MASTER DEGREE THESIS

ICT FOR SMART SOCIETIES

Deep Learning Methods for Compressed Sensing Image Reconstruction

---

POLITECNICO DI TORINO

Supervisor:
Enrico MAGLI

Co-supervisors:
Diego VALSESIA

Author:
Gianmarco DRAGONETTI
245750

JULY, 2020

A thesis submitted in fulfillment of the requirements for the
ICT FOR SMART SOCIETIES

written in

Image Processing and Learning Group
@ Politecnico di Torino,
Department of Electronics and Telecommunications
@ Politecnico di Torino

Abstract

Deep Learning Methods for
Compressive Sensing Image Reconstruction

Deep learning (DL) has become, in the last years, the principal way to process images in all the fields of image elaboration and the same goes for Compressive Sensing (CS), the new technique that is making its way in the world of image processing. Compressive Sensing is a full of potential technique, above all for what concerns the acquisition of images, in the case of MRI for example, or the transmission of very big images, e.g. the transmission of the images from satellites. In fact, CS, allowing, together with the single-pixel camera, to directly acquire very compressed data, avoid a waste of time and computational resources to acquire and compress images. Traditional compression techniques are not universal techniques and they suit up for the data, understanding each time how to compress it in the better way. CS, instead, provides a universal way for data acquisition that will speed up these processes and will allows to better and more easily transmit these data too. Unfortunately, CS is not so good in reconstructing the acquired signals and this is its biggest problem. In fact, the iterative classical methods, like minimization of $l1 - norm$, could also take days to exactly reconstruct an image and all the gain of the acquisition and transmission phase is lost. DL aims at solving this problem and, even if the accuracy could not be of 100% of course, the latest results show its great potential. In this thesis, we have evaluated the efficiency of these DL based methods for CS reconstruction and developed an algorithm considering various block size (an important parameter in image processing with NN), fixed and trained sensing matrices and various CS ratios. The results obtained far exceed the state of the art in terms of accuracy and PSNR and remain comparable in terms of elaboration time. Moreover, the algorithm developed works very well with every kind of image.

KEYWORDS **Deep learning**, **Compressed Sensing**, **Neural Networks**, **Image processing**, **PSNR**

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Deep Learning (DL) has emerged, in the last years, as the best way to approach image processing problems that until few years ago were not even imaginable to solve in short time. On the other side, Compressed Sensing (CS) it is starting to be increasingly studied and studied in depth for the potential it offers. The application of DL to CS goes very well with maybe the biggest problem in CS, the reconstruction phase. This phase, in fact, could take days before getting the results and nowadays time is a very important resources, as well as memory, computing power and bandwidth. The combination of these two methods aims at solving all this problems, overcoming the classical iterative and optimization methods used to reconstruct CS signals. The always better computing power, together with the progresses made with GPUs have make the realization of DL based methods to CS reconstruction possible. The field of applications are many, starting from a faster MRI to an easier transmission of satellite images to a simple better usage of resources in general. Compression techniques, in fact, acquire lots of data just to suit up the algorithm to the data and then throw a very big part of them. With CS it is possible to determine a universal way of acquisition, independent from data.

In order to infer the behaviour of a DL based method on a CS measurement, we have applied different settings and conducted different experiments. The algorithm developed is actually an autoencoder composed by two main parts, a CS sub-net (encoding and initial reconstruction) and a Deep Reconstruction Network (decoding). The all works has been developed in Python, principally thanks to the Keras library. First of all with some preproccesing, data augmentation and cropping, we have prepared our dataset

for the experiments. Then we have trained and tested the algorithm with trained and fixed matrices and different block sizes and CS Ratios. To quantify the results PSNR has been chosen as principal metric so the focus was not on the real visual structure of the image. The results are then compared to the state of the art. Finally, the results obtained in this work are, in much part, better, or at least equal, than the state of the art with great results obtained for certain values of CS Ratio.

1.2 Thesis outline

The rest of the thesis is organized as follows:
Chapter 2 presents a brief historical overview on image processing and a review of the related literature that exploits the application of Deep Learning based methods to CS.

Chapter 3 reviews the key theoretical concepts behind the methods, approaches and models we used to perform experiments are based and that are useful to better understand the work developed in this thesis. In particular the focus is on Compressed Sening and Deep Learning, the main topics of this work, and Python and the tools used to develop this work. The overview on CS and DL starts from the very beginning, trying to easily explain what they are and what they are used for, and arrives to the description of the methods that are then actually used in this thesis.

In Chapter 4 the focus is on the framework structure. In particular, are described all the components of the network, with a particular attention to the two main parts, i.e. the CS sub-net and the Deep Reconstruction Network, and also to the pre and post processing under which data had to go.

In Chapter 5 the numerical results are discuss and assessed, giving a detailed description of the settings and the dataset used to test the network. Finally the performances are evaluated and a comparison with related work is made.

Chapter 6 is the conclusive section of the work, aiming at summarize the motivation and the algorithm developed, giving a look to a brief overviw on the obtained results.

Chapter 2

History of image processing and literature research

This chapter provides, in the first paragraph, a brief review about the history of image processing while, in the second part, the description of the most recent studies in the field of Compressive Sensing applied to Image Reconstruction and, in particular, all the studies taken into account for this work and that are focused on the application of deep learning methods to speed up the reconstruction phase.

2.1 History of image processing

> *"In the history of the image, the transition from analogue to digital creates a break which in its principle is equivalent to the atomic weapon in the history of armaments or the genetic manipulation of biology. From the access point to the immaterial, the computerized image itself becomes immaterial, quantified information, algorithm, matrix of numbers that can be modified at will and indefinitely through a calculation operation. Then what catches the view is nothing more than a logical-mathematical model, provisionally stabilized."*
> *(Régis Debray)* [2]

In his history, the man has always tried to capture the world around him and managed to when, in the early years of 1800, was invented the photography. Since then, the photography has obviously changed a lot, and the "earthquake" in its history came with the digitalization. In 1958 Russell Kirsch, for the first time ever, digitalized a photo 2.1 and from then digitalization began. The first totally digital camera was built in 1977, but the quality of the photo

FIGURE 2.1: The first photo to be digitalized

wasn't yet as high as the analogue one, so, before the 90s, the digital photography didn't have a great success because of the poor technology. When the quality of the photo became good enough, the digital begin to become the standard. The digitalization has allowed a much easier transmission of photos and another important factor was the possibility to get instantaneous pictures. Along with the advent of the digital technology, the image processing techniques begun to be studied and developed. Actually, another great advantage of the digital photo is that it can be manipulated and transformed very easily. The image processing, as well as the digital photo in general, became of common use in the first years of 2000, while remaining a prerogative of military industry, particularly in the astronomic field, and university and medical research until then. The first image processing technique have been developed in the early 1920s when some coding techniques were applied to transmit a television signal from London to New York and then, thanks to the space race and, once again, to the improvement in the technology, image processing knew a tremendous growth. A milestone in image processing history was the introduction of a new way to deal with pixels, i.e. the concept of gradient. Until 2002, in fact, images were elaborated pixel-per-pixel, but, with the introduction of the gradient, the attention became to be focused on the difference between pixels. Other examples of tools used in image processing are, for example, the Fourier Transform, the histogram analysis, thresholding and finally the spatial operators like kernel convolution and so, image filtering (e.g. mean filter, median filter, blur filter). Besides these techniques, that are focused on denoising or, principally, aim at improving the

image quality, also the compression techniques should be considered, being storage and transmission two main problems nowadays. Image processing continues to evolve more and more, and its new frontiers are Deep Learning and Compressive Sensing. The first to improve image quality, perform classification, feature extraction and pattern recognition and the second to allow a better manipulation of the images, avoiding a waste of data and improving the speed of acquisition in fields like MRI and allowing a better and more efficient transmission. These two fields are, indeed, the ones around which this work revolves and will be better explained in chapter 3.

2.2 Deep learning methods for Compressed Sensing in literature

We generally group existing *CS reconstruction methods* of images into two categories:

- Traditional optimization-based CS methods;
- Recent network-based CS methods

This thesis work focused principally on the latter ones to then compare them to the work developed in this thesis. These methods, in fact, are the ones that have been taken more into consideration in recent times.

More in detail, some algorithms for CS image reconstruction based on deep learning are:

- ISTA-Net and ISTA-Net$^+$;
- Scalable Convolutional Neural Network for Image Compressed Sensing;
- Solving linear inverse problems with GAN priors:;

2.2.1 ISTA-Net and ISTA-Net$^+$

ISTA-Net and its enhanced version ISTA-Net$^+$ are particularly suited for natural images and they are the first algorithms analyzed. Authors in [9] want to map a classic Iterative Shrinkage Thresholding Algorithm update steps to a deep network architecture. To do this they developed a strategy to solve the

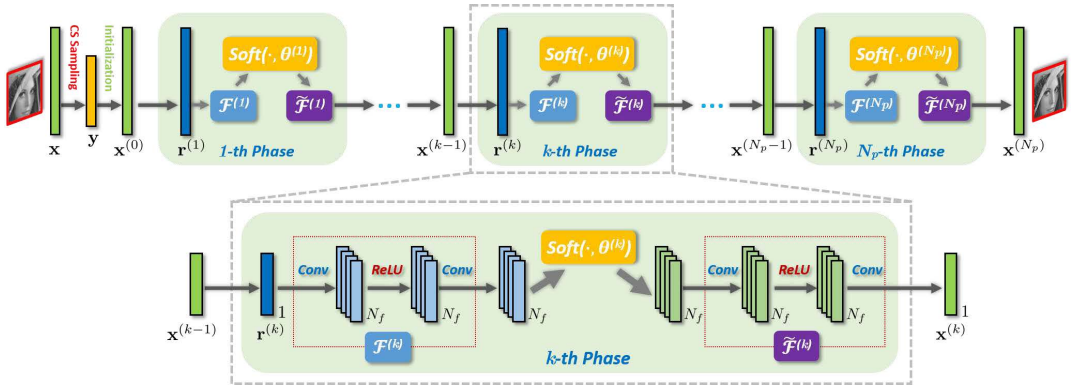


FIGURE 2.2: ISTA-Net Framework [9]

proximal mapping associated with the sparsity-inducing regularizer using non-linear transforms.

ISTA-NET framework consists of mapping each classic ISTA update step into a deep network architecture in which there is a fixed number of phases that correspond to iteration in the traditioanl algorithm. [1] use a general non-linear transform to sparsify the images and, in particular, in the paper taken into account authors used a combination of two convolutional operators separated by a ReLU. Furthermore all the parameters in this algorithm are learned end-to-end, rather than being hand-crafted.

So instead of two equations in the update steps, now we can find two modules. The first called $r^{(k)}$ corresponds to the evaluation of reconstructed signal at step k.

$$r^{(k)} = x^{(k-1)} - \rho^{(k)}\Phi^{(T)}(\Phi x^{(k-1)} - y). \tag{2.1}$$

The only, but big, difference with traditional ISTA is that now ρ can vary across iterations instead of being fixed. The second module is the $x^{(k)}$ module. Also in this case, the module is a particular case of proximal mapping associated to the non-linear transform $\mathcal{F}(\cdot)$, based on the assumptions that each element $x^{(k)} - r^{(k)}$ has an independent distribution with common zero mean and variance σ^2.

In this algorithm all the parameters are learned as NN parameters and their initialization is made using directly a linear mapping insted of using random values as in most of the proposed literature. Hence, given any input CS measurement y, its corresponding ISTA-Net initialization $x(0)$ is computed as: $x(0) = Q_{\text{inity}}$, with Q_{inity} equal to linear mapping matrix. The loss function is

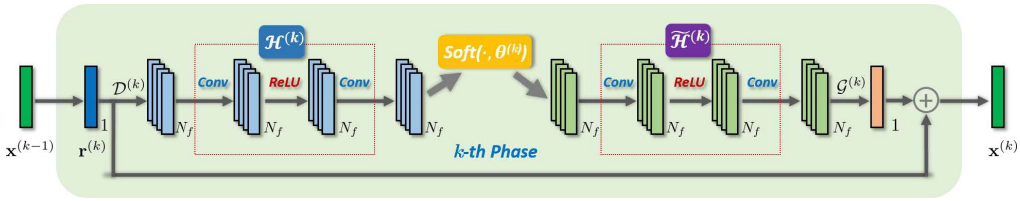


FIGURE 2.3: ISTA-Net+ Framework [9]

defined as:

$$\mathcal{L}_{\text{total}}(\Theta) = \mathcal{L}_{\text{discrepancy}} + \mathcal{L}_{\text{constraint}} \tag{2.2}$$

The optimization in ISTA-Net+ is due to the fact that the natural images are more compressible. The substantial difference with ISTA-Net is in module $x^{(k)}$ where, instead of 2.1, we have:

$$x^{(k)} = r^{(k)} + G(\tilde{H}(soft(H(D(r^{(k)})), \theta))) \tag{2.3}$$

In Figure 2.2 and Figure 2.3 are shown the ISTA-Net Framework block diagram and the ISTA-Net+ $k-th$ phase in which is possible to notice the changes with respect to the $k-th$ phase of traditional version of the algorithm like the addition of the recursive term r^{k} and the additional filters $\mathcal{D}^{k}$ and $\mathcal{G}^{k}$ to retreive some missing high frequencies.

The results shown in [9] greatly overcome the results obtained from traditional iterative algorithms for every CS Ratio and they perform better also for what concerns the computational time. Furthermore in [9] is highlighted how ISTA-Net+ is very flexible as it's possible to let learnable parameters be shared or unshared among the different phases and it is worthwhile to note that ISTA can be extended to other CS domains, since it doesn't impose a specific structure to the sampling matrix unlike much part of known algorithms.

2.2.2 Solving Linear Inverse Problems Using GAN Priors: An Algorithm With Provable Guarantees

In [7], authors try to explain how to use Generative Adversarial Network (GAN) priors instead of hand-crafted ones for linear inverse problems. Moreover they provide theoretical guarantees on the rate of convergence of the algorithm.

In particular the algorithm is based onto two steps:

- Gradient descent update

- Projection on the span of generator

And it's easy to understand it from Figure 2.4

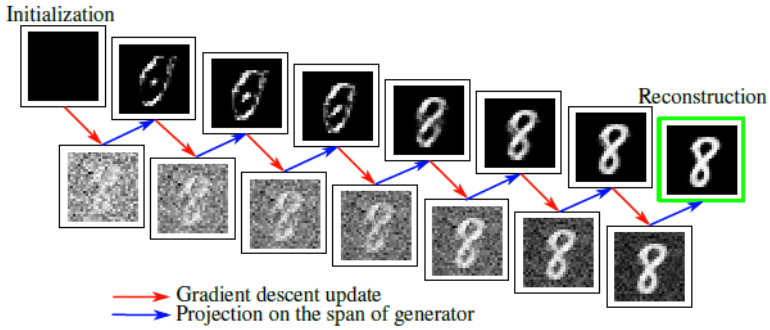


FIGURE 2.4: PGDGAN scheme. Red arrow for GD and blue ones for projection step. [7]

So, they start with a zero vector as initial estimate x_0 and then, with subsequent iterations of classical gradient descent, followed by projection of the output on the span of generator, they finally obtain the desired reconstruction.

The experiment include the train of two GANs, one simpler and trained on the MNIST dataset and one more complex base on CNN (Convolutional Neural Networks) and trained on CelebA dataset.

The algorithm begins considering an ill-posed linear inverse problem expressed in equation 2.4:

$$\hat{x} = \arg\min_{x \in \mathcal{S}} ||y - Ax||^2 \tag{2.4}$$

Generally, to solve this kind of problems a typical gradient descent is used, but in this way is not possible to ensure that the generated image belongs to set $\mathcal{S}$. So, to avoid this, a projection of solution on the range of generator function after each gradient descent update is implemented. In fact, projecting any vector $\sqcap$ in the span of the generator authors are sure to obtain the closest image to $\sqcap$.

The results of [7] are very good, above all in the first experiment, where the improvement with respect to a classical LASSO algorithm, for example, is very high. Figure 2.5

In the second experiments PGD-GAN gives better results than other algorithms, in particular LASSO, again, and CSGM, although they could not be good enough anyway, as it's possible to see form Figure 2.6.

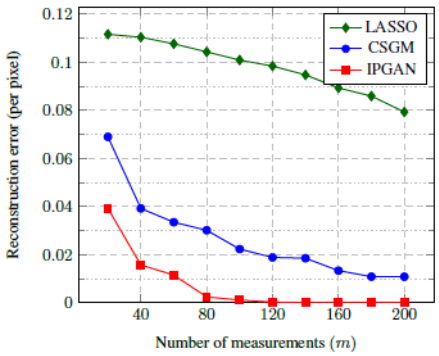


FIGURE 2.5: PGDGAN error against LASSO and CSGM. [7]

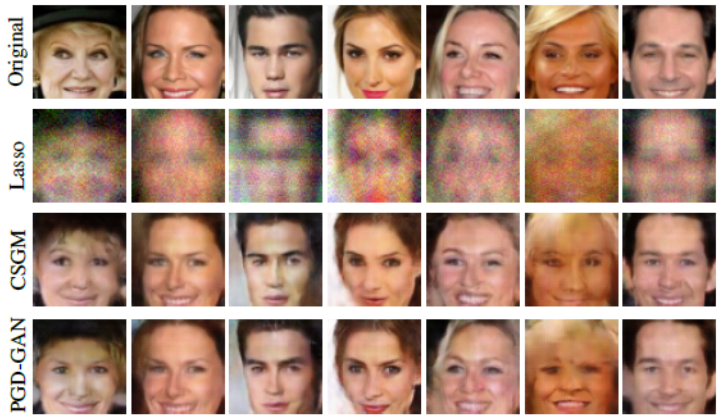


FIGURE 2.6: PGDGAN results on CelebA dataset against LASSO and CSGM. [7]

2.2.3 Scalable Convolutional Neural Network for Image Compressed Sensing

Authors in [8] provide a Scalable Convolutional Neural Network framework to achieve scalable sampling and scalable reconstruction with only one model.

Figure 2.7 shows how SCS-Net is organized. It is block-based and designed as a single sampling matrix plus some Hierarchical Layers (HL). First of all there is a Base Layer (BL) that provides an initial reconstruction of the image directly from the measurements. This initial reconstruction is then used by the Enhancement Layers to improve the quality of the image. The ELs, differently from the BL, firstly generate a reconstruction residual and then, referring to the lower layers results, improve the initial reconstruction quality.

In each layer, the initial reconstruction is followed by a Deep Reconstruction Network (DRN), shown in Figure 2.8. The DRN has a hourglass-shape and

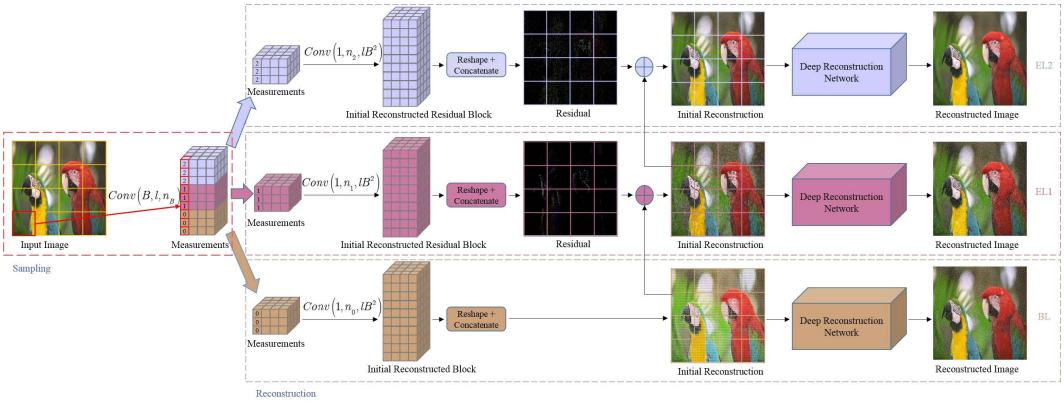


FIGURE 2.7: SCS-Net scheme with two ELs.[8]

includes six kinds of operations, i.e. feature extraction, shrinking, non-linear mapping, expanding, feature aggregation, and skip connection. All these operations are convolution layers with different size filters except skip connection. All the layers are followed by a *ReLU* except for the last. Then is added a skip connection between the initial and final reconstruction.

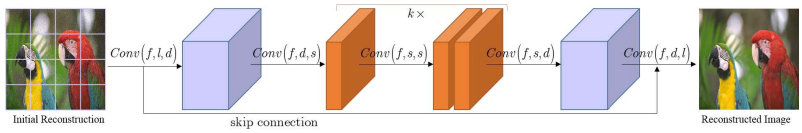


FIGURE 2.8: Structure of Deep Network Reconstruction layer.[8]

Moreover [8] allows fine granular scalability to reconstruct images also at lower sampling ratios then the ones used to train the layers. It is possible simply removing some measurements bases from the sampling matrix with a greedy algorithm, that selects the most important ones, the ones that less impact on PSNR, and delete the others.

The algorithm is trained on 200 images of BSDS500 dataset and tested, with other well-known algorithms, on three popular datasets (SET5, SET14 and BSDS100). The sampling ratios taken into account are 0.01, 0.05, 0.1, 0.2, 0.3, 0.4 and 0.5. The results obtained overcome the state of the art both in quality and computational time. Also in case of a non-trained sampling ratio [8] outperforms the performances of compared methods taken into account as ISTA-Net and ISTA-Net$^+$.

Chapter 3

Background

In the last few years, in the field of image processing and reconstruction, Deep Learning has played an increasingly important role. Deep Learning (DL) is essentially a subset of Machnine Learning (ML), principally based on multi-layer Artificial Neural Network (ANN). The multi-layer nature of these kind of ANN is, in fact, what gives the name "Deep". In particular, a special kind of DL framework have been developed more and more in the last years, the autoencoder. This kind of framework try to automatically learn, in a unsupervised way, the encoding of a set of data and is used for various aims, the most important of which, is the problem of dimensionality reduction. Talking about dimensionality reduction, it is useful to introduce the other fundamental technique that underlies this work, the Compressive/Compressed Sensing/Sampling (CS). This is a mathematical technique that allows to solve underdetermined system and that is mostly applied in signal processing. The other and last argumets that will be treated in this chapter are the tools used in this work and in particular Python and its libraries.

3.1 Compressed Sensing

In the field of telecommunications the primary purpose is the exchange of information and information propagates by means of the signals. Since the last years of XX century, digital electronics has known a very impressive growth thanks to what it has replaced the analogue one in almost every field. Digital signals are, in fact, much easier to manipulate with respect to the analogue ones, but natural signals are analog, so a conversion to make them digital is needed. The analog to digital conversion consist of 4 phases: Filtering, Sampling, Quantization and finally Coding. In particular, Sampling is the

phase on which the attention of this paragraph will be placed and, speaking of sampling, it cannot be possible to avoid talking about the Shannon-Nyquist theorem, which states that the minimum sampling frequency of a signal, so as not to lose information and avoid the aliasing phenomenon, phenomenon whereby two sampled signals become indistinguishable, must be greater than or equal to 2 times the maximum signal frequency:

$$f_s \geq 2 \cdot f_{max} \tag{3.1}$$

This theorem is the basis of any type of signal acquisition protocol, whether it is naturally limited in bandwidth or not, in fact also signals such as images, which are not naturally bandwidth limited and whose sampling frequency is dictated by the resolution desired, actually they are sampled according to the Shannon-Nyquist theorem, since, before sampling this type of signals, an anti-aliasing filter is often applied. [1] As said at the beginning of this paragraph, the primary objective of telecommunications is the exchange of information. So, once sampled, the signal must be transmitted and, perhaps, manipulated and these operations obviously require bandwidth and computational capacity resources. With the development of technology, the number of data we acquire has increased dramatically and consequently to transmit these data you need more and more available bandwidth and to process them you need an ever greater calculation capacity. On the other hand, in reality, less and less data are used, it is in fact common to use compression and preprocessing techniques of this data to try to use as few resources as possible. For these reasons, in recent years, a new technique called Compressive Sensing (CS) has made its way, which is nothing more than a mathematical framework that allows to solve under-determined systems and that, applied in the field of image processing, allows to represent signals in a compact way, compressed precisely. This framework tries to overcome the Shannon-Nyquist theorem by going to use a number of data as close as possible to the number of degrees of freedom which is much less than the number of the actual data of a signal and this also make the signal much easier to manipulate and transmit. CS relies on two main concepts: Sparsity and Incoherence.
A signal is said to be sparse when most of its components are equal to zero. In particular, according to [4]:

Definition 3.1.1. *A signal $x \in \mathbb{R}^n$ is said to be $k-$sparse if it has at most k non-zero entries, i.e. $\|x\|_0 \leq k$*

In real life, signals are rarely exactly sparse, so it's needed to approximate

the concept of sparsity with the concept of compressibility. This means that x will be considered sparse also if it has a sparse representation in some basis Φ

$$x = \Phi c \qquad with\ c\ sparse \tag{3.2}$$

Now, the big problem is that nowadays a very large amount of data are collected, but very few of them are then actually used. Most part of them, in fact, is collected only to make compression possible and then are discarded. For example, for an image of megapixel resolution, compressed in JPEG 2000, just few kilobytes of data are taken.
So, tipically, what is done, once data are acquired, to compress a signal, is to approximate the signal x with a signal $\hat{x} \in \Sigma_k$, defined as the set of all $k - sparse$ signals, such that the approximation error is minimum:

$$\sigma_k(x)_p = \min_{\hat{x} \in \Sigma_k} \|x - \hat{x}\|_p \tag{3.3}$$

$\hat{x}$ will be obtained by taking just $k \ll n$ components c_i of c (remember), that are the ones carrying almost all the information, but this, as it easily to understand, is an adaptive way to compress signals, because the choice is made each time a different signal is considered. Obviously, the smaller k could be, the more compressible the signal is. This procedure is called Best k-terms Approximation. CS, instead, allows to directly acquire compressed data in a universal way, thanks to incoherent sensing, and performing as well as an adaptive way like Best k-terms Approximation. So, while in traditional sensing protocols, y is obtained after having sensed x, in CS, is possible to obtain directly the compressed data y. The sensing operation (fig. 3.1) can be easily explained as:

$$y = Ax$$

with $y \in \mathbb{R}^{m \times 1}$, $x \in \mathbb{R}^{n \times 1}$ and $A \in \mathbb{R}^{m \times n}$, $m \ll n$

First of all some assumptions are needed. In particular, x is assumed to be non-sparse and can be written as $x = \Phi c$, with c sparse. Now, the sensing problem doesn't change actually. The only difference is that, following the assumptions above, $y = A\Phi c$ will be solved for a certain $\hat{c}$ and clearly not only A, but $A\Phi$ should be properly choosen to obtain the right measurements, and it's here that incoherence comes in. Incoherence refers to the fact that the sensing matrix, unlike the signal of interest, should have a very dense

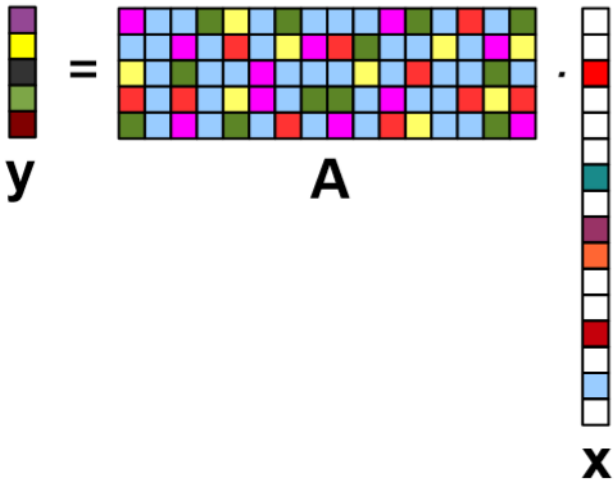


FIGURE 3.1: Sensing operation in CS, [5]

representation in the representation basis Φ. The motivation is easily understandable in fig 3.2, in which is possible to see that a sparse basis, like a chain of delta, will give, as output, a vector y that is null.

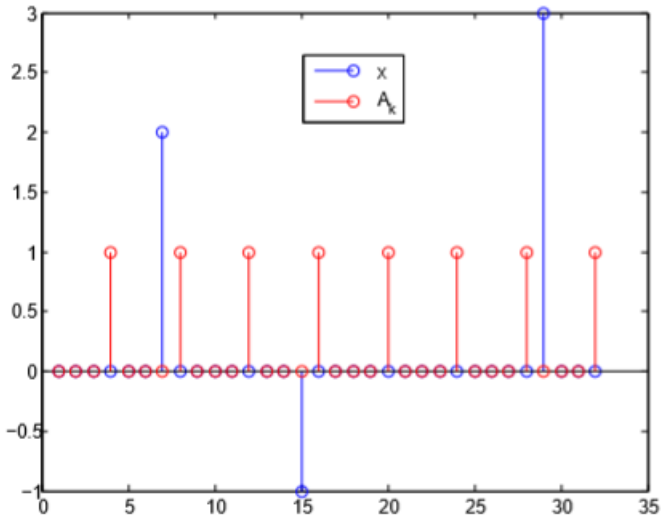


FIGURE 3.2: Coherent sensing, [6]

Coherence is defined as:

$$\nu(A, \phi) = \sqrt{n} \max_{1 \leq k,j \leq n} |\langle A_k, \phi_j \rangle|, \qquad \nu \in [1, \sqrt{n}] \tag{3.4}$$

where A_k and Φ_j are the columns vector of A and Φ. So, if the coherence between the sensing matrix and the representation basis is low, then it will be good for CS operations. It can be proved that random matrices have a

very low coherence with any fixed basis. The last operation in CS is, obviuosly, the reconstruction of the signal, that represents nowadays the most important challenge. The classical method used to recover the signal is the minimization of $l_1 - norm$, since the system is underdetermined and has infinte solutions:

$$\min_{\hat{x} \in \mathbb{R}^n} ||\hat{x}||_1 \qquad s.t. \quad y = Ax \tag{3.5}$$

More and more methods have been studied and developed during the years to better exploit the possibilities given by CS, and these methods can be grouped in two main categories: iterative and deep-learning based. The first ones are almost based on thresholding and, even if results in terms of quality are very good, they are very complex and take very long times to execute. The last ones have had a very impressive growth in the last years thanks to their speed, they just need to be trained once, and lower computational complexity. The focus of this work will be posed on them.

3.2 Deep Learning

The fields of artificial intelligence and machine learning have had a truly incredible development in all areas of technology thanks to the development of hardware technology and the increase in available data, essential to allow a good training phase in machine algorithms learning. In particular, in the field of image processing, a branch of machine learning has managed to catalyze the attention of the scientific community, Deep Learning. Deep Learning owes its name to the computing system on which it is based, the deep Artificial Neural Networks. An ANN is a computing system that aspires to function exactly as a human brain works and therefore tries to imitate the processes that take place within it. What has prompted us to deepen the study of these models is their ability to model very complex nonlinear functions and to be "trained". The elementary components of these models are neurons and "synapses", i.e. the connections between one neuron and another. Being nothing more than an interconnection of nodes, neural networks can be defined as direct and weighted graphs. The neurons receive an input, process it and produce an output, which is then transmitted to other neurons through the weighted links. The basic unit of an ANN is the Perceptron (fig. 3.3).

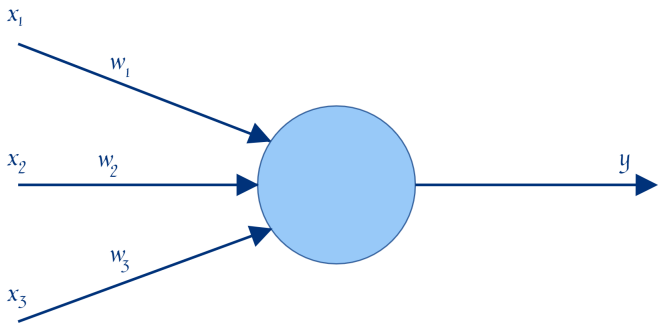


FIGURE 3.3: Perceptron

This works in a very simple way, in fact it is nothing more than a neuron that executes a dot product and compares it with a threshold b, called bias.

$$y = \begin{cases} 0 & if \sum (w_j \cdot x_j) + b \leq 0 \\ 1 & if \sum (w_j \cdot x_j) + b > 0 \end{cases}$$

If the result is greater than zero, the neuron "activates" otherwise it does not. However, Perceptron does not take into account small changes, in fact the output can only have two values. However, what you want from a neural network is that small changes in input produce small changes in output, or you want that the Network could learn, which is why the processed input is no longer compared with a threshold, but an activation function σ is applied and formula 1 becomes :

$$y = \sigma(\sum (w_j \cdot x_j) + b) \tag{3.6}$$

An example of activation function often used in this work is the *ReLU* which is shown in fig 3.4.

Neurons are organized in layers. Neurons belonging to one layer can only be connected with neurons from another layer. There are three types of layers: the input layer, which is the one that receives data from the outside, the output layer, which is the one that produces the results and, in the end, all the layers that are between the input layer and the output one are called hidden layers.

Being ANN a tool to solve an optimization problem, it is necessary to introduce the Cost or Loss Function, fundamental in the training phase of the ANN, giving it a measure of how close to a good results is the output. It could

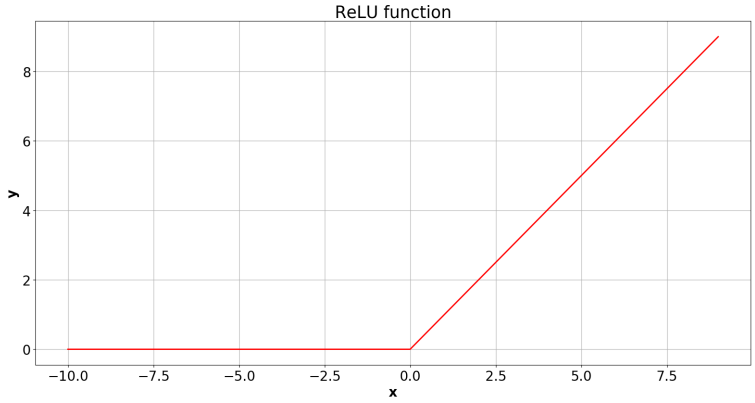


FIGURE 3.4: Example of ReLU function in interval $[-10, 10]$

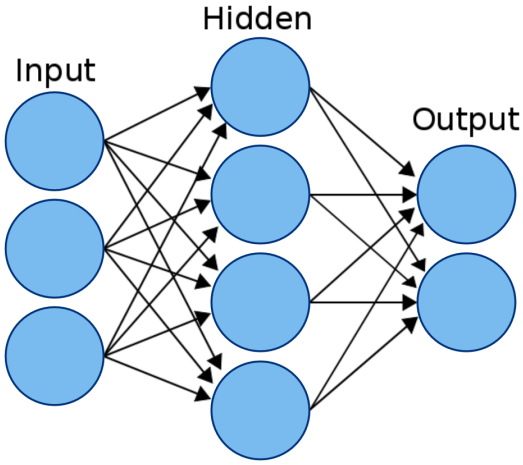


FIGURE 3.5: Scheme of a simple 3-layers feed-forward ANN

be suited for the problem taken into account and there are lots of methods to minimize it, the most used of which is the Gradient Descent. Together with the optimization methods, another algorithm is used to train most ANNs, the Error Backpropagation Algorithm. Its goal is to understand how changing a parameter can afflict the Loss Function, so as to allow a good update of them and so a good training. It works following the chain rule, very briefly, at the end of each pass forward in the network, the error in "sent" backward to optimally adjust weights and biases of each neuron in the network.

DL, by the way, stand on a particular kind of ANN, the Deep Neural Networks. The main feature of this kind of network is just the number of hidden layers present, that must be greater than one. Specifically, the ANNs used in Deep Learning for image processing are more complex. So as not to lose the information about the dimensionality of the image, in fact, the input of the network should be a two-dimensional input, in case of grey-scale images,

or a three-dimensional input, in case of RGB images. A simple feed-forward NN could also be used theoretically but, in a simple network like this one, every pixel of an image becomes a relevant variable and this leads to a big problem. In fact, for a small image of dimension 30x30, for example, a fully-connected layer that takes it in input should have 900 weights to pass it to the next layer. This brings, in a Deep NN, to an exploding number of variables as the number of layers increases. To avoid this problem, in DNN applied to image processing, simple matrix multiplications have been replaced by convolutions. Actually, convolution operator, in ANNs, allows to considerably reduce the number of variables in a layer and transforms the images into features maps, decreasing the computational complexity and the memory resources needed. The layer in which a convolution is computed is called Convolutional Layer and in there the input is convolved with multiple matrices, called filters or kernels, usually of very small dimension, e.g. $(3x3)$ or $(5x5)$.

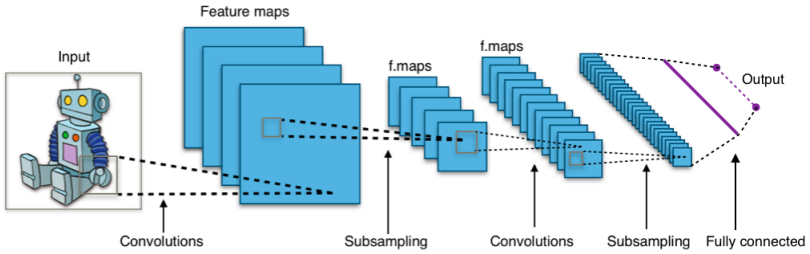


FIGURE 3.6: Typical Architecture of a CNN for Image Processing

What happens in a Convolutional Layer is shown in fig. 3.6 , where it is possible to see that the input is convolved with many filters, each of which corresponds to a neuron in the layer and give as output a features map. All the kernels in a layer share the same weights and this is one of the most important features of a Convolutional Neural Network (CNN). Thanks to this, in fact, the number of variables to learn in a layer depends only on the filter size and, as said before, being the filters dimensions very small, CNN allow to pass from thousands of variables to learn to just a few tens.

The layers in CNN try to simulate the behaviour of the human eye, for this reason each neuron in a layer receives in input informations about a small part of the image. This part of the image is called *receptive field* and, just like in human eye, it is the region of the space on which a stimulus must be present to activate a neuron. In CNN the receptive field is given by a

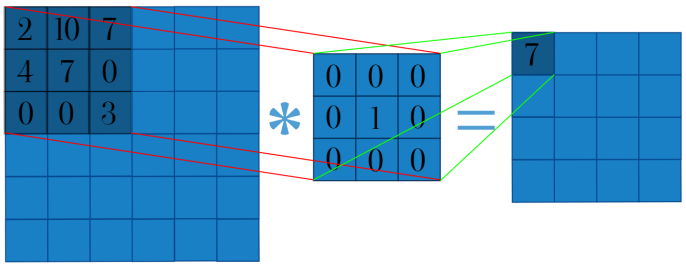


FIGURE 3.7: Example of Convolution between matrices

sparse connectivity between the layers, in this way, in fact, neurons of a layers will receive input just from some of the neurons of the previous layer, corresponding to a certain region of the space. To better understand this concept it would be easy to take a look at fig. 3.7 in which is shown the output of th convolution. So in the matrix at the right of the image, imaging each cell as a neuron, it could be seen that, for example, the receptive field of the cell in position $(1,1)$ is the squared region of space that goes from cell $(0,0)$ and extends until cell $(3,3)$ in the square to the left of the image (previous layer). To reduce the dimensionality of the problem, a way could be to increase the *stride* of the convolution, that is how many pixels the kernel moves each time. Normally it is equal to 1 so, as easy to understand, the receptive fields of each neuron overlaps very much, but it is possible to increase the stride, but big strides are not used so much due to the fact that a big stride makes you lose information about a space region. A big stride, equal to the size of the kernel so they don't overlap, is used to simulate some kind of sampling in autoencoders. Often, following a Convolutional Layer, it could be find a Pooling layer, that is another way, the most used one, to reduce the dimensionality of the problem and also to reach a higher level of abstraction.. The last important tool used in CNN is the *Padding*. This is a very fundamental tool, thanks to which is possible to maintain constant the dimension of the image and not to lose information about edges.

Fig. 3.8 shows how Padding works, from which can be easily understood the kernel convolving with the new image, formed by the input plus the zeros all around forming a ledge, in such a way that the dimension of output remains equal to the one of the input.

Thanks to all this tools, Deep CNN results well suited for computer vision, combining the best mathematical tools to analyse images with the deepness that allow the network to better learn all the features present in the input in a

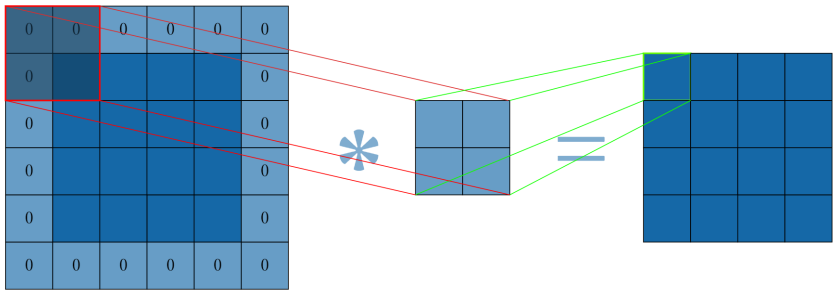


FIGURE 3.8: Convolution between matrices with Zero Padding. It maintains, in output, the same shape of the input matrix.

hierarchical way, from the basic ones, like lines, to the most complex of every kind.

3.3 Python: Tools and Libraries

The programming language used for this work is Python. It is a high level, object-oriented language that has had an impressive growth in the last year for what concern the development of Machine Learning algorithms. What has made Python so popular in the ML field is the fact that it is possible to find packages and libraries suited for most, if not all, of the instrument useful for ML. To develop this work many libraries were used, the main ones being: Keras, PIL and SciPy (NumPy and Matplotlib).

3.3.1 Keras

Keras [1] is a pythonic high level interface for Machine Learning algorithms development. Giving a higher level of abstraction with respect to other similar libraries it results also more intuitive and suited for designing complex networks. It supports also various backends like Tensorflow, the one used in this work, and Theano. Keras is particularly suitable to implement NN that are not just straight-forward, thanks to its language simplicity in fact, it is possible to easily implement very complex neural network structures. It also allow the user to perform data augmentation and general pre-processing of the data. The most used commands in this work are the following:

- Conv2D
- Lambda

[1] https://keras.io/

- Add, Subtract
- ImageDataGenerator
- Model, Compile, Fit

Furthermore, thanks to Keras Tensorflow backend is possible to go deep in the network and well manipulate tensors inside the NN, that pure Keras wouldn't support, as shown in the following code that shows the COMBINATION function used to merge the patches entering the network into a whole image.

```
from keras import backend as K

def COMBINATION(x):
    tf.executing_eagerly()
    final=[]
    x=K.reshape(x,(batch_size,3,3,block_size**2))
    for j in range(0,batch_size+1):
        really = []
        tmp = x[j]
        tmp=K.reshape(tmp,(3,3,block_size,block_size))
        tmp=K.reshape(tmp,(9,block_size,block_size))
        for i in range(0,7,3):
            tmp_2 = K.concatenate((tmp[i],tmp[i+1],
            tmp[i+2]), axis=1)
            really.append(tmp_2)
        pippo = K.concatenate(really, axis=0)
        final.append(pippo)
    final = tf.stack(final,axis = 0)
    #final=K.constant(final)
    final=K.reshape(final,(batch_size,patch_size,patch_size,1))
    return final
```

Conv2D

The `Conv2D` command is one of the base command that Keras provides. It is used to implement a Convolutional layer that has, as input, an image,

or however, a three-dimensional input (actually 4 dimensional: (batch_size, width, height, channel)) to be convolved with a two-dimensional kernel.

```
IN = Input(shape=(96, 96, 1))
FIRST=Conv2D(filters=nb,
             kernel_size=(32,32),
             kernel_initializer='random_normal',
             strides=32, data_format='channels_last',
             padding='valid', use_bias=False,
             trainable=False)(IN)
```

In the code above is shown an example of definition of a Conv2D layer in which it is possible to see all the parameters that can be set. First of all there is the `filters` parameters, that stands for the number of filters that the user want to use in that layer, of course setting it, it will be set also the dimension of the output. Following, there is the `kernel\_size` that set the dimension of the filter for that layer, followed by the `strides` parameter that allow the user to choose how many pixels to move on at each step. Notice that in a 2D layer, the strides parameter is a tuple of 2 values. `Padding` is a parameter that make the programmer choose if mantaining the input dimension or not, padding, precisly, the input with zeros. `Data\_format` allows to specify the order of the input dimension, e.g. `channel\_fisrt` or `channel\_last`. `Dilatation\_rate` is an integer, or a tuple, that specify the field of the convolution with the kernel. More specifically, it specifies the pixels that should not be convolved, i.e. setting this parameter is possible to compute the convolution with a kernel (3x3), but in a field (5x5), for example, leaving the excess pixels as they are. One of the most important parameters, above all in applications like auto-encoders or the one developed in this work, is the `kernel\_initializer`. Thanks to it, it is possible to choose the filter. There are default filters to choose from, or it's possible to design your own filter. For an application like the one developed in this work, this parameter has been one of the most important, representing, actually, the form of the *SensingMatrix*. Just like for the kernel, you can choose also the initializer for the bias setting the `bias\_initializer` parameter, that is usually equal to *"zeros"*. "Regularizers allow you to apply penalties on layer parameters or layer activity during optimization", while constraints "allow setting constraints (eg. non-negativity) on model parameters during training. They are

per-variable projection functions applied to the target variable after each gradient update" **??**. In this work, this command has been use not just to implement a Convolutional Layer for feature extraction, but also to implement the sampling of the input image, thanks to a particular set of parameters that will be discussed in the next chapters.

Lambda

```
Output=Lambda(COMBINATION)(Tensor)
```

The `Lambda` layer in Keras is a customizable layer. It is suited to encapsulate simple operations (not present by default) into a Keras Layer (COMBINATION function in the code highlighted at the beginning of the section). It allows to apply every transformation or operation to a tensor, like, for example, a customized merge function. This layer has been used to implement a `Combination` layer, that had to reassemble the tensor into an image, being the image in input cut into patches.

Add and Subtract

```
Ouput1=Subtract()([Input,Tensor])
Ouput2=Add()([Ouput1,Tensor])
```

`Add` and `Subtract` are two commands that allow the programmer to make operations on the tensors (generalization of the concept of vectors) in the network. The first one makes possible to sum tensors between them. Obviously the tensors must have the same dimensions. The same goes for the `Subtract` command, which instead performs the subtraction operation. This one, in particular, has been used, in this work, to evaluate the residual image, useful to obtain the initial reconstruction.

ImageDataGenerator

```
transform_parameters_1={'theta': 90}
transform_parameters_2={'theta': 90, 'flip_vertical': 1}
transform_parameters_3={'theta': 180}
transform_parameters_4={'theta': 180, 'flip_vertical': 1}
```

```
transform_parameters_5={'theta': 270}
transform_parameters_6={'theta': 270, 'flip_vertical': 1}
transform_parameters_7={'flip_vertical': 1}

AUG=ImageDataGenerator()

 for i in Train:
     Aug_Train.append(AUG.apply_transform(x=i,
     transform_parameters=transform_parameters_1))
     Aug_Train.append(AUG.apply_transform(x=i,
     transform_parameters=transform_parameters_2))
     Aug_Train.append(AUG.apply_transform(x=i,
     transform_parameters=transform_parameters_3))
     Aug_Train.append(AUG.apply_transform(x=i,
     transform_parameters=transform_parameters_4))
     Aug_Train.append(AUG.apply_transform(x=i,
     transform_parameters=transform_parameters_5))
     Aug_Train.append(AUG.apply_transform(x=i,
     transform_parameters=transform_parameters_6))
     Aug_Train.append(AUG.apply_transform(x=i,
     transform_parameters=transform_parameters_7))
```

This command, unlike the previous ones, is not used to implement a layer. Keras, in fact, provides this class to perform generation of data or *data augmentation*. This last one is very important in image processing. Thanks to it, it is possible to expand the Training dataset and, as consequence, improve the results. Actually, what is done in data augmentation are simple operation that, in the case of the images, consist of rotation and mirroring, for example. In the work developed in this thesis the transformation used to perform *data augmentation* with the `ImageDataGenerator` class are shown in the code above. In particular, in that code, are performed 7 kind of transformation (rotations, flippings) that, in the end, have increased the size of the training dataset by a factor of seven.

Model, Compile, Fit

These three commands are used to instantiate a `Model` Class object and train it. In particular `Model.compile` is used to define the loss function, in this case

Mean Squared Error, and the optimizer, in this case the Adam optimizer. The `fit` command, instead, is used to launch the training phase and it is possible to specify some parameters like batch size, number of epochs, dataset to be used as Training set and Target set as well as the dataset for the validation phase and in the end you can choose to enable the shuffle mode. This commands will be described better and more deeply in section 4.2.2.

3.3.2 PIL

```
def Create_Dataset():
    data=[]
    for name in glob('./train/*?.jpg'):
        image=Image.open(name)
        h, w = image.size
        im = image.resize((h-(h%96),w-(w%96)), resample = 0)
        h, w = im.size
        im = np.asarray(im.convert('L')).T
        # if(im.shape[0]==320):
        im = (im/255)-0.5
        im=np.expand_dims(im,axis=-1)
        data.append(im)
    data=np.array(data)
    return data

.
.
.

 immagini=[]
 for i in range(0,77):
     immagini.append(Image.fromarray(prova[i].astype(np.uint8)))

 for k in range(0,11):
     for i in range(0,7):
         immagini[(7*k)+i].save(nomi[k]+str(i)+".png")
```

PIL (Python Imaging Library) is a free Python library suited to process and manipulate images. It allows every kind of operations on images, from pixel-per-pixel manipulation to filtering and it also allows to change the format of the images. The highlighted code shows the usage of this library and it is possible to understand how easy and comprehensible it is. Despite its capabilities, in this thesis, this library has been used just in a basic way, since it was just needed to load, resize and, finally, save the images.

3.3.3 SciPy

```
def COMBINATION_NP(im,h,w,h_m,w_m):
    im=np.array(im)
    hi=int(h/h_m)
    wi=int(w/w_m)
    really=[]
    for i in range(0,im.shape[0],wi):
        tmp = np.concatenate(im[i:i+wi], axis=1)
        really.append(tmp)
    if(hi==1):
        return(np.reshape(np.array(really), (h,w)))
    pippo=np.concatenate(really, axis=0)
    h,w=np.shape(np.array(pippo))
    final=np.reshape(np.array(pippo), (h,w))
    return final
```

```
def CROPPING(im, z):
    h,w,c = np.shape(im)
    x=[]
    x.append(im.reshape(h//patch_size, patch_size, -1,
    patch_size).swapaxes(1, 2).reshape(-1, patch_size,
     patch_size))
    x=np.array(x)
    x=x.reshape((h//patch_size*w//patch_size, patch_size,
     patch_size))
    z.append(x)
    return z
```

```
def PREDICT_MIA(im, loaded_model):
    h,w,c=np.shape(im)
    s=(h,w)
    s0=(480,288)
    s1=(288,480)
    if(h*w==138240):
        l = []
        CROPPING(im, l)
        l=np.array(l)
        l=l.reshape((15,96,96,1))
        pred=loaded_model.predict(l)
        pred_comb=COMBINATION_NP(pred, h, w)
        final=np.expand_dims(pred_comb, -1)
        return final
```

SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering."[2] In particular, to develop this work, NumPy and Matplotlib have been used. The first is a library developed to provide numerical computing in Python, above all to manipulate arrays, matrices and so multi-dimensional arrays. Its use in this work was fundamental since it has been used to develop all the functions to prepare the data and create the *Train*, *Test* and *Validation* dataset for the NN and the customized `PREDICT` function, which aims was to prepare the input of the NN cropping it into patches, using the ad hoc `CROPPING` function, send it to the Keras `predict` function and, finally, re-assembly the patches to get the final result with the `COMBINATION_NP` function. Thanks to NumPy, in fact, is very easy to manipulate images as three-dimensional matrices, so it has been preferred to Keras and PIL to perform these tasks outside the NN. Matplotlib is, instead, a library for 2D plotting very used by Python programmers. In the following codes is displayed an example of this library usage.

```
import matplotlib as mpl
from matplotlib import rcParams, rc
```

[2] https://www.scipy.org/

```
rcParams.update({'font.size': 30})
plt.rcParams['font.size'] = 22
plt.rcParams['axes.labelsize'] = 20
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['axes.titlesize'] = 25
plt.rcParams['xtick.labelsize'] = 20
plt.rcParams['ytick.labelsize'] = 20
plt.rcParams['legend.fontsize'] = 20
plt.rcParams['figure.titlesize'] = 15
plt.rcParams["figure.figsize"] = (16,9)
plt.rcParams["axes.grid"] = True


nomi=["Barbara","Boats","Cameraman","Fingerprint","Flinstones",
      "Foreman","House","Lena","Monarch","Parrot","Peppers"]
k=0
for i in range(0,77,7):
    plt.figure()
    plt.ylabel('PSNR')
    plt.xlabel('No. of Layer')
    plt.plot(errori16[i:i+7],'go-',linewidth=3,markersize=5)
    plt.plot(errori16_nt[i:i+7],'ro-',linewidth=3,markersize=5)
    plt.plot(errori32[i:i+7],'bo-',linewidth=3,markersize=5)
    plt.plot(errori32_nt[i:i+7], 'yo-',linewidth=3,markersize=5)
    plt.plot(errori48[i:i+7],'co-',linewidth=3,markersize=5)
    plt.plot(errori96[i:i+7],'mo-',linewidth=3,markersize=5)
    plt.legend(['BS=16 Trainable', 'BS=16 Not Trainable',
    'BS=32 Trainable', 'BS=32 Not Trainable',
    'BS=48 Trainable', 'BS=96 Trainable'])
    plt.title("PSNR of"+nomi[k]+"for each layer and block size")
    plt.savefig(nomi[k]+".png", dpi=300)
    k=k+1
    plt.close()
```

In the first part there is a setting of the parameters of the plots while in the last part there is the part regarding the plot itself, with all the features that could be applied to a plot, like title, legend, colours and width of the line, labels and type of the markers.

Chapter 4

Deep Learning method for Compressed Sensing image reconstruction

This chapter provides a detailed description of the structure of the framework developed to implement a deep learning based image reconstruction from a compressed sensed input. All parts of the framework will be described in great detail, in particular the Deep Reconstruction Network and the Compressed Sensing sub-net. Besides the framework itself, will be also present all the phase of pre-processing of the data and the prediction along with the post processing. All the algorithms are written in Python with the help of libraries like Keras, PIL and SciPy. At the end of the chapter will also be presented sections about the performance evaluation used in this work and the setting of the algorithm for the experiments.

4.1 Framework structure

This paragraph will try to well explain the structure of the framework developed in this work, analysing in details the capabilities of each macro-layer. In figure 4.1 is possible to see a block-diagram of the structure of a layer, with five macroscopic blocks, plus the input one and output one of course.

The layers are the following:

- Data Preprocessing
- CNN
  - Sampling

  - Initial Reconstruction
  - Deep Reconstruction Network
- Data Post Processing

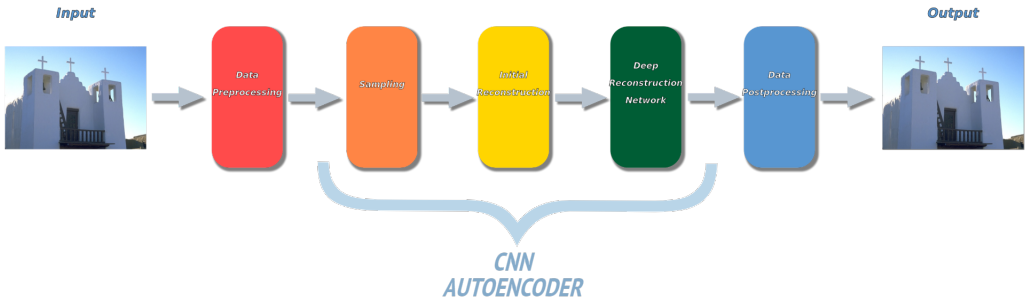


FIGURE 4.1: Basic structure of a layer of the implemented framework

4.1.1 Data Preprocessing

This section aims to provide a detailed description of the Preprocessing that data underwent to make them ready and transform them into the input of the auto-encoder, treated in section 4.1.2.

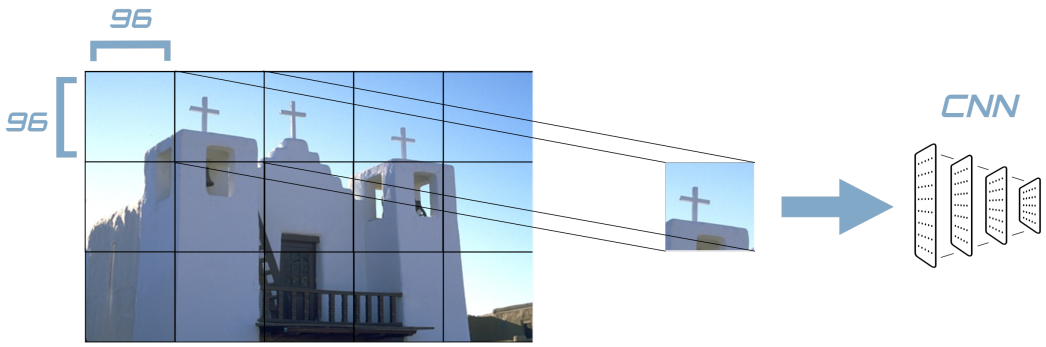


FIGURE 4.2: Cropping phase

Fig 4.2 shows how an image is cropped into multiple patches of size $(96, 96)$, to not overload the network, thanks to the *CROPPING* function.

```
def CROPPING(im, z):
    h,w,c = np.shape(im)
    x=[]
    x.append(im.reshape(h//patch_size, patch_size, -1,
```

```
5       patch_size).swapaxes(1, 2).reshape(-1, patch_size,
6        patch_size))
7       x=np.array(x)
8       x=x.reshape((h//patch_size*w//patch_size, patch_size,
9        patch_size))
10      z.append(x)
11      return z
```

In line 4-6 of the python code above, the image is cropped and then each patch is appended to a list. So, the list obtained is then converted into an array of matrices (i.e. images) that are then used for two goals. In training phase, these patches, obtained from all the images coming from train dataset, are used to train the network, while in prediction phase, the image is just one and the patches obtained are sent to the *PREDICT_MIA* function to get the reconstruction. A variant of the *CROPPING* function is used also in the *PREDICT_MIA* function, because the network cannot be built with using variable dimensions. So, while in the training phase the dimensions of the images coming from the Train dataset are always the same, in the reconstruction phase we want the network to reconstruct images of all size. To do this, the images are manipulated so that it becomes possible to divide them into (96,96) patches. For example, just like in figure 4.3, an image bigger than the ones used for training (the dimensions used are (480,320) or (320,480)) is padded until having dimensions that could be divided perfectly by 480 or 320. Being then, the batch size equal to 15, the patches undergo the CNN by group of 15 to be then assembled at the end of the `PREDICT_MIA` function.

4.1.2 CNN Autoencoder

This section will explain the three central layers illustrated in figure 4.1, i.e. *Sampling*, *Initial Reconstruction* and *Deep Reconstruction Network*, forming together the Autoencoder implemented in this thesis. Actually the network is based on hierarchical layers that allows to get the reconstruction of the input for different CS Ratios, each of the layer exploiting the lower layer initial reconstruction to improve the performances, like shown in fig. 4.8. We can say that the final structure is indeed a stack of n network represented in fig 4.1 The structure of this network has been developed starting from [8], that has been the guide for this work development.

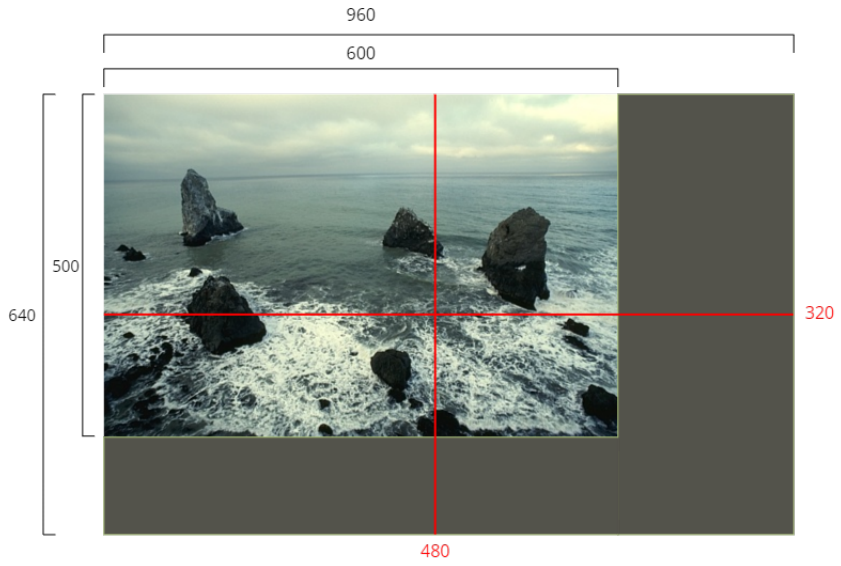


FIGURE 4.3: Image padded before undergoing the CNN

Sampling

The network, as already mentioned, could be divided into three macro blocks. Starting from the first, the *Sampling* one, this block is actually equivalent to a Keras layer. To simulate the sampling function it has been used a classical convolutional layer with the `strides` parameter equal to the size of the filter.

```
FIRST=Conv2D(filters=nb, kernel_size=(32,32),
             kernel_initializer='random_normal',
             strides=32,
             data_format='channels_last',
             padding='valid',
             use_bias=False,
             trainable=False)(INPUT)
```

In the highlighted code is shown the setting of the layer that simulate the sampling. The *kernel_size* is set to $(32, 32)$ because we want to sample blocks of this size (fig. 4.4, 4.5) and so, also the strides parameter is set to 32 to don't allow filters to overlap. Besides, other important parameters are the *kernel_initializer* that is a random normal kernel, so it is fixed, the *padding* parameter, that is set to *'valid'*, meaning that the dimension will not be maintained and so, that the image will be compressed. The layer is not trained and

have no bias and no activation function. This, in fact, allowed to simulate a simple operation of sampling, convolving the image with non-overlapping filters, making it possible to be implemented in Keras, instead of make it as preprocessing operation, outside of the network.

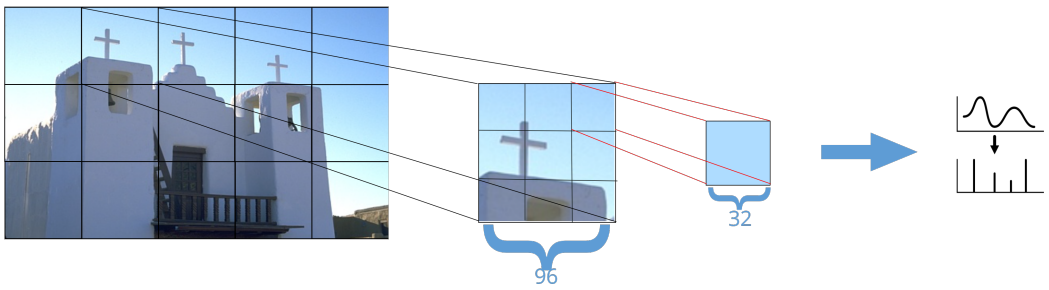


FIGURE 4.4: Blocks making in sampling phase

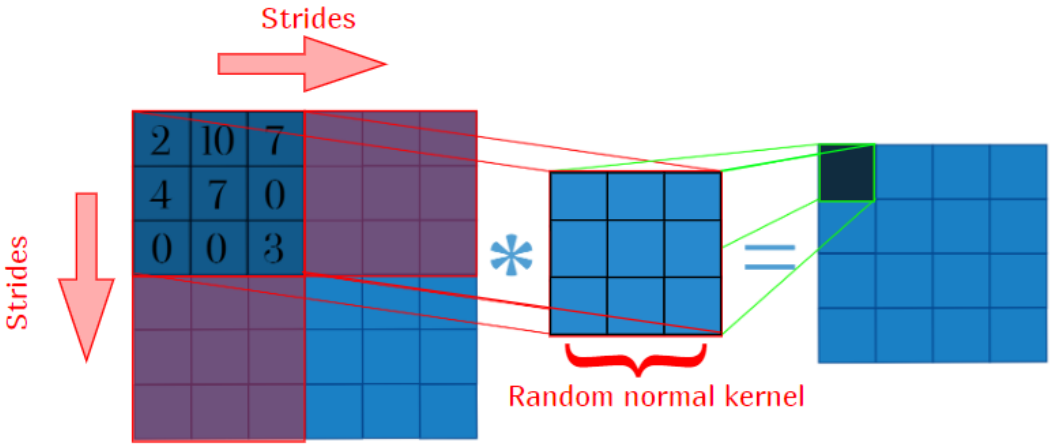


FIGURE 4.5: Example of non overlapping Convolution to simulate sampling.

Initial Reconstruction

The first step to get a CS reconstruction is to get an initial reconstruction, that will then undergo the Deep Reconstruction Network (DRN). The initial reconstruction network is composed by three layers, two of them are convolutional and one is a combination layer that you need to reshape and then put back together the blocks. The first layer transform the sub-sampled image into n features maps and then these features maps are converted into higher abstraction level features maps (fig. 4.6).
The number n depends on the CS Ratio chosen to implement the CS measurements, while the second layer output dimensions should be choose such

as they are useful to re-assemble the blocks into the whole image. These characteristics will be treated better in chapter 5.

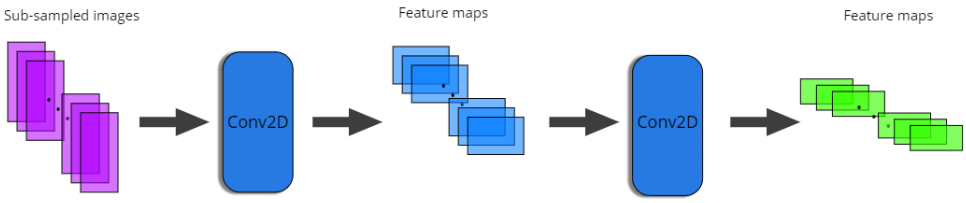


FIGURE 4.6: Scheme of the first two layers of initial reconstruction network

After the two `Conv2D` layers we find a Keras' Lambda layer that is used to wrap the `COMBINATION` function where the tensors (patches) are manipulated using the Keras' Tensorflow Backend. Actually the `COMBINATION` function has been developed to implement two layers: a reshape layer and a combination layer. The first layer is used to obtain again blocks of the desired size and the second input them together to obtain the whole image (fig. 4.7)

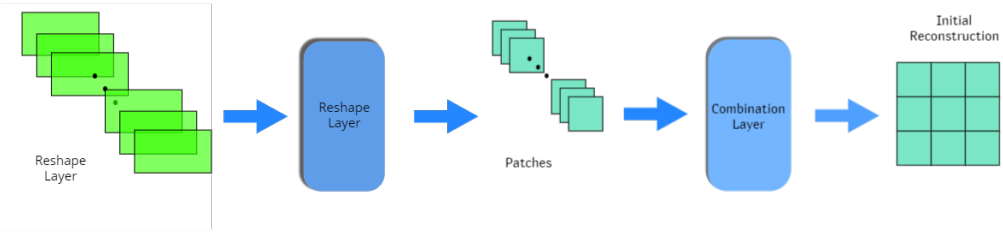


FIGURE 4.7: Scheme of Combination procedure of Initial Reconstruction Network

The procedure explained until now is the basic procedure for the basic layer of the network. For all the higher remaining layers there is one more step, that consists of the sum of the residual image (between the first initial reconstruction and the input) plus the initial reconstruction of the lower level, that allows to obtain a better initial reconstruction. The figure 4.8 well explain the behaviour of the network.

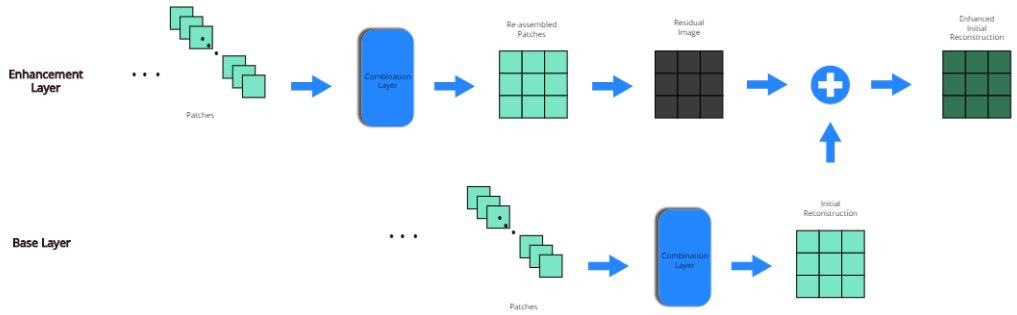


FIGURE 4.8: Initial reconstruction network for enhancement layer

Deep Reconstruction Network

The Deep Reconstruction Network is the most important part of this work, even if it could look like it is very simple in the structure.

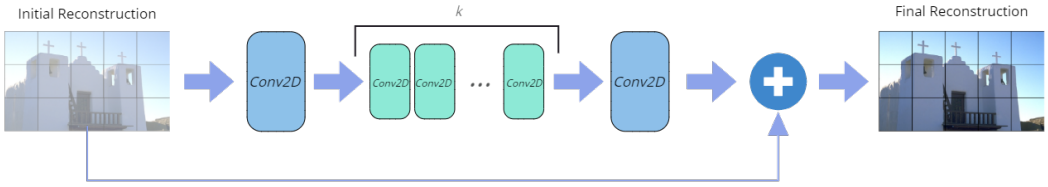


FIGURE 4.9: Deep Reconstruction Network scheme

As figure 4.9 shows, the DRN is composed by a series of Convolutional layers in cascade, that, based on the state of the art, form a hourglass shape network. This network is used to implement, just like in [8], six operations: feature extraction, shrinking, non-linear mapping, expanding, feature aggregation, and skip connection. Actually the skip connection is not implemented with a convolutional layer, but with a simple `Add` layer. The activation function used for these layers is the ReLU, except for the `Add` one, of course.

Differently from the other parts of the entire network, this part has remained fixed for all the experiments and its settings will be discussed in the next chapter.

Data Postprocessing

This section will discuss the post-processing under which the data go at the end of the network. This processing is very similar to what happens inside the CNN when patches are concatenated to obtain the whole image. In fact, the image we've been talk about in the previous sections actually is not the

whole image, but a patch. What enter in the CNN is a batch of patches that are elaborated individually until they are re-assembled. So, almost in a recursive way, at the end of the CNN the patches must be concatenated to get the reconstructed image. In the case the input image to predict has been padded (fig. 4.3), the padding part must be obviously cut off, and so, again, in a recursive way, first of all patches are concatenated and then the images obtained are concatenated themselves once again to obtain finally the whole image.

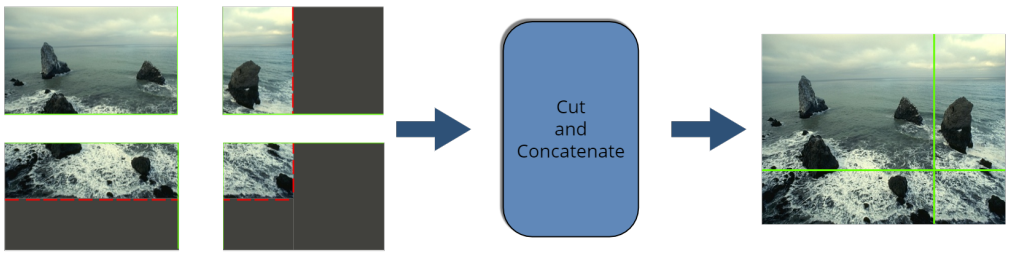


FIGURE 4.10: Cut and concatenation of final reconstructed image

4.2 Experiments and numerical settings

In this work we have considered some different settings in order to perform our experiments and this section will focus on them, in particular on the settings of the network, both CNN and CS sub-net described in section 4.1. They will be analyzed from a microscopical point of view, describing their features and, above all, their training phase. Will also be presented the preprocessing of the data to make them ready to train the network.

4.2.1 Preprocessing and Data Augmentation

The main problem of Deep Learning, and Machine Learning in general, is not just find a proper setting for the network. The main problem is indeed having a good and, above all, large dataset on which training the algorithm. The dataset is maybe the most important part in this kind of algorithms, in fact, the largest is the dataset the most the network can learn. Obviously is not enough for a good dataset to be large, it also have to be various so the network can learn lots of different things. Besides, we want that a NN can classify images independently from where the subject of the image is or from which point of view it is seen and we want that a NN could recognize a man, for example, whether he is in front of the camera or upside down while jumping. For all this reasons has been introduced the technique of data augmentation. Thanks to it, in fact, is possible to enlarge a dataset and also make it more various to obtain better performances from the algorithm and avoid overfitting too. The data augmentation techniques are many and often not all of them are needed. It depends on what the algorithm requires. Even if the dataset is already large, performing data augmentation could be a very good choice because it can improve the number of relevant patterns or features in the dataset. Data augmentation together with a good preprocessing of the data can improve drastically the performances of an algorithm. The dataset used in this work is the BSDS500 both for train and for validation, while for the test phase the famous SET11 have been used. The first operation performed on the data is the conversion of RGB images into greyscale images. Then the data have been normalized between $-0,5$ and $0,5$ (eq. 4.1). The normalization in addition of improving the performances also reduces drastically the elaboration time that is a very important parameter, above all

in CNN.

$$\hat{x} = \frac{x}{Max_{pixel}} - \mu \tag{4.1}$$

After having converted and normalized the dataset, patches have been created. The reason of creating patches is very simple. The dimension of a normal image is too large for a NN and the elaboration time could be very very long. This is why we have decided to train the network with small patches of dimension $(96, 96)$, as already described in subsection 4.1.1. Doing that we have also enlarged the dataset by a factor of 15, passing from 200 images to 3000. The last operation on the dataset is the data augmentation. In the code below are shown the transformations applied in this work and in figure 4.11 is shown an example.

```
transform_parameters_1={'theta': 90}
transform_parameters_2={'theta': 90, 'flip_vertical': 1}
transform_parameters_3={'theta': 180}
transform_parameters_4={'theta': 180, 'flip_vertical': 1}
transform_parameters_5={'theta': 270}
transform_parameters_6={'theta': 270, 'flip_vertical': 1}
transform_parameters_7={'flip_vertical': 1}
```

The transformation are very simple and are all about rotation kind because of the nature of the dataset that is very various and of course because of the kind of algorithm implemented not for classification, but for image reconstruction.

FIGURE 4.11: Example of some kind of data augmentation implemented in this work

Other kind of transformation are unnecessary in this case, but with just these few and simple kind of transformation the dataset has increased again its size from 3000 to 24000 patches, i.e. growing by a factor of 8.

4.2.2 Settings

This section will be focused on the settings used to implement the algorithm and so all the features of the network as well as the settings of the training phase. It will follow the order presented in section 4.1 starting first from the Compressive Sensing sub-net and then focusing on the Deep Reconstruction Network and finally the training phase.

Compressive Sensing sub-network

The Compressive Sensing is a technique that allows to acquire compressed data in a universal way to avoid waste of time, memory and computational resources and also to transmit data in a easier way. As the name itself says, the pivot point in this technique is the compression. The user can decide autonomously how to compress data, choosing the type of the sensing matrix, and so the basis in which the data will be represented, e.g. random matrix with a Gaussian or a Bernoulli distribution or a Fourier basis. The other parameter the user can decide is about how much he wants to compress the data, i.e. the number of rows of the sensing matrix. This parameter is called *CS Ratio* In this work the experiments have been conducted for different values of CS ratio and ortho-normal sensing matrices, both trained and fixed. All the experiments have been then repeated for various block sizes.
The CS Ratios used are: 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5 each of which corresponding to a layer in the stack.
The sensing matrix used, instead, is a random *ortho-normal* kernel implemented in the first Conv2D layer of Keras.
Finally the block sizes taken into account have been: $(16, 16)$, $(32, 32)$, $(48, 48)$, $(96, 96)$ (i.e. the whole image)
Thanks to the Python and, above all, Keras simplicity, we could set all these parameters just in one command line.

```
nb=int(np.floor(CS_ratio*block_size*block_size))
IN = Input(shape=(patch_size, patch_size, 1))
FIRST=Conv2D(filters=nb, kernel_size=block_size,
             kernel_initializer='random_normal',
             strides=block_size,
             data_format='channels_last', padding='valid',
             use_bias=False, trainable=False)(IN)
```

The input size for the image is $(96, 96)$, as already discussed in section 4.1 and the batch size is equal to 15, with a resulting input of dimension $(15, 96, 96, 1)$ The number of filters, i.e. the number of channels in output of the convolutional layer, n_b has been set to:

$$n_b = \lfloor CS_{ratio} * block\_size^2 \rfloor$$

Of course one of the most important parameter of this layer is the strides parameter that must be equal to block size to not overlap filters. Right after the sensing layer, i.e. the first convolutional layer, we have another convolutional layer with $block\_size^2$ filters with dimension equal to (1,1).

```
SECOND=Conv2D(filters=block_size**2, kernel_size=(1,1),
              strides=1, data_format = "channels_last",
              padding='valid', use_bias=True)(FIRST)
```

This convolutional layer with these settings allows to obtain an output that could be easily reshaped into blocks of block size dimension that are then concatenated. The next step in the network is the reshape and concatenate layer, implemented thanks to the Lambda layer provided by Keras of course, that wrap a customized function, already described in subsection 3.3.1 and reported down here too.

```
from keras import backend as K

def COMBINATION(x):
    tf.executing_eagerly()
    final=[]
    x=K.reshape(x,(batch_size,3,3,block_size**2))
    for j in range(0,batch_size+1):
        really = []
        tmp = x[j]
        tmp=K.reshape(tmp,(3,3,block_size,block_size))
        tmp=K.reshape(tmp,(9,block_size,block_size))
        for i in range(0,7,3):
            tmp_2 = K.concatenate((tmp[i],tmp[i+1],
            tmp[i+2]), axis=1)
            really.append(tmp_2)
        pippo = K.concatenate(really, axis=0)
```

```
        final.append(pippo)
    final = tf.stack(final,axis = 0)
    #final=K.constant(final)
    final=K.reshape(final,(batch_size,patch_size,patch_size,1))
    return final
```

The COMBINATION function just reshape the vectors of dimensions (1,1,1024) into blocks of dimensions (32,32,1) and then concatenate them in order to obtain the initial reconstruction. For all the higher layer in the stack there is another layer that computes the residual image between the first initial reconstruction and the input image and then add to it the initial reconstruction of the layer right below.

```
Ouput1=Subtract()([Input,Tensor])
Ouput2=Add()([Ouput1,Tensor])
```

Deep Reconstruction Network

Once obtained the initial reconstruction, what we actually want is to enhance it until obtaining a very good quality image. To do this a Deep Reconstruction Network (DRN) has been developed and implemented. The DRN has been designed, as the state of the art suggests, with a hour glass shape. The first and last layer are, in fact, "bigger" than the others in the middle. The implementation in python is highlighted below.

```
l = 1
d = 128
s = 32

FIFTH=Conv2D(filters=d, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(FOURTH)
SIXTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(FIFTH)
SEVENTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
```

```
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(SIXTH)
EIGHTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(SEVENTH)
NINTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(EIGHTH)
TENTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(NINTH)
ELEVENTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(TENTH)
TWELVETH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(ELEVENTH)
THIRTEENTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(TWELVETH)
FOURTEENTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(THIRTEENTH)
FIFTEENTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(FOURTEENTH)
SIXTEENTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(FIFTEENTH)
SEVENTEENTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(SIXTEENTH)
EIGHTEENTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(SEVENTEENTH)
NINETINTH=Conv2D(filters=s, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(EIGHTEENTH)
```

```
TWENTIETH=Conv2D(filters=d, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(NINETINTH)
FINAL=Conv2D(filters=1, kernel_size=(3,3), strides=1,
data_format = "channels_last", activation=relu, padding='same',
use_bias=True)(TWENTIETH)
OUT=Add()([FINAL,FOURTH])
```

Precisely, we have seventeen layers composing the DRN with the function of feature extraction, shrinking, non-linear mapping, expanding and feature aggregation, as already mentioned in section 4.1. All the layers convolve their input with a (3,3) kernel with strides equal to 1 and, above all, zero padding to maintain always the same input dimension. The first layer in this subnet has 128 filters as well as the last of the hour glass shaped part, i.e. the fifteenth layer. All the "small" layers in the middle have, instead, 32 filters. All the layers are followed by a ReLU activation function. The last two layers are different. The very last one is the layer that implement the skip connection, i.e. that add the initial reconstruction to the reconstructed image coming out from the DRN, the second last layer, on the contrary, is used almost as a reshape layer, having just one filter to obtain an output with the right dimensions.

Training Phase

The most important part, in a Deep Learning algorithm, along with the implementation of it, of course, is the training phase. The training phase of Convolutional Deep Neural Network is often very long, so it has to be designed properly to not waste time. The Keras commands useful to implement the training phase are not so many, in particular they are: `Model`, `Model.compile` and `Model.fit`. The first one is the command used to create an instance of the `Model` class, i.e. create an object that is indeed the model, the network 4.12.

```
model = Model(inputs=IN, outputs=OUT)
model.compile(loss='mean_squared_error', optimizer=my_adam_1)
model.fit(Train_96, Train_96, epochs=50, batch_size=15,
        validation_data=(Val_96,Val_96), shuffle=True)
```

```
model.compile(loss='mean_squared_error', optimizer=my_adam_1)
model.fit(Train_96, Train_96, epochs=30, batch_size=15,
        validation_data=(Val_96,Val_96), shuffle=True)
model.compile(loss='mean_squared_error', optimizer=my_adam_1)
model.fit(Train_96, Train_96, epochs=20, batch_size=15,
        validation_data=(Val_96,Val_96), shuffle=True)
```

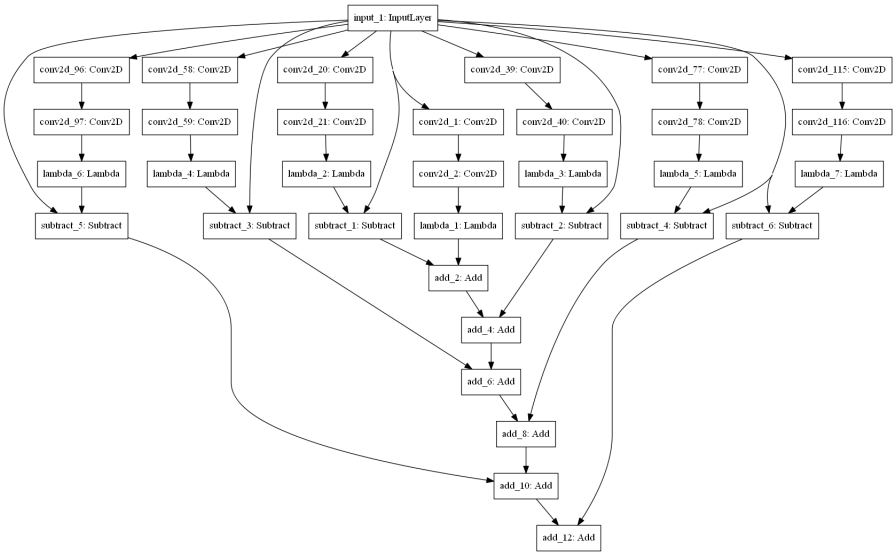


FIGURE 4.12: Graph of the network implemented in this work obtained with Keras utils

Once the model has been built, it has to be compiled, i.e. it is needed to define the loss function and the optimizer to be used.
The loss function used is the Mean Squared Error:

$$MSE = \frac{\sum_{i=1}^{n} (x_i - \hat{x}_i)^2}{n} \tag{4.2}$$

The optimizer is the *Adam (Adaptive Moment Estimation) Optimizer*, that is a variant of the Stochastic Gradient Descense that take into accounts the moments of the gradient (first, m, and second, v).
The moments are updated as follows:

$$m_t + 1 = \beta_1 m_t + (1 - \beta_1) \nabla Q(w_t) \tag{4.3}$$

$$v_t + 1 = \beta_1 v_t + (1 - \beta_2)(\nabla Q(w_t))^2 \tag{4.4}$$

Where w_t are the weights at iteration t and Q is the loss function. The bias correction of the moments is defined as

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{4.5}$$

and

$$\hat{v}_t = \frac{v_t}{1 - \beta_1^t} \tag{4.6}$$

Weights updating is:

$$w_{t+1} = w_t - \eta \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} \tag{4.7}$$

With β_1 and β_2 hyper-parameters always equal to 0.9 and 0.999, η is the learning ratio and ϵ smoothing term.

The last model method is the `fit` method. The `fit` method is actually the one that "launches" the training phase. When defining it, you have to specify the Training set, the Target set, the batch size, the number of epochs and, if you want, the validation data. Besides, you can also choose to make Keras shuffle the data, just like what we have done in this work. Like already said in the previous section, the dataset used is an augmented version of BSDS500, that is also the Target set, being this algorithm a reconstruction one and not a classification one. BSDS500 also provides a Test set and a Validation set, this last one used to improve the performances in training. The training phase has been divided into three steps: the first 50 epochs are trained with learning ratio equal to 10^-3, the epochs going from 51 to 80 use a learning ratio equal to 10^-4 and the last 20 epochs a learning ratio of 10^-5, for a total number of epochs equal to 100. The procedure of decreasing the learning ratio with the increasing of the iterations is called *annealing*

```
my_adam_1=keras.optimizers.Adam(learning_rate=0.001,
            beta_1=0.9, beta_2=0.999, amsgrad=False)
my_adam_2=keras.optimizers.Adam(learning_rate=0.0001,
            beta_1=0.9, beta_2=0.999, amsgrad=False)
my_adam_3=keras.optimizers.Adam(learning_rate=0.00001,
            beta_1=0.9, beta_2=0.999, amsgrad=False)
```

4.2.3 Performance evaluation

To evaluate the performance of a Deep Learning algorithm, particularly when it is focused on images, there are three kind of metrics used a lot in the state of the art.
The first is the accuracy, that is the most used parameter for Neural Network in general and can assume different meanings, but in this case is a simple MSE and is not the most important one. The second one is the *Peak Signal to Noise Ratio* that is the best metric to evaluate how good a compressed or reconstructed image is, always with respect to the ground-truth. In our specific case this is the most important metrics. The third and last one metric is the execution time, useful because time is an important resource and in DL for images a lot of time is needed. The times taken into account are both the Training phase time and the reconstruction time, the most import one.

PSNR

Given two images, the *Peak Signal to Noise Ratio* (PSNR) is defined as:

$$PSNR = 20 \cdot \log_{10} \left(\frac{MAX\{I\}}{\sqrt{MSE}} \right) \tag{4.8}$$

Where I is the original image, following $MAX\{I\}$ is the maximum value among pixels, and MSE, of course, is the error evaluated between I and the compressed, or in this case, reconstructed image. MSE in images becomes:

$$MSE = \frac{1}{M \cdot N} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} ||I(i,j) - R(i,j)||^2 \tag{4.9}$$

Being a fractional number, it is obvious that if the reconstruction is perfect, i.e. the MSE is equal to zero, the PSNR takes an infinite value. Moreover, PSNR is not a measure of the quality of the image itself, so if you want to have a measure of how an image looks at the eye of a man you should choose other metrics like Structural Similarity Index (SSIM).

In figure 4.13 are shown examples of reconstructed images with different values of PSNR and it is possible to see that sub-figure 4.13a with respect to 4.13b and 4.13c have not a big difference in terms of PSNR, while being a very bad

(A) PSNR=14.81 dB (B) PSNR=16.84 dB (C) PSNR=16.89 dB (D) PSNR=51.25 dB

FIGURE 4.13: Reconstructed images with different values of PSNR

quality image, where, besides, is also possible to recognize the blocks used for sampling. The PSNR has been computed using the built-in function of Scikit-image library.

Execution time

The execution time is a very important metric when speaking about Deep Learning an Compressed Sensing, above all. As already discussed in chapter 3, in section 3.1, one of the biggest problem of CS is the time that iterative algorithms take to perfectly reconstruct compressed signal, in general. For this reason, is very important for a DL-based CS reconstruction algorithm to be fast in reconstruction phase. In training phase, in fact, the time is much much longer and can take days, but is just a lump sum to pay. Once the algorithm is trained you will not care anymore about this. The reconstruction time is, instead, very important and is, in fact, the reason behind this work, as well as all the works in this field. Despite the importance of this metric, it is very important that is a non absolute metric, depending, in fact, on the CPU or GPU used. In this case a normal notebook station with i7-7500U @2.70 GHz CPU has been used for the test, while for the training phase a Nvidia QuadroP6000 has been used. (N.B. the execution time could be drastically reduced with a suited hardware, considering that normal PCs do not use all the CPU resources and that performances are afflicted by all the processes running in a PC)

Chapter 5

Numerical Assessment

This chapter will provide the numerical results obtained developing this work, analysing every results obtained, with particular attention to the most significant ones, for each different configuration of settings.

In particular, the analysis will move from describing the setting of the network to the datasets used for test it, until explaining the results. The results are based on the metrics illustrated in chapter 4. At the end of the chapter will also be illustrated a comparison with the the other works taken into account, with a collection of charts of all the results obtained for Set11 5.19.

5.1 Block size = 32

The first experiment made is based on the classical settings used by [8]. The dataset for training is always the BSDS500, the input images dimension is $(96, 96)$ and the block size for sampling is $(32, 32)$. The structure of the network, as well as the parameters like optimizer, loss function, batch size and number of epochs, never changes and it has been already discussed in chapter 4. For this value of block size we have conducted two kind of experiments, depending on the sensing matrix, that has been fixed in the first experiments and trained in the second one, like in [8].

5.1.1 Fixed sensing matrix

The very first experiment made has used a fixed sensing matrix because we wanted to create something universal, not just adapting to the dataset. The results have been very good since the beginning and will be shown for each sampling ratio.

CS Ratio = 0.01

This CS Ratio is equivalent to the first layer in the network, i.e. the Base Layer. Obviously its results are the worst and an example of reconstructed image, with a PSNR=18.8 dB, is shown in figure 5.1

(A) Ground truth Barbara (B) Reconstructed Barbara

FIGURE 5.1: Reconstructed image for block size = 32

From the reconstructed image is just possible to see a shadow of the original image, with a very bad quality. Moreover, the blocks-making procedure during sampling leaves very evident marks. The same results are obtained for every kind of image like fig. 5.2 shows for a aerial image (PSNR=14.82dB).

(A) Ground truth aerial image (B) Reconstructed aerial image

FIGURE 5.2: Reconstructed aerial image for block size = 32 and CS Ratio=0.01

The time needed for the compression and reconstruction of an image of dimension $(256, 256)$, like the ones from Set11, goes between $\sim 0.8s$ and $\sim 5s$. For a bigger image like the aerial shown in fig. 5.2 , that has dimensions $(964, 1300)$, the algorithm needs $\sim 10.27s$.

CS Ratio = 0.05

The model corresponding to this CS ratio, equivalent to the first Enhancement Layer, is the one that give the best results, both in visual terms and

PSNR terms. The "Barbara" reconstructed image 5.3 is, in fact, reconstructed with a PSNR=51.78dB, that is a very impressive result.

(A) Ground truth Barbara (B) Reconstructed Barbara

FIGURE 5.3: Reconstructed image for block size = 32 and CS Ratio=0.05

As easy to understand, with a certain value of PSNR the quality of the image is expected to be very high and almost equal to the original one. The same goes for all the other kinds of image tested, shown in figures 5.4-5.5

(A) Ground truth aerial image (B) Reconstructed aerial image

FIGURE 5.4: Reconstructed aerial image for block size = 32 and CS Ratio=0.05

The execution times for the reconstruction remain comparables with all the others for Set11, taking values between $\sim 0.8s$ and $\sim 5s$. For the aerial image, instead, the time grows reaching $\sim 36.19s$

(A) Ground truth parrots image

(B) Reconstructed parrots image

FIGURE 5.5: Reconstructed aerial image for block size = 32 and CS Ratio=0.05

CS Ratio = 0.1, 0.2, 0.3, 0.4, 0.5

In the previous sections, corresponding to CS Ratio=0.01 and CS Ratio=0.05, i.e. the worst and the best results have been showed. For this reason, in this section, will be showed the rest of the results for all the remaining CS Ratio. In figure 5.6 are displayed the reconstruction of "Barbara" image for CS Ratio from 0.1 to 0.5.

The reconstruction are better and better as we increase the CS Ratio, like we expected to be. Moreover, while PSNR grows from 20.77dB to 41dB, the execution time remain always the same. In the end, we can say that the network works very well, except for the first layer where the reconstruction is not even recognizable, with a block size equal to 32, above all with a CS Ratio=0.05

(A) Ground truth Barbara

(B) Reconstructed Barbara CS Ratio=0.1

(C) Reconstructed Barbara CS Ratio=0.2

(D) Reconstructed Barbara CS Ratio=0.3

(E) Reconstructed Barbara CS Ratio=0.4

(F) Reconstructed Barbara CS Ratio=0.5

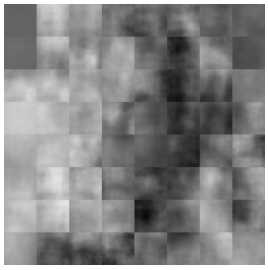
FIGURE 5.6: Reconstructed image for block size = 32 and CS Ratio=0.1-0.5

5.1.2 Trained sensing matrix

Training the sensing matrix, like we were expecting, improved the performances of the reconstruction, giving higher PSNR values for almost every CS Ratio considered (fig. 5.8), except for CS Ratio= 0.4. However, the behaviour of the network is the same, with a peak at the first EL and an almost uniform growth after while the BL always results as the worst. In figure 5.7 the result of the base layers for both trained5.7b and fixed 5.7c matrices are displayed and it is possible to see how the subject, Barbara, is becoming more recognizable, so the image is better in terms not only of PSNR but also for what concern the structure of the image itself.

(A) Ground truth Barbara

(B) Reconstructed Barbara with trained sensing matrix



(C) Reconstructed Barbara with fixed sensing matrix

FIGURE 5.7: Reconstructed Barbara for block size = 32, CS Ratio=0.05 and both trained (B) and fixed (B) sensing matrix

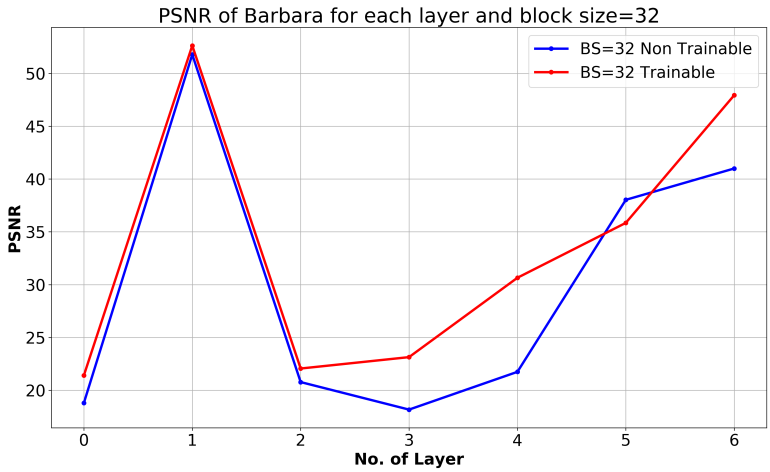


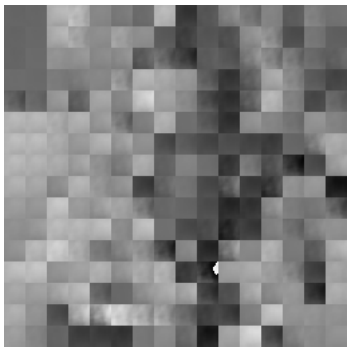
FIGURE 5.8: Behaviour of Barbara's PSNR with varying CS Ratio (Layers) and block size = 32, for both trainable and fixed sensing matrix

5.2 Block size = 16

The second experiment made in this work has been conducted with a block size equal to $(16, 16)$. Also for this case, two kind of matrices have been taken into account. Differently from the previous section, this section, as well as

the next ones, will provide only the most significant results, i.e. the best and worst. Starting from the worst ones. Just like the experiment conducted with block size = 32, also for block size = 16 the worst results are obtained for the Base Layer. Again, this is comprehensible since the sampling ratio is one over one hundred, so that taking a very good results is really difficult. In figure are displayed the reconstruction for CS Ratio=0.01 and CS Ratio=0.05 with respect to ground truth image.

(A) Ground truth Barbara image



(B) Reconstructed Barbara image, CS Ratio=0.01

(C) Reconstructed Barbara image, CS Ratio=0.05

FIGURE 5.9: Reconstructed Barbara image for block size = 16 and CS Ratio=0.01, 0.05

The PSNRs graph in figure 5.10 shows once again how, with a trained matrix we obtain a gain for almost every CS Ratio, except for the higher ones, where the fixed matrix seems to work better. The results are very similar to what we obtained for block size = 32, above all with trained matrices, with a maximum variation, with fixed sensing matrices, of 7 dB in the last layer.

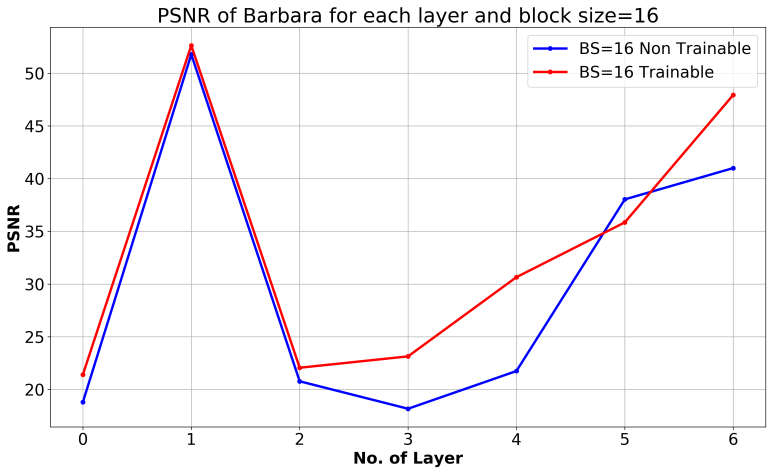


FIGURE 5.10: Behaviour of Barbara's PSNR with varying CS Ratio (Layers) and block size = 16, for both trainable and fixed sensing matrix

5.3 Block size = 48, 96

This section will discuss about the results obtained with the higher values of block size used in this work. In the state of the art this sizes are rarely used in block-based algorithms to not overload the network. Moreover, having this size of blocks in an already very small image (N.B. (96,96) are very small dimensions for image) create a very big receptive field and so the operations made from the network are negatively afflicted. Like already said, this size make more difficult the training phase too, due to the very high number of parameters that are formed (3.2). Speaking about results, we have a general decay of performances, with better results obtained only for the initial reconstruction. Despite this, the PSNR's behaviour for block size = 48 remains always the same with a peak in CS Ratio = 0.05 and a constant growth from CS Ratio = 0.1 to CS Ratio = 0.5. Something different happens for block size = 96 where the PSNR goes worse and worse after having reached a peak, even if with a low value, in the second layer as usual. Foreman and Parrot are exceptions and their PSNRs are constantly decreasing 5.19. Also in visual terms the reconstructed image are very very bad, as figure shows., with recongnizible subjects in just two cases out of seven.

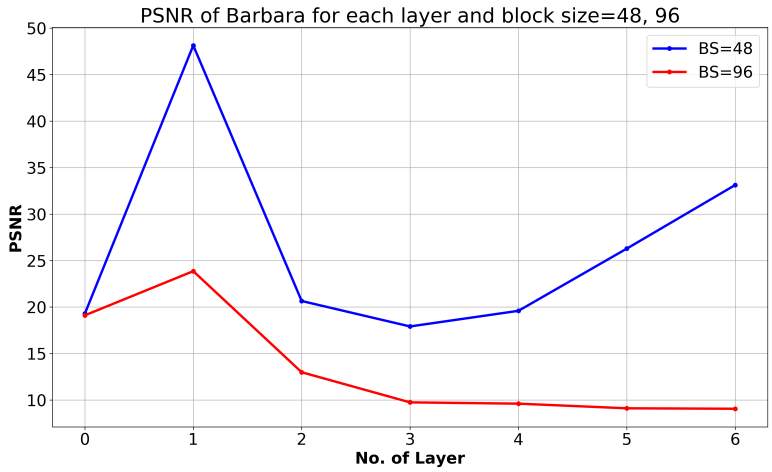


FIGURE 5.11: Behaviour of Barbara's PSNR with varying CS Ratio (Layers) and block size = 48, 96

5.4 Different dataset

The last experiment has been conducted on different kinds of image to analyse the results and the behaviour of the algorithm also on non-natural images. Results in fig. 5.12 - 5.14 demonstrate how the algorithm manage to reconstruct every kind of image with just one particular exception that are the almost monochromatic (white and black) parts of the images. The images used are an MRI, PPT3 and an aerial image, and in every case the best results are, once again, obtained with a sampling ratio of 0.05. The rest of CS Ratios taken into account works well with the aerial one, but, as just said, the white background of PPT3 and the black one of the MRI negatively afflict the performances, even if the body part in MRI is still well reconstructed. The reported results consider only the block size = 16, that is the one giving the best results, but the behaviour remain stable in the other cases, except, as it is easy to guess, for block size = 96. The average execution time for this images is in every case about 40s.

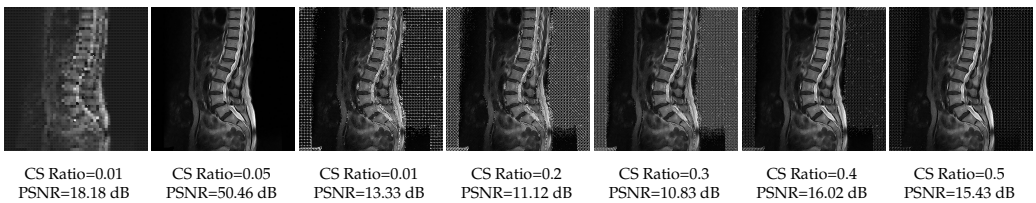
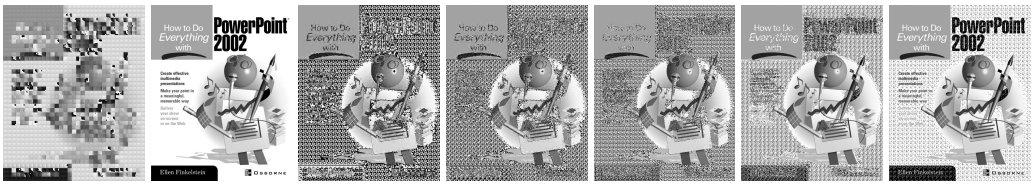


FIGURE 5.12: Reconstruction of MRI for each CS Ratio and block_size = 16, fixed sensing matrix



CS Ratio=0.01
PSNR=10.95 dB

CS Ratio=0.05
PSNR=52.51 dB

CS Ratio=0.01
PSNR=6.23 dB

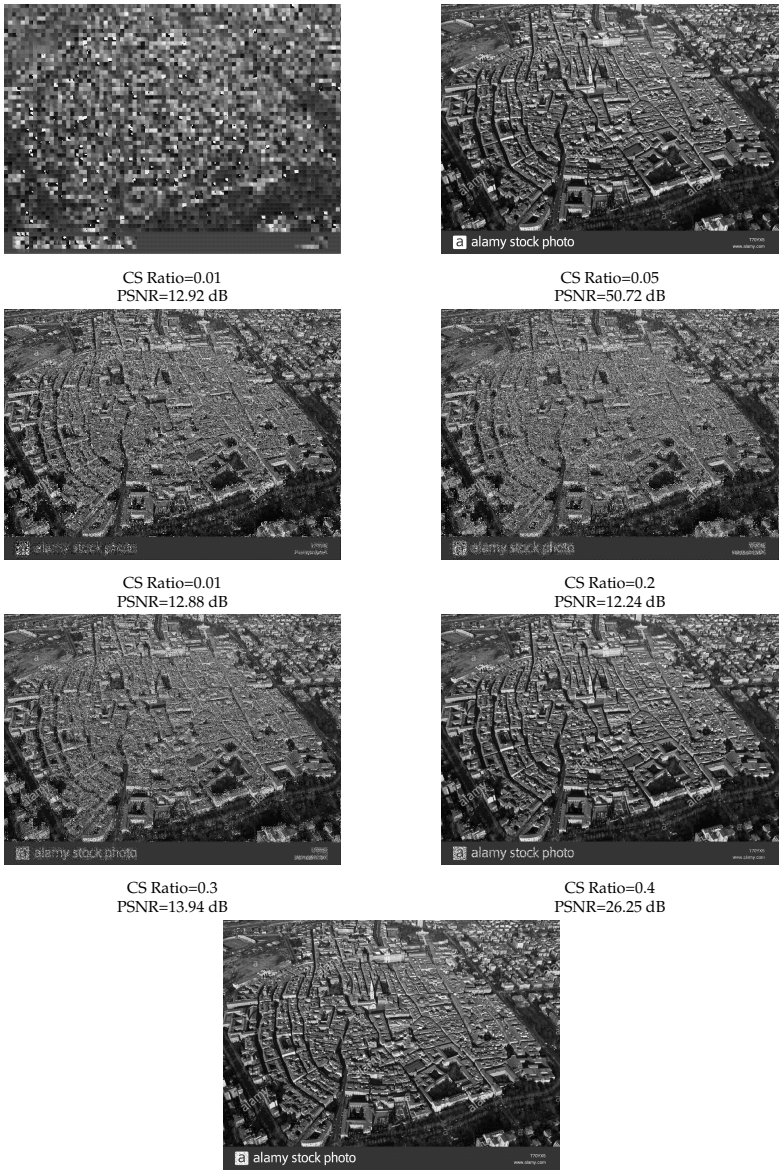
CS Ratio=0.2
PSNR=6.54 dB

CS Ratio=0.3
PSNR=6.95 dB

CS Ratio=0.4
PSNR=9.69 dB

CS Ratio=0.5
PSNR=14.23 dB

FIGURE 5.13: Reconstruction of PPT3 for each CS Ratio and block_size = 16



CS Ratio=0.01
PSNR=12.92 dB

CS Ratio=0.05
PSNR=50.72 dB

CS Ratio=0.01
PSNR=12.88 dB

CS Ratio=0.2
PSNR=12.24 dB

CS Ratio=0.3
PSNR=13.94 dB

CS Ratio=0.4
PSNR=26.25 dB

CS Ratio=0.5
PSNR=34.89 dB

FIGURE 5.14: Reconstruction of an aerial photo for each CS Ratio and block_size = 16

5.5 Comparisons with related works

In conclusion, the implemented algorithm obtains very good results for block_size = 16, 32, 48 in which, considering the best results obtained by each algorithm, overcome or at least equalizes SCS-Net and ISTA-Net$^+$. In particular for CS Ratio = 0.05 results of reconstructed images are very impressive with also an almost perfect reconstruction in case of CS Ratio=0.05 for block_size = 16, 32.

In figure 5.15 are reported the results of Butterfly reconstruction of ISTA$^+$ compared to other algorithms with CS Ratio = 0.25, while, in figure 5.16, are shown the results of the implemented algorithm for the Butterfly image with block_size =32. It's possible to notice how the reconstructed images with CS ratio = 0.05, 0.4, 0.5 overcome the results of ISTA-Net with a minimum difference of PSNR value of almost 7dB (figure 5.19).

In figure 5.17 are, instead, reported the results for the Parrot image of SCS-Net compared to other algorithms with CS Ratio = 0.1 in which it can be seen that the reconstruction reaches a PSNR value of 28.10dB. In the end, in figure 5.18 are shown the results of the implemented algorithm. Also in this case for CS Ratio = 0.05 the results are way better than SCS-Net, but for other CS Ratios, just like 0.1, the one taken into account by SCS-Net, the results are not so good staying always under 20 dB. However it has to be considered that the image taken into account, Parrot, is one of the images giving the worst results.

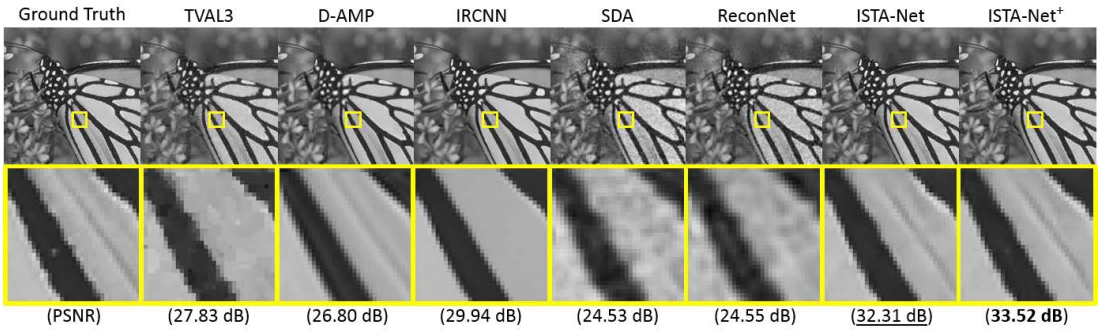


FIGURE 5.15: Butterfly ISTA

In the end we can say that, for every block size, it does not matter what CS ratio we consider, the implemented algorithm overcomes or at least almost equalizes the state of the art performances.

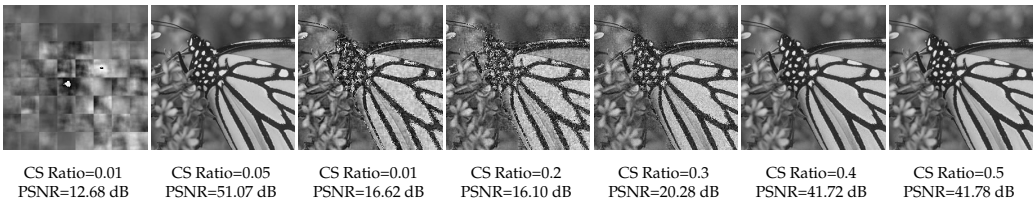


FIGURE 5.16: Reconstruction of Butterfly for each CS Ratio and block_size = 32

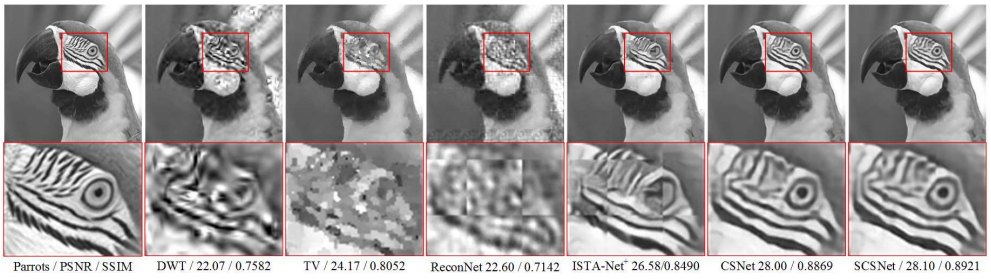


FIGURE 5.17: Parrot SCS-Net

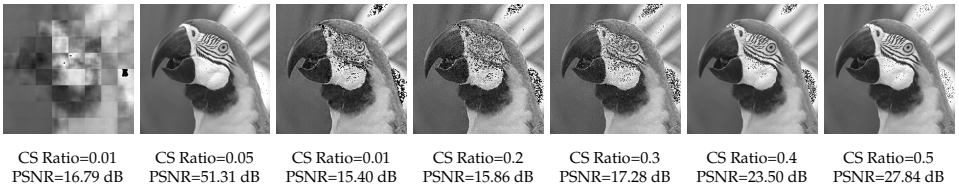


FIGURE 5.18: Reconstruction of Parrot for each CS Ratio and block_size = 32

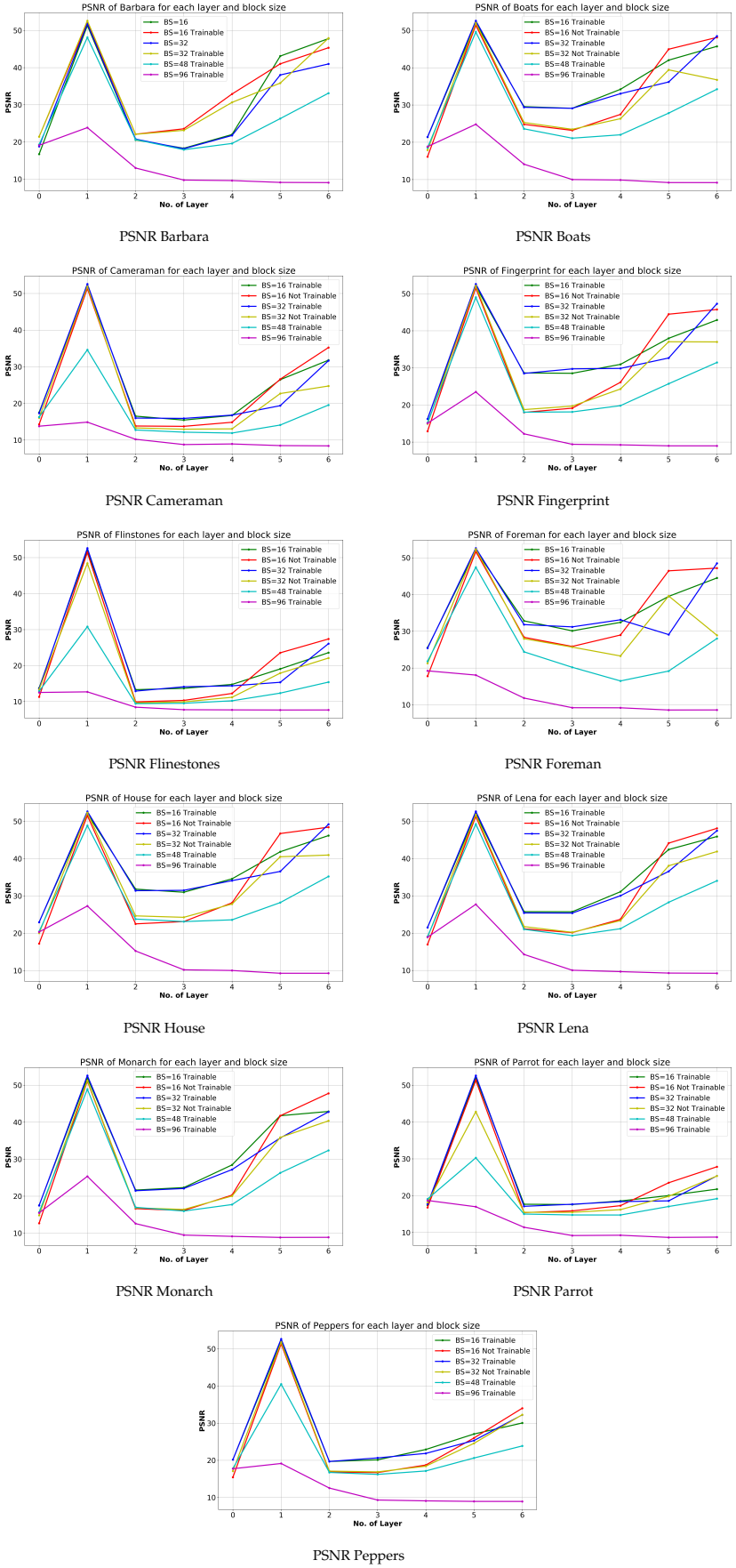


FIGURE 5.19: PSNR of each image from SET11 for each values of block_size

Chapter 6

Conclusion

Nowadays the biggest problems in image processing and, above all, transmission are the lack of resources like memory and time. For this reason, Deep Learning and Compressed Sensing have been studied and exploited more and more in image processing field. Lots of application have been developed thanks to DL from classification to reconstruction and recognizing tasks. In particular, recently, this approach has been applied to CS, to try to exploit better its own capabilities. CS is, in fact, a full of potential technique that has never been used massively due to the difficulties to acquire compressed data and principally to reconstruct quickly the compressed signal, but that can give great results in terms of speed of transmission, bandwidth employment, speed of acquisition (MRI) case and usage of memory. In this thesis DL based methods have been studied and applied to compressed sensed images to try to get a good reconstruction in a short time. In this work we have been focused above all on the sensing operation with a fixed sensing matrix and we have tried two different approaches conducting experiments considering both block based and full image based approaches. Achieved results show that a block based approach gives much better performances, but that trained sensing matrices do not give the improvements someone can expect. In particular, the obtain results manage to reach very high values of PSNR with all the tried block size, above all with a CS Ratio of 0.05, and, on the contrary, a full image based approach give not even remotely acceptable results, not so much in terms of PSNR, that however takes very low levels, as in terms of structure and quality of the image, that turns out to be always unrecognizable. Finally a comparison with the state of the art has been conducted, where the implemented algorithm showed better, or at least equal, performances with respect of the other works taken into account, reaching values of PSNR equals to almost 53 dB which is an impressive result. The block sized has resulted more incisive in afflicting the performances than training

the sensing matrix, as well as the structure of the image has to be considered very well, considering that the monochromatic part are reconstructed with some difficulties. The other important actor having major repercussions on the results is the initial reconstruction, which in the first layer helps a lot to improve performances, but then it does not manage to do the same thing for all the other layers. It could be tried to use the basic initial reconstruction for all the layers to verify if the results improve.

Bibliography

[1] E. J. Candes and M. B. Wakin. "An Introduction To Compressive Sampling". In: *IEEE Signal Processing Magazine* 25.2 (2008), pp. 21–30.

[2] R. Debray. "Vie et mort de l' image. Une histoire du regard en Occident". In: (Nov. 1992), pp. 230–231.

[3] E.Magli. In: *Compressive sensing, Theory and applications*. 2018, pp. 28–64.

[4] E.Magli. In: *Compressive sensing, Theory and applications*. 2018, p. 31.

[5] E.Magli. In: *Compressive sensing, Theory and applications*. 2018, p. 49.

[6] E.Magli. In: *Compressive sensing, Theory and applications*. 2018, p. 56.

[7] Viraj Shah and Chinmay Hegde. "Solving Linear Inverse Problems Using GAN Priors: An Algorithm with Provable Guarantees". In: (Feb. 2018).

[8] W. Shi et al. "Scalable Convolutional Neural Network for Image Compressed Sensing". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 12282–12291. DOI: 10.1109/CVPR.2019.01257.

[9] J. Zhang and B. Ghanem. "ISTA-Net: Interpretable Optimization-Inspired Deep Network for Image Compressive Sensing". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 1828–1837. DOI: 10.1109/CVPR.2018.00196.