (5): nn.SpatialBatchNormalization (4D) (16)
          |    }
```

```
           `-> (2): nn.Identity
           ... -> output
      }
      (2): nn.CAddTable
      (3): cudnn.ReLU
    }
    (3): nn.Sequential {
      [input -> (1) -> (2) -> (3) -> output]
      (1): nn.ConcatTable {
        input
          |`-> (1): nn.Sequential {
          |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
          |      (1): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
   without bias
          |      (2): nn.SpatialBatchNormalization (4D) (16)
          |      (3): cudnn.ReLU
          |      (4): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
   without bias
          |      (5): nn.SpatialBatchNormalization (4D) (16)
          |    }
           `-> (2): nn.Identity
           ... -> output
      }
      (2): nn.CAddTable
      (3): cudnn.ReLU
    }
  }
  (5): nn.Sequential {
    [input -> (1) -> (2) -> (3) -> output]
    (1): nn.Sequential {
      [input -> (1) -> (2) -> (3) -> output]
      (1): nn.ConcatTable {
        input
          |`-> (1): nn.Sequential {
          |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
          |      (1): cudnn.SpatialConvolution(16 -> 32, 3x3, 2,2, 1,1)
   without bias
          |      (2): nn.SpatialBatchNormalization (4D) (32)
          |      (3): cudnn.ReLU
          |      (4): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
   without bias
          |      (5): nn.SpatialBatchNormalization (4D) (32)
          |    }
           `-> (2): nn.Sequential {
                 [input -> (1) -> (2) -> output]
                 (1): nn.SpatialAveragePooling(1x1, 2,2)
                 (2): nn.Concat {
                   input
                     |`-> (1): nn.Identity
                      `-> (2): nn.MulConstant
                      ... -> output
                 }
               }
           ... -> output
      }
      (2): nn.CAddTable
      (3): cudnn.ReLU
    }
    (2): nn.Sequential {
      [input -> (1) -> (2) -> (3) -> output]
      (1): nn.ConcatTable {
        input
          |`-> (1): nn.Sequential {
```

```
         |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
         |      (1): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
   without bias
         |      (2): nn.SpatialBatchNormalization (4D) (32)
         |      (3): cudnn.ReLU
         |      (4): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
   without bias
         |      (5): nn.SpatialBatchNormalization (4D) (32)
         |    }
          `-> (2): nn.Identity
          ... -> output
     }
     (2): nn.CAddTable
     (3): cudnn.ReLU
   }
   (3): nn.Sequential {
     [input -> (1) -> (2) -> (3) -> output]
     (1): nn.ConcatTable {
       input
         |`-> (1): nn.Sequential {
         |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
         |      (1): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
   without bias
         |      (2): nn.SpatialBatchNormalization (4D) (32)
         |      (3): cudnn.ReLU
         |      (4): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
   without bias
         |      (5): nn.SpatialBatchNormalization (4D) (32)
         |    }
          `-> (2): nn.Identity
          ... -> output
     }
     (2): nn.CAddTable
     (3): cudnn.ReLU
   }
 }
 (6): nn.Sequential {
   [input -> (1) -> (2) -> (3) -> output]
   (1): nn.Sequential {
     [input -> (1) -> (2) -> (3) -> output]
     (1): nn.ConcatTable {
       input
         |`-> (1): nn.Sequential {
         |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
         |      (1): cudnn.SpatialConvolution(32 -> 64, 3x3, 2,2, 1,1)
   without bias
         |      (2): nn.SpatialBatchNormalization (4D) (64)
         |      (3): cudnn.ReLU
         |      (4): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
   without bias
         |      (5): nn.SpatialBatchNormalization (4D) (64)
         |    }
          `-> (2): nn.Sequential {
                [input -> (1) -> (2) -> output]
                (1): nn.SpatialAveragePooling(1x1, 2,2)
                (2): nn.Concat {
                  input
                    |`-> (1): nn.Identity
                     `-> (2): nn.MulConstant
                     ... -> output
                }
              }
          ... -> output
```

```
      }
      (2): nn.CAddTable
      (3): cudnn.ReLU
    }
    (2): nn.Sequential {
      [input -> (1) -> (2) -> (3) -> output]
      (1): nn.ConcatTable {
        input
          |`-> (1): nn.Sequential {
          |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
          |      (1): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
    without bias
          |      (2): nn.SpatialBatchNormalization (4D) (64)
          |      (3): cudnn.ReLU
          |      (4): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
    without bias
          |      (5): nn.SpatialBatchNormalization (4D) (64)
          |    }
          `-> (2): nn.Identity
          ... -> output
      }
      (2): nn.CAddTable
      (3): cudnn.ReLU
    }
    (3): nn.Sequential {
      [input -> (1) -> (2) -> (3) -> output]
      (1): nn.ConcatTable {
        input
          |`-> (1): nn.Sequential {
          |      [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
          |      (1): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
    without bias
          |      (2): nn.SpatialBatchNormalization (4D) (64)
          |      (3): cudnn.ReLU
          |      (4): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
    without bias
          |      (5): nn.SpatialBatchNormalization (4D) (64)
          |    }
          `-> (2): nn.Identity
          ... -> output
      }
      (2): nn.CAddTable
      (3): cudnn.ReLU
    }
  }
  (7): cudnn.SpatialAveragePooling(8x8, 1,1)
  (8): nn.View(64)
  (9): nn.Linear(64 -> 10)
}

>>>>>>> ece01a6c88b2ea9153728cdbf63dcf2c83a18f6a
```

Bibliography

[1] W. Hackbusch, *Tensor spaces and numerical tensor calculus*. Springer Science & Business Media, 2012, vol. 42.

[2] Wikipedia. (2018). https://en.wikipedia.org/w/index.php?title=Tensoroldid=826559352, [Online]. Available: `https://en.wikipedia.org/wiki/Tensor?oldformat=true` (visited on 01/03/2018).

[3] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems", *Computer*, vol. 42, no. 8, 2009.

[4] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Application of dimensionality reduction in recommender system-a case study", Minnesota Univ Minneapolis Dept of Computer Science, Tech. Rep., 2000.

[5] A. Paterek, "Improving regularized singular value decomposition for collaborative filtering", in *Proceedings of KDD cup and workshop*, vol. 2007, 2007, pp. 5–8.

[6] Wikipedia. (2016). Universal Approximation Theorem, [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Low-rank_approximation&oldid=822406390` (visited on 03/01/2018).

[7] C. Tai, T. Xiao, Y. Zhang, X. Wang, *et al.*, "Convolutional neural networks with low-rank regularization", *arXiv preprint arXiv:1511.06067*, 2015.

[8] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation", in *Computer Vision and Pattern Recognition*, 2014.

[9] R. Girshick. (2015). Pyfaster-rcnn implementation, [Online]. Available: `https://github.com/rbgirshick/py-faster-rcnn/blob/master/tools/compress_net.py` (visited on 03/02/2018).

[10] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications", *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[11] alex gossmann. (2016). Understanding tucker decomposition, [Online]. Available: `http://www.alexejgossmann.com/tensor_decomposition_tucker/` (visited on 01/03/2018).

[12] L. De Lathauwer, B. De Moor, and J. Vandewalle, "A multilinear singular value decomposition", *SIAM journal on Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1253–1278, 2000.

[13] (2016). Tensorlab: A matlab package for tensor computations, [Online]. Available: `https://www.tensorlab.net/` (visited on 03/02/2018).

[14] M. Astrid and S.-I. Lee, "Cp-decomposition with tensor power method for convolutional neural networks compression", in *Big Data and Smart Computing (BigComp), 2017 IEEE International Conference on*, IEEE, 2017, pp. 115–118.

[15] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications", *CoRR*, vol. abs/1511.06530, 2015. arXiv: 1511.06530. [Online]. Available: http://arxiv.org/abs/1511.06530.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385.

[17] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions", *arXiv preprint arXiv:1405.3866*, 2014.

[18] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned cp-decomposition", *arXiv preprint arXiv:1412.6553*, 2014.

[19] Y. Shitov, "How hard is the tensor rank?", *arXiv preprint arXiv:1611.01559*, 2016.

[20] C. J. Hillar and L.-H. Lim, "Most tensor problems are np-hard", *Journal of the ACM (JACM)*, vol. 60, no. 6, p. 45, 2013.

[21] J. Håstad, "Tensor rank is np-complete", *Journal of Algorithms*, vol. 11, no. 4, pp. 644–654, 1990.

[22] S. Nakajima, M. Sugiyama, S. D. Babacan, and R. Tomioka, "Global analytic solution of fully-observed variational bayesian matrix factorization", *Journal of Machine Learning Research*, vol. 14, no. Jan, pp. 1–37, 2013.

[23] (2016). Tensorlab: References, [Online]. Available: https://www.tensorlab.net/doc/zreferences.html (visited on 03/02/2018).

[24] T. K. Akar. (2016). The canonical tensor decomposition and its applications to social network analysisits applications to social network analysis, [Online]. Available: http://www.cid.csic.es/homes/rtaqam/web_tricap/pres/acar.pdf (visited on 03/02/2018).

[25] A.-H. Phan, A. Cichocki, and P. Tichavsky, "On fast algorithms for orthogonal tucker decomposition", in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, IEEE, 2014, pp. 6766–6770.

[26] R. Badeau and R. Boyer, "Fast multilinear singular value decomposition for structured tensors", *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 3, pp. 1008–1021, 2008.

[27] (2017). Torch: A scientific computing framework for luajit, [Online]. Available: torch.ch (visited on 08/20/2017).