Alma Mater Studiorum · Università di Bologna

SCHOOL OF SCIENCE
Laurea Magistrale in Matematica
Curriculum Advanced Mathematics for Applications

ANALYSIS AND IMPLEMENTATION OF DICTIONARY LEARNING TECHNIQUES IN A DIGITAL TWIN FRAMEWORK

Supervising Professor:
Chiar.ma Prof.ssa
Margherita Porcelli

Co-supervisor:
Dott.ssa
Domitilla Brandoni

Presented by:
Laura Cavalli

III Session
Academic year 2022/23

This work is supported by the EUROCC Italy
National Competence Center.
The Competence Center is part of EUROCC project
funded by the European High-Performance
Computing Joint Undertaking (JU)
under grant agreement No 101101903.

Contents

Abstract

Negli ultimi anni, la tecnologia dei Gemelli Digitali (Digital Twins) sta svolgendo un ruolo cruciale in diversi contesti, dalla agricoltura intelligente alla manutenzione predittiva, dalla sanità alla modellazione meteorologica, ottimizzando processi e prevenendo eventuali guasti degli apparecchi. Un Gemello Digitale consiste in una replica virtuale di un oggetto fisico, di un sistema o di un processo, aggiornata in tempo reale attraverso uno scambio continuo di enormi quantità di dati tra sensori IoT (Internet of Things) nel mondo reale e il sistema digitale. Questi dati posso essere utilizzati per addestrare sul sistema digitale modelli di intelligenza artificiale che, eseguendo analisi automatiche sui dati, sono in grado di prendere decisioni e attuarle sul sistema reale. Di conseguenza, una gestione efficiente dei dati diventa essenziale per sfruttare al massimo il potenziale di questa tecnologia. Questa tesi, pertanto, mira a presentare un metodo di compressione basato sulla tecnica di Dictionary Learning (DL). Il Dictionary Learning mira a trovare un dizionario di elementi, una "base" o "atomi", che possano essere combinati in modo sparso per approssimare efficacemente i dati originali. Grazie alle proprietà del DL, vedremo come l'algoritmo di compressione presentato in questa tesi sia in grado di comprimere dati senza avere un impatto significativo sull'accuratezza del modello di intelligenza artificiale e richiedendo però minime risorse computazionali, rendendolo adatto ad essere implementato anche su dispositivi non complessi come i sensori per IoT.

Introduction

In the last years Digital Twin technology is playing a crucial role in several contexts, from smart agriculture to predictive maintenance, from healthcare to weather modeling in terms of energy and water savings, processes optimization and equipment failure prevention. A Digital Twin can be described as an up-to-date virtual replica of a physical object, system or process through a continuous exchange of massive data between IoT (Internet of Things) sensors on real world and digital system and viceversa: these data are commonly used to train AI models which perform automatic data analysis for decision making [1]. As a consequence, an efficient data handling becomes essential to achieve the full potential of this technology. To this end one of the most cost-effective solution can be a data compression tool which has no significant impact on the AI model accuracy. Among the available state-of-the-art tools to reduce data size we explore Dictionary Learning (DL), a robust matrix factorization algorithm that, in contrast to deep neural network based solutions, does not require powerful computer resources. This aspect makes it suitable to run on light devices such as IoT sensors. In this thesis we applied DL algorithms to compress different types of data and we compare the performance of standard deep learning algorithms trained on both original and compressed data. Dictionary Learning techniques seek to discover an overcomplete set of basis functions (atoms) able to represent in a sparse manner a given set of data samples. In detail, given a matrix of training signals $Y \in \mathbb{R}^{m \times N} (m \ll N)$, DL aims to find a dictionary $D \in \mathbb{R}^{m \times n} (m \ll n)$ and a sparse matrix $X \in \mathbb{R}^{n \times N}$ to represent $Y \approx DX$. It can be also formulated as a two variable, non-convex, constrained optimization problem of the form

$$\begin{aligned} \min_{D,X} \|Y - DX\|_F^2 \quad s.t. \quad \|\mathbf{x}_l\|_0 \leq s, \quad l = 1, ..., N \\ \|\mathbf{d}_j\|_2 = 1, \quad j = 1, ..., n \end{aligned} \tag{1}$$

where the sparsity level s is fixed and $\|.\|_F^2$, $\|.\|_2$ and $\|.\|_0$ represent respectively the Frobenius, L^2 and L^0 norm.

This thesis is structured as follows. The first two chapters contain an overview of the existing literature. Specifically, the first chapter introduces the Sparse Representation and Dictionary Learning problems both from a mathematical and algorithmic point of view, providing different methods and different formulations of the same problem. Furthermore, the chapter also mentions DL primary application domains such as

image denoising, inpainting, classification, and our focus, compression. The second chapter contains an overview of the main structures of the Convolutional Neural Networks (CNNs) architectures. This chapter plays a crucial role in presenting the techniques through which we will evaluate the effectiveness of our compression methods. Indeed, when evaluating the quality of DL compression, it becomes more meaningful to compare the information resulting from the training via CNNs of both the original and compressed datasets, rather than focusing exclusively on the error $\|Y - DX\|_F^2$. The final chapter presents the numerical results and contains our contribution to the problem. In particular, the purpose of the final chapter is to emphasize how Dictionary Learning techniques provide a valuable tool for data compression within the context of Digital Twins. The chapter first provides a brief introduction to Digital Twins, and then explores the application of our compression algorithms in this context, analyzing their performances on various frameworks and types of datasets. In particular, we focused on two different types of data: images and timeseries. Our experiments show that the compression algorithm developed in this thesis is able to compress a redundant dataset by up to 70/80%, maintaining nearly identical classification performance to the original dataset and demanding minimal computational resources.

This work is the result of the internship founded by the european project EUROCC Italy carried out at the HPC department of Cineca [2], a non-profit consortium composed by 116 Italian universities and public institutions. Cineca is the most powerful supercomputing centre for scientific research in Italy, providing support to the country's scientific community, public administrations and companies in different areas such as high performance computing, artificial intelligence and quantum computing. Among the many challenges faced at Cineca, undoubtedly one of them is dealing with an increasingly larger amount of data generated by a wide variety of applications. Since 2022, Cineca hosts Leonardo, the fourth most powerful supercomputer in the world, according to the November 2022 TOP500 list [3].

All the codes necessary to reproduce the experiments shown in this thesis are available at the following link: `https://github.com/lauracavalli/DL4DT.git`.

Chapter 1

The Sparse Representation problem and the Matrix Dictionary Learning problem

In this chapter we will discuss the Sparse Representation problem, the Dictionary Learning problem and their main applications. We will show that Dictionary Learning is an optimization problem which aims to find a matrix $D \in \mathbb{R}^{m \times n}$ called *Dictionary* and a sparse vector $\mathbf{x} \in \mathbb{R}^n$ in order to obtain a good sparse representation $\mathbf{y} \approx D\mathbf{x}$ for a class of signals $\mathbf{y} \in \mathbb{R}^m$. This is achieved by minimizing the error between the signal $\mathbf{y}$ and its sparse representation $D\mathbf{x}$ in the least squares sense. It follows that, thanks to the sparsity of $\mathbf{x}$, each signal is a linear combination of a few *atoms*, as the dictionary columns are usually named. Sparse representation has a great appeal in signal processing and related applications due to its ability to capture only the essentials of a signal. Its main fields of application are data compression, denoising, inpainting and classification.

We now discuss the main aspects of the problem starting from the Sparse Representation problem.

1.1 Introduction to the Sparse Representation problem

Let us start by defining three key-concepts: Dictionary, Sparse Representation and Sparsity level.

Definition 1 (Dictionary)**.**

A Dictionary $D = [\mathbf{d}_1, ..., \mathbf{d}_n] \in \mathbb{R}^{m \times n}$ is a collection of n signals $\mathbf{d}_1, ..., \mathbf{d}_n \in \mathbb{R}^m$ called *atoms*, such that D is:

- overcomplete (i.e. m $\ll$ n)
- normalized ($\|\mathbf{d}_i\| = 1 \quad \forall\, i = 1, ..., n$)

Definition 2 (Sparse Representation)**.**

We say that a vector $\mathbf{y}$ has a *sparse representation* in the basis $D = [\mathbf{d}_1, ..., \mathbf{d}_n]$ if it can be written as a linear combination of a few atoms, i.e.

$$\mathbf{y} = \sum_{j=1}^{n} x_j \mathbf{d}_j = \sum_{j \in \mathcal{S}} x_j \mathbf{d}_j$$

where $\mathcal{S} := \{\ j \mid x_j \neq 0\ \}$ is the support of the signal.

Definition 3 (Sparsity level)**.**

The *Sparsity level* s is defined as the number of atoms involved in the representation of a single signal, i.e.

$$s := |\mathcal{S}| = \|\mathbf{x}\|_0\ ,$$

where $\|\mathbf{x}\|_0$ denotes the 0-norm of $\mathbf{x}$ and represents the number of non-zero elements of the vector $\mathbf{x}$. The notion of sparsity is not exactly defined, but it assumes that $\|\mathbf{x}\|_0 \ll n$ and, more often than not, $\|\mathbf{x}\|_0 \ll m$.

The goal of the *Exact* Sparse Representation problem is to find the sparsest representation of a signal $\mathbf{y}$ given a dictionary D, which consists in finding the sparsest vector $\mathbf{x}$ which is the solution of the linear system $\mathbf{y} = \mathrm{D}\mathbf{x}$, as reported below.

$$\min_{\mathbf{x}} \|\mathbf{x}\|_0 \quad s.t. \quad \mathbf{y} = D\mathbf{x}\ . \tag{1.1}$$

In many cases, however, the signal $\mathbf{y}$ for which one wants to calculate the sparse representation is affected by noise. For this reason is more common to consider the *Approximate* Sparse Representation, whose definition is presented below, instead of the exact one.

Definition 4 (Approximate Sparse Representation)**.**
Given a dictionary $D \in \mathbb{R}^{m \times n}$ and a noise vector $\mathbf{v} \in \mathbb{R}^m$ such that $\|\mathbf{v}\|_2^2 = \delta^2$, where $\delta > 0$ is a constant indicating the noise level of $\mathbf{v}$, it is said that a signal $\mathbf{y} \in \mathbb{R}^m$ admits an approximate sparse representation if there exists a sparse vector $\mathbf{x} \in \mathbb{R}^n$ such that

$$\mathbf{y} = D\mathbf{x} + \mathbf{v}\ . \tag{1.2}$$

The Approximate Sparse Representation problem of a signal can be therefore formulated in two different ways, i.e. as

$$\min_{\mathbf{x}} \|\mathbf{x}\|_0 \quad s.t. \quad \|\mathbf{y} - D\mathbf{x}\|_2 \leq \delta \tag{1.3}$$

or equivalently

$$\min_{\mathbf{x}} \|\mathbf{y} - D\mathbf{x}\|_2 \quad s.t. \quad \|\mathbf{x}\|_0 \leq s \tag{1.4}$$

where δ is a positive constant and s is a fixed positive integer, which constitute the tolerances, respectively, on the norm of the error in problem (1.3) and on the sparsity of the solution in problem (1.4). All the problems above are NP-hard: there are no polynomial time algorithms guaranteed to always solve it. However, there exist many algorithms for solving sparse representation problems (1.3) and (1.4) which are guaranteed to succeed under certain conditions. We will deepen this topic in section 1.4.1.

1.2 Introduction to the Dictionary Learning problem

As previously mentioned, the goal of Dictionary Learning is, given a signal $\mathbf{y} \in \mathbb{R}^m$, to find both a dictionary $D = [\mathbf{d}_1, ..., \mathbf{d}_n] \in \mathbb{R}^{m \times n}$ and a sparse vector of coefficients $\mathbf{x} \in \mathbb{R}^n$ such that

$$\mathbf{y} \approx D\mathbf{x} \, .$$

Actually, to better learn the dictionary D it is more meaningful working with several signals instead of one signal at a time. For this reason, in many applications the following matrix formulation of DL problem is used.

Given a matrix of training signals $Y \in \mathbb{R}^{m \times N} (m \ll N)$, DL aims to find a dictionary $D \in \mathbb{R}^{m \times n} (m \ll n)$ and a sparse matrix $X \in \mathbb{R}^{n \times N}$ to represent $Y \approx DX$.
A visualization of the problem is given in Figure 1.1.

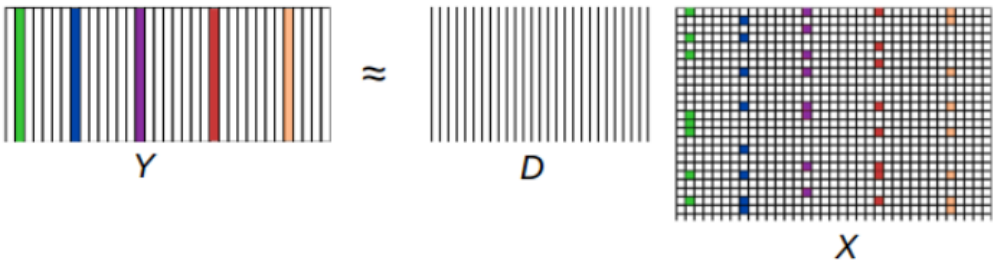

Figure 1.1: Matrix visualization of DL problem. Figure 5.1 in [4]

The DL problem can be formulated as a two variable, non-convex, constrained optimization problem of the form

$$\begin{aligned} \min_{D,X} \|Y - DX\|_F^2 \quad s.t. \quad & \|\mathbf{x}_l\|_0 \leq s, \quad l = 1, ..., N \\ & \|\mathbf{d}_j\|_2 = 1, \quad j = 1, ..., n \end{aligned} \tag{1.5}$$

where the sparsity level s is fixed and $\|Y - DX\|_F^2$ represents the squared Frobenius norm of the error matrix

$$\|Y - DX\|_F =: \|E\|_F := \sqrt{\sum_{i=1}^{m} \sum_{l=1}^{N} e_{il}^2} \, .$$

The first constraint in (1.5) imposes that each signal $\mathbf{y}_l$ has a sparse representation with at most s non-zero coefficients, while the second one ensures that the columns of the dictionary are normalized. This last property removes the indetermination due to a possible multiplicative factor expressed through a diagonal matrix, that can multiply D and divide X without changing the objective of (1.5).

Remark 1.2.1 (Properties of DL problem)**.**

The DL problem (1.5) is **non-convex** due to the sparsity level constraints which involves the l_0-norm which actually is a semi-norm[1]. Furthermore, even though several regularized formulation of the DL problem employ convex norms for the sparse representation, the DL problem itself is still non-convex due to the bi-linearity between D and X. Non-convexity of the DL problem is a central issue since it implies the possible presence of several local minima and the computation of the global minimum is not guaranteed. Furthermore, the problem is even **NP-hard**. Finally, the DL problem formulated as in (1.5) exhibits several symmetries, hence admits multiple global optima. In fact, for a solution (D, X), the pair $(DP, P^{-1}X)$ is also a solution, where P is a permutation matrix [2] whose nonzero elements are ± 1.

Despite the non-convexity of the problem and avoiding the permutation issues, there exist actually some criteria under which a unique solution of (1.5) is guaranteed. These criteria are shown in detail in [5], section 4, Theorem 3, but shortly we can state that if there are enough signals and their representations are sparse, the solution to $Y = DX$ is unique.

1.3 Alternative formulations of the DL problem

The DL problem can be formulated not only as previously seen in (1.5), but in other several ways, each one promoting a different aspect of the problem. Given an application, we should therefore choose the most appropriate formulation according to the properties of the problem that we want to emphasize. Formulations (1.6) and (1.7) below include the minimization of the norm of X in the objective function, looking for the *sparsest* representation matrix that best represents the signals in Y, according to the positive

[1] A semi-norm satisfies all the properties of a norm except the homogeneity one, i.e. it is not always true that $\|\alpha \mathbf{u}\|_0 = \alpha \|\mathbf{u}\|_0$ with $\alpha \in \mathbb{R}^+$.

[2] A permutation matrix is a square binary matrix that has exactly one entry equal to 1 in each row and each column and 0s elsewhere.

parameter λ. Furthermore, (1.7) replaces the 0-norm of X which promotes sparsity with its convex relaxation represented by the 1-norm. This relaxation makes indeed the problem easier to solve.

$$\min_{D,X} \|Y - DX\|_F^2 + \lambda \|X\|_0 \quad s.t. \quad \|\mathbf{d}_j\|_2 \leq 1, \quad j = 1, ..., n \tag{1.6}$$

$$\min_{D,X} \|Y - DX\|_F^2 + \lambda \|X\|_1 \quad s.t. \quad \|\mathbf{d}_j\|_2 \leq 1, \quad j = 1, ..., n \tag{1.7}$$

We stress that even though the objective function is only slightly different, these formulations are not equivalent and the solutions indeed change.

Another formulation of the DL problem can be considered to enhance the sparsity of the representation matrix X, imposing a constraint on the representation error as shown below.

$$\min_{D,X} \|X\|_0 \quad s.t. \quad \|Y - DX\|_F^2 \leq \epsilon \ . \tag{1.8}$$

1.4 The algorithms

In this section we will focus on the minimization of the representation error as shown in formulation (1.5) through an alternate optimization approach, which has been found to be the most successful one. The alternate optimization approach consists in updating a subset of the variables while keeping the others fixed, taking care to alternate the subsets such that all variables are optimized. In the case of DL problem, this approach splits the problem into two subproblems: **sparse coding** and **dictionary update**. More precisely, given the signal matrix Y and an initial dictionary D, at each iteration first the minimization problem in X is solved while D is fixed (sparse coding) and then the minimization problem in D is solved while keeping X fixed (dictionary update). Algorithm 1 below summarizes the procedure.

Algorithm 1: DL by Alternate Optimization [6]

Given the matrix of signals $Y \in \mathbb{R}^{m \times N}$, the sparsity level s, the initial dictionary $D \in \mathbb{R}^{m \times n}$ and the number of iterations K ;

for *k=1,...,K* **do**

- Sparse coding : keeping D fixed, compute sparse representations X ;
- Dictionary update: keeping the nonzero pattern Ω fixed, compute new dictionary D (X may be changed or not);
- Atoms normalization;

end

We stress that an initial dictionary is required at the beginning of Algorithm 1. The typical initialization methods are either to select randomly n signals or to simply generate the atoms randomly. In both cases we must take care to normalize the atoms.

Alternatively, before the formalization of the DL problem, it was common to solve these types of problems as sparse representation problems by using fixed dictionaries such as the Haar matrix [7] or the Discrete Cosine Transform (DCT) matrix [8] which are typically used in signal compression techniques. For this reason, an alternative initialization of the dictionary is given by these two matrices. Since the DL problem has multiple local minima and the algorithms are based on local optimization methods, it is expected that different initializations may lead to different results.

1.4.1 Sparse Coding

In the sparse coding step, the dictionary D is fixed and the sparse representations X has to be computed, leading to the following minimization problem

$$\min_X \|Y - DX\|_F^2 \quad s.t. \quad \|\mathbf{x}_l\|_0 \leq s, \quad l = 1, ..., N \,. \tag{1.9}$$

Remark 1.4.1. We notice that

$$\|E\|_F^2 = \sum_{l=1}^{N} \|\mathbf{e}_l\|_2^2 = \sum_{l=1}^{N} \|\mathbf{y}_l - D\mathbf{x}_l\|_2^2 \,.$$

Hence, solving the sparse coding problem (1.9) is equivalent to solve N approximate sparse representation problems, one for each signal

$$\min_{x_l} \|\mathbf{y}_l - D\mathbf{x}_l\|_F^2 \quad s.t. \quad \|\mathbf{x}_l\|_0 \leq s, \qquad \forall\, l = 1, ..., N \,. \tag{1.10}$$

As already mentioned at the end of section 1.1, in literature there exist many algorithms for solving these sparse representation problems. They actually cannot find the optimal solution in general, but some of them are guaranteed to succeed under certain conditions and in general have a good practical behavior (see the conditions in section 1.5 of [6]). Among them we can detect two main classes of algorithms: *greedy* algorithms which build the support by selecting one atom at a time based on a local objective, and *convex-relaxation-based* algorithms which consists of replacing the 0-pseudo-norm with an actual norm, obtaining a convex optimization problem. We will focus on greedy algorithms since it has been noticed that they are usually very fast. In particular, we will always use the Orthogonal Matching Pursuit (OMP) Algorithm for this task, because of its good trade-off between quality and complexity.

1.4.1.1 Orthogonal Matching Pursuit (OMP) Algorithm

The Orthogonal Matching Pursuit ([9]) is an iterative greedy algorithm that selects at each step the atom best correlated with the residual part of the signal. Then it produces a new approximation by projecting the signal onto the dictionary elements that have already been selected.

As shown in Algorithm 2, it starts from an empty support $\mathcal{S} = \emptyset$ and error $\mathbf{e} = \mathbf{y}$ and, until the sparsity level s is not reached and/or the residual is greater than a given error bound ϵ, OMP firstly chooses the atom which is the most correlated with the current residual in order to minimize the next one, i.e.

$$\mathbf{d}_k \quad \text{where} \quad k = \text{argmax}_{j \notin \mathcal{S}} \left| \mathbf{e}^T \mathbf{d}_j \right| .$$

Note that OMP never selects the same atom twice because, as we will see, the residual is orthogonal to the atoms that have already been chosen. Then, after the update of $\mathcal{S}$ atoms ($\mathcal{S} = \mathcal{S} \cup \{k\}$), OMP computes the new coefficients by solving the following least square problem

$$\mathbf{x}_{\mathcal{S}} = \underset{\xi}{\text{argmin}} \left\| \mathbf{y} - D_{\mathcal{S}} \boldsymbol{\xi} \right\|_2 , \tag{1.11}$$

where $D_{\mathcal{S}}$ is the restriction of the matrix D to the columns with indices from $\mathcal{S}$. The least squares problem (1.11) can be solved in different ways. The usual solution is given by

$$\mathbf{x}_{\mathcal{S}} = (D_{\mathcal{S}}^T D_{\mathcal{S}})^{-1} D_{\mathcal{S}}^T \mathbf{y} , \tag{1.12}$$

but later on we will discuss two more efficient approaches that have been proposed in [10]. Finally it updates the residual $\mathbf{e} = \mathbf{y} - \sum_{j \in \mathcal{S}} \mathbf{x}_j \mathbf{d}_j$ and checks the stopping criteria.

Algorithm 2: OMP with Naive approach [6]

Given the signal $\mathbf{y} \in \mathbb{R}^m$, the sparsity level s, the initial dictionary $D \in \mathbb{R}^{m \times n}$ and the stopping tolerance ϵ ;
Initialize $\mathcal{S} = \emptyset$, $\mathbf{e} = \mathbf{y}$;
while $|\mathcal{S}| < s$ *and* $\|\boldsymbol{e}\| > \epsilon$ **do**
  $k = arg\max_{j \notin \mathcal{S}} |\mathbf{e}^T \mathbf{d}_j|$;
  $\mathcal{S} = \mathcal{S} \cup \{k\}$;
  $\mathbf{x}_{\mathcal{S}} = (D_{\mathcal{S}}^T D_{\mathcal{S}})^{-1} D_{\mathcal{S}}^T \mathbf{y}$;
  $\mathbf{e} = \mathbf{y} - D_{\mathcal{S}} \mathbf{x}_{\mathcal{S}}$
end

As already mentioned, computing from scratch the least squares solution as in (1.12) at each iteration is a naive approach. Since the current matrix $D_{\mathcal{S}}$ is updated by simply appending one column, we could get a more efficient implementation by exploiting the least square solution just computed at the previous iteration. The most famous approaches make use of the Cholesky decomposition of $D_{\mathcal{S}}^T D_{\mathcal{S}}$ ([11], section 2.2) or the QR decomposition of $D_{\mathcal{S}}$ ([11], section 2.3). Both approaches will be presented in the following sections.

1.4.1.2 OMP-QR

We first focus on how the QR decomposition of $D_{\mathcal{S}}$ can improve the naive formulation of OMP. We use the following notation

$$D_{\mathcal{S}} = QR \ ,$$

where Q is an orthogonal matrix (i.e. $Q^TQ = I$) and R is an upper triangular matrix. It is not restrictive to consider R as an invertible matrix, since if $D_{\mathcal{S}}$ is not a full-rank matrix, it is always possible to use the economic-QR.

Given a solution $\mathbf{x}_{\mathcal{S}}$ of (1.11), then $\mathbf{x}_{\mathcal{S}}$ also satisfies the normal equation

$$D_{\mathcal{S}}^T D_{\mathcal{S}} \mathbf{x}_{\mathcal{S}} = D_{\mathcal{S}}^T \mathbf{y} \ . \tag{1.13}$$

Considering the QR decomposition of $D_{\mathcal{S}}$ we get

$$D_{\mathcal{S}}^T D_{\mathcal{S}} \mathbf{x}_{\mathcal{S}} = D_{\mathcal{S}}^T \mathbf{y} \iff R^T \underbrace{Q^T Q}_{I} R\mathbf{x}_{\mathcal{S}} = D_{\mathcal{S}}^T \mathbf{y} \iff R^T R \mathbf{x}_{\mathcal{S}} = D_{\mathcal{S}}^T \mathbf{y} \ . \tag{1.14}$$

Denoting with $\mathbf{b} := D_{\mathcal{S}}^T \mathbf{y}$, OMP-QR finds $\mathbf{x}$ by solving the following two triangular linear systems:

$$\begin{aligned} R^T \mathbf{z} &= \mathbf{b} \ , \\ R\mathbf{x}_{\mathcal{S}} &= \mathbf{z} \ . \end{aligned} \tag{1.15}$$

Algorithm 3 shows how to simply update at each iteration the matrices R and Q in order to compute the vector $\mathbf{x}$ only at the end of the algorithm, when the support $\mathcal{S}$ is determined. Consider the i-th iteration of the OMP algorithm and denote with Q_i and R_i the matrices Q and R at the current iteration. We first find the atom $\mathbf{d}_k$ of the original dictionary which is the most correlated with the current residual, and we add its index to the index set $\mathcal{S}_i$. Then, the direction of the new atom $\mathbf{d}_k$ can be computed by subtracting from it its least square projection onto the span of the atoms indexed by $\mathcal{S}_{i-1}$, i.e. $\mathbf{d}_k - Q_{i-1}(Q_{i-1}^T \mathbf{d}_k)$. Then OMP updates Q_{i-1} by adding this unit-norm direction

$$Q_i := \begin{bmatrix} Q_{i-1} & \frac{\mathbf{d}_k - Q_{i-1}(Q_{i-1}^T \mathbf{d}_k)}{\sqrt{\|\mathbf{d}_k\|^2 - \|Q_{i-1}^T \mathbf{d}_k\|^2}} \end{bmatrix} \ . \tag{1.16}$$

Since $D_{\mathcal{S}_i} = Q_i R_i = \begin{bmatrix} D_{i-1} & \mathbf{d}_k \end{bmatrix}$, we see that OMP can update R by

$$R_i := \begin{bmatrix} R_{i-1} & Q_{i-1}^T \mathbf{d}_k \\ \mathbf{0}^T & \sqrt{\|\mathbf{d}_k\|^2 - \|Q_{i-1}^T \mathbf{d}_k\|^2} \end{bmatrix} \ . \tag{1.17}$$

Finally, OMP computes the residual recursively

$$\mathbf{e}_i = \mathbf{e}_{i-1} - \mathbf{q}_i \mathbf{q}_i^T \mathbf{y}$$

where $\mathbf{q}_i$ is the last column of Q_i.

The whole procedure is summarized in Algorithm 3.

Algorithm 3: OMP with QR decomposition [11]

Given the signal $\mathbf{y} \in \mathbb{R}^m$, the sparsity level s, the initial dictionary $D \in \mathbb{R}^{m \times n}$ and the stopping tolerance ϵ ;

Initialize $\mathcal{S}_0 = \emptyset$, $\mathbf{e}^0 = \mathbf{y}$ and $\mathbf{b} = []$;

for $i = 1, ..., s$ **do**

$k = arg\max_{j \notin \mathcal{S}_{i-1}} \frac{|\mathbf{d}_j^T \mathbf{e}^{i-1}|}{\|\mathbf{d}_j\|_2}$;

Update $\mathcal{S}_i = \mathcal{S}_{i-1} \cup \{k\}$;

Update $\mathbf{b} = \begin{bmatrix} \mathbf{b} \\ \mathbf{d}_k^T \mathbf{y} \end{bmatrix}$;

Compute recursively the QR dec. of $D_{\mathcal{S}_i} = Q_i R_i$ by using (1.16) and (1.17);

$\mathbf{e}_i = \mathbf{e}_{i-1} - \mathbf{q}_i \mathbf{q}_i^T \mathbf{y}$;

if $\|\boldsymbol{e}^i\|_2 < \epsilon$ **then**

stop

end

end

Compute $\mathbf{x}_{\mathcal{S}}$ by solving $R^T R \mathbf{x}_{\mathcal{S}} = \mathbf{b}$ as in (1.15).

1.4.1.3 OMP-Cholesky

An alternative approach uses the Cholesky decomposition of $D_{\mathcal{S}}^T D_{\mathcal{S}}$. The matrix $D_{\mathcal{S}}^T D_{\mathcal{S}}$ is indeed symmetric and positive-definite and it is updated every iteration by simply appending a single row and column to it.

Consider the i-th iteration. We denote with A_i the matrix $D_{\mathcal{S}_i}^T D_{\mathcal{S}_i}$ and with $\mathbf{d}_k$ the atom added at the current iteration. It is easy to check that, given the Cholesky factorization of $A_{i-1} = L_{i-1} L_{i-1}^T \in \mathbb{R}^{(n-1)\times(n-1)}$, the Cholesky factorization of

$$A_i = \begin{bmatrix} A_{i-1} & \mathbf{v} \\ \mathbf{v}^T & c \end{bmatrix} \in \mathbb{R}^{n \times n} \quad \text{where} \quad \mathbf{v} = D_{\mathcal{S}_{i-1}}^T \mathbf{d}_k \quad \text{and} \quad c = \|\mathbf{d}_k\|_2^2 = 1$$

is given by $A_i = L_i L_i^T$ with

$$L_i = \begin{bmatrix} L_{i-1} & \mathbf{0} \\ \mathbf{w}^T & \sqrt{c - \mathbf{w}^T \mathbf{w}} \end{bmatrix} \quad \text{where} \quad \mathbf{w} \quad \text{solves} \quad L_{i-1} \mathbf{w} = \mathbf{v} \quad .$$

Finally we have to compute the solution by solving the system $L_i L_i^T \mathbf{x}_{\mathcal{S}} = D_{\mathcal{S}}^T \mathbf{y}$. Notice that the vector $\mathbf{b} := D_{\mathcal{S}}^T \mathbf{y} \in \mathbb{R}^{|S|}$ is simply updated at each iteration by adding the scalar product $\mathbf{d}_k^T \mathbf{y}$ as its last component. This process is repeated until the sparsity constraint is reached or the norm of the residual is under a certain threshold ϵ and it is reported in detail in Algorithm 4.

Algorithm 4: OMP with Cholesky decomposition, [11] and [6]

Given the signal $\mathbf{y} \in \mathbb{R}^m$, the sparsity level s, the initial dictionary $D \in \mathbb{R}^{m \times n}$ and the stopping tolerance ϵ ;

Initialize $\mathcal{S} = \emptyset$, $\mathbf{e} = \mathbf{y}$, $L = 1$, $\mathbf{b} = []$;

while $|\mathcal{S}| < s$ *and* $\|\boldsymbol{e}\| > \epsilon$ **do**

  $k = arg\max_{j \notin \mathcal{S}} |\mathbf{e}^T \mathbf{d}_j|$;

  Update $\mathbf{b} = \begin{bmatrix} \mathbf{b} \\ \mathbf{d}_k^T \mathbf{y} \end{bmatrix}$;

  if $|\mathcal{S}| > 1$ **then**

    Compute $\mathbf{w}$ by solving the triangular system $L\mathbf{w} = D_{\mathcal{S}}^T \mathbf{d}_k$;

    Update $L = \begin{bmatrix} L & \mathbf{0} \\ \mathbf{w}^T & \sqrt{1 - \mathbf{w}^T \mathbf{w}} \end{bmatrix}$;

  end

  Update $\mathcal{S} = \mathcal{S} \cup \{k\}$;

  Compute new solution by solving : $LL^T \mathbf{x}_S = \mathbf{b}$;

  $\mathbf{e} = \mathbf{y} - D_{\mathcal{S}} \mathbf{x}_{\mathcal{S}}$;

end

1.4.2 Dictionary Update

The second subproblem of the DL algorithm alternating approach is the dictionary update that consists in solving the following minimization problem

$$\min_{D,(X)} \|Y - DX\|_F^2 \quad s.t. \quad X_{\Omega^c} = 0 \qquad \|\mathbf{d}_j\|_2 = 1, \quad j = 1, ..., n \tag{1.18}$$

where the sparsity pattern $\Omega := \{ (i, l) \mid x_{il} \neq 0 \}$ is fixed.

Remark 1.4.2.

$X_{\Omega^c} = 0$ is equivalent to the previous sparsity constraint $\|\mathbf{x}_l\|_0 \leq s, \forall\, l = 1, ..., N$.

Several algorithms have been proposed for the dictionary update task. Some of them consider formulation (1.18) as a minimization problem only in the variable D, while others involve also the sparse matrix X. We will report an example of each type. A complete review of dictionary update algorithms can be found in [6].

1.4.2.1 Method of Optimal Directions (MOD)

In this approach the dictionary update step is seen as a quadratic optimization problem in the variable D. Due to the convexity of the error $\|Y - DX\|_F^2$ with respect to D, its minimization amounts to setting to zero its gradient

$$\nabla(\|Y - DX\|_F^2) = 2(DX - Y)X^T \ . \tag{1.19}$$

Therefore, imposing (1.19) equal to zero, we get that dictionary update is carried out by computing

$$D = YX^T(XX^T)^{-1} = YX^{\dagger} \ , \tag{1.20}$$

where $X^{\dagger} := X^T(XX^T)^{-1}$ represents the Moore-Penrose-Woodboury pseudo-inverse of X. The main complexity of this dictionary update lies in the computation of the pseudo-inverse $X^{\dagger}$. This issue can be overcome by using the Cholesky decomposition of the matrix XX^T as shown in [6], section 3.4.

1.4.2.2 K-SVD

The K-SVD method [12] presents a whole new approach to the dictionary update step, which is done jointly with an update of the sparse representation coefficients related to it, resulting in accelerated convergence.

K-SVD algorithm was purposely designed to be a direct generalization of the K-means algorithm. This is possible due to a strong relation between the sparse representation and clustering: in the latter each sample is represented by *one* of the descriptive vectors $\{\mathbf{d}_k\}_{k=1}^K$, while in the former each sample is represented as a linear combination of *several* vectors $\{\mathbf{d}_k\}_{k=1}^K$.
Specifically, K-Means algorithm is developed in two steps per each iteration:

- given $\{\mathbf{d}_k\}_{k=1}^K$, assign the training samples to their nearest neighbor;
- given that assignment, update $\{\mathbf{d}_k\}_{k=1}^K$ to better fit the samples;

K-SVD algorithm follows the same approach, since the dictionary update step updates one column at a time, fixing all columns in D except one ($\mathbf{d}_k$) and finding a new column $\mathbf{d}_k$ and new values for its coefficients that best reduce the Mean Square Error (MSE). The novelty of this algorithm and its similarity with K-Means, then, lies in two main properties: firstly, the columns of D are computed sequentially and secondly, it allows changing the relevant coefficients. This last property accelerates the descent of the error since the subsequent column updates will be based on more relevant coefficients.

Using the same approach of Coordinate Descent methods ([6], section 3.3.2), we can rewrite our objective function as

$$\|Y - DX\|_F^2 = \|Y - \underbrace{\sum_{i=1}^{n} \mathbf{d}_i\mathbf{x}_i^T}_{=DX}\|_F^2 = \|\underbrace{Y - \sum_{i\neq j} \mathbf{d}_i\mathbf{x}_i^T}_{=:F_j} - \mathbf{d}_j\mathbf{x}_j^T\|_F^2 = \|F_j - \mathbf{d}_j\mathbf{x}_j^T\|_F^2 \ ,$$

where $\mathbf{x}_i^T$ represent the i-th row of X. Therefore, $\forall\ j = 1, ..., n$ problem (1.18) can be equivalently formulated as

$$\begin{aligned} \min_{\mathbf{d}_j, \mathbf{x}_j} \|F_j - \mathbf{d}_j\mathbf{x}_j^T\|_F^2 \quad s.t. \quad & X_{\Omega^c} = 0 \\ & \|\mathbf{d}_j\|_2 = 1, \quad j = 1, ..., n \end{aligned} \tag{1.21}$$

Problem (1.21) can be seen as an attempt to approximate the matrix F_j with a rank-1 matrix given by $\mathbf{d}_j \mathbf{x}_j^T$. From this interpretation follow the idea to use the Singular Value Decomposition (SVD) of F_j

$$F_j = \sum_{i=1}^{r} \sigma_i^j \mathbf{u}_i^j \mathbf{v}_i^{jT}$$

Easily follows that the solution of (1.21) is given by

$$\begin{aligned} \mathbf{d}_j &= \mathbf{u}_1^j \\ \mathbf{x}_j &= \sigma_1^j \mathbf{v}_1^j \end{aligned}$$

where $\sigma_1^j, \mathbf{u}_1^j, \mathbf{v}_1^j$ are respectively the singular value, the left singular vector and the right singular vector of F_j with largest absolute value.

Finally, we must check whether this solution satisfies the constraints of the dictionary update step (1.18). On one side, the singular vectors have norm equal to 1, hence the normalization constraint is satisfied. However, on the other side, SVD usually does not return sparse singular vectors, hence this routine could violate the sparsity constraint. This issue can be easily overcome by projecting our problem onto the subspace defined by the sparsity pattern Ω previously computed in the sparse coding step. For this reason, (1.21) can be equivalently formulated as the following minimization problem

$$\min_{\mathbf{d}_j} \|P_\Omega(F_j - \mathbf{d}_j \mathbf{x}_j^T)\|_F^2 = \min_{\mathbf{d}_j} \| \underbrace{(F_j)_{\mathcal{I}_j} - \mathbf{d}_j X_{j,I_j}}_{:=E_{\mathcal{I}_j}} \|_F^2 , \tag{1.22}$$

where $P_\Omega \in \mathbb{R}^{s \times m}$ is the orthogonal projector on Ω and $\mathcal{I}_j := \{l : |(j,l) \in \Omega\}$ represents the indexes of the signals that are represented by the atom $\mathbf{d}_j$. Notice also that $(F_j)_{\mathcal{I}_j}$ and $\mathbf{d}_j X_{j,I_j}$ are matrices in $\mathbb{R}^{s \times N}$ where $s \leq m$.

Remark 1.4.3. (on orthogonal projections)
An orthogonal projection is a linear operator from a space Ω to itself, such that it can be written as $P = QQ^T$ where Q has orthonormal columns. Follows that such P is symmetric and $P = P^2$. Furthermore we can prove that the only eigenvalues of an orthogonal projection are 0 and 1, hence follows that $\|P\|_F = 1$.

The K-SVD algorithm is reported in detail in Algorithm 5. Starting from the matrix of signals $Y \in \mathbb{R}^{m \times N}$, the initial dictionary $D \in \mathbb{R}^{m \times n}$ and the representation matrix $X \in \mathbb{R}^{n \times N}$, it first defines the error $E = Y - DX$. Then, for each column of D, it extrapolates $(F_j)_{\mathcal{I}_j}$ (referred as F in the algorithm) from the error relative to $\mathcal{I}_j$ and then, by considering the SVD of F, it computes $\mathbf{d}_j$ and $\mathbf{x}_j$. It concludes each iteration by updating the error with the values just found.

Algorithm 5: K-SVD Dictionary Update [6]

Given the matrix of signals $Y \in \mathbb{R}^{m \times N}$, the initial dictionary $D \in \mathbb{R}^{m \times n}$ and the representation matrix $X \in \mathbb{R}^{n \times N}$;

$E = Y - DX$;

for *j=1,...,n* **do**

  $F = E_{I_j} + \mathbf{d}_j X_{j,I_j}$;

  compute $\sigma_1, \mathbf{u}_1, \mathbf{v}_1$ of F ;

  $\mathbf{d}_j = \mathbf{u}_1$ and $X_{j,I_j} = \sigma_1 \mathbf{v}_1^T$;

  $E_{I_j} = F - \mathbf{d}_j X_{j,I_j}$;

end

A drawback of the K-SVD method is the high computational cost of the SVD. However, it actually needs only the first singular vectors, which can be computed much easier than the full SVD. Furthermore, there exist many implementations of SVD routine developed in the most used programming languages (for example the Matlab function `svds` which computes the truncated SVD).

To overcome even more the computational complexity of SVD, in [13] has been proposed the Approximated K-SVD (AK-SVD) where the algorithm converges to the first singular values and vectors of $F_j \quad \forall \; j = 1, ..., n$ without explicitly computing them through SVD. The idea behind this algorithm consists in optimize *first* the atom and *then* the associated representations, not simultaneously like in K-SVD. This implies a splitting of the problem into two smaller subproblems, one on the atom first and the next on the representation. For further details on AK-SVD we recall [6], section 3.5.

1.5 DL applications

As already mentioned at the beginning of this chapter, DL problem has a great appeal in various applications due its ability to "capture the essentials of a signal" [6]. Therefore, once we have clarified which DL algorithm we prefer to use and why, we will start this section by providing in Example 1 an elucidation of the meaning behind this phrase. Afterward, we will focus on main problems like denoising, inpainting, classification and compression, as presented in [6]. In particular, compression tasks, which form the central focus of this thesis, will be further examined in Chapter 3.

1.5.1 Choice of the DL algorithm

For the implementation of the Dictionary Learning algorithm presented in Algorithm 1, we started from the Python library `dictlearn` [14] that implements many different version of DL algorithms.

For what concern the sparse coding step, the library allows to choose between OMP-Cholesky algorithm or a custom algorithm. Since the most famous OMP approaches are OMP-Cholesky and OMP-QR we also implemented our own version of the OMP-QR algorithm. This can be easily integrated in the main algorithm of the library by simply setting the `transform_algorithm` parameter of `DictionaryLearning` function as the name of the custom function. We then proceeded to compare their performances by looking at the decrease of the error

$$\frac{\|Y - DX\|_F}{\sqrt{mN}} \tag{1.23}$$

with respect to the iterations and the elapsed time. Since QR and Cholesky factorizations are both direct approaches to solve the same problem (1.12), we expect they return the same solution but with different computational times. We, therefore, run the DL algorithm either with OMP-Cholesky or OMP-QR for the sparse coding step and both times with K-SVD for the dictionary update step on artificial data Y, D, X generated with the Python function `make_sparse_coded_signal` from the `sklearn.datasets` library. In this way we are sure that DX is a sparse representation of Y. For both algorithms was generated the same random initial dictionary of size 20×50 and 1500 data signals of dimension 20 with sparsity level equal to three. The number of iterations was set equal to 100. As we expected, we can see from Figure 1.2 that the error trends exactly coincide. However, while the DL algorithm implemented with OMP-QR for sparse coding step requires 19.11 seconds to conclude the iterations, the DL algorithm implemented with OMP-Cholesky, with its 28.22 seconds, is slightly slower to conclude the iterations.

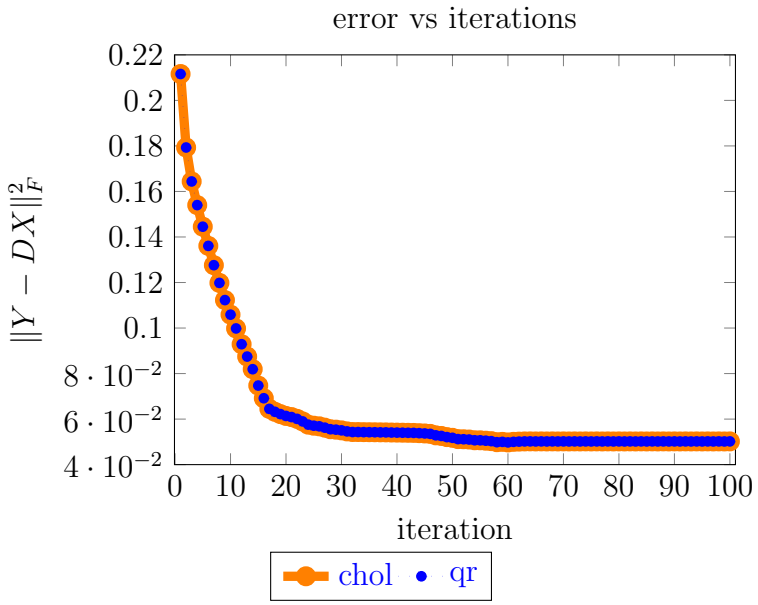

Figure 1.2: DL with OMP-QR and OMP-Cholesky compared.

A different way to test the quality of a DL algorithm is to check if the optimized dictionary coincides with the "true" one D. Of course looking at the error $\|\hat{D}-D\|_F$ is not meaningful since, as we have seen, the DL problem is invariant to atoms permutations. A more relevant analysis can be done by sweeping through the atoms of the true dictionary and finding the closest one in the computed dictionary by measuring the distance between the atoms as

$$1 - |\hat{\mathbf{d}}^T \mathbf{d}| \ ,$$

where $\hat{\mathbf{d}}$ and $\mathbf{d}$ are respectively atoms of the optimized dictionary and the true dictionary. A distance less than 0.01 is considered a success. Taking the ratio between the number of true atoms that can be found in the optimized dictionary $\hat{D}$ and the total number of atoms we obtain the *recovery ratio* of the DL algorithm. Taking the mean value out of 50 trials we get that DL algorithm both with OMP-QR and OMP-Cholesky and the K-SVD for the dictionary update step can recover more than 90 %.

Since the algorithms are equivalent but the use of OMP-Cholesky seems to lead to a more time consuming DL algorithm, from now on we will always use our implementation of OMP-QR for the sparse coding step. Moreover, as already shown in equation (1.10), the OMP algorithm computes the columns of the sparse matrix X separately, splitting the sparse coding problem (1.9) in several subproblems independent from each other. For this reason the custom OMP-QR algorithm can be further accelerated by parallelizing the code, making maximum use of the avaiable resources of an HPC cluster. In particular, for this task we employed the `multiprocessing` Python library [15] (in particular its `Pool` object), a package that enables the creation of processes through an API similar to the threading module. Specifically, whenever OMP-QR was used, we partitioned the signal Y into four distinct blocks and conducted individual OMP-QR processes on each. For further details on Marconi100 architecture and on the functioning of an HPC cluster see Appendix B and [16].

Figure 1.3 reports the same experiment shown in Figure 1.2 while also incorporating the representation of the error's behavior throughout the iterations of the DL algorithm implemented with the parallel version of OMP-QR for the sparse coding step and K-SVD, as always, for the dictionary update step.

We can notice that the error trends still coincide, as expected, but the latest added algorithm takes 8.89 s to conclude the iterations. Therefore, the DL algorithm implemented with our parallel version of OMP-QR has a speed-up of 2.15 with respect to the DL with serial OMP-QR. It cannot scale properly, but the cause might be that the matrix Y is not sufficiently large to need the use of 4 CPUs.

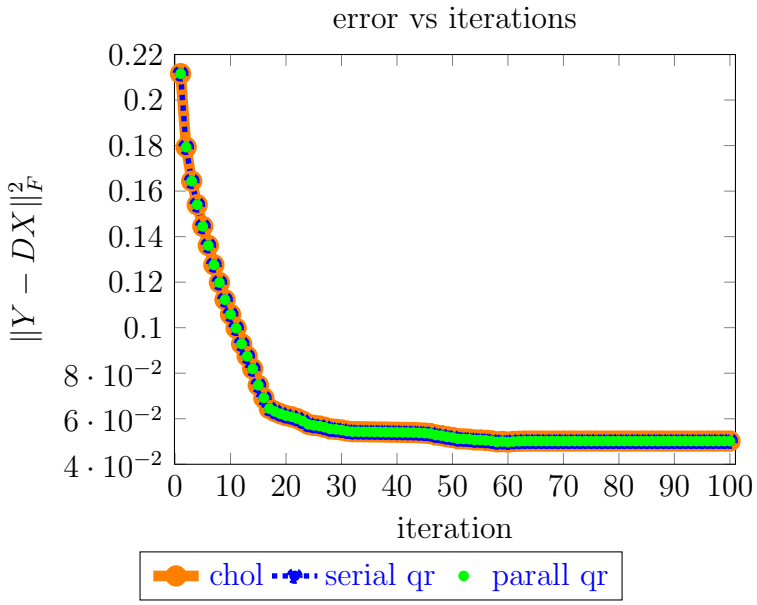

Figure 1.3: DL with serial OMP-QR, parallel OMP-QR and OMP-Cholesky compared.

For what concern the dictionary update step, the library allows to choose among several different algorithms such as K-SVD, AK-SVD, MOD and many others through the `fit_algorithm` parameter of `DictionaryLearning` algorithm. We decided to focus on K-SVD and AK-SVD, comparing the behaviour of the error with respect to the iterations running the DL algorithm always with parallel OMP-QR for the sparse coding step and either K-SVD or AK-SVD for the the dictionary update step. We set the number of iteration as $K = 20$ and we run it on a random dataset with $N = 6000$ samples of $m = 784$ features each and a trained dataset with $n = 800$ atoms and sparsity level $s = 10$.

From the theory we know that, while in K-SVD the decrease is maximal, AK-SVD only progresses towards the minimum (see [6] for further details), therefore we expect this last one less accurate, but slightly faster. In Figure 1.4 we report the error trends.

The error trends are as expected. Moreover, the DL algorithm, when employing AK-SVD during the dictionary update phase, concludes its iterations in 93.49 seconds, whereas with K-SVD, it takes 96.73 seconds. In the following analysis we will prioritize achieving a lower error over reducing computational time. For this reason, from now on we will always use the K-SVD algorithm for the dictionary update step.

Concluding, in all the numerical experiments in this thesis, unless stated otherwise, we will always use the DL algorithm implemented with our parallel version of OMP-QR for the sparse coding step and the K-SVD algorithm (imposing `fit_algorithm = "ksvd"`) for the dictionary update step. Where needed, we will refer to this algorithm as "DL_ompqr_parallel_ksvd".

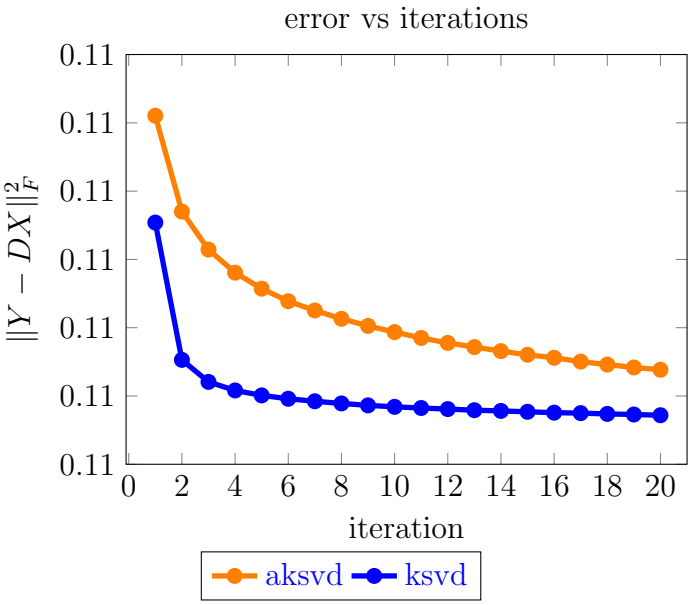

Figure 1.4: DL with K-SVD and AK-SVD compared.

Example 1.

So far, we often wrote that DL techniques are able to "capture the essential of a signal". The goal of this example is to show and clarify what we mean by this phrase.

In previous sections we have shown that, thanks to DL factorization, a signal can be always written as a linear combination of few atoms. For this reason we expect these atoms to be able to capture the most important features of the given signal. Figure 1.5 and Figure 1.6 shows the most representative atoms of the learned dictionary of two images with different geometric patterns. As expected, in the first case (Fig. 1.5) the atoms detect the boundary of the circles and the circles themselves, while in the second one (Fig. 1.6) the main features are stripes of different width and in different positions. Both the original images have dimensions 256×256 pixels and they have been split in 15376 patches of 10×10 pixels (stepsize $= 2$) to train a dictionary of $n = 150$ atoms. The sparsity target was set to $s = 1$ and the DL process was iterated $K = 10$ times.

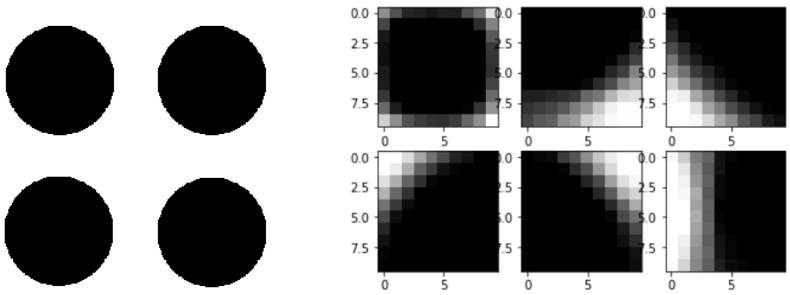

Figure 1.5: On the left: original image with discs. On the Right: some atoms of the dictionary.

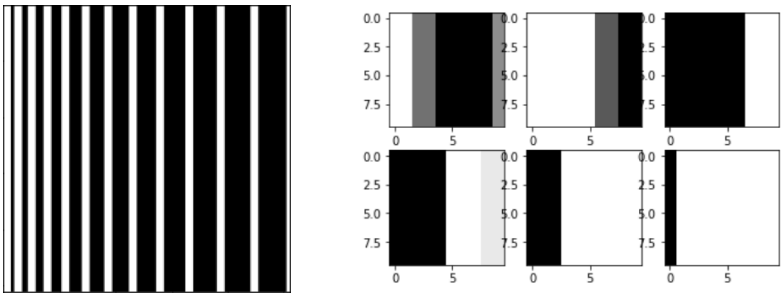

Figure 1.6: On the left: original image with stripes. On the Right: some atoms of the dictionary.

We additionally report in Figure 1.7 how the atoms of the starting dictionary look like, in order to show how they change during the optimization process: from random vectors they gradually get the shape of the most relevant features of the signal.

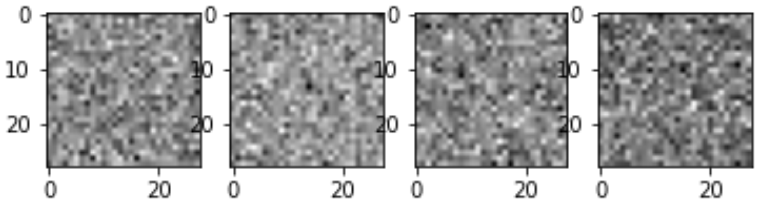

Figure 1.7: Atoms of the random starting dictionary.

1.5.2 Denoising

The goal of Denoising, as the word suggest, is to remove or reduce some noise from a signal (an image in this case), while preserving its essential features and details. This type of problem is usually approached by splitting the noisy image $\mathbf{y}$ in N patches of dimensions $\sqrt{m} \times \sqrt{m}$ which are then vectorized and collected in a matrix of training signals $Y \in \mathbb{R}^{m \times N}$. Assuming at first that an appropriate dictionary D is given, the Denoising process is equivalent to solve the following problem

$$\min_{\mathbf{x}_i} \|\mathbf{x}_i\|_0 \qquad s.t. \quad \|\mathbf{y}_i - D\mathbf{x}_i\|_2 \leq \epsilon \quad i = 1, ..., N \tag{1.24}$$

where ϵ is positive and proportional to the noise variance σ^2. Having the dictionary D fixed, problem (1.24) can be easily solved by OMP algorithms presented in section 1.4.1. Given $\mathbf{x}_i \; \forall i = 1, ...N$ solutions of problem (1.24), the denoised image is then given by

$$Z = [\mathbf{z}_1, ...\mathbf{z}_N] := [D\mathbf{x}_1, ...D\mathbf{x}_N].$$

Introducing also the learning phase of the dictionary, i.e. first solving the DL problem

$$\min_{X,D} \|X\|_0 \qquad s.t. \quad \|\mathbf{y}_i - D\mathbf{x}_i\|_2 \leq \epsilon, \quad i = 1, ..., N$$

and then computing the denoised image by $Z = DX$, we can achieve even better results.

Example 2 shows a full denoising process via DL applied to the image "Lena".

Example 2.
We started by applying an additive random noise of magnitude $\approx 10e^{-2}$ and splitting the image in 15625 patches, 8×8 pixels each (stepsize $= 2$). We then applied both the usual DL algorithm and the version with OMP-Cholesky for the sparse coding step for $K = 30$ times, learning a dictionary of $n = 441$ atoms and setting the sparsity level as $s = 10$. Finally we merged the patches to visualize the reconstructed image. As expected, from Figure 1.8 we can see that the images have almost the same quality, indeed the one which used OMP-QR has PSNR[3] equal to 79.23 while the PSNR of the one denoised with OMP-Cholesky is about 79.05.

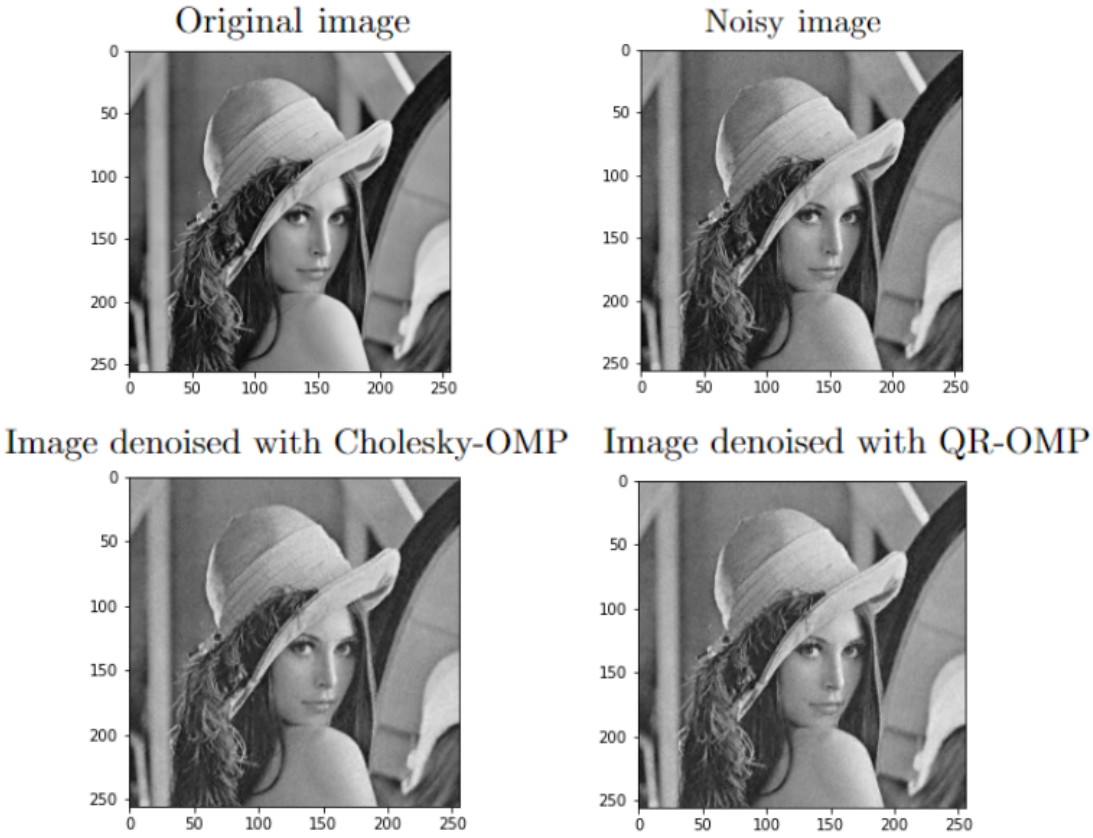

Figure 1.8: Attempts of denoising with DL techniques.

[3]Peak signal-to-noise ratio (PSNR) is the ratio between the maximum possible power of an image and the power of corrupting noise that affects the quality of its representation

1.5.3 Inpainting

The Inpainting problem can be considered as a special case of denoising, where parts of the signal are simply missing. Inpainting task consists in filling the gaps in the signal by training a dictionary using the available parts of the signal.

Given a patch $\mathbf{y}$, we denote the *corrupted components index set* as $I = \{i : y_i \text{ is corrupted}\}$ and with y_{I^c} the vector obtained by removing the corrupted components (i.e whose indexes are in I) from $\mathbf{y}$. D_{I^c} is the matrix obtained by removing from D the columns whose indexes are in I. Finally, we can compute the sparse coefficient $\hat{\mathbf{x}}$ solving the usual sparse coding step (with respect to y_{I^c} and D_{I^c}) and the corrupted image is recovered following the rule

$$\hat{y}_i = \begin{cases} y_i, & \text{if } i \notin I \\ (D\hat{\mathbf{x}})_i, & \text{if } i \in I \end{cases}.$$

We remark that the order in which the pixels are restored plays an important role, since the restored pixels are used for the computation of the remaining missing pixels: a hole in an image is naturally filled from the exterior to the interior. In order to do so we must distinguish the *target region* Ω (area to be inpainted), its boundary $\partial\Omega$ and the *source region* which is the remaining area. Then for all pixels on the target boundary $\partial\Omega$ is assigned a priority $P(p)$ which states the order in which pixels will be recovered.

1.5.4 Classification

Sparse representation can be very useful in supervised classification problems, thanks to its ability, again, to catch the essentials of a signal.

Let us take into consideration, for example, the well-known MNIST dataset [17]. This is made of hundreds of images representing handwritten digits, each one paired with its label. The dataset is split in a training set made of 60 thousands images of 28×28 pixels each and in a training set with the remaining 10 thousand images.

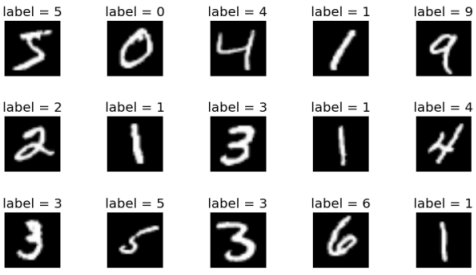

Figure 1.9: example of some elements of the MNIST dataset

Therefore the signal matrix $Y \in \mathbb{R}^{784\times 60000}$ can be visualized as

$$Y = \begin{bmatrix} \mathrm{vec}\left(\boxed{0}\right) & \dots & \mathrm{vec}\left(\boxed{9}\right) \end{bmatrix}$$

and we can also define the label matrix $H \in \mathbb{R}^{10\times 60000}$

$$H = \left[\begin{array}{cccc|cccc|c|c|cccc} 1 & 1 & \dots & 1 & & & & & & & & & \\ & & & & 1 & 1 & \dots & 1 & & & & & \\ & & & & & & & & & & 1 & 1 & \dots & 1 \end{array}\right]$$

which contains the labels of each training image, i.e. if $Y(:,i)$ contains the digit p then $H(:,i) = \mathbf{e}_p = p$ -th canonical vector.

The image classification process via DL is divided into training and testing steps.

In the training step, given the dataset Y we find D and X such that $Y \approx DX$ using DL techniques. It is interesting to notice that the columns of the dictionary D are compressed representations of the redundant information contained in Y, as we can see from Figure 1.10.

Figure 1.10: Visualization of the most significant atom for each digit.

From the sparse representation just found, we can then compute the classifier W which is defined as

$$W := \operatorname{argmin} \|H - WX\|_F \ ,$$

i.e. W is a link between the sparse representation of an image and its label.

Finally the testing step follows: given a new image $\hat{\mathbf{y}}$ we first compute its sparse representation $\hat{\mathbf{x}}$

$$\hat{\mathbf{x}} = \operatorname*{argmin}_{\mathbf{x}} \|\hat{\mathbf{y}} - D\mathbf{x}\|_2 \quad s.t. \quad \|\mathbf{x}\|_0 \leq s \ ,$$

and then, given $\hat{\mathbf{h}} := W\hat{\mathbf{x}}$, compute the prediction of its class as

$$\hat{p} = \operatorname*{argmax}_p |\hat{\mathbf{h}}|_p \ .$$

Often the two stages of the training step are summed up into a unique minimization problem formulated as

$$\min_{D,X,W} \|Y - DX\|_F^2 + \alpha\|H - WX\|_F^2 \ .$$

This formulation forces the representation matrix X to both represent the dataset Y and to classify well.

1.5.5 Compression

Compression is the most known and intuitive application of DL in real-world problems. Indeed, if the dictionary is known, a signal $\mathbf{y} \in \mathbb{R}^m$ can be represented by just s values contained in the vector $\mathbf{x}$. Also in the case with a learned dictionary this approach is convenient for the representation of the signals in Y as $Y \approx DX$. Indeed the coefficient matrix $X \in \mathbb{R}^{n\times N}$ is very sparse, having just sN non-zero elements. In addition the number of atoms n is variable and the number of samples N is usually very huge. Therefore, if N, s and n are such that $mN \gg mn + sN$, Dictionary Learning representation is clearly convenient, achieving a compression rate of

$$\frac{Compressed\ size}{Uncompressed\ size} = \frac{mn + sN}{mN} \approx \frac{s}{m}$$

for high values of N. The most famous examples of compression are JPEG and JPEG-2000 [18] which are applications of sparse representation problem using the Discrete Cosine Transform (DCT) matrix as fixed dictionary. In Chapter 3, dedicated to numerical experiments, we will explore the DL compression properties, taking into consideration different types of dataset and studying the behaviour of the error and other metrics as we modify the parameters n and s.

Chapter 2

Deep Learning and Convolutional Neural Networks

Since ancient times mankind has always described in myths and legends its fantasies on mechanical beings with human-like characteristics. However, the formal foundation of Artificial Intelligence (AI) as a scientific discipline emerged only in the mid-20th century. In 1956, a group of researchers organized the Dartmouth Conference, considered the birthplace of AI, where they aimed to explore how machines could simulate human intelligence. Artificial Intelligence is, indeed, a multidisciplinary field that focuses on creating intelligent machines capable of performing tasks that typically require human intelligence, such as to perceive, reason, learn, and make decisions, mimicking or surpassing human cognitive abilities. Since then, AI experienced less fortunate periods as the so called "AI winter" in 1960s and 1970s as well as the recent times of prosperity called "AI renaissance" fueled by the exponential growth in computing power and the availability of big data which has provided useful resources for training AI algorithms. Furthermore, the birth of Deep Learning, a subfield of Machine Learning, with its ability to automatically learn hierarchical representations, has revolutionized AI applications across various fields, such as image recognition, speech synthesis, and language translation, leading to unprecedented progresses and pushing the boundaries of what is possible in Artificial Intelligence.

In this chapter we will focus on the formal definition of these three fascinating areas as AI, Machine Learning and Deep Learning introducing also the concept of Neural Networks, the quintessential example of Deep Learning models. Let us start by giving a brief introduction to each concept, focusing both on their main differences and common aspects.

Artificial Intelligence (AI) is a broad field, which refers to the use of technologies to build machines and computers that have the *ability to mimic cognitive functions associated with human intelligence*, such as being able to see, understand, and respond to spoken or written language, analyze data, make recommendations, and more [19]. An example of AI is MYCIN [20], an computer program developed at Stanford University

in the 1970s which uses AI to simulate the judgment of a human in the diagnosis and treatment of bacterial infections. MYCIN, using a vast amount of information about various bacteria, their characteristics, the symptoms they cause, and the effectiveness of different antibiotics against specific infections and following some previously set rules, was able to generate a list of possible diagnoses and recommend appropriate antibiotic treatments by asking questions to the patients about their symptoms, laboratory results, and medical history by a text-based interface. These programs can be powerful tools in assisting complex decision-making tasks, but they are limited to the knowledge and rules explicitly programmed into them and may not adapt to new situations or learn from experience like machine learning-based approaches.

Machine Learning (ML) is a field of AI that automatically enables a machine or system to *learn and improve from experience*, by extracting patterns from raw data. An application of Machine Learning techniques in real life is for example spam e-mail filtering. Indeed, ML algorithms can be trained to identify patterns that differentiate spam from legitimate e-mails by simply looking at their contents. The potential of this system is that, over time, it keeps improving its performance by incorporating user feedback and retraining the model with new data.

Deep Learning is a particular kind of Machine Learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts, with *each concept defined in relation to simpler concepts* [21]. The adjective *deep* comes from an attempt to visualize a deep graph, i.e. with many layers, showing how the concepts are built on top of each other. The most powerful and famous Deep Learning application at the moment is GPT-4 [22] on which Chat-GPT is based. Chat-GPT is a state-of-the-art language model developed by OpenAI able to answer questions, provide information, and engage in conversations with humans on a wide range of topic in a way that mimics completely natural human conversation.

Figure 2.1 illustrates the relationship among these three different AI disciplines and a timeline showing the periods in which they have been developed.

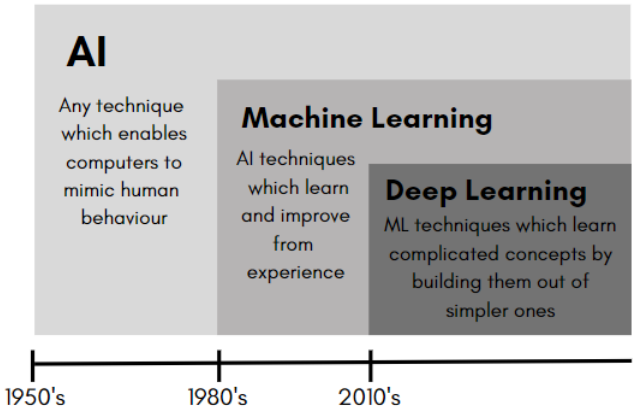

Figure 2.1: A Venn diagram showing how AI, ML and Deep Learning are related.

2.1 Deep Feedforward Networks

The fundamental example of a deep learning model is the Deep Feedforward Neural Network (DFNN), which is a particular kind of Artificial Neural Network, or simply Neural Network (NN). A NN is a computing system initially designed to emulate the biological neural networks that constitute animal brains. As well as in our brain the information spreads from one neuron to another through synapses, a NN can be seen as a directed graph whose vertices, called nodes, represent the neurons and whose edges represent the connections between them. Each neuron processes the information it receives by applying some nonlinear function of a weighted sum of the incoming connections. The nodes of such a network are ordered and they can be aggregated into groups, called layers, which are connected with each other in a hierarchical way. There exist three types of layers : the first layer is called **input layer** and is made of the input variables, similarly the last one is called **output layer** and is made of the output variables of the neural network. Finally, the **hidden layers** are all the layers in between and contain all the variables which are neither inputs nor outputs. The purpose of the hidden layers is to capture and transform the input data into more abstract and higher-level representations. When there is one or more hidden layers we talk about Deep Neural Networks (DNNs).

Formally speaking, the NN architecture is a machine learning procedure whose goal is to determine an operator F which approximates a given functional [1]

$$f : \mathcal{V}^{(0)} \to \mathcal{V}^{(L)}$$

through a minimization process of a loss functional. Notice that the functional f is known only on a subset of the domain $\mathcal{V}^{(0)}$ called *training set* and that $\mathcal{V}^{(L)}$ represent the output space. Recalling the idea of hidden layers just introduced, the approximant functional F can be obtained by the following composition

$$F = F^{(0)} \circ ... \circ F^{(L)} \ , \tag{2.1}$$

where $F^{(l)} : \mathcal{V}^{(l)} \to \mathcal{V}^{(l+1)}$ for each $l = 0, ..., L$, represents the operations performed between the l-th and the $(l+1)$-th layer and takes in input a function $h^{(l)} : \mathcal{G}^{(l)} \to \mathbb{R}$ which represents the l-th layer. In practical implementations, the domains $\mathcal{G}^{(l)}$ are discrete sets $\{1, ..., N_l\}$ and $h^{(l)}$ usually denotes $\{h^{(l)}(j)\}_{j \in \mathcal{G}^{(l)}}$. In a DFNN, each $F^{(l)}$ acts on $h^{(l)}$ by first applying a linear operation on it and then a nonlinear one. We will denote the linear operation as

$$W^{(l)} h^{(l)} + b^{(l)} \ , \tag{2.2}$$

where $W^{(l)} : \mathcal{V}^{(l)} \to \mathcal{V}^{(l+1)}$ is a linear operator that can be represented as $N_{l+1} \times N_l$ matrix $\{W^{(l)}(j, i)\}_{j \in \mathcal{G}^{(l+1)}, i \in \mathcal{G}^{(l)}}$ whose elements are called *weights* and $b^{(l)} \in \mathcal{V}^{(l+1)}$ is a further additive element often referred to as *bias* term. The nonlinear function will be

[1] notation from [23].

denoted with $s^{(l+1)} : \mathcal{V}^{(l+1)} \to \mathcal{V}^{(l+1)}$ and called *activation function*. Summing up, each layer $h^{(l+1)}$ is obtained as a function of the previous layer by the following recursive formula

$$h^{(l+1)} := F^{(l)}(h^{(l)}) = s^{(l+1)}(W^{(l)}h^{(l)} + b^{(l)}) \ . \tag{2.3}$$

The architecture of a Deep FNN is shown in Figure 2.2.

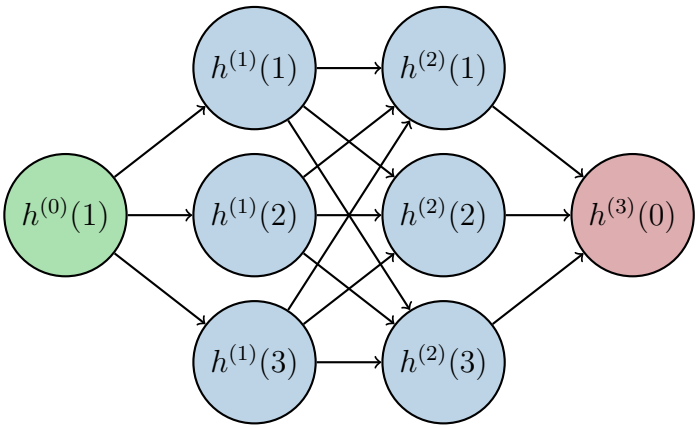

Figure 2.2: An example of DFNN with one input variable (green), two hidden layers (blue) of 3 variables each and one output variable (red).

To better understand how a neural network works in practice, in Example 3 we report the most simple example of a DFNN, the McCulloch-Pitss neuron.

Example 3. (The McCulloch-Pitts (MCP) neuron [24])
The McCulloch-Pitts neuron is a precursor of DFNNs. It is the first attempt of a mathematical representation of the neural activity and it dates back to 1943. MCP neuron's architecture, as shown in Figure 2.3, consists of some inputs $h^{(0)}(1), h^{(0)}(2) \in \{0, 1\}$ which represent the signal received by the neuron, a weighted summation function which linearly combines the inputs with weights $W^{(0)}(1,1), W^{(0)}(1,2) \in \{-1, 1\}$ representing the strength and influence of the connection and a thresholding function s which returns 1 if the weighted sum of the inputs exceeds a threshold represented by the bias b, and 0 otherwise. In other words, if the signal is enough strong its information is passed forward, otherwise it is stopped. Generalizing to more input variables, given the input vector $h^{(0)} = [h^{(0)}(1), ..., h^{(0)}(N_0)]^T$, the neuron will return an output $F(h^{(0)}) \in \{0, 1\}$ which can be mathematically formulated as

$$F(h^{(0)}) = s\left(\sum_{i=1}^{N_0} W^{(0)}(1,i)h^{(0)}(i) - b\right) \ ,$$

where s is the activation function

$$s(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases} \ .$$

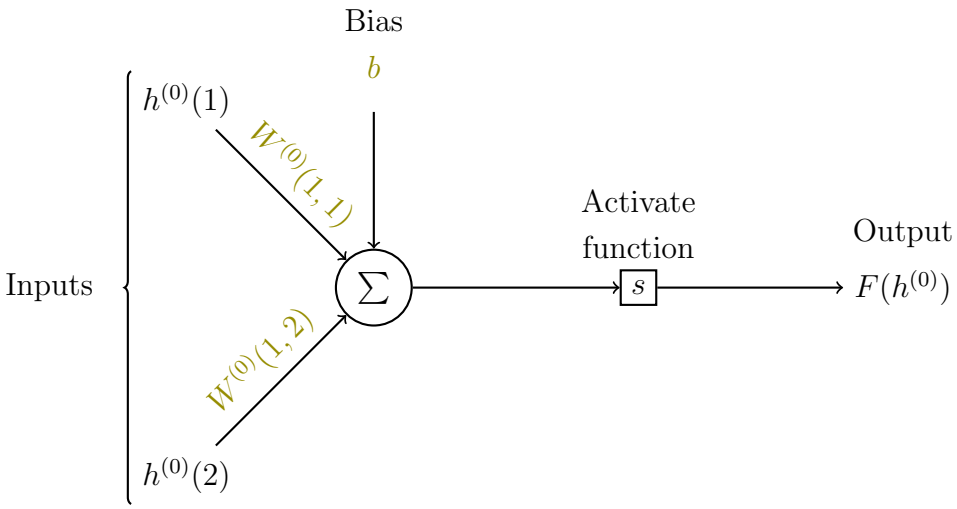

Figure 2.3: Visualization of MCP neuron with 2 inputs variables. We reported in green the free parameters.

Example 3 not only describes clearly how a neuron works, but also shows how a neural network can be seen as a tool able to approximate a target functional $f(h^{(0)})$ by learning the right parameters, such as the weights $\{W^{(l)}(j,i)\}_{j\in\mathcal{G}^{(l+1)},i\in\mathcal{G}^{(l)}}$ and the bias $\{b^{(l)}(i)\}_{i\in\mathcal{G}^{(l)}}$, which drive $F(h^{(0)};u)$ to match with the given functional $f(h^{(0)})$. For sake of simplicity, u denotes the parameters $u=(w,b)=(\{W^{(l)}(j,i)\}_{j\in\mathcal{G}^{(l+1)},i\in\mathcal{G}^{(l)}},\{b^{(l)}(i)\}_{i\in\mathcal{G}})$.

We conclude this section with a pivotal result: the Universal Approximation Theorem. The theorem, reported below, states that a standard multilayer feed-forward network with a single hidden layer containing a finite number of hidden units and with arbitrary activation function can universally approximate any function in $C(\mathbb{R}^m)$, the space of continuous functions on $\mathbb{R}^m$. The output units are always assumed to be linear and, for notational convenience, we explicitly formulate the theorem only for the case with only one output unit.

Theorem 1. Universal Approximation Theorem [25]
Let s be an arbitrary activation function. Let $H\subseteq\mathbb{R}^m$ be compact.
Then for all f $\in C(H)$, for all $\epsilon>0$: there exist $n\in\mathbb{N}$, $b(i),W^{(0)}(i,j),W^{(1)}(i)\in\mathbb{R}, i\in\{1,...,n\}, j\in\{1,...,m\}$ such that

$$F^{(n)}(h(1),...,h(m))=\sum_{i=1}^{n}W^{(1)}(i)s\left(\sum_{j=1}^{m}W^{(0)}(i,j)h(j)+b(i)\right)$$

is an approximation of the function f; that is

$$\|f-F^{(n)}\|<\epsilon\ ,$$

where n denotes the number of hidden units.

For further details and the proof of the theorem we recall [25]. Moreover, Kurt Hornik showed in [26] that it is not the specific choice of the activation function, but rather the

multilayer feedforward architecture itself which gives neural networks the potential of being universal approximators, stressing out the importance of the network of being deep.

Remark 2.1.1. (On the importance of HPC in Deep Learning)

We conclude this section showing how the use of HPC, presented in detail in Appenidix B, plays a fundamental role in Deep Learning. GPT-4 [22], whose abilities have already been introduced at the beginning of this section, is a great example. Indeed, GPT-4 parameter details are undisclosed but rumored to be around 100 trillion. Training such a massive neural network in a canonical way would take years and years, this is why all the latest neural networks, taking advantage of their highly parallelizable nature, make an impressive use of GPU and HPC to speed up computational times and lower the costs. The exact settings of GPT-4 are still not known, but what is known is that the training of GPT-3, its previous version, was done on 1024 GPUs and took 34 days [27].

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a particular kind of neural networks designed for processing grid-like structured data such as time-series or images. Its name is due to the use of the *convolution* function as linear operator $W^{(l)}$ in at least one of its layer l. In detail, as shown in Figure 2.4, a CNN is usually composed by three different types of layers: convolutional layers, pooling layers and fully connected layers. Indeed, the hidden layers of such a network are given by the sequentially application of a convolutional layer, a nonlinear activation function and a pooling layer several times, while the output layer is returned only after passing through many fully connected layers. For this reason, in CNN context, (2.1) can be rewritten more specifically as [2]

$$F = \phi \circ \Upsilon^k \circ ... \circ \Upsilon^1 \circ \Pi \circ s \circ T_{\Psi_f^{(L)}} \circ ... \circ \Pi \circ s \circ T_{\Psi_f^{(0)}} \quad ,$$

where :

- $T_{\Psi_f^{(l)}} : \mathcal{V}^{(l-1)} \to \mathcal{V}^{(l)}$ represents the l-th convolutional layer, with $l \in \{1, ..., L\}$;
- $s : \mathcal{V}^{(l)} \to \mathcal{V}^{(l)}$ represents a nonlinear layer;
- $\Pi : \mathcal{V}^{(l)} \to \mathcal{V}^{(l)}$ represents a pooling layer;
- $\Upsilon^j : \mathcal{V}^{(L)} \to \mathcal{V}^{(L)}$ represents the fully connected layers, with $j \in \{1, ..., k\}$,
- $\phi : \mathcal{V}^{(L)} \to \mathcal{V}^{(L)}$ represents the nonlinear output layer.

In this section we will walk through the basic workings of a CNN, following all the steps layer by layer.

[2] notation from [28].

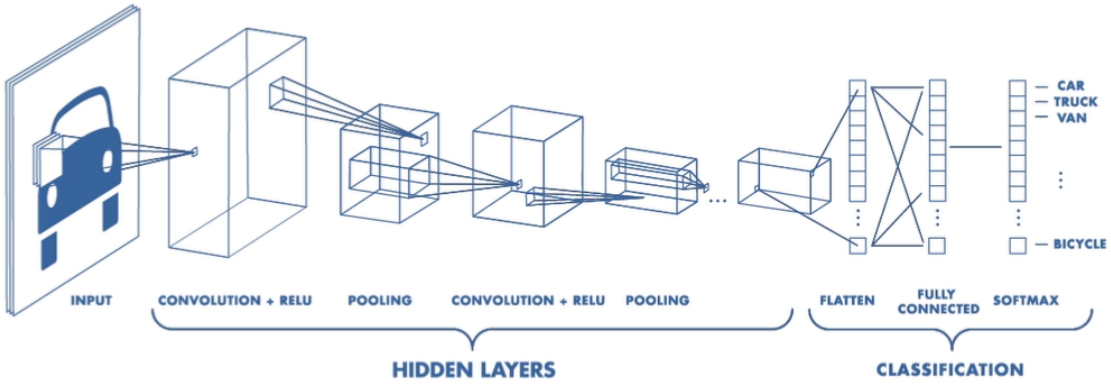

Figure 2.4: Architecture of a CNN [29].

2.2.1 Convolutional layer

CNNs use convolutional layers to extract local features from the input images. Each convolutional layer consists of multiple learnable filters, also called kernels, that scan the input data in a sliding window manner, as it will be shown later. The kernels are small matrices that *convolve* with local patches of the input to produce activation maps, also known as feature maps. Each filter is specialized in detecting a particular pattern or feature. By applying multiple filters, the network can learn different features at different layers.

Notice that time series are defined by a finite number of observation through a finite time-interval and images are defined by a finite number of pixels. Therefore, in this context, it would be more realistic to assume the inputs in a discrete and finite domain. For this reason, we start with the definition of discrete convolution function in one-dimensional case.

Definition 5. [21]

Given two functions $h, k : \mathbb{Z} \to \mathbb{R}$, we define the discrete convolution between h and k as

$$(h * k)(t) := \sum_{m=-\infty}^{\infty} h(m)k(t-m) \, . \tag{2.4}$$

In CNN terminology, the first argument, h, is often referred to as the input and the second one, k, as the kernel. Notice that the convolution function is symmetric, since

$$(h*k)(t) := \sum_{m=-\infty}^{\infty} h(m)k(\underbrace{t-m}_{=:a}) = \sum_{t-a=-\infty}^{\infty} h(t-a)k(a) = \sum_{a=-\infty}^{\infty} k(a)h(t-a) =: (k*h)(t) \, .$$

This implies that $(h * k)(t)$ can be equivalently rewritten as

$$(h * k)(t) = \sum_{m=-\infty}^{\infty} k(m)h(t-m) \, . \tag{2.5}$$

In this type of application is likely to have the kernel k with finite support in a discrete set as $\{-M, -M+1, \dots, M-1, M\}$. We therefore can rewrite (2.5) as

$$(h * k)(t) := \sum_{m=-M}^{M} k(m)h(t-m) \ .$$

However, for computational issues, in many neural network libraries *cross-correlation* function is preferred to convolution. Cross-correlation is defined as

$$(h \star k)(t) := \sum_{m=-\infty}^{\infty} h(m)k(t+m) \ . \tag{2.6}$$

Similarly to convolution, cross-correlation is symmetric as well, hence (2.6) can be rewritten as

$$(h \star k)(t) := \sum_{m=-\infty}^{\infty} k(m)h(t+m) \ .$$

In machine learning applications the input is usually a multidimensional array of data: black and white images are two dimensional and RGB images are three dimensional. We therefore conclude reporting the definition of convolution (2.7) and cross-correlation (2.8) in N-dimensional case :

$$(h*k)(t_1, ..., t_m) := \sum_{m_1=-\infty}^{\infty} \sum_{m_2=-\infty}^{\infty} ... \sum_{m_N=-\infty}^{\infty} h(m_1, ..., m_N)k(t_1 - m_1, t_2 - m_2, ..., t_N - m_N) \tag{2.7}$$

$$(h\star k)(t_1, ..., t_m) := \sum_{m_1=-\infty}^{\infty} \sum_{m_2=-\infty}^{\infty} ... \sum_{m_N=-\infty}^{\infty} h(m_1, ..., m_N)k(t_1 + m_1, t_2 + m_2, ..., t_N + m_N) \tag{2.8}$$

where $h, k : \mathbb{Z}^N \to \mathbb{R}^N$.

Applying everything presented so far to NN context we obtain a formal definition of convolutional layer. Recalling the notation used in (2.3), a convolutional layer is defined as

$$h^{(l+1)} = s^{(l+1)}(W^{(l)}h^{(l)} + b^{(l)}) = s^{(l+1)}(\Psi_k^{(l+1)} \star h^{(l)} + b^{(l)}) \ , \tag{2.9}$$

where $\{\Psi_k^{(l+1)}\}_{k=1,\dots,n_l+1}$ is the set of filters associated to the $(l+1)$-th layer. Equation (2.9) can be further developed by replacing the convolution $\Psi_k^{(l+1)} \star h_l$ with its matricial form, as shown in detail at the end of Example 4.

Example 4.

Being a linear operator, discrete cross-correlation between the input and the kernel can be equivalently formulated in a matricial way. Given, for example, an image H as input data h and a matrix K with the set of learnable parameters representing the kernel k, the operation described by discrete cross-correlation, as we will see in this example,

is equivalent to slide across the height and width of the input image, performing a dot product between the kernel and restricted portions of the input image.

Suppose having an image of dimension 3×4 pixels and a 2×2 kernel defined as

$$X := \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \quad , \quad K := \begin{bmatrix} w & x \\ y & z \end{bmatrix} .$$

To determine the cross-correlation between them, we start by computing the dot-product between the 2×2 portion of the image X in the upper-left corner (the green highlighted part in Figure 2.5) and the kernel K. Then, we set the *stride* which is a parameter that control how far, i.e. how many pixels, the filter will move from one portion of the image to the next one. In this example we set the stride equal to one, therefore we shift the filter by one column and repeat the dot-product, as shown in Figure 2.6.

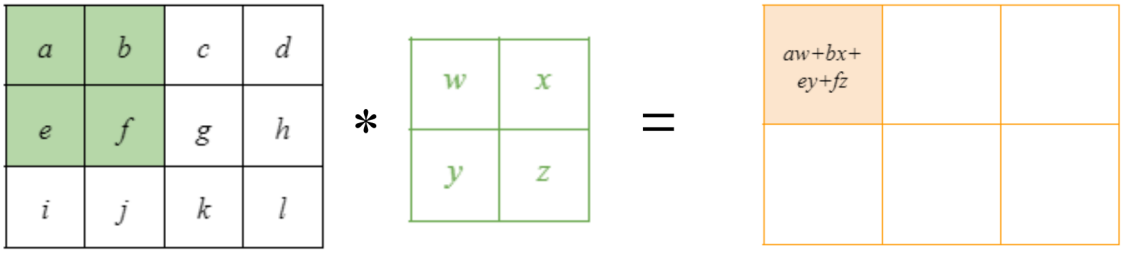

Figure 2.5: First convolution

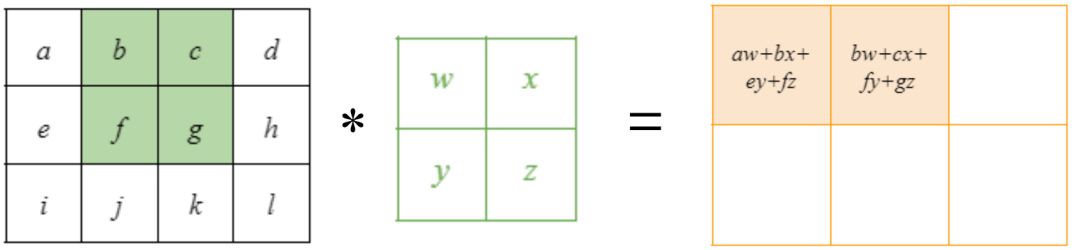

Figure 2.6: Second convolution

We repeat it until all the portion of the image have been taken into consideration according to the dimension of the image, of the kernel and the stride. In Figure 2.7 are shown all the 2×2 portions of the image X that are multiplied to the kernel K.

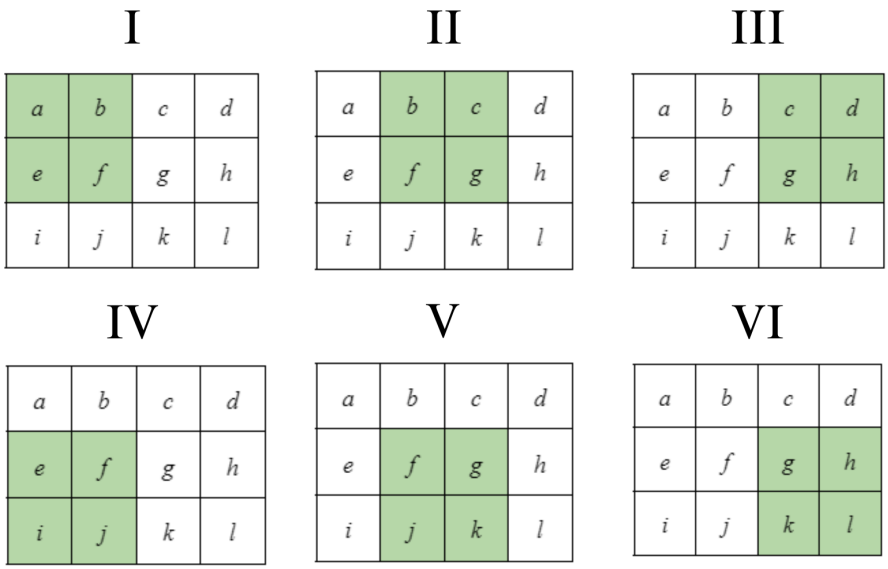

Figure 2.7: How a 2×2 filter passes through the entire image with the stride equal to 1.

Finally, putting all the information together, cross-correlation can be also expressed as a single matrix-vector multiplication as shown in Figure 2.8, in which the matrix is very sparse, because the kernel is small with respect to the input, and has few distinct values. Notice that each row of such a matrix is the same as the previous one, just shifted to the right by 1.

$$\begin{bmatrix} w & x & 0 & 0 & y & z & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & w & x & 0 & 0 & y & z & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & w & x & 0 & 0 & y & z & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w & x & 0 & 0 & y & z & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w & x & 0 & 0 & y & z & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & w & x & 0 & 0 & y & z \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ j \\ k \\ l \end{bmatrix} = \begin{bmatrix} aw + bx + ey + fz \\ bw + cx + fy + gz \\ cw + dx + gy + hz \\ ew + fx + iy + jz \\ fw + gx + jy + kz \\ gw + hx + ky + lz \end{bmatrix}$$

Figure 2.8: An example of cross correlation convolution in two dimensions represented as matrix multiplication. [21]

A real-case example on how discrete cross-correlation actually affects an input image is shown in detail in Example 5.

Example 5.

In this example we will illustrate the impact of discrete cross-correlation on an image, pointing out its ability to extract local features. Let us consider three simple filters defined as

$$K_1 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}, \; K_2 = \begin{bmatrix} 1 & 1 & -1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}, \; K_3 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}.$$

As their structure suggests, we expect K_1 to promote vertical features, K_2 to encourage horizontal ones and K_3 to detect the contours. Picking up the image with vertical stripes already used in Example 1, Figure 2.9 shows both the original image and the output of the convolution between the image and the three filters K_1,K_2 and K_3.

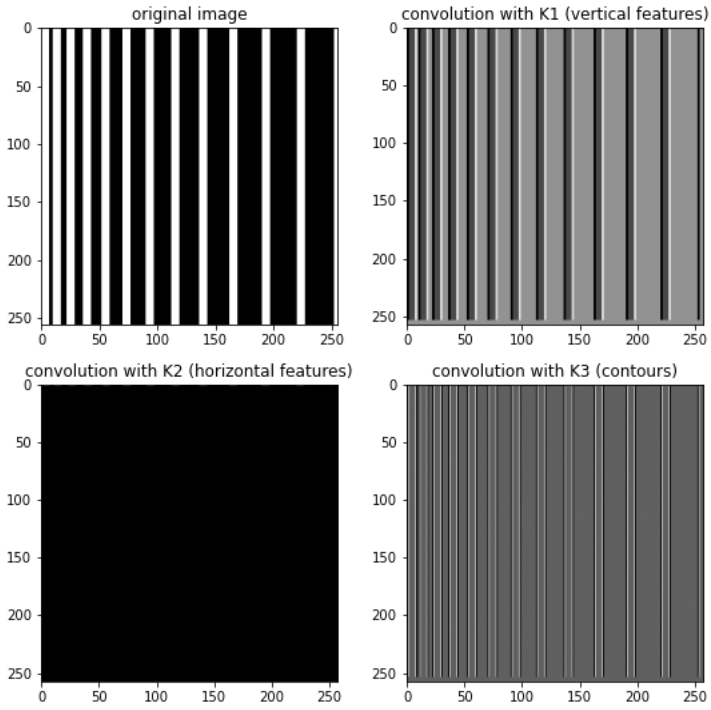

Figure 2.9: Effect of convolution between an image with vertical stripes (upper left corner) and K_1,K_2 and K_3 filters.

As expected, convolution with K_1 catches the vertical lines of the image, while convolution with K_2, which should emphasize horizontal features, returns a black image as expected, since the original image has not any horizontal pattern. Convolution with K_3 highlights, instead, the left and right edge of each stripe.

Convolution leverages three important ideas that can help improving a machine learning system: sparse interactions, parameter sharing and equivariant representations [21]. Let us describe each one of them in detail.

- **sparse interaction :**
  in trivial neural network every output unit interacts with every input unit. Convolution neural networks, as already noticed, are characterized by sparse interaction, since the kernel is smaller than the input. This implies the storage of fewer parameters, reducing the memory requirement of the model. Figure 2.10 shows that in an usual neural network each node affects all the units in the next layers, while in a convolutive one it affects only some of them. However, in a deep convolutive network, although direct connections are sparse, a unit is indirectly connected to all or most of the input layer.

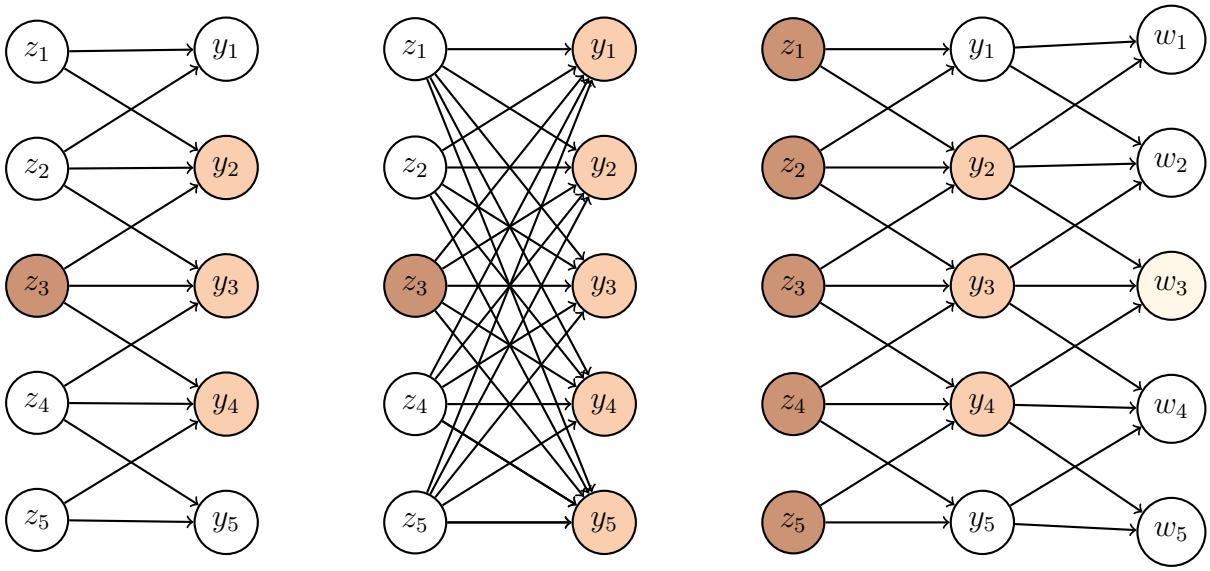

Figure 2.10: On the left : convolutional neural network, z_3 affects 3 units.
In the middle : nonsparse neural network, z_3 affects all units. On the right : convolutional neural network, all the units z in the first layer affect w_3.

- **parameter sharing :**
  in a traditional NN, each element of the kernel matrix is used exactly once when computing the output of a layer. In a convolutional NN, instead, each member of the kernel is used almost at every position of the input. Therefore, if there are m inputs and n outputs, in the traditional case the kernel matrix K requires $m \times n$ parameters, while, in the convolutional case, k parameters are enough, where k is usually several orders of magnitude less than m. Since m and n are usually roughly the same size, k is almost insignificant compared to $m \times n$.
  In Example 4, k is equal to 4 which is much lower than $m \times n = 12 \times 6 = 72$.
  Thanks to sparse interaction and parameter sharing, convolution is by far more

efficient than dense matrix multiplication in terms of memory requirements.

- **equivariant representations :**
  a function f is said to be equivariant to a function g when applying g to the input before applying f yields the same output as applying f to the output of g, i.e. $f(g(x)) = g(f(x))$. The convolution of the filter with the input of the layer allows the network to be equivariant to translation. Indeed, given a translation operator T_v, follows that

$$([T_v h] \star k)(t) = \sum_m T_v h(m) k(t+m) = \sum_m h(m+v) k(t+m) = \tag{2.10}$$

$$= \sum_m h(m) k(t+m+v) = \sum_m h(m) k((t+v)+m)) = \tag{2.11}$$

$$= [T_v(h \star k)](t) \tag{2.12}$$

  Therefore, shifting the intensity of all image pixels to the right by one unit and then applying convolution yields the same result as first applying convolution then shifting. The same does not hold with other transformations like rotation or scaling, for which other mechanisms are needed [30].

2.2.2 Hidden activation functions

After each convolution and also at the very end of the network, an activation function is applied element-wise to introduce nonlinearity into the network, providing great approximation flexibility to the model. The introduction of nonlinearity is crucial: consider, for example, a neural network with the identity as activation function at each layer. The depth of the model would be no more meaningful, since the NN could be simply reformulated with just one linear combination of the input variables. Linear models are indeed appealing due to their simplicity and therefore reliability, but they have the obvious defect that they cannot learn any complex task. That is the reason why nonlinear activation functions play a pivotal role in NN.

We distinguish two types of activation functions: the hidden ones, explored in this section, are used for the computation of the hidden units, while the output ones, presented in sub-section 2.2.4.1, return, as the word suggests, the output units. The former are designed to capture the most important features of the signal and add nonlinearity to the model, while the choice of the latter depends on the type of the model output we want to obtain. For further details we recall [21]. Let us focus now on the most used hidden activation functions.

The **Rectified Linear Unit Activation Functions** (ReLU) and its generalizations are right now the most used activation functions for hidden units in CNNs because of their property to be piecewise linear. Applying these functions to the output of a linear

transformation yields a nonlinear transformation still remaining very close to linear, preserving many of the properties that make linear models easy to optimize. The "classic" ReLU is defined as

$$ReLU : \mathbb{R} \to \mathbb{R} \ :$$

$$ReLU(z) = max\{0, z\} \ .$$

It follows that it is zero when the argument is negative and equal to the argument when it is equal or greater than zero, as shown in Figure 2.11.

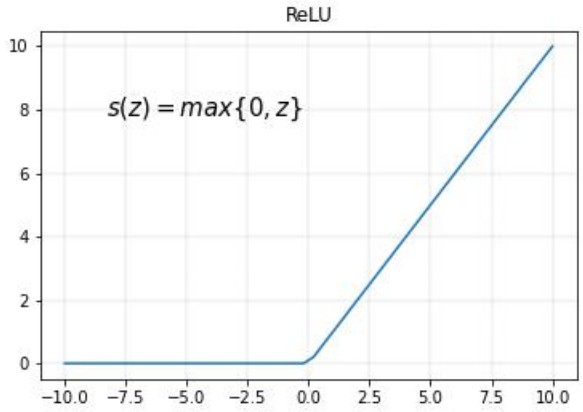

Figure 2.11: Rectified Linear Unit Activation Function.

A drawback of ReLU is that all the negative values become zero immediately generating the so called "dead neurons", which are neurons with negative (and potentially high) weights and bias that, being cancelled by ReLU, will no longer be updated in the following layers. The presence of dead neurons reduces, with no doubt, the model's capacity to get much information as possible from data. To overcome this problem has been proposed a couple of slightly different versions of ReLU activation function which are shown in Figure 2.12 and defined as

$$Leaky \ / \ Parametric \ ReLU(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha z, & \text{otherwise} \end{cases} = \max\{0, z\} + \alpha \min\{0, z\} \ .$$

We talk about **Leaky ReLU** when the parameter α is set equal to 0.01, otherwise, if α is learned with the other neural-network parameters, it is called **Parametric ReLU**. Usually, in CNNs context, this step is called *Detector Stage*.

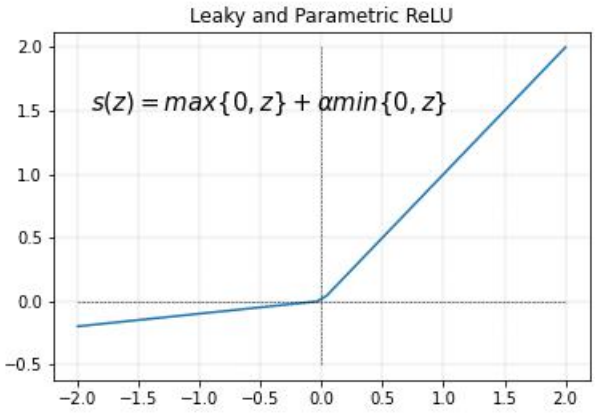

Figure 2.12: Leaky or Parametric ReLU Activation Function.

2.2.3 Pooling layer

After the image has passed through a convolutional layer and ReLU activation function, the next step is the Poolying layer. Pooling layers are used to downsample the feature maps generated by the convolutional layers while retaining the most salient information. It helps reducing the computational complexity of the network and provides a form of translational invariance, meaning the network can still recognize patterns even if they are slightly shifted. The most common type of pooling used in CNNs is **max pooling**. In max pooling, each region of the input feature map is divided into small sub-regions, and the maximum value within each sub-region is selected as the representative value for that region. This process effectively captures the most prominent feature within each region and reduces the number of parameters. Another common type of pooling is average pooling, where instead of selecting the maximum value, the average value within each region is computed. By selecting the maximum or average value within each region, slight variations in the position or orientation of the features are less likely to affect the overall result. This property is referred to as "Invariance to local translation" and it can be very useful if we care more about whether some feature is present than exactly where it is.

See Figure 2.13 for an example of how max-pooling works and its advantages.

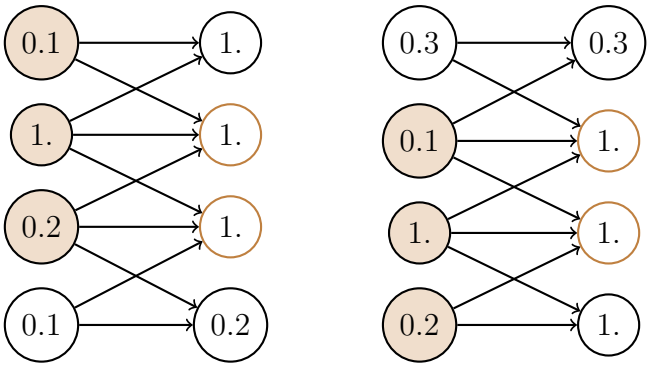

Figure 2.13: On the left : a view of the output of a convolutional layer. The first layer shows the units given by the detector stage while the second one shows the outputs of max-pooling with a pooling region width of three pixels. On the right : a view of the same network after the input has been shifted down by one pixel. Each value in the first layer has changed, but only half of the values in the second layer have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location [21].

Remark 2.2.1. (Difference between translation Equivariance and Invariance)
In section 2.2.1 we have proved that convolution operator is *equivariant* to translations, while in this section we have stated that max-pooling is *invariant* to local translations. These two properties are often confused due to their similar names, but they actually have different natures. The first one, translation equivariance, means that to every shift of the input correspond the same shift to its convolution, while the second one, translation invariance, is the ability to ignore positional translations of the input. The combination of these two properties leads the whole CNN model to be translational invariant. Indeed, consider a generic hidden layer $h^{(l)}$ of a CNN. We know that the next hidden layer $h^{(l+1)}$ is given by a combination of a convolutional layer, the ReLU and a pooling layer as

$$h^{(l+1)} = F^{(l)}(h^{(l)}) = \Pi \circ s \circ T_{\Psi_f^{(l)}}(h^{(l)}). \tag{2.13}$$

Equation (2.13) proves how translation equivariance of convolution and translation invariance of max pooling lead to a translation invarance to the whole model. Since in hidden and output layers does not occur any translation, we can just focus on the case of translation of the input layer h_0 for a vector $\mathbf{v}$ as follows.

In the next layers the CNN proceeds as if there were no translation. CNN's translation invariance is a crucial property. When determining whether an image contains an a face, for example, the location of the eyes with pixel-perfect accuracy will not be needed : the basic information the model needs is just the presence of an eye on the left side of the face and an eye on the right side of the face

Before moving to the description of the last layers of a CNN, as the fully connected layers and the output layer, we report in the following a real-case example showing how convolutional and pooling layers actually affect an input image.

Example 6.
In this example we trained a real CNN with two convolutional layers made of, respectively, 32 and 64 filters of 3×3 size, each one followed by ReLU activation function and a max-pooling layer with a pooling region width of two pixels on the full MNIST dataset. Figure 2.14 summarizes all the characteristics of this CNN. Choosing as sample image

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten (Flatten)           (None, 1600)              0

 dropout (Dropout)           (None, 1600)              0

 dense (Dense)               (None, 10)                16010

=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
```

Figure 2.14: Profile of the considered CNN.

the digit shown in Figure 2.9, the goal of this example is looking at which changes it undergoes passing through all these layers.

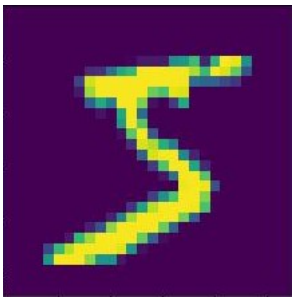

Figure 2.15: sample image representing digit 5 from MNIST training set.

In particular, Figure 2.16 and 2.17 show how the two convolutional layers affect it.

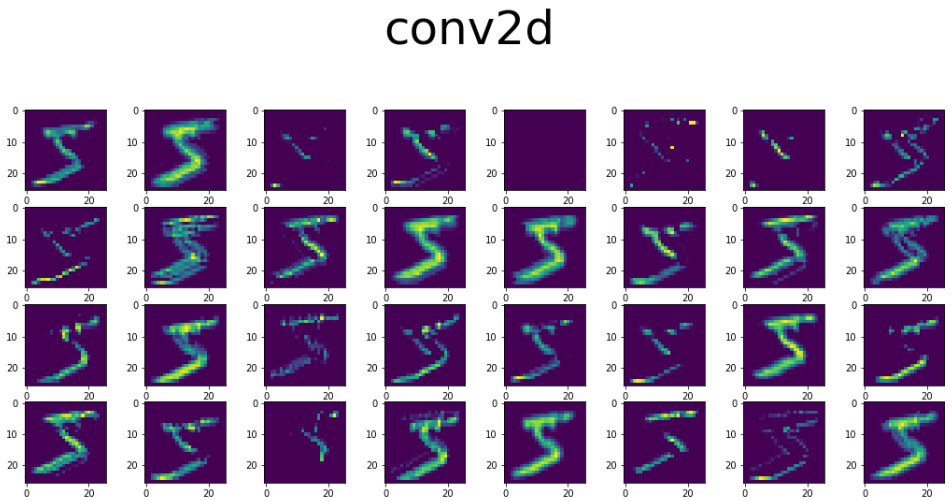

Figure 2.16: Visualization of the output of the first convolutional layer.

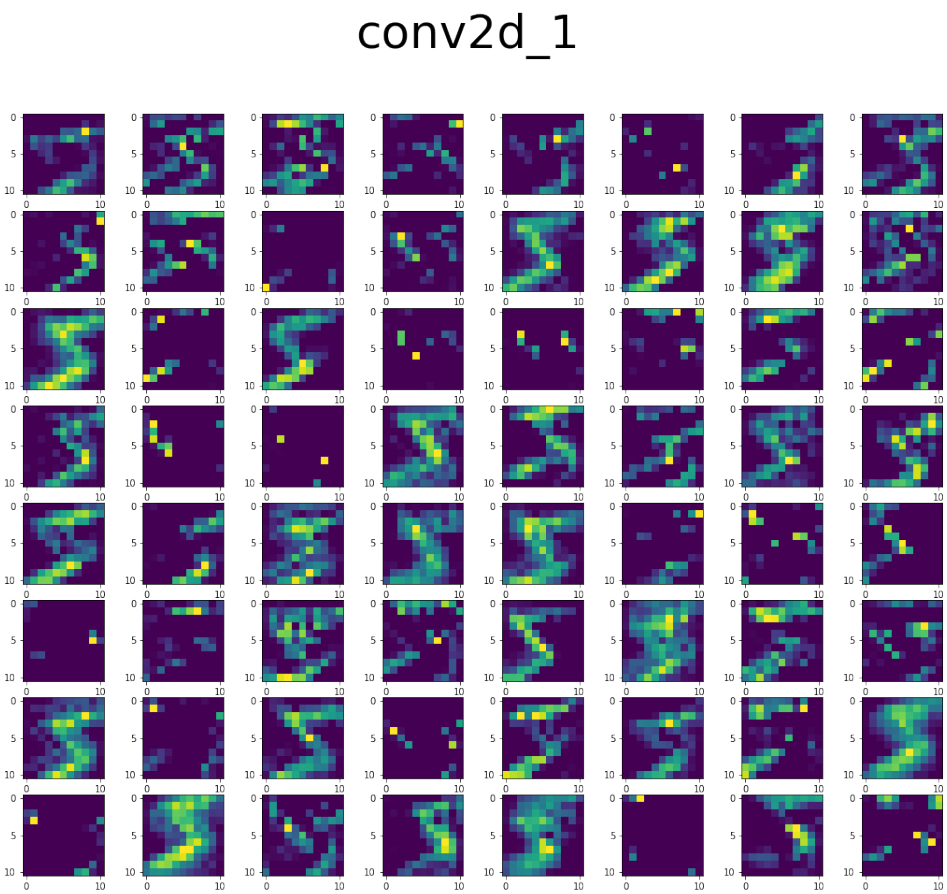

Figure 2.17: Visualization of the output of the second convolutional layer.

First of all, we notice that, while the first layer is able to catch the most important feature of the sample acting as an edge detector, in contrast, digging deeper into the network, some results more abstract, catching finer details. Furthermore, according to what we stated in the previous sections, the size of the output decreases after each layer. Indeed, looking at the column "Output Shape" of the CNN profiler in Figure 2.14, we can see that, starting from a 28×28 image, we get 32 images of size 26×26 pixels after the first convolutional layer made of 32 filters , which become 13×13 images after the first max-pooling layer. Since the second convolutional layer is made of 64 filter, this layer returns 64 images of size 11×11 pixels which become 5×5 after the last max-pooling layer.

2.2.4 Fully connected layers and Output layer

After several convolutional and pooling layers, the resulting feature maps are flattened into a 1-dimensional vector and fed into one or more fully connected layers. These layers resemble traditional neural network layers, where each neuron is connected to every neuron in the previous layer. The fully connected layers perform high-level reasoning and classification based on the learned features.

The final fully connected layer in the CNN is the output layer, returned by the output activation function. Its configuration depends on the specific task the CNN is designed for. For example, in image classification, the output layer may consist of neurons representing different classes, and the network's output would be a probability distribution over these classes, indicating the likelihood of the input image belonging to each class. Let us see in detail a couple of output activation functions.

2.2.4.1 Output activation functions

In CNNs the most used activation functions are the Sigmoid and Softmax.

The **Sigmoid** or **Logistic** activation function is the most simple one. It is defined as

$$sigmoid : \mathbb{R} \to (0, 1) \ :$$

$$sigmoid(z) = \frac{1}{1 + e^{-z}} \ .$$

Its shape is shown in Figure 2.18.

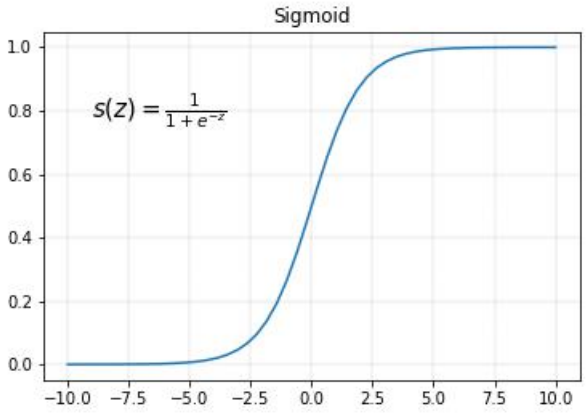

Figure 2.18: Sigmoid activation function.

Since its codomain is the real interval $(0, 1)$, the Sigmoid activation function is especially used for models where the output has to be a probability. In particular, it is an ideal activation function for binary classification problems such as the usual e-mail spam classifier which uses the Sigmoid to determine the probability of an e-mail to be classified as spam or not. For multiclass classification, as the recognition of digits in MNIST dataset, we can use instead the **Softmax** function. The Softmax function is defined as

$$softmax : \mathbb{R}^n \to (0,1)^n \ :$$
$$softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \ , \quad \forall \ i = 1, ..., n$$

where n is the number of classes. Softmax with $n = 2$ returns exactly Sigmoid, indeed given $\mathbf{z} = [z, 0]$ follows that

$$softmax(\mathbf{z})_0 = \frac{e^z}{e^0 + e^z} = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}} = sigmoid(z) \ ,$$
$$softmax(\mathbf{z})_1 = \frac{e^0}{e^0 + e^z} = \frac{1}{1 + e^z} = 1 - sigmoid(z) \ .$$

2.3 Training a (deep feedforward) neural network

Once the design of a NN is determined and, in particular, of a CNN, the next task is training it. The training of a neural network is a supervised learning process which learns to optimize its parameters (weights and biases) by minimizing an average loss function usually defined as

$$R(u; \hat{\mathbf{y}}, \mathbf{y}) = R(w, b; \hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} L(\hat{y}_i - y_i)$$

which quantifies, sample by sample, the discrepancy between the predicted output of the network ($\hat{\mathbf{y}} := F(h^{(0)}; u)$) and the true output ($\mathbf{y} := f(h^{(0)})$) and then computes

the average value over all the n samples. The choice of the "fiducial term" L depends on the specific task the neural network is designed to solve. Different tasks, such as classification or regression, may require different types of loss functions. Here follow the most used terms.

- **Mean Squared Error (MSE):** commonly used in regression problems, where the goal is to predict continuous values. It measures the average squared difference between the predicted and true values, i.e.

$$MSE(u; \hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \ .$$

- **Binary Cross-Entropy:** used in binary classification tasks, where the goal is to classify inputs into one of two classes. It measures the dissimilarity between the predicted probabilities and the true binary labels as

$$BCE(u; \hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{n} \sum_{i=1}^{n} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i) \ .$$

- **Categorical Cross-Entropy:** used in multi-class classification problems, where the goal is to classify inputs into one of several mutually exclusive classes. It computes the average dissimilarity between the predicted class probabilities and the true class labels by generalizing the formula above as

$$CCE(u; \hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{M} y_i^j \cdot log(\hat{y}_i^j) \ ,$$

  where M is the number of classes.

Binary and Categorical Cross-Entropy loss functions' formulations are based on the definition of Entropy and Cross-Entropy used in Information Theory. In this field, the Entropy of a random variable represents the average level of "information" or "surprise" of the variable's outcomes and it is defined as follow.

Definition 6.
Given a discrete random variable X, which takes values in $\mathcal{X}$ and has distribution $p : \mathcal{X} \to [0, 1]$, the Entropy of X is defined as

$$H_p(X) := -\sum_{x \in \mathcal{X}} p(x) \ log(p(x)) \ .$$

Notice that $-log(p(x)) = log\left(\frac{1}{log(p(x))}\right)$, hence the more an event is unlikely to happen, i.e. $p(x)$ close to 0, the more we get information from this event if it actually happen, i.e. the more is the Entropy. Similarly, when an event is almost sure, i.e. $p(x)$ is close to 1, we do not gain much information when it occurs.

Cross-Entropy is nothing more than a generalization of the Entropy and it is defined as follow.

Definition 7. [31]

Given p and q two different probability distributions over the same underlying set of events $\mathcal{X}$, the Cross-Entropy of the distribution q relative to a distribution p over a given set $\mathcal{X}$ is defined as

$$H_{pq}(X) := -\sum_{x \in \mathcal{X}} p(x)\ log(q(x))\ ,$$

If $p(x)$ would match to $q(x)$, Cross-Entropy and Entropy values will obviously match as well.

The loss functions just described are often non-convex, due to the nonlinearity of the activation functions in the hidden layers. This implies that neural networks are usually trained by using iterative, gradient descent-based algorithms that simply drive the cost function to a very low value, rather than providing the convergence to a global minimum. We therefore briefly recall how the gradient-descent method works.

2.3.1 The Gradient Descent method

Gradient descent, also called steepest descent or batch gradient descent, is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$. Starting from an initial approximation x_0, the algorithm aims to move iteratively along the steepest descent direction towards a minimum of f. We call "descent direction for f at x" a step p that makes an angle of strictly less than $\frac{\pi}{2}$ radians with $-\nabla f(x)$, i.e. a p such that

$$p^T \nabla f(x) < 0\ . \tag{2.14}$$

It follows that the steepest descent direction for f at x is $p = -\nabla f(x)$, making it clear why it is also called "gradient descent". The whole procedure is summarized in Algorithm 6.

Algorithm 6: (Batch) Gradient Descent Algorithm [32]

Given a starting solution x_0, the number of iterations $maxit$ and the stopping tolerance ϵ

for $k = 0, ..., maxit$ **do**

- compute $\nabla f(x_k)$;
- compute stepsize γ;
- compute $x_{k+1} = x_k - \gamma \nabla f(x_k)$;
- **if** $\|\nabla f(x_{k+1})\| \leq \epsilon$ **then**
  - **return**. (x_{k+1} is the approximate solution)
- **end**

end

The algorithm produces a descending sequence of approximations $\{x_k\}$, indeed, recalling (2.14) and assuming γ positive and sufficiently small, follows that

$$f(x_{k+1}) - f(x_k) = f(x_k - \gamma \nabla f(x_k)) - f(x_k) = \\ = f(x_k) - \gamma (\nabla f(x_k))^T \nabla f(x_k) + O(\gamma^2) - f(x_k) < 0$$

The Gradient Descent algorithm plays an important role in the training of a neural network, since it is employed in the minimization of the loss function R with respect to any weight or bias in the network. Denoting with $W^{(l)}(j,k)$ the weight which connects the k-th neuron in the $(l-1)$-th layer and the the j-th neuron in the l-th layer, and with $b^{(l)}(j)$ the bias relative to the j-th neuron of the l-th layer, the goal is to compute the following partial derivatives

$$\frac{\partial R}{\partial W^{(l)}(j,k)} \, , \, \frac{\partial R}{\partial b^{(l)}(j)} \tag{2.15}$$

The algorithm designed for the computation of such partial derivatives is called **Backpropagation** and it is described in the following section.

2.3.2 Backpropagation

Backpropagation algorithm is an efficient way to compute the gradients of the loss function with respect to the model's parameters and it is entirely based on the Chain Rule of Calculus, whose definition is reported here.

Definition 8. [21]

Let x be a real number and let f,$g : \mathbb{R} \to \mathbb{R}$. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \, .$$

Generalizing to $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$, if $y = g(x)$ and $z = f(y)$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \, .$$

Now we are ready to compute the partial derivatives in (2.15). Let us start from the partial derivative of R with respect to the weights. From the Chain Rule, we know that

$$\frac{\partial R}{\partial W^{(l)}(j,k)} = \sum_{i=1}^{N_l} \frac{\partial R}{\partial z^{(l)}(i)} \frac{\partial z^{(l)}(i)}{\partial W^{(l)}(j,k)} \tag{2.16}$$

where the sum is over all neurons i of the l-th layer and $z^{(l)}(i) = \sum_{m=1}^{N_{l-1}} W^{(l)}(i,m) h^{(l-1)}(m) + b^{(l)}(i)$ is the weighted input to the i-th neuron in layer l, i.e. $h^{(l)}(i)$. Of course, the

weighted input $z^{(l)}(i)$ of $h^{(l)}(i)$ neuron depends on the weight $W^{(l)}(j,k)$ only when $i=j$ and $m=k$, so $\dfrac{\partial z^{(l)}(i)}{\partial W^{(l)}(j,k)}$ vanishes when $i \neq j$. Therefore (2.16) becomes

$$\frac{\partial R}{\partial W^{(l)}(j,k)} = \frac{\partial R}{\partial z^{(l)}(j)} \frac{\partial z^{(l)}(j)}{\partial W^{(l)}(j,k)} =: \delta^{(l)}(j) h^{(l)}(k) \ , \tag{2.17}$$

where, for sake of simplicity, we denoted with $\delta^{(l)}(j)$ the partial derivative $\dfrac{\partial R}{\partial z^{(l)}(j)}$. For what concern the partial derivative of the cost function with respect to the bias, we can proceed similarly obtaining

$$\frac{\partial R}{\partial b^{(l)}(j)} = \sum_{i=1}^{N_l} \frac{\partial R}{\partial z^{(l)}(i)} \frac{\partial z^{(l)}(i)}{\partial b^{(l)}(j)} = \frac{\partial R}{\partial z^{(l)}(j)} \frac{\partial z^{(l)}(j)}{\partial b^{(l)}(j)} = \frac{\partial R}{\partial z^{(l)}(j)} = \delta^{(l)}(j) \ . \tag{2.18}$$

Summing up, thanks to backpropagation, the gradients of the loss function in (2.15) can be computed as

$$\frac{\partial R}{\partial W^{(l)}(j,k)} = \delta^{(l)}(j) h^{(l)}(k) \quad , \quad \frac{\partial R}{\partial b^{(l)}(j)} = \delta^{(l)}(j) \ .$$

The computation of $\delta^{(l)}(j)$, which we recall corresponds to $\dfrac{\partial R}{\partial z^{(l)}(j)}$, strongly depends on the shape of the cost function R, hence we cannot further develop this partial derivative here, but we can say that it heavily depends on the outputs, since the cost function is defined on the outputs as well. Moreover, notice that all the loss functions seen at the beginning of this section are defined as an average over the cost functions R_i of each training sample i, as $R = \dfrac{1}{n} \sum_{i}^{n} R_i$. This property allows us to compute separately the partial derivatives $\dfrac{\partial R_i}{\partial b}$ and $\dfrac{\partial R_i}{\partial W}$ for each single training sample which is more straightforward, and then take the average of the gradients of all the training samples using therefore just one step of gradient descent in each iteration. However, the number of training samples can be quite large and processing all of them at each iteration in one go can be computationally expensive and memory-intensive. This problem can be solved by considering at each iteration only a subset of the training sample. Such a subset is usually referred to as **batch**. When the gradient in gradient descent algorithm is not computed using the entire training dataset at each iteration, but only with the samples in the batch, we talk about "Mini-Batch Gradient Descent". Usually the batch samples are extracted at random, therefore the Gradient algorithm becomes a stochastic optimization process. The algorithm can be iterated many times on the training dataset. The number of these iterations is referred to as **epochs**. For further details about stochastic gradient algorithms in Machine Learning we refer to [33].

In the next page we sum up everything seen so far in this chapter in an single algorithm which describes the training phase of a CNN.

Algorithm 7: Training of a CNN

Set $max\_epoch = \#epochs$, $L = \#$layers, $\beta = \#$batches, $t =$ batch size, $maxit =$ maximum number of iterations for each batch, s activation function, Π pooling layer function, $\{\Upsilon^j\}_{j=1,...,r}$ fully-connected layers functions, ϕ softmax function, ϵ the stopping tolerance and γ stepsize[a].

Given a dataset $\mathcal{D}$ compute the training set $\mathcal{T}^{(r)} = \{(x_1, y_1), ..., (x_N, y_N)\}$.

Split the training set $\mathcal{T}^{(r)}$ in batches $\{B_k\}_{k=1,...,\beta}$ s.t. $B_k = \{(x_k, y_k), ..., (x_{k+t}, y_{k+t})\}$.

for *epoch = 1,...,max_epoch* **do**
  for B_k *in* $\{B_k\}_{k=1,...,\beta}$ **do**
    for (x_i, y_i) *in* B_k **do**
      $h^{(0)} = \hat{x}_i$
      Compute the predicted outputs $\hat{y}_i$
      for $l = 1, ..., L$ **do**
        $h^{(l)} = \Pi(s(W^{(l)}h^{(l-1)} + b^{(l)}))$
      end
      $\hat{y}_i = \phi \circ \Upsilon^r \circ ... \circ \Upsilon^1(h^{(L)})$,
      Compute the loss functions R_i
      $R_i = loss(\hat{y}_i - y_i)$, where *loss* refers to che custom loss function
    end
    $$R = \frac{1}{t}\sum_{i=k}^{k+t} R_i$$
    if $R < \epsilon$ **then**
      break
    end
    for $l = L - 1, ..., 1$ **do**
      for (x_i, y_i) *in* B_k **do**
        Compute the partial derivatives with backpropagation
        $$\frac{\partial R_i}{\partial b^{(l)}(j)} = \delta^{(l)}(j)$$
        $$\frac{\partial R_i}{\partial W^{(l)}(j,k)} = \delta^{(l)}(j)h_i^{(l)}$$
      end
      Update the parameters with Gradient Descent
      $$W^{(l)}(j,k) = W^{(l)}(j,k) - \gamma\sum_{i=k}^{k+t}\frac{\partial R_i}{\partial W^{(l)}(j,k)}$$
      $$b^{(l)}(j) = b^{(l)}(j) - \gamma\sum_{i=k}^{k+t}\frac{\partial R_i}{\partial b^{(l)}(j)}$$
    end
  end
end

[a] The stepsize can be constant or diminishing through the iterations. The choice of the stepsize is crucial for the convergence of the algorithm and different stepsizes define distinct methods. Such methods are reviewed in [33].

2.4 Training, Validation and Testing

To asses the approximation quality of a NN we must introduce the concept of training, validation and test set. These sets are subsets of the available data and have different purposes. The training set is the largest subset of data used to train a NN as just shown in section 2.3. The training set can consist of just input data without any guess of their output, in the case of "unsupervised" training, or, as in our case, can consists of input data and corresponding known output values or "targets" as happens in the "supervised" case. During the training phase, the model learns from this data by adjusting its internal parameters, such as weights and biases, to minimize the difference between predicted outputs and the true labels. The validation set is a disjoint subset of data that is used during the training process to fine-tune the model's hyperparameters. Hyperparameters are configuration settings that determine the network structure and that cannot be learned directly from the training data. Examples of hyperparameters are the number of hidden layers, batch size, which is the number of samples in a batch, and epochs. By evaluating the model's performance on the validation set, different combinations of hyperparameters can be compared, and the best configuration can be selected. This helps to prevent overfitting, a new concept presented in detail in section 2.5. Finally, the test set is the remaining disjoint subset of data and it is used to assess the final performance of a trained neural network model. It simulates real-world scenarios by providing data that the model has never encountered before. The test set should be representative of the data that the model is expected to handle in practice. By evaluating the model on the test set, it is possible to estimate how well it will perform on unseen data.

2.5 Overfitting and Underfitting

Overfitting and Underfitting are two common problems that occur when training machine learning models. These issues arise when the model's performance does not generalize well to new, unseen data. We talk about overfitting when a machine learning model performs exceptionally well on the training data but fails on validation/test data. In other words, an overfitted model has learned the training data too closely, including noise or random fluctuations, instead of capturing the underlying patterns of the data. A clear sign of overfitting is therefore a low training error, but an high validation/test error. On the other side, underfitting occurs when a model is too simple and not only it fails on new data, but performs badly also on training data. It is characterized by high training and validation/test errors. Figures 2.19 and 2.20 show the typical behaviour of the accuracy and loss function, respectively in an overfitted and underfitted model with respect to the number of epochs.

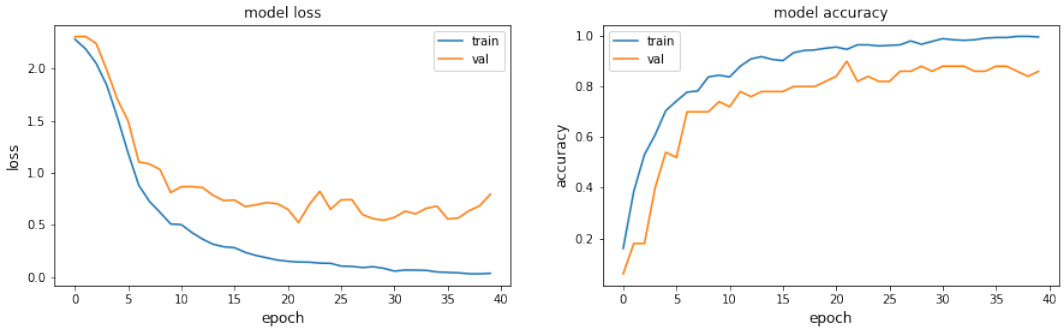

Figure 2.19: Behaviour of accuracy and loss function in a overfitted model.

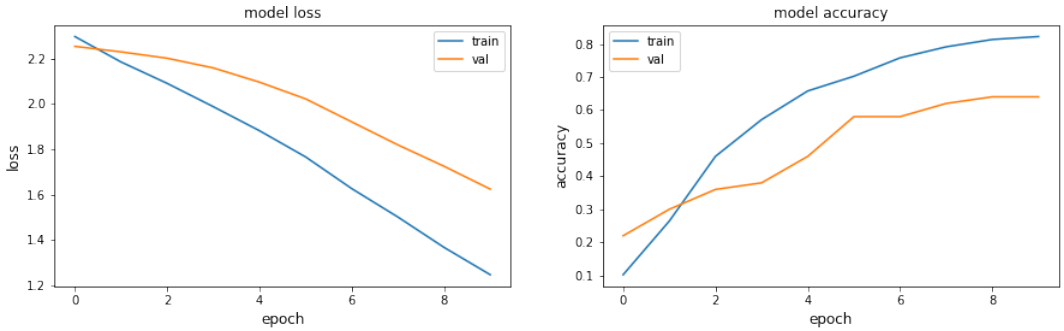

Figure 2.20: Behaviour of accuracy and loss function in a underfitted model.

The plots shown in the Figures 2.19 and 2.20 result from two neural networks trained on 500 digits of MNIST' training set and tested on 100 digit of MNIST' test set. In particular, Figure 2.21 report the profiler of the CNN used for the overfitted example, while Figure 2.22 shows the profiler of the CNN used for the underfitted example.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 128)         73856

 max_pooling2d_2 (MaxPooling  (None, 1, 1, 128)        0
 2D)

 flatten (Flatten)           (None, 128)               0

 dropout (Dropout)           (None, 128)               0

 dense (Dense)               (None, 10)                1290

=================================================================
Total params: 93,962
Trainable params: 93,962
Non-trainable params: 0
```

Figure 2.21: Profiler of the overfitted model.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 8)         80

 max_pooling2d (MaxPooling2D  (None, 13, 13, 8)        0
 )

 flatten (Flatten)           (None, 1352)              0

 dropout (Dropout)           (None, 1352)              0

 dense (Dense)               (None, 10)                13530

=================================================================
Total params: 13,610
Trainable params: 13,610
Non-trainable params: 0
```

Figure 2.22: Profiler of the underfitted model.

Chapter 3

Numerical Results

The purpose of this final chapter is to show the significant utility of Dictionary Learning (DL) techniques in real-world applications. In particular, we will emphasize how DL provides a valuable tool for data compression within the context of Digital Twins. The chapter is organized as follows: in section 3.1 we will provide a brief introduction to Digital Twins, while in section 3.2 we will explore the application of DL in this context, analyzing its performance on various types of datasets and frameworks. All the codes necessary to reproduce the experiments shown in this chapter are available at the following link: https://github.com/lauracavalli/DL4DT.git.

3.1 Brief introduction to Digital Twins

Digital twins are up-to-date virtual replicas of physical objects, systems, processes or any entity we want to represent digitally. These state-of-art technologies have different fields of application, ranging from manufacturing to healthcare, education to urban simulations. In particular, digital twin technologies play an important role in Industry 4.0, the fourth industrial revolution, where the digitalisation of the production systems has revolutionized manufacturing processes, increasing productivity and profitability, reducing the impacts of disruptions and enabling data-driven decision-making. Figure 3.1 helps visualize how a digital twin approximately works.

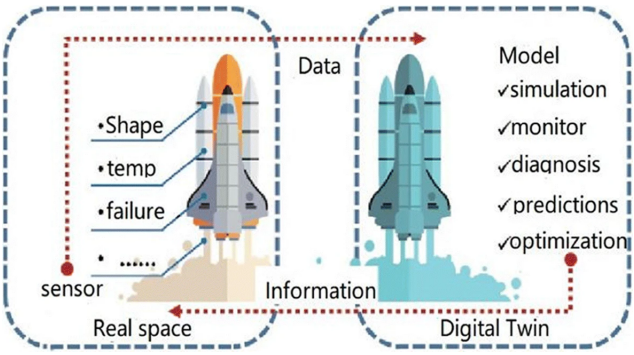

Figure 3.1: Schematic representation of a digital twin [34].

As reported in [35], the origin of the Digital Twin concept can be traced back to the 1960s when NASA first introduced the idea of "twinning" for their Apollo program. The concept involved creating physical duplicates on Earth that accurately matched their systems in space. This allowed them to simulate various scenarios, test different cases and conditions, and evaluate the behavior and performance of their systems. However, the concept did not gain significant traction until the early 2000s when Michael Grieves introduced the notion of digital twins in the manufacturing industry. Subsequently, in 2017, Gartner recognized it as one of the top 10 strategic technology trends, and many industry giants adopted the concept, gaining increased attention and significance.

This section provides an outline of a typical workflow for a digital twin.

The first step is collecting data and information of the physical system, such as temperature, pressure and other relevant variables, by using various types of sensors such as IoT devices which send periodic updates on the status of their connected devices. With "IoT (Internet of Things) device" we refer to a physical device, placed on the system, which is connected to internet and has the ability to send and receive data. With the help of IoT devices, the relevant data are transmitted to a middleware platform, where they are collected and processed. The next step is to generate meaningful information from raw data collected so far. During this stage, big-data architectures and analysis techniques can be used for deriving business decisions. Furthermore, deploying AI and Machine Learning algorithms can enhance the transformation process by predicting trends, detecting correlations and providing insights. As a result, a digital twin not only enables to digitally represent in real-time the physical counterpart of the system and monitor its current values, but also allows an analysis of future scenarios, detecting incoming anomalies or find its best settings. Finally, after completing all the analyses, a feedback is sent to an interfaces on the physical system through web or mobile platforms that would allow users to visualize real-time updates of the physical systems and inform the user of the outcome, which may include warning about an incoming anomaly, suggesting future system developments, or providing advice on how to proceed in a given situation. Figure 3.2 provides a schematic overview of the three fundamental layers that

constitute a digital twin. On the left, there is the "hardware" component, which is composed of IoT sensors placed on the physical system, which could range from a car or a plane to an industrial process or even an entire city. Additionally, this layer includes actuators that detect the process output message and actuate it within the system. In the middle, we have the "middleware" phase, where the collected data are transmitted, stored, and processed within the digital counterpart. Finally, on the right, the "software" layer is presented. Here, the data play a crucial role in generating analytics, dashboards, digital simulations of the environment, and designing future states of the system with the assistance of AI and ML techniques. To handle these vast amount of data [1] and perform such computationally-intensive tasks, HPC system are often employed in this stage, enabling the DT framework to operate at its full potential, allowing for more sophisticated simulations, real-time monitoring, optimization, and integration of advanced technologies such as the training of massive neural networks on extensive datasets.

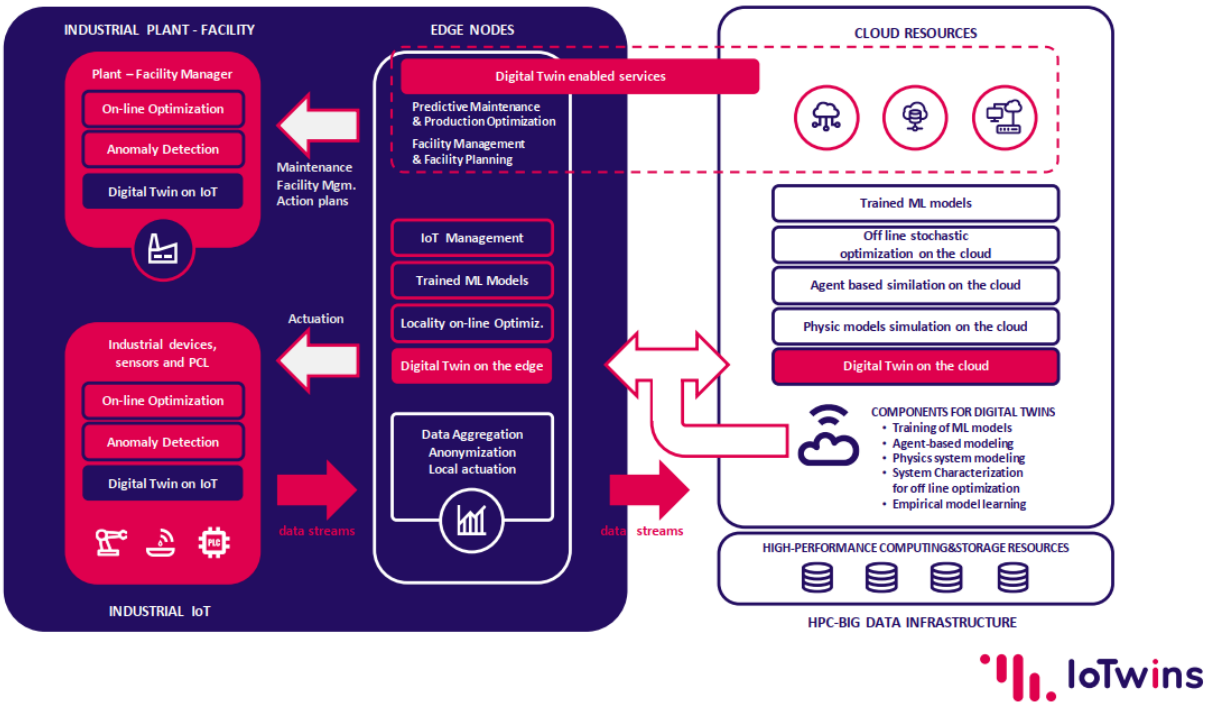

Figure 3.2: Components of a digital twin system. Image taken from a presentation of IoTwins [37], an European project which aims to lower the barriers for the uptake of Industry 4.0 technologies to optimize processes and increase productivity, safety and environmental impact through the digital twin.

Let us now analyze the advantages and disadvantages of digital twin technologies. As mentioned earlier, some of their advantages include real-time monitoring, predictive maintenance, and the ability to simulate potential scenarios.

[1] For example in the "M100 ExaData" project [36] were collected 49.9 T in two year and a half, with sampling rate of few seconds.

They also offer highly-informed decision support, optimized scheduling and increased efficiency, all of which contribute to faster development cycles and cost savings.

However, in order to make sure the digital twin represent a synchronized real-time copy of the physical system, the network connecting the physical and digital systems should fulfill a set of requirements such as low latency of real-time communication, data security and quality. Regarding the last two requirements, it is essential to implement measures such as authentications and Blockchain technologies to ensure that data cannot be intercepted or modified by untrusted external users during their transmission. As for latency, a high level of synchronization can ensures the digital twin and the represented entity mirror each other as closely as possible and offer a high quality of experience of users in real time communications. This task, however, is not easy to achieve, due to the large amount of data required to establish a reliable digital representation. To reduce the communication latency two techniques can be used: deploying high speed network links by using high speed cables such as optical fibers, or data compression. The purpose of the next section is to demonstrate that DL techniques can indeed improve digital twins' synchronization providing an exceptional compression tool.

3.2 Integration of Dictionary Learning techniques in a digital twin framework: the workflow

As already mentioned, reducing data latency is one of the main challenges within the DT context. To overcome this problem we propose the following workflow. First of all, the data are collected from the physical device and represented as a matrix Y. The entire process of DL factorization is then applied to Y, resulting in the extraction of the dictionary D and the sparse matrix X. Both D and X are transmitted to the digital counterpart. Subsequently, the compressed data $\tilde{Y}$ is restored on the digital system using D and X as $\tilde{Y} = DX$, making it possible to apply AI techniques. Sometimes, repeating this procedure every time data need to be transferred between the two systems could be impractical. In such situations the dictionary D can be computed only once and stored both on the physical and digital system. Consequently, whenever there is a need to compress and transfer data, the physical system only needs to compute the sparse matrix X using sparse coding techniques and send it to the digital system. The latter approach shows several advantages. Firstly, Sparse Coding demands significantly lower computational resources compared to full DL, as proved in Chapter 1. Secondly, transmitting only the sparse matrix X is considerably easier and faster than sending both the dictionary D and the sparse matrix X separately. Moreover, users have the flexibility to specify under which conditions the dictionary D has to be updated, in order to have more reliable results. For example, a reasonable choice can be updating the dictionary after a fixed period of time or when the accuracy of the AI algorithm on the compressed dataset starts to decrease too much. We will refer to these conditions as *user_conditions*

in the forthcoming Algorithm 8. As we will prove, this workflow is very effective since DL techniques allow to massive compression preserving main important features of the dataset.

The workflow above has been schematically resumed in the following algorithm.

Algorithm 8: Workflow of a digital twin process with DL techniques.

```
Collect data on the physical counterpart as a matrix Y.
Send Y to the digital system.
Compute the dictionary D and the sparse matrix X with DL factorization of Y
 on the digital system.
i = 0
while True do
    if i==0 then
        Send the dictionary D to the physical system and store it.
    end
    else
        Compute X with sparse coding on the physical system.
        Send X to the digital system.
    end
    i = i + 1
    Compute Ỹ = DX on the digital system.
    Apply ML algorithm using Ỹ as dataset.
    if user_conditions then
        break
    end
end
```

3.2.1 A first analysis of DL compression performances.

This section will explore the performance of our DL algorithm "DL_ompqr_parallel_ksvd" on two types of datasets, images and timeseries, and show how its behavior changes varying the parameters such as the number of atoms n and sparsity level s. For images, we employed the well-known MNIST dataset [17], while for timeseries, we used the FordA dataset [38]. Detailed descriptions of both datasets can be found in Appendix A.

For both datasets, we analyzed the behaviour of the error

$$E = \frac{\|Y - DX\|_F}{\sqrt{mN}} \tag{3.1}$$

with respect to the iterations. The number of iterations has been set equal to $K = 20$ for MNIST and $K = 40$ for Ford-A. The sparsity level s was set based on the number of classes k_p of the database. In the case of the MNIST dataset, we set the sparsity level as $s = 2k_p, 3k_p$ and $5k_p$ as if we want to consider 2, 3 or 5 samples for each class,

while for the Ford-A dataset, which is less redundant, we set $s = 20k_p, 25k_p$ and $30k_p$. The number of atoms, denoted as n, was set based on the compression percentage we wanted to achieve which were 40, 50, 60, 70 % . With MNIST dataset we pushed the compression up to 80 %, due to its significant redundancy. The compression percentage **cp** is defined as

$$\mathbf{cp} = \left(1 - \frac{mn + sN}{mN}\right) \times 100 \, .$$

Figures 3.3, 3.4 and 3.5 present an analysis of the DL performance on the MNIST dataset. In detail, Figure 3.3 shows the error's behaviour with respect to the iterations for $s = 20$ and different compression rates, Figure 3.4 considers $s = 30$ and Figure 3.5 considers $s = 50$. In this last case we do not report the scenario with 40 and 50 % of compression, as they would have required over 24 hours to complete 20 iterations. From the theory we know that our algorithm does not converge to a solution, but that it produces, instead, a descending sequence of approximations. Therefore, fixed the parameters s and n, we expect the error to decrease at each iteration.

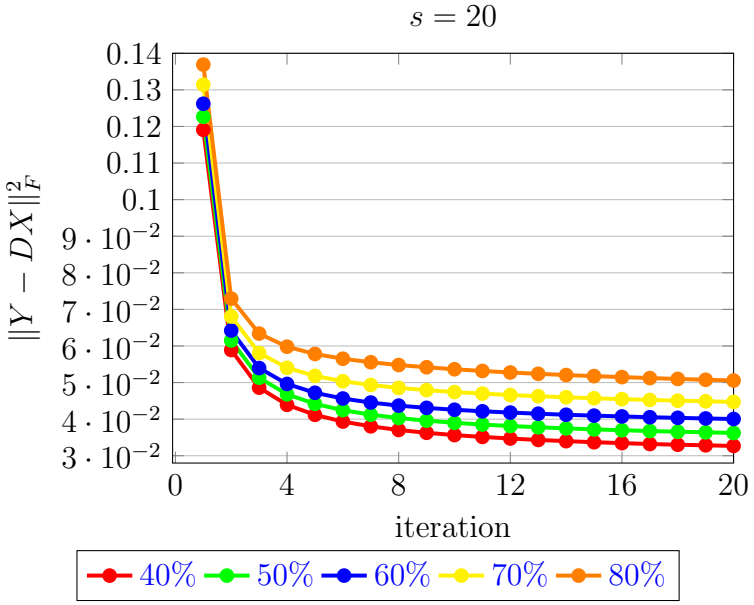

Figure 3.3: DL on MNIST : error w.r.t n. iterations for $s = 20$

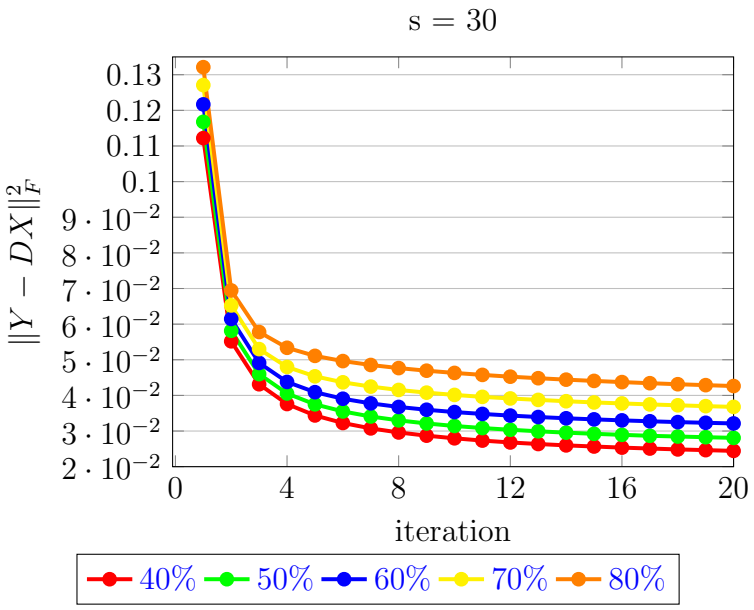

Figure 3.4: DL on MNIST : error w.r.t n. iterations for $s = 30$.

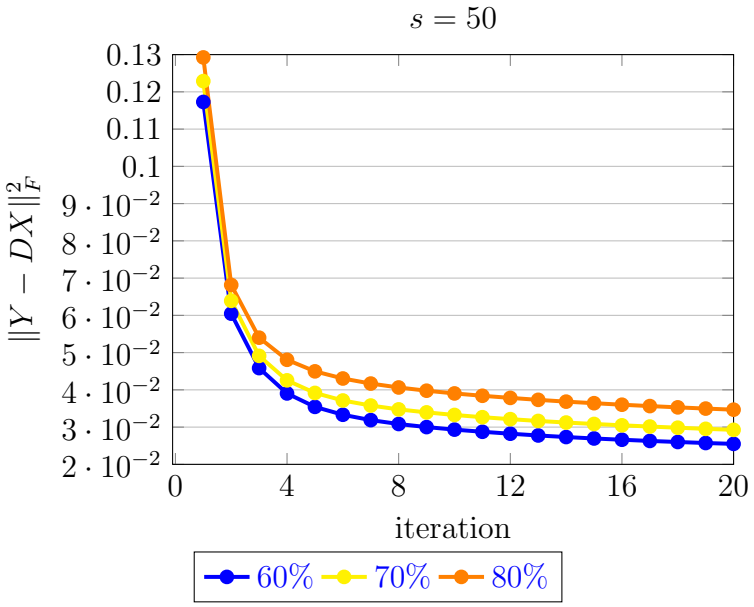

Figure 3.5: DL on MNIST : error w.r.t n. iterations for $s = 50$.

As expected the error shows a descending behaviour with respect to the number of iterations. In particular, after 20 iterations the algorithm reaches an error between 0.023 and 0.05, requiring around 17 hours : therefore, the parallelization of OMP-QR is crucial; otherwise the algorithm would have required an impractical amount of time to finish the iterations. Furthermore, we expect that, when the number of atoms n is held constant, the error decrease as the parameter s increases, as signals are represented by a larger number of atoms. Similarly, we expect that increasing n leads to even better results, as a broader set of atoms can be chosen as the most representative for each signal. Figure 3.6 shows the error's behaviour with respect to the sparsity level n, for

different values of s while Figure 3.7 reports the same experiment with respect to the compression percentage.

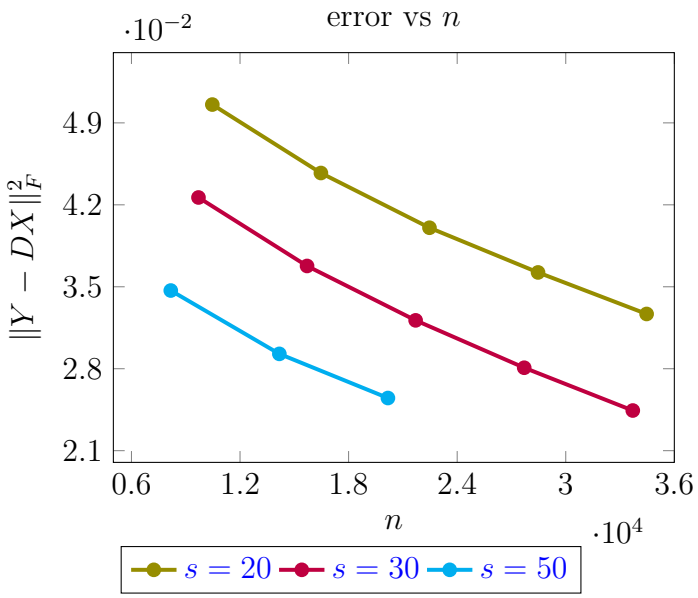

Figure 3.6: DL on MNIST : error w.r.t the number of atoms n

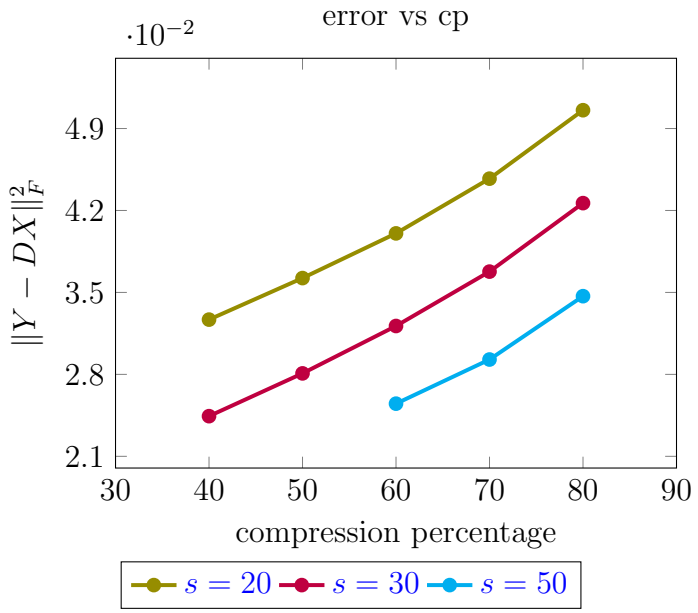

Figure 3.7: DL on MNIST : error w.r.t the compression percentage

The obtained results coincide with our expectations. Furthermore, we observe that DL techniques can yield remarkably high levels of compression. For instance, it can achieve 80% compression with an error of 0.034, or even achieve an error as low as 0.024 with a compression percentage of 40%.

These impressive compression performances are due to the redundant nature of the MNIST dataset, which is a result of its low variability. Indeed, as shown in Figure 3.8, the MNIST dataset consists of about 6.000 images for each digit all sharing the same background and frequently showing digits in identical positions or with similar handwriting styles.

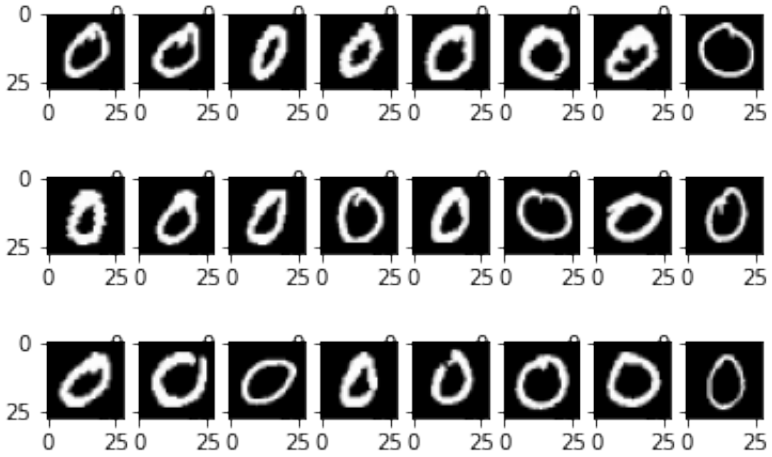

Figure 3.8: Some examples of MNIST' digit 0.

At this point the reader could wonder when an error value can be consider satisfactory. For this reason we reported in Figure 3.9 different reconstructed digits of MNIST dataset with the corresponding error. This visual representation helps understanding whether an error can be considered acceptable or not.

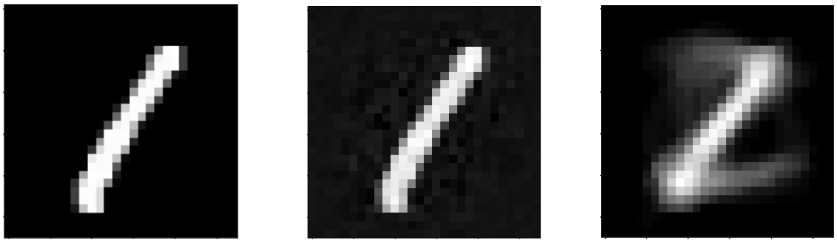

Figure 3.9: From left to right : error = 0, error = 0.02, error = 0.21.

For a more robust analysis let us now move on timeseries dataset.
Figure 3.10 analyzes the DL performance on FordA dataset. In detail, it shows the error's behaviour with respect to the iterations for $s = 30, 40$ and 50 and different compression percentages. As before, fixed the parameters s and n, we expect a descending behaviour the error. Furthermore, we expect an higher overall error compared to the previous case, as Ford-A is a more complex dataset. However, we also expect shorter computational times due to its significantly smaller size.

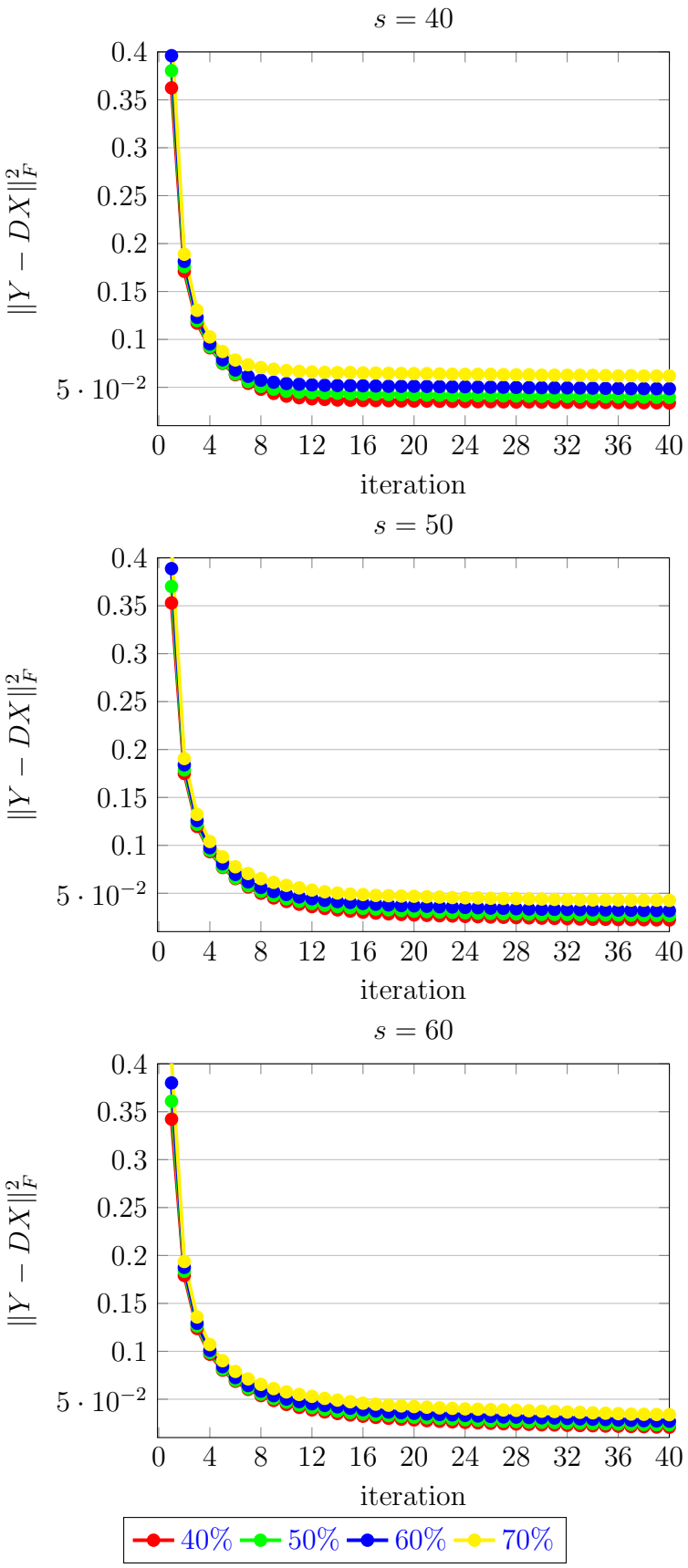

Figure 3.10: DL on FordA dataset: error w.r.t n. iterations ($s = 40, 50, 60$).

As expected the error shows a descending trend as the number of iterations increases. In particular, after 40 iterations the algorithm reaches an error between 0.02 and 0.048, similarly to those obtained in the previous case, and requires around 20 minutes, which is significantly less than the 20 hours of the previous case.

Figures 3.11 and 3.12 show the relation between the error and, respectively, the compression percentage and the number of atoms n. As before, we expect that, when the number of atoms n is held constant, the error decrease as the parameter s increases, as signals are represented by a larger number of atoms. Similarly, we expect that increasing n or, equivalently, the compression percentages leads to better results.

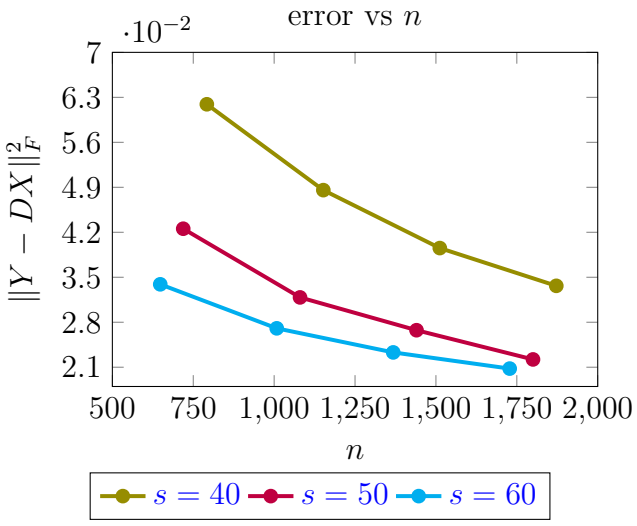

Figure 3.11: DL on Ford-A : error w.r.t the number of atoms n

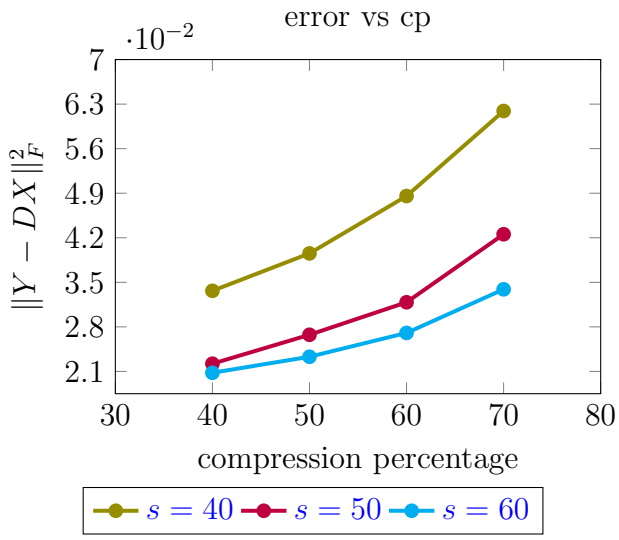

Figure 3.12: DL on Ford-A : error w.r.t the compression percentage

From Figure 3.12 we can see that, in the same context as above, even with a compression of 70 % we can reach an error equal to 0.034. This proves that DL is an excellent

candidate for compression tasks also for timeseries datasets. Nevertheless, as already mentioned, is clear that compressing time series can be more challenging if compared to a dataset such as MNIST, since they represent real-world data with more complex patterns.

Let us now step back and revisit the context of digital twin. We have observed that DL techniques offer an exceptionally valuable compression tool, facilitating the data transfer from physical system to its digital counterpart. Nevertheless, as our ultimate goal is to process and extract valuable insights from the data, when evaluating the quality of DL compression, it becomes more meaningful to compare the information resulting from ML processes which are trained both the original and compressed datasets, rather than focusing exclusively on the error (3.1).

In the following section we will illustrate this concept and explore a couple of digital twin applications, simulating a digit recognition process using the MNIST dataset and employing the timeseries data for anomaly detection.

3.2.2 Analysis of DL compression performances through CNNs.

So far, we did not go into the details about how digital counterpart's structure is. There exists various kinds, but, for this project we used HPC systems, in particular Cineca's Marconi100 cluster [39]. Indeed, for both our tasks, image recognition and anomaly detection, we utilized Convolutional Neural Networks (CNN) that can be easily implemented and parallelized, for instance with Horovod [40], on HPC clusters. This is another advantage of digital twin systems : transferring data to a more powerful machine capable of performing more advanced computations is more convenient than attempting to replicate the same process on the physical counterpart.

Let us start by evaluating MNIST compression through DL by building a CNN specifically designed to detect the various digits present in the dataset and examining its performance on the compressed dataset. For this purpose, we choose the convolutional neural network with two convolutional layers with ReLU activation function, two max-pooling layers and softmax as output function reported in Figure 3.13 as suggested in [41].

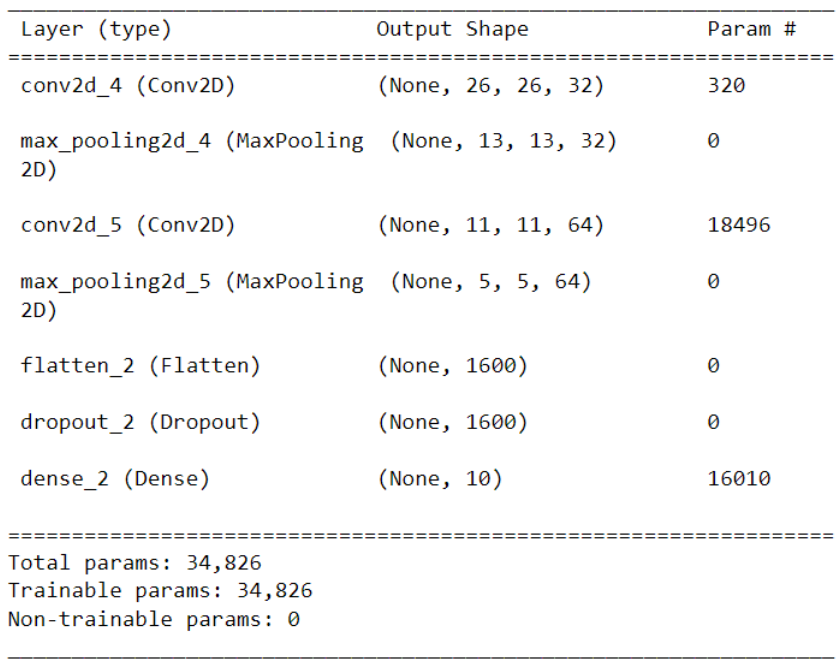

Figure 3.13: Profiler of the CNN used for digit recognition on the MNIST dataset

After training the CNN on both original and compressed datasets, the next step is to asses the CNN's performance on the same test set. In particular, in both this case and the following one, we will not use the original test set but the one obtained by multiplying the dictionary D_{train} learned from the training set with the sparse matrix obtained applying the parallelized OMP-QR on the dictionary D_{train} and the original test set. We used a DL compression with sparsity level s equal to $s = 20, 30, 50$ and different compression percentages. To evaluate the performance of the CNN, we used the *accuracy*, which is defined as the ratio of the number of correct predictions over the total number of predictions. Table 3.2.2 and Figure 3.14 compare the test accuracy achieved by the CNN trained on the original dataset with the ones obtained by training the CNN on the compressed datasets. We expect that increasing the compression rate will lower the accuracy. Our hope is that the accuracy obtained with the compressed dataset remains comparable to the one obtained with the original dataset.

s	n	% compr.	accuracy
/	/	0 %	99.23 %
20	10469	80 %	97.22 %
20	16469	70 %	96.81 %
20	22469	60 %	97.49 %
20	28469	50 %	97.04 %
20	34469	40 %	96.79 %

s	n	% compr	accuracy
/	/	0 %	99.23 %
30	9704	80 %	97.03 %
30	15704	70 %	97.38 %
30	21704	60 %	97.39 %
30	27704	50 %	97.09 %
30	33704	40 %	96.91 %

s	n	% compr	accuracy
/	/	0 %	99.23 %
50	8173	80 %	97.69 %
50	14173	70 %	97.62 %
50	20173	60 %	97.28 %
50	26173	50 %	96.84 %
50	32173	40 %	97.1 %

Table 3.1: Accuracy of different compression settings with $s = 20, 30$ and 50 on MNIST dataset.

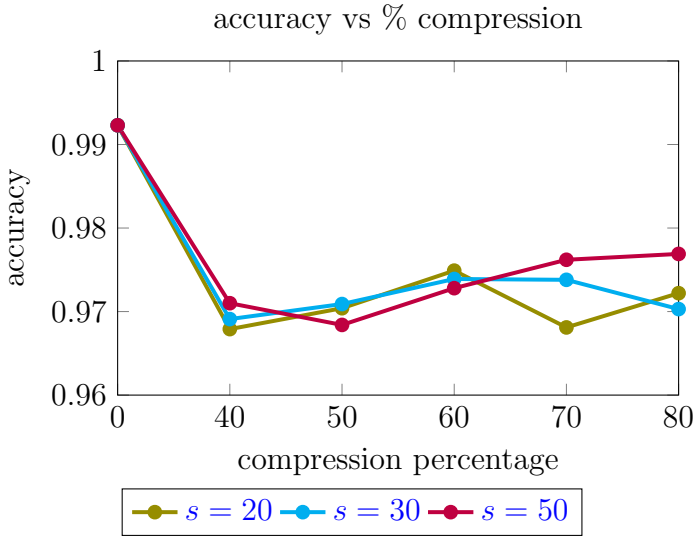

Figure 3.14: Accuracy with respect to the compression percentage

As expected, the accuracy computed on the compressed datasets is lower than the one computed on the original dataset. Despite not matching exactly the original accuracy, we still achieve extremely good results, reaching even an accuracy of 97.7 % with a compression of 80%. Differently from what we expected, we notice that the accuracy remains almost constant as the compression percentages decreases. This is probably due to the redundant nature of the MNIST dataset, which makes it possible to achieve

high accuracy levels even with high levels of compression. A natural question arises regarding the accuracy achieved when the dataset undergoes random compression, such as preserving only 20% of samples chosen randomly. In these scenarios, we find that the randomly compressed dataset can achieve 92% accuracy, which is quite lower than the 97.7% accuracy obtained with our DL compression. In a context with more complex data, we would expect a more significant difference between the accuracy achieved with random compression and that achieved with our technique.

Let us now explore how the accuracy is distributed among the individual classes. Figure 3.15 presents a comparison of the test accuracy of each class obtained from the original dataset, depicted in grey, and the accuracy related to the compressed datasets with $s = 20, 30, 50$ and a compression percentage of 60 %, represented in green,blue and red, respectively.

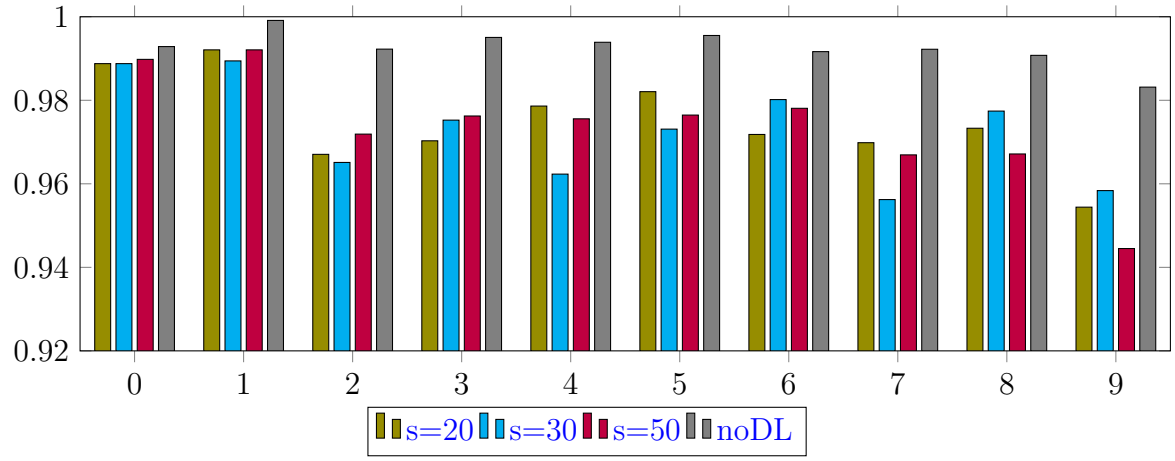

Figure 3.15: Comparison of test accuracy of each MNIST digit between original dataset and its compressed versions.

All classes achieve a good level of accuracy; however, we notice that, while the compression has no significant impact on the recognition of digits 0 and 1, the other digits appear to be more challenging for the CNN to recognize. For example, digit 9 has a recognition accuracy of 94,45 % with $s = 60$ and 60 % of compression, while with the original dataset it can reach 98,3 %.

We will now perform the same analysis on the FordA dataset. We expect less accurate results as a timeseries dataset is less redundant compared to the MNIST dataset. In this examples we need a CNN able to perform anomaly detection. For this task we used the CNN reported in Figure 3.16 with three one-dimensional convolutional layers with ReLU activation function, one global average pooling layer and softmax as output function as suggested in [42].

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 500, 1)]          0

 conv1d (Conv1D)             (None, 500, 64)           256

 batch_normalization (BatchN  (None, 500, 64)          256
 ormalization)

 re_lu (ReLU)                (None, 500, 64)           0

 conv1d_1 (Conv1D)           (None, 500, 64)           12352

 batch_normalization_1 (Batc  (None, 500, 64)          256
 hNormalization)

 re_lu_1 (ReLU)              (None, 500, 64)           0

 conv1d_2 (Conv1D)           (None, 500, 64)           12352

 batch_normalization_2 (Batc  (None, 500, 64)          256
 hNormalization)

 re_lu_2 (ReLU)              (None, 500, 64)           0

 global_average_pooling1d (G  (None, 64)               0
 lobalAveragePooling1D)

 dense (Dense)               (None, 2)                 130

=================================================================
Total params: 25,858
Trainable params: 25,474
Non-trainable params: 384
_________________________________________________________________
```

Figure 3.16: Profiler of the CNN used for anomaly detection on the FordA dataset.

Let us have a look to the comparison of the test accuracy achieved by the CNN trained on the original dataset with the test accuracy obtained by the CNN trained on the compressed dataset. For this analysis, as shown in Table 3.2.2, we considered $s = 30, 40$ and 50 with several compression percentages. The same results are also graphically presented in Figure 3.17.

s	n	% compr	accuracy
/	/	0 %	96.31 %
40	1872	40 %	90.45 %
40	1512	50 %	90.23 %
40	1152	60 %	89.77 %
40	792	70 %	90.19 %

s	n	% compr	accuracy
/	/	0 %	96.31 %
50	1800	40 %	90.89 %
50	1440	50 %	90.72 %
50	1080	60 %	90.41 %
50	719	70 %	90.87 %

s	n	% compr	accuracy
/	/	0 %	96.31 %
60	1728	40 %	90.19 %
60	1368	50 %	89.96 %
60	1008	60 %	90.85 %
60	648	70 %	90.85 %

Table 3.2: Accuracy of different compression settings with $s = 40, 50$ and 60 on Ford-A dataset.

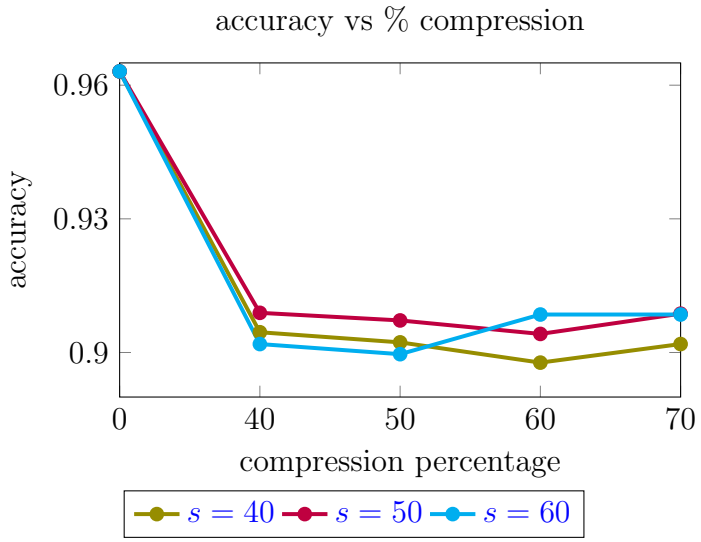

Figure 3.17: Accuracy with respect to the compression percentage

Also in this case we can not reach the original accuracy level. However, we obtained an overall accuracy that almost reaches 91 % even with high levels of compression. These are very promising results considering the real-world nature of the dataset and that a random compression of 60 % of the original dataset would reach an accuracy of only 52 %. This result confirms what was mentioned above: the more complex the dataset becomes, the more essential automatic compression techniques like DL become crucial if compared to random dataset reduction.

Now, let us explore the accuracy of normal (label -1) and problematic (label 1) measurements separately. Figure 3.18 presents a comparison of the test accuracy levels of the two classes obtained from the original dataset, depicted in green, and from the compressed datasets with $s = 40, 50, 60$ and a compression of 60 %, represented in green, blue and red respectively.

The DL compression on this timeseries dataset is highly satisfactory, as a CNN trained on the dataset compressed by 60% can detect problematic measurements with an accuracy of 87.94 %. We recall that the Ford-A dataset is a balanced dataset, with 1846 training samples and 681 test samples for the problematic measurements and 1755 training samples and 639 test sample for the good ones.

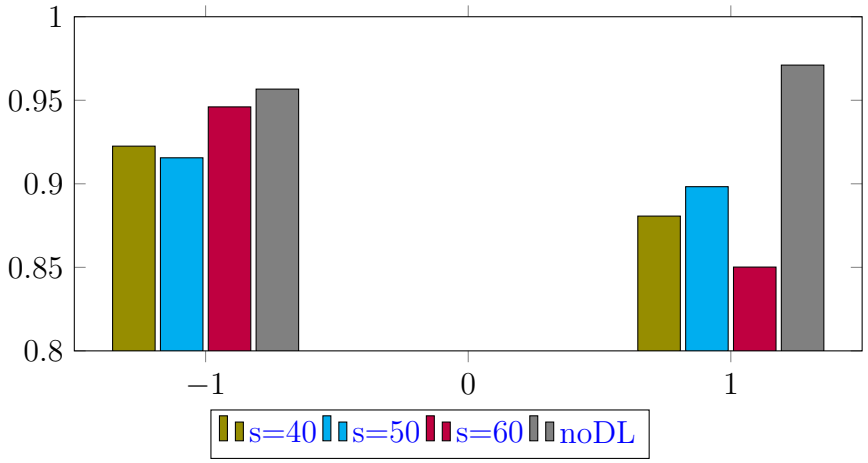

Figure 3.18: Comparison of test accuracy of each class between original dataset and its compressed versions.

We conclude this section with one final example, which, by using everything we have seen so far, demonstrates how Dictionary Learning can be introduced into the Digital Twin context and its actual efficiency.

Example 7.

This example aims to reproduce the workflow presented in Algorithm 8. Let us start by using our "DL_ompqr_parallel_ksvd" algorithm on a large amount of data, as we aim for a rather extensive dictionary to make the model more robust. In this example, we apply the algorithm to the entire training MNIST dataset with $s = 30$ and 60 % of compression. After some time, a new dataset, smaller than the starting one, is collected on the physical counterpart and needs to be transmitted to the digital system. We simulate the new dataset by using a portion of the test MNIST dataset, specifically the first 5000 samples. This time, we do not need to recompute the DL on the new dataset, but only its sparse representation X_1 by means of our version of OMP-QR with respect to the dictionary D previously computed. Therefore, we can transfer only the matrix X_1 to the supercomputer and train the neural network on the new dataset DX_1. The neural network has been tested on other 3000 samples coming from the MNIST test set, getting an accuracy of 92 %. Summing up, the pros of this workflow is that the DL algorithm, which takes long time and more computational resources, is used only occasionally, and in its place, we prefer OMP, in particular OMP-QR, which is lighter and faster. Furthermore, the data transmission is reduced to transferring only the sparse matrix X_1, whose size is at most $3(s \times N)$, instead of $m \times n + 3(s \times N)$. We cannot replicate the same experiment using the Ford-A dataset because its test set, consisting of only 1320 samples with 500 features each, is too small to extract sufficient simulated new data and the corresponding test samples needed to achieve an accurate recognition level while ensuring an overcomplete dictionary.

The numerical experiments presented in this chapter prove that DL technique is an great valuable compression tool, achieving high compression rates of up to 70/80 % while maintaining accuracy levels similar to the ones achieved without compression. Thus, Dictionary Learning offers a powerful tool that can significantly enhance the digital twin framework's capabilities, speeding up data transmission and significantly reducing latency between the physical and digital systems.

Conclusions

The purpose of this thesis was to introduce a new efficient and lightweight compression tool within the Digital Twins framework. In particular, among the available compression techniques, we explored Dictionary Learning (DL), a robust matrix factorization algorithm that, in contrast to deep neural network based solutions, does not require powerful computer resources, making it well-suited for deployment on lightweight devices like IoT sensors. Furthermore, it also has minimal impact on the accuracy of AI models trained on compressed data.

The first chapter introduced the basis of the DL problem, presenting it as a two variable, non-convex, constrained optimization problem which aims to find a matrix $D \in \mathbb{R}^{m \times n}$ called Dictionary and a sparse vector $\mathbf{x} \in \mathbb{R}^n$ in order to obtain a good sparse representation $\mathbf{y} \approx D\mathbf{x}$ for a class of signals $\mathbf{y} \in \mathbb{R}^m$. As we have seen, this goal is achieved by minimizing the error between the signal $\mathbf{y}$ and its sparse representation $D\mathbf{x}$ in the least squares sense. This optimization problem can be addressed with an alternate optimization approach, which, in this context, consisted in splitting the optimization problem into two subproblems: sparse coding step and dictionary update step. For each problem many different techniques and strategies can be used. At the end of the first chapter we have shown that the DL technique that is more suitable for our purposes employes OMP-QR for the sparse coding step and KSVD for dictionary update. Therefore we decided to use such an implementation for the rest of the work. Clarified our DL algorithm, in the second chapter we gave an overview of Convolutional Neural Networks (CNNs) architecture.

The core of this thesis lays in Chapter 3. Besides introducing the Digital Twin context which has been the background of our applications, this chapter mainly presented the numerical results obtained with our "DL_ompqr_parallel_ksvd" algorithm. To this end, we considered two different type of applications: first we focused on images, applying our algorithm on MNIST dataset and then on timeseries, considering the FordA timeseries dataset. The numerical experiments reported in Chapter 3 show how effective the algorithm "DL_ompqr_parallel_ksvd" is. In particular, on very redundant dataset, such as MNIST, the algorithm has an excellent behaviour, managing to compress the dataset up to 80 % while preserving key information and therefore keeping the digit recognition accuracy almost unchanged. Indeed, while the recognition accuracy of the original dataset is around 99 %, the accuracy of the dataset compressed up to 80 % with

DL technique is around 97.5 %. In more complex and less redundant dataset such as Ford-A dataset, the essential information are more difficult to detect and the algorithm is less effective. Indeed, starting from an anomaly detection accuracy of 96 % with the original dataset, with our algorithm we can reach at most an accuracy of 91 % with a compression of 30 %. It is worth noticing that in the latter setting with a random compression we would have obtained an accuracy of 52 %. Due to the low computational cost of the OMP-QR algorithm, this approach allows for on-device data compression, particularly useful with devices like IoT sensors, effectively reducing data exchange between devices while retaining the most crucial information. The results presented in this thesis would not have been possible without the HPC resources provided by Cineca that gave a substantial acceleration to the whole workflow almost dubling the speed of the OMP-QR algorithm and the training of the Neural Networks through parallelization techniques. In conclusion, we can state that the compression algorithm "DL_ompqr_parallel_ksvd" effectively reduces the dataset's memory demand, resulting in faster data transmission and reduced latency between distinct systems. It achieves this while demanding minimal computational resources and maintaining nearly identical classification performance to the original dataset. Such a compression tool can be extremely powerful and helpful whenever there is a consistent exchange of data, such as within a Digital Twin context. Further investigations on different datasets can be carried out to prove the effectiveness of the DL on a wider range of contexts. For instance, a new interesting scenario could be considering larger and less redundant dataset, such as ImageNet [43], or sparse matrices, an option that would be of particular interest in the context of some Digital Twins where the data are collected in LiDAR format.

Appendix A

Description of the databases

The theories and models presented in this thesis have been tested mainly on two different types of data : images and timeseries. Here follows a detailed description of each dataset that was used.

- **Images**

  1. MNIST dataset :
     the MNIST database (Modified National Institute of Standards and Technology database) [17] is a large database made of 28×28 images representing handwritten digits from 0 to 9 made by high school students and employees of the United States Census Bureau. It contains 60,000 training images and 10,000 testing images. Figure A.1 shows some sample images from MNIST test dataset.

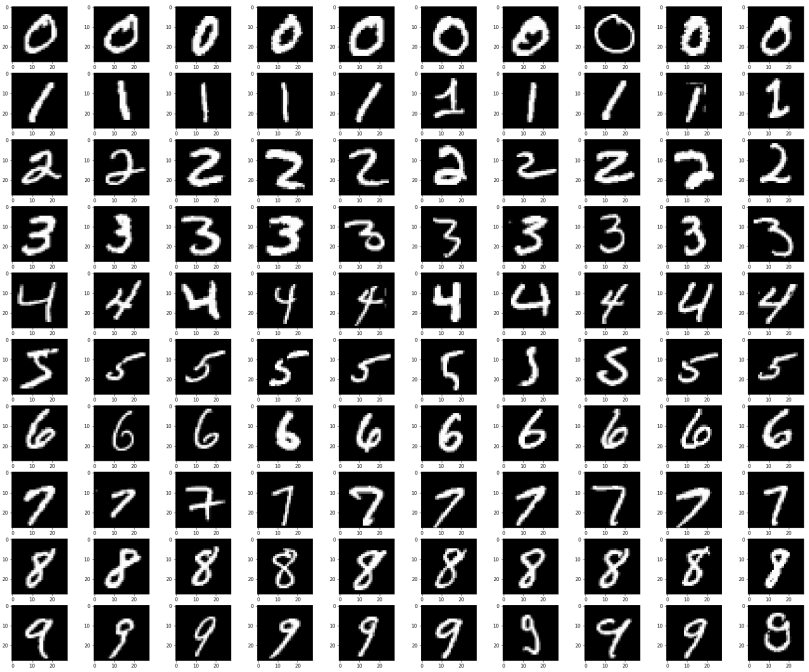

Figure A.1: Some sample images from MNIST dataset.

The dataset can be easily imported from Keras with the Python command

`tensorflow.keras.datasets.mnist.load_data()`.

- **Timeseries**

1. FordA :
   the FordA [38] is part of the WCCI 2008 Competition Program and was motivated by an automotive application. This dataset is a balanced timeseries dataset from the UCR archive and it is made of 3601 training instances and another 1320 testing instances. The training set provided also the classification labels, where +1 indicates the presence of a specific symptom and -1 indicates that the symptom does not exist. The problem is a binary classification task. The full description of this dataset can be found at [44].

Appendix B

Introduction to High Performance Computing clusters

The goal of this appendix is to provide a brief overview of High Performance Computing clusters, with specific emphasis on those employed in this thesis.

A supercomputer, also called *computing cluster*, is a group of hundreds of servers, called *nodes*, connected together to create a single, more powerful machine. As we can imagine, they are becoming more and more interesting lately because of two of their main properties: their ability to deal with large amount of data and their speed in returning results. There exist different types of computing clusters: we will focus on High Performance Computing (HPC) ones. Their strength lies in their ability to perform high-speed parallel processing of complex computations on multiple servers. This could be crucial in several applications such as Urgent Computing. We talk about Urgent Computing when supercomputers are involved for emergency computations that would have taken too long otherwise. Cineca has used Urgent Computing techniques in the Exscalate4CoV project ([45],[46],[47]), cooperating with traditional research to shorten the time for the development of drugs against Coronavirus.

B.1 General structure of a supercomputer

These are the main steps and components of a supercomputer:

1. **Login nodes** : first of all the user needs to interface with the machine. For this reason there exist the login nodes which let the user work with their files and data and interact with the Scheduler.

2. **Scheduler and Master Node** : the task of the Scheduler is to assign compute nodes with certain characteristics required by the user on a previously written *jobscript*, while the task of the Master node is running the Scheduler and organizing how the computational resources are distributed among users. These two components represent a fundamental part of an HPC machine since they allow to

optimize the use of the cluster satisfying hundreds or thousands of users which require different resources at different times.

3. **Compute nodes**: nodes made of CPUs, memory and GPUs which actually runs the user's algorithm. We will focus on their architecture in the next section.

4. **Parallel Filesystem**: it is designed to perform I/O operations in parallel, allowing simultaneous access to the filesystem from multiple nodes.

Figure B.1 schematically sums up the architecture of an HPC cluster.

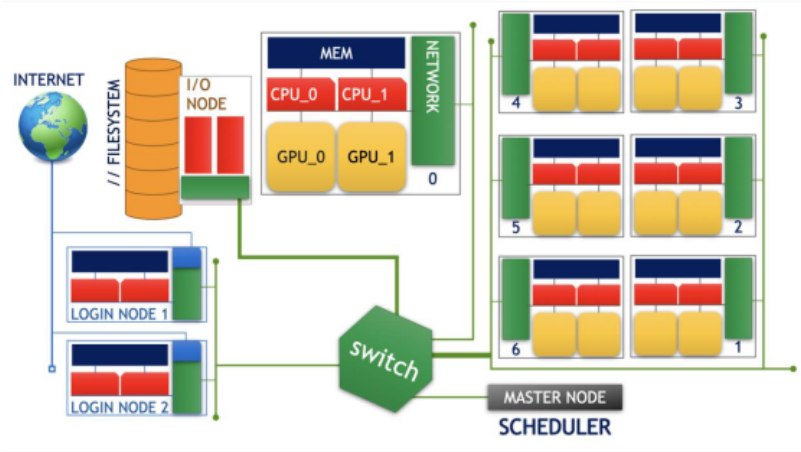

Figure B.1: HPC cluster architecture [48]

B.2 What is a *jobscript*

The user can communicate with the cluster and runs its own algorithms on some compute nodes by submitting a text-file called *jobscript* to the Scheduler.
The jobscript is a text file that defines the resources (**SLURM directives** in MARCONI100 [49]), the software needed, the variables and the actual algorithm (**variables environment** and **execution line**)
An example of a jobscript is reported in the next page.

```
#!/bin/bash

#SBATCH --job-name=myname
#SBATCH --output=job.out
#SBATCH --error=job.err
#SBATCH --account=account_name
#SBATCH --mail-type=ALL
#SBATCH --mail-user=user@email.com
#SBATCH --time=00:30:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=36
#SBATCH --gres=gpu:4
#SBATCH --cpus-per-task=2
#SBATCH --mem=10GB
#SBATCH --partition=partition_name

echo "I'm working on MARCONI100!"
```

B.3 Architecture of a compute node

These are the nodes dedicated to computations.
They are made of 2 or more CPUs, a great amount of memory, one or more GPUs.
We now give some details of the two Processing Units :

- **CPU (Central Processing Units)** : its primary tasks are to perform calculations and manage the data necessary to run the applications. Every CPU has multiple *cores* (from 32 to 56) which are the actual chips that performs calculations and it is connected to the server memory and to the I/O devices.
- **GPU (Graphics Processing Units)** : its main characteristic is having a very high number of cores (hundreds) and therefore being able to exploit an incredibly high level of parallelism, at a price of low memory. For these reasons it allows to deliver an higher number of FLOPS (Floating Point Operations per Second), but it needs to pass trough to the GPU to store data, which requires a significant transferring time.

B.4 Parallel Computing

At the begging of this appendix we stated that the strength of an HPC cluster lies in its "parallel computing ability". In this section we will explain better what "Parallel Computing" means and how it can be applied.
Parallel Computing is the simultaneous use of multiple compute resources to solve a computational problem. The problem is indeed broken into discrete parts whose instructions are executed simultaneously, with the advantage of being time and cost saving.
In a cluster, Parallelism can be expressed in various forms:

- **Shared memory parallelism** (threading): all the instances share the same memory that the initial process allocated. Therefore the communication between the instances is fast, but sharing the memory could create overlapping problems.
- **Distributed memory parallelism** (task): each instance has its own memory. There is no more the overlapping problem but there is a performance loss caused by a slow communication.
- **Hybrid parallelism**: it combines the previous two methods and it is the most used form in real applications. For example, in an HPC cluster could be convenient setting a threading parallelism within each node while a task parallelism between nodes.

To measure the goodness of the parallelisation effort we need to compare the time of serial execution with the time of parallel execution. We can do it by the **Speedup** ($S := \frac{T_{serial}}{T_{parallel}}$) or **Efficiency** ($E := \frac{S}{p}$, where p is the number of cores).
Lastly, we have to take into account also the dimensions of the problem, because, clearly, the bigger the problem dimensions, the higher the computing time. This issue can be overcome by distributing the work across more processors. Here comes into play the definition of **Scalability** which is the ability of a parallel system to proportionate increase in parallel speedup with the addition of more processors. Figure B.4 shows the behaviour of Speedup, on the top, and Efficiency, on the bottom, with respect to the number of processors and for different problem sizes. We can see that as the dimension of the problem grows, speed-up will grow as well, if enough parallel processors are added.

Actually, there exist two different types of Scalability : we have Strong Scalability when, fixed the problem dimension, the efficiency does not decrease while increasing the number of cores. We have Weakly Scalability when the efficiency does not decrease when problem dimension per processor is kept almost unchanged.

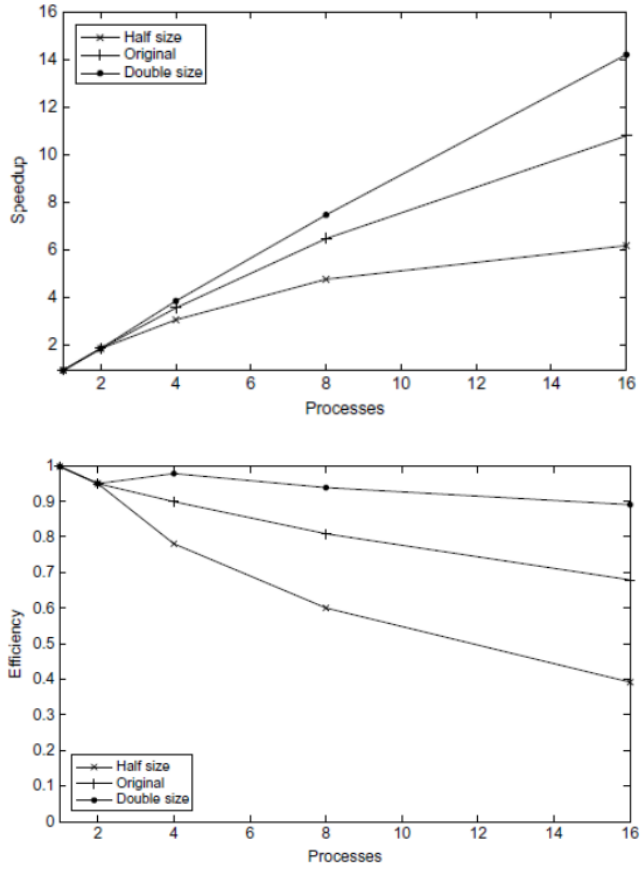

Figure B.2: Speed up and Efficiency with respect to number of processors and problem sizes [50].

B.5 HPC in Cineca

Cineca is a non-profit consortium whose members include the Italian Ministry of Education, the Italian Ministry of Universities and Research, 69 Italian universities and 27 national public research centres, for a total of 112 members. As of today, it is the largest Italian supercomputing centre and one of the most important worldwide. It supports the European and Italian scientific community by means of high performance computing, develops management systems for the university administrations, designs and develops information systems for businesses, health care organizations and public administration [2].

For the purpose of this thesis we employed the following Cineca clusters:

- **Marconi100 :** Marconi100 is an IBM system installed in 2020 and dismissed in July 2023. It was composed by 980 nodes, each one made of two IBM POWER9 AC922 CPUs running at 3.1GHz featuring 16 cores each and 4 threads per core. This configuration added up to a total of 32 cores and 128 threads per node. Additionally, each node was equipped with 256 GB of memory and 4 paired GPUs connected by a technology developed by NVIDIA called NVLink 2.0. Figure B.5

illustrates the architecture of a Marconi100 node. In 2020 was in the top 10 of the TOP500, the list of the 500 most powerful supercomputer in the world [3].

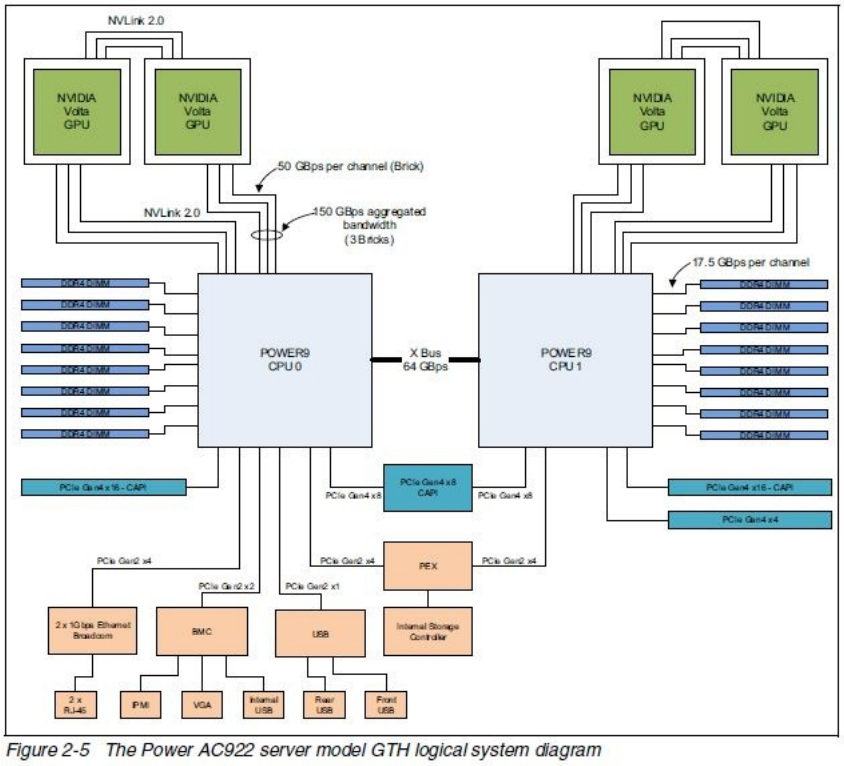

Figure B.3: architecture of a MARCONI100 node [16]

- **Galileo100 :** Galileo100 was installed in August 2021 as a replacement for Galileo. It serves as Cineca Tier-1 infrastructure dedicated to scientific research and is co-funded by the European ICEI (Interactive Computing e-Infrastructure) project. This infrastructure was engineered by DELL and consists of 528 computing nodes. Each node is equipped with two Intel Cascade Lake 8269 processors running at 2.4GHz, with each processor having 24 cores. Specifically, these 528 nodes are distributed in the following manner: 348 are standard or "thin" nodes, while 180, called "fat" nodes, are designated as data processing.

Furthermore, Cineca hosts and manages also **Leonardo**, a new supercomputer that has been recently installed in the new data center located in the Technopole of Bologna. It is one of the three pre-exascale systems announced by EuroHPC Joint Undertaking [51]. The main goal of Leonardo is to support European academic and industrial researchers to develop applications able to tackle the most urgent challenges affecting our times, as climate change, pandemics, weather forecasting and prediction of extreme events, formulation of new materials. With its 4992 computing nodes providing 250 PetaFLOPS in computational power, 2800 TeraBytes of RAM and 110 PetaBytes of storage, Leonardo reached the 4th position on the November 2022 TOP500 list.

Bibliography

[1] Mohsen Attaran and Bilge Gokhan Celik. Digital twin: Benefits, use cases, challenges, and opportunities. *Decision Analytics Journal*, 6:100165, 2023.

[2] CINECA. Cineca-organization. https://www.cineca.it/en/about-us/organization.

[3] the list TOP500. List of top-500 supercomputers. https://www.top500.org/.

[4] D. Brandoni. *Dottorato di ricerca in Matematica. Tensor-Train decomposition for image classification problems.* 2022.

[5] M. Aharon, M. Elad and A. Bruckstein. *On the uniqueness of overcomplete dictionaries, a practical way to retrieve them. Linear Algebra Appl. 416(1), 48–67 (2006).*

[6] B. Dumitrescu and P. Irofti. *Dictionary Learning Algorithms and Applications.* Springer, 2018.

[7] Wikipedia. Haar wavelet. https://en.wikipedia.org/wiki/Haar_wavelet.

[8] Dave Marshall. The discrete cosine transform (dct). https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node231.html.

[9] J. A. Tropp. *Greed is good: Algorithmic results for sparse approximation," IEEE Trans. Inf. Theory, vol. 50, pp. 2231–2242, Oct. 2004.*

[10] S. A. Chen, S. Billings and W. Luo. *Orthogonal least squares methods and their application to nonlinear system identification," Intl. J. Contr., vol. 50, no. 5, pp. 1873–1896, 1989.*

[11] M.G. Sturm, B.L. Christensen. *Comparison of orthogonal matching pursuit implementations, in Proceedings of European Signal Processing Conference (EUSIPCO), Bucharest (2012), pp. 220–224.*

[12] M. Aharon, M. Elad and A. Bruckstein. *K-SVD: an algorithm for designing overcomplete dictionaries for sparse representation. IEEE Trans. Signal Process. 54(11), 4311–4322 (2006).*

[13] M. Rubinstein, R. Zibulevsky and M. Elad. *Efficient implementation of the K-SVD algorithm using batch orthogonal matching pursuit. Technical Report CS-2008-08, Technion University,Haifa.* 2008.

[14] B. Dumitrescu and P. Irofti. dictlearn. https://pypi.org/project/dictlearn/#description.

[15] Python.org. multiprocessing — process-based parallelism. https://docs.python.org/3/library/multiprocessing.html.

[16] CINECA. Ug3.2: Marconi100 userguide. https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.2%3A+MARCONI100+UserGuide.

[17] Kaggle. Mnist dataset. https://www.kaggle.com/datasets/hojjatk/mnist-dataset.

[18] G.K. Wallace. *"The JPEG picture compression standard, " IEEE Transaction on Consumer Elecronics, Multimedia Engineering, Digital Equipment Corporation, Maynard, Massachussetts. Vol. 38, No. 1, Feb. 1992.*

[19] Google cloud. Artificial intelligence (AI) vs. machine learning (ML). https://cloud.google.com/learn/artificial-intelligence-vs-machine-learning.

[20] Edward Hance Buchanan, Bruce G; Shortliffe. *Rule-based expert systems : the MYCIN experiments of the Stanford Heuristic Programming Project.* Reading, Mass. : Addison-Wesley, 1984.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[22] OpenAI. Gpt-4. https://openai.com/gpt-4.

[23] N. Montobbio. *Dottorato di ricerca in Matematica. A metric model of the visual cortex.* 2019.

[24] W. S. McCulloch and W. Pitts. *A logical calculus of ideas immanent in nervous activity.* 1943.

[25] Balázs Csanád Csáji. *MSc thesis. Approximation with Artificial Neural Networks.* 2001.

[26] K. Hornik. *Approximation Capabilities of Muitilayer Feedforward Networks.* Neural Networks, vol. 4, 1991. https://web.njit.edu/~usman/courses/cs677_spring21/hornik-nn-1991.pdf.

[27] J. Casper P. LeGresley M. Patwary† V. Korthikanti D.Vainbrand P.Kashinkunti

J.Bernauer† B. Catanzaro† A. Phanishayee M. Zaharia D. Narayanan, M. Shoeybi. *Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM*. NVIDIA, Stanford University, Microsoft Research, 1991. https://arxiv.org/pdf/2104.04473.pdf.

[28] F.Bertoni. *Dottorato di ricerca in Matematica. LIE SYMMETRIES IN CORTICAL INSPIRED CNNS*. 2022.

[29] Mathworks. What are convolutional neural networks? — introduction to deep learning. https://it.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765html.

[30] T.S. Cohen and M. Welling. *Group equivariant convolutional networks.* 2016.

[31] Wikipedia. Cross-entropy. https://en.wikipedia.org/wiki/Cross-entropy.

[32] M.Porcelli. Power point presentation of the class in matrix and tensor techniques for data science, part 2. 2021/2022.

[33] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[34] Jinkang Guo Zhihan Lv. *Application of Digital Twins in multiple fields*. Multimed Tools Appl 81, 26941–26967, 2022. https://doi.org/10.1007/s11042-022-12536-5.

[35] Maggie Mashaly. *Connecting the Twins: A Review on Digital Twin Technology its Networking Requirements, Procedia Computer Science*. Procedia Computer Science, Volume 184, Pages 299-305, 2021. https://doi.org/10.1016/j.procs.2021.03.039.

[36] Di Santi C. Molan M. et al. Borghesi, A. M100 exadata: a data collection campaign on the cineca's marconi100 tier-0 supercomputer. https://doi.org/10.1038/s41597-023-02174-3.

[37] IoTwins. Big data platform for optimized and replicable industrial and facility management models. https://www.iotwins.eu/.

[38] A. Bagnall. Forda dataset. http://www.timeseriesclassification.com/description.php?Dataset=FordA.

[39] CINECA. Marconi100, the new accelerated system. https://www.hpc.cineca.it/hardware/marconi100.

[40] LF AI & Data Foundation. Horovod official website. https://horovod.ai/.

[41] fchollet. Simple mnist convnet. https://keras.io/examples/vision/mnist_convnet/.

[42] hfawaz. Timeseries classification from scratch. https://keras.io/examples/timeseries/timeseries_classification_from_scratch/.

[43] Princeton University Stanford Vision Lab, Stanford University. Imagenet webpage. https://www.image-net.org/.

[44] Joerg D. Wichard. Classification of ford motor data. http://www.j-wichard.de/publications/FordPaper.pdf.

[45] CINECA. Dall'urgent computing all'ai per la sanità del futuro. https://www.cineca.it/news/dallurgent-computing-allai-la-sanita-del-futuro.

[46] CINECA. Exscalate4cov progetto di riferimento in europa per il coronavirus. https://www.cineca.it/news/exscalate4cov-progetto-di-riferimento-europa-il-coronavirus.

[47] CINECA. Exscalate4cov - un caso di urgent computing. https://www.youtube.com/watch?v=OLf3frVgQl4.

[48] Giacomo Guiduzzi. Development of ai benchmarks to monitor the performances of a supercomputer.

[49] SLURM. Sbatch commands on slurm. https://slurm.schedmd.com/sbatch.html.

[50] P.Dagna. Introduction to parallel computing. https://learn.cineca.it/pluginfile.php/8669/mod_resource/content/1/01-Introduction_to_Parallel_Computing.pdf.

[51] EuroHPC Joint Undertaking. The european high performance computing joint undertaking (eurohpc ju). https://eurohpc-ju.europa.eu/index_en.