Alma Mater Studiorum
University of Bologna

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

Master Thesis

Compression of Convolutional Neural Network using Tensor Decomposition

Author:
Ali Alessio Salman

Thesis Supervisor:
Prof. Stefano Mattoccia

Thesis Advisors
Ph.D Matteo Poggi
Ph.D Fabio Tosi

DISI
Dipartimento di Informatica - Scienza e Ingegneria

III Session
2017/2018

Abstract

In recent years, deep neural networks have received lots of attention, been applied to a large number of tasks on which they achieved dramatic accuracy improvements. Among these, Convolutional Neural Networks (CNN) have been the key players for visual recognition tasks. These models combine deep networks with the convolution operation, resulting in millions or even billions of parameters and computations, thus, a fundamental ingredient for their success, have been high-end GPUs.
However, recent developments in fields such as smart wearable and mobile devices, self-driving cars, virtual reality, unmanned drones among many others, have created an unprecedented opportunity for researchers to tackle fundamental challenges with respect to computer vision. For this reason, it is important to deploy deep learning systems on devices with limited resources for real-time applications.
Over the years, the trend for Convolutional Neural Networks saw them grow larger and larger to achieve more impressive performance. For example, the ResNet-50 model[**resnet**], with 50 convolutional layers needs over 95MB memory for storage and much more floating computations to process a single image. Therefore it is crucial, in this context, to investigate new effective ways of both compressing and speeding up these networks.
Recent works in literature have shown that it is indeed possible to shrink a model's parameters without sacrificing its accuracy. This is achievable by exploiting their redundant architecture. A convolutional layer can be seen as a four-dimensional tensor with a specific structure. Because of the latter, new works have proposed tensor decomposition methods as a compression for the CNN's convolutional layers. Following this cue, this thesis intends to deepen the knowledge of the aforementioned applications.
The thesis is structured as follows: Chapter 1 will introduce the problem stated above and the context in which this thesis will operate. Chapter 2 will illustrate the state of the art techniques regarding CNN compression, while Chapter 3 will provide a thorough analysis of the CNNs architecture.
Chapter 4 will then introduce the fundamentals of tensor decomposition and will provide a deep insight into how to apply tensor decomposition on CNNs, ending with a design proposal. Chapter 5 will show, through experiments, how the above-mentioned methods perform on five different models. Finally, Chapter 6 will summarize the work done during the course of this project and will suggest new ideas for future developments.

Abstract

Negli ultimi anni, le reti neurali profonde hanno ricevuto molta attenzione, essendo state applicate in un vasto numero di applicazioni con risultati spesso eccezionali. Tra queste, le reti neurali convoluzionali o Convolutional Neural Networks (CNN), hanno avuto un ruolo chiave per i compiti di riconoscimento visivo. Questi modelli combinano reti neurali profonde con l'operazione di convoluzione, risultando in milioni o addirittura miliardi di parametri e di calcoli; pertanto un ingrediente fondamentale per il loro successo sono state le GPU di fascia alta. Tuttavia, i recenti sviluppi in settori quali dispositivi indossabili intelligenti e mobili, auto a guida autonoma, realtà virtuale, droni senza pilota e molti altri, hanno creato, per i ricercatori, opportunità senza precedenti di affrontare le sfide fondamentali rispetto alla visione artificiale. Per questo motivo, è importante implementare sistemi di apprendimento profondo (deep learning) su dispositivi con risorse limitate per applicazioni in tempo reale.
Ad esempio, il modello ResNet-50 [**resnet**], una rete con 50 strati convoluzionali, ha bisogno di oltre 95 MB di memoria per l'archiviazione e molto di più per i calcoli necessari a classificare una singola immagine. Pertanto, in questo contesto, è fondamentale indagare su nuovi modi efficaci di comprimere e accelerare queste reti.
I recenti lavori in letteratura hanno dimostrato che è possibile ridurre i parametri di un modello senza sacrificarne l'accuratezza. Ciò è realizzabile sfruttando la loro architettura ridondante. Uno strato convoluzionale, infatti, può essere visto come un tensore quadridimensionale con una struttura specifica. Seguendo questo spunto, questa tesi intende approfondire la conoscenza delle applicazioni di cui sopra. La tesi è strutturata come segue: Il Capitolo 1 introdurrà il problema sopra indicato ed il contesto in cui questa tesi opererà. Il Capitolo 2 illustrerà le tecniche avanzate relative alla compressione delle CNN, mentre il Capitolo 3 fornirà un'analisi approfondita dell'architettura delle CNN.
Il Capitolo 4 introdurrà quindi i fondamenti della decomposizione tensoriale e fornirà una visione approfondita di come poterla applicare su una CNN, terminando con una nuova proposta architetturale. Il Capitolo 5 mostrerà, attraverso diversi risultati sperimentali, come i metodi sopra citati si comportino su cinque modelli diversi. Infine, il capitolo 6 riassumerà il lavoro svolto durante il corso di questo progetto lasciando qualche proposta per possibili sviluppi futuri.

Acknowledgements

I would like to thank professor Stefano Mattoccia for his patience to supervise this thesis. It is always interesting to get in touch with the high-level research of its projects which, again, made me discover new exciting topics while also bringing a precious level of satisfaction to my work. In this regard, I wish to thank my two advisors, Matteo Poggi and Fabio Tosi, who happens to be also an exquisite friend.

Contents

List of Figures

List of Tables

List of Abbreviations

SVD	**S**ingular **V**alue **D**ecomposition
HOSVD	**H**igher **O**rder **S**ingular **V**alue **D**ecomposition
CPD	**C**anonical **P**olyadic **D**ecomposition
SGD	**S**tochastic **G**radient **D**escent
NN	**N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
CONV	**Conv**olutional layer
BN	**B**atch **N**ormaization
TD block	**T**ensor **D**ecomposition block

A mio padre, che mi ha insegnato a giocare
A mia zia, che mi ha insegnato la pazienza
A mio nonno, che mi ha insegnato a non prendermi sul serio
A quel cuore che oggi, nonostante tutto, saprebbe battere più forte del mio.

Chapter 1

Introduction

1.1 Computer Vision

Computer Vision (CV) is an interdisciplinary field that deals with how machines can be able to gain high-level understanding of images and videos. Its aim is to enable computers to perform visual tasks in the way the human visual system can do.
More specifically, it deals with image acquisition and processing and image feature extraction. The term *feature* in this context, means that kind of peculiar information embedded in an image that is important to have a semantic understanding of it, thus relevant to solve a specific task.

Computer vision involves a wide range of techniques from a variety of scientific fields. It leverages on image processing (i.e. 2D signal processing) to elaborate images in ways that are convenient for feature extraction. In this regard, the most important operation is the 2D *discrete convolution*.
Once extracted, the features can be used in a plethora of ways: image classification, object detection, video tracking, autonomous driving, visual-guided robotics and so on.

Image features are usually encoded into a *feature vector* that is representative of the whole image; this piece of information is then further processed with specific techniques according to the goal. In order to achieve excellent results in visual tasks, a huge amount of information is required. It is at this point that computer vision embraces *machine learning* and *optimization*, which together define a broad range of methods to process a large number of information and create programs that are able to learn autonomously from data.

1.1.1 Machine learning

There is an extremely broad space of important problems for which no closed-form solution is known, computer vision being one of those. What is the right equation to process pixels of an image and understand its contents? Thus far there is no procedural mathematical solution to this question. For this kind of problems, we often gamble on *machine learning* (ML), which can be broadly defined as enabling

programs to autonomously learn without being explicitly programmed.

With respect to computer vision, machine learning has produced few popular algorithms like *support vector machines* (SVM). Figure 1.1 shows prominent ML algorithms widely studied in 2012, for different research areas. However, the former showed few drawbacks as they had to be manually tuned for each specific domain and were not able to achieve excellent results.

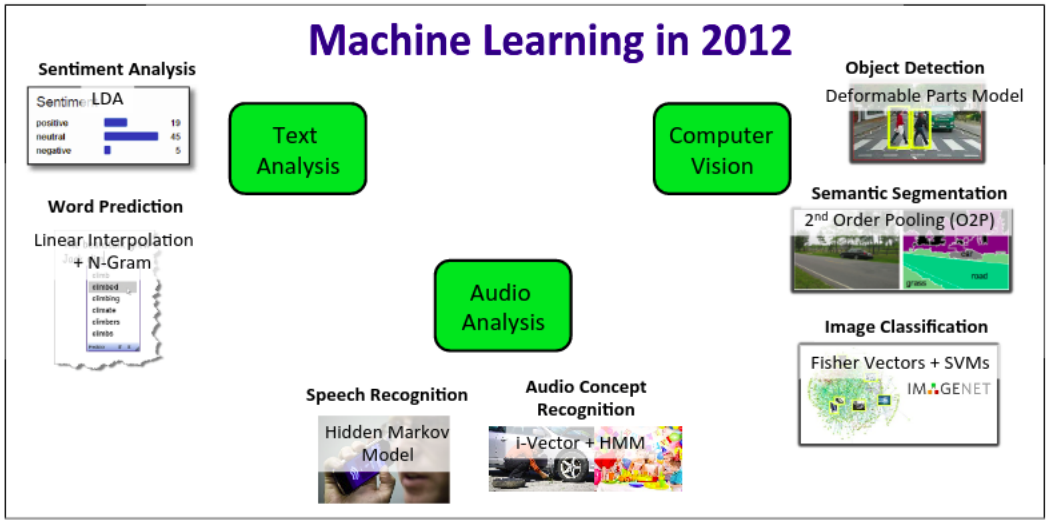


Figure 1.1: For the year 2012, each of these problems was addressed by different methods. Image from [**cnn-design**]

1.2 The advent of Deep Learning

Although Artificial Neural Networks (ANN) are well known since 1960, before the early years of 2000 was impractical to train ANN models, because of their computational requirements. Moreover, there was a fundamental ingredient that was missing to actually train models with many parameters: *data*. Together with the recent explosion of *Big Data*, a class of algorithms that fall under the name of *deep learning* polarized the attention. In fact, these algorithms are able to outperform any other method provided that the proper (large) number of data required for training is available. This is shown in figure 5.3.

Deep Neural Networks (DNNs) are neural networks with more than one hidden layer (possibly many more). These networks are able to model any mathematical function, a peculiarity that makes DNNs powerful and flexible and, furthermore, the appointed candidate for problems where no procedural mathematical solution is known. The difference between regular ANNs and DNNs is depicted in fig. 1.3.

When data became ubiquitous, the machine learning community discovered that deep learning was able to unlock significant achievements and outperform every other

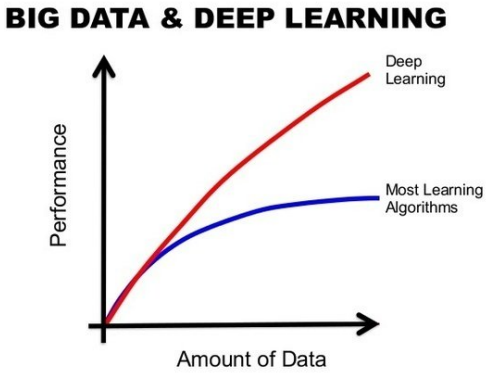


Figure 1.2: The relationship between Deep Learning and Big Data: when a lot of data is available, DL outperforms any other algorithm.

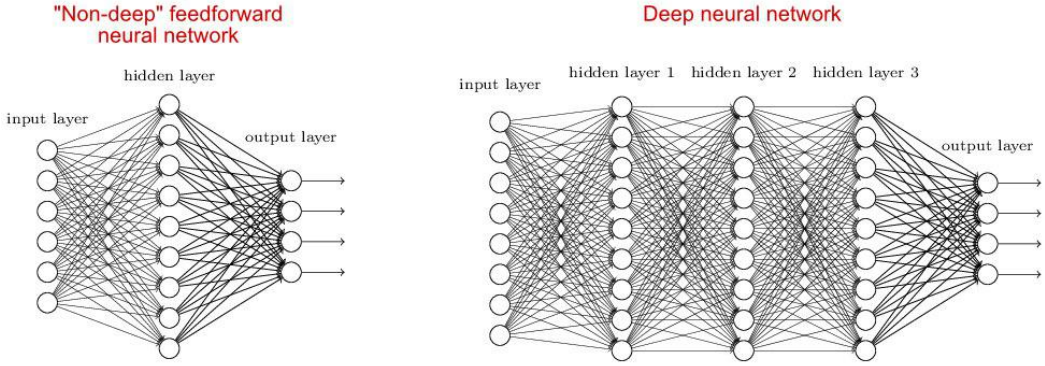


Figure 1.3: An example of ANN and DNN. Each of the circle is called artificial neuron, or just neuron.

algorithm in a wide array of ML problems; a new trend was born.

In figure 1.4, it is possible to appreciate how DL have monopolized many research areas previously addressed by different techniques.

Indeed, Deep Neural Networks (DNNs) have achieved remarkable performance for a wide range of applications, including but not limited to computer vision, natural language processing and speech recognition. In the interest of this thesis, it is important to notice that visual recognition tasks are now based mainly on *Convolutional Neural Networks* (CNNs). These models will be at the core of this research.

Convolutional neural networks, to which we will refer from now on with its abbreviation (CNNs), are essentially DNNs paired up with *convolution*, a mathematical operation that is essential to image processing.
Their main building block is the *convolutional layer*, a stage of the network in which convolution operations are computed between a 3D input tensor and a 3D kernel tensor, in order to extract image information. It is easy to intuit that this operation can be computationally very intensive.

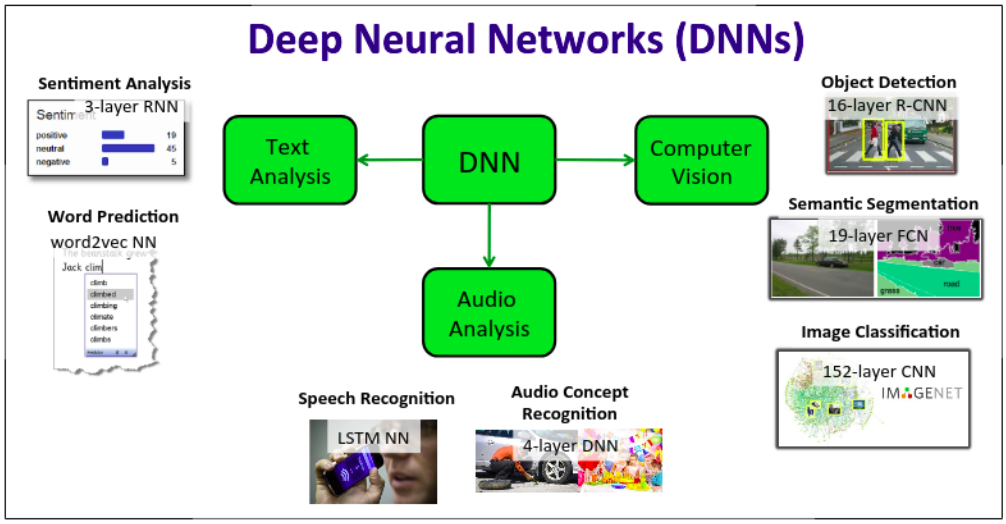


Figure 1.4: Deep Neural Networks delivers the best accuracy for each of these fields. Image from [**cnn-design**]

Table 1.1: Computation complexity and size of state-of-art Convolutional Neural Networks

	Parameters			Computation		
	Size(M)	Conv(%)	FC(%)	FLOPS(G)	Conv(%)	FC(%)
AlexNet	61	3.8	96.2	0.72	91.9	8.1
VGG-S	103	6.3	93.7	2.6	96.3	3.7
VGG16	138	10.6	89.4	15.5	99.2	0.8
NIN	7.6	100	0	1.1	100	0
GoogLeNet	6.9	85.1	14.9	1.6	99.9	0.1
ResNet-18	5.6	100	0	1.8	100	0
ResNet-50	12.2	100	0	3.8	100	0
ResNet-101	21.2	100	0	7.6	100	0

A thorough overview of convolutional neural networks will be presented in Chapter 3.

Big data aside, these breakthroughs go abreast with the increasing computing resources of nowadays. For example, one groundbreaking result in natural image recognition was achieved in 2012 by AlexNet[**alexnet**], a CNN that was trained using multiple graphic processing units (GPUs) on about 1.2M images. Since then, the performance of DNNs has been steadily improving.

For many tasks, CNNs have been reported to be able to outperform humans. The problem however, is that the more these models grow in complexity and performance, the higher the computational resources needed to employ them.

In table 1.1, complexity for state-of-art CNNs are reported. It is notable, how a model like VGG-16, which is widely used, requires more than 500MB of storage and over 15G FLOPs to classify a single image.

1.3 Problem statement

Existing deep convolutional neural network models are computationally expensive and memory intensive, hindering their deployment in devices with low memory resources or in *real-time* applications with strict latency requirements. After all, visual recognition tasks and mobility go side by side.

Given *equivalent accuracy*, a CNN with less parameters would expose a series of advantages:

- **Less overhead when exporting new models**: if we take a look at an important recent trend such as that of autonomous driving, it is immediately clear how big of a benefit this can be. Companies like Tesla periodically transfer new models from their servers to customers' cars, also called over-the-air (OTA) update. This is how Tesla *Autopilot* improves [**tesla**]. An OTA with a network of the size of VGG-16, for instance, would require 500MB of data transfer from the server to the car. Thinner models instead would make frequent OTA updates more feasible, thus being a benefit to security as well.

- **Feasible FPGA and embedded systems deployment**: FPGAs often have less than 10MB of on-chip memory [1]. A sufficiently small model could be stored on an FPGA and used for inference in important applications as *stereo vision* [**mattoccia**]. Additionally, a light model could be deployed on *Application-Specific Integrate Circuits* (ASICs) that could fit on a smaller die.
  This would enable a wide range of applications from unmanned drones to intelligent glasses

- **Energy efficiency**: recent years have seen a spike in the number of mobile devices such as unmanned drones, smartphones, intelligent glasses, etc. Beside the already mentioned computational requirements, these devices would suffer from battery draining, making artificial intelligence application impracticable. Moreover, the aforementioned autonomous driving industry aims at being the most environment-friendly vehicle ever designed. Hence, they need to optimize energy saving on every side.

Clearly, there are several advantages behind the search for smaller CNNs with same accuracy. In addition to the points above, during the course of this project I had the chance to get a closer look to the recent developments of M. Poggi and S. Mattoccia with respect to real-time stereo vision [**poggi-wear**] [**poggi-crosswalk**].

Therefore, a natural thought is to perform model compression and acceleration in deep networks without significantly decreasing the model performance. This is the goal of this thesis.

[1] For example, the Xilinx Vertex-7 FPGA has 8.5 MBytes of on-chip memory and does not provide off-chip one.

Having introduced the motivation behind this work, the next chapter will introduce the current state-of-the-art approaches.

Chapter 2

State of the art

In recent years tremendous progresses have been made with regard to CNN model compression and acceleration. In this chapter an overview of the most significant techniques is provided. Note that details and terminology of CNNs are later explained in chapter 3.

2.1 Model Design vs. Model Compression

Convolutional neural networks compression can be addressed in two main different ways: the first one is to focus on exploring the best practices to build smaller networks without sacrificing accuracy; the other one consists of taking a pre-trained successful model and compress the existent weights to extract a thinner version of it but with same knowledge, and so the accuracy.
This is the first point to keep in mind.
The other distinction is based on which layers to address, namely convolution layers and fully-connected one (layers detailed explanation will follow in chapter 3). This depends on which, among the following, is the priority:

- **acceleration**: most of the computation in CNNs are done in the first convolutional layers; thus these need to be compressed in order to achieve a significant speedup;
- **compression**: most of the parameters are in the last fully connected layers; therefore a reduction on the former is required to save on memory consumption.

For the reasons above, each of the presented schemes will be labeled with the possible area of application.

The approaches proposed so far can be classified into four main categories: *(i)* parameter pruning; *(ii)* low-rank factorization; *(iii)* knowledge distillation; *(iv)* compact network design.

2.2 Parameter pruning and sharing

Pruning was proposed before deep learning became popular, and it has been widely studied in recent years [**brain**]. Based on the assumption that lots of parameters

in deep networks are unimportant or unnecessary, pruning methods try to remove parameters that are not crucial to the model performance.

In this way, networks becomes more sparse and thus show few benefits: the sparsity of the structure acts as a regularization technique against over-fitting, hence improving the model generalization; the computations involved in the pruned parameters are omitted, thus the computational complexity can be reduced; finally, sparse parameters require less disk storage since they can be stored in compressed formats.

Pruning methods can be applied on both model design and compression and since they address single weights, can be applied on both fully connected and convolution layers. Moreover, they can be further classified into three categories: network pruning, model quantization and structural matrix design.

2.2.1 Network Pruning

As aforementioned, the core assumption of pruning is that many parameters are redundant and unnecessary. However, techniques differ on how to assess which parameters are less important than others and the granularity of the actual pruning.

Figure 2.1 shows the various pruning levels for a convolutional layer.

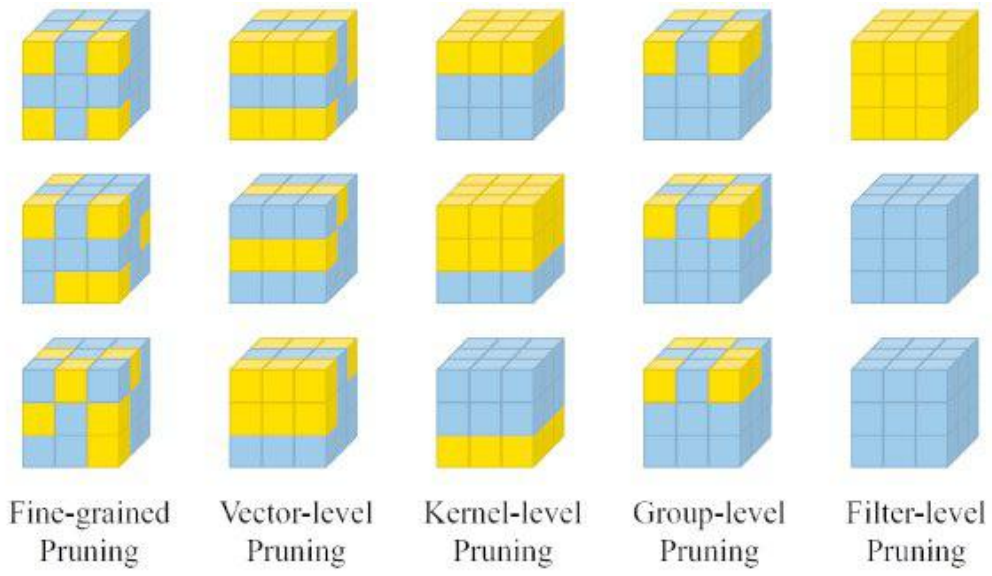


Figure 2.1: Different pruning granularity for a $3 \times 3 \times 3$ convolutional filter. Image from [**survey2018**]

Fine-grained pruning Fine-grained pruning methods remove parameters in an unstructured way, i.e., any unimportant parameters in the convolutional kernels can be pruned without particular constraint. This gives large degrees of freedom but requires more manual parameter tuning.

An early approach to network pruning was the biased weight decay [**pruning-biased**] that consisted in cutting away weights in a magnitude-based fashion. Later works, as the Optimal Brain Damage [**brain-damage**] and the Optimal Brain Surgeon[**brain-surgeon**], assessed the search of unimportant parameters based on the Hessian of the loss function and show better results.
However, it is unaffordable for deep networks to compute the second order derivatives for each weight due to a huge computational complexity.

A recent improvement that enabled to prune uninformative weights in a pre-trained CNN model, was proposed by Han et al. in [**deep-compression**]. Their *"deep compression framework* consists of three important steps:

1. pruning redundant connections and retrained the sparse connections;
2. quantization of the remaining weights;
3. Huffman coding to encode the quantized weights in a loss-less format.

By using the former method, Han et. al managed to compress AlexNet by 35× with no drop in accuracy; this method achieved state-of-the-art The pipeline of the former is shown in figure 2.2

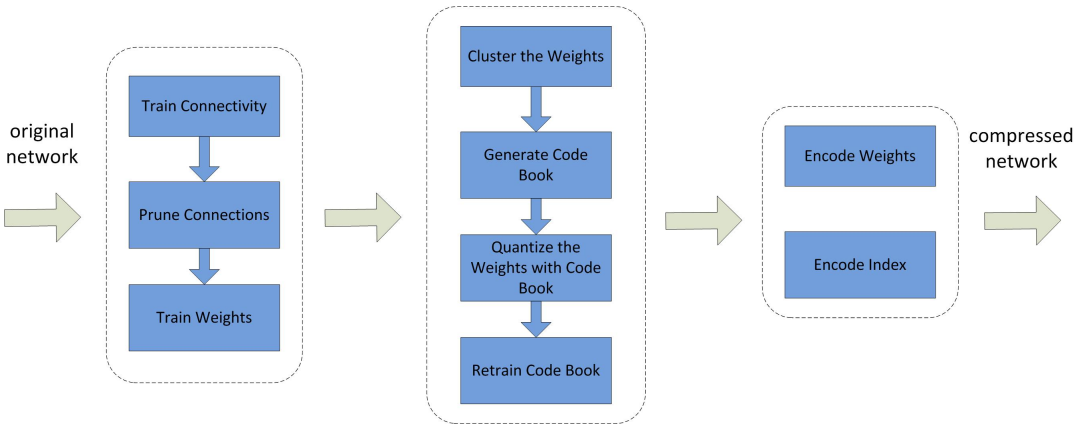


Figure 2.2: The three steps compression method proposed in [**deep-compression**]: pruning, quantization and encoding.

Group-level pruning Group-level pruning apply the same sparse pattern on the filters, so that convolutional filters can be represented as thinned dense matrix. An example of this can be seen in figure 2.3.

Since this type of pruning produces thin dense matrices, convolution operation can be implemented as thinned dense matrix multiplication leveraging on the Basic Linear Algebra Subprograms (BLAS) and hence achieving higher speed-ups (almost linear with the sparsity level).

One way to implement this, is by employing group-sparisity regularizers. As explained in chapter 3, regularizers are added to the loss function in order to penalize too complex models. By applying a group-sparsity regularizer, Lebedev and Lempitsky [**lebedev2**] showed how to train a CNN easily with group-sparsified weights.

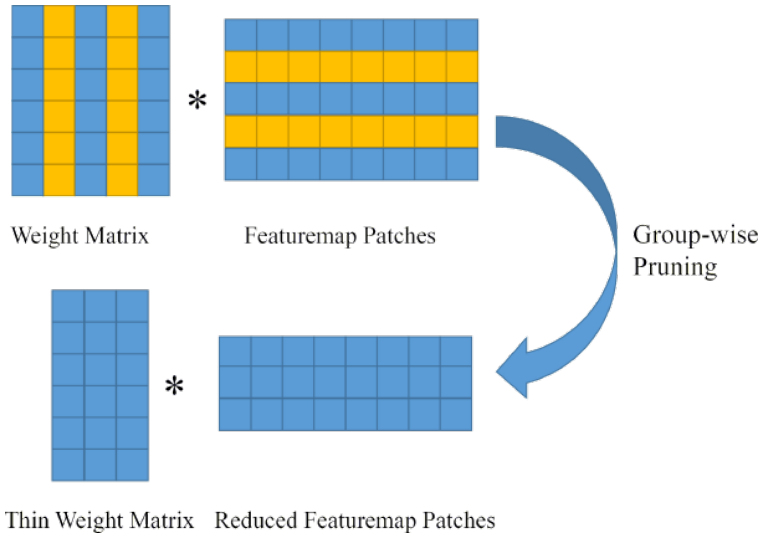


Figure 2.3: An example of group-level pruning, from [**survey2018**].

Filter-level pruning Filter-level pruning, as the name suggests, prunes the individual filters (or channels) of the CNN, thus making the network thinner. After pruning filters on one layer, the following layers' channels are also pruned. For this reason, compared to other pruning, this method is more suited to accelerate deep networks

In order to properly choose the filter channels to prune, custom parameters have to be imposed. For example, it's possible to guide a layer filter pruning by using the next layer's feature map as a guide i.e., by minimizing the reconstruction error on the latter. In this way, the layer will optimize a subset of its filters to obtain the same result as the original model. This strategy has been proposed by [**Luo2017**].

Other methods have always applied similar per-layer constraints.

Considerations

- **Application**: convolutional and fully-connected
- **Drawbacks**: Pruning techniques are elastic and be applied on every layer and model. However, all pruning criteria require a careful manual setup of sensitivity per-layer, which demands fine-tuning of the new imposed parameters. This can be inconvenient in some scenario.

2.2.2 Quantization and Binarization

Network quantization involves compressing the original network by reducing the number of bits required to represent each weights. This strategy is further classifiable into two groups: *scalar and vector* quantization and *fixed-point* quantization.

Scalar and vector quantization Scalar and vector quantization technique has a long history, and it was original used for data compression. Through this method the original data can be represented by two basic elements:

- *a codebook* that contains a set of "quantization centers";
- *a set of quantization codes* used to indicate the assignment of the quantization centers.

Most of the time, the cardinality of quantization centers is far smaller than the number of original parameters.
In addition, the quantization codes can be encoded by lossless encoding methods as Huffman coding, which was also a fundamental gear of the deep compression pipeline mentioned in section 2.2.1. For this reason scalar and vector quantization can achieve high compression ratio and can be applied on both convolutional [**WU2016**] and fully-connected layer [**gong**].

Fixed-point quantization Resource consumption of CNN is not only based on the rough number of weights, but also on the activations and the gradient computation during the backward pass in the training phase. To tackle this, fixed-point quantization methods are divided into:

1. quantization of weights;
2. quantization of activations;
3. quantization of the gradient: these methods are also able to speed-up training, for obvious reasons.

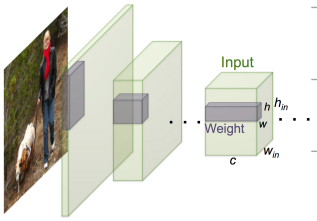
Compression with 8-16-32 bits fixed-point have been experimented with fair results. [CIT NEEDED]. Interestingly, it has been proposed that 16-bit fixed-point was adequate to train small models, while 32-bit fixed-point were required in the case of bigger nets.

In the extreme scenario of 1-bit representation of each weight, that is, *binary weight neural networks*, there are also many works that directly train CNNs with binary weights, for instance, BinaryConnect [**binaryconnect**], BinaryNet [**binarynet**] and XNOR-Networks [**XNOR**].

The main idea is to directly learn binary weights or activations during the model training, so that it is possible to get rid of floating computations once for all.

Among all, XNOR-net achieved remarkable results on the ImageNet dataset, outperforming other quantized nets by a large margin. It reached the same accuracy of AlexNet, while saving 32× memory usage and 58× faster convolution operations. An overview of XNOR-net is reported in figure 2.4

An interesting point made by XNOR-net is that it is the first state-of-the-art network that can be trained on CPUs and can be a candidate for real-time operations. It is indeed already been deployed on smartphone by [**Wxnor-ai**].



	Network Variations	Operations used in Convolution	Memory Saving (Inference)	Computation Saving (Inference)	Accuracy on ImageNet (AlexNet)
Standard Convolution	Real-Value Inputs 0.11 -0.21 ... -0.34 -0.25 0.61 ... 0.52; Real-Value Weights 0.12 -1.2 ... 0.41 -0.2 0.5 ... 0.68	+ , − , ×	1x	1x	%56.7
Binary Weight	Real-Value Inputs 0.11 -0.21 ... -0.34 -0.25 0.61 ... 0.52; Binary Weights 1 -1 ... 1 -1 1 ... 1	+ , −	~32x	~2x	%56.8
BinaryWeight Binary Input (**XNOR-Net**)	Binary Inputs 1 -1 ... -1 -1 1 ... 1; Binary Weights 1 -1 ... 1 -1 1 ... 1	XNOR , bitcount	~32x	~58x	%44.2

Figure 2.4: XNOR-Net overview with computation speedups and memory savings.

Considerations

- **Application**: convolutional and fully-connected
- **Drawbacks** the accuracy of said binary nets is significantly lowered when dealing with large CNNs such as GoogLeNet. Besides, the approximation techniques for binary weights don't yet take into account the effects on accuracy loss

.

2.3 Knowledge Distillation

Knowledge distillation is different from other methods since it tries to built a smaller model that can have a totally different architecture but same accuracy on the same problem domain.
Its mechanism resembles much more what happens in the human world: in fact, it works by transferring the knowledge from a large network, *the teacher*, to a much smaller one i.e., *the student*. Hence, it is also called teacher-student network.

Originally introduced by Caruana et al. [**caruana**] only on shallow models, it has been now reproposed. By utilizing what is known as the *"dark knowledge"* transferred from the teacher network, the smaller model can achieve higher accuracy than training merely by the "raw" class labels.

As of 2018, three main works have paved the road to this interesting application:

1. Hinton et. al [**hinton-KD**] proposed to improve the student network training with the *softmax* layer's output of the teacher, i.e. the (log)probabilities of each class. This scheme is reported in figure 2.5.

2. Following this line, Romero et al. [**romero-KD**] proposed *FitNets* as teachers for thinner and deeper networks. They went further by training the student not only on the softmax probabilities of the teacher but also on its intermediate feature maps.

3. Finally, in [**greci-KD**] Zagorukyo and Komodakis proposed an even more interesting approach: the student network was trained on the *attention maps*. Attention maps are defined as those feature maps where the network held most of its knowledge. Once these maps are found, we can transfer only their content to the student, resulting in an even thinner model. This new branch goes with the name of *Attention Transfer* (AT).

Moreover, the teacher can be an ensemble of models [**Wensemble**] with modules called "specialists" who focus on only training the student on sub-parts of the problem. Noticeably, these methods did not show the need for regularization techniques such as dropout (see Chapter 3) since the transferred knowledge, in a sense, is already filtered and optimized.

Considerations

- **Application**: training from scratch only.
- **Drawbacks**: Training with KD technique is way less costly. However, currently these approaches can only be applied to classification tasks with softmax loss functions. Another drawback is that certain assumptions are too strong to make the performance comparable to other methods.

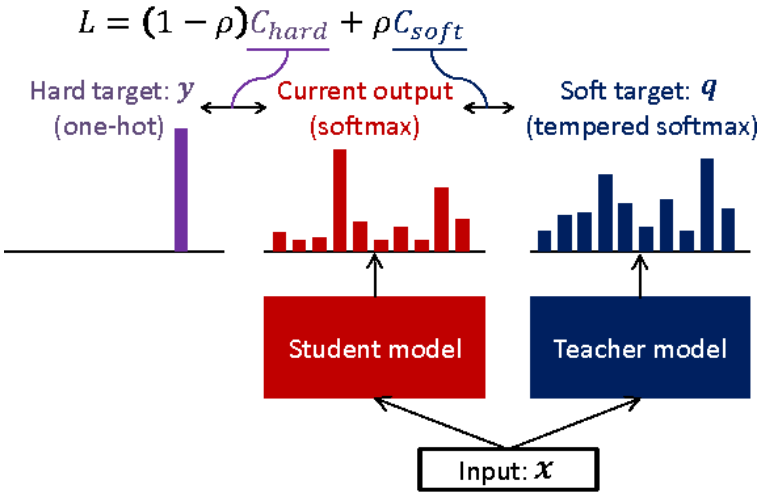


Figure 2.5: Teacher-student scheme: the student learn on a mix of

2.4 Compact Network Design

While other methods try to optimize execution and memory consumption for a given model without changing its architecture, compact network design aims at designing better models in the first place.
It is based on some empirical principles that come out over the years, through different breakthroughs in the field. A summary of the former is listed here, while an comprehensive explanation is provided in Chapter 3, where CNN architecture is discussed.

The elements of an optimal CNN design are:

- *micro-architecture design of the network building blocks*
- $[1 \times 1]$ *convolutions*
- *network branching*
- *depthwise separable convolution*

Considerations

- **Application**: training from scratch, most of the time. However, these principles can be combined with low-rank approximations methods when substituing a layer with a new approximated block, that could follow these rules.
- **Drawbacks**: Since this method only aims at building better models, it does not make much sense to compare it to the others in terms of advantages and pifalls. It is just a general framework of good practices to be kept in mind

2.5 Low-rank Factorization

The convolutional kernel of a convolution layer $W \in R^{s \times t \times d \times d}$ is a 4-D tensor, with the four dimension corresponding to the number of input and output channels and the

kernel size respectively (here squared for simplicity). Low rank factorization methods try to find a way to determine an approximate tensor $\hat{W}$ that is close to W but has far less parameters, facilitating the computations.

This can be done in a variety of ways that all plays on how many layers we want to arrange the decomposed tensor $\hat{W}$. For instance, it can be arranged in 2 layers holding 2 dimensions of the original one, or in 4 by one, or even in asymettric 3 layers decomposition. The former is possible by manipulating the tensor and utilizing different decomposition techniques, such as SVD [**zhang2015SVD**], Tucker [**Tucker-mobile**], CPD [**lebedev**] and others.

This approach can be embeddable in a decomposition pipeline as:

- select an over-parametrized layer;
- compress the layer with one of the mentioned algorithms;
- embed the decomposed layer into the model instead of the old one;
- perform some iteration of fine-tuning to recover the approximation error;
- start again.

As these methods will be seen in details in chapter 4, we can omit, for the moment, the details.

Considerations

- **Application**: pre-trained models and new architectures, both FC and convolutional layers.
- **Drawbacks**: As we will see, these methods are good candidates to be implemented in a pipeline, i.e. their integration is straightforward. However, tensor decomposition algorithms are expensive and not so easy to play around at first. Besides, the compression happens layer by layer and therefore does not take into account the loss of information related to another layer. Furthermore, training could take many iterations before convergence.

2.6 Other methods

There are several other methods to address this new trend like *transferred convolutional filters*, *dynamic capacity networks*, etc. For an exhaustive reference lists, please refer to [**survey2017**].

Remarkable trends are also showing up on the hardware side of the challenge, with FPGA/ASIC-based accelerators re-gaining popularity over GPUs, aiming at real-time applications, low energy consumption and high-throughput optimization

This is important for the scope of this thesis, as the final goal of this project is to deploy optimized models on FPGA, Intel Movidius or Raspberry Pi.

2.7 Discussion

The presented methods are orthogonal. It is possible to combine two or three of them together to maximize the compression by applying the best suited technique on a specific scenario. For example, in the case of object detection where both convolution and fully connected layers are needed, it is feasible to compress the former with low-rank factorization and the latter with pruning. Moreover, when a small accuracy drop is tolerable, quantization can also be applied as a final step after the aforementioned methods.

One pitfall, however, is that most of these methods require non trivial hyper-parameters configurations. Hence, it is not straightforward to combine them as one could get stuck in the hyper-parameters tuning loop before finding the best setup. This is probably due to the early age of these techniques.

As for this work, since the future goal will be to optimize custom pre-trained models of the VisionLab of the University of Bologna, only two methods are feasible candidates: network pruning and low-rank factorization. Among these, the latter is the only one that provides a possible end-to-end pipeline to the problem.

Therefore, this thesis will focus on the low-rank factorization technique; specifically, on further developing *tensor decomposition methods*, which do not seem to be fully explored yet and could have promising applications.

Chapter 3

Convolutional Neural Networks

In this chapter an overview on the convolutional neural networks is presented. Being a wide subject, an in-depth theoretical discussion would account only for the content of this thesis, which is why the topics are introduced with the aim of having a smattering to understand the applications developed in the following chapters.

3.1 Introduction

The convolutional neural networks, to which we will refer from now on with the abbreviation *CNN* are an evolution of the regular *deep artificial networks* characterized by a particular architecture extremely advantageous for visual tasks which made them, over the years, very effective and popular. They were inspired by the biological research of Hubel and Wiesel who, by studying cats' brains, had discovered that their visual cortex contained a complex organized cell structure. The latter were sensitive to small local portions of the visual field, called *receptive fields*. Receptive fields acts as perfect local filters to understand the local correlation of the objects in an image. As these systems are the most efficient in nature for image comprehension, researchers have thus attempted to simulate them.

3.2 Architecture overview

CNNs are deep neural networks consisting of several layers that act as extractors of the features and a fully-connected network at the end, which acts as a classifier, as shown in the figure 3.1.

The stages that perform feature extraction are called *convolutional* layers, they are the fundamental building blocks of CNNs to which they give the name. The latter are usually followed by a non-linear *activation function* and a *pooling* layer. A classical example of a CNN is represented in 3.2.

Every type of layer has a different goal that fall into four main categories: (i) feature extraction; (ii) non-linear activations; (iii) parameters reduction and skimming. Namely, while convolutional layers search for features of image features, pooling layers squeeze the input to reduce the otherwise huge number of parameters of a CNN;

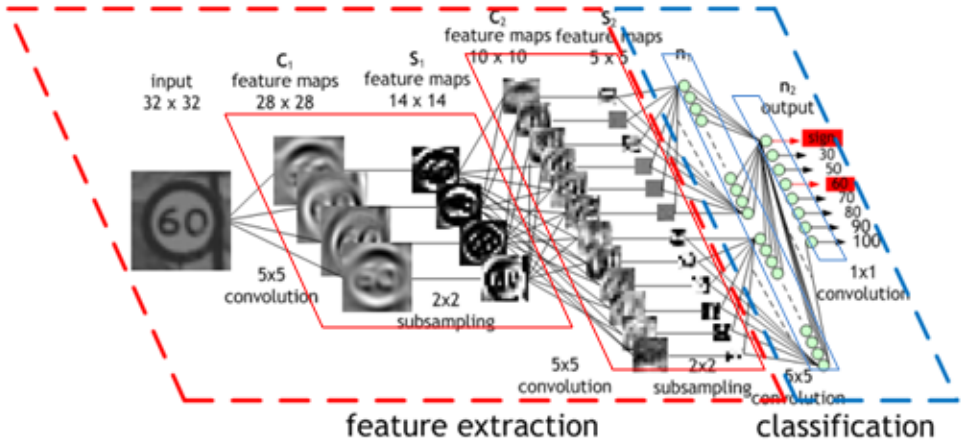


Figure 3.1: Architecture of a CNN that classifies road signs: the image highlights the division between the layers that act as a feature extractor and the final classifier

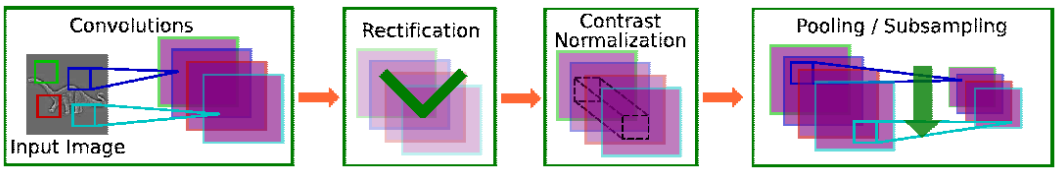


Figure 3.2: Typical layers of a convolutional neural networks

non-linear units serves to enhance the strongest features and weaken the less important ones. i.e. those that have stimulated the neurons less (it is said that they act as "squashing").

From figure 3.1, we can also note that to each input image correspond, in the various layers, different groups of output images, which are called *feature maps*. Feature maps are the result of the convolution operation carried out through a filter bank (also called kernels), which are nothing more than matrices with useful values to search for certain characteristics in the images.
Finally, after the convolutional steps, the feature maps are "flattened" in vectors and entrusted to a "classical" neural network (like a multi-layer perceptron, support vector machines, ...) that performs the final classification.

The number of convolution stages is arbitrary. Initially, when CNN became famous thanks to Y. LeCun, who trained a CNN called *"LeNet5"* to recognize digits [**lenet**], this number ranged from 2 to 4. In 2012, Alex Krizhevsky et al [**imagenet2012**] trained a CNN which consists of 5 convolution layers, 60 million parameters and 650 thousand neurons. They obtained the lowest error rate by far on the ImageNet ILSVRC-2010 dataset, containing 1.2 million images divided into 1000 categories.
Since then, things have evolved with terrific speed, and the same ImageNet challenge of 2015 was won by a model with 152 layers [**resnet**].

As CNN models grow deeper and wider, the need for compression strategies have become more and more important.

r

3.2.1 Convolutional layer

To fully understand what actually happens inside a CNN, it is necessary to introduce the concept of convolution between matrices, and to understand how this is important to process a digital image through a filter.

3.2.2 Spatial 2D convolution

Originally, convolution is defined in functional analysis as a mathematical operator on two functions f and g that produces a third function, that measures the integral of the pointwise multiplication of f and g translated by a certain amount τ.

Formally:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \tag{3.1}$$

A digital image can be considered as a matrix A of $I \times J$ real or discrete values. Each matrix value is called *pixel* and its indices are also called coordinates: each pixel $A(i, j)$ represents the intensity at the position indicated by the indices.
A "filter" or "kernel" is a small matrix used to apply a certain transformation to an image e.g., blurring, sharpening, edge detection and more. The transformation is applied by means of a convolution operation between the input image and the filter. Thus, images and filters can be modeled as 2D signals. Moreover, images and filters are finite supports and hence the integral can be approximated by a finite summation. Therefore, the convolution is discrete and can be defined as:

$$Y(i, j) = X * h = \sum_{u=-k}^{k} \sum_{v=-k}^{k} x(i, j)h(i-u, j-v) \tag{3.2}$$

Each pixel of $Y(i, j)$ is the result of a weighed sum the sub-region that is centered in the pixel indicated by the coordinates i, j. Intuitively, a 2D spatial convolution consists in sliding a window of size $2k + 1 \times 2k + 1$ on the image X and compute the operation indicated in 3.2 for each pixel centered on the window.

An example of convolution is represented in figure 3.3.

In the convolutional layers, a convolution operation is performed between the input image(s) and an arbitrary number K of filters. These filters have particular values, called *weights*, best suited to enhance certain characteristics of the inputs.
The basic idea is that initially these weights are chosen according to some random distribution and then they are improved at each iteration by the *backpropagation* algorithm [**backprop**].
In this way, the network trains its filters to extract the most important features of the training set examples; thus by changing training set the values of the filters will be different. For example, the weights of a network trained on images of vertical posts

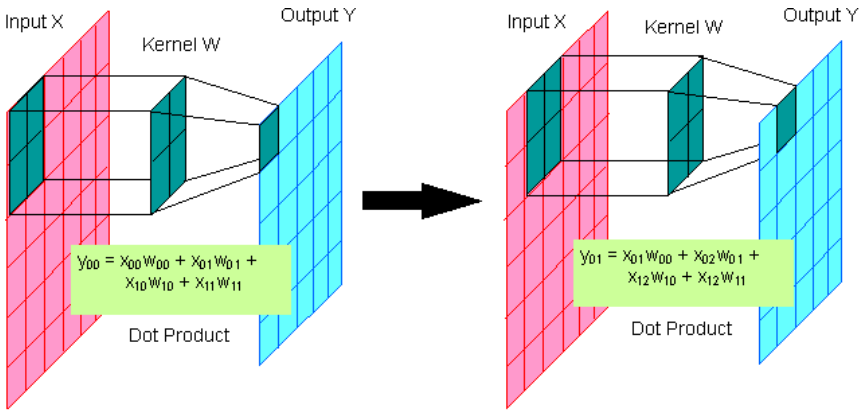


Figure 3.3: Convolution with a kernel: first two steps

will be different from one trained on images of cars; in the first case the values will be calibrated to recognize long vertical orientations, while in the second to recognize more complex rectangular shapes. Therefore, in CNNs the actual learning accumulates in the convolutional filter banks. Artificial *neurons* in these networks, must be understood as the individual filters.

There are several so called *"hyperparameters"* that have to be set manually in the convolutional layers:

1. the filter size F: also called *receptive field*. Each filter searches for a specific feature in a local area of the image, thus the filter size is the receptive field of the single neuron. They typically are of size 3×3, 5×5 or 7×7.

2. The K number of filters: for each layer, this value defines the depth of the convolution operation output. In fact, by placing the feature maps on top of each other, you get a cube in which each "slice" is the result of the operation between the input image and the corresponding filter. The depth of this cube depends precisely on the number of filters.

3. The *"stride"* S: defines of how many pixels the convolution filter slides at each step. For instance, if the horizontal stride is set to 2, the filter will move towards the x-axis of 2 pixels at a time, skipping one column each time and thus producing a smaller output.

4. The *"padding"* P: defines the measure with which we want to add pad to the input with zeros to preserve the same size on the output. In general, when stride $S = 1$, a value of $P = (F - 1)/2$ ensures that the output will be of the same size as the input.

When processing images with CNNs, inputs are typically three-dimensional, characterized by the height H_1, the amplitude W_1 and the number of input channels D_1. Knowing the parameters specified above, we can calculate the size of the output of a

convolution layer:

$$H_2 = (H_1 - F + 2P)/S + 1$$
$$W_2 = (H_1 - F + 2P)/S + 1$$
$$D_2 = K$$

In this regard, we can observe the "volume of neurons" of the first convolution layer in figure 3.4. Each neuron is said to be *locally connected*, i.e. spatially connected only to one local region of the input but to the whole depth.

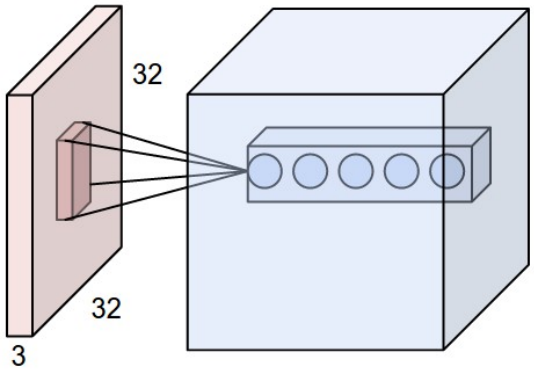


Figure 3.4: Each neuron is connected to only 1 local region of the input but in total depth (i.e. the three color channels in thi). The depth of the output is given by the K number of filters, in this case 5

Note that there are five neurons along the depth and each one looks at the same region of the input but with different weights. In fact, each filter map of the volume of neurons, has its own set of weights. The former is particularly useful when searching for different features on the same region of the input. This is best showed in figure 3.5.

On the other side, each neuron of the same filter map share the same set of weights. This important property is called *parameter sharing* and has the advantage of reducing the number of parameters by a significant factor, saving a lot of computation and memory. This is also depicted in 3.5.

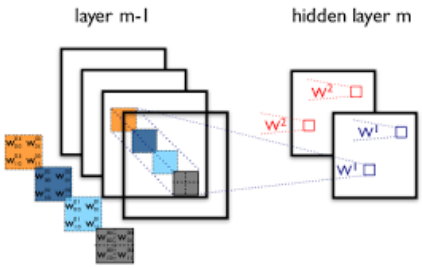


Figure 3.5: Each kernel of the same map shares the same weights, while different feature maps have different weights in order to search for different type of features.

Convolutional layers show interesting properties. Firstly, they are translation-invariant: in fact, if a feature in an image is translated, it will be translated by the same amount in the feature map, but will still be detected since each neuron in that map share the same weights. Also, feature maps will remain unchanged elsewhere. This property is the basis to its robustness with respect to translations and distortions of the input image.

secondly, by lining up several convolution layers, a CNN is able to have a more "abtract" (i.e. semantic) understanding of the image. The first convolution layer deals with extracting features directly from the raw pixels of the image and it stores them in the feature maps. This output then becomes the input of a subsequent convolution level, which will do a second extraction of the characteristics, combining the information of the previous layer. From this multi-level abstraction comes a greater understanding of the image features.

3.2.3 Convolution complexity

In order to understand the complexity of a convolutional layer in terms of multiplications-addictions (mult-adds) is better to break down what happens exactly in a convolutional forward pass with an example. Let K be a squared kernel of size 3×3, U an input feature map of size $12 \times 12 \times 16$ where "12" is the spatial dimension of the input and "16" its channel depth, and V an output feature map of depth 32.

Each of the 16 channels of U is traversed by 32 3×3 kernels resulting in 512 ($16x32$) feature maps. Next, they will be merged in 1 feature map out of every input channel by adding them up. That means there will be 32 sums and thus the output size will be of 32, as said above.

More generally, given a squared kernel of size $d \times d$, an input feature map of size $H \times W \times S$ where H and W are the spatial dimension of the input and S its depth, and an output feature map of depth T, the number of multiplications-additions operation of a *single* convolutional layer amounts to:

$$S \cdot d \cdot d \cdot T \tag{3.3}$$

It is important to remind that this calculations come from the fact that *all* input maps are convolved with the kernel T times i.e., once for *each* output channel.

By multiplying the number of mult-adds by the numner of CONV layers of a whole CNN model, it becomes clear how computationally heavy these layers can be. That is why, over the years, different designs of convolutional layers came up to tackle this issue.

The most important of the latters will be presented later in this chapter.

3.2.4 Separable convolution

A separable convolution defines, in an image processing context, a convolution that can be split in multiple steps, by decomposing its kernel in vector components according to particular algebraic properties, like singular value decomposition.

Every 2D matrix, as a matter of fact, can be decomposed into two vectors with optimal accuracy.This makes possible to compute two 1D convolution with the two vectors instead of one 2D convolution with the whole kernel. More details on matrix and tensor decomposition are in chapter 4.

Example: Sobel filters In order to make things simpler, we can use as an example the famous Sobel filter, which is often used in image processing. The Sobel filter for y axis is defined as:

$$\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3.4}$$

Turns out the above filter can be factorized into two vector components, which when multiplied together through an outer product give back the original Sobel:

$$[1, 0, -1] \otimes [1, 2, -1]^T = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

This operation requires 6 multiplications instead of 9.

Applying this pattern to the kernels of convolutional layers, it is possible to speedup computation and save on memory usage. More precisely, given a squared kernel of size $d \times d$, this intuition make the complexity drop from $O(d^2)$ to $O(d)$.

3.2.5 ReLU

Over the years, various non-linear activation functions have been tried with CNNs, and the *Rectified Linear Unit* (ReLU) has been firmly established over the years. The ReLU is likely to be the most similar to be the biological activation mode of our neurons [**Relu**], and is defined as:

$$f(x) = max(0, x)$$

Y. Few years ago, Yann LeCun stated that ReLU is unexpectedly *"the most important single element of the entire architecture for a recognition system"*.

This may be due to 2 reasons:

1. the polarity of the features is very often irrelevant to recognize objects;
2. the ReLU avoids that when running pooling (section 3.2.6) two features that are both important but opposite polarities cancel each other out.

3.2.6 Pooling layer

Another property that is needed in order to improve the results in machine vision, is the recognition of the important features regardless of their position in the image, as the goal is to strengthen the effectiveness against the translations and distortions.

This can be achieved by decreasing the spatial resolution of the image, which also favors a greater speed of computation and it is, at the same time, a countermeasure against *overfitting*, since it reduces the number of parameters. The pooling layer gets N images of a specific resolution R as input and produces an output that of $R/4$ i.e., it drops 75% of the input. In fact, the most common form of pooling layer uses 2x2 filters, which divide the image into 4 non-overlapping regions of pixels and for each of them pick only one pixel.

The criteria employed to select the winning pixel are vary:

- average pooling: the output is the average of the pixel values of the pool;
- median pooling: the output is the median of the pixel values of the pool;
- LP-pooling: the output is the p-norm of the pixel matrix;
- max pooling: the output is the pixel with the highest value.

Among these, the one that proved to be the most effective is *max pooling* [**Wcs231**], figure 3.6.

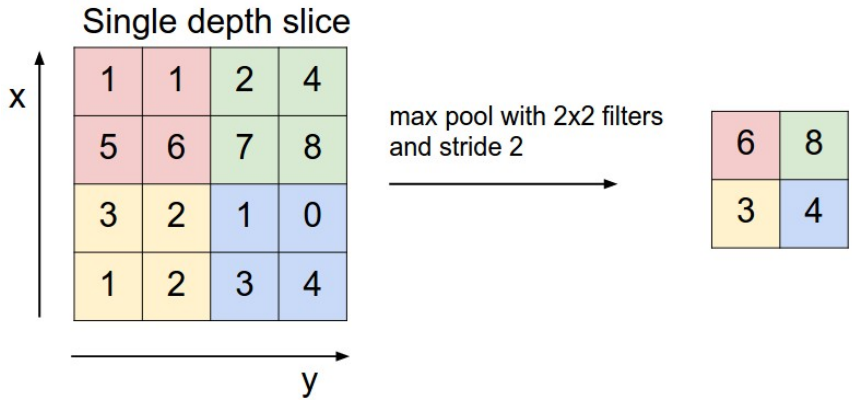


Figure 3.6: Max pooling: the output image will have 1/4 of the input pixels.

Through the network it is possible to gradually have a higher number of feature maps and therefore the richness of the representation of the features; and a decrease in the input resolution. These factors combined together give a strong degree of invariance to the geometric transformations of the input.

3.2.7 Fully connected layer (FC)

In the fully connected layer, all output matrices from convolution layers are flattened and given as a vector input to a fully connected neural network that will act as a classifier. The calculations in this final part are no more than matrix-multiplications plus a bias. By stacking three or more of them together we will obtain what is known as a *multi-layer percepetron*.

In figure 3.7, a CNN can be observed in the "act" of classifying a car image. The network filters are displayed during all the processing levels of the input, and then they terminate in a fully connected layer that outputs a probability. This probability is then translated into a score, from which the winning class is chosen.

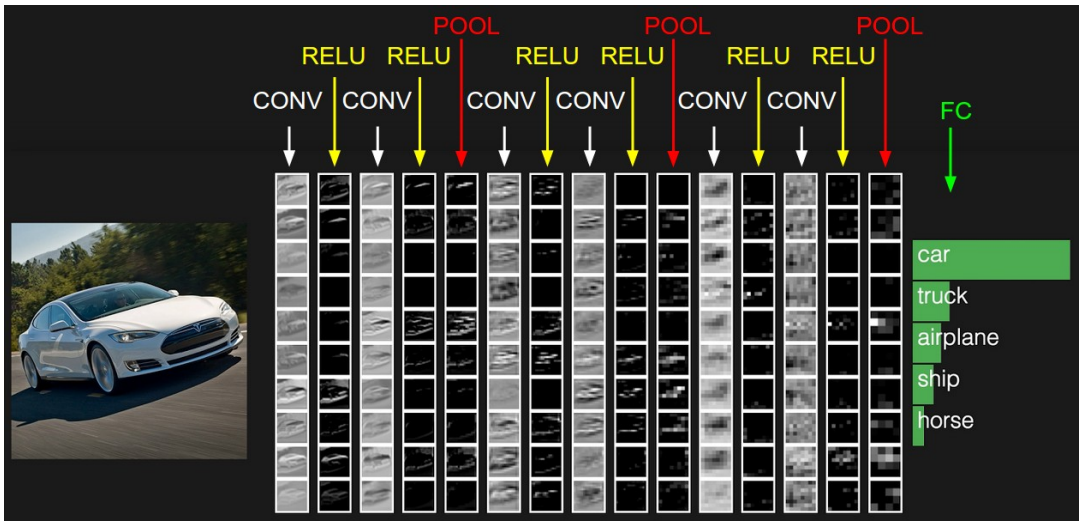


Figure 3.7: Typical CNN in a classification task; the winning class is the one with the highest probability, indicated at the end

Conversion of FC layers

It has been noted recently that, except for the mode of connection, these neurons are functionally identical to those of convolution layers (both compute matrix multiplications). Thus, FC layers can be replaced with convolution layers that have a receptive field equal to the resolution of the images of the input [**WCS231layer**].

This conversion has two main advantages:

- *efficiency*: all deep learning frameworks implements significant optimizations for the convolution operation, since convolutions are very commonly used. That could not be the case for less frequently used operations like matrix multiplications.

- *input dimensions*: $[1 \times 1]$ convolutions enable to use image of different dimensions during testing. In fact, the convolution operation will slide on the whole image, one pixel at a time, which is impossible to do with FC layers. Therefore, in case the input image is bigger than those expected, the output of the

network would be of size O_w, O_h, M (instead of a simple $(1, 1, M)$). Thus, the output will contain a map of probabilities with spatial references. This trick has been particularly useful in CNNs for object detection such as R-CNN and its successors [**rcnn**].

In Figure 3.12 the complete architecture of a CNN is portrayed. It's noticeable how the resolution of the image is reduced to each pooling layer (also called subsampling) and how each pixel of the feature maps derives from the receptive field on the set of all the feature maps of the previous level.

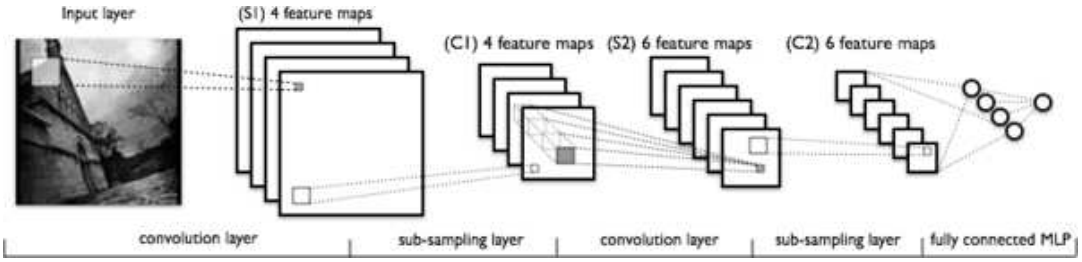


Figure 3.8: Architecture of a CNN

Summary Convnets expose a plethora of unique characteristics that made them the state of the art for computer vision. A quick recap of CNNs key points follows:

- *pipeline architecture*: in the general case CNNs are a sequence of CONV-RELU-(POOL) stacked together with a final linear classifier;
- *parameter sharing*: in order to reduce the otherwise huge number of parameters, each neuron from the same feature map share the same set of weights;
- *local connectivity*: each neuron is connected only to a small patch of the image; the size of this image is called *receptive field*;
- *feature extraction*: lining up convolutional layers plus non-linear units enable CNNs to capture more abstract feature at every stage, building up a semantic understanding of the image;
- *translation-invariance*: thanks to parameter sharing and local connectivity features are captured even if they are translated (in the input images);

3.3 CNN design

In order to tackle a wide range of challenges, CNNs have become more and more complex in recent years. CNN comprise an enourmous design space, with numerous options for micro-architectures (see 3.3.3), solvers and all the different hyperparameters; according to [**cnn-design**], a small region of the CNN design space contains 30 billion different CNN architectures. Hence, it must not be surprising that the research community has been continuously discovering patterns that leads to more powerful models.

Some of these significant milestones are introduced in this section.

3.3.1 One by one convolution layer

One by one convolution was first introduced in the architecture of the popular *"Network-In-Network"* [**NIN**], whose goal was to going "deeper" with convolutions. As already mentioned, stacked convolutional layers build up a semantic understanding of the image. Over the years, many experiments have shown how, in general ,adding more layers lead to better results.

However, convolution operations are costly, expecially when computed on a high number of filters like in convolutional layers. The intuition behind NIN was that [1×1 convolution acts as a trick to make the network deeper while also being cheap. In fact, the complexity in this scenario drops down to $O(ST)$, with S and T being the number of input and output maps respectively.

In this regard, it is interesting to notice that one by one convolutions are used to perform *dimensionality reduction*: if we set $T < S$, the number of output channels will be reduced while the image size and features will be preserved.
Thus, it also acts as a feature pooling technique across the channels i.e., it reduces the number of channels while enhancing the strongest features, just like spatial pooling does on image patches (spatial pooling will be introduced in more details in section 3.2.6).

Practically, one by one convolutions are no more than regular matrix multiplications (like in fully-connected layers) that perform linear re-combinations of the input pixel and thus are fast. It must be noted though, that even if the spatial size of the kernel is [$1x1$] these multiplications are not a simple point-wise scaling, because they perform a dot product on one pixel but over the whole depth of the input channels.

There is more to it than just a sum of features across channels, though. One by one convolutions act like *"coordinate-dependent transformation in the filter space"* [**NIN**]. It is important to point out that while this transformation is strictly linear, it is most often followed by non-linear activation layers. Therefore, this transformation is learned through gradient descent during training while also being robust against

over-fitting thanks to the reduced number of parameters.

Looking at it from a wider point of view, one by one convolution builds up a mini deep neural network whose output is fed into the rest of the network. Hence the name Network-In-Network.

As it will be explained in chapter 4, one by one convolution are one of the fundamentals to perform CNN compression through tensor decomposition. In fact, in order to boast a significant factor of compression on ordinary CNN structures, some tensor decomposition techniques pair this design strategy with the intuition of separable kernels described in section **??**.

3.3.2 Depth-wise convolution layer

Depthwise separable convolutions perform spatial convolution while keeping the channels separate and then follow with a depthwise $[1 \times 1]$ convolution. This convolution is often erroneously called just *"separable"* which collides with the definition given in 3.2.4.

Figure 3.9 shows the structure of a depthwise convolution.

Recalling the example in section 3.2.3, it gets evident how these layers give an important reduction factor over standard convolution.

For a depthwise separable convolution, the 16 input channels are traversed with only 1 3×3 kernel each, giving in output 16 feature maps instead of 32. Now, before any merge happens, these 16 feature maps are traversed with 32 1×1 convolutions each and only then are added together. This results in 656 $(16 \times 3 \times 3 + 16 \times 32 \times 1 \times 1)$ parameters opposed to the 4608 $(16 \times 32 \times 3 \times 3)$ parameters from the example of section 3.2.3.

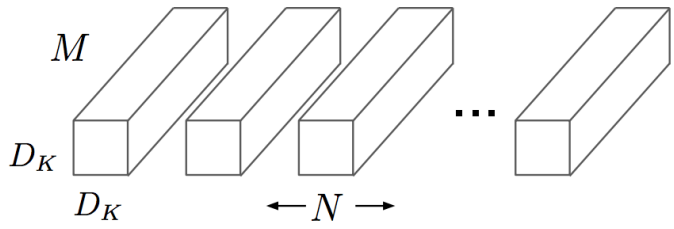
Formally the computational cost of this type of convolution is:

$$S \cdot d \cdot d + S \cdot T \tag{3.5}$$

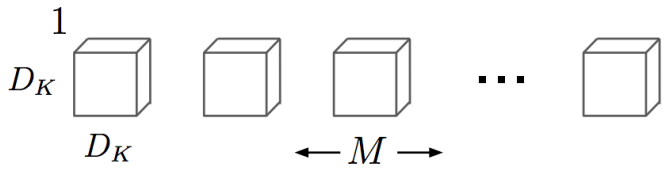
Another important parameter of this design is the so called *depth multiplier* which sets the number of depthwise convolutional output channels per input map.

The example above is a specific implementation of a depthwise separable convolution where the depth multiplier is 1. This is by far the most common setup for such layers and have been made popular by F.Chollet within the Xception network [**chollet**].

The core hypothesis of his work is that the mapping of cross-channels correlations and spatial correlations in the feature maps of convolutional neural networks can be *entirely decoupled*.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters

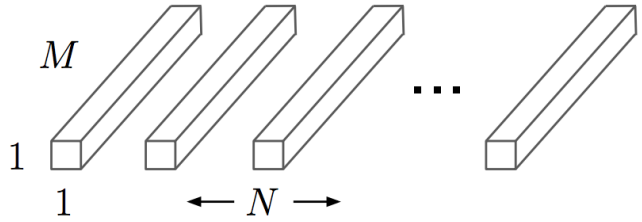


Figure 3.9: Depthwise convolution and regular convlution.

Looking at the results, this hypothesis held true as Xception outperforms even the popular *inception* modules, from which they took inspiration from.

As the author puts it in conclusions:

> We expect depthwise separable convolutions to become a cornerstone of convolutional neural network architecture design in the future, since they offer similar properties as Inception modules, yet are as easy to use as regular convolution layers.

This thesis takes this hypothesis further by also decoupling the spatial convolution into a vertical and an horizontal step.

3.3.3 CNN Micro-architecture

With the trend of designing very deep CNNs, it becomes cumbersome to manually select filter dimensions for each layer. To tackle this issue, various various higher level building blocks, or *modules* have been proposed. These modules, comprised of multiple convolution layers with a specific fixed organization, are then combined with additional ad-hoc layers to form a complete network. Hence, CNN *micro-architecture*

is a term that refers specifically to the internal organization of these individual modules.

To describe how these building blocks are combined together to form an end-to-end CNN it is used the term *macro-architecture* instead.

Inception module

One by one convolution, described in section 3.3.1, was immediately applied with success in GoogLeNet, which was the winner of the ImageNet challenge in 2014[**googlenet**]. Its main contribution was the developement of the *inception module*, depicted in figure 3.10

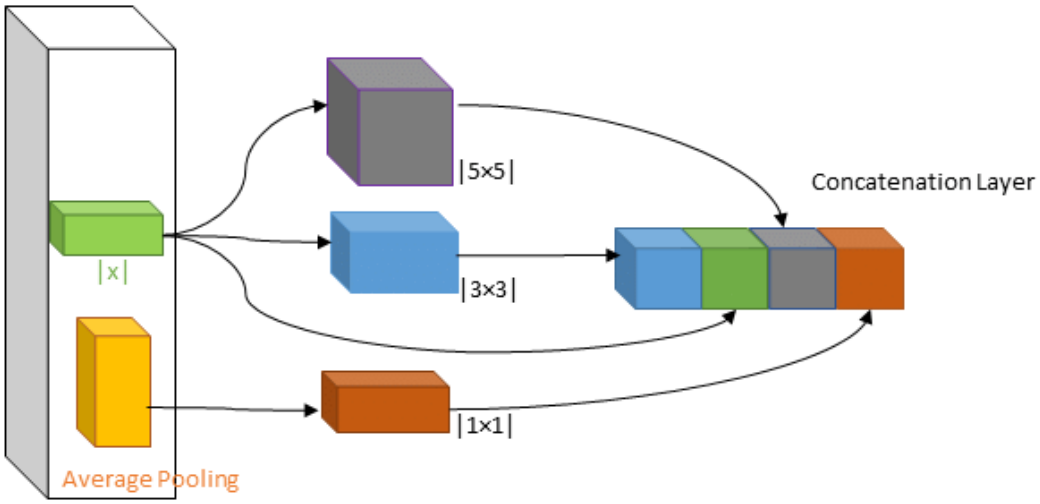


Figure 3.10: Inception module: the building block of GoogLeNet.

This module address the issue of not knowing a priori which kind of layer is the best choice to extract the features. Therefore this module provides a solution not to have to decide at all. In fact, it combines four different layers together so to have a richer feature representation.

Throughout experiments the authors found out that the model was able, in a sense, to choose on its own, i.e. some type of layers were more beneficial on a stage of the network with respect to another, thus showed richer feature maps. Additionally, this architecture allows the model to recover both local feature via smaller convolutions and more abstracted features with larger convolutions.

Although this was already a breakthrough on its own, perhaps the most important concept of the work was *dimensionality reduction*. Taking a closer look at figure 3.10, it is easy to notice the presence of diverse $[1 \times 1]$ convolution layers. These layers help to keep the number of parameters reasonably low by performing dimensionality reduction as explained in section 3.3.1.

Therefore, GoogLeNet was able to extract many different representations of the input image while avoiding a parameter explosion. Afterwards, different updated version of the inception modules were proposed. [**incpetion3**] [**inception4**].

3.3.4 Residual block

Residual blocks were made popular in 2015 as the building block of *deep residual networks* called ResNet, from Microsoft Research Asia [**resnet**]. ResNet is a *very* deep CNN,comprising of 152 layers in its original version, which imposed itself as state of the art in all ILVSRC competition categories: classification, detection and localization.

Residual blocks are based on the insight of *residual learning* (figure **??**, that works as follows:

1. the input goes through a `conv-relu-conv` sequence, producing a certain output called $F(x)$;
2. the original input is added to the latter: $H(x) = F(x) + x$;
3. $F(x)$ is called the residual learning.

Residual learning changed the state of the art by suggesting some new ideas:

1. instead of throwing away the original input and use only the feature maps representations of it, residual block keep the original information throughout the whole network;
2. each block peform a kind of fine-tuning (see later) of the information learned by the previous layers, thus being easier to optimize;
3. residual learning helps training very deep models by addressing the *vanishing gradients* issue. Arguably the summation with the residual contribute to the flow of information through very deep networks. However, this is not a certain explanation. The authors also argue that the model acts like an ensemble of network, taking classification knowledge from different blocks according to the class.

3.3.5 Fire module

The Fire module was recently proposed within the SqueezeNet model by Iandola et al. 2017 [**squeezenet**], which reached AlexNet like[**AlexNet**] accuracy on ImageNet while being *50X* smaller.

The design principles followed by the authors constitutes an interesing insights to achieve high accuracy with low number of parameters:

- Strategy 1.Replace 3x3 filters with 1x1 filters.Given a budget of a certain number of convolutionfilters, we will choose to make the majority of these filters 1x1, since a 1x1 filter has 9X fewerparameters than a 3x3 filter.
- Strategy 2.Decrease the number of input channels to 3x3 filters. This is done by the squeeze layers.
- Strategy 3.Downsample late in the network so that convolution layers have large activation maps, thus contributing to more powerful feature extraction.

These ideas led to the creation of the Fire module, illustrated in figure 3.11

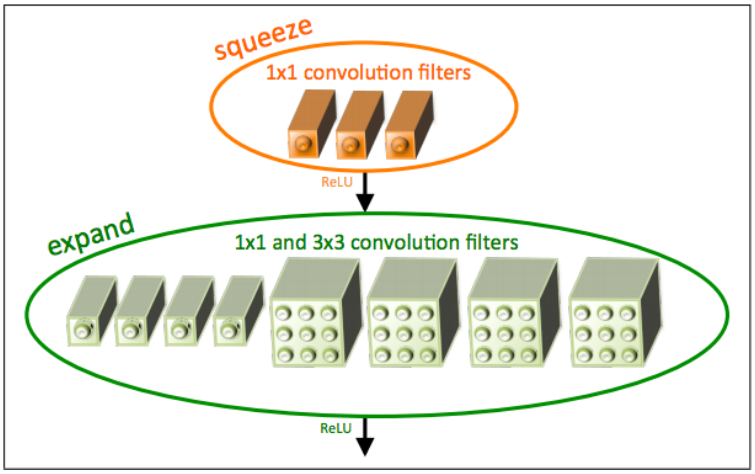


Figure 3.11: Fire-module: building block of SqueezeNet.

MobileNets

In the lights of these results, *MobileNets* a recent work from Google [**mobilenets**] which combines the design principles of SqueezeNet with the depthwise convolution proposed by Xception, have been integrated as the standard mobile-ready networks in a recent update of the popular framework Tensorflow [**tensorflow**].

3.4 Deep Learning

In this section the main concepts to actually train a CNN are introduced. It is also provided a quick recap on general neural networks training procedurs; however, as an in-depth discussion of the former is out of the scope of this thesis, this is not to be considered as an exhausting explanation. For further details, please see the references.

3.4.1 Training a NN: quick recap

Neural nets have many hidden layers and each one of them has a non-linear output. Therefore it is not trivial to train them without a specific procedure. Luckily, in 1985,

Rumelhart-Hinton-Williams came up with the algorithm that is still at the core of deep learning: *backward propagation of errors* or simply backprop.

Loss function

To understand backprop is necessary to introduce the concept of *loss function*. The loss function (or cost function) measures the discrepancy between the desired output and the actual output of the net. Therefore it defines the direction in which the net should learn i.e. the goal of the learning is to *minimize* the loss function.
Clearly, the loss function is modeled according to the specific task.

Once the loss function is defined, we can adopt backprop to minimize the error of the network.

Backprop

Backprop utilizes the *chain-rule* to compute the derivative of compound functions in order to calculate the gradient of the loss function, with respect to the network weights. Then, the computed error is used to update the network weights according to the gradient-descent algorithm (**??**).

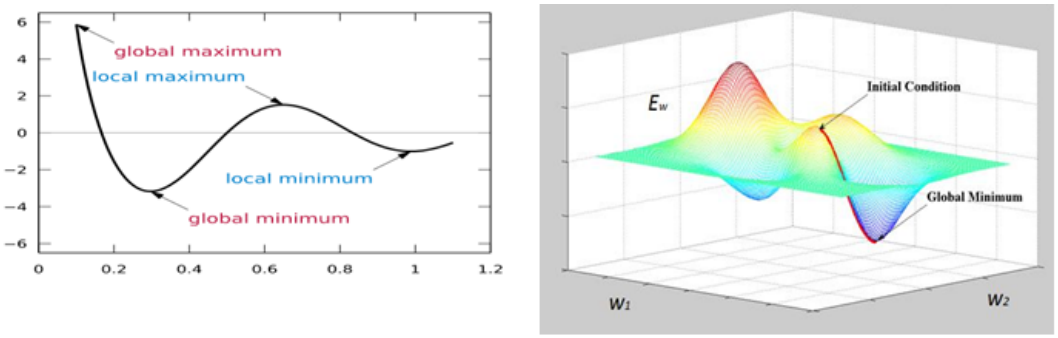


Figure 3.12: Architecture of a CNN

It is called backward propagation because the error is computed form the output of the net and back-propagated to all layers. The weights of each layer are then updated according to their contribution to the error, i.e. if a layer contributed heavily to the wrong output, its weights will change more drastically.

An important parameter of the learning is the *learning rate* η, which defines the magnitude of the update for each iteration.

Optimization

Backprop and loss function define by themselves a procedure to train a neural network. However, once we have the error contribution for each layer, the strategy that defines the weights update is a problem of optimization techniques.

There are several optimization methods for deep learning, which are out of the scope of this thesis. The most popular and successful as of 2018 are:

- Stochastic Gradient Descent (SGD) with Nesterov Momentum Update;
- RMSProp (adaptive);
- ADAM (adaptive).

A discussion of the aboves is out of the scope of this thesis. For more details, see [**sgd-overview**].

Supervised learning

Supervised learning is an effective way to train a neural network to classification tasks. In the case of images, it works by giving as input to the network an image and an associated label. The network computes its classification (the output is the proposed label) and compares it with the given label. If the labels correspond, then it reinforeces the paths the led to a good classification. Otherwise, it corrects the weights by using BackProp and optimization strategies.

3.4.2 Data preprocessing

In the training phase it is a good norm to normalize the input data so that have zero mean and unitary variance. It turns out that this is important for the parameter update step, indepdentely from the actual optimization algorithm.

3.4.3 Data Augmentation

In order to make the model more robust to unseen examples, a diffuse technique is data augmentation. The images are modified to make the dataset larger and more diverse. Often used techniques usually are amongs the following:

- Scaling
- Translation
- Rotation (at 90 degrees)
- Rotation (at finer angles)
- Flipping
- Adding Salt and Pepper noise
- Lighting condition
- Perspective transform

Batch normalization

Batch normalization layers help both in faster convergence and achieve a higher overall accuracy. In practice, it involves normalizing the data, as in the preprocessing step, before every layer. In fact, during the forward pass, the processing applied by the weights to the original input modifies it in a way that affects the learning process of the subsequent layers. Batch normalization helps avoiding this.

3.4.4 Initial solutions

Initial solutions are a very important aspect of deep learning. In fact, if the initial values of the weights are too small, the signal flowing through the network gets weaker and weaker, till it is useless to the learning process (also called, vanishing gradients).

On the other hand, if these values are too high the signal coming from a layer becomes too strong and that may cancel some other important but weaker contributions. For these reason a good initialization is important.

Xavier initialization

Proposed in [**xavier**], this type of initialization makes sure the weights are neither too high or too small. It chooses random values from a uniform random distribution with zero mean and variance equals to $\frac{1}{n}$, where n is the number of input features. This have proven to work well in practice. For more details, please see the references.

3.4.5 Overfitting

Over-fitting is a problem that emerges when the network becomes overly complex and bounded to the training set, from which it also learns the noise. Thus the network becomes incapable of generalizing to new examples, and, although the high accuracy on the training set, will get poorly predictions on unseen data.

The most common reason of these are:

- the model have too many parameters;
- dataset has too few examples;
- the training was carried out for too long.

There are, however, also several countermeasures to prevent overfitting:

- a larger dataset;
- early stopping: stop training before the model gets too complex. Usually, this is done by dividing the dataset into training and validation. When the accuracy on the validation data is not improving anymore it is possible to stop the training, since it will not bring any more benefit.

- Regularization: it consists in adding a term to the loss function, that penalizes too complex models. Depending on the form of regularization, the weights become more or less uniform.

As we will see during the experiments, tensor decomposition acts as a form of regularization and can prevent overfitting, leading to a better overall generalization of the model.

3.5 Applications

The high efficiency, the peculiar, advantageous architecture together with the huge technological progress of the hardware, have made CNN the most promising system for visual tasks, with the most varied application areas such as: recognition and facial tagging (think about Facebook), intelligent image search (think of Google Photos), autonomous cars, smartphones, robots, drones, (video) games and more. CNNs have also had excellent results in natural language processing; in the discovery of drugs, since by predicting the interactions between certain molecules and proteins they have helped discover potential biomolecules for the treatment of Ebola [**WCNN**], just to name a few others. Already 3 years ago, in an article published by the KTH [**Overfeat**] artificial vision department, the use of "OverFeat", a CNN trained for the 2013 ImageNet Challenge, was analyzed. The article remakrs how they used this CNN "off-the-shelf" that is to say, ready, and without further training it, testing it against other finely perfected state-of-the-art methods developed until then. As tests, they have chosen activities that are gradually getting further and further from the original task for which OverFeat has been trained and, they have verified that OverFeat outclasses the above mentioned methods on any dataset (see the article for details) despite having been trained only through ImageNet. The article closes with a sentence that I quote:

> *"Thus, it can be concluded that from now on, deep learning with CNN has to be considered as the primary candidate in essentially any visual recognition task."*

3.5.1 Comparison with humans

In 2011, CNNs first beat the man by reaching a 0.56 % error against 1.16 % of humans on the recognition of road signs in the "German Traffic Sign competition run by IJCNN 2011" competition.

Two years ago, in the annual ILSVRC competition, now considered by the community as the "Artificial Vision Olympics", Microsoft Research Asia presented ResNet [**resnet**]: a 152-layer CNN that lowered the classification error (1000 classes) to *only*

3.6%. The result is impressive, given that a human being more or less able to recognize all the classes has an error of about 5-10%, see [**Wkarpa**].

Another historically difficult task for artificial vision was the recognition of partially occluded, upside down or displaced seen from different angles. However, in 2015 a team from the Yahoo Labs managed to get a CNN [**WMit**] to learn this task.
The last milestone in chronological order of the "Man vs. Machine" challenge is undoubtedly AlphaGo[**WAlphaGo**]. AlphaGo is the first program to succeed in beating a professional human player (Lee Sedol, 18 times world champion) to the ancient Chinese game of Go. Go is known to be computationally extremely complex: there are 10^{170} possible combinations of the chessboard, a higher number than the atoms in the known Universe. It is therefore unsuitable for a "brute force" approach.

AlphaGo is based on a combination of deep learning + tree search. In particular, it uses 3 CNNs: 2 "policy networks" to choose the strategy that most likely will lead to a win and 1 "value network" as an heuristic function to evaluate the correctness of an hypothetical move. In addition, the output of these networks is combined with a Monte Carlo Tree Search to get a final answer on the next move to play. More details can be found on the exhaustive paper published by DeepMind [**AlphaGo**].

These results are sufficient to understand the potential of convolutional neural networks.

Chapter 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.

4.1 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 4.1.

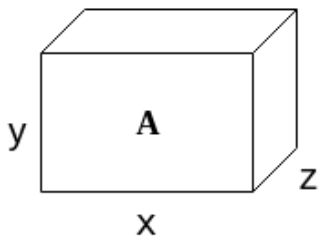


Figure 4.1: A third order tensor.

4.1.1 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 [**tensor-hackbusch**]) is a tensor that can be written as a product of tensors of the form:

$$T = a \circ b \circ \ldots \circ d \tag{4.1}$$

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

The product used in 4.1 is an outer product for vectors and will be defined in details in the coming section.

4.1.2 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 4.2. 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.

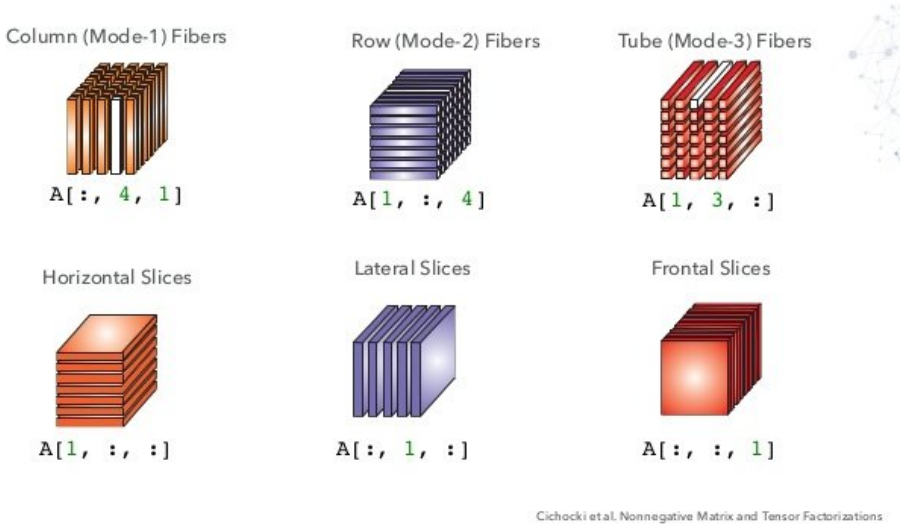


Figure 4.2: Fibers and slices of a tensor according to each mode.

To give another cue on the above, 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 [**WTensor**] to modelize multi-dimensional data. In the CNN scenario, a CONV layer with $\mathcal{W}$ weights is defined through a tensor of size:

$$dim(\mathcal{W}) = [T \times S \times D \times D] \tag{4.2}$$

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 4.2 the formal tools to wield tensors are given and in section 4.3 the application of these methods to convolutional layers are presented.

4.1.3 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:

$$M = U\Sigma V^* \tag{4.3}$$

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:
$$M^* = (\bar{M})^T = \bar{M * T}$$

- 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 4.3, 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.

4.1.4 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 [**recsys1**] [**recsys2**]; for instance, Netflix has launched a global competition to find the best implementation of SVD for clusters to improve its collaborative filtering technique [**recsys3-netflix**]. 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 4.1.1 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:

$$M = \sum_i A_i = \sum_i \sigma_i U_i \circ V_i^T \tag{4.4}$$

this turns out to be enable a fast convolution computation:

$$F = \sum_i^R \sigma_i (I * U_i) * V_i \tag{4.5}$$

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:

$$\left\| M - \hat{M} \right\|, \quad with\ \hat{M} = \sum_i U_i \circ V_i \tag{4.6}$$

The Eckart-Young theorem [**Weckart**] 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 2. In particular, the work of Cheng et al. [**zhang2015SVD**] 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:

$$Y = W * X + B \tag{4.7}$$

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:

$$(U_{m\times r}\Sigma_{r\times r}V^T_{n\times t})X + B = U_{m\times r}(\Sigma_{r\times r}V^T_{n\times r}X) + B \tag{4.8}$$

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 [**rcnn**], a very famous CNN built for region proposal and object detection. The implementation for Caffe is also available on the author github page, at [**Wgithub-rcnn**].

4.2 Tensor mathematical tools

4.2.1 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 [**tensor2009kolda**]. 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 4.2.1.

- **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:

$$\mathbf{Y} = \mathbf{X} \times_k M \tag{4.9}$$

  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 4.2.1, we can

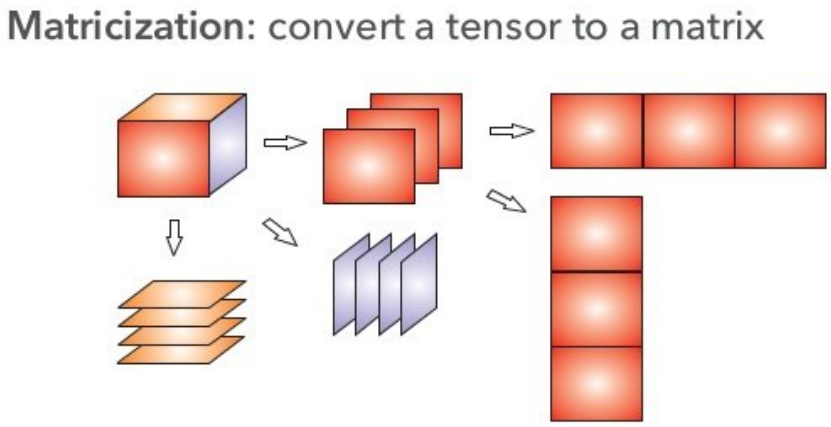


Figure 4.3: Tensor unfolding or matricization along different modes.

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:

$$\mathbf{Y} = \mathbf{X} \times_1 M, Y \in F^{I \times J \times K} \tag{4.10}$$

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

$$y_{n,j,k} = \sum_i x_{i,j,k} \cdot b_{n,j} \tag{4.11}$$

A visual example is depicted in 4.2.1.

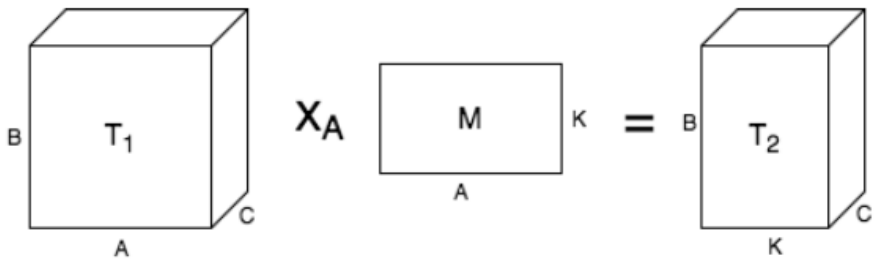


Figure 4.4: An example of a k-mode product i.e., a tensor-matrix multiplication of a 3-dimensional tensor.

Few interesting properties of the k-mode product are:

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

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

$$\mathbf{Y} = \mathbf{X} \times_1 v, \mathbf{Y} \in \tag{4.12}$$

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

$$y_{j,k} = \sum x_{i,j,k} \cdot a_i \tag{4.13}$$

An example is illustrated in 4.2.1.

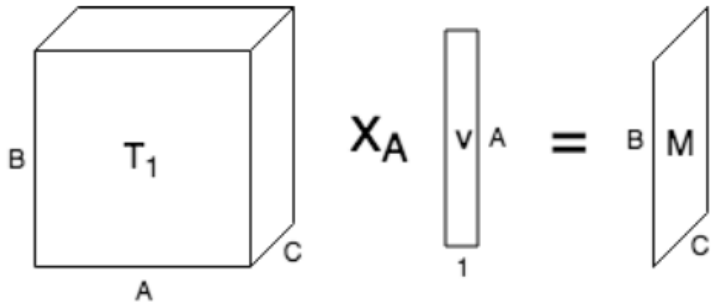


Figure 4.5: An example of a tensor-vector product.

- **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:

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix} \tag{4.14}$$

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:

$$\mathbf{a} \circ \mathbf{b} = \mathbf{a}\mathbf{b}^{\mathrm{T}} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} = \begin{bmatrix} a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_1 & a_3b_2 & a_3b_3 \\ a_4b_1 & a_4b_2 & a_4b_3 \end{bmatrix}. \tag{4.15}$$

  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 4.6.

  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-$(d1 + d2)$.

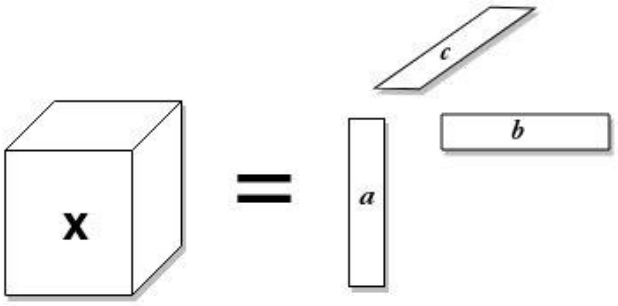


Figure 4.6: Representation of third-order tensor with an outer product of vectors.

- **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:

$$A \odot B = [a_1 \otimes b_1, a_2 \otimes b_2, \ldots, a_N \otimes b_R] \in \mathbf{R}^{MN \times R} \tag{4.16}$$

  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.

4.2.2 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 $I_n \times R_n$:

$$\mathrm{X} = \mathbf{G} \times_1 A^{(1)} \times_2 A^{(2)} \times_3 \ldots \times_N A^{(N)} \tag{4.17}$$

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 :

$$X = \mathbf{G} \times_1 A \times_2 B \times_3 C = \sum_r \sum_s \sum_t \sigma_{r,s,t} u_r \circ v_s \circ w_t \tag{4.18}$$

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 4.7.

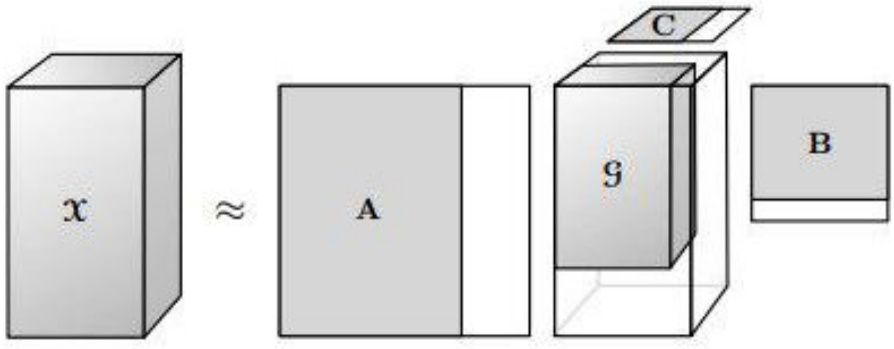


Figure 4.7: A Tucker decomposition of a three-modes tensor

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

$$X_{(1)} = A\mathbf{G}_{(1)}(C \otimes B)^T \tag{4.19}$$

$$X_{(2)} = B\mathbf{G}_{(2)}(C \otimes A)^T \tag{4.20}$$

$$X_{(3)} = A\mathbf{G}_{(3)}(B \otimes A)^T \tag{4.21}$$

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

$$vec(X) = (C \otimes B \otimes A)vec(\mathbf{G}) \tag{4.22}$$

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 4.1.4; 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 [**WTucker**], 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 4.8. The tensor $\mathbf{G}$ is said to be *"ordered"* and *"all-ortoghonal"*. For further explanation see [**multilinear**].

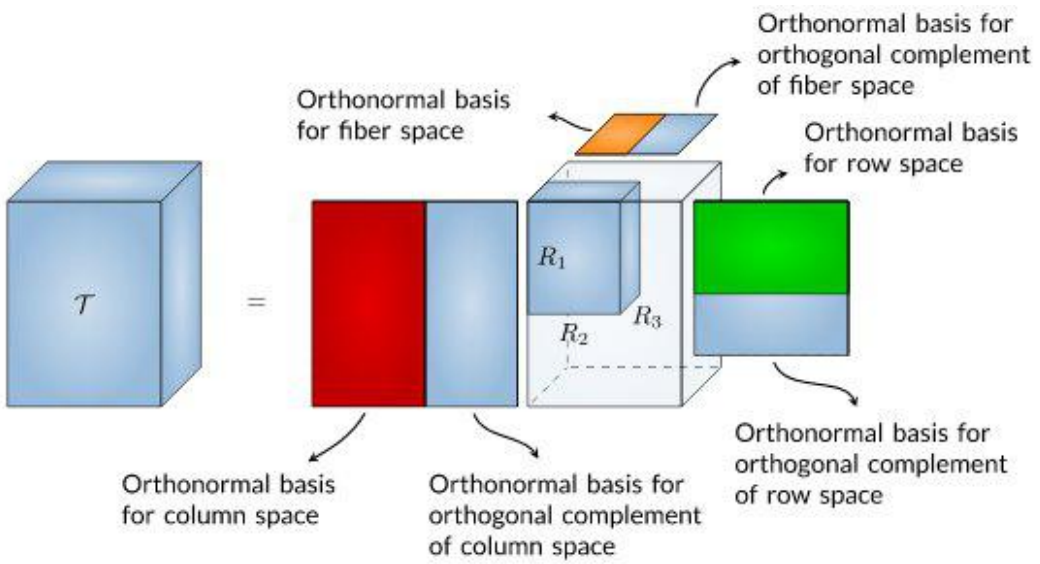


Figure 4.8: An Higher Order SVD of a third-order rank-(R1-R2-R3) tensor and the different spaces, from Tensorlab [**WTensorlab**].

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 $A(k)$ 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
$G \leftarrow X \times 1(A^{(1)})^T \times 2(A^{(2)})^T \times 3 \cdots \times N(A^{(N)})^T$

(4.23)

4.2.3 Canonical Polyadic Decomposition

The Polyadic Decomposition (PD) [**tensor2009kolda**] 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 4.2. 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:

$$T \approx \sum_{r=1}^{R} \vec{u}_r^{(1)} \circ \vec{u}_r^{(1)} \circ \cdots \circ \vec{u}_r^{(N)}. \tag{4.24}$$

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 $\mathbf{G}$ 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.

4.3 Application of tensor decompositon on CNN

4.3.1 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:

$$V(x, y, t) = \sum_{i=x-\delta}^{x+\delta} \sum_{j=y-\delta}^{y+\delta} \sum_{s=1}^{S} K(i - x + \delta, j - y + \delta, s, t) U(i, j, s) \tag{4.25}$$

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.

4.3.2 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:

$$\sum_{r=1}^{R} K^x(i-x+\delta,r)K^y(j-y+\delta,r)K^s(s,r)K^t(t,r) \tag{4.26}$$

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 4.26 into 4.25 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:

$$V(x,y,t) = \sum_r K^t(t,r)\left(\sum_i\sum_j K^x(i-x+\delta,r)K_r^y(i-y+\delta,r)\left(\sum_s K^s(s,r)U(i,j,s)\right)\right) \tag{4.27}$$

Equation 4.27 can be splitted into four steps that constitutes a sequence of four squeezed convolutions. The former convolutions will replace the single more costly one:

$$U^s(i,j,r) = \sum_{s=1}^{S} = K^s(s,r)U(i,j,s) \tag{4.28}$$

$$U^{sy}(i,y,r) = \sum_{j=y-\delta}^{y+\delta} K^y(j-x+\delta,r)U^s(i,j,r) \tag{4.29}$$

$$U^{syx}(x,y,r) = \sum_{i=x-\delta}^{x+\delta} K^x(i-x+\delta,r)U^{sy}(i,y,r) \tag{4.30}$$

$$V(x,y,t) = \sum_{r=1}^{R} K^t(t,r)U^{syx}(x,y,r), \tag{4.31}$$

where U^s, U^{sy} $Usyx$ are intermediate tensors i.e., map stacks.

This decomposition scheme is illustrated in figure 4.9.

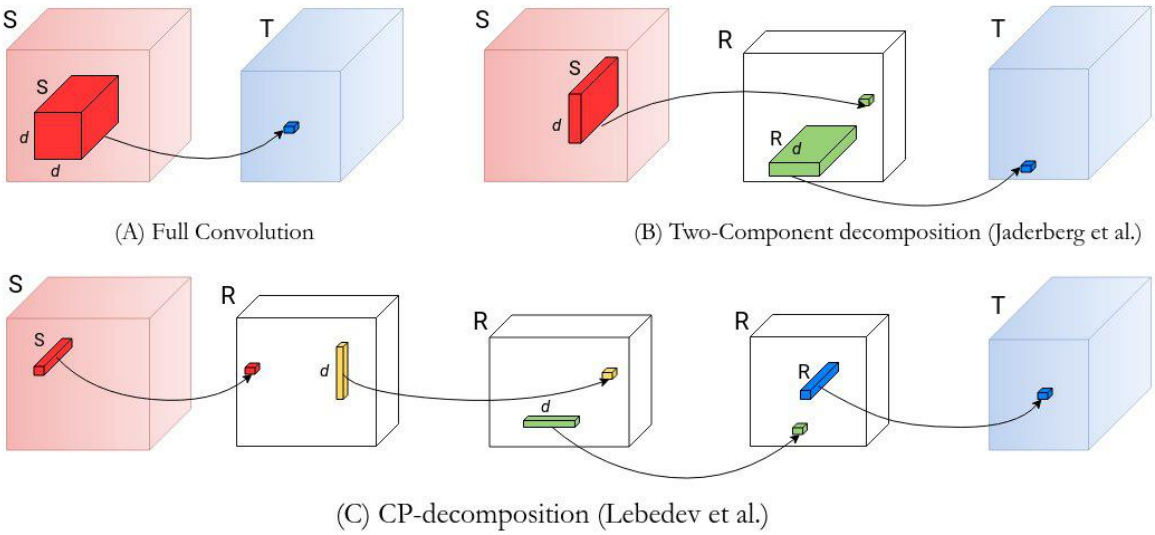


Figure 4.9: Tensor Decompositions for speeding up a generalized convolution. Each box correspond to a feature map stack within a CNN, (frontal sides are spatial dimensions). Arrows show linear mappings and demonstrate how scalar values on the right are computed. Initial full convolution (A) computes 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 map 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 (4.28-4.31)

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 3.3.2 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 4.3.4.

CPD-3

There is another way to compute a convolutional layer with CP that has been suggested in[**astrid2017**] 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 ($[3 \times 3] \quad [5 \times 5]$) and so the factorization gain is not significant compared to the much bigger channels size [**Tucker-mobile**]; 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:

$$U^s(i,j,r) = \sum_{s=1}^{S} = K^s(s,r)U(i,j,s) \tag{4.32}$$

$$U^{syx}(i,y,r) = \sum_{i=x-\delta}^{x+\delta} \sum_{j=y-\delta}^{y+\delta} K^{xy}(i-x+\delta, j-x+\delta, r)U^s(i,j,r) \tag{4.33}$$

$$V(x,y,t) = \sum_{r=1}^{R} K^t(t,r)U^{syx}(x,y,r), \tag{4.34}$$

This process is also described in figure 4.10 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.

4.3.3 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:

$$K(i,j,s,t) = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} G_{r_1,r_2,r_3,r_4} K^x_{r1}(i) K^y_{r2}(j) K^s_{r3}(s) K^t_{r4}(t) \tag{4.35}$$

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.[**Tucker-mobile**]:

$$K(i,j,s,t) = \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} \mathbf{G}_{i,j,r_3,r_4}(j) K^s_{r3}(s) K^t_{r4}(t) \tag{4.36}$$

Plugging equation 4.36 into 4.25 a new equation for the Tucker convolutional forward pass is obtained:

$$V(x,y,t) = \sum_{i=x-\delta}^{x+\delta} \sum_{j=y-\delta}^{y+\delta} \sum_{s} \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} \mathbf{G}(i-x+\delta)(j-y+\delta) r_3 r_4 K_{r3}^{s}(s) K_{r4}^{t}(t) X(i,j,s) \tag{4.37}$$

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

$$U^{s}(i,j,r) = \sum_{s=1}^{S} = K^{s}(s,r) U(i,j,s) \tag{4.38}$$

$$U^{syx}(i,y,r) = \sum_{i=x-\delta}^{x+\delta} \sum_{j=y-\delta}^{y+\delta} \mathbf{G}_{i,j,r_3,r_4} U^{s}(i,j,r) \tag{4.39}$$

$$V(x,y,t) = \sum_{r=1}^{R} K^{t}(t,r) U^{syx}(x,y,r), \tag{4.40}$$

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 4.10 illustrate equation 4.38-4.40.

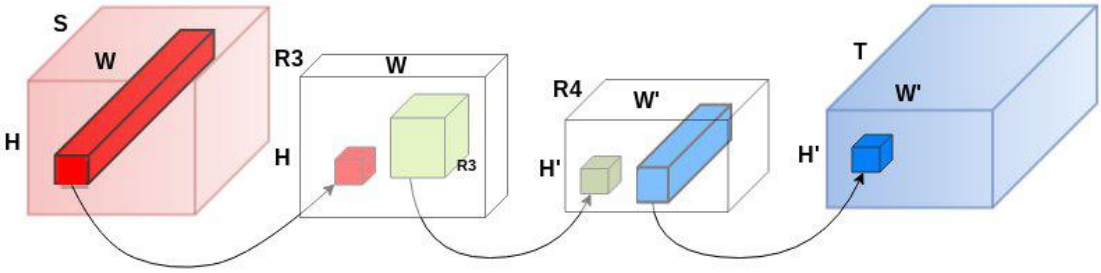


Figure 4.10: Tucker-2 Decompositions for speeding-up a generalized convolution. Each box corresponds to a 3-way tensor $X, Z, Z^{'}$ ^{and}Y in equation (4.38-4.40). Arrows represent linear mappings and illustrate how each scalar value on the right is computed. Red tube, green cube and blue tube correspond to 1x1, dxd and 1x1 convolution respectively.

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.

$$U^s(i,j,r_3) = \sum_s K^s(s,r_3)U(i,j,s) \tag{4.41}$$

Then we apply 1D vertical convolution:

$$U^{sy}(i,y,r_3) = \sum_j K^y(j-y+\delta,r_3)U^{sx}(i,j,r_3) \tag{4.42}$$

a 1D horizontal convolution:

$$U^{syx}(x,y,r_3) = \sum_i K^x(i-x+\delta,r_3)U^{sy}(i,y,r_3) \tag{4.43}$$

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

$$U^{syx\sigma}(x,y,r_4) = \sum_{r_3} \mathbf{G}_{1,1,r_3,r_4}U^{syx}(x,y,r_3) \tag{4.44}$$

and finally reshape the output channel:

$$V(x,y,t) = \sum_{r_4} K^t(t,r_4)U^{syx\sigma}(x,y,r_4) \tag{4.45}$$

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 to 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[**resnet**] have already shown how to tackle this issue.

4.3.4 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.[**jaderberg2014**] and Zhang et al. [**zhang2015SVD**] 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. [**lebedev**], 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

$$\mathcal{C}_r = \frac{O(\mathcal{K}_{std})}{O(\mathcal{K}_{dec})} \tag{4.46}$$

which in the case of CP-decomposition equals to:

$$\mathcal{C}_r = \frac{STd^2}{R(S+2d+T)} \tag{4.47}$$

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:

$$\mathcal{S}_r = \frac{STd^2HW}{RSHW + dRH'W + RdH'W' + RTH'W')} \tag{4.48}$$

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 [**rank-hard1**].

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 4.4.5

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:

$$mathcalC_r = \frac{d^2ST}{R(S+d^2+T)} \tag{4.49}$$

Table 4.1: 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)	$R(S+d^2*T)$

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

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. [**zhang2015SVD**], in table 4.1 can be seen a comparison of parameters reduction for the different methods explored.

4.4 In-depth discussion

4.4.1 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* [**rank-hard1**][**rank-hard2**][**rank-hard3**].

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 4.2.2
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'`.

In figure 4.11 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.

iterative compression and fine-tuning A simple solution that has been applied by different authors is an iterative careful manual tuning [**astrid2017**] [**lebedev**] 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 [**astrid2017**] they define a metric to measure the *sensitivity* of a layer l as:

$$S_l = \frac{Loss_l}{Total \quad Loss} \tag{4.50}$$

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.

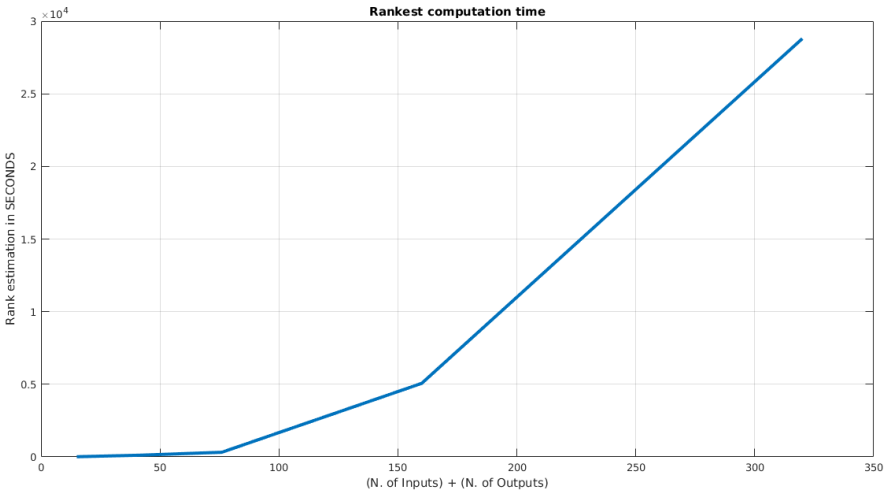


Figure 4.11: Evaluation of the 'rankest' method for different sizes of input and output maps. Note that the maps sizes can be even higher in CNNs. Clearly, an iterative rank estimation approach does not scale well.

In Lebedev et al. [**lebedev**] 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. [**nakajima2013**] 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 4.12.

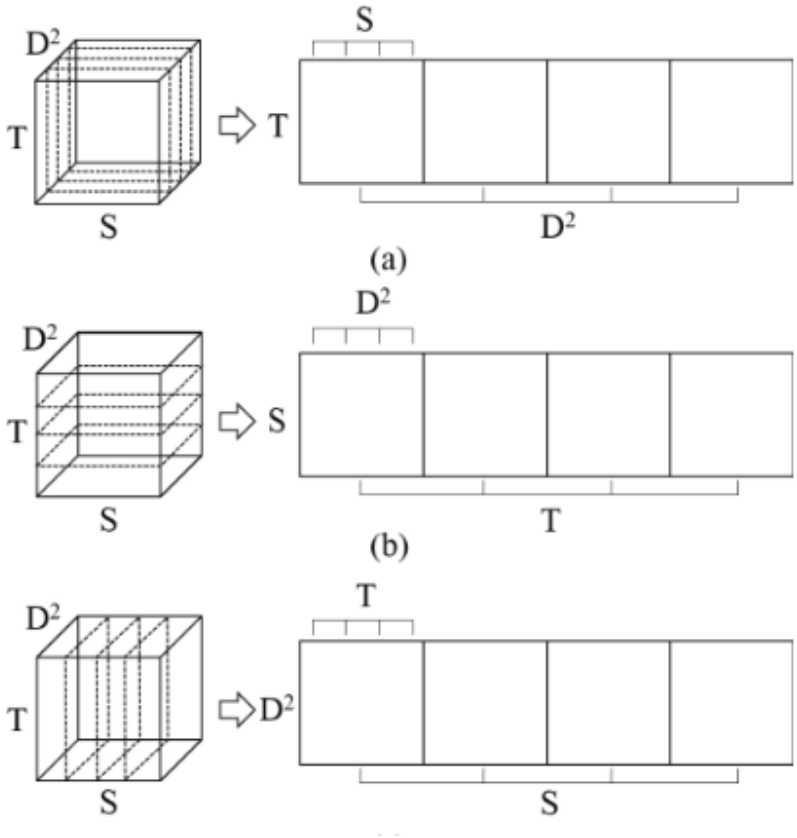


Figure 4.12: Mode-1 (top left), mode-2 (top-right), and mode-3 (bottom left) tensor unfolding of a third-order tensor. **Not shown**: after unfolding, the rank R of the resulting matrices is computed with VBMF. (Bottom right): Then, given the rank R, the Tucker decomposition produces a core tensor $\mathcal{S}$ 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.

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. [**Tucker-mobile**]. 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 [**astrid2017**]: 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.

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 4.13

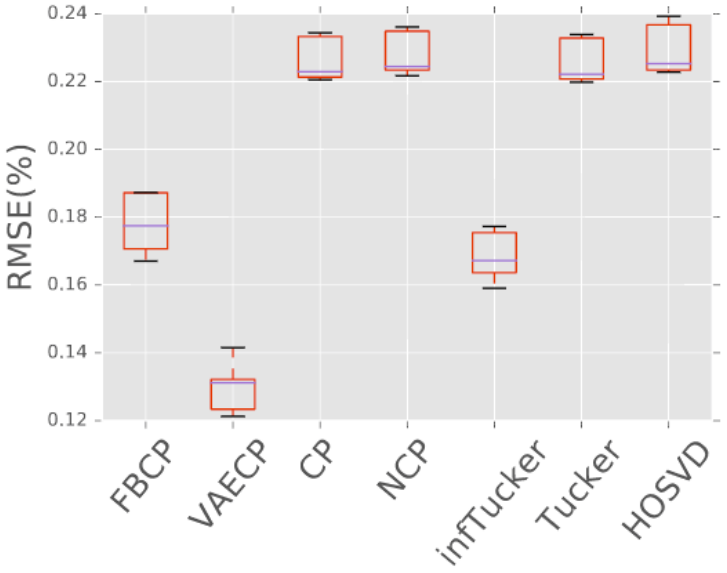


Figure 4.13: Comparison of different optimization algorithms to compute tensor decomposition on real datasets. The method proposed by Liu et al. outperforms other methods by a large margin.

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 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.

4.4.2 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 [**WTensorlab**].
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 [**WTensorlab-ref**].

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 has been implemented and the results can be observed in figure 4.14.

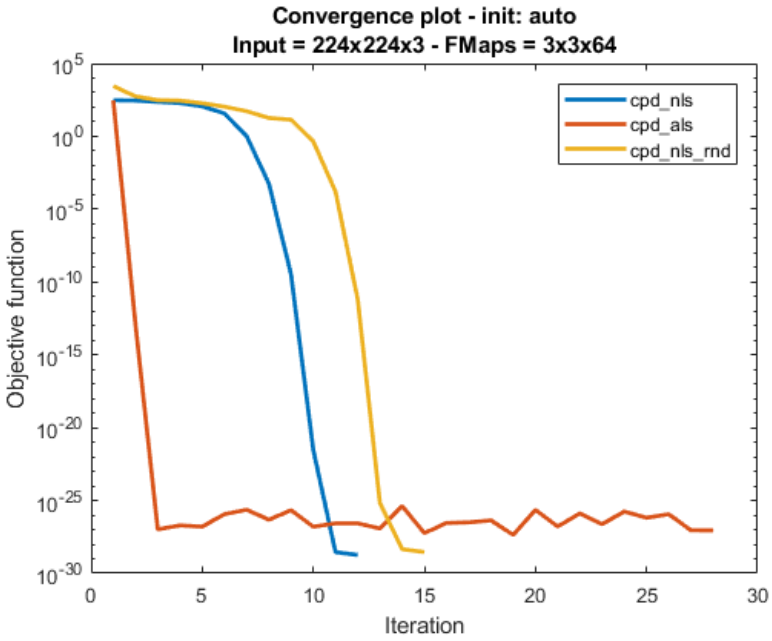


Figure 4.14: Comparison of different optimization algorithms to compute CPD; between yellow and blue line only the init method is diverse. It is evident how ALS is indeed faster but more unstable and less precise.

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 [**WCPD-talk**]. Their results on accuracy and timing are reported in figure 4.16.

It is also interesting to see how these methods depend heavily on the chosen rank R. To this aim, a script to test these methods with respect ot the rank are shown in 4.15.

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.

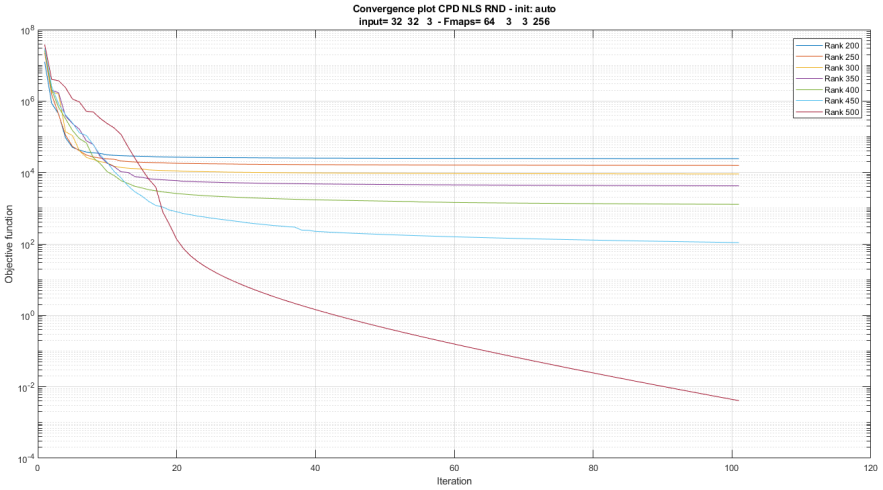


Figure 4.15: Comparison of CP-NLS decomposition for different ranks: the higher the R-terms, the higher is the accuracy. For R=500 the decomposition is much more accurate than other ranks.

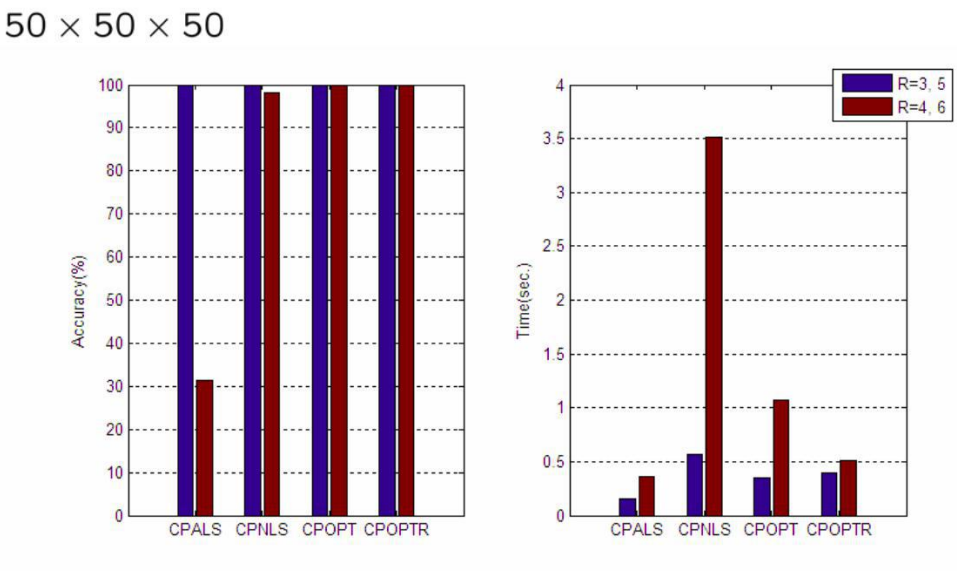


Figure 4.16: Accuracy and timing comparison between various CP-decomposition algorithms. ALS is always fast but much less accurate than CP-OPT. Image from [**WCPD-talk**].

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 4.2.2 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 [**tucker-fast1**] [**tucker-fast2**] and coule be subject for further experiments in future work.

4.4.3 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 4.3.2. 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"* (TD-block) can be finally introduced in figure 4.17.

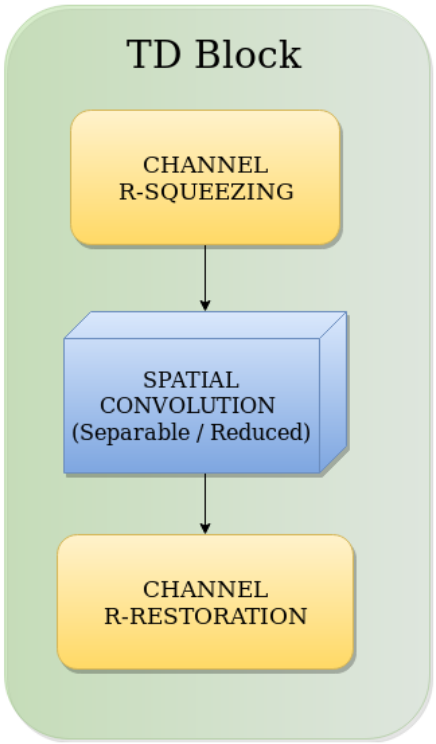


Figure 4.17: A generic Tensor Decomposition (TD) block micro-architecture. The first and last layers performs channels squeezing/restoration while the actual convolution is performed inside these two layers. This convolution can be either separable or standard; either way the cross-channel parameters redundancy is exploited.
The *R*-prefix stands for 'rank', which is the only hyper-parameter.

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.

4.4.4 TD-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 TD-block is a good fit to substitute a whole layer after training, then it will likely be as good if applied from the beginning.
To this aim, it is interesting how *Mobile Nets* [**mobilenets**], that builds upon a similar architecture that exploits channels redundancy, have been included in Tensorflow Lite [**WTFlite**], the mobile-optimized version of the most popular deep learning framework [**tensorflow**], becoming the go-to solution to deploy CNNs on mobile.

Indeed, in the next chapter several experiments about this approach demonstrate how TD-blocks can perform well when training models from scratch.

This suggests *a new way* of building standard CNNs.

4.4.5 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 TD-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.

Chapter 5

Experimental Results & Analysis

In chapter 4 the theoretical background of tensor decomposition has been introduced. The way Tucker and CP decomposition can be applied on convolutional layers has suggested a specific micro-architecture, hence the TD-block has been introduced. Following this line, this chapter will show how to apply the TD-block as a building block and as a compression block for five different models.

5.1 Experimental setup

In this section, we break down the dataset and the five different architectures on which tensor decomposition has been applied. The core strategies of the experiments will be described and motivated.

5.1.1 Tools

All the experiments have been executed in Ubuntu 17.10 `Intel(R) Core(TM) i7-4510U @2.00 GHz` cpu for testing and on a `NVIDIA GeForce 840M` with 2GB of memory for training. Most of the experiments have been implemented in *PyTorch* [**pytorch**] motivated by its flexibility and fast prototyping. For example, network surgery and decomposition are more intuitive in PyTorch than other frameworks. Regarding experiment 5 instead, the implementation was done in TensorFlow[**tensoflow**] and Keras[**keras**].

About tensor decomposition, two frameworks have been adopted for the experiments:

1. `TensorLab` toolbox for MATLAB [**Wtensorlab**]: very comprehensive tool for general tensor computations. It provides a plethora of algorithms and optimizations to compute tensor factorizations.
   There are also interesting tools for tensor visualization that could be useful for further investigations on the convolutional layer tensor structure.

2. `Tensorly` toolbox for Python [**tensorly**]: it is a light wrapping around NumPy but provides very few optimizations compared to TensorLab; for instance only ALS and HOOI to compute CP and Tucker respectively. Nevertheless, it gets the job done and easily embeddable into deep learning frameworks, as it is built

on NumPy.
However, for large convolutional layers with many activations (as the first ones) with large dense tensors it can stall or occur in memory errors (it saturates the RAM).

Pipeline implementation

The decomposition pipeline has been implemented as a python extensible class. This 'decomposer' class provides methods for CP, Tucker and Xavier decomposition with different configurations. It also features the possibility to manually select the desired compression ratio for a particular layer. The work supports Pytorch and Keras for now, but will be extended to TensorFlow in the near future.

5.1.2 Datasets

Experiments have been conducted on two datasets. As for other compression techniques in literature, a classification dataset, CIFAR10, has been selected as the testbed for most experiments.
Furthermore, the TD-block design has been tested on the KITTI dataset, to learn automaticatic estimation of disparity maps.

CIFAR-10

Although the CIFAR10 is not among the most challenging datasets, it provides a relatively fast way to test new ideas while also being not as trivial as MNIST.

The dataset consists of 60000 color images of size 32×32 pixels divided into 10 different classes, with 6000 images per class. 50000 of the images will be used for the training process and 10000 will be used to test the network.
The 10 classes in the dataset are: airplane, automobile, bird, cat, deer, dog, frog,horse, ship and truck. All those classes are completely mutually exclusive, not as in other datasets. Some examples of those classes can be seen in figure **??**.

The batch size for every training has been set to 32, as a common chioce for this dataset.

KITTI

The KITTI stereo/flow benchmark consists of 194 training image pairs and 195 test image pairs, saved in loss less png format. This dataset is provideds a set of real image pairs to evaluate stereo matching algorithms. Specifically, it will be used to compress CCNN [**WCCNN**], a deep convnet that has been able to infer from scratch an effective confidence measure by only using as input cue the disparity map.

Since this model is trained on 9× patches of large input images, training will be performed only 20 images while 174 will act as a test set. Although the size of

the patches is relatively small compared to the disparity map, it provides to CCNN enough cues to infer the degree of uncertainty for each point.

LeNet1

This model is a variation on the classic LeNet[**lenet**] model from Y.LeCun. It is composed by four convolution layers followed by ReLUs, and two fully-connected layers for the final classification. Max pooling and dropout completes the architecture, which is reported in details in table **??**. The total number of parameters is 1250858, i.e. ≈ 1.25 million.

This model will be used in order to test the TD-block micro-architecture effectiveness.

Different models: LeNet-1 (classic) LeNet-Conv (with fc layers converted in conv) LeNet-2 (the one used in Zhang et al., metodo maragno) NIN (Network in Network used in Zhang et al.) CCNN (Poggi)

LeNet-Conv

The `LeNet-Conv` model is similar to the previous one with the difference of having convolutional layers as classifiers instead of fully-connected ones. The conversion has been applied as described in chapter 3. Additionally it features BatchNorm layers. The whole architecture is depicted in table 5.1.

Therefore it has identical accuracy while being fully decomposable. The number of trainable parameters is also the same.

5.1.3 LeNet-2

This is another variation on LeNet, with fewer layers with bigger filter sizes and more channels, i.e. it is less deep but wider. This architecture is useful as it was also employed by Zhang et al. [**zhang2015SVD**] and hence can be a good measure of comparison between compression methods.

Table 5.2 summarizes the full architecture.

subsectionCCNN Proposed by Poggi and Mattoccia [**WCCNN**] at BMVC 2016, CCNN is a single channel network capable of infering the confidence measure of a pixel just by taking as the only input cue its disparity map. CCNN outperformed state-of-the-art methods with margin peaks of 20%, being the first method in this specific task to be based on deep learning.

The architecture of CNN is made of a single channel network that takes as input N×N patches, each one containing disparity values normalized between zero and one, represented by a 1×N×N tensor.

LAYER	CHARACTERISTICS
CONV	32 filters 3×3, padding=1, stride=1
ReLU	-
BNORM	eps = 1e-5 momentum=0.1
CONV	32 filters 3 × 3, padding=0 stride=1
ReLU	-
BNORM	eps = 1e-5 momentum=0.1
POOL	pool size [2, 2] stride=2
CONV	64 filters 3 × 3, padding=1 stride=1
ReLU	-
BNORM	eps = 1e-5 momentum=0.1
CONV	64 filters 3 × 3, padding=0 stride=1
ReLU	-
BNORM	eps = 1e-5 momentum=0.1
POOL	pool size [2, 2] stride=2
BNORM	eps = 1e-5 momentum=0.1
CONV	512 filters 6 × 6, padding=0, stride=1
ReLU	-
BNORM	eps = 1e-5 momentum=0.1
CONV	#Classes (=10) filters 1 × 1, padding=0, stride=1
BNORM	eps = 1e-5 momentum=0.1

Table 5.1: LeNet-Conv architecture.

LAYER	CHARACTERISTICS
CONV	192 filters 5×5 padding=2, stride=1
ReLU	-
POOL	pool size [2, 2] stride=2
CONV	128 filters 5 × 5, padding=2 stride=1
ReLU	-
POOL	pool size [2, 2] stride=2
CONV	256 filters 3 × 3, padding=2 stride=1
ReLU	-
POOL	pool size [2, 2] stride=2
CONV	64 filters 1 × 1, padding=0 stride=1
ReLU	-
Dropout	p=0.5
FC	256 units
Dropout	p=0.5
FC	#Classes (=10) units

Table 5.2: LeNet-2 architecture, used in Zhang et al. [**zhang2015SVD**].

The first part of this model is made of $\frac{N-1}{2}$ convolutional layers (with N equals to the patch size), each one followed by a Rectifier Linear Unit (ReLU). Each convolutional layer contains F filters of size 3×3. No padding or stride is applied, making the final output of the convolutional layers, a F×1×1 tensor (each layer reduces the initial size N by 2 pixels), directly forwarded to the fully-connected part of the network deploying two layers, made of L neurons each, followed by ReLUs (1). The final layer collapses into a single neuron in charge of the regression. According to a common methodology usually deployed when dealing with deep architectures, the authors replaced fully-connected layers with convolutional layers made of L 1x1 kernels. This makes possible to train the network on image patches as well as to compute a dense confidence map with a single forward of the full resolution image with a 0-padding of $\frac{N-1}{2}$ around it, keeping for the output the same w×h size of the input disparity map due to the absence of pooling operations or stride factors inside the convolutional layers.

The CCNN architecture is described in figure 5.1.3.

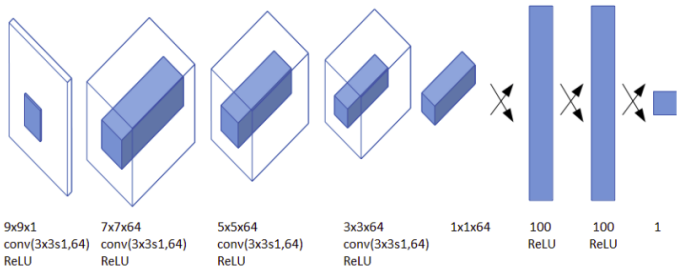


Figure 5.1: Architecture of CCNN to infer the CM from from the raw disparitymap. It is a single channel network, designed for 9×9 image patches. Four convolutional layers apply 64 overlapping kernels (stride equal to 1) of size 3×3. Two fully-connected layers made of 100 neurons each (i.e., 100 1×1 convolution kernels) lead to the final regression node.

.

5.1.4 Network-In-Network

The Network-In-Network model (NIN) has been proposed in [**NIN**] as a peculiar model which first leveraged on one-by-one convolution, explained in chapter 3. It features a fully-convolutional architecture with three main layers that can be decomposed, conv1, conv4 and conv7.

The modified version, used also in Zhang et al [**zhang2015SVD**], is depicted in figure 5.2.

5.1.5 Core strategy

As previously mentioned, the experiments follow the micro-architecture pattern with the goal of testing the effectiveness of the TD-Block on both new architectures trained from scratch and pre-trained models.

Layer name	NIN	Low-rank NIN
conv1	$5 \times 5 \times 192$	$K_1 = 10$
conv2,3	$1 \times 1 \times 160, 1 \times 1 \times 96$	
conv4	$5 \times 5 \times 192$	$K_2 = 51$
conv5,6	$1 \times 1 \times 192, 1 \times 1 \times 192$	
conv7	$3 \times 3 \times 192$	
conv8,9	$1 \times 1 \times 192, 1 \times 1 \times 10$	

Figure 5.2: Network in Network architecture. The low rank version is the one by Zhang et al[**zhang2015SVD**]

TD-Block design

As for the model design, different TD-block configurations have been tested. The base model is the one depicted in **??**. The middle convolution block can be either separable or regular spatial convolution, depending on the type of decomposition.
Since this is an arbitrary design choice, the following points were weighed the decision:

- since we are going to train from scratch, there is no particular reconstruction error constraint to recover.
- the goal is to find the most effective way to compress a CNN.

Therefore the type of convolution in the middle have always been the separable one suggested in section 4.3.1 of chapter 4, proposed by Lebedev et al. [**lebedev**]. This configuration as a building block, represents an original idea that has not been fully explored, while being also the more aggressive compression.

Two main variations on the original TD-block were tried:

(i) adding a BatchNorm layer between each convolution;

(ii) adding ReLUs plus BatchNorm between each convolution.

This design scheme has been tested on LeNet-1 and CCNN.
Note that other configuration are possible and will be explored in future work.

TD-Block compression

For each model, compression has been tried in both the Tucker and CP way, as illustrated in chapter 4.

Given that Tucker decomposition is more conservative, in the experiments most of the time, it represents a good trade-off for fast convergence and good approximation.

On the contrary, CP-decomposition is more aggressive and during the experiments it has been pushed by enforcing smaller ranks compared to Tucker, and thus getting a more compressed model. As indicated in the framework for tensor decomposition proposed in 4.4.5 of the previous chapter, sometimes VBMF estimates ranks that

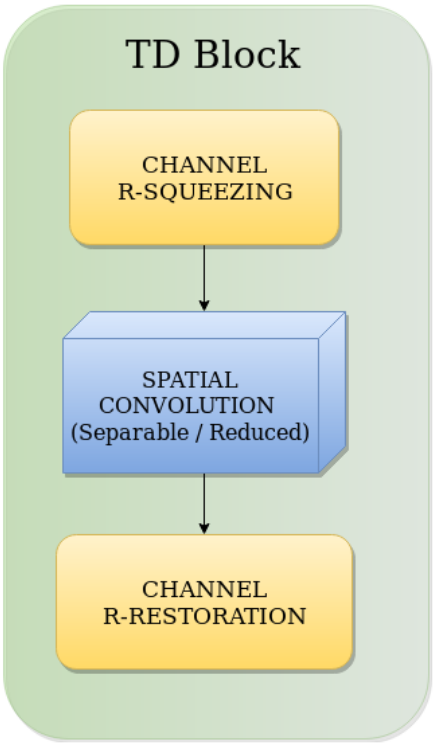


Figure 5.3: The Tensor Decomposition (TD) block proposed in chapter 4. Variations of these block have been tried several times, with ReLU and batch normalization layers in between the three stages.

are too high, in the sense that it is possible to set a lower rank and then gamble on fine-tuning to recover (in higher time) the same accuracy.

Weight initialization

As mentioned in chapter 3, weight initialization is important to get a faster convergence to the global minimum. The initialization is, in fact, together with the optimizer, the main tool that is able to change how the BackProp algorithm starts and ends through the loss.

After decomposing a tensor, the weights need to be adjusted to recover the accuracy; therefore, it is also possible to exploit the structure of the decomposition but initialize the weights in a different way. The former would assume that, since the TD-block is appropriate to substitute that specific layer, a uniform random initialization could also reach a global minimum after few iteration.

Xavier Xavier initialization makes sure the weights are "just right" (not too large neither too small), keeping them in a reasonable range. Basically, Xavier init selects the adequate standard deviation in order to avoid the well-known issues of vanishing and exploding gradients.

LAYER	CHARACTERISTICS
TD-Block	$32 \times [1,1] + R \times 3 + 3 \times R + 32 \times [1,1]$
ReLU	-
TD-Block	$32 \times [1,1] + R \times 3 + 3 \times R + 32 \times [1,1]$
ReLU	-
POOL	pool size [2, 2] stride=2
Dropout	p=0.25
TD-Block	$64 \times [1,1] + R \times 3 + 3 \times R + 64 \times [1,1]$
ReLU	-
TD-Block	$64 \times [1,1] + R \times 3 + 3 \times R + 64 \times [1,1]$
ReLU	-
POOL	pool size [2, 2] stride=2
Dropout	p=0.25
TD-Block	$64 \times [1,1] + R \times 6 + 6 \times R + 512 \times [1,1]$
TD-Block	$512 \times [1,1] + R \times 1 + 1 \times R + 10 \times [1,1]$

Table 5.3: Architecture of LeNet-1 with CP-like TD-blocks. Each TD-block has four convolution layers. The number of parameters is reported in the characteristics column. Note that, each TD-block has BatchNorm layers between each convolution.

As suggested for future work by Kim et al. [**Tucker-mobile**], Xavier init has been tried throughout the experiments.

5.2 Experiment 1: TD-Design LeNet-1

This experiment aims at discovering the effectiveness of the TD-block design.

Training was done for 50 epochs, with Adam optimizer and a learning rate $\eta = 0.001$ decreasing by 0.1 every 20 epochs. In this way, the baseline model reached an accuracy of 74%.

5.2.1 TD-Block architecture

The modified architecture of LeNet-1 has two main differences: (i) it converts the FC-layers to convolutional ones and (ii) substitutes each of the layer with its corresponding TD-block, with a CP-like configuration.

The full decomposed architecture has only 27K parameters, boasting a 45× compression ratio. The rank has been set differently for the first convolution layers (lower rank) and the last ones (higher rank), with values in the range $[10-40]$ Details about the architecture are reported in table 5.3.

5.2.2 Original vs. Decomposed

Training with the same setup leads to the result shown in figure **??**.

Clearly, the TD-block design outperforms the baseline by a significant margin, by reaching an accuracy of 82% against the original 74% of the LeNet-1 model.

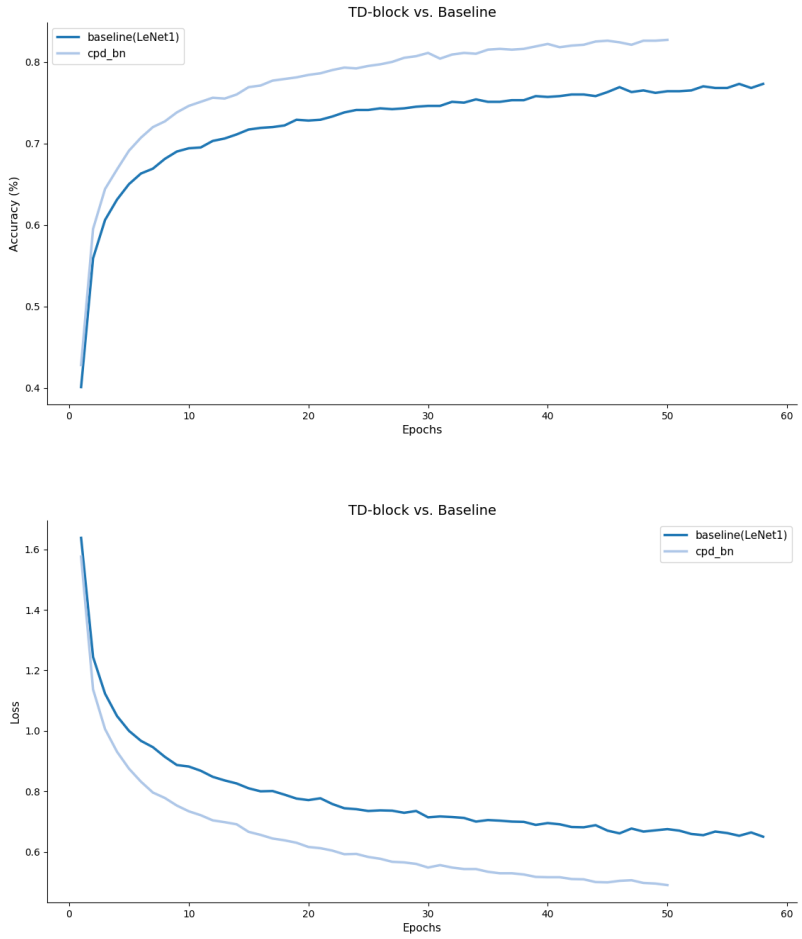


Figure 5.4: TD Architectures comparison for training accuracy and training loss.

By looking at the training loss we can see that it also converges faster to a better minimum.

5.2.3 TD configurations comparison

To avoid confusion the CP-like configuration of the TD-block will have a CP-prefix. That aside, three main configuration of the TD-block were tested:

1. CP-BN: `CONV-BN-CONV-BN-CONV-BN-CONV-BN`
2. CP-BN-RELU: `CONV-BN-RELU-CONV-BN-RELU-CONV-BN-RELU-CONV-BN-RELU`
3. CP-BN-Xavier: same as the CP-BN but with Xavier initialization.

As can be notice, the original configuration from Lebedev et al [**lebedev**] is missing. The main issue is that in the proposed all TD-block architecture 5.3, the number of convolutional layers skyrockets from 4 to 24 ($\times$ 4, plus the conversion of the FC layers). Therefore, it probably incurs in the vanishing gradient problem, as the training is too slow to converge.

The comparison between the three proposed configurations is shown in figure. 5.5

RELU effect : As can be seen, the RELU configuration performs noticeably worse than the other two, which are closely comparable. As also found by F. Chollet in the context of depthwise separable convolution[**chollet**], the absence of non-linearity leads to both faster convergence and better final performance. According to the author, it may be that the depth of the intermediate feature spaces on which spatial convolutions are applied is critical to the usefulness of the non-linearity: for feature spaces with many channels the non linearity is useful to enhance the most important ones, but for shallow ones - the CP-like architecture employs a 1-to-1 channel connection - it becomes possibly harmful, due to a loss of information.

In other words, when the information contained in the layers is redundant non-linearities helps to extract the important features and squash away the rest; on the other side, the filters in the TD- block are so few that are "all important" and RELU would cut away important information.

On Xavier init : Surprisingly, Xavier initialization did not improve the overall performance of the network. At testing time the two configurations were similar with the regular one getting an 82% and the xavier one an 81% of accuracy. This may be due to the heavy batch normalization use amidst the convolutional layers that already helps convergence a lot.

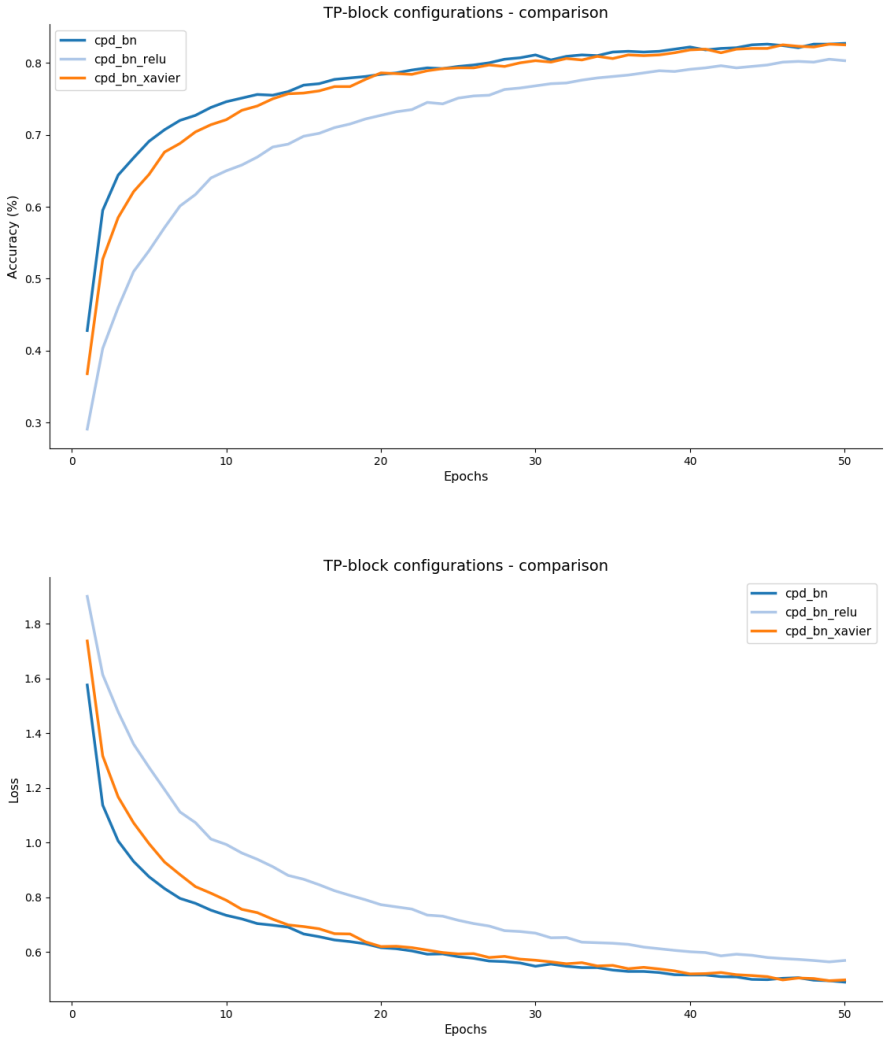


Figure 5.5: TD Architectures comparison for training accuracy and training loss.

Model	Parameters	Test Accuracy	$\mathcal{C}_r$
LeNet-1 (baseline)	≈ 1.2 Million	74%	-
CP-BN	≈ 27K	82% (+8%)	45 ×
CP-BN-Xavier	≈27K	81% (+7%)	45 ×
CP-BN-RELU	≈ 27K	77% (+3%)	45 ×

Table 5.4: Accuracy and parameters for the baseline and the different configurations of TD-block models.

5.2.4 Overall

The TD architecture have proven to converge faster and to be able to generalize better than the baseline model, while having only 2% of the original number of parameters. This leads to few observations:

- the very low number of parameters acts as a form of *regularization*, preventing the model to be overly complex, thus avoiding over-fitting.
- There is a very high percentage of redundancy between the channels of the CNN. This is in line with what has been explained in chapter 3 about network design.
- There is an exploitable correlation also between the 9 (in this case) values of each kernel. This is a good hint for future applications of separable convolution (see chapter 3, section 3.2.4), that will provide an even more effective compression when the kernel size is 5 of higher.
  Differently from the previous point, this architecture has not been exploited yet in literature.
- FC layers can be converted and then decomposed as well, going from 1.18 millions down to ≈8 thousands. The latter, of course, brings most of the compression ratio.

Table 5.4 reports a summary of the results.

This is a promising result and says something about how many parameters in a CNN are actually necessary. Of course, it also depends on the complexity of the dataset, nevertheless a good approximation seems to plausible even for larger models.

5.2.5 Future ideas

With regard to the loss of information caused by RELU, an idea may be suggested by deep residual networks **??**. By summing up the original input of the TD block to the output, as in Resnet, the loss of information could be recovered. Furthermore, it may even improve overall accuracy. Therefore, a residual learning configuration of the TD-block could be investigated in future work.

5.3 Experiment 2: TD-Compression LeNet-Conv

In this experiment we are going to actually compress a similar architecture to one trained from scratch in the previous experiment. During the experiments, few questions came out like which layer should be compressed before; in which direction should the compression go and such. The results of this compression could be helpful to setting up guidelines for the future.

Decomposition pipeline Experiments have shown there is no much difference between the order of the decomposition, as long as the decomposition are computed accurately, that is.

Basically, four strategies can be outlined:

1. *ordered*: from first layer to the last;
2. *reversed*: from last layer to the first;
3. *larger first*: the first layer to be compressed is the one with the highest number of parameter. This can be tricky, but once it is optimized, the other layers should not cause many troubles even with aggressive compression rates.
   The downside of this method is in the pipeline strategy itself. If the compression is performed on each layer subsequently and in an automatic manner (e.g. executed by an automated script) without a supervision, it could result in a bad compression. This is because the largest layer is usually the hardest to be compressed accurately and if the approximation is too harsh, it could be impossible to recover. Consequently, the error propagates on the subsequent compressions as well, resulting, as said, in a drop in accuracy.
4. *smaller first* layers get compressed in ascending order of number of parameters. This is probably the safest bet to achieve a good compression. However, it also hides a pitfall. For instance, let A,B and C three convolutional layers with 100K, 700K and 1.7M parameters respectively. Let's say the compression rate has been kept low on A and B in order to do a very aggressive compression only on C (that will contribute more on the overall size), and let C be particularly sensitive to the overall accuracy of the network. In this latter scenario, we could find that the only good approximation of C is a very soft one and thus ending up with very few parameters less than the original model.

Results of CP-decomposition in both ordered and reversed way are shown in plots 5.6. As it can be seen, results are almost stackable.

5.3.1 CP

Contrary to what usually reported in literature[**lebedev**], Parafac decomposition was comparable to Tucker stability for this experiment. Especially on smaller layers, CP

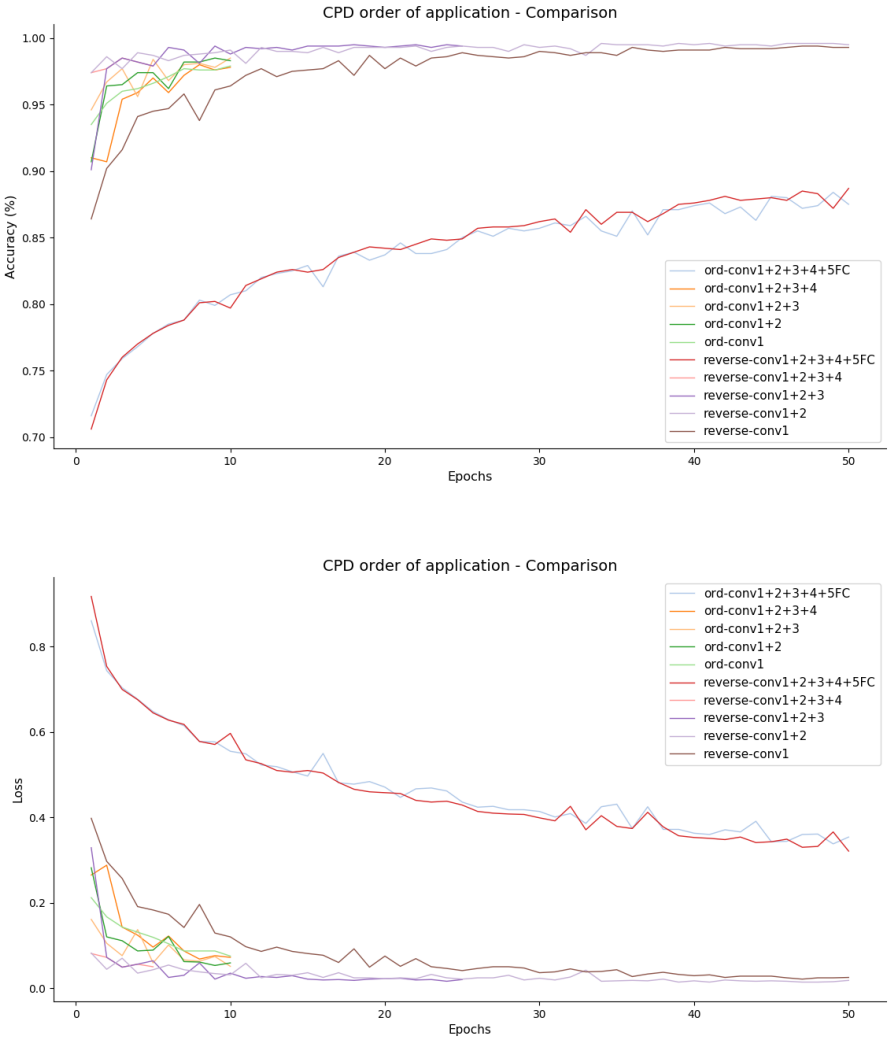


Figure 5.6: CP-decomposition: the order of the layers decomposition does not affect overall performance.

Parafac Decomposition					
Layer	**Start**	**Fine-tuned**	**Params**	**Compres.**	$\mathcal{C}_r$
CONV4	Acc: 79% Loss : 0.193	Acc: 77% Loss: 0.023	36928	2878	**13x**
CONV(4+3)	Acc: 77% Loss: 0.082	Acc: 77% Loss: 0.018	18496	2206	**8.5x**
CONV(4+3+2)	Acc: 76% Loss: 0.300	Acc: 77% Loss: 0.033	9248	732	**13x**
CONV(4+3+2+1)	Acc: 77% Loss: 0.081	Acc: 77% Loss: 0.033	896	442	**2x**
CONV2FC1 Rank=170	Loss: 0.738	Acc: 80% Loss: 0.195	1180160	100472	**12x**
CONV2FC1 Rank=50	**Loss: 0.981**	**Acc: 80% Loss: 0.321**	**1180160**	**29912**	**40x**
Overall	**Acc: 79%**	**Acc: 80%**	**1252480**	**42922**	**29x**

Table 5.5: A comprehensive summary of the CP-decomposition on LeNet-Conv. Start indicates the testing accuracy and training loss before decomposing the layer, while the fine-tuned column shows the same metrics after the decomposion+fine-tuning step. In bold the most interesting results.

converges very fast (5.6), as Tucker usually do.

When dealing with the FC2Conv layer with more than a million parameters, the training is not as smooth but it still converges to a global minimum (loss is 0.19), improving the baseline results by 1% while compressing the whole model by boasting a 29× compression, with a peak compression rate $\mathcal{C}_r$ of 40× on the larger layer.

A recap of CP results is in table 5.5. Note the difference between the suggested VBMF rank for the last layer (R=170) and the rank imposed by selecting the desired compression ratio (R=50).

5.3.2 Tucker

Tucker achieved similar results but with a weaker compression, probably due to the conservative rank estimation of VBMF. Enforcing of Tucker ranks by selecting a desired compression ratio has been also implemented in the 'decomposer' class but did not achieve the same accuracy of CP at this time.

Loss and training accuracy can be observed in figure **??**. Training have been done with early stopping, that is why the first layers decomposition stopped around epoch 10, when they actually reach the minimum. The largest layer (FC2CONV) has more than 1 million weights, that is why its convergence takes much more. Nevertheless ,

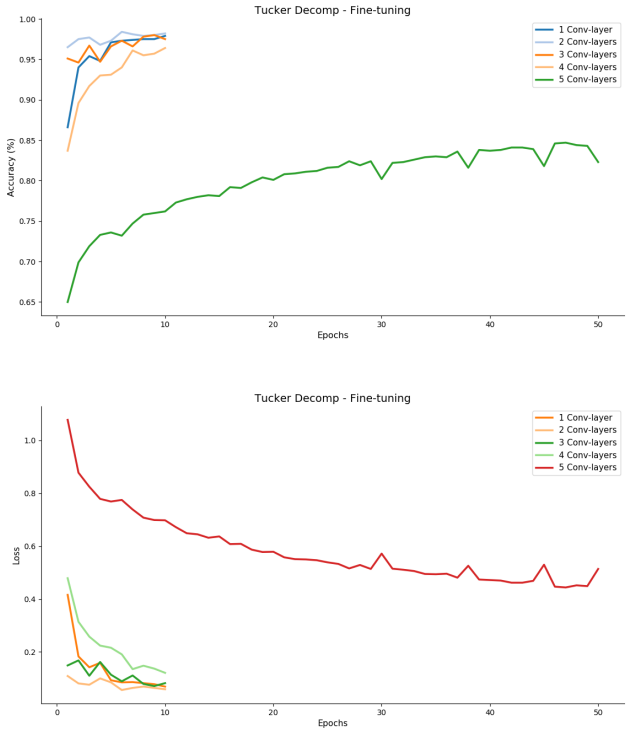


Figure 5.7: Tucker decomposition: most of the layers reach the global minimum very fast. For obvious reasons, the largest layer comprised of 1.18 million parameters takes more time to converge.

the decomposition seems to bring new benefits to the generalization capacity of the model, as reported in the previous experiment. In fact, the overall accuracy on the test set at the end of the decopmosition is of 77%.

Mixed Decomposition : interestengly, it is also possible to mix Tucker and CP in a gradually more aggressive decomposition. This is perfomed by first applying Tucker on the full convolutional layer, and then applying CP decomposition on the central block of the Tucker decomposition. The latter, in fact, features a squared regular spatial convolution that can be further accounted by CP decomposition. This scheme was tried only after the last convergence of Tucker and it managed to reduce the size of the layer by another 5X while showing *no* accuracy drop (i.e. 77% as before).

5.3.3 Xavier

It has also tested the CP-Xavier method: applying CP-decomposition architecture first (without computing the actual weights) and initializing with Xavier uniform distribution.

Training loss is shown in 5.8. Xavier converges as fast as CP in this case but being much smoother, and while fluctuating a bit on testing (77-74-75-77) at the

end achieves an optimal accuracy of 79% as Tucker. Hence, this means there is no accuracy drop.

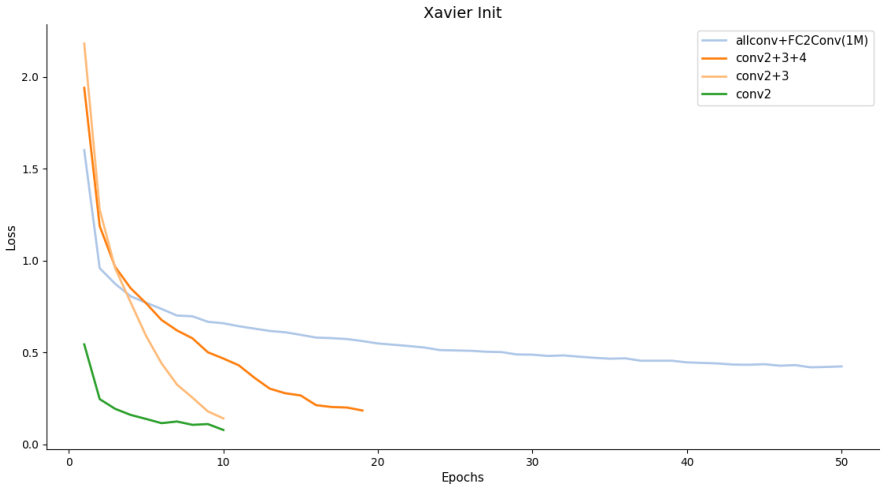


Figure 5.8: CP decomposition with Xavier initialization. Convergence as fast as the other methods, but in a smoother way, confirming the benefit of this type of initialization.

5.3.4 Overall

This experiment have shown how the order of the decomposition does not affect the final result as long as the approximation can be recovered by fine-tuning for 10 to 50 epochs. Tucker and CP-Xavier had achieved similar results on both compression and accuracy while CP decomposition managed to improve accuracy while having a whopping ≈ 30 less parameters.

Overall, all decomposition methods showed promising results. Also, mixed decomposition could represent a combination of Tucker conservative decomposition and CP aggressive one into one pipeline.

5.4 Experiment 3: TD-Compression LeNet-2

This second LeNet configuration has more dense layers that can be harder to compress but that could also lead to a higher compression rate. In this experiment we will also see how Xavier init compares with CP decomposition with different results compared to the previous experiment. The training of the baseline model have been done without data augmentation (differently from [**zhang2015SVD**]) and the baseline model reached an accuracy of 74% in 50 epochs. That must be kept in mind for comparison results.

The only layers that have been decomposed are the first three convolutions, in reversed order (conv3-conv2-conv1). However, it must be noted that the last fully-connected layers can be decomposed by applying SVD on its matrices, as explained in chapter 4, section 4.1.4.

5.4.1 CP

CP decomposition performs very well on this network too, confirming the results of the previous experiment. The convergence of all the three decomposition is shown in 5.9. Interestingly, the decomposition of all three layers converges better than the one of only the second and the one. This may be due to some kind of cross-layer redundancy that can be improved by decomposition only on that specific junction of layers.

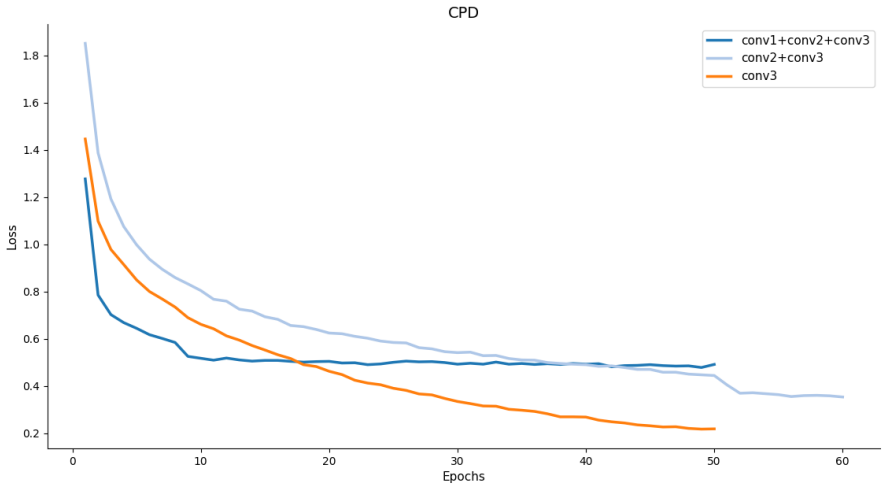


Figure 5.9: CP decomposition on LeNet-2.

By compressing the whole model of 4.3× and the largest layer of over 30× it is able to achieve an accuracy on the test set of 82%, which a +3% improvement on baseline model. Remarkably, when enforcing a compression rate of 52X on conv3, as reported in [**zhang2015SVD**], CP managed to achieve a 77% accuracy on the test set, which is still better then baseline.

More details are shown in table 5.6.

5.4.2 Tucker

Tucker decomposition improved the accuracy as well, accounting for +6%. The overall compression ratio is much smaller than CP when both uses VBMF rank estimation. However, Tucker proved to be more stable and training took less iterations than the CP correspectives. Sometimes 5 epochs against 30-40 required by CP. Furthermore, when selecting a very high compression rate, like 63X, on the third convolutional

Table 5.6: CP-overall compression results

Layer	Accuracy	$\mathcal{C}_r$	Speed-up
Conv1	80% (+6%)	4X	2.3X
Conv2	82% (+8%)	31X	1.7X
Conv3 R=VBMF	80% (+6%)	25X	1.2X
Conv3 R=39	**77% (+3%)**	**52X**	**1.5X**
Overall	**81% (+7%)**	**4.5X**	**3.25X**

Table 5.7: Tucker overall compression results.

Layer	Accuracy	$\mathcal{C}_r$	Speed-up
Conv1	75% (+1%)	0.77X	2.7X
Conv2	79% (+8%)	6.3X	1.7X
Conv3 R3,R4=VBMF	80% (+6%)	14.5X	1.2X
Conv3 R3=6, R4=41	**76% (+2%)**	**63X**	**1.5X**
Overall	**80% (+6%)**	**3.5X**	**3.25X**

layer, the training managed to converge as well, to a 76%. The latter is an improving on baseline while providing an aggressive compression as well.

5.4.3 Xavier comparison

After several experiments, an interesting point about Xavier init came out. Depending on the quality of the decomposition, CP can converge faster and to a better minimum. This may be due to initial solution, ironically, of the tensor decomposition algorithm itself. An example is shown in figure 5.10, where two different CP decompositions, A and B, for the same layer converge in different ways; xavier init trajectory is almost identical to B, but inferior to the better A-CP decomposition. On the other hand, Tucker decomposition is more stable and converge much faster than a random xavier init. As shown in figure 5.11.

5.4.4 Overall

The reported results seem to improve upon the one reported by Zhang et al [**zhang2015SVD**]. However, the models may be not fully comparable, due to the lack of data augmentation and the different frameworks.

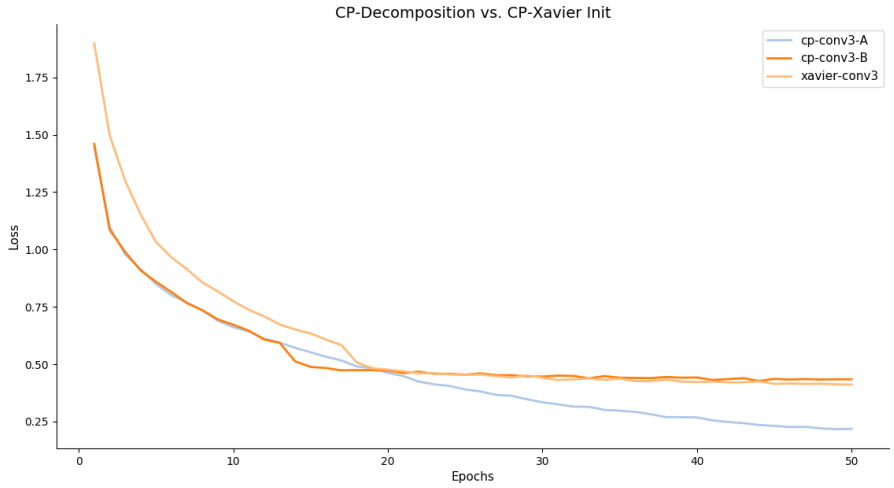


Figure 5.10: CP vs. Xavier init comparison for different decomposition of the same layer.

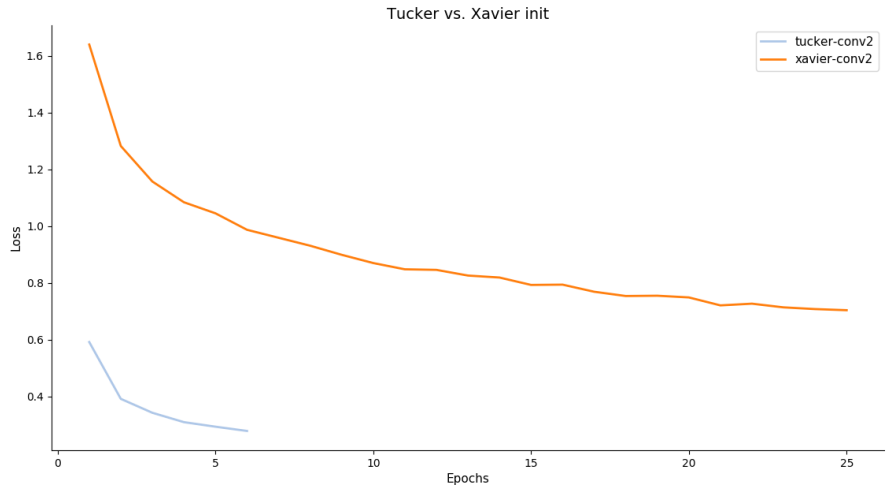


Figure 5.11: Tucker vs. Xavier init comparison for conv2 layer.

Noticeably, even with very high compression rates (> 50x) both decomposition techniques managed to achieve an improvement over baseline, with CP getting higher accuracies in the case of such aggressive compression ratios.

5.5 Experiment 4: TD-Compression NIN

NIN models introduced in the previous sections, is able to get to 86% on CIFAR-10 without data augmentation. This specific model can be decomposed only on layer 1, 4 and 7. The other layers are all one-by-one convolutions and thus not suitable for tensor decomposition.

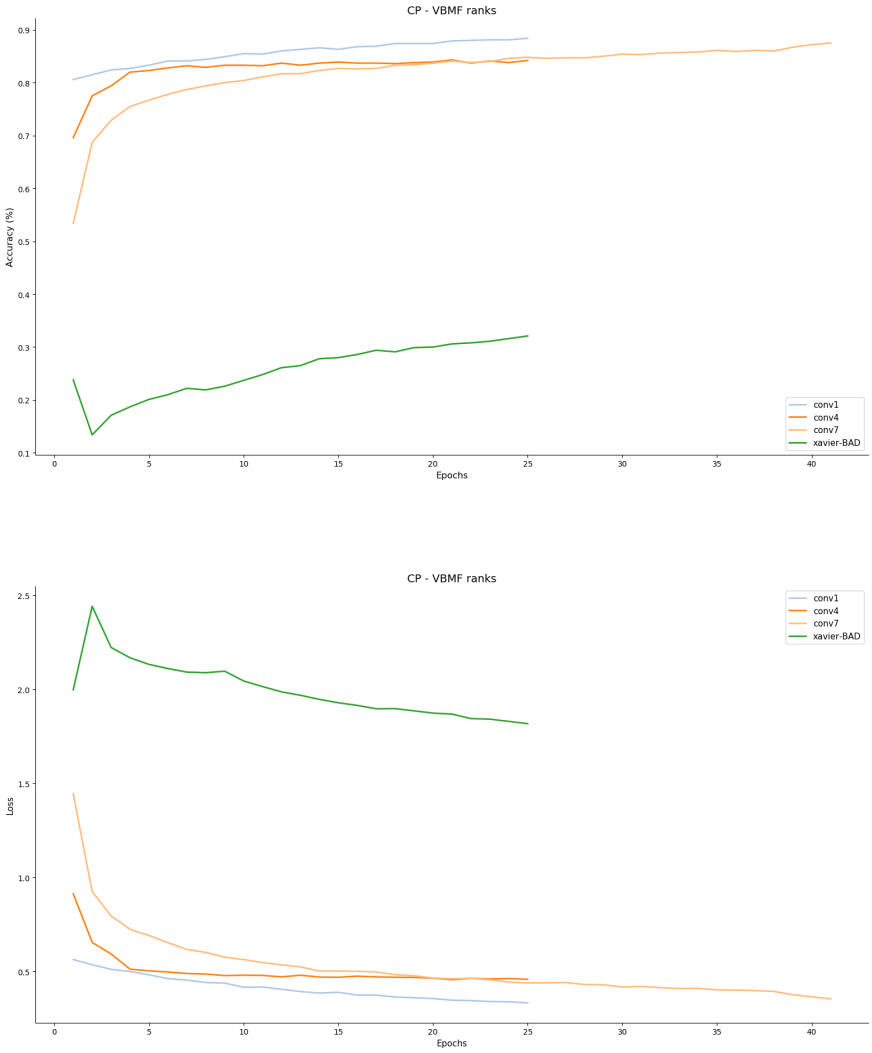


Figure 5.12: Training metrics for CP-decomposition of NIN model.

5.5.1 CP

CP-decomposition achieved a 5X compression ratio while improving the overall accuracy by 1 point. The training curve in this case was also smooth. Training results are shown in 5.12 while details about compression ratio and overall speedup are reported in table 5.8.

5.5.2 Tucker

Tucker has proven again to be stable and consistent among the decompositiong. Choosing the ranks with VBMF, it compressed NIN by 2x. Training metrics are shown in figure 5.13. More details are in table 5.9.

Table 5.8: CP overall results on NIN.

Layer	Accuracy	$\mathcal{C}_r$	Speed-up
Conv1 R=14	86% (+0%)	4.7X	2.0X
Conv7 R=64	87% (+1%)	12.8X	1.7X
Conv4 R=100	86% (+0%)	15X	1.2X
Overall	**87% (+1%)**	**~5X**	**2.5X**

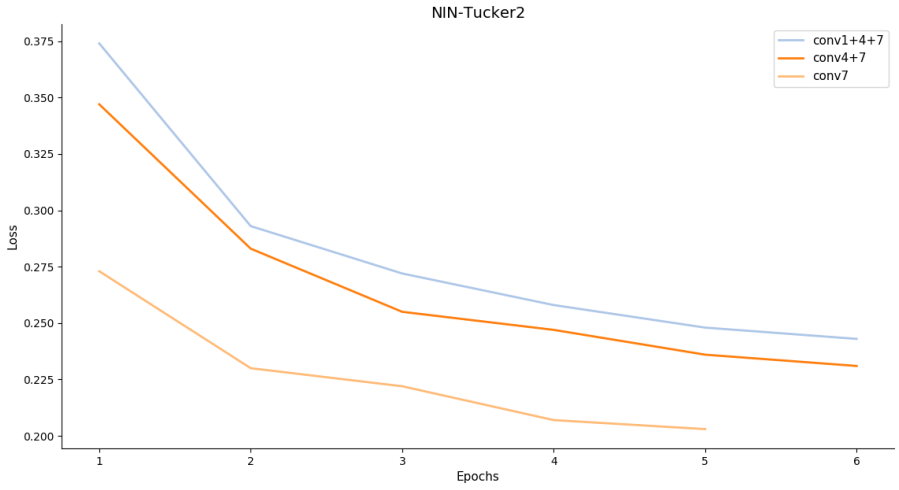


Figure 5.13: Tucker compression on NIN's three decomopsable layers. Note how bad Xavier init performes on one of them.

5.5.3 Overall

With this model CP decomposition got better results, both in compression and in final accuracy. Both methods have shown no drop in accuracy compared to the baseline NIN.

5.6 Experiment 5: TD-Design CCNN

With respect to real-time application, an interesting scenario is represented by 3D stereo vision. Stereo vision is a popular technique to infer depth from two or more images. In this field, confidence measures aim at detecting uncertain disparity assignments. From the observation that recurrent local patterns which occurs in disparity maps can tell a correct assignment from a wrong one, CCNN models the confidence formulation as a regression problem by analyzing disparity maps generated by a stereo

Table 5.9: Tucker compression results on NIN.

Layer	Accuracy	$\mathcal{C}_r$	Speed-up
Conv1	86% (+0%)	1X	1.4X
Conv7	86% (+0%)	6.7X	1.7X
Conv3 R3,R4=VBMF	86% (+0%)	7.5X	1.2X
Overall	**86% (+0%)**	**∼2X**	**2.5X**

vision system.

CCNN has only $\approx 128K$ parameters and generats a full confidence map in only 630 ms, thanks to its fully convolutional architecture. Being already fast and small, CCNN represents a challenging testbed for tensor decomposition design. Nevertheless, the experiments shows that it can be improved by applying CP-like tensor decomposition blocks.

5.6.1 TD-Block architecture

As it is already a fully convolutional architecture, it is relatively easy to re-design CCNN with the TD-block architecture described in section **??**. Thus, the so modeled architecture will feature four TD-blocks made of 4 convolutional layers each, but, differently from the design of experiment 1 **??** *without* batch normalization layers in between.

Two different models were tested:

1. CCNN with 4 TD-blocks, one per each convolution layer and the rest of the model unchanged: `TD-RELU-TD-RELU-TD-RELU-TD-RELU-FC1-FC2-REGRESSION`$_{HEAD}$
2. CCNN with 4 TD-blocks plus a further decomposition of the FC-Conv layers into a combination of two smaller layers, according to an approximation inspired by singular value decomposition (SVD) **??**.

For each of the TD-block the compression ratio gain $\mathcal{C}_r$ is given by

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

, where S, T stand for input and output maps dimension and d for the kernel size, respectively and R is the decomposition rank.

As for the SVD decomposition scheme, the idea is depicted as follows:
Given a FC layer of size $A \times B$, it is possible to divide it into two smaller layers of size $A \times R$ and $R \times B$ respectively. In this way the complexity goes from $S \cdot T$ to $R(S + T)$. Hence, a complexity advantage is guaranteed as long as $R < \frac{S \cdot T}{S+T}$.

MODEL	PARAMS	AUC (Test Set)
CCNN (baseline)	128K	0.1222
TD Design 1	36K	0.1192
TD Design 2	**27K**	**0.1186**
Optimal	-	**0.1073**

Table 5.10: AUC evaluation on 174 images of the KITTI stereo benchmark dataset. The smallest model outperforms the baseline.

For both schemes, the rank R was set to be equal to 20, resulting in 36K parameters for the first decomposed model and 27K thousand for the second.

5.6.2 Training

The network was trained for 14 epochs on 9×9 patches of the first 20 images of the KITTI stereo dataset, providing approximately 2.6 million samples.
The loss function to minimize is the Binary Cross Entropy (BCE) between the output o and a label t on each sample i of the mini-batch.

5.7 Evaluation Results

Evaluation assessment was done by ROC curve analysis, which is commonly used when dealing with confidence measures. After ROC curves are depicted for each image, the Area Under The Curve is then used to evaluate the capability of the confidence measure to distinguish correct disparity assignments from erroneous ones, with respect to the optimatl solution.

For the two models described above, the evaluation gives optimal results. In fact, both architectures outperforms the baseline as reported in table 5.10. Remarkably, the smaller architecture was the best scorer.

For each model, the AUC comparison with the optimal solution is depicted in 5.14. We can see that the models go very close to the optimal solution.

5.7.1 Discussion

These last results are surprising, as the baseline had already very few parameters compared to typical CNN architectures. This suggests that, after all, some sorta of redundancy must is present in both the kernels and the channels. It may due to the fact that CCNN process overlapping 9×9 patches that could contain redundant information that can be exploited by cross-channel pooling, as performed by the TD-block design.

Moreover, the fewer parameters may act as another form of regularization and thus enhances the generalization capabilities of the network. Further investigations are needed to deeply understand the decomposed architecture's benefits, such as applying a different kind of channel pooling (e.g. varying the stride) and then test if the same results still applies. This is left as future work.

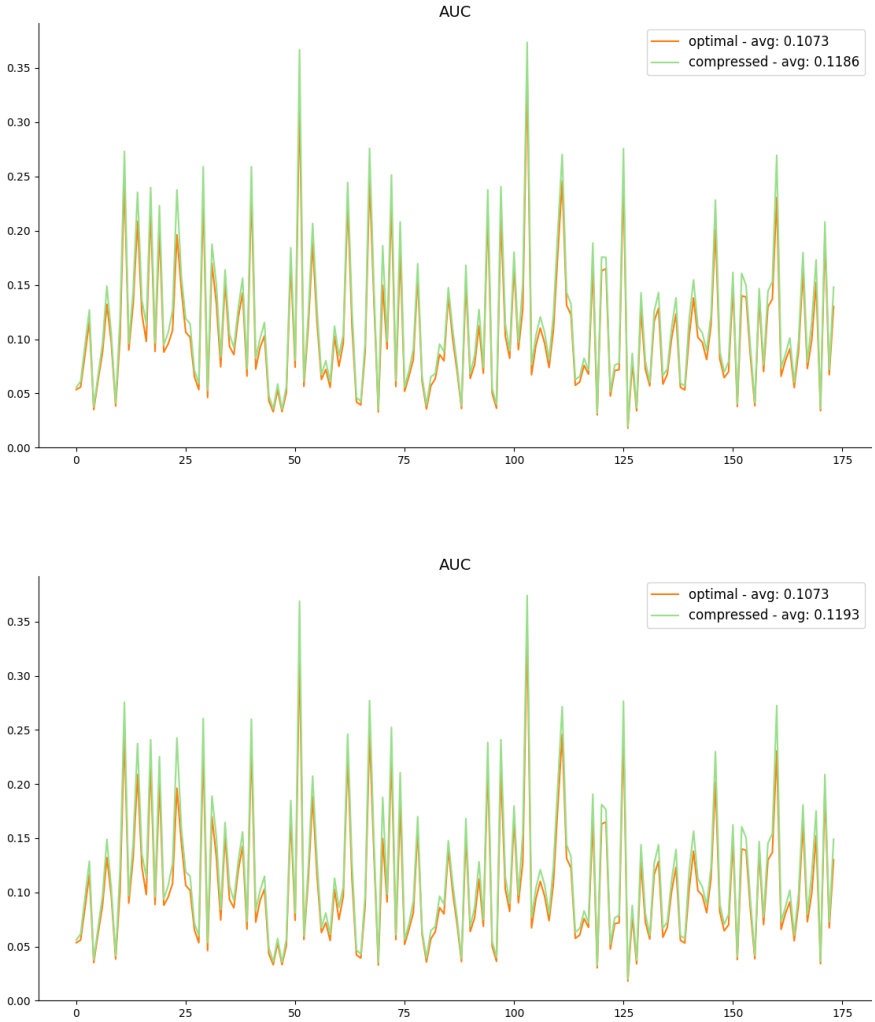


Figure 5.14: Area Under the Curve (AUC) computed per each test image for the two decomposed models (in green) and the optimal solution (in orange).

5.8 Analysis of the results

Most experiments have proven that tensor decomposition is a good candidate for the task it has been chosen for. For both Tucker and CP, different type of networks have never shown drop in accuracy. Xavier initialization however, even if sometimes beneficial and easier to implement, did not achieve good results on NIN and LeNet-2 models, compared to an accurate decomposition, that is.

Tucker has been generally more stable than CP, showing faster convergence and thus being more reliable. However, CP decomposition has almost always been a better fit to perform an aggressive decomposition. The latter, though, needs more iterations to converge and thus may not be always feasible to perform. A curios pattern regard that of mixed decompositions that gradually perform conservative and aggressive decomposition, towards which more investigation is needed.

As for the objectives given in chapter 4 with respect to the TD-block, the expectations have been met in both model compression and model design. This TD-block pattern, managed to improve the accuracy of LeNet-1 by 8% despite having 45x less parameters. Moreover, it achieved a remarakable result on the CCNN testbed, improving the baseline - which is state-of-the-art - by a fair margin, while having 6x less parameters.

Chapter 6

Conclusions and future work

This thesis have conducted an in-depth investigation on tensor decomposition techniques, providing insights on their application intrinsics, making them less cumbersome to use with respect to Convolutional Neural Networks. In this regard, diverse strategies have been explored, both in the number of decomposition stages and techniques and in the pipeline employed to actually perform them. Following the principles of compact and efficient design of Convolutional Neural Networks, a uniform design for Tensor Decompositions (TD) blocks has been proposed at the end of Chapter 4.

The experimental results presented in Chapter 5 have successfully validated the proposed TD-block design, showing promising developments. Five different models have been analyzed, throughout a structured discussion about the advantages and pitfalls of one strategy respect to the other. Tucker decomposition have proven to be more stable and almost always to converge faster while providing a decent compression rate.
Although CP-Decomposition needs, in general, more iterations to converge, it boasts a more effective way to achieve higher compression ratio, providing that the decomposition is accurate. If that isn't the case, it could get stuck in a local minimum and not being recovarable. To solve this problem, the experiments have shown that inserting Batch Normalization layers in-between the convolutional layers of the TD-block can help to avoid local minimum and eventually find the global one. Through these techniques, the compression ratio achieved was surprsingly high, with peaks of $(40 - 50\times)$ less parameters of the baseline models.

As for the TD-block design only, its application as a building block of LeNet-1 in the first experiment and of CCNN in the last one have shown remarkable results. The former resulted in an overall accuracy improvement of +8 points while being $45\times$ smaller; the latter have improved the accuracy of confidence measure prediction over baseline model on the KITTI Stereo Evaluation dataset, having a fifth of the baseline parameters.
This last result is important in two ways: first, it shows a new way of designing CNNs that are both smaller and more effective; second, it is one of the very few results tested

on a visual task different than pure image classification.

Many points came out that need to be addressed in future work. First of all, the rank selection. Even though a framework for tensor decomposition has been proposed in Chapter 4, the methods still requires too much manual tuning. VBMF proved to be a good tool for rank estimation when we want to preserve the accuracy; however, if a stronger compression rate is desired, manual tuning is required.
Moreover, the decomposition does not take into account the global optimization of all layers, which can cause problems.

A future direction could involve a *learning-to-learn* strategy [**learn**] in which the optimal rank compression rate is learned by another model. A possible candidate could be reinforcement-learning.
Another possible research would revolve on improving the TD-block design, by adding, for example, residual learning.

In the lights of what has been said, tensor decomposition remains amongst the best candidate to compress CNNs in being perfect to embded into the CNN design space.