Alma Mater Studiorum · University of Bologna

School of Science
Department of Physics and Astronomy
Master Degree in Physics

OPTIMIZATION AND APPLICATIONS OF DEEP LEARNING ALGORITHMS FOR SUPER-RESOLUTION IN MRI

Supervisor:
Prof. Gastone Castellani

Co-supervisor:
Dr. Nico Curti

Submitted by:
Mattia Ceccarelli

Academic Year 2019/2020

Abstract

The increasing amount of data produced by modern infrastructures requires instruments of analysis more and more precise, quick, and efficient. For these reasons in the last decades, Machine Learning (ML) and Deep Learning (DL) techniques saw exponential growth in publications and research from the scientific community. In this work are proposed two new frameworks for Deep Learning: Byron written in C++, for fast analysis in a parallelized CPU environment, and NumPyNet written in Python, which provides a clear and understandable interface on deep learning tailored around readability. Byron will be tested on the field of Single Image Super-Resolution for NMR imaging of brains (Nuclear Magnetic Resonance) using pre-trained models for x2 and x4 upscaling which exhibit greater performance than most common non-learning-based algorithms. The work will show that the reconstruction ability of DL models surpasses the interpolation of a bicubic algorithm even with images totally different from the dataset in which they were trained, indicating that the *generalization* abilities of those deep learning models can be sufficient to perform well even on biomedical data, which contains particular shapes and textures. Ulterior studies will focus on how the same algorithms perform with different conditions for the input, showing a large variance between results.

Contents

1 Introduction **3**
- 1.1 Neural Network and Deep Learning 4
- 1.2 Super Resolution 8
  - 1.2.1 Bicubic Interpolation 9
  - 1.2.2 Image Quality 9
- 1.3 Nuclear Magnetic Resonance 11

2 Algorithms **14**
- 2.1 Frameworks 14
- 2.2 Layers 16
- 2.3 Timing 32

3 Datasets and Methodology **37**
- 3.1 Models 37
- 3.2 Train Dataset: DIV2K 41
- 3.3 NMR Dataset 43

4 Results **46**
- 4.1 Upsample Comparisons 46
- 4.2 Scores by Angle 51
- 4.3 Error localization 54
- 4.4 Brain Extraction 59

5 Conclusions **66**

Chapter 1

Introduction

This thesis work aim is to evaluate the upsampling perfomances of pre-trained Deep Learning Single Image Super-Resolution models on Biomedical images of human brains.
The first chapter focuses on the fundamentals of the techniques named in this work with special emphasis on Deep Learning, Super Resolution and image analysis, essential for understanding the implementations of the two frameworks and the main methodologies applied during the study.
The second chapter includes the mathematical and numerical explanations of the most important algorithms implemented in Byron and NumPyNet and a brief description of the two frameworks. Moreover I will report the timing measurements againts a popular deep learning framworks called Tensorflow for the most importants layers in image analysis.
In the third chapter, I firstly describe the models in details by focusing on the reasoning the respective authors put during the construction of the architectures. Then the attention is moved to the description of the dataset used for training (DIV2K) and for testing (NMR) and how the images has been fed to the networks.
The final results are collected in the last chapter divided into subsection which answer different questions: how well DL models can reconstruct an High Resolution image starting from a Low Resolution one? How well they can *generalize* their "knowledge" on new data? How the orientation of the input influences the result? In which parts of the images the models struggle the most? The methods has been evaluated through common metrics in image analysis: *Peak Signal to Noise Ratio* (PSNR) and *Structural SIMilarity index* (SSIM). Secondarily, to remove eventual backgrounds effect from the analysis, we introduced FSL BET (Brain Extraction Tool) which is a software frequently used in NMR studies to mask images and remove uninteresting data.
In the end, I discuss the conclusions and propose possibile future developments for the work.

1.1 Neural Network and Deep Learning

A neural network is an interconnected structure of simple procedurals units, called nodes. Their functionality is inspired by the animals' brain and from the works on learning and neural plasticity of Donald Hebb [13]. From his book:

> *Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability.[...] When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased*

which is an attempt to describe the change of strenght in neural relations as a consequence of stimulations. From the so-called *Hebbian Theory* rose the first computational models such as the *Perceptron*, *Neural Networks* and the modern *Deep Learning*. The development of learning-based algorithms did not catch up with the expected results until recently, mainly due to the exponential increase in available computational resources.

From a mathematical point of view, a neural network is a composition of non-linear multi-parametric functions. During the *training phase* the model tunes its parameters, starting from random ones, by minimizing the error function (called also loss or cost). Infact, machine learning problems are just optimization problems where the solution is not given in an analytical form, therefore trough iteratives techniques (generally some kind of gradient descent) we progressively approximate the correct result.

In general, there are 3 different approaches to learning:

- **supervised** It exists a labeled dataset in which the relationship between features (input) and expected output is known. During training, the model is presented with many examples and it corrects its answers based on the expected response. Some problems tied to supervised algorithms are classification, regression, object detection, segmentation and super-resolution.
- **unsupervised** In this case, a labeled dataset does not exist, only the inputs data are available. The training procedure must be tailored around the problem under study. Some examples of unsupervised algorithms are clustering, autoencoders, anomaly detection.
- **reinforced** the model interacts with a dynamic environment and tries to reach a goal (e.g. winning in a competitive game). For each iteration of the training process we assign a reward or a punishment, relatively to the progress in reaching the objective.

This work will focus on models trained using labeled samples, therefore in a supervised environment.

Perceptron

The Perceptron (also called *artificial neuron*) is the fundamental unit of every neural network and it is a simple model for a biological neuron, based on the works of Rosenblatt [24]. The *perceptron* receives N input values $x_1, x_2, ...x_N$ and the output is just a linear combination of the inputs plus a bias:

$$y = \sigma(\sum_{k=1}^{N} w_k x_k + w_0) \tag{1.1}$$

where σ is called *activation function* and $w_0, w_1, ...w_N$ are the trainable weights.

Originally, the activation function was the *Heaviside step function* whose value is zero for negative arguments and one for non-negative arguments:

$$H(x) := \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases} \tag{1.2}$$

In this case the perceptron is a *linear discriminator* and as such, it is able to learn an hyperplane which linearly separates two set of data. The weights are tuned during the training phase following the given update rule, usually:

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n + \eta(t - y)\boldsymbol{x} \tag{1.3}$$

where η is the learning rate ($\eta \in [0, 1]$) and t is the true output. If the input instance is correctly classified, the error $(t - y)$ would be zero and weights do not change. Otherwise, the hyperplane is moved towards the misclassified example. Repeating this process will lead to a convergence only if the two classes are linearly separable.

Fully Connected Structure

The direct generalization of a simple perceptron is the *Fully Connected Artificial Neural Network* (or *Multy Layer Perceptron*). It is composed by many Perceptron-like units called nodes, any one of them performs the same computation as formula 1.3 and *feed* their output *forward* to the next layer of nodes. A typical representation of this type of network is shown in figure 1.1.

While the number of nodes in the input and output layers is fixed by the data under analysis, the best configuration of hidden layers is still an open problem.

The mathematical generalization from the perceptron is simple, indeed given the i-th layer its output vector $\boldsymbol{y}_i$ reads:

$$\boldsymbol{y}_i = \sigma(W_i \boldsymbol{y}_{i-1} + \boldsymbol{b}_i) \tag{1.4}$$

where W_i is the weights matrix of layer i and $\boldsymbol{b}_i$ is the i-th bias vector, equivalent to w_0 in the perceptron case. The output of the i-th layer becomes the input of the next one until the output layer yields the network's answer.

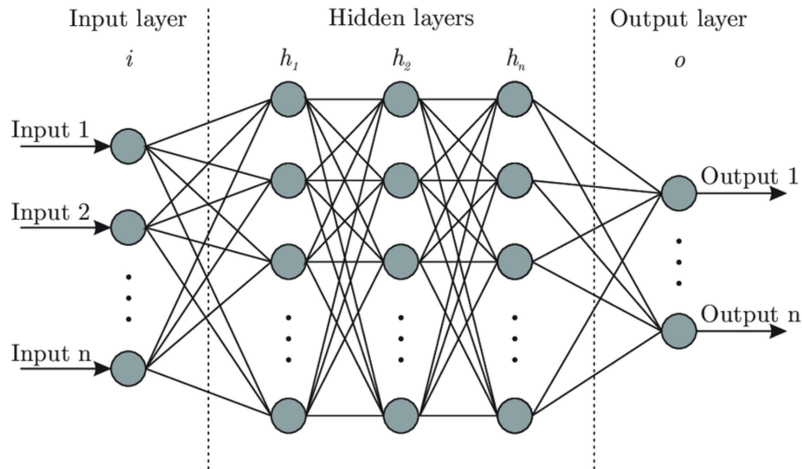


Figure 1.1: *A common representation of a neural network: a single node works as the perceptron described before. The network is composed by the input layer, the hidden layers and the output layer. The depth of the network is determined by the number of hidden layers.*

As before, σ is the activation function which can be different for every node, but it usually differs only from layer to layer. How to chose the best activation function is yet to be understood, and most works rely on experimental results.

In a supervised environment, the model output is compared to the desired output (*truth*) by means of a cost function. An example of cost function is the sum of squared error:

$$C(W) = \frac{1}{N}\sum_{j=1}^{N}(y_j - t_j)^2 \tag{1.5}$$

where N is the dimensionality of the output space. C is considered as a function of the model's weights only since input data and *truth labels* t are fixed.

Those architectures are *universal approximators*, that means given an arbitrarly complex function, there is a fully connected neural network that can approximate it.

This type of network is called *feed forward* because the information flows directly from the input to the output layer: however, it exists a class of models called *Recurrent* where this is not the case anymore and feedback loops are possible, but they are outside the scope of this work.

Gradient Descent

To minimize the loss function an update rule for the weights is needed. Given a cost funtion $C(w)$, the most simple one is the gradient descent:

$$w \leftarrow w - \eta \nabla_w C \tag{1.6}$$

The core idea is to modify the parameters by a small step in the direction that minimizes the error function The lenght of the step is given by the *learning rate* η, which is a hyperparameter chosen by the user, while the direction of the step is given by $-\nabla_w C$, which point towards the steepest descent of the function landscape.

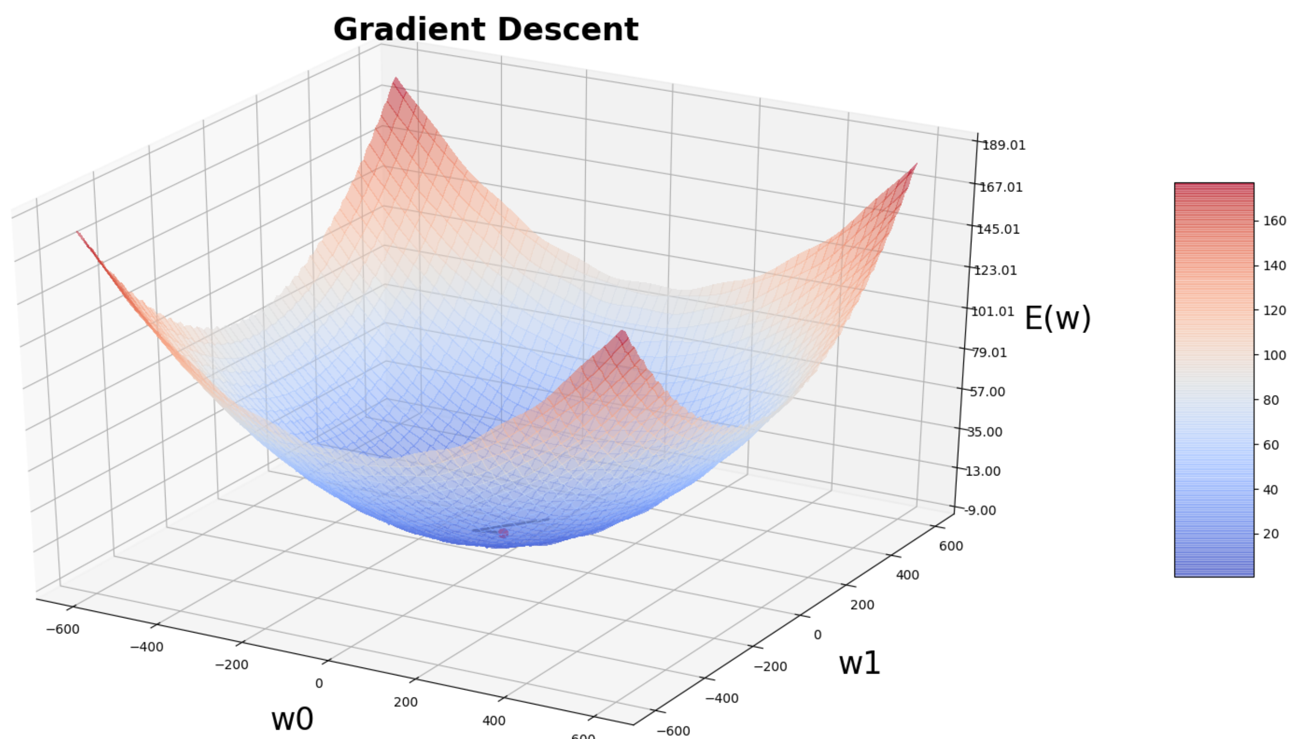


Figure 1.2: *Visual example of gradient descent for a model with 2 weights. The idea is to modify the weights to follow the direction of the steepest descent for the landscape of the error function*

The speed at which the algorithm converges to a solution and the precision of said solution are greatly influenced by the update rule. More complex and efficient update rules do exist, but they follow the same idea as the gradient descent.

Error Back Propagation

The most common algorithm used to compute the updates to weights in the learning phase is the *Error Back Propagation*. Consider a Neural Network with L total layers, each with a weight matrix W_l with $l = 1, 2, \dots L$. Given a differentiable cost function C, which depends from $W_1, W_2, \dots W_L$, let's define:

$$\boldsymbol{z}_l = W_l \boldsymbol{y}_{l-1} + \boldsymbol{b}_l \tag{1.7}$$

$$\boldsymbol{a}_l = \sigma(\boldsymbol{z}_l) \tag{1.8}$$

respectively the *de-activated* and *activated* output vectors of layer l with N neurons (or N outputs.) and define:

$$\boldsymbol{\delta}_l = (\frac{\partial C}{\partial z_l^1}, \dots, \frac{\partial C}{\partial z_l^N}) \tag{1.9}$$

the vector of errors of layer l. Then we can write the 4 equations of back propagation for the fully-connected neural network [19]:

$$\boldsymbol{\delta}_L = \nabla_a C \odot \sigma'(\boldsymbol{z}_L) \quad \text{The network error vector} \tag{1.10}$$

$$\boldsymbol{\delta}_l = (W_{l+1}^T \boldsymbol{\delta}_{l+1}) \odot \sigma'(\boldsymbol{z}_l) \quad \text{The error vector for layer } l \tag{1.11}$$

$$\frac{\partial C}{\partial b_l^j} = \delta_l^j \quad \text{The j-th bias update} \tag{1.12}$$

$$\frac{\partial C}{\partial w_l^{jk}} = a_{l-1}^k \delta_l^j \quad \text{The update for the weight indexed } j,k \tag{1.13}$$

where $\odot$ is the Hadamard's product. Those equations can be generalized for others kind of layers, as I will show in the next chapters.

The full training algorithm is:

1 Define the model with random parameters
2 Compute the output for one of the inputs
3 Compute the loss function $C(W)$ and the gradients $\frac{\partial C}{\partial w_l^{jk}}$ and $\frac{\partial C}{\partial b_l^j}$ for each l.
4 Updates the parameters following the update rule,
5 Iterate from step 2 until the loss is sufficiently small

1.2 Super Resolution

The term Super-Resolution (SR) refers to a class of techniques that enhance the spatial resolution of an image, thus converting a given low resolution (LR) image to a corresponding high resolution (HR) one, with better visual quality and refined details. Image super-resolution is also called by other names like image scaling, interpolation, upsampling and zooming [4]. Super resolution can also refers to its "hardware" (and best-known) implementation, the *super resolution microscopy*, which aim is to overcome the diffraction limit: indeed, the development of super-resolved fluorescense microscopy won a Nobel price in chemistry in 2014, though its technicalities reside outside the scope of this work, which focused on its numerical counterpart.

As described before, the training of a supervised model happens by means of examples: in the case of classification the network is presented with many couples *features-label* that compose the *train set*. The objective is to find the correct labels for a set of samples never saw before called *test set*.

For digital images, the *features* are the pixels which compose a 2 dimensional or 3 dimensional (for RGB picture) grid-like structure, the label is usually represented as 1

dimensional vector as large as the binary representation of the number of classes the model is supposed to discern: a neural network produces a map between a very large *features space* and a smaller one.

This behaviour is slightly different for Super-Resolution: indeed, when training a SR model we are talking about *image-to-image* processing and as such, both the features space and the labels are images. The dataset is built from a single series of high resolution (HR) images which are downsampled to obtain the low resolution (LR) counterpart: the couples LR-HR are fed to the network respectively as input and label just like in a classification problem; this time though, the network will map a smaller feature space into a larger one.

The models I'm going to use in this work are trained on images downsampled using the *bicubic interpolation*.

1.2.1 Bicubic Interpolation

The *Bicubic interpolation* is a common algorithm used in image analysis either to downsample or upsample an image. This operation is also called *re-scaling* and its purpose is to interpolate the pixel values after a resize of the image, respectively after shrinking or expanding it, e.g as a consequence of zooming. The name comes from the highest order of complexity of the operation used in the algorithm, which is a cubic function. Given a pixel, the interpolation function evaluates the 4 pixel around it by applying a filter defined as:

$$k(x) = \frac{1}{6}\begin{cases} (12-9B-6C)|x|^3 + (-18+12B+6C)|x|^2 + (6-2B) & \text{if } |x| < 1 \\ (-B-6C)|x|^3 + (6B+30C)|x|^2 + (-12B-48C)|x| + (8B+24C) & \text{if } 1 \leq |x| < 2 \\ 0 & \text{otherwise} \end{cases} \tag{1.14}$$

where x identifies each pixel below the filter. Common values used for the filter parameters are $B = 0$ and $C = 0.75$ (used by OpenCV library) or $B = 0$ and $C = 0.5$ used by Matlab and Photoshop. The scale factor of the down/up sampling can assume different values according to the user needs; for this work, I used an upsampling factor of $\times 2$ and $\times 4$ and the algorithm is from the Python version of the library OpenCV [6]. The main aims of SR algorithms are to provide a better alternative to standard upsampling and obtain a better quality image both from a qualitative (visual perception) and a quantitative poin of view.

1.2.2 Image Quality

While the human eye is a good qualitative evaluator, it is possible to define different quantitative measures between two images to quantify their similiraties.

PSNR

One of the most common in Image Analysis is the *Peak Signal To Noise Ratio* or PSNR. It is usually employed to quantify the recostruction capabilities of an algorithm given a lossy compression, w.r.t the original image. The mathematical espression reads:

$$PSNR = 20 \cdot \log_{10}(\frac{max(I)}{MSE}) \tag{1.15}$$

where $max(I)$ is the maximum available value for the image I, namely 1 for floating point representation and 255 for an integer one. MSE is the *Mean Squared Error*, which is a common metrics in data analysis used to quantify the mean error of a model. It is defined as:

$$MSE = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} (I(i,j) - K(i,j))^2 \tag{1.16}$$

where H and W are the spatial dimensions of the original image I and the recostruction K. The metric can be generalized to colored image by simply adding the depth (RGB channel) dimension.

Even though an higher PSNR generally means an higher recostruction quality, this metric may performs poorly compared to other quality metrics when it comes to estimate the quality as perceived by human eyes. An increment of 0.25 in PSNR corresponds to a visible improvement.

SSIM

Another common metric is the *Structural SIMilarity index* or SSIM. It has been developed to evaluate the structural similraties between two images, while incorporating important perceptual phenomena, including luminance and contrast terms. For that, it should be more representative of the qualitative evaluation as seen by humans. The SSIM index is defined as:

$$SSIM(I,K) = \frac{1}{N} \sum_{i=1}^{N} SSIM_i(x_i, y_i) \tag{1.17}$$

Where N is the number of windows in the images, usually of size 11×11 or 8×8. For every box, the index is:

$$SSIM(x_i, y_i) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{({\mu_x}^2 + {\mu_y}^2 + c_1)({\sigma_x}^2 + {\sigma_y}^2 + c_2)}, \tag{1.18}$$

where x and y are two equally-sized regions in two different images, μ is the average value in the region, σ^2 is the variance, σ_{xy} is the covariance between the regions and c_1 and c_2 are two constants to stabilize the division.

Both SSIM and PSNR can be useful in Deep Learning applications as target functions or as post-training quality measures. To compute PSNR and SSIM I used the function of the Python library *scikit image* [28], for their precision, efficency and ease to use.

SR pre-trained models will be evaluated in their reconstruction capabilities against the bicubic interpolation using as benchmark an available dataset of NMR images of brain. I'd like to point out that the deep learning architecture tested in this work are trained on general purpose datasets which are very different from biomedical pictures available: the first problem is that MRI images are single channeled (gray-scaled) as opposed to the RGB images which those models are trained on, however this can be easily solved by artificially add depth by concatenating the same image 3 times; by doing so, the models elaborate tree different outputs that can be compared against each others. The second issue is that the models never had a chance to learn the particular shapes contained in animals' brain: although that could be seen as a major drawback, their generalization capability should be sufficient to perform well even outside their optimal "environment". The datasets will be discussed in later chapters.

1.3 Nuclear Magnetic Resonance

The term NMR identifies an experimental technique called Nuclear Magnetic Resonance. It has been indipendently developed by two reserch groups led Felix Bloch and Edward Purcell, both awarded with the Nobel prize in Physics in 1952. Initially, NMR was tied to studies in fundamental physics and particularly to solid state physics: its theoretical and technological evolution in the years allowed numerous applications also in the biological and medical fields. Differently from other invasive techniques from nuclear medicine or radiology, which employ ionising radiations dangerous for the organism, in NMR the only source of energy administration is represented by two types of electromagnetic fields: static and radio-frequency. By applying those fields to nuclei which posses magnetic properties it is possible to analyze the macroscopic structure of the sample. The most used nuclei are $^1H, ^2H, ^{31}P, ^{23}Na, ^{14}N, ^{13}C, ^{19}F$, which is really advantegeous in the biological field, given that the subjects are most likely rich in H_2O, therefore in ^1H. This allows to detect signals of large intensity from the samples and measure a high ratio signal/noise even with few acquisitions.

As stated before, NMR is applied to nuclei which posses magnetic properties, therefore they have an angular momentum $\boldsymbol{I}$ (spin) associated with a magnetic momentum $\boldsymbol{\mu} = \gamma(\frac{h}{2\pi})\boldsymbol{I}$ where γ is the gyromagnetic ratio, which depends from the nucleus.

Considering an ensemble of ^1H with $I = \frac{1}{2}$, if a static magnetic field $\boldsymbol{B}_0 = H_0\hat{z}$ is applied, the magnetic momenta of the nucleus will orientate themselves along the parallel or anti-parallel direction of the fields, assuming two discrete values of energy. According to Boltzmann statistic, the majority of nuclei will orientate in such a way to minimize the energy (parallel to $\boldsymbol{B}_0$).

The energy difference between the two level is given by:

$$\Delta E = \gamma \left(\frac{h}{2\pi} B_0 \right) = \left(\frac{h}{2\pi} \omega_0 \right) \tag{1.19}$$

where ω_0 is the Larmor Frequency. The energy that must be given to perturb the system from its equilibrium condition follow the resonance condition 1.19. The perturbation is represented by a radio-frequency electromagnetic field $\boldsymbol{B}_1$ perpendicular to $\boldsymbol{B}_0$, oscillating at the Larmor frequency of the system ω_0. After removal of the perturbation, it is possible to measure the relaxation times T_1 and T_2 of the longitudinal and trasversal components of the nuclei magnetization.

For the nuclei, the temporal evolution of $\boldsymbol{\mu}$ under the influence of the static fields $\boldsymbol{B}_0$ is given by:

$$\frac{d\boldsymbol{\mu}}{dt} = \gamma \boldsymbol{\mu} \times \boldsymbol{B}_0 \tag{1.20}$$

and by separating the three components it yields:

$$\frac{d\mu_x}{dt} = \gamma \mu_x B_0 \tag{1.21}$$

$$\frac{d\mu_y}{dt} = \gamma \mu_y B_0 \tag{1.22}$$

$$\frac{d\mu_z}{dt} = 0 \tag{1.23}$$

which highlights how the magnetic moment performs a precession around the z axis with frequency $\omega_0 = \gamma B_0$

The radio-frequency magnetic field $\boldsymbol{B}_1$, with $B_1 << B_0$ and frequency ω is perpendicular to $\boldsymbol{B}_0$ lying in the xy plane and can be obtained by an oscillating fields generated by a coil traversed by a radio-frequency current.

In the most general case of nuclear magnetization, if the system is influenced by field $\boldsymbol{B}_0$ and $\boldsymbol{B}_1$ it is in a non-equilibrium condition described by the *Block Equations* for each components:

$$\frac{dM_z}{dt} = -\frac{M_z(t) - M_z(0)}{T_1} \qquad \text{longitudinal relaxation} \tag{1.24}$$

$$\frac{dM_{xy}}{dt} = -\frac{M_{xy}(t) - M_{xy}(0)}{T_2} \qquad \text{trasversal relaxation} \tag{1.25}$$

Their integration brings:

$$M_z(t) = M_z(0) exp\left(\frac{-t}{T_1}\right) + M_z(0)\left(1 - exp\left(\frac{-t}{T_1}\right)\right) \tag{1.26}$$

$$M_{xy}(t) = M_{xy}(0) exp\left(\frac{-t}{T_2}\right) \tag{1.27}$$

Bloch's equations are fundamental for the choice of the sequence of excitation and subsequent acquisition and elaboration of the signal. Once the perturbation action ends, its possible to follow the de-excitation of the macroscopic magnetization M , which tends to realign to the field $\boldsymbol{B}_0$. The signal produced by the variation of M is measured by an induction electromagnetic coil around the sample in an ortogonal direction w.r.t the static field. The NMR signal, called FID (Free Induction Decay), is approximately monochromatic and oscillates at Larmor frequency, decaying exponenetially as a functio of T_2. For Image formation (MRI), excitation sequences are opportunely chosen in such a way to emphasize the dipendence of FID from three parameters: protonic density ρ, T_1 and T_2.
MRI can be differentiated in two types: T1-weighted sequences and T2-weighted sequences, which show different information. The former are considered the most "anatomical" and result in images that most closely appoximate the appearences of tissues: fluid have low signal intensity (black), muscle and gray-matter has a intermediate signal intensity (grey) and fat and white-matter have a high signal intensity (white).
The latter instead, have a high signal intensity for fluid an fat, an intermediate intensity for muscle and grey-matter, and low intensity for white-matter, which appears dark-ish. In figure 1.3 is shown a comparison between the two:

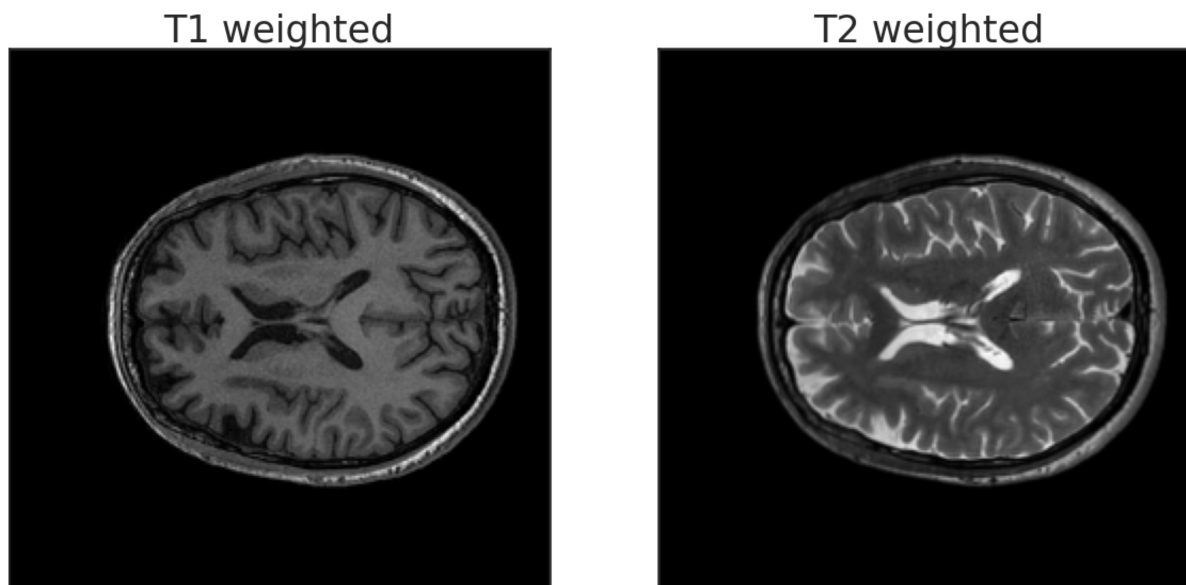


Figure 1.3: *Comparison between a T1-weighted slice (left) and a T2-weighted slice (right) for the same patient*

Chapter 2

Algorithms

A wide range of documentations and implementations have been written on the topic of Deep Learning and it is more and more difficult to move around the different sources. In recent years, leaders in DL applications became the multiple open-source Python libraries available on-line as Tensorflow [2], Pytorch [22] and Caffe [16]. Their portability and efficiency are closely related on the simplicity of the Python language and on the simplicity in writing complex models in a minimum number of code lines. Only a small part of the research community uses deeper implementation in C++ or other low-level programming languages and between them should be mentioned the darknet project of Redmon J. et al. which has created a sort of standard in object detection applications using a pure Ansi-C library. The library was developed only for Unix OS but in its many branches (literally *forks*) a complete porting for each operative system was provided. The code is particularly optimized for GPUs using CUDA support, i.e only for NVidia GPUs. It is particularly famous for object detection applications since its development is tightly associated to an innovative approach at multi-scale object detections called YOLO (*You Only Look Once*), that recently reached its fourth release [5]. The libraries built during the develompment of this thesis are all inspired by the efficiency and modularity of darknet and make an effort to not only replicate but expand on their work, both in performances, functionalities and solved issues.

In this section I will describe the mathematical background of these models and to most theoretical explanation discuss the numerical problems associated, tied to the development of two new libraries: NumPyNet [8] and Byron [9].

2.1 Frameworks

NumPyNet is born as an educational framework for the study of Neural Network models. It is written trying to balance code readability and computational performances and it is enriched with a large documentation to better understand the functionality of each script. The library is written in pure Python and the only external library used is Numpy [21] (a base package for the scientific research). As I will show in the next sections, Numpy

allows a relatively efficient implementation of complex algorithms by keeping the code as similar as possible to the mathematic computations involved.

Despite being supplied by wide documentations, it is often difficult for novel users to move around the many hyper-links and papers cited in all common libraries. NumPyNet tries to overcome this problem with a minimal mathematical documentation associated to each script and a wide range of comments inside the code.

An other "problem" to take into account is associated to performances. On one hand, libraries like Tensorflow are certainly efficient from a computational point-of-view and the numerous wraps (like *Keras*) guarantee an extremely simple user interface. On the other hand, the deeper functionalities of the code and the implementation strategies used are unavoidably hidden behind tons of code lines. In this way the user can perform complex computational tasks using the library as black-box package. NumPyNet wants avoid this problem using simple Python codes, with extreme readability also for new users, to better understand the symmetry between mathematical formulas and code. The simplicity of this library allows us to give a first numerical analysis of the model functions and, moreover, to show the results of each function on an image to better understand the effects of their applications on real data. Each NumPyNet function was tested against the equivalent Tensorflow implementation, using an automatic testing routine through PyTest [20]. The full code is open-source on the Github page of the project. Its installation is guaranteed by a continuous integration framework of the code through Travis CI for Unix environments and Appveyor CI for Windows OS. The library supports Python versions ≥ 2.7.

As term of comparison we discuss the more sophisticated implementation given by the Byron library. Byron (*Build YouR Own Neural network*) library is written in pure C++ with the support of the modern standard C++17. We deeply use the C++17 functionality to reach the better performances and flexibility of our code. What makes Byron an efficient alternative to the competition is the complete multi-threading environment in which it works. Despite the most common Neural Network libraries are optimized for GPU environments, there are only few implementations which exploit the full set of functionalities of a multiple CPUs architecture. This gap discourages multiple research groups on the usage of such computational intensive models in their applications. Byron works in a fully parallel section in which each single computational function is performed using the entire set of available cores. To further reduce the time of thread spawning, and so optimize as much as possible the code performances, the library works using a single parallel section which is opened at the beginning of the computation and closed at the end.

The Byron library is released under MIT license and publicly available on the Github page of the project. The project includes a list of common examples like object detection, super resolution, segmentation. The library is also completely wrapped using Cython to enlarge the range of users also to the Python ones. The complete guide about its installation is provided; the installation can be done using CMake, Make or Docker and the Python version is available with a simple setup.py. The testing of each function is

performed using Pytest framework against the NumPyNet implementation (faster and lighter to import than Tensorflow) [7].

2.2 Layers

As described above, a neural network can be considered as a composition of function: for this reason every Deep Learning framework (e.g. Keras/Tensorflow, Pytorch, Darknet) implement each function as an independent object called *Layer*. In Byron and NumPyNet, each layer contains at least 3 methods:

- **forward** the forward method compute the output of the layer, given as input the previous output.
- **backward** the backward method is essential for the training phase of the model: indeed, it computes all the updates for the layer weights and backpropagates the error to the previous layers in the chain.
- **update** the update method applies the given update rules to the layer's weights.

By stacking different kind of layers one after another, it is possible to build complex models with tens of millions of parameters. For the purposes of this work, I'm going to describe layers used in super resolution, however, Byron is developed also for different applications (object detection, classification, segmentation, style transfer, natural language processing etc...) and as such, many more layers are available.

Convolutional Layer

A Convolutional Neural Network (CNN) is a specialized kind of neural network for processing data that has known grid-like topology [11], like images, that can be considered as a grid of pixels. The name indicates that at least one of the functions employed by the network is a convolution. In a continuos domain the convolution between two functions f and g is defined as:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{2.1}$$

The first function f is usually referred to as the input and the second function g as kernel. For Image Processing applications we can define a 2-dimensional discrete version of the convolution in a finite domain using an image I as input and a 2 dimensional kernel k:

$$C[i,j] = \sum_{u=-N}^{N} \sum_{v=-M}^{M} k[u,v] \cdot I[i-u, j-v] \tag{2.2}$$

where $C[i,j]$ is the pixel value of the output image and N, M are the kernel dimensions. Practically speaking, a convolution is performed by sliding a kernel of dimension $N \times M$ over the image, each kernel position corresponds to a single output pixel, the value of

which is calculated by multiplying together the kernel value and the underlaying pixel value for each cell of the kernel and summing all the results, as shown in figure 2.1:

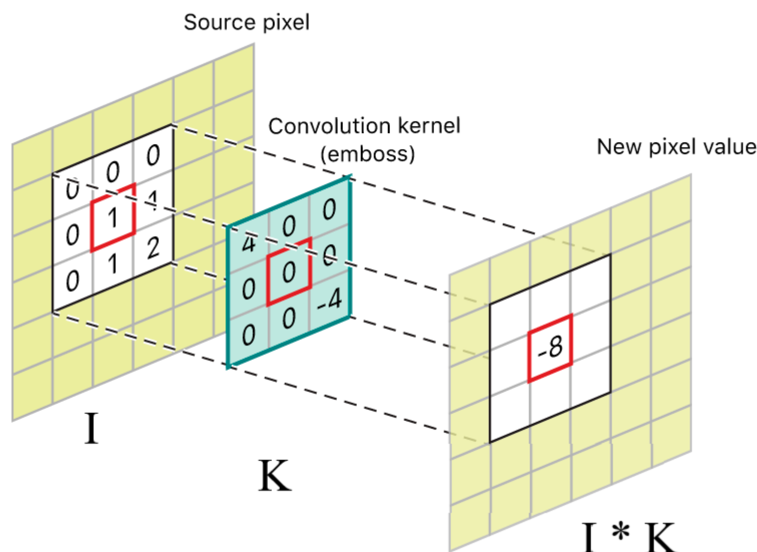


Figure 2.1: *Visual representation of a convolution of an Image I with a kernel of size 3*

The convolution operation is also called *filtering*. By chosing the right kernel (filter) it is possible to highlight different features. For this reason the convolution operation is commonly used in image analysis: some of the most common applications are denoising, edge detection and edge enhancement.

The convolutional layer (CL) object is the most used layer in DL image analysis, therefore its implementation must be as efficient as possible. Its purpose is to perform multiple (sometimes thousands) convolution over the input to extract different high-level features, which are compositions of many low-level attributes of the image (e.g edges, simple shapes). In the brain/neuron analogy, every entry in the output volume can also be interpreted as an output of a neuron that looks at only a small region, the neuron's *receptive field* in the input and shares parameters with all the neuron spatially close. As more CLs are stacked, the receptive field of a single neuron grows and with that, the complexity of the features it is able to extract. The local nature of the receptive field allows the models to recognize features regardless of the position in the images. In other words, it is independent from translations [11].

The difference from a traditional convolutional approach is that instead of using pre-determined filters, the network is supposed to learn its own. A CL si defined by the following parameters:

- **kernel size**: it is the size of the sliding filters. The depth of the filters is decided by the depth of the input images (which is the number of channels.). The remanining 2 dimensions (width and height) can be indipendent from one another, but most implementations require squared kernels.

- **strides**: defines the movement of the filters. With a low stride (e.g. unitary) the windows tends to overlap. With high stride values we have less overlap (or none) and the dimension of the output decrease.
- **number of filters**: is the number of different filters to apply to the input. It also indicates the depth of the output.
- **padding**: is the dimension of an artificial enlargement of the input to allow the application of filters on borders. Usually, it can be interpreted as the number of rows/columns of pixel to add to the input, however some libraries (e.g Keras) consider it only as binary: in case is true, only the minimum number of rows/columns are appended to keep the same spatial dimension.

Given the parameters, it is straightforward to compute the number of weights and bias needed for the initialization of the CL: indeed, suppose an image of dimensions (H, W, C) slided by n different 3-D filters of size (k_x, k_y) with strides (s_x, s_y) and padding p, then:

$$\#weights = n \times k_x \times k_y \times C \tag{2.3}$$

$$\#bias = n \tag{2.4}$$

Note that the number of weights does not depend on the input spatial size but only on its depth. It is important because a fully convolutional network can receives images of any size as long as they have the correct depth. Moreover, using larger inputs do not requires more weights, as is the case for fully connected structure.

The output dimensions are (out_H, out_W, n) where:

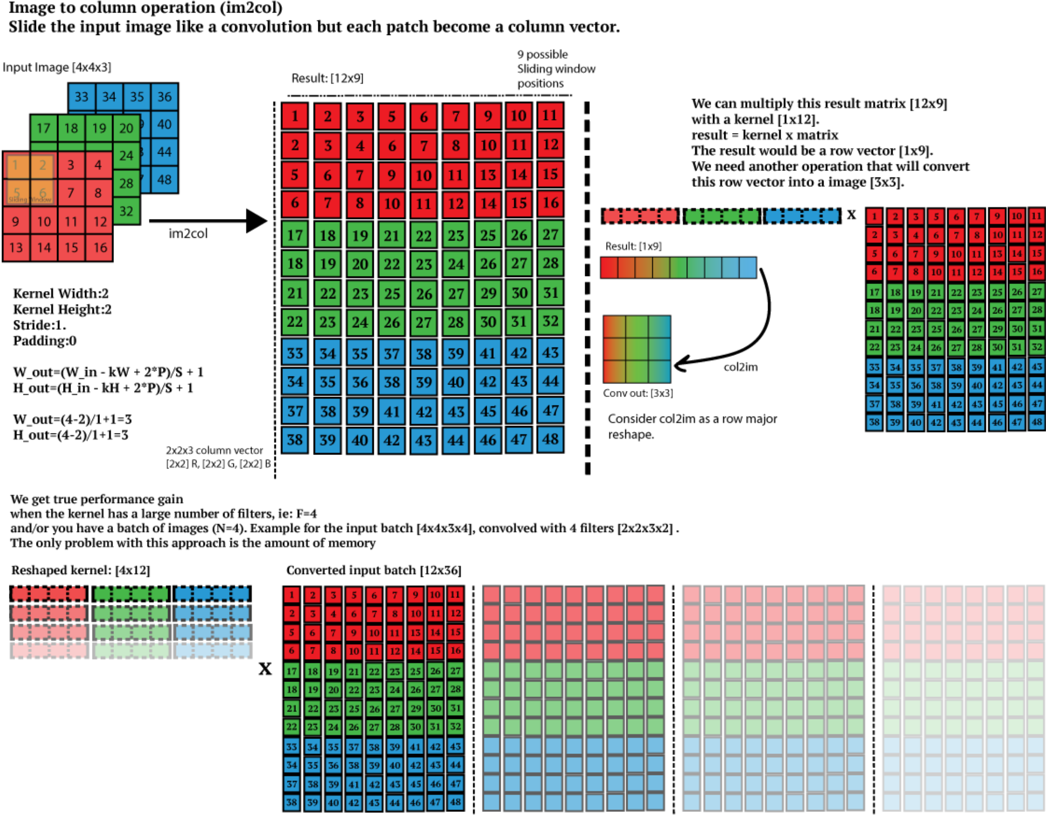
$$out_H = \lfloor \frac{H - k_x + p}{s_x} \rfloor + 1 \tag{2.5}$$

$$out_W = \lfloor \frac{W - k_y + p}{s_y} \rfloor + 1 \tag{2.6}$$

Even if the operation can be implemented as described above in equation 2.2, this is never the case: it is certainly easier but also order of magnitude slower than more common algorithms. A huge speed up in performances is given by realising that a discrete convolution can be viewed as a single matrix multiplication. The first matrix has as rows each filters of the CL, while the second matrix has as columns every windows of the image traversed by the kernels, as shown in figure 2.2.

This re-arrengement is commonly called im2col. The main downside is that a lot more memory is needed to store the newly arranged matrix. The larger the number of kernels, the higher is the time gain of this implementation over a naive one.

Another important optimization comes from linear algebra considerations and is called *Coppersmith-Winograd algorithm*, which was designed to optimize the matrix product. Suppose we have an input image of just 4 elements and a 1-D filter mask with size 3:



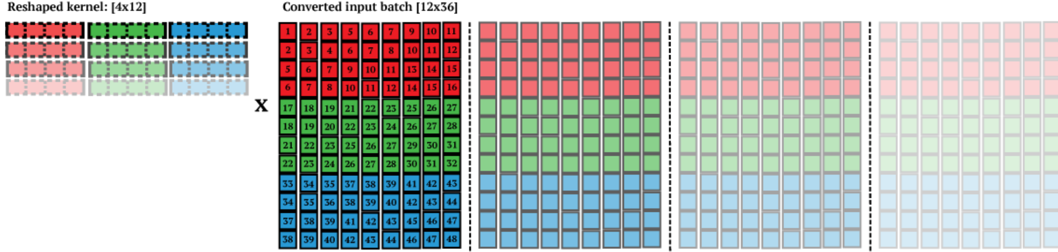


Figure 2.2: *Scheme of the im2col algorithm using a* $2 \times 2 \times 3$ *filter with stride 1 on a* $4 \times 4 \times 3$ *image. The matrix multiplication is between a* $n \times 12$ *and a* 12×9 *matrixes.*

$$\text{img} = \begin{bmatrix} d0 & d1 & d2 & d3 \end{bmatrix} \qquad \text{weights} = \begin{bmatrix} g0 & g1 & g2 \end{bmatrix} \tag{2.7}$$

we can now use the im2col algorithm previously described and reshape our input image and weights into

$$\text{img} = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix}, \qquad \text{weights} = \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} \tag{2.8}$$

given this data, we can simply compute the output as the matrix product of this two matrices:

$$\text{output} = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} d0 \cdot g0 + d1 \cdot g1 + d2 \cdot g2 \\ d1 \cdot g0 + d2 \cdot g1 + d3 \cdot g2 \end{bmatrix} \tag{2.9}$$

The Winograd algorithm rewrites this computation as follow:

$$\text{output} = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1 + m2 + m3 \\ m2 - m3 - m4 \end{bmatrix} \tag{2.10}$$

where

$$\begin{aligned} m1 &= (d0 - d2)g0 \qquad & m2 &= (d1 + d2)\frac{g0 + g1 + g2}{2} \\ m4 &= (d1 - d3)g2 \qquad & m3 &= (d2 - d1)\frac{g0 - g1 + g2}{2} \end{aligned} \tag{2.11}$$

The two fractions in $m2$ and $m3$ involve only weight's values, so they can be computed once per filter. Moreover, the normal matrix multiplication is composed of 6 multiplications and 4 addition, while the winograd algorithm reduce the number of multiplication to 4, that is very significant, considering that a single multiplication takes 7 clock-cycles and an addition only 3. In Byron we provide the winograd algorithm for square kernels of size 3 and stride 1, since it is one of the most common combinations in Deep Learning and the generalization is not straightforward.

In the backward operation is important to remember that each weight in the filter contributes to each pixel in the output map. Thus, any change in a weight in the filter will affect all the output pixels. Note that the backward function can still be seen as a convolution between the input and the matrix of errors δ^l for the updates and as a full convolution between δ^l and the flipped kernel for the error δ^{l-1}. In the case the windows of kernels overlap, updates are the sum of all the contributing elements of δ^l.

Pooling

Pooling operations are down-sampling operations, so that the spatial dimensions of the input are reduced. Similarly to what happens in a CL, in pooling layers a 3-D kernel of size $k_x \times k_y \times C$ slides across an image of size $H \times W \times C$, however the operation performed by this kind of layers is fixed and does not change during the course of training. The two main pooling functions are max-pooling and average-pooling: as suggested by the names, the former returns the maximum value of every window of the images super-posed by the kernel, as shown in figure 2.3.

The latter instead, returns the average value of the window and can be seen as a convolution where every weight in the kernel is $\frac{1}{k_x \cdot k_y}$. The results expected from an Average pooling operations are shown in figure 2.4.

Other popular pooling functions include the L^2 norm of a rectangular neighborhood or a weighted average based on the distance from the central pixel.

A typical block of a convolutional network consists of three stages: In the first stage a CL performs several convolutions in parallel, in the second stage each convolution result is

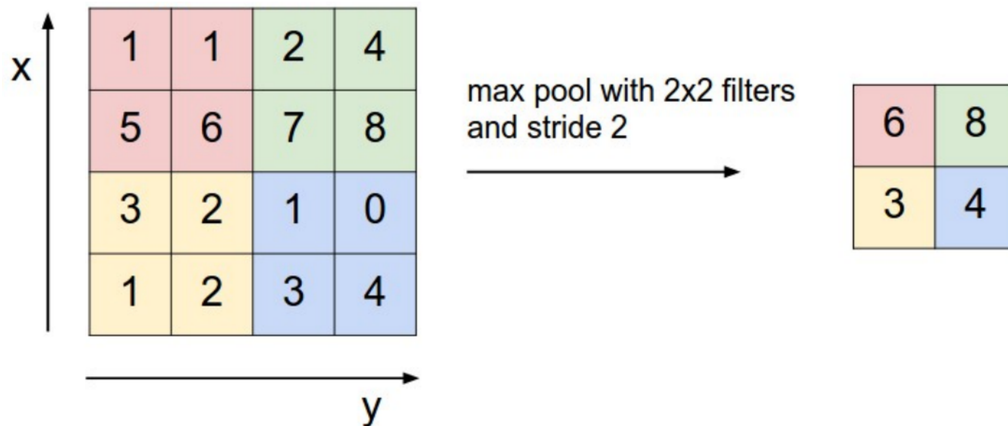


Figure 2.3: *Scheme of maxpool operations with a kernel of size* 2×2 *and stride* 2 *over an image of size* 4×4*. Picture from CS231n*

Figure 2.4: *Average pooling applied to a test image: (left) the original image, (center) average pooling with a* 3×3 *kernel, (right) average pooling with a* 30×30 *kernel. The images have been obtained using NumPyNet*

run through a non-linear activation function (sometimes called *detector*) and in the third stage a pooling function is used to further modify the output. The modification brought by pooling is helpful in different ways: first of all, it is a straightforward computational performance improvement, since less features also means less operations. Moreover, in all cases, pooling helps to make representation approximately invariant to small translation of the input and invariance to local translation can be a useful property if the objective is to decide wether a feature is present rather than where it is located [11]. The reductions of features can also prevent over-fitting problems during training, improving the general performances of the model.

A pooling layer is defined by the same parameters as a CL, minus the number of filters; moreover, also the output dimensions for Pooling layers are the same as for CLs, however, they have no weights to be trained.

Due to the similarities with the CL it is possible to implement a pooling layers through the im2col algorithm, as an example, the NumPyNet implementation shown in the snippet below make use of the function `asStride` to create a `view` of the input array:

Listing 2.1: NumPyNet version of Maxpool function

```
import numpy as np

class Maxpool_layer(object):

  def __init__(self, size=(3, 3), stride=(2, 2)):

    self.size = size
    self.stride = stride
    self.batch, self.w, self.h, self.c = (0, 0, 0, 0)
    self.output, self.delta = (None, None)

  def _asStride(self, input, size, stride):

    batch_stride, s0, s1 = input.strides[:3]
    batch,         w,  h  = input.shape[:3]
    kx, ky      = size
    st1, st2    = stride

    view_shape = (batch, 1 + (w - kx)//st1, 1 + (h - ky)//st2) + input.
    shape[3:] + (kx, ky)

    strides = (batch_stride, st1 * s0, st2 * s1) + input.strides[3:] +
    (s0, s1)

    subs = np.lib.stride_tricks.as_strided(input, view_shape, strides=
    strides)

    return subs

  def forward(self, input):

    kx , ky  = self.size
    st1, st2 = self.stride
    _, w, h, _ = self.input_shape

    view = self._asStride(input)

    self.output = np.nanmax(view, axis=(4,5))

    new_shape = view.shape[:4] + (kx*ky, )

    self.indexes = np.nanargmax(view.reshape(new_shape), axis=4)

    self.indexes = np.unravel_index(self.indexes.ravel(), shape=(kx, ky
    ))

    self.delta = np.zeros(shape=self.out_shape, dtype=float)

    return self
```

A `view` is a special `numpy` object which retains the same information of the original array arranged in a different way, but without occupying more memory. In this case, the re-arrengement is very similar to an im2col, with the only difference that we are not bound to any number of dimensions. The resulting tensor has indeed 6 dimensions. Since no copy is produces in this operation we can obtain a faster execution.

In pooling layer the backward function is similar to what we saw for convolutional layers, this time we don't have to compute the weights updates though, only the error to back-progate along the network. For maxpool layers, only the maximum input pixel for every window is involved in the backward pass. Indeed, if we consider the simple case in which the forward function is:

$$m = max(a, b) \tag{2.12}$$

and, as described in the dedicated chapter, we know that $\frac{\partial C}{\partial m}$ is the error passed back from the next layer: the objective is to compute $\frac{\partial C}{\partial a}$ and $\frac{\partial C}{\partial b}$. If $a > b$ we have:

$$m = a \quad \Rightarrow \quad \frac{\partial C}{\partial m} = \frac{\partial C}{\partial a} \tag{2.13}$$

m does not depends on b so $\frac{\partial C}{\partial b} = 0$.

So the error is passed only to those pixels which value is maximum in the considered window, the other are zeros. In figure 2.5 an example of forward and backward pass for a maxpool kernel of size 30 and stride 20.

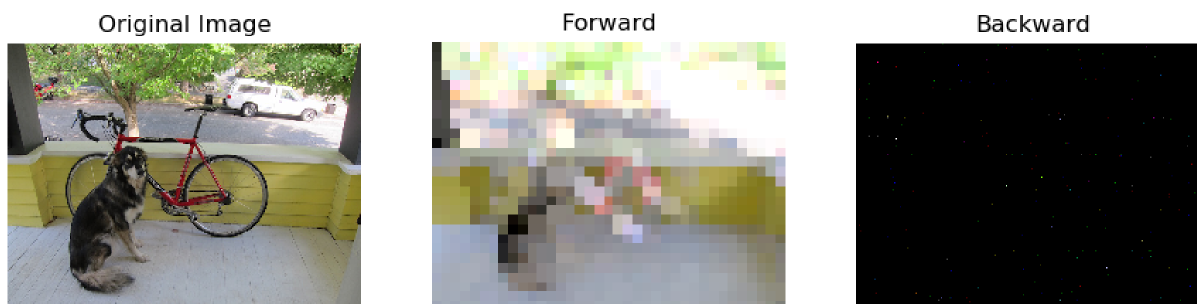


Figure 2.5: *Max pooling applied to a test image: (left) the original image, (center) max pooling with a* 30×30 *kernel and stride* 20*, (right) max pooling errors image. Only few of the pixels are responsible for the error backpropagation. The images have been obtained using NumPyNet*

The backward pass for the average pool layer is the same as for the CL, considering that in this case the "weights" are fixed.

Shortcut Connections

An important advancement in network architecture has been brought by the introduction of Shortcut (or Residual) Connections [12]. It is well know that deep models suffer from *degradation problems* after reaching a maximum depth. Adding more layers, thus increasing the depth of the model, saturates the accuracy which eventually starts to rapidly decrease. The main cause of this dergradation is not overfitting, but numerical instability tied to gradient backpropagation: indeed, as the gradient is back-propagated through the network, repeated multiplications can make those gradients very small or, alternatevely, very big, This problem is well known in Deep Learning and takes the name of *vanishing/exploding gradients* and it makes almost impossible to train very large models, since early layers may not learn anything even after hundreds of epochs. A residual connection is a special shortcut which connects 2 different part of the network with a simple linear combination. Instead of learning a function $F(x)$ we try to learn $H(x) = F(x) + x$, as shown in figure 2.6:

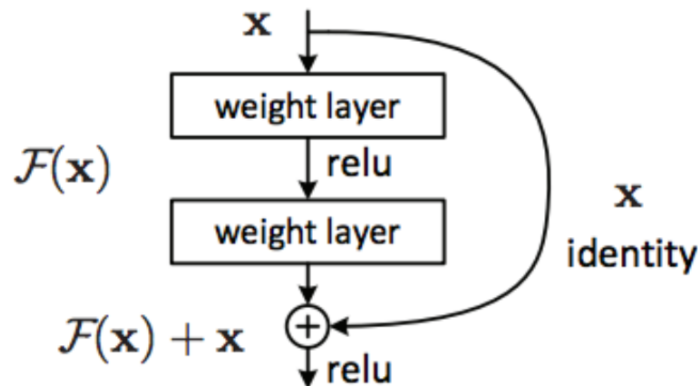


Figure 2.6: *Scheme of the shortcut layer as designed by the authors [12]. The output of the second layer become a linear combination of the input x and its own output.*

During the back propagation the gradient of higher layers can easily pass to the lower layers, without being mediated, which may cause vanishing or exploding gradient.
Both in NumPyNet and Byron, we chose to generalize the formula as:

$$H(x_1, x_2) = \alpha x_1 + \beta x_2 \tag{2.14}$$

Where x_1 is the output of the previous layer and x_2 is the output of the layer selected by `index` paramter. Indeed, even the shortcut connection can be implemented as a stand-alone layer, defined by the following parameters:

- **index** is the index of the second input of this layer x_2 (the first one x_1 is the output of the previous layer).
- **alpha** the first coefficient of the linear combination, multiplied by x_1.

- **beta** the second coefficient of the linear combination, multiplied by x_2.

. The backward function is simply:

$$\frac{\partial C}{\partial x_1} = \frac{\partial C}{\partial H}\frac{\partial H}{\partial x_1} = \delta \cdot \alpha \tag{2.15}$$

for the first layer and:

$$\frac{\partial C}{\partial x_2} = \frac{\partial C}{\partial H}\frac{\partial H}{\partial x_2} = \delta \cdot \beta \tag{2.16}$$

for the second layer. Again, δ is the error backpropagated from the next layer. Residuals connections were first introduced for image classification problems, but they rapidly become part of numerous models for every kind of application tied to Image Analysis.

Pixel Shuffle

Using pooling and convolutional layers with non unitarian strides is a simple way to downsample the input dimension. For some applications though, we may be interested in upsampling the input, for example:

- in image to image processing (input and output are images of the same size) it is common to perform a compression to an internal encoding (e.g Deblurring, U-Net Segmentation).
- project feature maps to a higher dimensional space, i.d. to obtain a image of higher resolution (e.g Super-Resolution)

for those purposes the *transposed convolution* (also called *deconvolution*) was introduced. The transposed convolution can be treated as a normal convolution with a sub-unitarian stride, by upsampling the input with empty rows and columns and then apply a single strided convolution, as shown in figure 2.7.

Although it works, the transposed convolution is not efficient in terms of computational and memory cost, therefore not suited for modern convolutional neural networks. An alternative is the recently introduced *sub-pixel convolution* [25] (also called Pixel Shuffle). The main advantages over the deconvolution operation is the absence of weights to train: indeed the operation performed by the Pixel Shuffle (PS) Layer is deterministic and it is very efficient if compared to the deconvolution, since it is only a re-arrengement of the pixels.

Given a scale factor r, the PS organizes an input $H \times W \times C \cdot r^2$ into an output tensor $r \cdot H \times r \cdot W \times C$, which generally is the dimension of the high resolution space. So, strictly speaking, the PS does not perform any upsample, since the number of pixels stays the same. In figure 2.8 is shown an example with $C = 1$:

As suggested by the authors, the best practice to improve performances is to upscale from low resolution to high resolution only at the very end of the model. In this way the

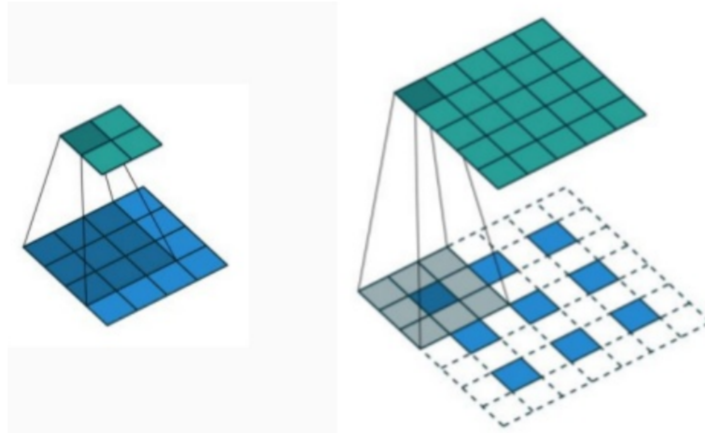


Figure 2.7: *example of deconvolution: (left) a normal convolution with size 3 and stride 1, (right) after applying a "zeros upsampling" the convolution of size 3 and stride 1 become a deconvolution*

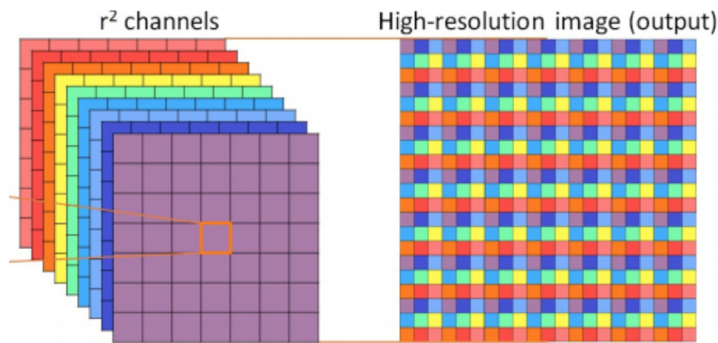


Figure 2.8: *Example of pixel shuffling proposed by the authors [25]. In this example, r^2 features maps are re-arranged into a single-channeled high resolution output.*

CL can efficienlty produce an high number of low resolution feature maps that the PS can organize into the final output.

In both NumPyNet and Byron, the pixel shuffle layer is defined only by the `scale` parameter, which lead the entire transformation. In the first case, it is possible to implement forward and backward using the functions `split`, `reshape`, `concatenate` and `transpose` of the `numpy` library [21]. This implementation has been tested against `tensorflow`'s `depth_to_space` and `space_to_depth`. Despite being available in most deep learning library, a low level `C++` implementation for the PS algorithm is hard to find. In Byron we propose a dynamic algorithm able to work for both `channel last` and `channel first` input. The algorithm is essentially a re-indexing of the input array in six nested for-loops. The first solution taken into account during the development was the contraction of the loops into a single one using divisions to obtain the correct indexes: however the amount of required divisions weights on the computational performances,

given that divisions are the most expensive operation in terms of CPU clock-cycles.

The backward function of this layer does not involve any gradient computation: instead, it is the inverse of the re-arrengement performed in the forward function.

Batch Normalization

When training a neural network, the standard approach is to separate the dataset in groups, called *batches* or *mini-batches*. In this way the network can be trained with multiple input at a time and the updates for the weights are usually computed by averaging in the batch. The number of examples in each batch is called *batch size*: this can varies from 1 to the size of the dataset. Using batch sizes different from one is beneficial in several ways. First, the gradient of the loss over a mini-batch is a better estimate of the gradient over the train set, whose quality improves as the batch size increases, but using the entire train set can be very costly in terms of memory usage and often impossible to achieve. Second, it can be much more efficient in modern architecture due to the parallelism instead of performing M sequential computations for single examples. [15]

Batch normalization is the operation that normalizes the features of the input along the batch axis, which allows to overcome a phenomenon in Deep Network training called *internal covariate shift*: whenever the parameters of the model change, the input distributions of every layer change accordingly. This behaviour produces a slow down in the training convergence because each layer has to adapt itself to a new distribution of data for each epoch. Moreover, the parameters must be carefully initialized. By making the normalization a part of the model architecture, the layer acts also as a regularizer, which in turn allows better generalization perfomances.

Let's M be the number of examples in the group and ϵ a small variable added for numerical stability, the batch normalization function is defined as:

$$\mu = \frac{1}{M} \sum_{i=1}^{M} x_i \tag{2.17}$$

$$\sigma^2 = \frac{1}{M} \sum_{i=1}^{M} (x_i - \mu)^2 \tag{2.18}$$

$$\hat{x}_i = \frac{(x_i - \mu)^2}{\sqrt{\sigma^2 + \epsilon}} \tag{2.19}$$

$$y_i = \gamma \bar{x}_i + \beta \tag{2.20}$$

where γ and β are the trainable weights of this layer. In the case of a tensor of images of size $M \times H \times W \times C$ all the quantities are multidimensional tensors as well and all the operations are performed element-wise.

The backward function can be computed following the chain rule for derivatives. As usual, define $\delta^l = \frac{\partial C}{\partial y}$ as the error coming from the next layer, the goal is to compute the updates for γ and β and the error for the previous layer. The updates are straightforward:

$$\frac{\partial C}{\partial \gamma} = \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial \gamma} = \sum_{i=1}^{M} \delta_i^l \cdot \hat{x}_i \tag{2.21}$$

$$\frac{\partial C}{\partial \beta} = \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial \beta} = \sum_{i=1}^{M} \delta_i^l \tag{2.22}$$

while the error requires more steps:

$$\frac{\partial C}{\partial x} := \delta^{l-1} = \delta^l \cdot \frac{\partial y}{\partial x} \tag{2.23}$$

where:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial \hat{x}}(\frac{\partial \hat{x}}{\partial \mu}\frac{\partial \mu}{\partial x} + \frac{\partial \hat{x}}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial x} + \frac{\partial \hat{x}}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial \mu}\frac{\partial \mu}{\partial x}) \tag{2.24}$$

By considering all the derivatives, we find:

$$\frac{\partial C}{\partial x_i} := \delta_i^{l-1} = \frac{M\delta_i^l \cdot \gamma_i - \sum_{j=1}^{M} \delta_j^l \cdot \gamma_i - \hat{x}_i \cdot \sum_{j=1}^{M} \delta_j^l \cdot \hat{x}_j}{M\sqrt{\sigma^2 + \epsilon}} \tag{2.25}$$

Knowing the correct operations, an example of implementation is shown in the snippet 2.2:

Listing 2.2: NumPyNet version of batchnorm function

```python

def forward(self, inpt):

  self._check_dims(shape=self.input_shape, arr=inpt, func='Forward')

  self.x    = inpt.copy()
  self.mean = self.x.mean(axis=0)
  self.var  = 1. / np.sqrt((self.x.var(axis=0)) + self.epsil)

  self.x_norm = (self.x - self.mean) * self.var
  self.output = self.x_norm.copy()

  self.output = self.output * self.scales + self.bias

  self.delta = np.zeros(shape=self.out_shape, dtype=float)

  return self

def backward(self, delta=None):

  invN = 1. / np.prod(self.mean.shape)

  # Those are the explicit computation of every derivative involved in
   BackPropagation
```

```
24
25  self.bias_update = self.delta.sum(axis=0)
26  self.scales_update = (self.delta * self.x_norm).sum(axis=0)
27
28  self.delta *= self.scales
29
30  self.mean_delta = (self.delta * (-self.var)).mean(axis=0)
31
32  self.var_delta = ((self.delta * (self.x - self.mean)).sum(axis=0) *
33                    (-.5 * self.var * self.var * self.var))
34
35  self.delta = (self.delta * self.var +
36                self.var_delta * 2 * (self.x - self.mean) * invN +
37                self.mean_delta * invN)
38
39  if delta is not None:
40    delta[:] += self.delta
41
42  return self
```

As we can see, in `numpy` it's possible to easily implement all element-wise operations with standard algebra.

In `Byron` we decided to implement this operation both as a standalone function and merged into convolutional and fully connected layers, with the latter being the most used in modern models since it achieves the best computational performances.

Activations

An important role in neural network is played by the choice of activation function. They are linear or non-linear functions which process the output of a neuron and bound it to a certain range. The introductions of non-linearities allows a Neural Network to model a wider range of functions and learn more complex relations in data patterns.

Many activation functions were proposed during the years and each one has its characteristics, but not an appropriate field of application. How to chose the best activation function in a given situation is still an open question: each one has its pros and cons in some situations, so each Neural Network library implements a wide range of them and it leaves the user perform their own tests. In table 2.1 is reported a full record of the activation functions and their derivatives implemented in `Byron` and `NumPyNet`. An important feature of any activation function, in fact, is that it should be differentiable since the main procedure of model optimization implies the back-propagation of the error gradients. As can be seen in tab. 2.1 it is easier to compute the activation function derivative as a function of it: this is an important type of optimization in computation term, since it reduces the number of operations and it allows to apply the backward gradient directly on the output, without storing the un-activated one.

In figure 2.9 is shown the effect of some functions along with they're respective backwards.

Name	Equation	Derivative
Linear	$f(x) = x$	$f'(x) = 1$
Logistic	$f(x) = \frac{1}{1+\exp(-x)}$	$f'(x) = (1 - f(x)) * f(x)$
Loggy	$f(x) = \frac{2}{1+\exp(-x)} - 1$	$f'(x) = 2 * (1 - \frac{f(x)+1}{2}) * \frac{f(x)+1}{2}$
Relu	$f(x) = \max(0, x)$	$f'(x) = \begin{cases} 1 & \text{if} \quad f(x) > 0 \\ 0 & \text{if} \quad f(x) \leq 0 \end{cases}$
Elu	$f(x) = \max(\exp(x) - 1, x)$	$f'(x) = \begin{cases} 1 & \text{if} \quad f(x) \geq 0 \\ f(x) + 1 & \text{if} \quad f(x) < 0 \end{cases}$
Relie	$f(x) = \max(x * 1e - 2, x)$	$f'(x) = \begin{cases} 1 & \text{if} \quad f(x) > 0 \\ 1e - 2 & \text{if} \quad f(x) \leq 0 \end{cases}$
Ramp	$f(x) = \begin{cases} x^2 + 0.1 * x^2 & \text{if} \quad x > 0 \\ 0 & \text{if} \quad x \leq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + 1 & \text{if} \quad f(x) > 0 \\ f(x) & \text{if} \quad f(x) \leq 0 \end{cases}$
Tanh	$f(x) = \tanh(x)$	$f'(x) = 1 - f(x)^2$
Plse	$f(x) = \begin{cases} (x + 4) * 1e - 2 & \text{if} \quad x < -4 \\ (x - 4) * 1e - 2 + 1 & \text{if} \quad x > 4 \\ x * 0.125 + 5 & \text{if} \quad -4 \leq x \leq 4 \end{cases}$	$f'(x) = \begin{cases} 1e - 2 & \text{if} \quad f(x) < 0 \text{ or } f(x) > 1 \\ 0.125 & \text{if} \quad 0 \leq f(x) \leq 1 \end{cases}$
Leaky	$f(x) = \begin{cases} x * C & \text{if} \quad x \leq 0 \\ x & \text{if} \quad x > 0 \end{cases}$	$f'(x) = \begin{cases} 1 & \text{if} \quad f(x) > 0 \\ C & \text{if} \quad f(x) \leq 0 \end{cases}$
HardTan	$f(x) = \begin{cases} -1 & \text{if} \quad x < -1 \\ +1 & \text{if} \quad x > 1 \\ x & \text{if} \quad -1 \leq x \leq 1 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if} \quad f(x) < -1 \text{ or } f(x) > 1 \\ 1 & \text{if} \quad -1 \leq f(x) \leq 1 \end{cases}$
LhTan	$f(x) = \begin{cases} x * 1e - 3 & \text{if} \quad x < 0 \\ (x - 1) * 1e - 3 + 1 & \text{if} \quad x > 1 \\ x & \text{if} \quad 0 \leq x \leq 1 \end{cases}$	$f'(x) = \begin{cases} 1e - 3 & \text{if} \quad f(x) < 0 \text{ or } f(x) > 1 \\ 1 & \text{if} \quad 0 \leq f(x) \leq 1 \end{cases}$
Selu	$f(x) = \begin{cases} 1.0507 * 1.6732 * (e^x - 1) & \text{if} \quad x < 0 \\ x * 1.0507 & \text{if} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) * 1e - 3 & \text{if} \quad f(x) 0 \\ (f(x) - 1) * 1e - 3 + 1 & \text{if} \quad f(x) > 1 \end{cases}$
SoftPlus	$f(x) = log(1 + e^x)$	$f'(x) = \frac{\exp(f(x)}{)} 1 + e^{f(x)}$
SoftSign	$f(x) = \frac{x}{\lvert x \rvert + 1}$	$f'(x) = \frac{1}{(\lvert f(x) \rvert + 1)^2}$
Elliot	$f(x) = \frac{\frac{1}{2} * S * x}{1 + \lvert x + S \rvert} + \frac{1}{2}$	$f'(x) = \frac{\frac{1}{2} * S}{(1 + \lvert f(x) + S \rvert)^2}$
SymmElliot	$f(x) = \frac{S * x}{1 + \lvert x * S \rvert}$	$f'(x) = \frac{S}{(1 + \lvert f(x) * S \rvert)^2}$

Table 2.1: *List of common activation functions with their corresponding mathematical equation and derivative. The derivative is expressed as function of $f(x)$ to optimize their numerical evaluation.*

Figure 2.9: *Activation functions applied on test image. From top to bottom: Elu, Relu and Logistic.*

The most used activation functions is undoubtedly the `ReLU` activation (Rectified Linear Unit.) [10]. Its main advantages are:

- Sparsity: a sparse representation of data is exponentially more efficient in comparison of a dense one.
- Vanish Gradient Reduction, since the derivatives does not vanishes during Deeper backpropagation.
- Easy and fast computation of both forward and backward.

Cost Function

A Deep Learning model is trained by minimizing a Cost Function, or, in other words, during training we want to adjust the parameters of the network in order to modify its output, which in turns, will be closer to the desired result. Therefore, it is important to

define what we consider the error function of the model. There are many kinds of loss functions, and none of them is able to work with every kind of data; moreover, there are no particular reasons to prefer a loss function over another one, given that they are both adapt to handle the output of the problem under analysis. One important property is the differentiability of those error functions, since they will be the starting point of chain rule for derivatives used in *Error Backpropagation*. Broadly speaking, loss functions can be separated into two categories: classification losses and regression losses. In the first case we want to predict a finite number of categorical values (classes), while in the second case the prediction is performed over a series of continuos values. The most common cost functions for Super Resolution are *Mean Squared Error* (MSE) and *Mean Absolute Error* (MAE), also called L2 loss and L1 loss. The former is defined as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2 \tag{2.26}$$

where $\boldsymbol{y}$ is the output vector of the model, $\boldsymbol{t}$ is the desired results and N is the dimension of the output. It is one of the most used loss function for different task, not only because is really simple, but also because it reaches good performances. Notice that minimizing the MSE naturally maximize the PSNR score define in previous chapter.

In MAE we replace the squared error with the absolute difference:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - t_i| \tag{2.27}$$

Both models used in this work have been trained using the L1 loss function.

2.3 Timing

I tested `Byron` performances on single layers through its python wrap `Pyron` against the popular deep learning framework `Tensorflow` on CPU. I measured the forward and backward times on 30 repetitions, scanning over the parameters of the different layers as fuctions of the number of threads, for the most important layers in image processing applications: Convolutional, Max-Pool and Pixel-shuffle layer. The number of threads indicates the level of parallelization involved during computations. The inputs are randomly generated tensors of dimensions $16 \times 512 \times 512 \times 3$ which should resemble a batch of images. The content of said images does not really matter, since we are only interested in measuring the processing times. For the first case, the results for *forward* and *backward* functions are summarized in figure 2.10:

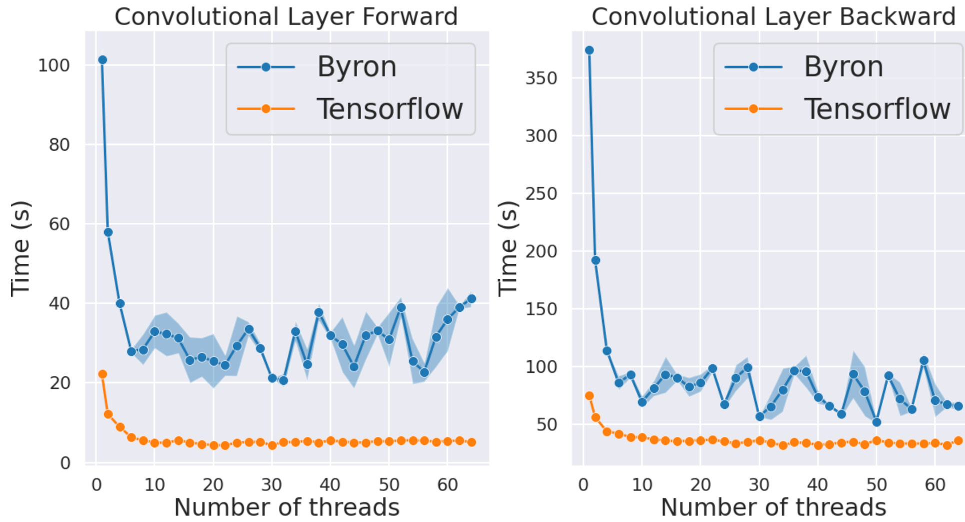


Figure 2.10: *Comparisons between time to perform forward (left) and backward (right) for Convolutional Layer in Byron and Tensorflow on a 4D tensor of size* $16 \times 512 \times 512 \times 3$ *for size 3, stride 1 and 100 filters.*

As expected, the performances improve with higher numbers of threads, however this behaviour last for less than 10, after which the times remain constant for both implementations. In particular, for `Byron`, the algorithm's time seems to be really unstable with higher number of opened threads. Nonetheless, the CPU times of `Tensorflow` are consistently better that `Byron`. Those considerations highlight that there may be problems in threads management during the operations in `Byron`, which do not influence the outputs, but greatly deteriorate timing performances. If we move to a bigger stride value, like 2, the results change as shown in 2.11. In particular, the trends become slighlty more stable a not far from what we are expecting. The conditions $size = 3$ and $stride = 1$ are also the condition for the *Winograd Algorithm*: this can reduce the range of code to check to find the sources of times instability. Another considerations come from the fact that passing from $stride = 1$ to $stride = 2$ in both axis, reduce the number of operations of a factor 4, which means, at least, that the single thread performances should reduce the time cost of 4 time. In `Byron` forward though, the time reduces to roughly 40%, while in `Tensorflow` the speed up seems consistent with that consideration.

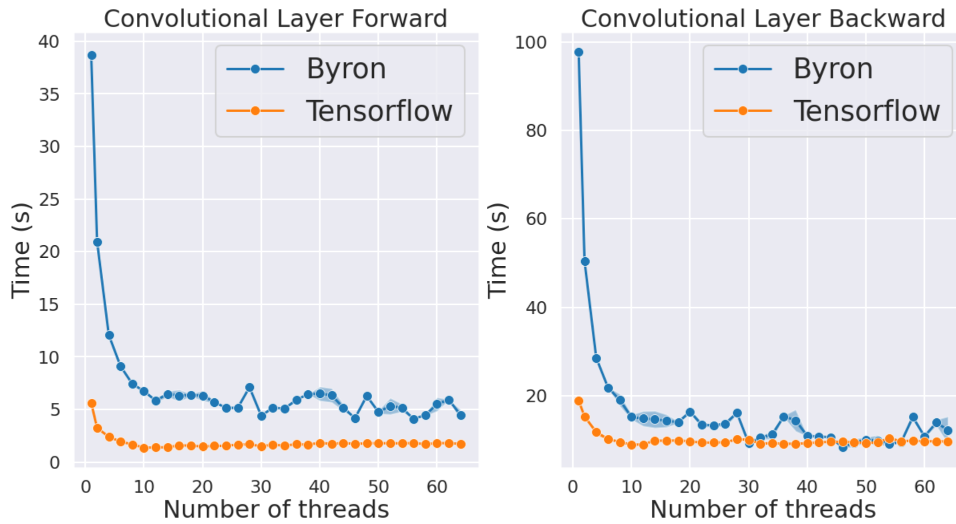


Figure 2.11: *Comparisons between time to perform forward (left) and backward (right) for Convolutional Layer in Byron and Tensorflow on a 4D tensor of size* $16 \times 512 \times 512 \times 3$ *for size 3, stride 2 and 100 filters.*

The second case taken into account is MaxPool Layer and the results are summarized in figure 2.12:

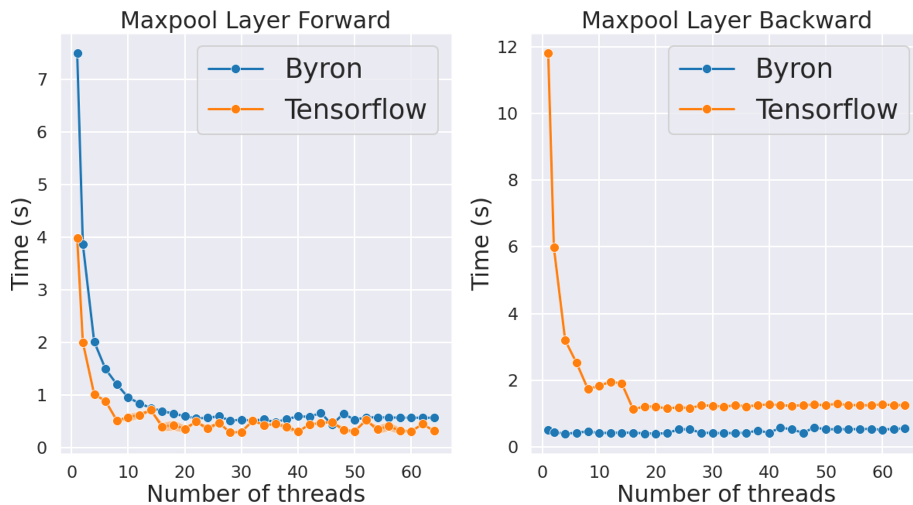


Figure 2.12: *Comparisons between times to perform forward (left) and backward (right) for Maxpool Layer in Byron and Tensorflow on a 4D tensor of size* $16 \times 512 \times 3$ *for size 3 and stride 1.*

In this case, the results seems to show comparable times for `Byron` and `Tensorflow` for the forward function and a faster implementation for the backward function. Again, the time gained by adding more threads does not improve much after a certain number: this can also be due to the dimensions of the input not being large enough to justify

opening such an high numbers of parallel operations. All-in all, this implementation seems to be performing as expected. Moreover, using an higher kernel size seems to favour `Byron` over `Tensorflow`, as shown in figure 2.13

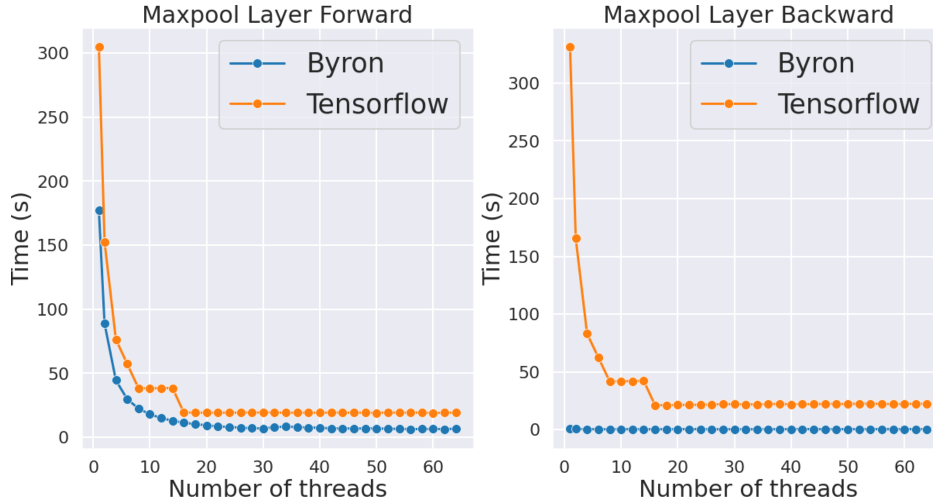


Figure 2.13: *Comparisons between times to perform forward (left) and backward (right) for Maxpool Layer in Byron and Tensorflow on a 4D tensor of size* $16 \times 512 \times 3$ *for size 30 and stride 1.*

This may indicates that the higher the numbers of operations, the higher is the advantage gained with `Byron`. In our implementation of Maxpool Layer, during the forward function we store the indexes needed for the backward operation: in this way we lose a bit of time during the forward pass, however the backward become just a single `for loop` on the array of indexes. Most likely, `Tensorflow` opted for a different implementation, given the results.

The third layer taken into consideration is the Pixel Shuffle, in particular I considered a scale factor of 6 with an input tensor of dimension $16 \times 512 \times 512 \times 108$. The singular number of input channels has been chosen to be compatible with the re-scaling performed by the pixel-shuffle operation. The `Tensorflow` implementation of Pixel shuffle seems to behave very differently from previous cases: indeed it does not scale at all with the number of threads, and times remain very similar from 1 to 64 parallel sections. Again, the single thread performances favour `Tensorflow` over `Byron`: although in the forward function `Byron` speed ups quickly with the higher number of threads. I would like to point out that even if the two operations of forward and backward are very similar, being only re-arrengement of the same number of pixels, the latter seems to be faster in both implementations. The other scale factor and input shapes tested does not show any different behaviour from the ones described in this case.

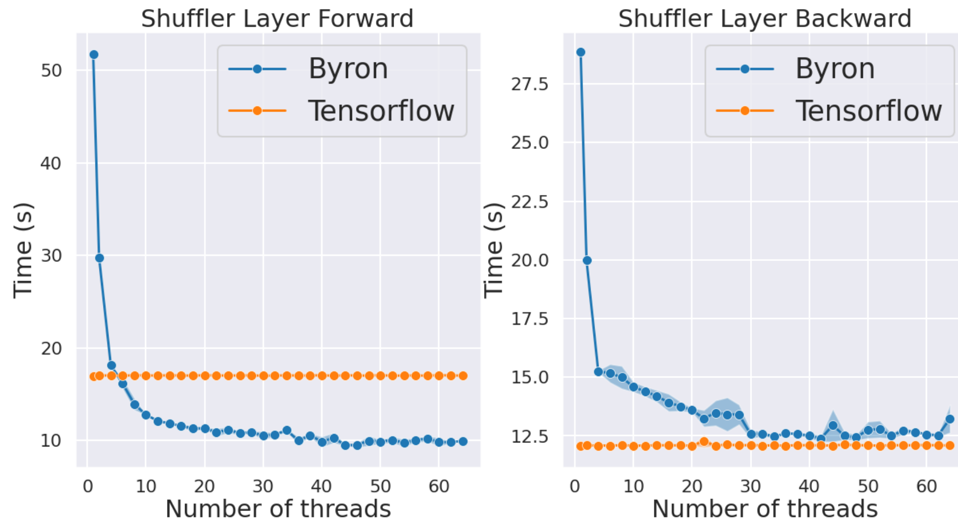


Figure 2.14: *Comparisons between times to perform forward (left) and backward (right) for Shuffler Layer in Byron and Tensorflow on a 4D tensor of size* $16 \times 512 \times 108$ *for scale 6*

Chapter 3

Datasets and Methodology

3.1 Models

As already described in previous chapters, the high level of modularity provided by `Byron` and `NumPyNet` allows to use different kind of models for many different purposes.
The two models chosen for super-resolution are `EDSR`, used to performs a x2 upsample, and `WDSR`, used to perform a x4 upsample. They are the winners of the *NTIRE* challenge (New Trends in Image Restoration and Enhancement) respectively for the year 2017 and 2018 and the model structure and weights are publicly available at their official github repositories; EDSR and WDSR.

EDSR

`EDSR` (Enhanced Deep Super-Resolution) [18] is the first model considered and ported in `Byron`. It is a Deep Neural Network specialized in Single Image Super Resolution (SISR) which won the NTIRE challege in 2017. Its structure is based on previous famous models in image analysis application, namely `SRResNet` and, consequently, `ResNet`. For this reason, the base unit of `EDSR` is the *residual block*, which makes extensive use of *residual connections*, described in the previous chapter. The major improvements from older models come from removal of batch-normalization between convolutional layers inside *residual blocks*: indeed the authors proved experimentally that normalization of the features reduces performances substantially, by getting rid of range flexibility [18]. Moreover, every Batch-Norm layer contains the same amount of weights as the convolutional layer preceeding it, therefore removing them saves approximately 40% of memory usage during training and this allows the construction of larger models. On the other hand, they also showed that deepening the model above a certain level would make the training procedure numerically unstable: the authors solved this issue by adopting a residual scaling with factor 0.1 in shortcut connections.

The structure of the model is summarize in figure 3.1. The model used in this work is called *EDSR_x2* by the authors and is composed by 32 *residual block*, each in turn is

composed by:

- A convolutional layer with 256 filters
- An activation layer with a ReLU function
- A convolutional Layer with 256 filters
- A linear combination pixel-by-pixel with the input of the residual block with weight respectively 0.1 and 1 (shortcut connection's α and β)

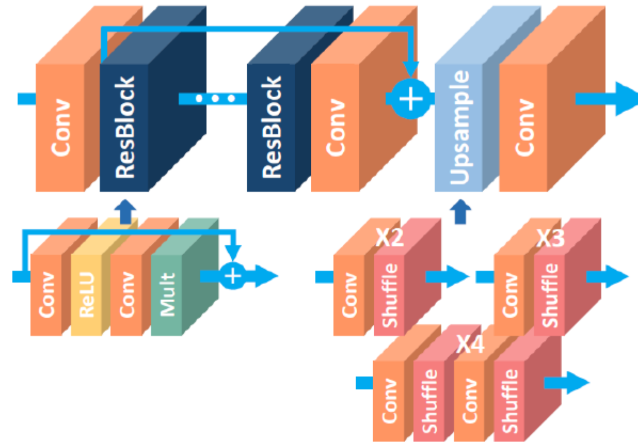


Figure 3.1: *architecture of the single scale SR Network (EDSR)*

The last part of the network is composed by a shortcut connection which performs a linear combination of the initial input with the output of the 32 residual block and by the up-sampling block:

- A convolutional layer with 1024 filters
- A pixel-shuffle with a scale factor $r = 2$
- A convolutional layer with 3 filter (the final output.)

In total, `EDSR` has more than 40 millions of parameters, summarized in table 3.1

Layer	Channels in/out	Filters Dim.	Parameters
Conv. Input	3 / 256	3x3	6912
Res. Block	256 / 256	3x3	1179648
Conv. Pre Short.	256 / 256	3x3	589824
Conv. Pre-Shuffle	256 / 1024	3x3	2359296
Conv. Output	256 / 3	3x3	6912

Table 3.1: *Table of parameters for the different sections of EDSR.*

The "heaviest" part of the model is clearly the Residual Blocks section, which alone contains more that 37 millions of parameters.

For training they used 48×48 RGB patches from LR images (from DIV2K) with the corresponding HR patches, augmenting the training data with random horizontal flips and 90° rotations. The optimizer is the ADAM [17] and the loss function is the L1: even if the L2 naturally maximize the PSNR (which is the only metrics evaluated in the challenge), they found that L1 loss provided a better convergence.

WDSR

The second model considered in this study and implemented in `Byron` is called Wide Deep Super Resolution (WDSR) [29] and it is the winner of the NTIRE challenge 2018.

The model used in ths work is called `WDSR_x4_A` and it is composed by 32 residual blocks, similarly to `EDSR`, the difference is that they are much lighter in this case, indeed a residual block is composed by:

- Convolutional layer with 192 filters
- Activation layer with ReLU function
- Convolutional Layer with 32 filters
- Shortcut connection with $\alpha = \beta = 1$

In their work, the authors designed the network to study the importance of *wide features* before ReLU activations. Indeed they slim the features of residual identity mapping pathway while expanding the features before activation. From this simple idea they claim `WDSR-A` is extremely effective for improving accuracy of single image super-resolution when the scale factor is between 2 and 4, but the performance drops quickly after this threshold.

Another important step is the introduction of weight normalization over batch normalization (BN) which does not introduce the troubles of BN while speeding up the convergence of Deep Neural Networks, by allowing the usage of higher learning rates [29].

Further improvements can be found in the structure of the network as shown in figure 3.2: they simplify the network architecture by removing redundant Convolution pre-shuffling. Thus, they modify the structure by introducing a single convolutional layer with size 5×5 that works directly on the input, extracting the low frequency features of the sample. Those modifications result in less parameters, without affecting the accuracy.

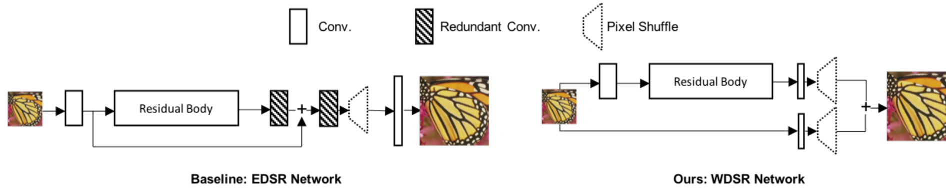


Figure 3.2: *comparisons between* **EDSR** *architecture on the left and* **WDSR** *architecture on the right*

A summary of the model's sections along with the numbers of weights they contains can be found in table 3.1:

Layer	Channels in/out	Filters Dim.	Parameters
Conv. Input	3 / 32	3x3	864
Res. Block Conv. (1)	32 / 192	3x3	55296
Res. Block Conv. (2)	192 / 32	3x3	55296
Conv. Pre Shuffle (1)	32 / 48	3x3	13824
Conv. Pre-Shuffle (2)	3 / 48	5x5	38400

Table 3.2: *Table of parameters for the different sections of WDSR.*

In total, `WDSR_A` is composed by more that 3 millions parameters, still 10 time less than `EDSR` while reaching similar results if compared on the same categories. The lesser amount of weights not only decreases memory usage during both test and training, but also greatly increse computational performance: indeed a single forward of `WDSR` is more than 10 times faster than `EDSR`. The training dataset is composed by 96×96 patches from every images of the training section of the DIV2K dataset. Again, the data are augmented with random horizontal flips and rotations and the optmizer is ADAM [17] minimizing the L1 loss functions.

3.2 Train Dataset: DIV2K

The training set is a general purpose dataset called DIV2K [3] and it has been employed to train and validate `EDSR` and `WDSR` for the *NTIRE* competition (*New Trends in Image Restoration and Enhancement*).

The dataset is composed by 1000 2K RGB images with a large diversity of contents, divided into:

- **Training set**: 800 HR images and 800 LR images obtained from the HR ones using different downscaling factor (2x, 3x, 4x) and different degrading factors.
- **Validation set**: 100 HR images and 100 LR images used as a test set to evaluate the models by the competitors.
- **Test set**: 100 LR images for which an HR version is made available only at the end of the competition. This is used by the competitors to test the models and for their final evaluation.

A qualitative proof of the results obtainable from the two models are shown in pictures 3.3, 3.4 and 3.5.

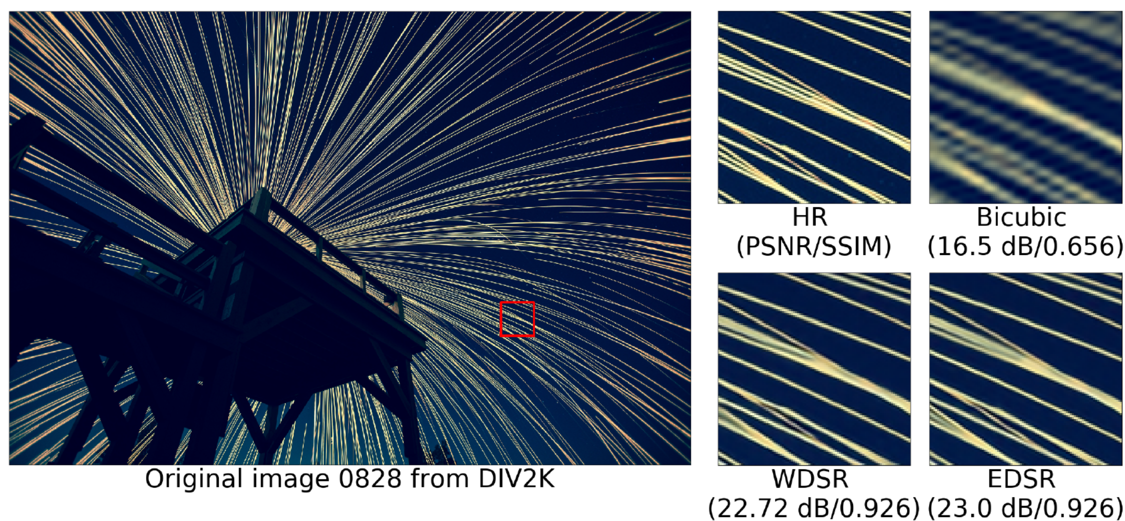


Figure 3.3: *Super Resolution visual example extracted from the DIV2K validation set. The quality score in terms of PSNR and SSIM are compared between a standard bi-cubic up-sampling and the EDSR and WDSR models.*

Figure 3.4: *Super Resolution visual example extracted from the DIV2K validation set. The quality score in terms of PSNR and SSIM are compared between a standard bi-cubic up-sampling and the EDSR and WDSR models.*

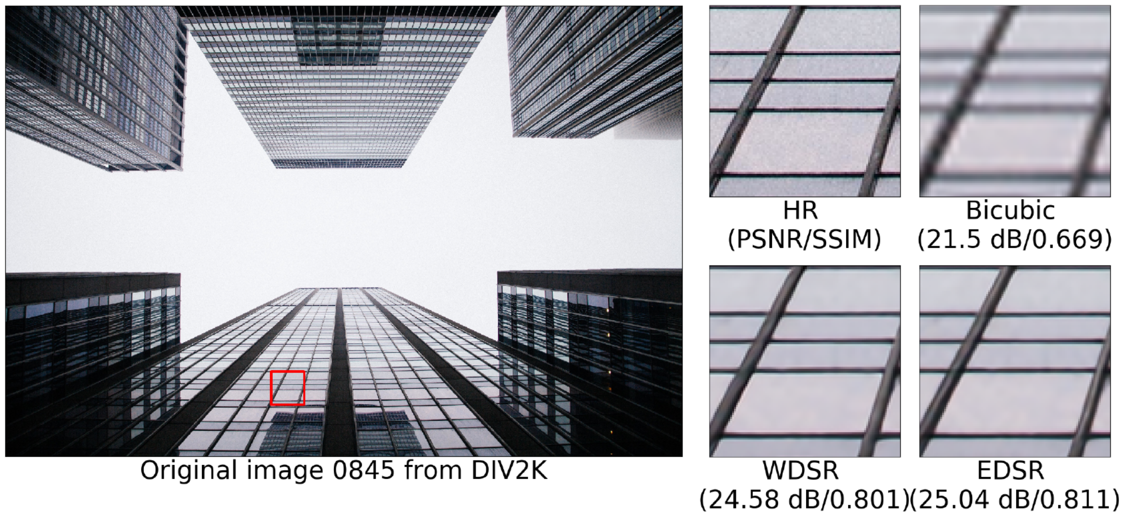


Figure 3.5: *Super Resolution visual example extracted from the DIV2K validation set. The quality score in terms of PSNR and SSIM are compared between a standard bi-cubic up-sampling and the EDSR and WDSR models.*

As can be seen in those pictures, the models have learned how to interpolate the complex line shapes and different kind of textures better than the bicubic algorithm. Given the great heterogeneity of contents inside the dataset, after the training phase the models are able to reconstruct a huge amount of distict shapes and textures. For this reasons we decided to test the performances of `WDSR` and `EDSR` on a set of NMR data.

3.3 NMR Dataset

To test the model on NMR images, we used a series of 5 patients weighted T_1 and T_2 sampled with a spatial frequency of $1mm \times 1mm \times 1mm$ for each direction (x, y, z), with a resolution of 256×256 for a total of 176 slices. For reference, in figure 3.6 are shown three different slices for the same patient at HR:

Figure 3.6: *HR* 256×256 *original image at three different stages of depth: (left) slice 30 where still a lot of information about the brain is hidden, (center) slice 100 which is a central slice where most of the information is stored, (right) slice 150 which starts the less informative area of the brain.*

The HR images (which will also be called *originals*) are convoluted with a gaussian kernel of size 3, stride 1 and standard deviation 1 with the function `cv2.GaussianBlur` of the library `OpenCV`. Then, they have been downsampled with the bicubic algorithm by two different scale factors, namely $\times 2$ and $\times 4$, with the function `resize`, also available from OpenCV, obtaining two distinct sets of LR images for every subject and for every weight, respectively of sizes $128 \times 128 \times 176$ and $64 \times 64 \times 176$. The gaussian blurring has been done to better resemble a LR data-acquisition, as if the images were obtained at low resolution directly and not coming from a downsampling. In figure 3.7 is shown an example of the images obtained by this procedure for a downscale factor of 2:

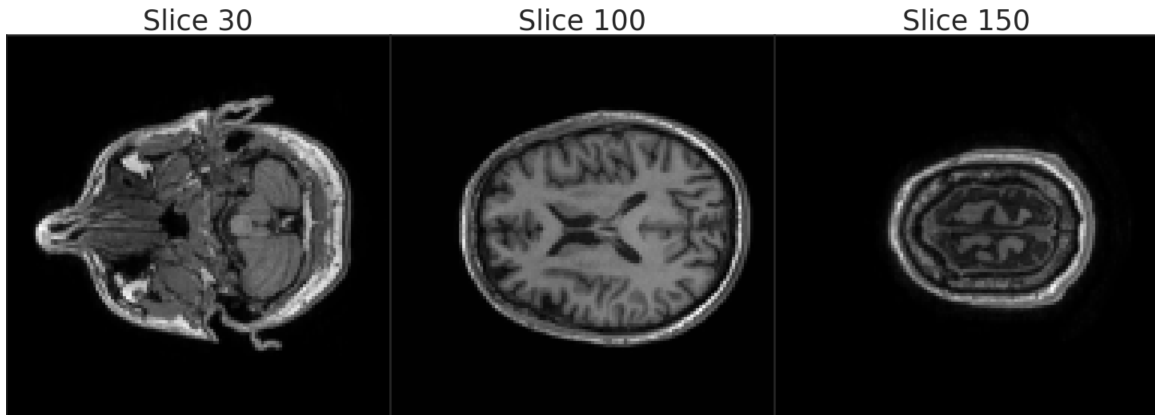


Figure 3.7: 128×128 *LR version of the same slices shown for the HR case.*

In figure 3.8 is shown an example of the images obtained by this procedure for a downscale factor of 4:

Figure 3.8: 64×64 *LR version of the same slices shown for the HR case.*

The LR images are used as input for `EDSR`, `WDSR` and the bicubic algorithm which re-upsample them respectively by factors $\times 2$, $\times 4$ and both, trying to recostruct an image as close as possible to the original one. As a further analysis, I decided to investigate how different input conditions influence the results for the three methods. In particular how an angle of rotation for the LR images can impact the final re-upsample: indeed this changes the orientation of lines, shapes, and textures, which can affect the reconstruction. Nonetheless it can give an important insight on the level of invariance of the two models and on the *explainability* of the results.

I divided the full angle into 20 sections, separated by a step of 18°. In figure 3.9 is shown an example of the kind of inputs fed to the models:

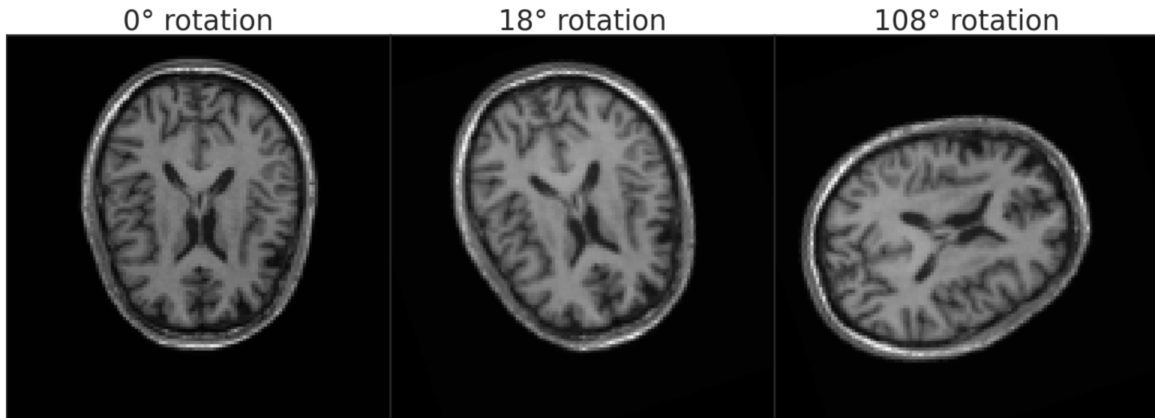


Figure 3.9: *Three of the 20 rotation angles used as input for Super Resolution models and Bicubic. (left) reference angle of 0 degree, (centre) angle step of 18 degree, (right) large rotation of 108° respect to the reference.*

The recostruction are compared with the original images using PSNR and SSIM values for every patient, weight, channel, scale factor and angle of rotation. As stated before, the NMR slices are 1-channeled gray-scale image while the SR models work on RGB image: this is solved by adding an artificial depth concatenating the same slice 3 times. The dataset of original HR images is publicly available from NAMIC [1].

Chapter 4

Results

In the following chapter I am going to report quantitative and qualitative results for the different analysis carried out during the work. I describe the quantitative results obtained by the different methods evaluated by means of PSNR, SSIM score and by a qualitative visual analisys. At the end of the chapter I will summarize the results and provides possible continuations for future analysis.

4.1 Upsample Comparisons

The `EDSR` model is used to upsample the images by a $\times 2$ factor so that the single slice is super-resolved from a 128×128 to a 256×256 spatial resolution. The `WDSR` model instead, is used to upsample the images by a $\times 4$ factor so that the single slice is super-resolved from a 64×64 to a 256×256 spatial resolution.

I decided to separate the analysis for the three output channels of the super-resolution since it can highlight particular behaviours. Moreover, the two kinds of MRI, T1-weighted and T2-weighted, will also be evaluated separately, because it can give a useful insight on what the models are able to “see”. In figure 4.1 are shown the average trends for PSNR and SSIM score for the three channels (Red, Green, Blue lines) and for the bicubic algorithm (Yellow lines), of cases weighted T1. They are averaged for all patients and angle of rotations. It is clear that there is a difference between the three outputs for the super resolution: namely there is a vertical shift in the trends passing from Red to Green and from Green to Blue, the latter being the best performer. Moreover, the “best” channel for `EDSR` outperforms consinstently the bicubic algorithm, and the second best outperforms the bicubic algorithm in the most informative section of the images.

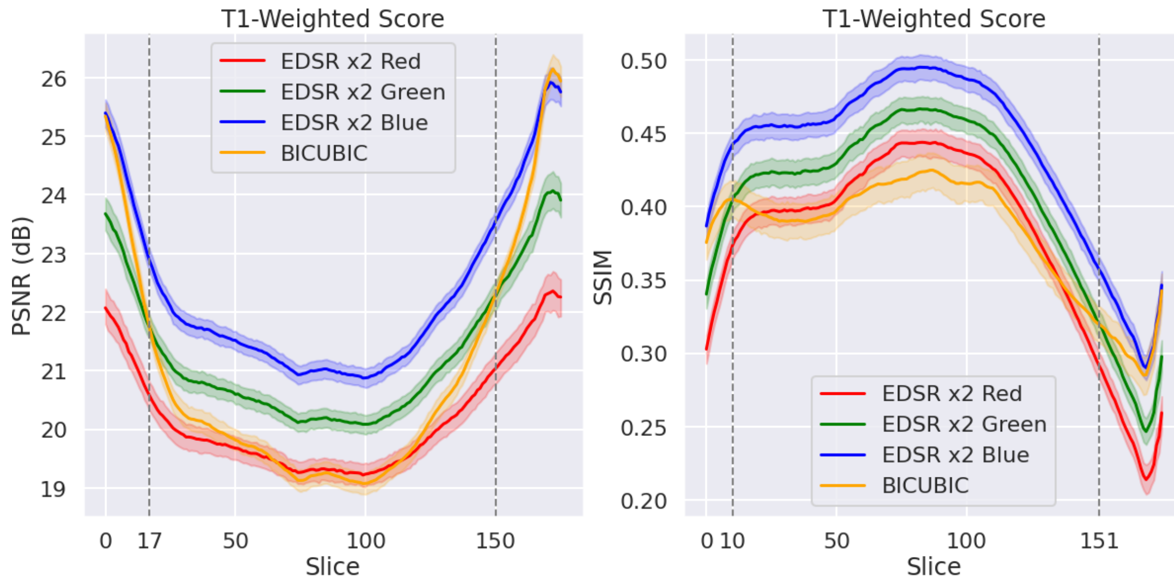


Figure 4.1: *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution EDSR model compared with the bicubic algorithm scores (Yellow) as functions of the slices. The average is performed for every patients and for every rotation, cosidering only T1 weighted NMR. The dotted lines highlights the slices where the bicubic and super-resolution green channel performances intersect.*

In figure 4.2 is shown an example on the kind of reconstruction the two methods are able to achieve on a significant slice:

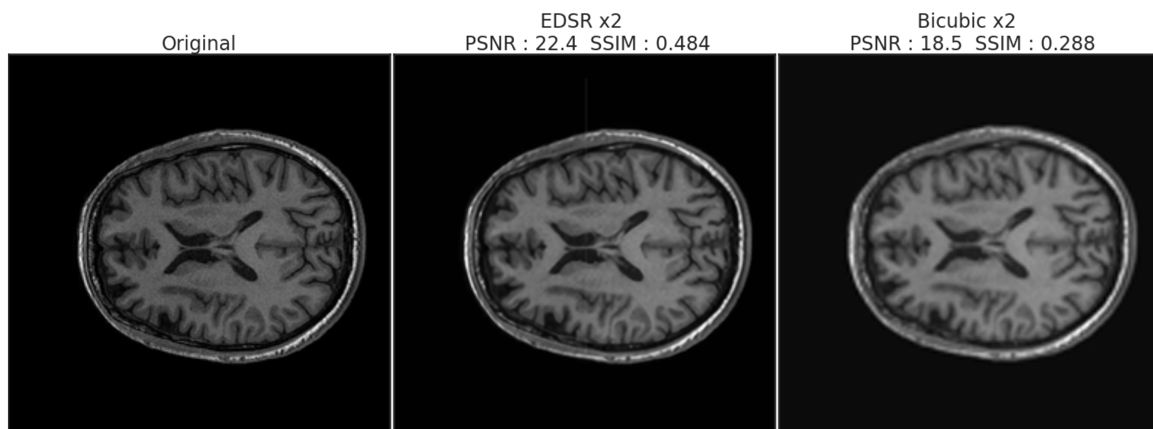


Figure 4.2: *(left) original image, (center) reconstruction performed with* `EDSR` *blue channel, (right) reconstruction using the bicubic method. The input in this case is not rotated.*

In the `EDSR` reconstruction there seems to be generally less noise, both around edges and more so in the gray area of the brain.

On the other hand, the results for T2-weighted NMRs are a bit different, indeed figure 4.3 highlights how the performances of EDSR are relatively worse compared with the previous case.

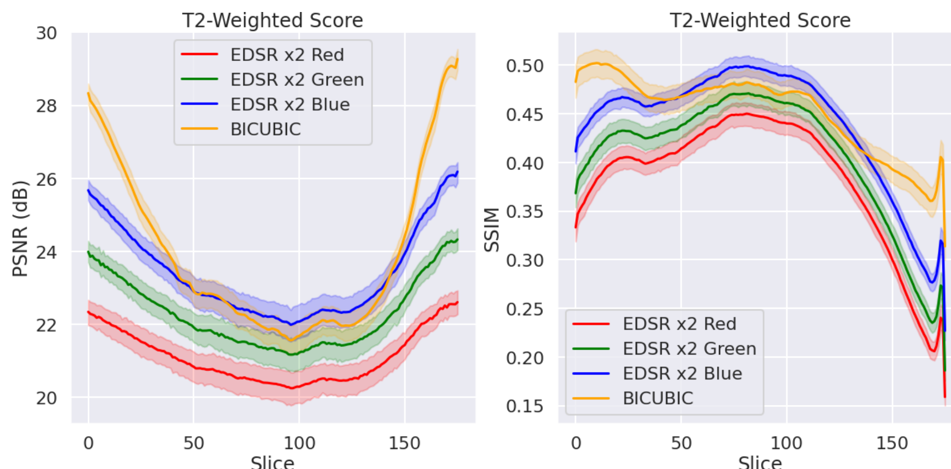


Figure 4.3: *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the super-resolution EDSR model compared with the bicubic algorithm scores (Yellow) as functions of the slices. The average is performed for every patients and for every rotation, considering only T2-weighted NMRs. In this case the bicubic seems to perform better, a part from the central section of the slice.*

in figure 4.4 is shown an example of the kind of reconstruction the methods can achieve on a significant slice:

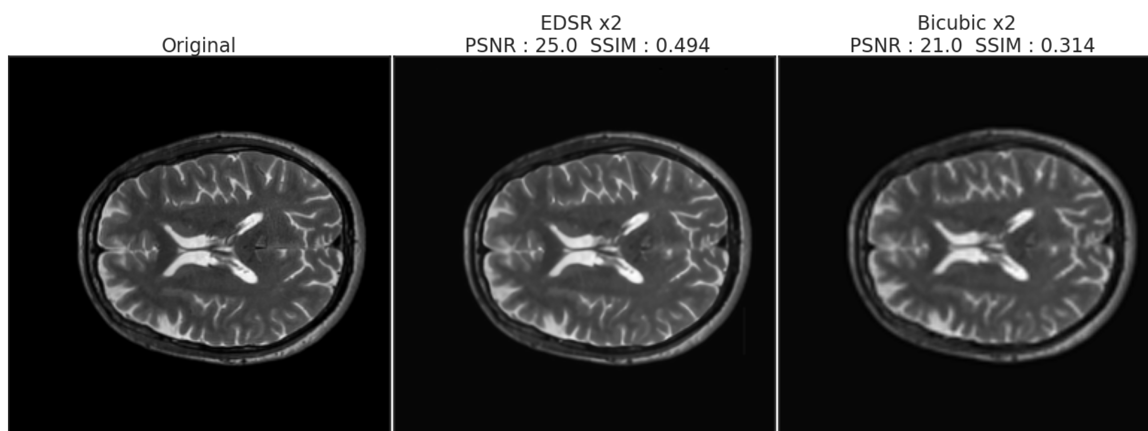


Figure 4.4: *(left) original image, (center) reconstruction performed with* `EDSR`*, (right) reconstruction using the bicubic method for a T2-weighted image. The input in this case is not rotated.*

In this particular case, the `EDSR` performed better that the bicubic algorithm, indeed the absolute difference image (in figure 4.15) indicates much less high frequency

component and a "smoother" visualization.

For the `WDSR` model, the three outputs are more consistents between each others and through the subjects they performs, on average, steadily better than the bicubic algorithm for T1-weighted NMR, as shown in figure 4.5:

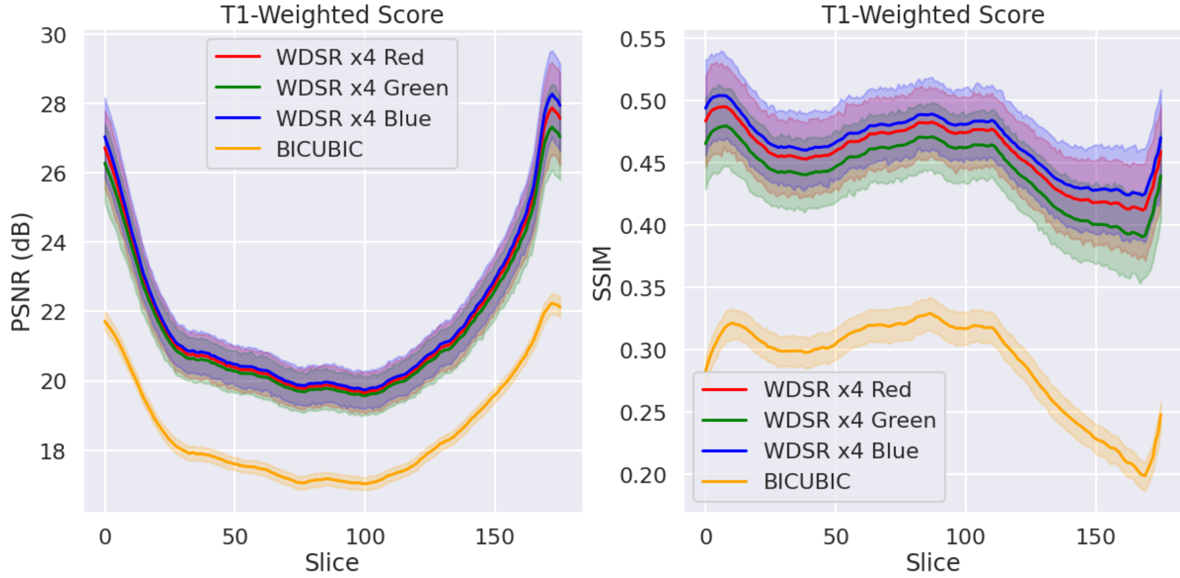


Figure 4.5: *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution WDSR model compared with the bicubic algorithm scores (Yellow) as functions of the slices. The average is performed for every patients and for every rotation, for T1-weighted NMRs.*

A recostruction performed starting from a donwscaled input 64×64 is shown in figure 4.6 for the two methods on a T1-weighted slice
The result is clearly worse than the `EDSR` cases since the $\times 4$ down-scaling is quite heavy. Indeed, the recostruction shows a lot of artifacts, particularly on the scalp of the subjects and they are more evident for the slices upscaled with the `WDSR`. In both cases the upsampling methods introduce a bias on the background of the image which can spoil the quantitative result. This effect can be explained if we consider that both models are not trained to upsample this kind of images, so the introduction of artifacts (such as background bias) may be a consequence of them trying to enhance the few signals gathered from the low-resolution inputs.

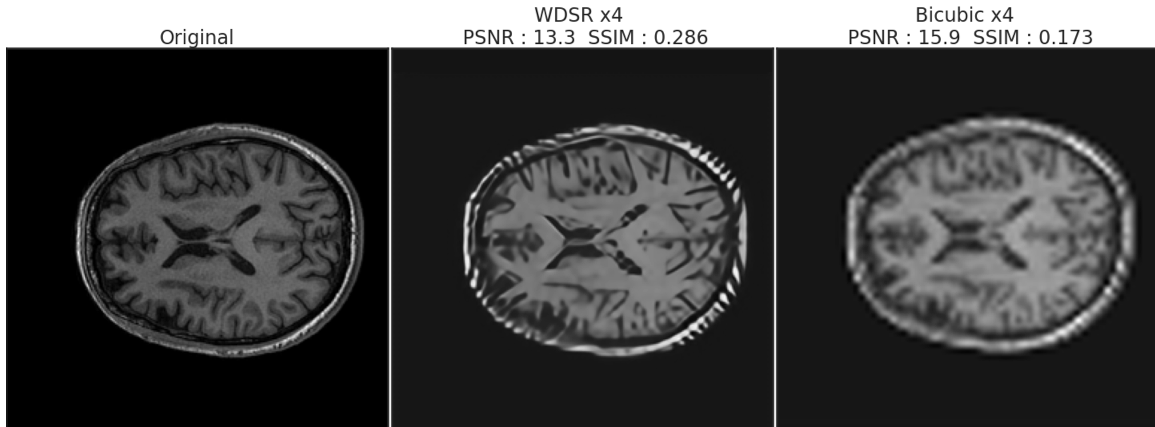


Figure 4.6: *(left) original image, (center) reconstruction performed with `WDSR`, (right) reconstruction using the bicubic method for a T1-weighted image. The input in this case is not rotated.*

The results for T2-weighted images do not change if we compare the Bicubic upsample with the `WDSR`, although it is possible to see that the scores are nearer to each others. The graphs are shown in figure 4.7.

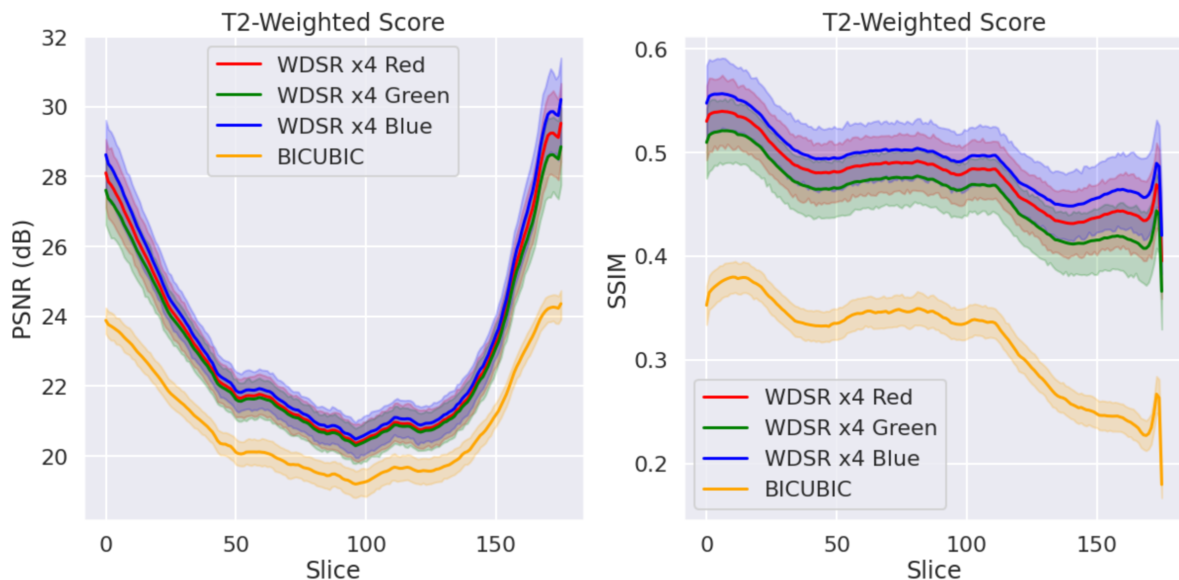


Figure 4.7: *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution WDSR model compared with the bicubic algorithm scores (Yellow) as functions of the slices. The average is performed for every patients and for every rotation, for T2-weighted NMRs.*

A recostruction performed starting from a donwscaled input 64×64 for a T2-weighted slice is shown in figure 4.8 for the two methods:

Figure 4.8: *(left) original image, (center) reconstruction performed with* `WDSR`*, (right) reconstruction using the bicubic method for a T2-weighted image. The input in this case is not rotated.*

The `WDSR` seems to obtain a smoother and less noisy than the bicubic up-sampling, both around edges and on Also, for T2-weighted images the number of artifacts in the scalp is reduced from figure 4.6, which shows a T1-weighted reconstruction. Moreover, the background is much more dark than the T1 reconstruction.

4.2 Scores by Angle

The next analysis focused on how the angle of rotation influences the final results. In figure 4.9 it is possible to see again how the scores greatly varies between SR channels, with the green one beeing the best performer. The average trends (mediated between all patients and slides) exhibit the presence of privileged angles for every method tested.

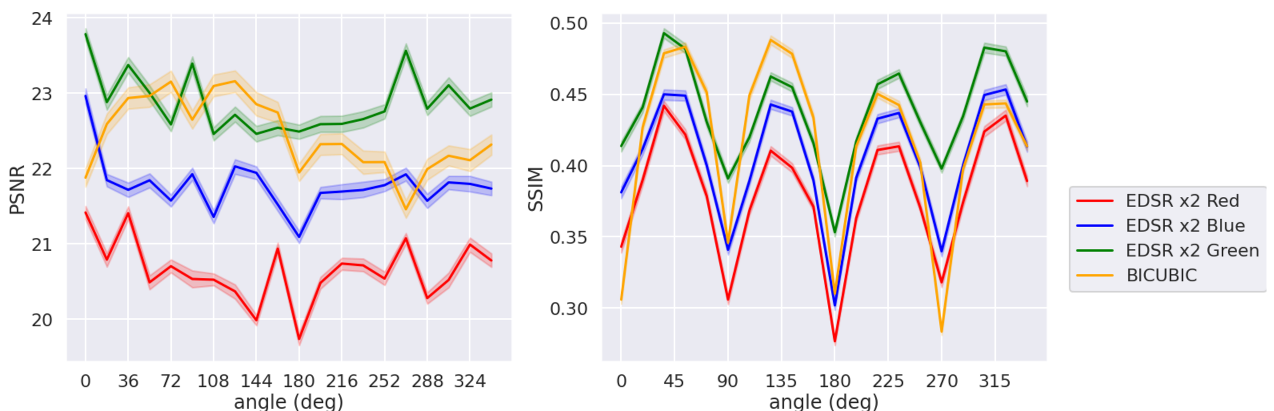


Figure 4.9: *average trends of PSNR (left) and SSIM (right) for* `EDSR` *x2 and bicubic algorithm as functions of the input angle of rotations.*

The `WDSR` shows a similar behaviour as the SSIM score in the previous case where angles different from $n\frac{\pi}{2}$ with $n = 0, 1, 2, 3$ performs better, as shown in figure 4.10.

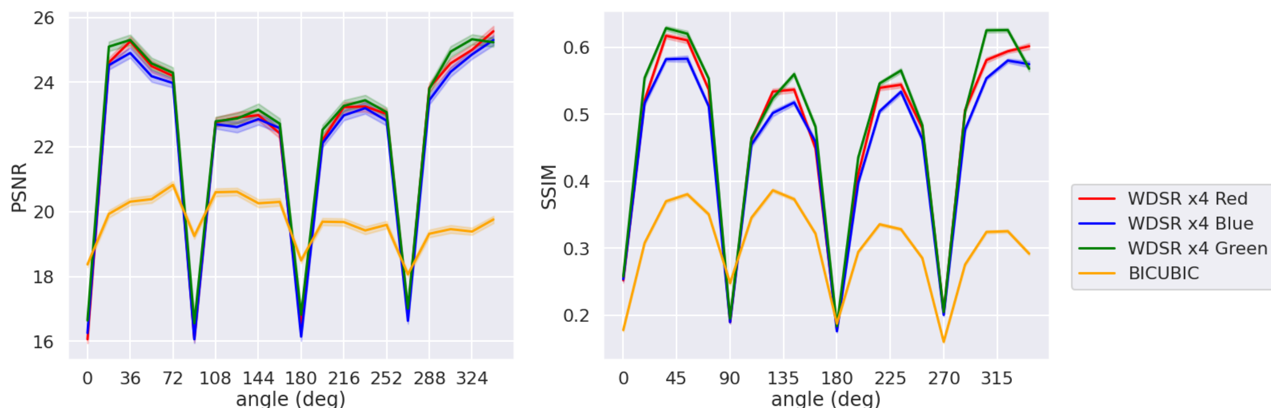


Figure 4.10: *average trends of PSNR (left) and SSIM (right) for* ***`WDSR`*** *x4 and bicubic algorithm as functions of the input angle of rotations.*

The results are congruos with what has been shown in the previous section in which the three channels (Red, Green and Blue lines) are consistent between themselves. In both cases there are angles for which the bicubic algorithm performs better than SR.

If we take a look at the images upscaled by x2 methods in figure 4.11, the difference between input angles is not really noticeable:

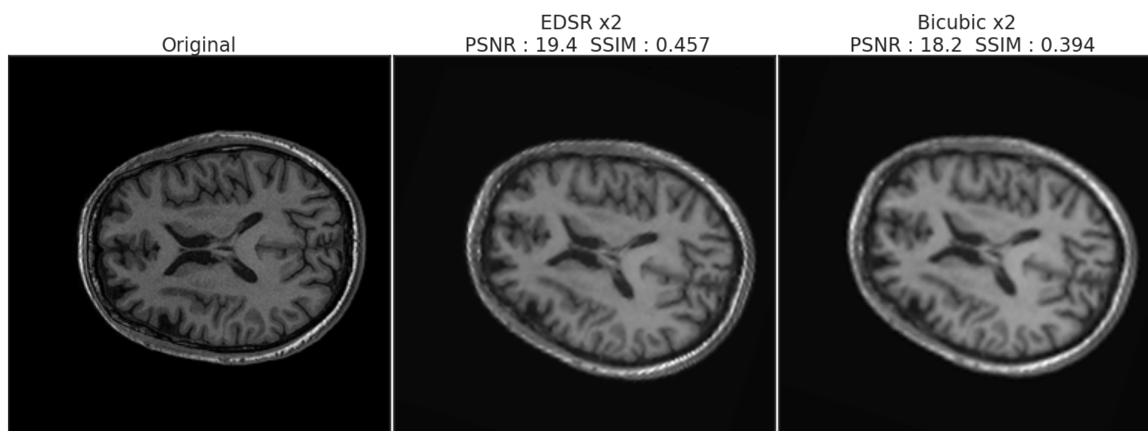


Figure 4.11: *comparison between* ***`EDSR`*** *x2 and bicubic, if compared with the images above, the level of recostruction is really similar to slices without rotations in figure 4.2*

On the other hand, the `WDSR` x4 upscaling shows far less artifact on angles different from $n\frac{\pi}{2}$ as can be seen in figure 4.12:

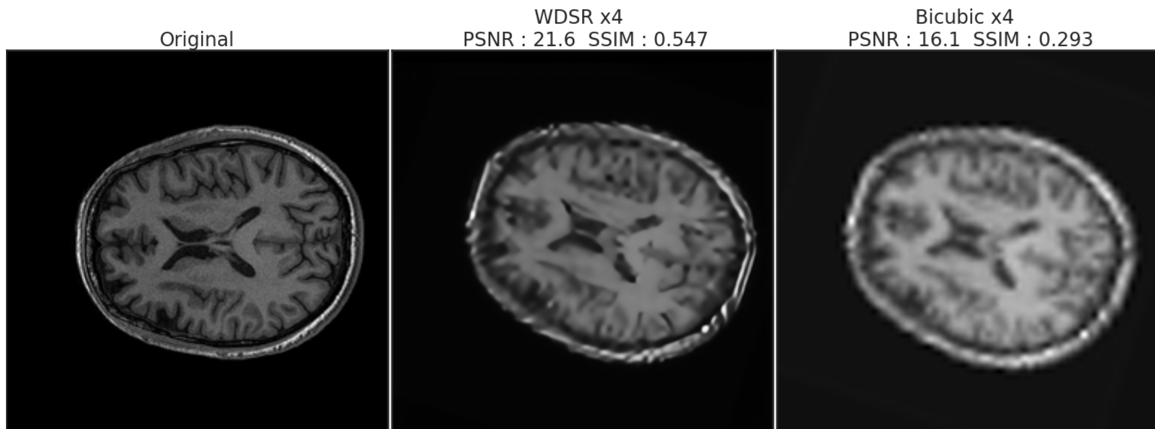


Figure 4.12: *comparison between `WDSR` x4 and bicubic, if compared with the images above, the level of recostruction is better than slices without rotations in figure 4.6. Indeed the number of artifacts seems to be far lower than before for `WDSR`.*

This can be due to different reasons: firstly, by rotating the images we technically performs an interpolation, particularly for angles differents from multiple of 90°; this can enhance or deteriorate lines and textures of the images and by results, strengthen or weaken the reconstruction of the methods. Secondly, the DIV2K dataset in which the models are trained and tested is intrinsically oriented: this means that if the images are not rotated during training, the models could have learnt a privileged angle of orientation for shapes and texture. Interestingly, as stated in [29] for `WDSR` and in [18] for `EDSR`, both models should have been trained by augmenting the dataset with randoms flips and rotations of the input images. Another possibility is the convolutional layers, which compose the majority of deep learning models, are not invariant for rotations, but only for transitions, that may introduces variations in the results.

4.3 Error localization

The following section will focus on where the error is localized in the images and how it is distributed. For doing that I computed the pixel-wise absolute difference between the original slice and the correspondent super-resolved one. As can be seen in figure 4.13 the major differences lie in the scalp of the subjects, which tends to be the most uninteresting part.

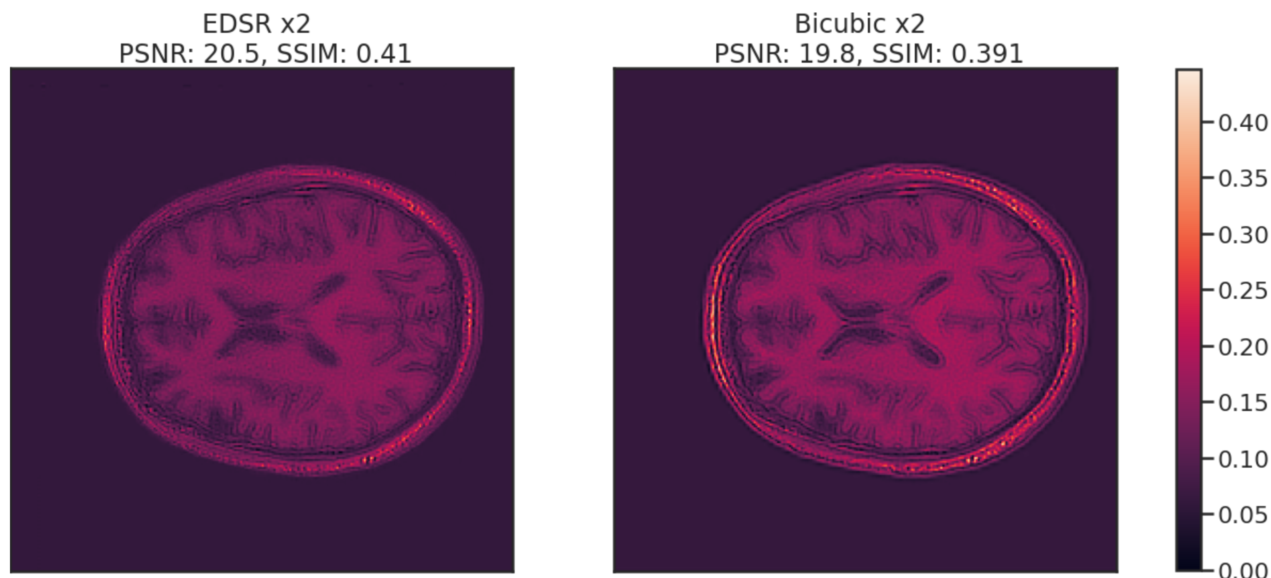


Figure 4.13: *Absolute differences for* **EDSR** *reconstruction (left) and bicubic (right), for a T1-weighted image. In both cases, the major differences seems to lies in the scalps of the subjecs. Though, it can be seen that the background is not zero, which means it has an impact on the scores.*

It is also clear by the differences, that the bias introduced by the two methods influences the results since there is a non-zero background which makes up a large part of the images. Indeed, by looking at the error distribution in figure 4.14, it appears that the background component is the most relevant part. However, by being prevalent on both methods, it does not influence the relative results. The second, and lower, peak of the two distributions represents the *brain component* of the images: by looking at them we can see that the distribution relative to the SR model is narrower and slightly shifted towards lower intensities, therefore justifying the better scores achieved by the model. Both histograms are mostly composed by values lower than 0: this means that the two models over-estimate most of the pixel values.

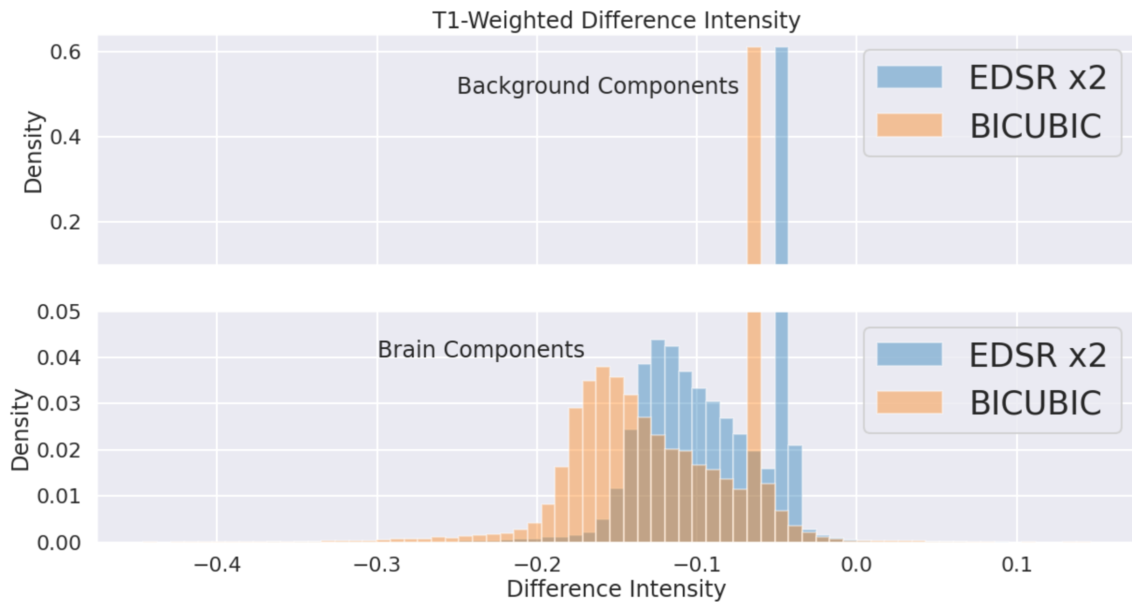


Figure 4.14: *Histograms of the distribution of differences pixel-by-pixel for the reconstruction performed by EDSR (Blue) and by the bicubic algorithm (Orange), for a T1-weighted NMR. The histogram has been cut between 0.05 and 0.1 on the y axis to better represent the lower parts.*

In the case of a T2-weighted sample, I show in figure 4.15 that the behaviour seems to be inversed: indeed this time the high frequency components of the differences are focused on the inner parts of the image and in particular around edges.

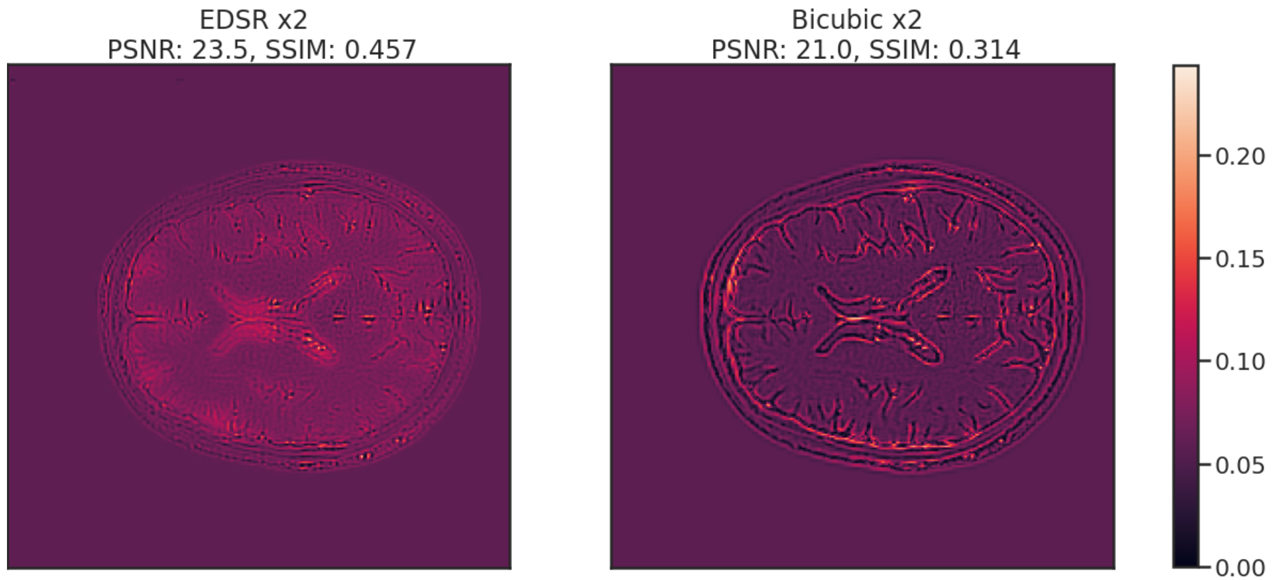


Figure 4.15: *Absolute differences of Super resolved image obtained with* **EDSR** *(left) and bicubic (right) for T2-weighted NMR. This time the major differences seem to be mostly located on the inner parts of the sample, in particular around edges.*

Also, the distribution of those differences is more concentrated around the peak of density, which, again, represents the background components, that is not zero. This concentration (and simmetry, in case of the bicubic upsample) is due to fact that there are much less white and gray pixels and a majority of dark-ish ones.

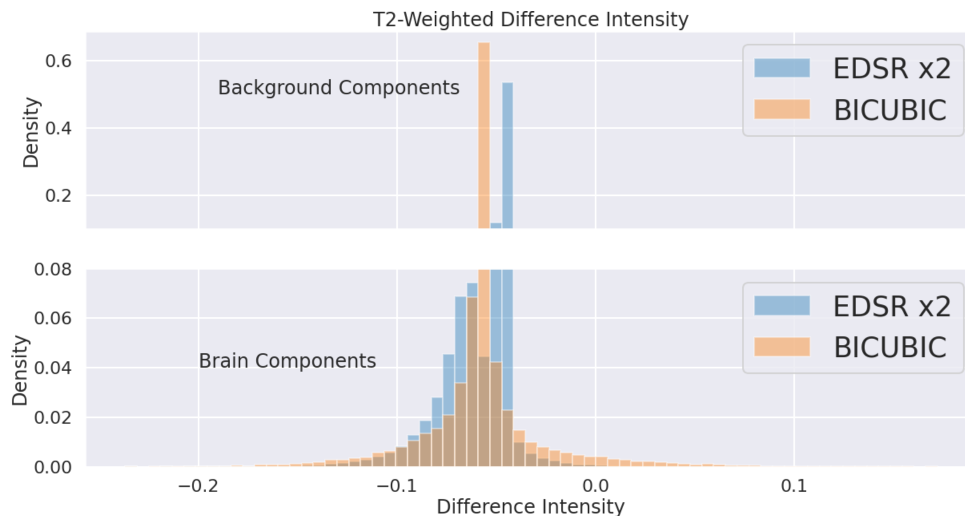


Figure 4.16: *Histograms of the distribution of differences pixel-by-pixel for a reconstruction performed by EDSR (Blue) and by the bicubic algorithm (Orange), for a T2-weighted NMR. The histogram has been cut between 0.08 and 0.1 on the y axis to better represent the lower parts.*

The same comparisons can be made also for `WDSR` and in general for an up-sampling scale factor of $\times 4$. In figure 4.17 I show the absolute differences of a T1 sample upscaled from 64×64 to HR.

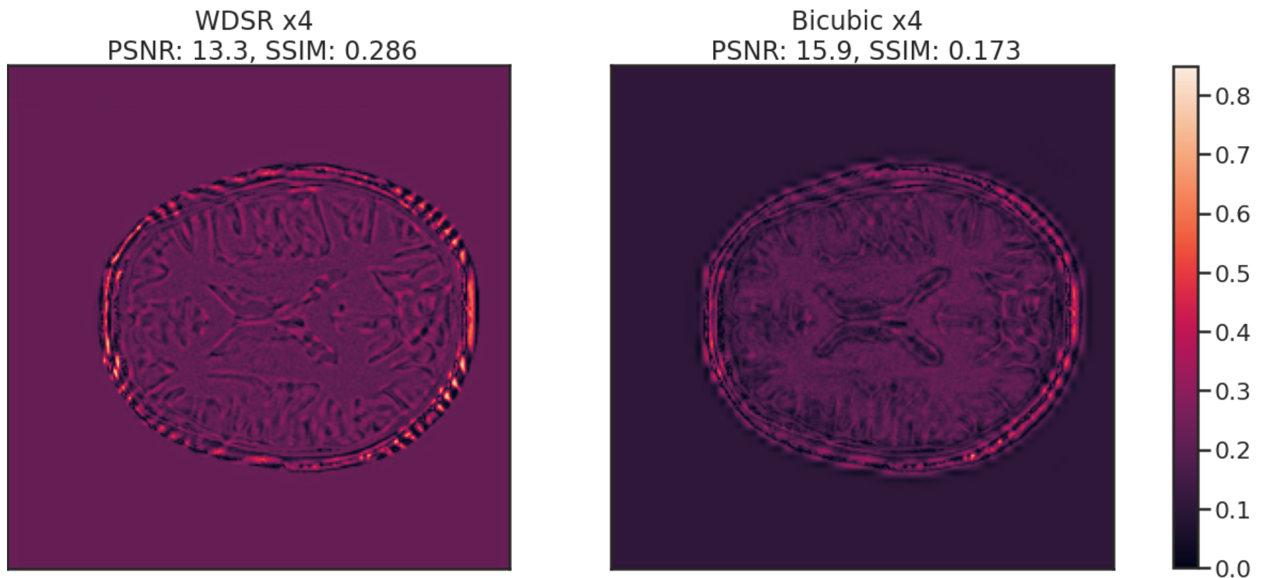


Figure 4.17: *Absolute differences of Super resolved images obtained with `WDSR` (left) and bicubic (right) for a T1-weighted NMR. Also in this case the major differences seems to be focused on the scalp of the subjects.*

First of all, also in this case the major differences are located around the scalp, this visualization also highlights better the artifacts created during the upsampling for both reconstructions. Another problem, is the strong background components of the `WDSR`. upsampling, which is much more evident than in the bicubic case. In the histograms in figure 4.18 is shown the distributions of the differences of the two reconstructions:

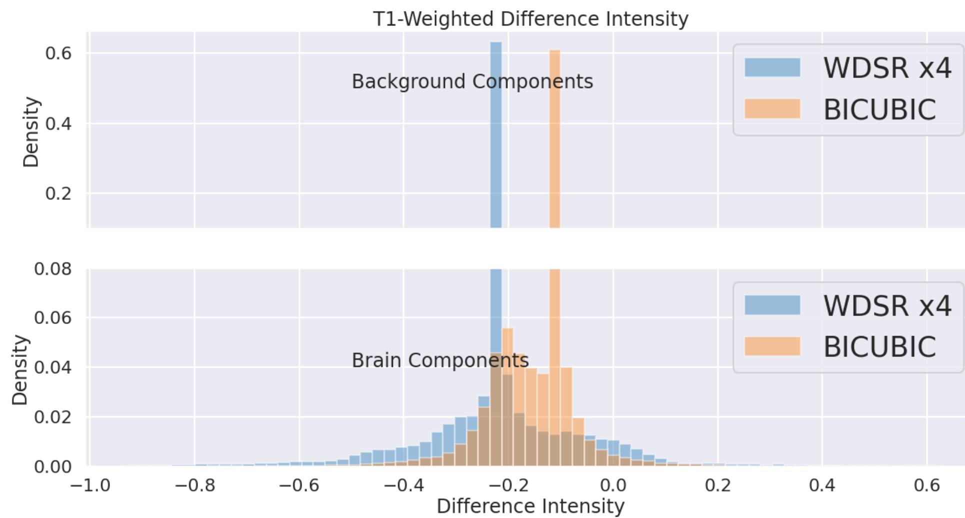


Figure 4.18: *Histograms of the distribution of differences pixel-by-pixel for a reconstruction performed by EDSR (Blue) and by the bicubic algorithm (Orange), for a T2-weighted NMR. The histogram has been cut between 0.08 and 0.1 on the y axis to better represent the lower parts.*

The same considerations that have been expressed for the previous cases, can be made also in this one: by looking at the distributions, both models over-estimate the majority of pixels in the images.

In figure 4.19 is shown the absolute difference between the original images and the two $\times 4$ reconstructions performed respectively by `WDSR` and the bicubic.

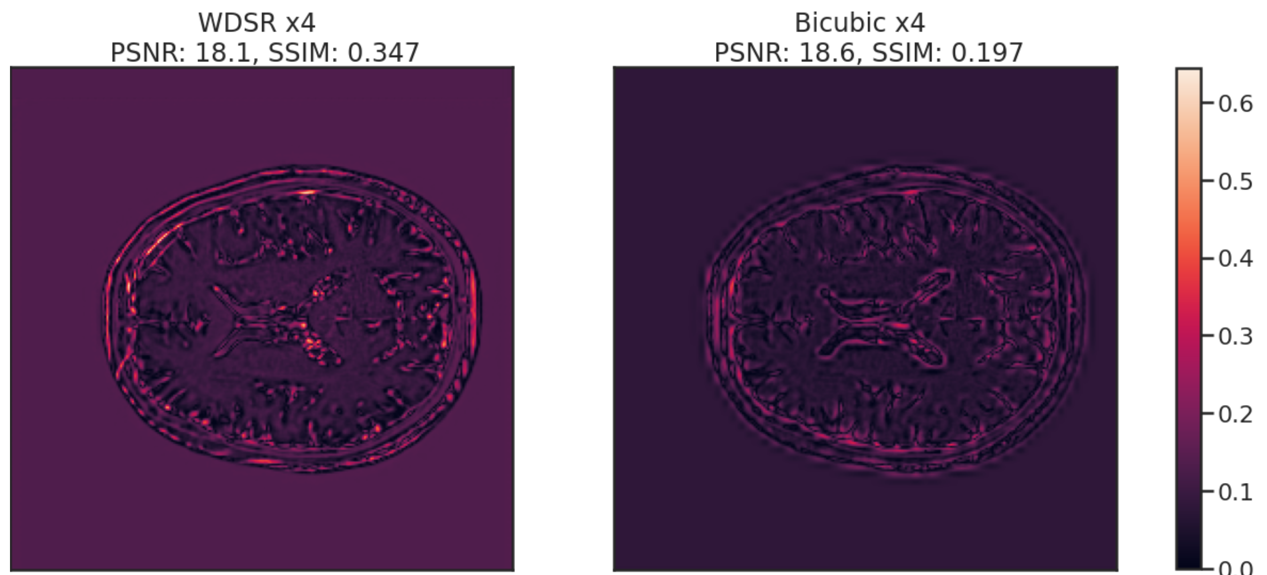


Figure 4.19: *Absolute differences of Super resolved images obtained with* `WDSR` *(left) and bicubic (right) for a T2-weighted NMR.*

In figure 4.20 is shown the relative histograms of distributions of differences for the slice shown above.

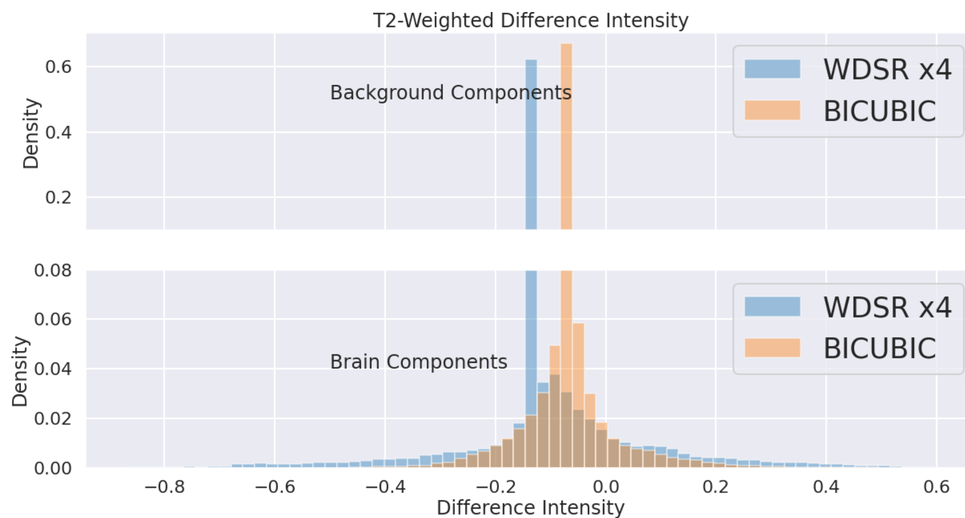


Figure 4.20: *Histograms of the distribution of differences pixel-by-pixel for a reconstruction performed by EDSR (Blue) and by the bicubic algorithm (Orange), for a T2-weighted NMR. The histogram has been cut between 0.08 and 0.1 on the y axis to better represent the lower parts.*

In this case the distribution seems to be more spreaded and almost symmetric around the peak, again sligtly shifted towards negatives values, which indicates an overestimation of pixel gray scale values for the two reconstructions.

The next analysis will focus on the study of how reconstructions performs with a standard post-processing tool for NMR and on the analysis of the same data by removing the background.

4.4 Brain Extraction

To avoid background related problems, I extended the analysis to images where the background and the scalp were removed with FSL BET (Brain Extraction Tool)[26]. FSL is a standard post-processing tool for NMR, it works by creating a binary mask for every slice and then applies it to the images: in this way is possible to extract only the relevant information from data. A binary mask is a tensor of zeros and ones with the same dimension as the original NMR, the "ones" correspond with the pixel we want to preserve. An example is shown in figure 4.21:

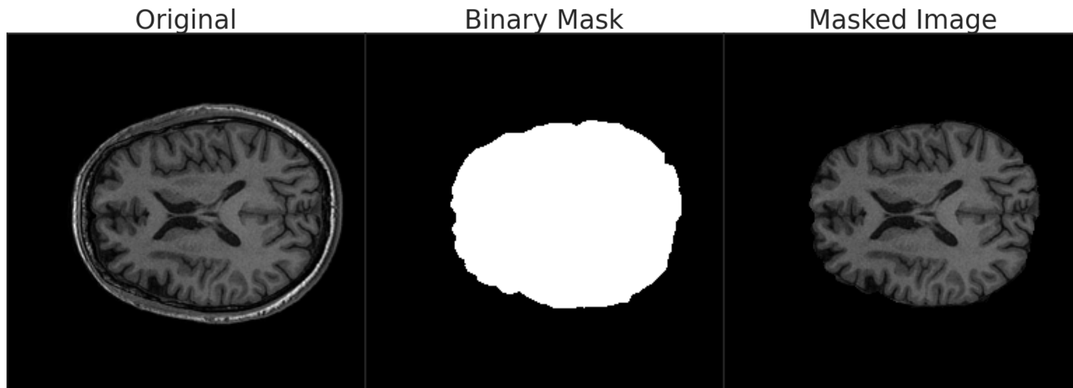


Figure 4.21: *Example of application of a mask on one of the slices of the original 3D map. (left) Original image, (center) Mask obtained from FSL BET and (right) Mask applyed to the original image.*

I reported two different approaches for a single patient: firstly, three different binary masks for original images, Super Resolved reconstructions and bicubic ones were extracted. In this way it is possible to quantify how well a standard post processing tool like BET performs on images obtained by upsampling. Secondly, I masked all three reconstruction with the same binary mask, obtained from the original high resolution NMR, in this way it is possible to compare directly the reconstruction capabilities of the methods, without including uninteresting data. The first analysis how well BET works on the reconstructed images, therefore I will consider distinct masks applied separately to the results. As shown in figure 4.22, Brain Extraction Tool not only removes the background, but also the entire scalp, which was shown to contains the main differences from the original image.

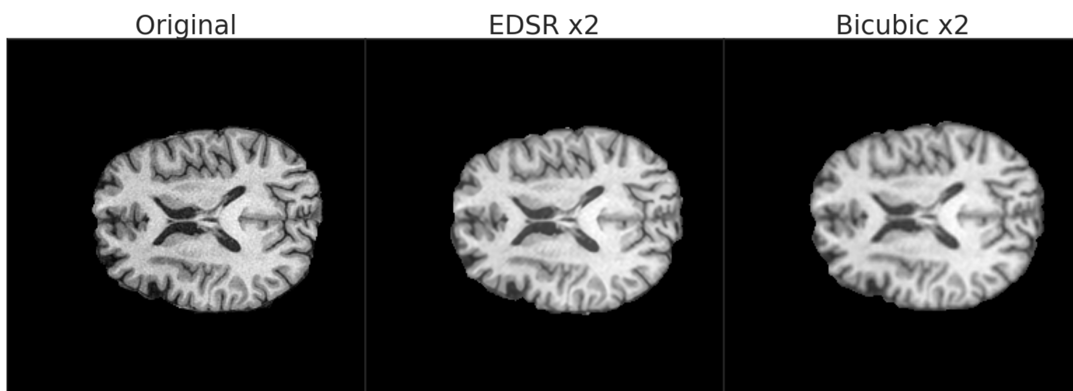


Figure 4.22: *Examples of Brain Extraction for original image (left), EDSR image (center) and Bicubic results (right). The binary masks are different for the three images.*

In figures 4.23 and 4.24 the differences pixel by pixel between super resolution's results and the original image are reported side-by-side with the corresponding Bicubic results.

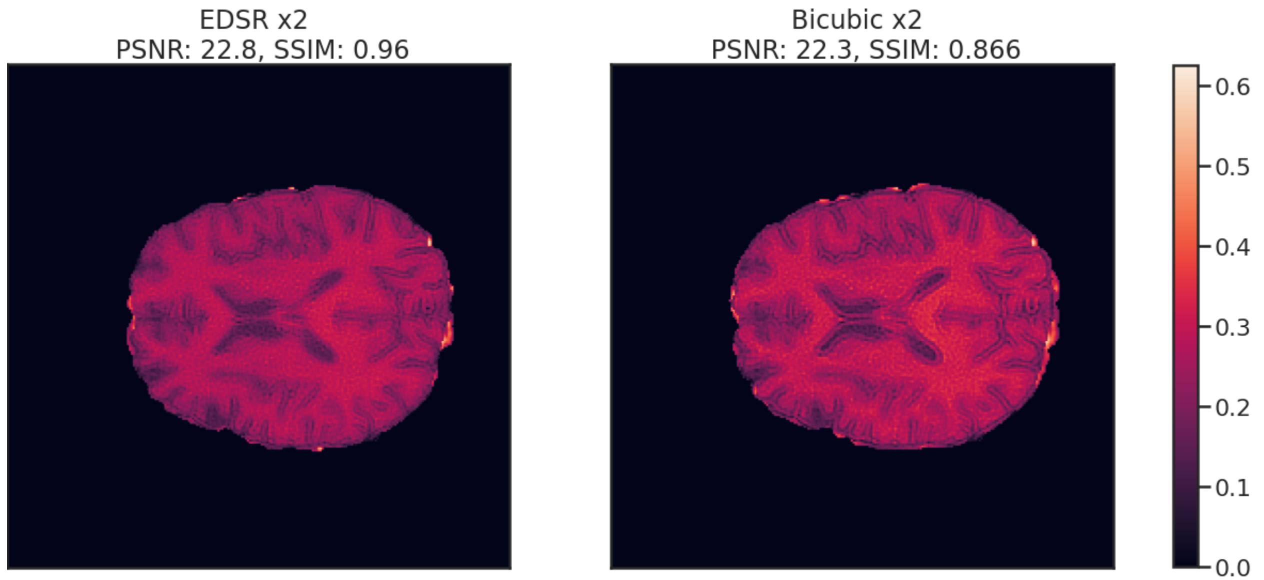


Figure 4.23: *Absolute differences for Super Resolution (EDSR, left) and bicubic (right). The images are obtained by computing a pixel-wise absolute difference with the original slice. We can see the main dissimilarities on the border of the two images, meaning that the BET didn't catch all the relevant informations on the reconstructed images.*

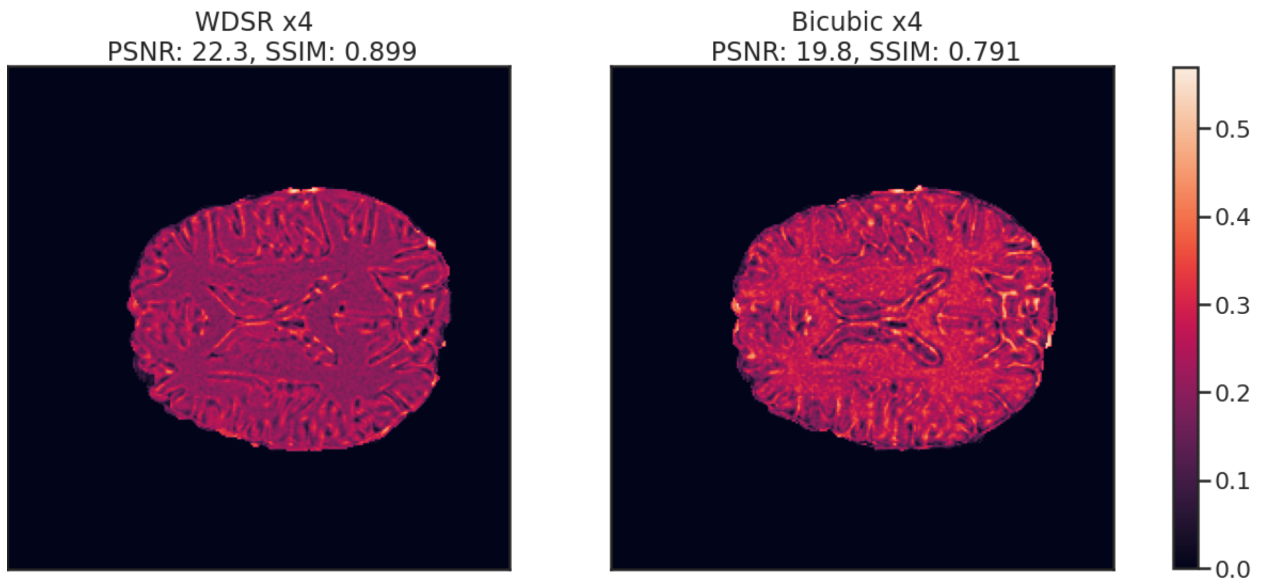


Figure 4.24: *Absolute differences for Super Resolution `WDSR`, left) and bicubic (right). In this case, the principal dissimilarities are not focused only on the outer parts of the brain and that is understandable, given the high level of downsampling.*

In the first case (figure 4.23) we can see the main dissimilarities are on the border of the two brains, meaning that BET did not catch all the relevant informations on

the reconstructed sample or, otherwise, the up-sampling highlighted . In any case, there some differences between masks, which can be quantified with an *Intersection over Union* (IoU), as described below.

In the second case (figure 4.24), the principal dissimilarities are not focused only on the outer parts of the brain, accordingly to what have been shown before, and that is understandable, given the high level of downsampling. Again there some discrepancies between images aroun the border, meaning that during re-upsample some of the informations are lost or over-amplified with respect to the original image.

The histogram of the distribution of differences in figure 4.25 shows how the background component is now shifted towards 0, implying that is not considered in the score computations. The distributions also shows a spread and a shift towards higher differences of the brain component: this is probably due to the higher contrast in the images obtained by brain extraction.

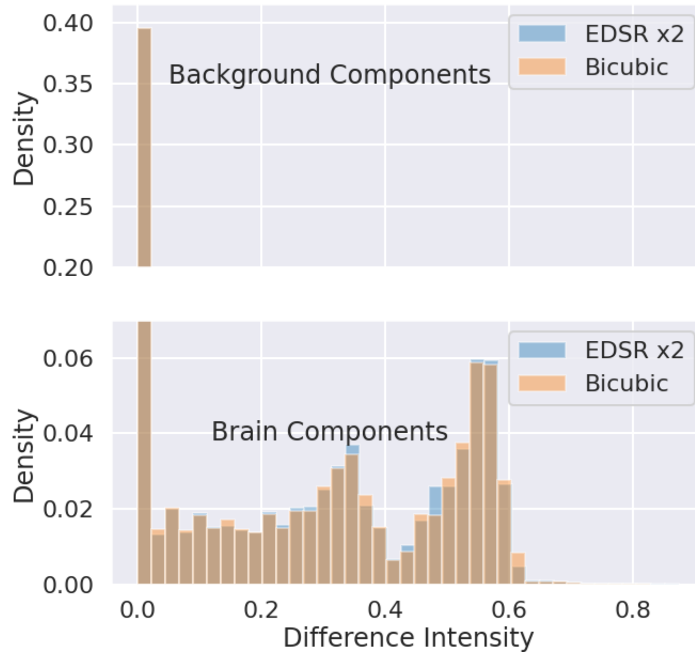


Figure 4.25: *Histogram of differences for an image obtained with x2 up-sampling and after Brain Extraction. The background componenent is completely shifted towards zero, while the Brain componentent spreaded and shifted toward higher intensity for contrast manipulation. The distributions for the two methods are nearly identical. The histogram has been cut on the y axis between 0.07 and 0.2 to better represent the Brain Component*

As a last evaluation I considered the Intersection over Union (IoU) score, which quantifies the overlap between different masks. The IoU is defined as:

$$IoU = \frac{A \cap B}{A \cup B} \tag{4.1}$$

where A, in this case, is the binary mask obtained by the original slice, while B is the mask obtained by one of the upsampling methods. The trends of IoU for the two models compared with the bicubic algorithm can be found in figure 4.26

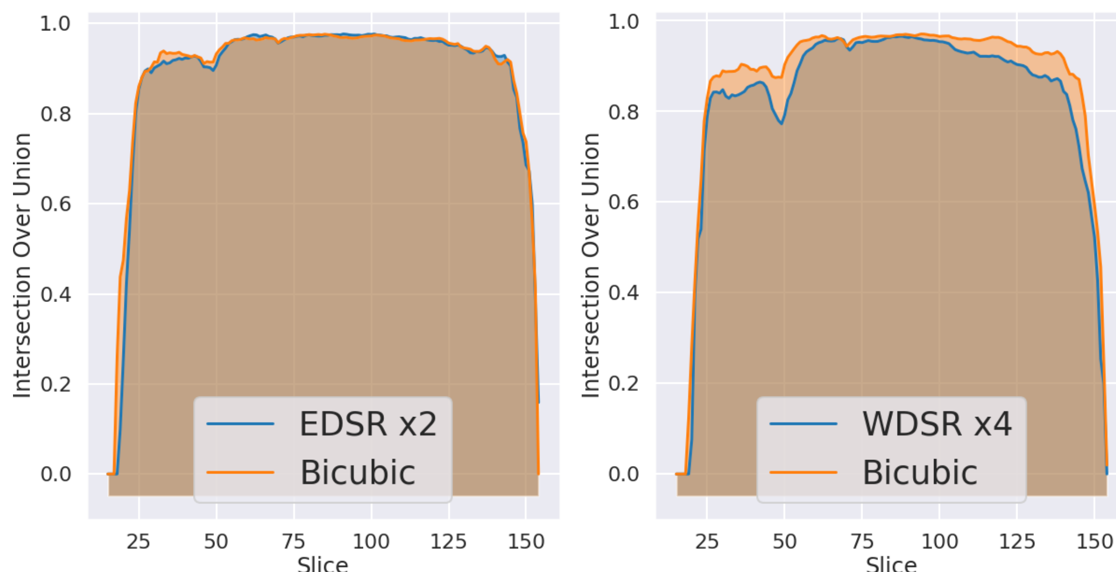


Figure 4.26: *Trends of IoU scores for the different upsamplig methods. (left) comparison between the mask obtained from* `EDSR` *and bicubic recostructions and (right) same graph for* `WDSR` *and bicubic. Notice that some slices are cut from comparisons since they are black images and both intersection and union are 0.*

The comparisons between IoU in the `EDSR` do not shows much of a difference with the bicubic algorithm. On the other hand, `WDSR` seems to be slightly penalized: that is most likely due to the artifacts present in the scalp of the subjects shown in figure 4.6 interfering with the brain extraction, which may makes the super-resolution model with this level of down-sampling less adapt to work with BET-like softwares
The second analysis proposed in this section regards how the upsampling methods performs considering the same mask applied for all the reconstruction: so that I can compute the scores considering only the same portion of the images, without worriying about background effects. Since the original High Resolution NMR is supposed to represents an objective "truth" for our study, the best available mask is of course the one acquired from its brain extraction. For `EDSR`, the results in figure 4.27 shows the average trends of PSNR and SSIM pre and post brain extraction for the selected patient, mediated over the rotations angle.

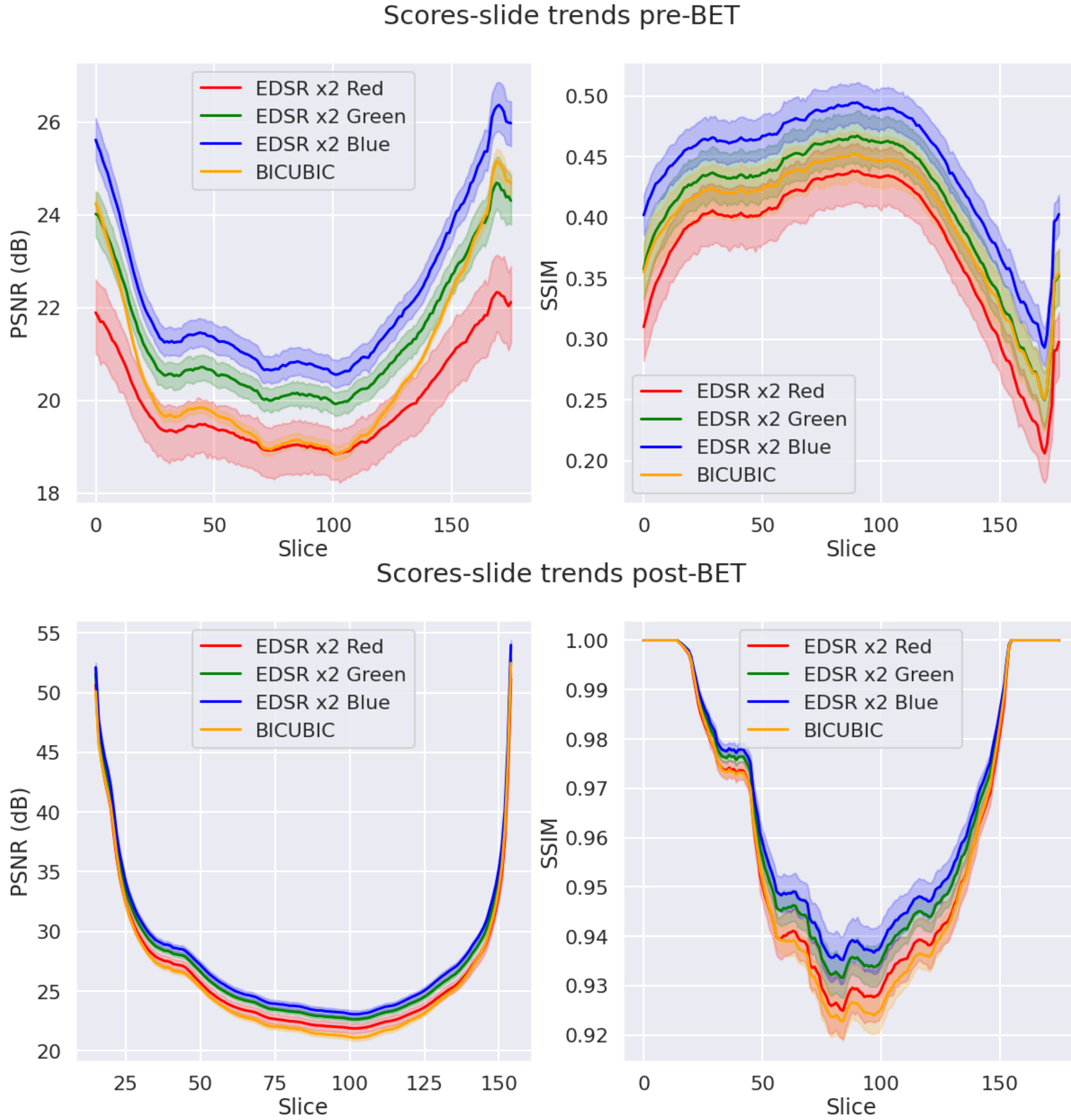


Figure 4.27: *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution* ***EDSR*** *model compared with the bicubic algorithm scores (Yellow) as functions of the slices. Above are the results pre brain extraction while below the ones post brain extraction. The average is performed for one patient and for every rotation, for T1-weighted NMRs.*

By comparing the pre and post BET performances, it is clear that the differences between channels become much lower, which may means that the discrepancies measured in the pre BET trends comes from artifacts and backgrounds related to "false-features" extrapolation which are different between channels and mitigated (or entirely removed) when masking the slices. However, at same time, the score discrepancies between bicubic and SR also become much lower than pre-BET, meaning that, on average, the background

components does favour the Super-Resolution, but still `EDSR` beats the upsampling of the bicubic by a fair amount after brain extraction.
The same comparisons for `WDSR` in figure 4.28 shows very similar results:

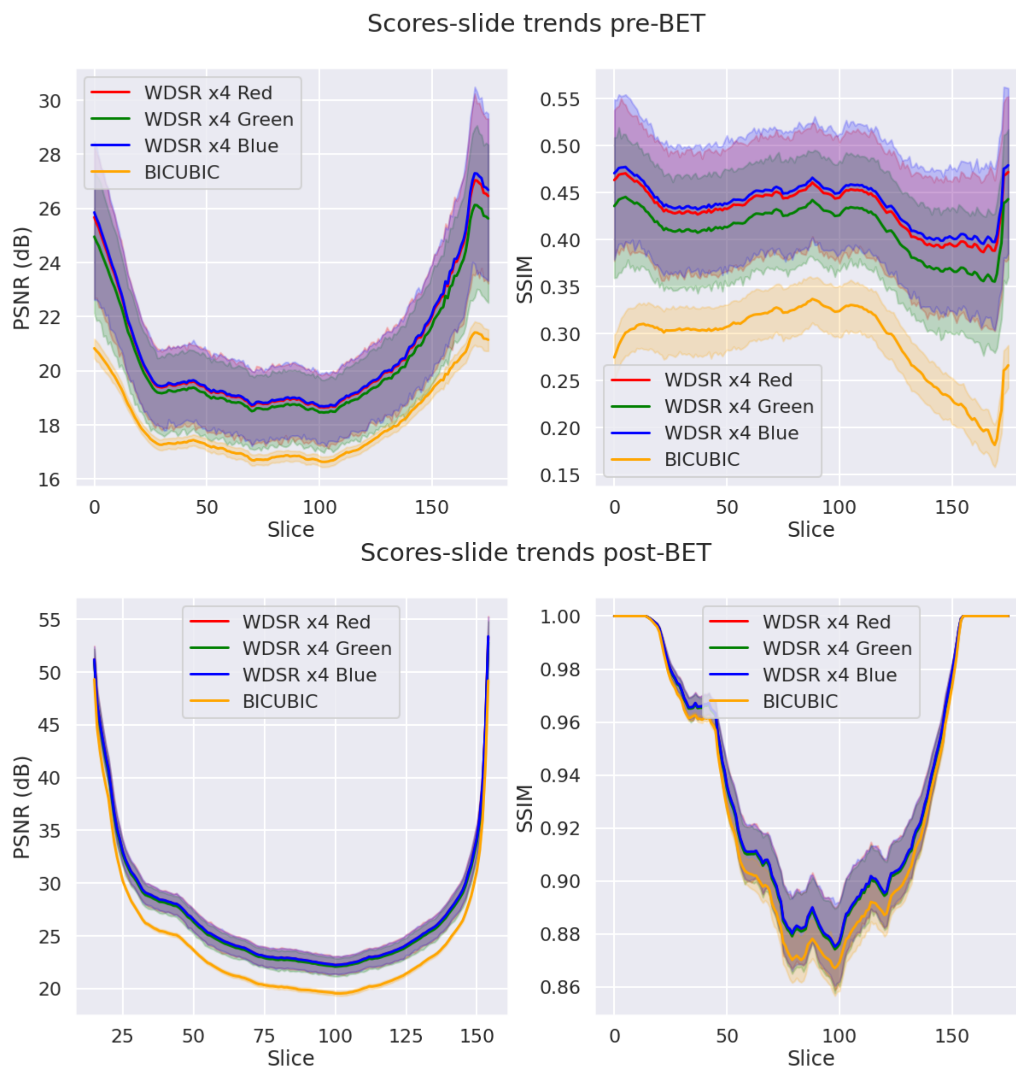


Figure 4.28: *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution* ***`WDSR`*** *model compared with the bicubic algorithm scores (Yellow) as functions of the slices. Above are the results pre brain extraction while below the ones post brain extraction. The average is performed for one patient and for every rotation, for T1-weighted NMRs.*

Indeed, even in this case, the SR performs better than the bicubic algorithm for all three channels, which are very consistent with each others. This may indicates that the differences between channels derived by what each of them extrapolate from the background, in terms of bias and artifacts.

Chapter 5

Conclusions

In this work I developed two novel frameworks for Deep Learning called `Byron` in C++ and `NumPyNet` in Python, the former focused on CPU optimizatioon while the latter focused on readability. I tested the computational performances of `Byron` for the most important layers in image analysis against the python library `Tensorflow`.
Moreover, the work extended to implementing and testing two models for super-resolution, `EDSR` and `WDSR`, on NMR images, which represent a new benchmark for those deep networks, originally trained and tested only on `DIV2K`. On average, the models show better reconstruction capabilities on the most interesting parts of the brain compared with the bicubic algorithm, although they introduce artifacts and biases: this may be a consequence of them trying to enhance the few signals gathered from low-resolution input and "seeing" false-features inside the background of the slices. Testing `EDSR`, an RGB model, on a gray-scale dataset also highlighted great variability between the three output channels, while `WDSR` seems to be more stable in this respect. On the other hand though, while `EDSR` results do not exhibit visible differences across angle of rotations, `WDSR`'s ones has been shown to improve for rotations different from multiples of 90°, possibly showing that its action is not invariant with rotations. The localization of errors through a pixel-wise absolute difference between reconstructions and high-resolution NMRs indicates that for both algorithms the major discrepancies are localized around the scalps of the subjects, which tends to be the most uninteresting section of the brains. Although, they also highlight that the background components of the images are Thus, the brain extraction has been helpful in removing both background effects and the high level of discrepancy in the scalp. This, in turn, lowered the differencies of performances between super-resolution RGB channels and bicubic algorithm for both models `EDSR` and `WDSR`, while also showing that the SR still shows promising results in terms of reconstruction of biomedical iamge, which is an indication of how well this models can generalize the "knowledge" acquired during training on a completely different dataset such as DIV2K, which does not contain any data about NMRs or other kind of biomedical image.
Some future developments for the work may include:

- An exhaustive algorithm for training in `Byron` to investigate the concept of *transfer*

learning: namely, storing knowledge gained while solving a problem and re-applying it to a different, but related, task. In this case, we can use the same dataset of NMR to continue the training of `EDSR` and `WDSR` starting from the parameters obtained by the respective authors. Of course, the networks can also be re-trained from scratch on NMR images only and compare the results.

- The implementation of both `Byron` and `NumPyNet` for GPUs to exploit at best every hardware.
- Extension of the analysis after Brain Extraction for all the patients and angle in order to remove artifacts and background biases which represented a major issue during evaluation.
- The timing sections in the second chapter shows that there still work to be done in the optimization of some of the layers presented for `Byron`.

List of Figures

1.1 *A common representation of a neural network: a single node works as the perceptron described before. The network is composed by the input layer, the hidden layers and the output layer. The depth of the network is determined by the number of hidden layers.* . . . 6

1.2 *Visual example of gradient descent for a model with 2 weights. The idea is to modify the weights to follow the direction of the steepest descent for the landscape of the error function* . . . 7

1.3 *Comparison between a T1-weighted slice (left) and a T2-weighted slice (right) for the same patient* . . . 13

2.1 *Visual representation of a convolution of an Image I with a kernel of size 3* 17

2.2 *Scheme of the im2col algorithm using a* $2 \times 2 \times 3$ *filter with stride 1 on a* $4 \times 4 \times 3$ *image. The matrix multiplication is between a* $n \times 12$ *and a* 12×9 *matrixes.* . . . 19

2.3 *Scheme of maxpool operations with a kernel of size* 2×2 *and stride* 2 *over an image of size* 4×4. *Picture from CS231n* . . . 21

2.4 *Average pooling applied to a test image: (left) the original image, (center) average pooling with a* 3×3 *kernel, (right) average pooling with a* 30×30 *kernel. The images have been obtained using NumPyNet* . . . 21

2.5 *Max pooling applied to a test image: (left) the original image, (center) max pooling with a* 30×30 *kernel and stride* 20, *(right) max pooling errors image. Only few of the pixels are responsible for the error backpropagation. The images have been obtained using NumPyNet* . . . 23

2.6 *Scheme of the shortcut layer as designed by the authors [12]. The output of the second layer become a linear combination of the input x and its own output.* . . . 24

2.7 *example of deconvolution: (left) a normal convolution with size 3 and stride 1, (right) after applying a "zeros upsampling" the convolution of size 3 and stride 1 become a deconvolution* . . . 26

2.8 *Example of pixel shuffling proposed by the authors [25]. In this example,* r^2 *features maps are re-arranged into a single-channeled high resolution output.* . . . 26

2.9 *Activation functions applied on test image. From top to bottom: Elu, Relu and Logistic.* 31
2.10 *Comparisons between time to perform forward (left) and backward (right) for Convolutional Layer in Byron and Tensorflow on a 4D tensor of size* $16 \times 512 \times 512 \times 3$ *for size 3, stride 1 and 100 filters.* 33
2.11 *Comparisons between time to perform forward (left) and backward (right) for Convolutional Layer in Byron and Tensorflow on a 4D tensor of size* $16 \times 512 \times 512 \times 3$ *for size 3, stride 2 and 100 filters.* 34
2.12 *Comparisons between times to perform forward (left) and backward (right) for Maxpool Layer in Byron and Tensorflow on a 4D tensor of size* $16 \times 512 \times 3$ *for size 3 and stride 1.* 34
2.13 *Comparisons between times to perform forward (left) and backward (right) for Maxpool Layer in Byron and Tensorflow on a 4D tensor of size* $16 \times 512 \times 3$ *for size 30 and stride 1.* 35
2.14 *Comparisons between times to perform forward (left) and backward (right) for Shuffler Layer in Byron and Tensorflow on a 4D tensor of size* $16 \times 512 \times 108$ *for scale 6* 36

3.1 *architecture of the single scale SR Network (EDSR)* 38
3.2 *comparisons between* **EDSR** *architecture on the left and* **WDSR** *architecture on the right* 40
3.3 *Super Resolution visual example extracted from the DIV2K validation set. The quality score in terms of PSNR and SSIM are compared between a standard bi-cubic up-sampling and the EDSR and WDSR models.* 41
3.4 *Super Resolution visual example extracted from the DIV2K validation set. The quality score in terms of PSNR and SSIM are compared between a standard bi-cubic up-sampling and the EDSR and WDSR models.* 42
3.5 *Super Resolution visual example extracted from the DIV2K validation set. The quality score in terms of PSNR and SSIM are compared between a standard bi-cubic up-sampling and the EDSR and WDSR models.* 42
3.6 *HR* 256×256 *original image at three different stages of depth: (left) slice 30 where still a lot of information about the brain is hidden, (center) slice 100 which is a central slice where most of the information is stored, (right) slice 150 which starts the less informative area of the brain.* 43
3.7 128×128 *LR version of the same slices shown for the HR case.* 44
3.8 64×64 *LR version of the same slices shown for the HR case.* 44
3.9 *Three of the 20 rotation angles used as input for Super Resolution models and Bicubic. (left) reference angle of 0 degree, (centre) angle step of 18 degree, (right) large rotation of 108° respect to the reference.* 45

4.1 *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution EDSR model compared with the bicubic algorithm scores (Yellow) as functions of the slices. The average is performed for every patients and for every rotation, cosidering only T1 weighted NMR. The dotted lines highlights the slices where the bicubic and super-resolution green channel performances intersect.* 47
4.2 *(left) original image, (center) reconstruction performed with* `EDSR` *blue channel, (right) reconstruction using the bicubic method. The input in this case is not rotated.* 47
4.3 *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the super-resolution EDSR model compared with the bicubic algorithm scores (Yellow) as functions of the slices. The average is performed for every patients and for every rotation, considering only T2-weighted NMRs. In this case the bicubic seems to perform better, a part from the central section of the slice.* 48
4.4 *(left) original image, (center) reconstruction performed with* `EDSR`*, (right) reconstruction using the bicubic method for a T2-weighted image. The input in this case is not rotated.* 48
4.5 *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution WDSR model compared with the bicubic algorithm scores (Yellow) as functions of the slices. The average is performed for every patients and for every rotation, for T1-weighted NMRs.* 49
4.6 *(left) original image, (center) reconstruction performed with* `WDSR`*, (right) reconstruction using the bicubic method for a T1-weighted image. The input in this case is not rotated.* 50
4.7 *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution WDSR model compared with the bicubic algorithm scores (Yellow) as functions of the slices. The average is performed for every patients and for every rotation, for T2-weighted NMRs.* 50
4.8 *(left) original image, (center) reconstruction performed with* `WDSR`*, (right) reconstruction using the bicubic method for a T2-weighted image. The input in this case is not rotated.* 51
4.9 *average trends of PSNR (left) and SSIM (right) for* `EDSR` *x2 and bicubic algorithm as functions of the input angle of rotations.* 51
4.10 *average trends of PSNR (left) and SSIM (right) for* `WDSR` *x4 and bicubic algorithm as functions of the input angle of rotations.* 52
4.11 *comparison between* `EDSR` *x2 and bicubic, if compared with the images above, the level of recostruction is really similar to slices without rotations in figure 4.2* 52

4.12 *comparison between `WDSR` x4 and bicubic, if compared with the images above, the level of recostruction is better than slices without rotations in figure 4.6. Indeed the number of artifacts seems to be far lower than before for `WDSR`.* 53
4.13 *Absolute differences for `EDSR` reconstruction (left) and bicubic (right), for a T1-weighted image. In both cases, the major differences seems to lies in the scalps of the subjecs. Though, it can be seen that the background is not zero, which means it has an impact on the scores.* 54
4.14 *Histograms of the distribution of differences pixel-by-pixel for the reconstruction performed by EDSR (Blue) and by the bicubic algorithm (Orange), for a T1-weighted NMR. The histogram has been cut between 0.05 and 0.1 on the y axis to better represent the lower parts.* 55
4.15 *Absolute differences of Super resolved image obtained with `EDSR` (left) and bicubic (right) for T2-weighted NMR. This time the major differences seem to be mostly located on the inner parts of the sample, in particular around edges.* 56
4.16 *Histograms of the distribution of differences pixel-by-pixel for a reconstruction performed by EDSR (Blue) and by the bicubic algorithm (Orange), for a T2-weighted NMR. The histogram has been cut between 0.08 and 0.1 on the y axis to better represent the lower parts.* 56
4.17 *Absolute differences of Super resolved images obtained with `WDSR` (left) and bicubic (right) for a T1-weighted NMR. Also in this case the major differences seems to be focused on the scalp of the subjects.* 57
4.18 *Histograms of the distribution of differences pixel-by-pixel for a reconstruction performed by EDSR (Blue) and by the bicubic algorithm (Orange), for a T2-weighted NMR. The histogram has been cut between 0.08 and 0.1 on the y axis to better represent the lower parts.* 58
4.19 *Absolute differences of Super resolved images obtained with `WDSR` (left) and bicubic (right) for a T2-weighted NMR.* 58
4.20 *Histograms of the distribution of differences pixel-by-pixel for a reconstruction performed by EDSR (Blue) and by the bicubic algorithm (Orange), for a T2-weighted NMR. The histogram has been cut between 0.08 and 0.1 on the y axis to better represent the lower parts.* 59
4.21 *Example of application of a mask on one of the slices of the original 3D map. (left) Original image, (center) Mask obtained from FSL BET and (right) Mask applyed to the original image.* 60
4.22 *Examples of Brain Extraction for original image (left), EDSR image (center) and Bicubic results (right). The binary masks are different for the three images.* 60

4.23 *Absolute differences for Super Resolution (EDSR, left) and bicubic (right). The images are obtained by computing a pixel-wise absolute difference with the original slice. We can see the main dissimilarities on the border of the two images, meaning that the BET didn't catch all the relevant informations on the reconstructed images.* 61
4.24 *Absolute differences for Super Resolution* `WDSR`*, left) and bicubic (right). In this case, the principal dissimilarities are not focused only on the outer parts of the brain and that is understandable, given the high level of down-sampling.* 61
4.25 *Histogram of differences for an image obtained with x2 up-sampling and after Brain Extraction. The background componenent is completely shifted towards zero, while the Brain componentent spreaded and shifted toward higher intensity for contrast manipulation. The distributions for the two methods are nearly identical. The histogram has been cut on the y axis between 0.07 and 0.2 to better represent the Brain Component* 62
4.26 *Trends of IoU scores for the different upsamplig methods. (left) comparison between the mask obtained from* `EDSR` *and bicubic recostructions and (right) same graph for* `WDSR` *and bicubic. Notice that some slices are cut from comparisons since they are black images and both intersection and union are 0.* 63
4.27 *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution* `EDSR` *model compared with the bicubic algorithm scores (Yellow) as functions of the slices. Above are the results pre brain extraction while below the ones post brain extraction. The average is performed for one patient and for every rotation, for T1-weighted NMRs.* 64
4.28 *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution* `WDSR` *model compared with the bicubic algorithm scores (Yellow) as functions of the slices. Above are the results pre brain extraction while below the ones post brain extraction. The average is performed for one patient and for every rotation, for T1-weighted NMRs.* 65

Bibliography

[1] Midas collection namic. http://insight-journal.org/midas/collection/view/190, 2010.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] E. Agustsson and R. Timofte. Ntire 2017 challenge on single image super-resolution: Dataset and study. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[4] S. Anwar, S. Khan, and N. Barnes. A Deep Journey into Super-resolution: A survey. *arXiv e-prints*, page arXiv:1904.07523, Apr. 2019.

[5] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv e-prints*, page arXiv:2004.10934, Apr. 2020.

[6] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[7] N. Curti. Implementation and optimization of algorithms in biomedical big data analytics. https://nico-curti2.gitbook.io/phd-thesis/, 2019.

[8] N. Curti and M. Ceccarelli. Numpynet: Neural network in pure numpy. https://github.com/Nico-Curti/NumPyNet, 2019.

[9] N. Curti, M. Ceccarelli, A. Baroncini, S. Sinigardi, and A. Fabbri. Byron: Build your own neural network library. https://github.com/Nico-Curti/Byron, 2019.

[10] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Pro-*

ceedings of Machine Learning Research, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[11] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *arXiv e-prints*, page arXiv:1512.03385, Dec. 2015.

[13] D. Hebb. *The Organization of Behavior*. 1949.

[14] A. Hore and D. Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th International Conference on Pattern Recognition*, pages 2366–2369, Aug 2010.

[15] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, page arXiv:1502.03167, Feb. 2015.

[16] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[17] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, Dec. 2014.

[18] B. Lim, S. Son, H. Kim, S. Nah, and K. M. Lee. Enhanced deep residual networks for single image super-resolution. *arXiv e-prints*, page arXiv:1707.02921, Jul 2017.

[19] M. A. Nielsen. *Neural Network and Deep Learning*. Determination Press, 2015.

[20] B. Okken. *Python Testing with Pytest: Simple, Rapid, Effective, and Scalable*. Pragmatic Bookshelf, 1st edition, 2017.

[21] T. Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–.

[22] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[23] J. Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[24] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[25] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network. *arXiv e-prints*, page arXiv:1609.05158, Sept. 2016.

[26] S. M. Smith. Fast robust automated brain extraction. *Human brain mapping*, vol. 3, 2002. doi:10.1002/hbm.10062.

[27] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for Simplicity: The All Convolutional Net. *arXiv e-prints*, page arXiv:1412.6806, Dec. 2014.

[28] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and t. scikit-image contributors. scikit-image: Image processing in Python. *arXiv e-prints*, page arXiv:1407.6245, July 2014.

[29] J. Yu, Y. Fan, J. Yang, N. Xu, Z. Wang, X. Wang, and T. Huang. Wide activation for efficient and accurate image super-resolution. *arXiv e-prints*, page arXiv:1808.08718, Aug 2018.