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

Abstract

Contents

1 Introduction **3**
- 1.1 Neural Network and Deep Learning . . . 3
- 1.2 Super Resolution . . . 7
  - 1.2.1 Bicubic Interpolation . . . 8
  - 1.2.2 Image Quality . . . 9
- 1.3 Magnetic Resonance . . . 10

2 Algorithms **11**
- 2.1 Frameworks . . . 11
- 2.2 Convolutional Neural Network . . . 13
- 2.3 Layers . . . 14
- 2.4 Timing . . . 28

3 Datasets and Methodology **29**
- 3.1 Models . . . 29
- 3.2 Train Dataset : DIV2K . . . 29
- 3.3 NMR Dataset . . . 31

4 Results **34**
- 4.1 EDSR . . . 34
- 4.2 WDSR . . . 35
- 4.3 . . . 36
- 4.4 Conclusions . . . 36

Chapter 1

Introduction

Brief introduction of 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 [12]. 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 didn't 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 kind of approach 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 correct 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 punishent, 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 [22]. 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 no weight is changed. 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 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:

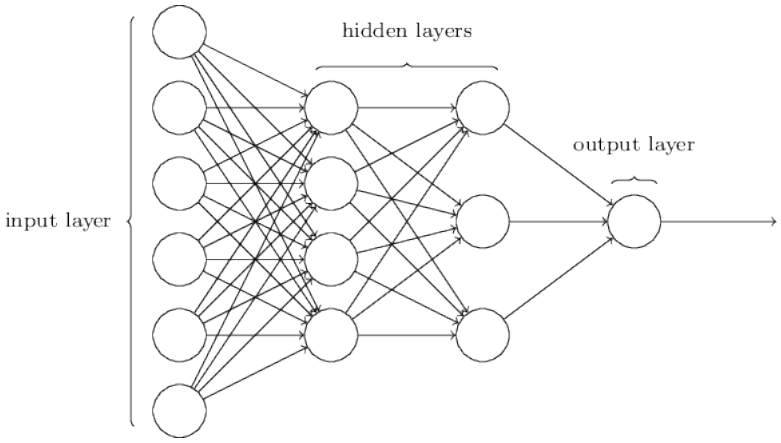

Figure 1.1: *A common representation of a neural network: a single node works as the perceptron described above.*

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.

As before, σ is the activation function which can be different for every node, but it usually differs only from layer to layer. The choice of the best function for a given problem is still an open issue.

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 true label 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 linearly 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 loop 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 direction that minimize 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.

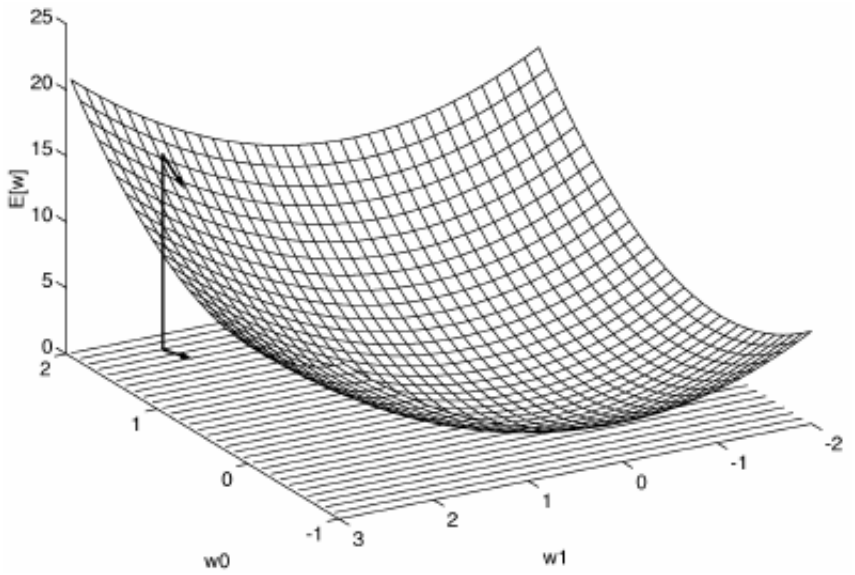

Figure 1.2: *Visual example of gradient descent for a model with 2 weights. The idea is to modify the weights to follow the direction of steepest descent for the landscape of the error function*

The speed at which the algorithm converge 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*. Given a differentiable cost function $C(W)$, 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 for a model with L total layer, and:

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

as the vector of errors of layer l. Then we can write the 4 equations of back propagation for the fully connected neural network [17]:

$$\boldsymbol{\delta}_L = \nabla_a C \odot \sigma'(\boldsymbol{z}_L) \tag{1.10}$$

$$\boldsymbol{\delta}_l = (W_{l+1}^T \boldsymbol{\delta}_{l+1}) \odot \sigma'(\boldsymbol{z}_l) \tag{1.11}$$

$$\frac{\partial C}{\partial b_l^j} = \delta_l^j \tag{1.12}$$

$$\frac{\partial C}{\partial w_l^{jk}} = a_{l-1}^k \delta_l^j \tag{1.13}$$

where $\odot$ is the element-wise product. Those equations can be generalized for others kind of layer, as I will show in the next chapters.

The full training algorithm is :

- define the model with random parameters
- compute the output for one of the inputs
- 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.
- updates the parameters following the update rule,
- iterate from step 2 until the loss is sufficiently small

1.2 Super Resolution

The term Super-Resolution (SR) referes to a class of techniques which aim is to 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 find the correct label for a set of sample 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. 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's 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:

$$SSIM(I,K) = \frac{1}{N}\sum_{i=1}^{N} SSIM(x,y) \tag{1.17}$$

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

$$SSIM(x, y) = \frac{1}{N} \sum_{i=1}^{N} \frac{(2\mu_x \mu_y + c_1)(2\sigma_{xy} + c_2)}{({\mu_x}^2 + {\mu_y}^2 + c_1)({\sigma_x}^2 + {\sigma_y}^2 + c_2)}, \tag{1.18}$$

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* [25], 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 Magnetic Resonance

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 [20] and Caffe [15]. 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 [19] (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 [18]. 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 ≥ 2.6.

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 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a specialized kind of neural network for processing data that has known grid-like topology [10], 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:

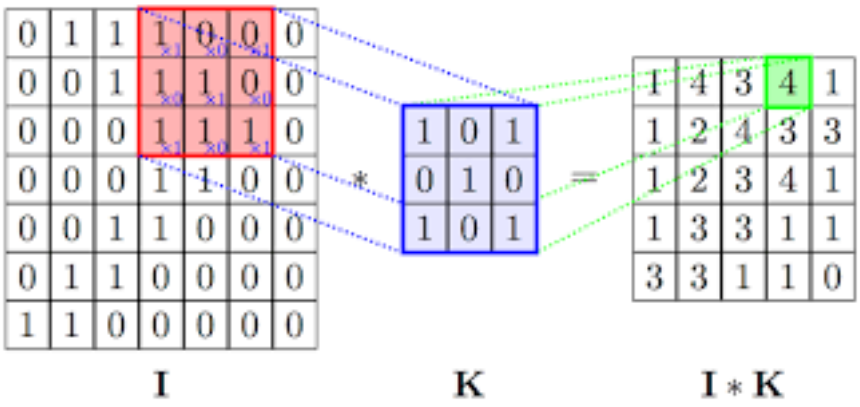

Figure 2.1: *Visual example of convolution of an image I 7x7 with a 3x3 kernel K.*

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.

2.3 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 kinds of layer 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

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 [10].

The difference from a traditional convolutional approach is that instead of using predetermined 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 (widht and height) can be indipendent from one another, but most implementation require square kernels.
- **strides** : defines the movement of the filters. With a low stride (e.g. unitary) the windows tends to overlap. With highr 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 dimensions 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. By performing a clever transformation of the input into a flat matrix, in which every columns yields the values that have to be multiplied by the filters for each output, 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:

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

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

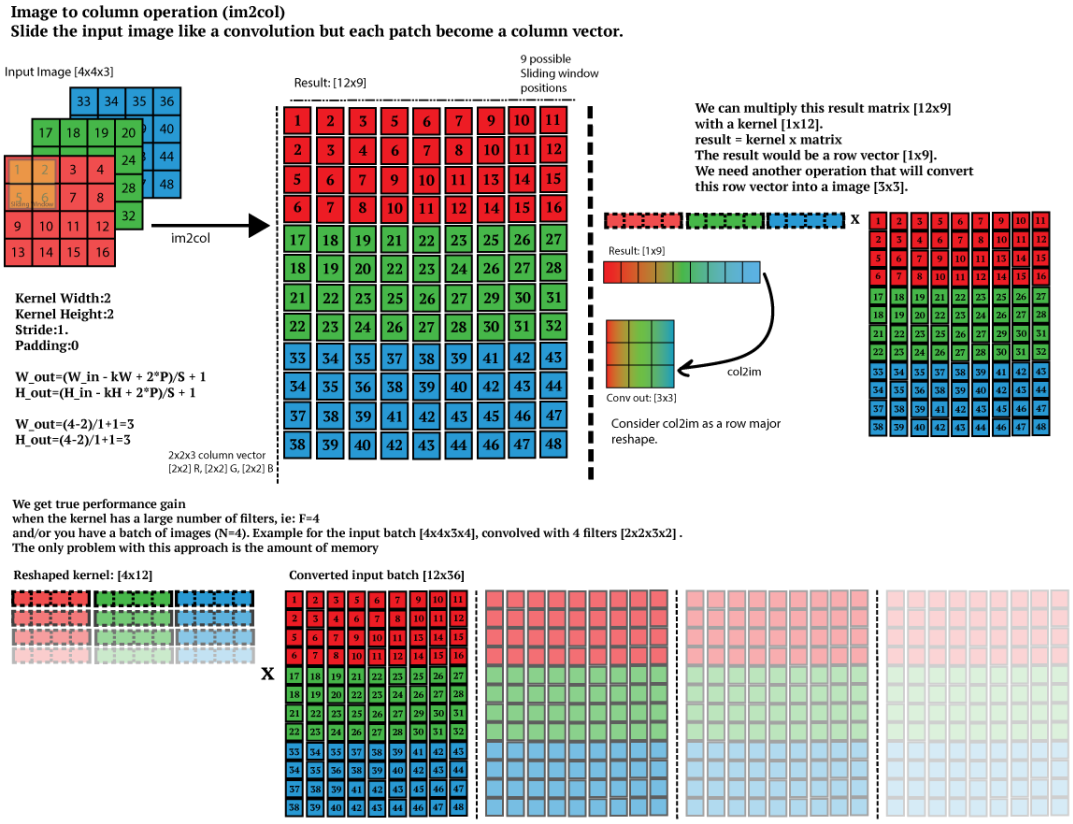

Figure 2.2: *Scheme of the im2col algorithm using a* $2 \times 2 \times 3$ *filter with stride 1 on a* $4 \times 4 \times 3$ *image. The matrix multiplication is between a* $n \times 12$ *and a* 12×9 *matrixes.*

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

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.

FINISH

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 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

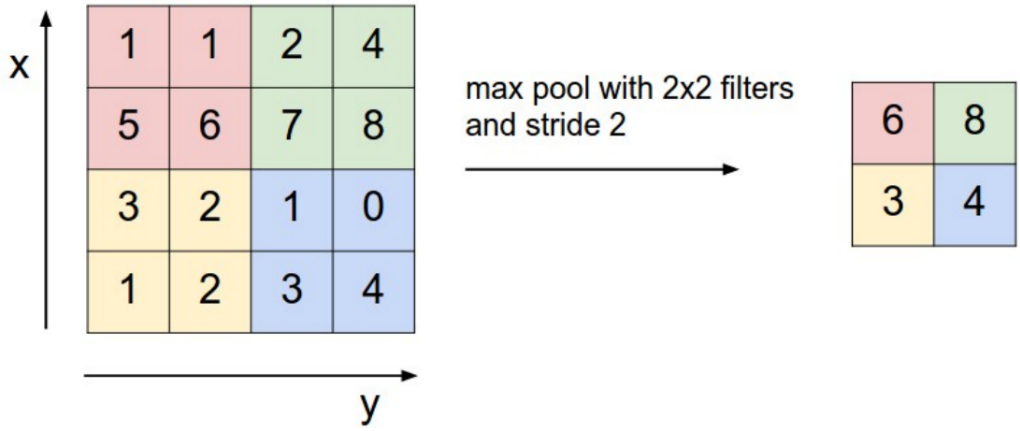

Figure 2.3: *Scheme of maxpool operations with a kernel of size* 2×2 *and stride* 2 *over an image of size* 4×4*. Picture from CS231n*

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*

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 [10]. 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:

- **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 (widht and height) can be indipendent from one another, but most implementations require square kernels.
- **strides** : defines the movement of the filters. With a low stride (e.g. unitary) the windows tends to overlap. With higher stride values we have less overlap (or none) and the dimension of the output decrease. Usually pooling operations have

a stride of 2.

- **padding** : is the dimensions 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. Most often than not, no padding is applied during pooling operations.

The output dimensions for Pooling layers are the same as for CLs, however, since the operations does not change during the training phase, they have no weights.

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 AvgPool function

```
import numpy as np

class Avgpool_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

    # Shape of the final view
    view_shape = (batch, 1 + (w - kx)//st1, 1 + (h - ky)//st2) + input.shape[3:] + (kx, ky)

    # strides of the final view
    strides = (batch_stride, st1 * s0, st2 * s1) + input.strides[3:] + (s0, s1)

    subs = np.lib.stride_tricks.as_strided(input, view_shape, strides=strides)
    # returns a view with shape = (batch, out_w, out_h, out_c, kx, ky)
    return subs

  def forward(self, input):

    self.batch, self.w, self.h, self.c = input.shape
```

```
32      kx, ky = self.size
33      sx, sy = self.stride
34
35      input = input[:, : (self.w - kx) // sx*sx + kx, : (self.h - ky) //
        sy*sy + ky, ...]
36      # 'view' is the strided input image, shape = (batch, out_w, out_h,
        out_c, kx, ky)
37      view = self._asStride(input, self.size, self.stride)
38
39      # Mean of every sub matrix, computed without considering the pad(np
        .nan)
40      self.output = np.nanmean(view, axis=(4, 5))
```

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 differece 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 backprogate 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 pixel 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.

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 [11]. Famously, 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

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*

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 lnear combination. Instead of learning a function $F(x)$ we try to learn $H(x) = F(x) + x$, as shown in figure 2.6:

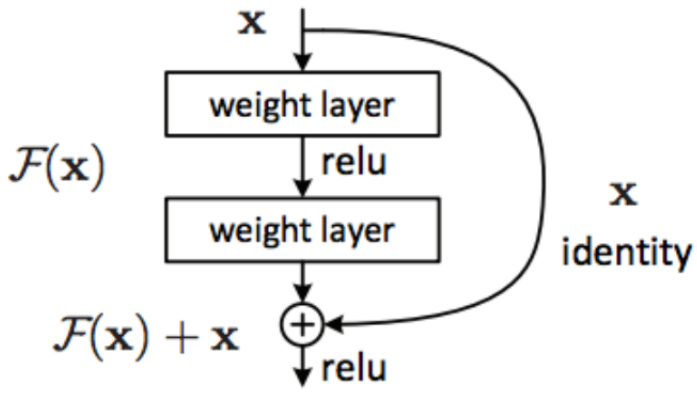

Figure 2.6: *Scheme of the shortcut layer as designed by the authors [11]. The output of the second layer become a linear combination of the input x and its own output.*

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.

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.

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`. 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 this 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 working, the transposed convolution is not efficient in terms of computational and memory cost, therefore not suited for modern convolutional neural network.

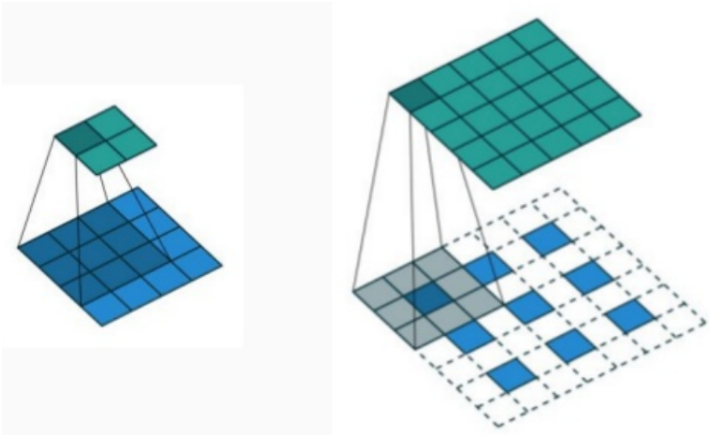

Figure 2.7: *example of deconvolution: (left) a normal convolution with size 3 and stride 1, (right) after applying a "zeros upsampling" the convolution of size 3 and stride 1 become a deconvolution*

An alternative is the recently introduced *sub-pixel convolution* [23] (also called Pixel Shuffle). The main advantages over the deconvolution operation is the absece of weights to train: indeed the operation perfermod by the Pixel Shuffle (PS) Layer is deterministic and it is very efficient if compared to the deconvolution, since it only performs 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 $C = 1$:

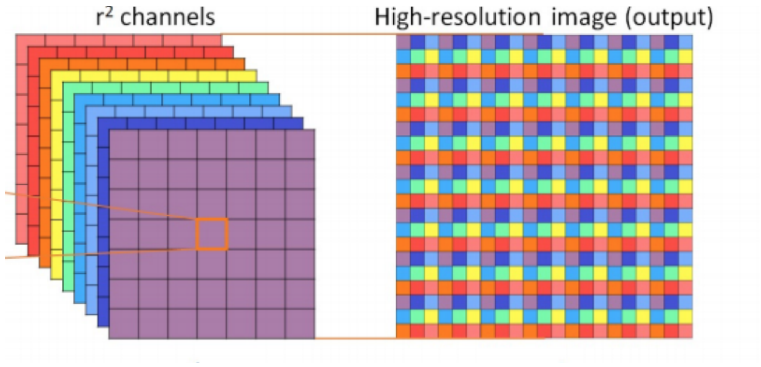

Figure 2.8: *Example of pixel shuffling proposed by the authors [23]. In this example, r^2 features maps are re-arranged into a single-channeled high resolution output.*

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 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 [19]. This implementation has been tested against `tensorflow`'s `depth_to_space` and `space_to_depth`. Despite beign 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 soulution 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 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. [14]

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 tensor 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 :

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

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

```
1
2 def forward(self, inpt):
3   '''
4   Forward function of the BatchNormalization layer. It computes the
     output of
```

```
  the layer, the formula is :
                  output = scale * input_norm + bias
  Where input_norm is:
                  input_norm = (input - mean) / sqrt(var + epsil)
  where mean and var are the mean and the variance of the input batch
   of
  images computed over the first axis (batch)
  Parameters:
    inpt  : numpy array, batch of input images in the format (batch, w,
    h, c)
  '''

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

  # Copy input, compute mean and inverse variance with respect the
   batch axis
  self.x    = inpt.copy()
  self.mean = self.x.mean(axis=0)                              # shape =
    (w, h, c)
  self.var  = 1. / np.sqrt((self.x.var(axis=0)) + self.epsil) # shape =
    (w, h, c)
  # epsil is used to avoid divisions by zero

  # Compute the normalized input
  self.x_norm = (self.x - self.mean) * self.var # shape (batch, w, h, c
   )
  self.output = self.x_norm.copy() # made a copy to store x_norm, used
   in Backward

  # Init scales and bias if they are not initialized (ones and zeros)
  if self.scales is None:
    self.scales = np.ones(shape=self.out_shape[1:])

  if self.bias is None:
    self.bias = np.zeros(shape=self.out_shape[1:])

  # Output = scale * x_norm + bias
  self.output = self.output * self.scales + self.bias

  # output_shape = (batch, w, h, c)
  self.delta = np.zeros(shape=self.out_shape, dtype=float)

  return self

def backward(self, delta=None):
  '''
  BackPropagation function of the BatchNormalization layer. Every
   formula is a derivative
  computed by chain rules: dbeta = derivative of output w.r.t. bias,
   dgamma = derivative of
  output w.r.t. scales etc...
```

```
47    Parameters:
48      delta : the global error to be backpropagated, its shape should be
       the same
49         as the input of the forward function (batch, w, h ,c)
50    '''
51
52    check_is_fitted(self, 'delta')
53    self._check_dims(shape=self.input_shape, arr=delta, func='Forward')
54
55    invN = 1. / np.prod(self.mean.shape)
56
57    # Those are the explicit computation of every derivative involved in
       BackPropagation
58    # of the batchNorm layer, where dbeta = dout / dbeta, dgamma = dout /
       dgamma etc...
59
60    self.bias_update = self.delta.sum(axis=0)                    # dbeta
61    self.scales_update = (self.delta * self.x_norm).sum(axis=0) # dgamma
62
63    self.delta *= self.scales                                     # self.
       delta = dx_norm from now on
64
65    self.mean_delta = (self.delta * (-self.var)).mean(axis=0)    # dmu
66
67    self.var_delta = ((self.delta * (self.x - self.mean)).sum(axis=0) *
68                      (-.5 * self.var * self.var * self.var))     # dvar
69
70    # Here, delta is the derivative of the output w.r.t. input
71    self.delta = (self.delta * self.var +
72                  self.var_delta * 2 * (self.x - self.mean) * invN +
73                  self.mean_delta * invN)
74
75    if delta is not None:
76      delta[:] += self.delta
77
78    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.

- table
- example images

- forward backward

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	$f(x) = \max(0, x)$	$f'(x) = \begin{cases} 1 & \text{if } f(x) > 0 \\ 0 & \text{if } f(x) \leq 0 \end{cases}$
Elu	$f(x) = \max(\exp(x) - 1, x)$	$f'(x) = \begin{cases} 1 & \text{if } f(x) \geq 0 \\ f(x) + 1 & \text{if } f(x) < 0 \end{cases}$
Relie	$f(x) = \max(x * 1e-2, x)$	$f'(x) = \begin{cases} 1 & \text{if } f(x) > 0 \\ 1e-2 & \text{if } f(x) \leq 0 \end{cases}$
Ramp	$f(x) = \begin{cases} x^2 + 0.1 * x^2 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + 1 & \text{if } f(x) > 0 \\ f(x) & \text{if } f(x) \leq 0 \end{cases}$
Tanh	$f(x) = \tanh(x)$	$f'(x) = 1 - f(x)^2$
Plse	$f(x) = \begin{cases} (x+4) * 1e-2 & \text{if } x < -4 \\ (x-4) * 1e-2 + 1 & \text{if } x > 4 \\ x * 0.125 + 5 & \text{if } -4 \leq x \leq 4 \end{cases}$	$f'(x) = \begin{cases} 1e-2 & \text{if } f(x) < 0 \text{ or } f(x) > 1 \\ 0.125 & \text{if } 0 \leq f(x) \leq 1 \end{cases}$
Leaky	$f(x) = \begin{cases} x * C & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} 1 & \text{if } f(x) > 0 \\ C & \text{if } f(x) \leq 0 \end{cases}$
HardTan	$f(x) = \begin{cases} -1 & \text{if } x < -1 \\ +1 & \text{if } x > 1 \\ x & \text{if } -1 \leq x \leq 1 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } f(x) < -1 \text{ or } f(x) > 1 \\ 1 & \text{if } -1 \leq f(x) \leq 1 \end{cases}$
LhTan	$f(x) = \begin{cases} x * 1e-3 & \text{if } x < 0 \\ (x-1) * 1e-3 + 1 & \text{if } x > 1 \\ x & \text{if } 0 \leq x \leq 1 \end{cases}$	$f'(x) = \begin{cases} 1e-3 & \text{if } f(x) < 0 \text{ or } f(x) > 1 \\ 1 & \text{if } 0 \leq f(x) \leq 1 \end{cases}$
Selu	$f(x) = \begin{cases} 1.0507 * 1.6732 * (e^x - 1) & \text{if } x < 0 \\ x * 1.0507 & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) * 1e-3 & \text{if } f(x)0 \\ (f(x)-1) * 1e-3 + 1 & \text{if } f(x) > 1 \end{cases}$
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.*

Loss Function

2.4 Timing

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.

EDSR
WDSR

EDSR

WDSR

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 factor.
- **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 3.1, 3.2 and 3.3.

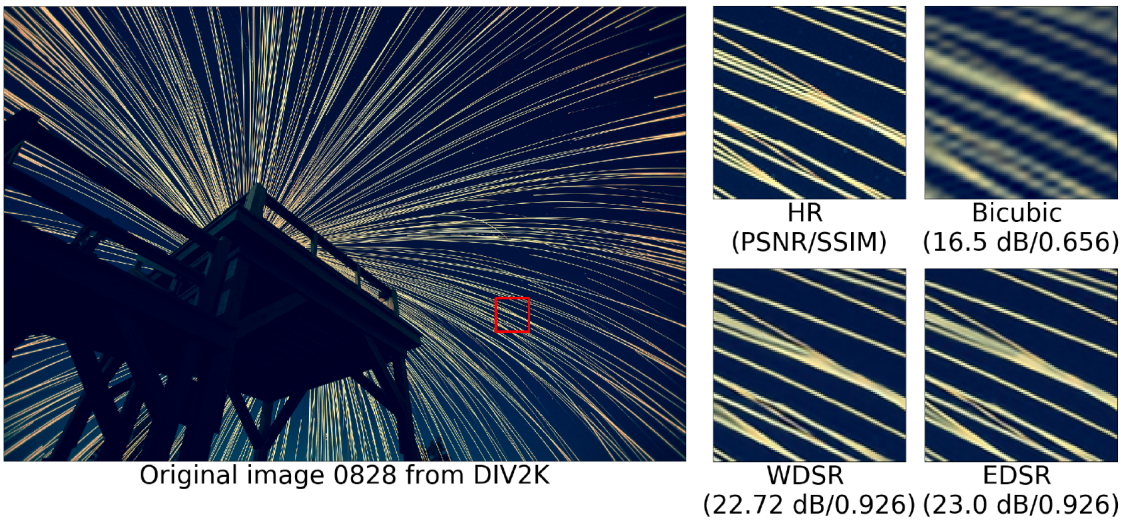

Figure 3.1: *Super Resolution visual example extracted from the DIV2K validation set. The quality score in terms of PSNR and SSIM are compared between a standard bi-cubic up-sampling and the EDSR and WDSR models.*

Figure 3.2: *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 that 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.

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.*

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 3.4 are shown three different slices for the same patient at HR:

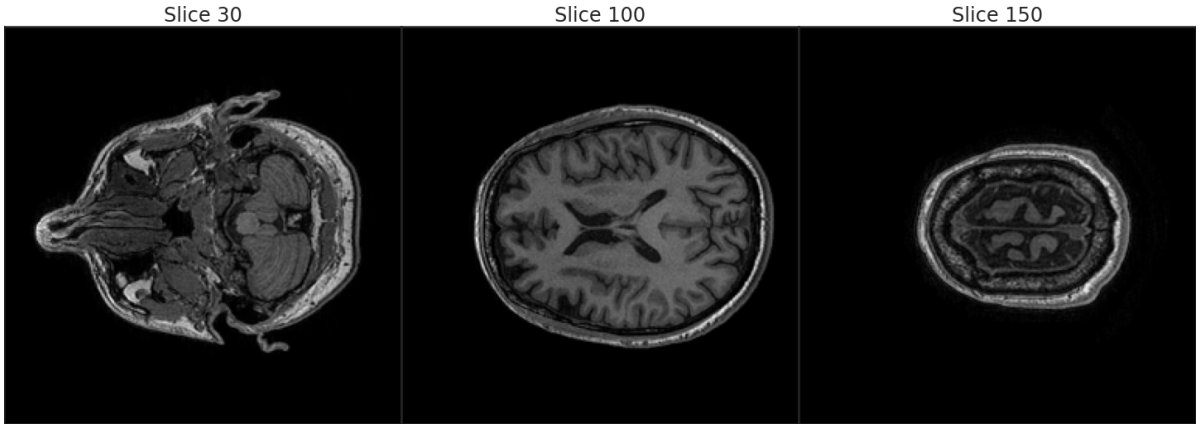

Figure 3.4: *HR* 256×256 *original image at three different stages of depth: (left) slice 30 where still a lot of information about the brain is hidden, (center) slice 100 which is a central slice where most of the information is stored, (right) slice 150 which starts the less informative area of the brain.*

Then, the HR images (which will also be called *originals*) have been downsampled with the bicubic algorithm by two different scale factors, namely 2x and 4x, obtaining two distinct sets of LR images for every subject and for every weight, respectively of sizes $128 \times 128 \times 176$ and $64 \times 64 \times 176$.

After all that, the LR images are convoluted with a gaussian kernel of size 3, stride 1 and standard deviation 1 with the function `cv2.GaussianBlur` of the library OpenCV. This 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 3.5 is shown an example of the images obtained by this procedure for a downscale factor of 2:

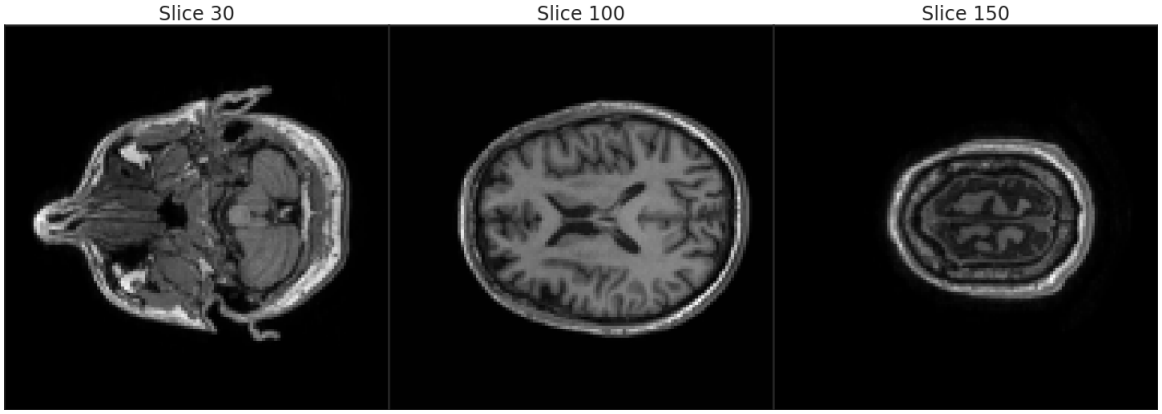

Figure 3.5: 128×128 *LR version of the same slices shown for the HR case.*

In figure 3.6 is shown an example of the images obtained by this procedure for a downscale factor of 4:

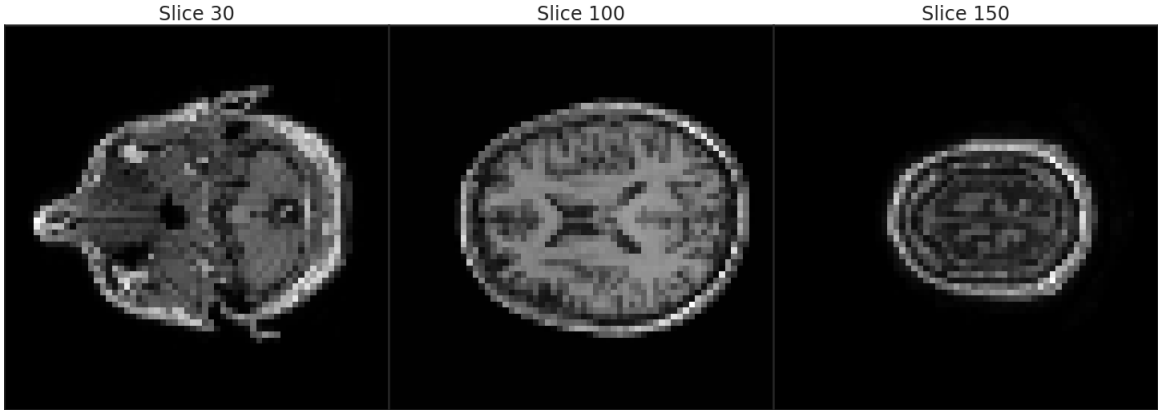

Figure 3.6: 64×64 *LR version of the same slices shown for the HR case.*

The LR images are used as input for `EDSR`, `WDSR` and the bicubic algorithm which re-upsample them respectively by factors 2x, 4x and both, trying to recostruct an image

as close as possible to the original one. As further analysis we decided to investigate how different input conditions influence the results for the three methods. In particular how an angle of rotation for the input images can impact the final re-upsample: indeed this changes the orientation of lines, shapes and textures in the images and can change 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 3.7 is shown an example of the kind of inputs fed to the models:

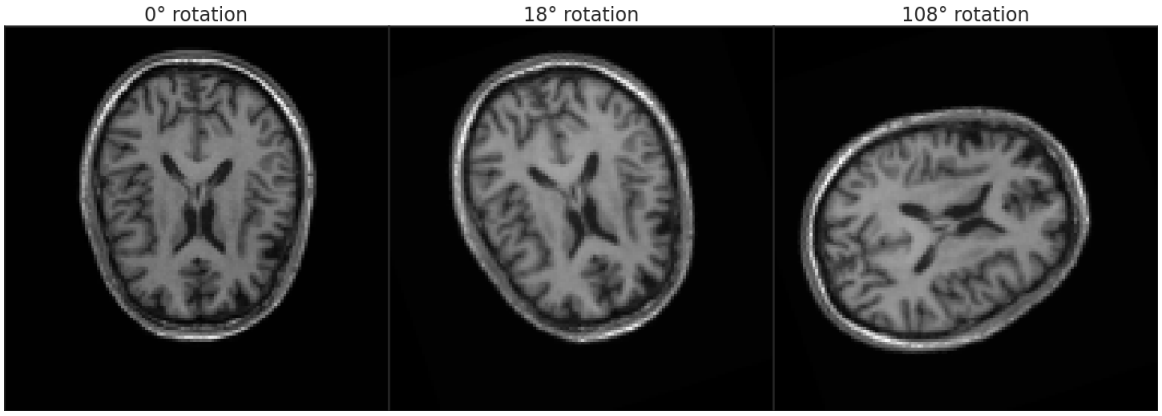

Figure 3.7: *Three of the 20 rotation angles used as input for Super Resolution models and Bicubic. (left) reference angle of 0 degree, (centre) angle step of 18 degree, (right) large rotation of 108° respect to the reference.*

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 MIDAS [1].

Chapter 4

Results

In the following chapter I'm 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 EDSR

The `EDSR` model is used to upsample the images by a x2 factor so that the single slice is Super Resolved from a 128×128 to a 256×256 spatial resolution. I decided to separate the analysis for the three output channel of the super-resolution since it can highlight particular behaviours. 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). It is clear that there is a difference in behaviour between the three output for the super resolution: namely

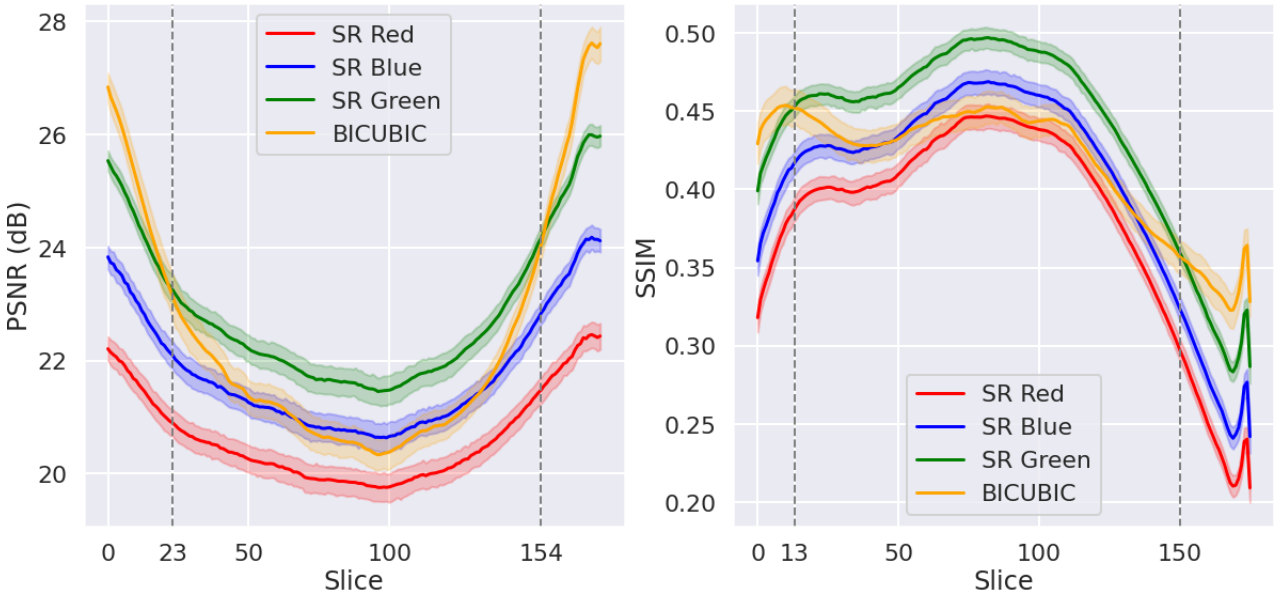

Figure 4.1: *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution model compared with the bicubic algorithm scores (Yellow) as functions of the slices. The average is performed for every patients and for every rotation. The dotted lines highlights the slices where the bicubic and Super Resolution performances intersect. The bicubic seems to perform better that SR only on the less informative section of the subjects*

For reference,

- boxplot with general results
- how performances varies with slice
- how the performances varies with angle
- comparisons of rotated angle reconstruction.
- where are located the differences
- how is the difference distributed (histograms)
- difference between t1 and t2

4.2 WDSR

The `WDSR` model is used to upsample the images by x4 factor

4.3

4.4 Conclusions

List of Figures

1.1 *A common representation of a neural network: a single node works as the perceptron described above.* 5
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 steepest descent for the landscape of the error function* 6

2.1 *Visual example of convolution of an image I 7x7 with a 3x3 kernel K.* . . 13
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.* 16
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* 18
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* 18
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* 21
2.6 *Scheme of the shortcut layer as designed by the authors [11]. The output of the second layer become a linear combination of the input x and its own output.* 21
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* 23
2.8 *Example of pixel shuffling proposed by the authors [23]. In this example,* r^2 *features maps are re-arranged into a single-channeled high resolution output.* 23

3.1 *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.* 30

3.2 *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.* 30
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.* 31
3.4 *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.* 31
3.5 128×128 *LR version of the same slices shown for the HR case.* 32
3.6 64×64 *LR version of the same slices shown for the HR case.* 32
3.7 *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.* 33

4.1 *Average trends of PSNR (left) and SSIM (right) for the three channels (Red, Blue, Green lines) of the Super Resolution model compared with the bicubic algorithm scores (Yellow) as functions of the slices. The average is performed for every patients and for every rotation. The dotted lines highlights the slices where the bicubic and Super Resolution performances intersect. The bicubic seems to perform better that SR only on the less informative section of the subjects* 35

Bibliography

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.