Alma Mater Studiorum · Università di Bologna

Physics and Astronomy Department
PhD Thesis in Applied Physics

Implementation and optimization of algorithms in Biological Big Data Analytics

Supervisor:

Prof. Daniel Remondini

Correlator:

Prof. Gastone Castellani
Prof. Armando Bazzani

Presented by:

Nico Curti

Session 2019/2020

Chapter 1

Deep Learning - Neural Network algorithms

Description of the modern deep neural networks. Computational problems and potential applications

1.1 Neural Network models

Neural Networks are mathematical models commonly used in data analysis. They are becoming a standard tool in Machine Learning and Deep Learning research and many complex problems can be easily solved by these models. From a theoretical point-of-view we can define a Neural Network as a series of non-linear multi-parametric functions. The model parameters are tuned during a so called *training section* in which we feed our model with a set of data with human supervision, i.e we have prior knowledge about the right and desired output of the model. After the training section we can verify the efficiency of our training using a new set of data, called *test set*, which is never seen by the model. If we have prior knowledge about the output of our test set we can compute the accuracy (or more generally the score) of our model; in the other case we will simply have an extrapolation of our data.

A wide range of documentations and implementations have been written on this topic and it is more and more hard to move around the different sources. Leader on this topic are became the multiple open-source Python libraries available on-line as *Tensorflow* [1], *Pytorch* [21] and *Caffè* [17]. 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 more deeper implementation in C++ or other low-level programming languages. About them it should be mentioned the *darknet project* of Redmon J. et al. which created a sort of standard in object detection applications using a pure Ansi-C library[1].

In this section we firstly retrace the mathematical background of these models. To each theoretical explanation we discuss the numerical problems associated and we provide an efficient custom implementation of each algorithm. The numerical aspects will be traced following two developed custom libraries: NumPyNet library [6] and Byron library [7].

[1] *Darknet* is framework for neural network model developing. It is written in pure Ansi-C by a Washington University research group. 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 the CUDA support, i.e only for NVidia GPUs. It is particularly famous for object detection applications since it firstly theorize a novel approach to multi-scale object detections called Yolo (You Only Look Once). The library developed in this work are all inspired on it. The large part of the original work developed is related to a deep optimization of this library either in terms of functionality and issues either in terms of computational performances.

NumPyNet was born as 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 [20] (a based package for the scientific research).

Despite all common libraries are correlated by a wide documentation is often difficult for novel users to move around the many hyper-links and papers cite in it. 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 in count is related to performances. Libraries like *Tensorflow* as certainly efficient by a computational point-of-view and the numerous wrappers (like *Keras* library) guarantees an extremely simple user interface. On the other hand the deeper functionality of the code and the implementation strategies used are unavoidably hidden behind tons of code lines. In this way the user can performs complex computational tasks using the library as black-box package. NumPyNet wants to overcome this problem using simple Python codes with extremely readability also for novel users to better understand the symmetry between mathematical formulas and code.

The simplicity of this library we will allow to give a first numerical analysis of the model functions and, moreover, to show the results of each function to a simple image to better understand the effects of their applications on real data[2]. Each NumPyNet function was tested against the *Tensorflow* implementation of the same methods with an automatic testing routine through *PyTest* [19]. 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 *Appevyor CI* for Windows users. The library supports Python version ≥ 2.6[3].

As term of comparison we will discuss the more sophisticated implementations into the Byron library. Byron (Build YouR Own Neural network) library is written in pure C++ with the support of the modern standard 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 code implementations which exploit the fully functionality of a multiple CPUs architecture. This gap discourage multiple research groups on the use of such computational intensive models in their applications. Byron works in a fully parallel section in which is single computational function is performed using the full set of available cores. To further reduce the time of thread spawn 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[4].

The Byron library is release under MIT license and public available on the Github page of the project. The project also includes a list of common examples like object detection, super resolution, segmentation, ecc. (see the next sections for further details about this models). The library is also completely wrapped using *Cython* to enlarge the range of users also to the Python ones. The complete guide to its installation is provided; it can be done using *CMake*, *Make* or *Docker* and the Python version is available with a simple *setup*. The testing of each function is performed using *Pytest* automatic framework against the

[2] Aware of the author no other example implementations have been done. This makes the NumPyNet library a useful tool for neural network study and a virtual laboratory for new neural network functions.

[3] The library provides also an Image object to load and process images. The object is based on OpenCV API [5]. OpenCV does not yet support Python version 2.7 and 3.3 so the whole NumPyNet package does not work on these two version of Python. You can just exclude the Image scripts from the package or use a novel wrap based on different library (e.g Pillow).

[4] For real-time applications also the time required for the thread spawn must be taken into account.

NumPyNet implementation (faster and lighter to import than *Tensorflow*).

We will use Byron library as term of comparison with the other common library used in Neural Network models and for each function we test its computational efficiency and scalability on multiple cores. Two machines will be used in the computational testing: a common laptop (8 GB RAM memory and 1 CPU i7-6500U, with 2 cores) and a classical bioinformatics server (128 GB RAM memory and 2 CPU E5-2620, with 8 cores each).

Starting from the next section we introduce the fundamental Neural Network model, the so-called *Simple Perceptron*. From the simplest model we will add complexity layers to overcome the relative problems (mathematical and numerical), introducing the main functionality of the modern Neural Network architectures.

1.1.1 Simple Perceptron

The fundamental unit of each Neural Network model is the *simple Perceptron* (or single neuron). The *Perceptron* it the simpler mathematical model of biological neuron and it is based on the Rosenblatt [26] model which identifies a neuron as a computational unit with input, synaptic weights and an activation threshold (or function). Following the biological model of Hodgkin and Huxley [14] (H-H model), we have an action potential, i.e the output of the neuron, given by

$$y = \sigma\left(\sum_{i=1}^{N} w_i x_i + w_0\right)$$

where σ is the activation function, w_i are the synaptic weights and x_i the inputs. The w_0 coefficient identifies the bias of the linear combination and it is left as parameter to be tune by the optimization algorithm (learning phase).

The connection weights w_i are tuned during the training section by the chosen updating rule. The standard updating rule is simply given by

$$w_i(\tau + 1) = w_i(\tau) + \gamma(t - y)x$$

where γ is the gain or step size ($\gamma \in [0, 1]$) and t is the desired output. In other words we have to firstly compute the difference between the current output and the desired one, i.e the error or cost function or loss function[5], and weight this error by the gain factor and the corresponding input. Repeating the error computation and the updating rule we can bring the weights to convergence. From a geometrical point-of-view this process is equivalent to an hyper-plane placement defined by $w_0 + < w, x >$ which splits an $n-$dimensional space into two half-spaces, i.e two desired classes.

The mathematical formulation already highlights the numerous limits of this model. The output function is a simple linear combination of the input with a vector of weights and so only linearly separable problems can be learned[6] by the *Perceptron*[7]. Moreover we can manage only two classes since an hyper-plane divide the space in only two half-spaces.

A key role is assumed by the activation function. The classical activation function used in the discrete Perceptron model is the *unit step function* (or *Heaviside step function*). If we chose a continuous and so differentiable activation function we can treat the problem using a continuous cost function. In this case we can define it as

[5] There are multiple loss functions in the Neural Network world. We will further discuss their use and their effective on a learning model in the next section.

[6] A simple mathematical proof of it can be found here.

[7] A classical example of learning problems is given by the XOR logic function. Since the XOR output is not linearly separable the Perceptron could not converge.

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(t_i - y_i)^2$$

where in this case both t_i and y_i are continuous variables, i.e floating point numbers. Now the updating rule can be given by the gradient of the cost function applied to the original weights as

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$$

where $\Delta\mathbf{w}$ is given by

$$\Delta\mathbf{w_i} = -\gamma\frac{\partial E}{\partial w_i} = -\gamma\sum_{i=1}^{N}(t_i - y_i)(-x_i)$$

which looks identical to previous updating rule but in this case we are managing real numbers and not simple class labels. Moreover in this way we compute the weight updates according to the full set of training sample and not for each sample (this approach leads to the so-called *batch*-update, i.e small subsets of data).

To implement this kind of model into a pure Python application we do not need extra libraries but we can just use the native keyword of the language. A possible implementation of this model was developed and release in a on-line gist. In this simple snippet we examine the functionality of the Simple Perceptron model across different logical functions and we proof its fast convergence on linear separable datasets[8].

An equivalent C++ implementation of the model is also provided and can be found in this other gist.

The model is too naive for computational efficiency discussions. Thus we can just observe how a learning algorithm could be easily implemented using basic programming language keywords either in Python either in C++.

1.1.2 Fully Connected Neural Network

To overcome problems arising from the Simple Perceptron model we can join together multiple Perceptron units into a more complex network of interaction in which the output of a neuron feed-forward the input of the next one. This is the Multi-layers Perceptron (MLP) configuration and if the graph is fully connected, i.e each neuron is connected to all the others, we talk about *fully connected neural networks* (or *dense* neural network, DNN).

Given the Perceptron formulas, the extrapolation to the MLP architecture is straightforward and given by

$$y = \sigma\left(X \cdot W + W_0\right)$$

where we simply pass from the vector formulation to the matrix one. The updating rule consequentially becomes

$$\delta W = \delta W + X^T \cdot \left(\frac{\partial f(y)}{\partial y} \cdot \delta^l\right) \qquad \delta W_0 = \sum_{i=0}^{m}\frac{\partial f(y)}{\partial y_i} \cdot \delta_i^l$$

where also in this case we simply pass to the matrix formalism and we convert the discrete

[8] The proof the non-linear separable convergence introducing an extra stop criteria during the weights tuning given by a maximum number of step.

format to a continuous one, i.e with continuous values we convert the error to a partial derivative. In the above equation δ^l represents the error passed from the next layer in the network structure[9].

From the re-iteration of such structures we can join together multiple fully connected layers and so obtain multiple neuron layers jointly together with different levels of complexity and units (an input layer followed by multiple *hidden* layers).

The fully connected Neural Networks overcome the told above *Perceptron* problems using a combination of linear functions (single *Perceptron* units) and they gain more useful properties:

- If the activation functions of *all* the hidden units in the Neural Network are linear, then the network architecture is equivalent to a network without hidden units.
- If the number of hidden units is smaller than either the number of input units either the number of output ones, then the network can generate transformations from inputs to outputs as much general as possible since the information is lost in the dimensionality reduction performed by the hidden units.
- We can find multiple weight configurations, i.e W matrices, which give us the same mapping function from inputs to outputs.

Given all the theoretical informations about this kind of model we can now pass to practical (numerical) considerations about their implementations.

Matrix Product

Despite the mathematical formulation of the model we have to take in count also an efficient implementation. From a numerical point-of-view we can notice that all the computation required by this kind of Networks (or layer if we consider it into an hybrid Neural Network architecture as we will see in the next sections) can be summarized into the matrix product evaluation. The matrix product is a well-known numerical problems and the complexity of the algorithm can be hardly reduced under $O(N^3)$[10]. A crucial role on this kind of algorithms is played by the cache accesses. The CPU cache is the hardware cache used by the CPU to store small portion of data in order to reduce the average cost (in time or energy consumption) to data access from the main memory. Cache optimization is one of the most difficult parts to perform writing an algorithm, but can lead to highest performance gains.

In the matrix product we have to multiply each row of a matrix A by each column of a second matrix B. We work in the assumption that each matrix is stored into an array of 1D or 2D without nested structures. In this case we can access to a contiguous memory portion of the first matrix since each row will be given by a series of sequential index locations (the row elements will be given by $x[0], x[1], \ldots, x[N]$). This configuration allows the cache optimization in the access to the first matrix since we can store in the small portion of cache memory a series of row elements and use them in a vectorization environment.

From the second matrix we have to extract the elements from each column. This means that the elements will be given by a discontinuous portion of memories (the column

[9] In the Back-Propagation Algorithm the error is passed by each layer to the previous one, starting from the output error computed according to chosen loss function.

[10] The complexity is often given in the assumption of only square matrices ($N \times N$) involved in the computation. For no-square matrix the algorithm complexity is given by the product of the three possible different matrix dimensions involved ($(N \times K) = (N \times M)(M \times K)$ brings to $O(NMK)$ complexity). More sophisticated implementation of the algorithm are able to reduce the algorithm complexity (e.g Strassen algorithm) but neither implementation is able to overcome the $O(N^{2.7})$ complexity up-to-now.

elements will be given by $x[0], x[M], x[2M], \ldots, x[N(M-1)]$). In this case we can not insert a full column into the cache memory and in consequence we will have a *cache-miss* at each iteration[11].

The simple matrix product as given by row-column multiplication is already affected by an intrinsic numerical problem which can drastically affect its performances. The simplest workaround of this problem is to perform a transposition of the second matrix to obtain a row-row matrix product[12]. In this way both matrices can be accessed in a sequential order. The total complexity of the computation increase to $O(N^2)$ (for the matrix transposition, in the better case) $+O(N^3)$ (for matrix product) but the numerical performances increase due to the cache-miss minimization[13].

Following back to our Neural Network implementation we can obtain the output values using the above technique. Moreover we can assumes from the beginning that the weight matrix is transposed and so remove the transposition step from the matrix product. This simple (but carefully studied) optimization allows us to obtain better results in the feed-forward evaluation but it paybacks a revision of the standard mathematical formulation and a carefully implementation of the code.

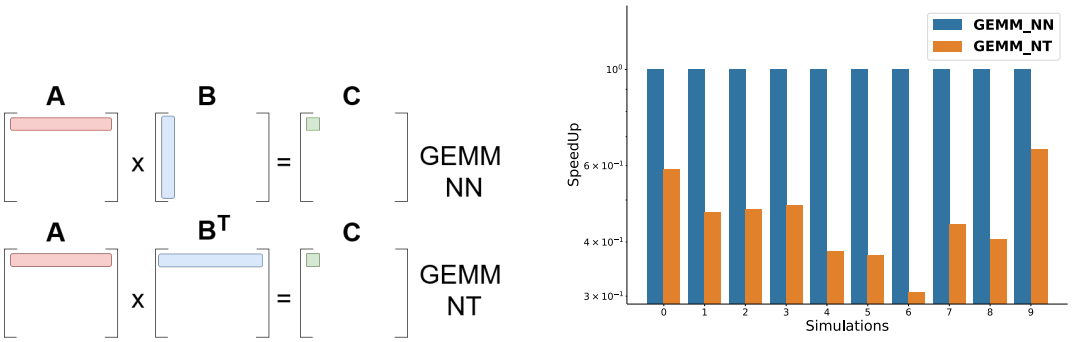

Figure 1.1: GEMM algorithms time performances. GEMM NN: matrix multiplication considering both the matrices in "normal" format, i.e $A \cdot B$. GEMM NT: matrix multiplication considering the first matrix in "normal" format and the second one transposed, i.e $A \cdot B^T$. We perform 100 tests of 1K runs each of both the gemm algorithms using the einsum function of Numpy library. The values are rescaled according to the mean time of the GEMM NN algorithm.

In the proposed numerical implementations of this model we implement both the matrix product cases to compare the performance results. We tested the two implementation inside Python using the einsum function provided by the Numpy package. In particular we evaluate the timing performances over 1000 applications of two the gemm functions (GEMM NN, i.e considering both matrices in "normal" shapes; GEMM NT, i.e considering the first matrix as "normal" and the second transpose) considering matrices of shapes (100×100). We performed 500 run and we save the minimum time obtained over the 10 realizations. In Fig. 1.1 we show the results rescaled by the mean time of the GEMM NN algorithm (reference). As can be seen in Fig. 1.1 the speedup of the GEMM NT matrix is evident and it is always faster than GEMM NN algorithm with a maximum of 3.2x in the speedup.

[11] The *cache-miss* happens when a required data can not be found into the cache and so its search has to be done in the main memory (RAM).

[12] In the discussion we have silently ignored the problems of matrix storage and the cache optimization for the resulting matrix accesses but in the above discussion we want to focus only on the main problems raising from the matrix product.

[13] The cache memory is a very tight portion of memory and it is impossible to completely remove cache-misses.

In the Byron library implementation we provide a parallelized version of this algorithm with also an avx support. In this way we could manually manage the register memory of the two matrices and obtain faster version of the GEMM algorithm (especially for dimensions proportional to powers of 2 which are very common in neural network models).

1.1.3 Activation Functions

Activation functions (or transfer functions) are linear or non linear equations which process the output of a Neural Network neuron and bound it into a limit range of values (commonly $\in [0, 1]$ or $\in [-1, 1]$). The output of a simple neuron[14] can be computed as dot product of the input and neuron weights (see previous section); in this case the output values ranging from $-inf$ to $+inf$ and moreover it is just a simple linear function. Linear functions are very simple to trait but they are limited in their complexity and thus in their learning power. Neural Networks without activation functions are just simple linear regression model (see the fully connected Neural Network properties in the previous section). Neural Networks are considered as *Universal Function Approximators* so the introduction of non-linearity allows them to model a wide range of functions and to learn more complex relations in the pattern data. From a biological point of view the activation functions model the on/off state of a neuron in the output decision process.

Many activation functions were proposed during the years and each one has its characteristics but not an appropriate field of application. The better activation function to use in a particular situation (to a particular problem) is still an open question. Each one has its pro and cons in some situations so each Neural Network library implements a wide range of them and it leaves to the user to perform his own tests. In Tab. 1.1 we show the list of activation functions implemented in our libraries with mathematical formulation and corresponding derivative (ref. activations.py for the code implementation). An important feature of any activation function, in fact, is that it should be differentiable since the main procedure of model optimization implies the backpropagation of the error gradients.

As can be seen in Tab. 1.1 it is easier to compute the activation function derivative as function of it. This is a (well known) important type of optimization in computation term since it reduces the number of operations and it allows to apply the backward gradient directly.

To better understand the effects of activation functions we can apply these functions on a simple test image and see the results. This can be easy done using the example scripts inserted inside our library.

In Fig. 1.2 the effects of the told above functions are reported on a test image. For each function we show the output of the activation function and its gradient. For visualization purposes the image values are rescaled $\in [-1, 1]$ before the input to the functions. From the results given in Fig. 1.2 we can better appreciate the differences between the mathematical formulas: a simple Logistic function does not produce evident effects on the test image while a Relu activations tends to overshadow the image pixels. This features of the Relu activation function are very useful in Neural Network model and they also determine important theoretical consequences which led it to be one of the most prominent solution for many Neural Network models.

The ReLU (Rectified Linear Unit) activation functions are, in fact, the most used into the modern Neural Networks models. Their diffusion is imputed to their numerical efficiency and to the benefits they bring [12]:

[14] We assume for simplicity a fully connected Neural Network neuron.

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

Figure 1.2: Activation functions applied on a testing image. **(top)** Elu function and corresponding gradient. **(center)** Logistic function and corresponding gradient. **(bottom)** Relu function and corresponding gradient.

- Information disentangling: the main purpose of a Neural Network model is to tune a discriminant function able to associate a set of input to a prior-known output classes. A dense information representation is considered *entangled* because small differences in input highly modifies the data representation inside the network. On the other hand, a sparse representation tends to guarantee a conservation of the learning features.
- Information representation: different inputs can lead different quantities of useful informations. The possibility to have null values in output (ref Tab. 1.1) allows a better representation of the representation dimension inside the network.
- Sparsity: sparsity representation of data are exponentially efficient in comparison to dense ones, where the exponential power is given by the number of no-null features [12].
- Vanish gradient reduction: if the activation output is positive we have a no-bound gradient value.

In the next sections we will discuss about different kind of Neural Network models and in all of them we choose to use Relu activation function in the major part of the layers.

1.1.4 Convolution function

A big revolution into the Neural Network research field was given by the introduction of the convolution functions. Convolutional Neural Network (CNN) are particularly designed for image analysis. Convolution is the mathematical integration of two functions in which the second one is translated by a given value:

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

In signal processing this operation is also called *crossing correlation* ad it is equivalent to the *autocorrelation* function computed in a given point. In image processing the first function is represented by the image I and the second one is a kernel k (or filter) which shifts along the image. In this case we will have a 2D discrete version of the formula given by:

$$C = k * I$$

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

where $C[i,j]$ is the pixel value of the resulting image and N, M are kernel dimensions.

The use of CNN in modern image analysis applications can be traced back to multiple causes. First of all the image dimensions are increasingly bigger and thus the number of variables/features, i.e pixels, is often too big to manage with standard DNN[15]. Moreover if we consider detection problems, i.e the problem of detecting a set of features (or an object) inside a larger pattern, we want a system ables to recognize the object regardless of where it appears into the input. In other words, we want that our model would be independent by simple translations.

Both the above problems can overcome by CNN models using a small kernel, i.e weight mask, which maps the full input. A CNN is able to successfully capture the spatial and temporal dependencies in an signal through the application of relevant filters.

The main parameter of this function are so given by the input dimensions and the filter/kernel dimensions, i.e the number of weight which we have to tune during the training. This is the basic idea behind the convolution function but in many cases (especially in modern deep learning neural network) we can sophisticate it playing with the possible movements of the filter mask. In particular, aside the kernel mask-size, we can also force the filter to jump along the image, i.e a discontinuous movement of the filter excluding some pixels. This parameter, called stride, defines the number of pixels to jump and it is often used to reduce further the output dimensions.

Given this theoretical background we can implement the convolution function in many different ways, using different mathematical approaches: a study on the computational efficiency will tell us which is the best approach to choose. The first (naive) approach is to use a brute force technique and implement the direct evaluation of the convolution functions as described above. This version is certainly the easier to implement but its computational performances are so worst than for sake of brevity we excluded it from our tests[16].

Taking into account what we have learned from the DNN models, we can re-formulate our problem using an efficient manipulation of the involved matrices to optimize the GEMM algorithm. A direct convolution on an image of size $(W \times H \times C)$ using a kernel mask of dimensions $(k \times k)$ requires $O(WHCk^2)$ operations and thus many matrix products. We can re-arrange the involved data to optimize this computation and thus evaluate a single matrix product: this re-arrangement is called im2col (or im2row) algorithm. The algorithm

[15] If we consider a simple image 224×224 with 3 color channels we obtain a set of $150'528$ features. A classical DNN layer with this input size should have 1024 nodes for a total of more than 150 million weights to tune.

[16] Compared to the other implementations the direct (brute force) convolution algorithm exceeds the computational time of order of magnitudes. For this reason it is not taken into account during our tests. A possible implementation in C++ is however provided into the Byron library.

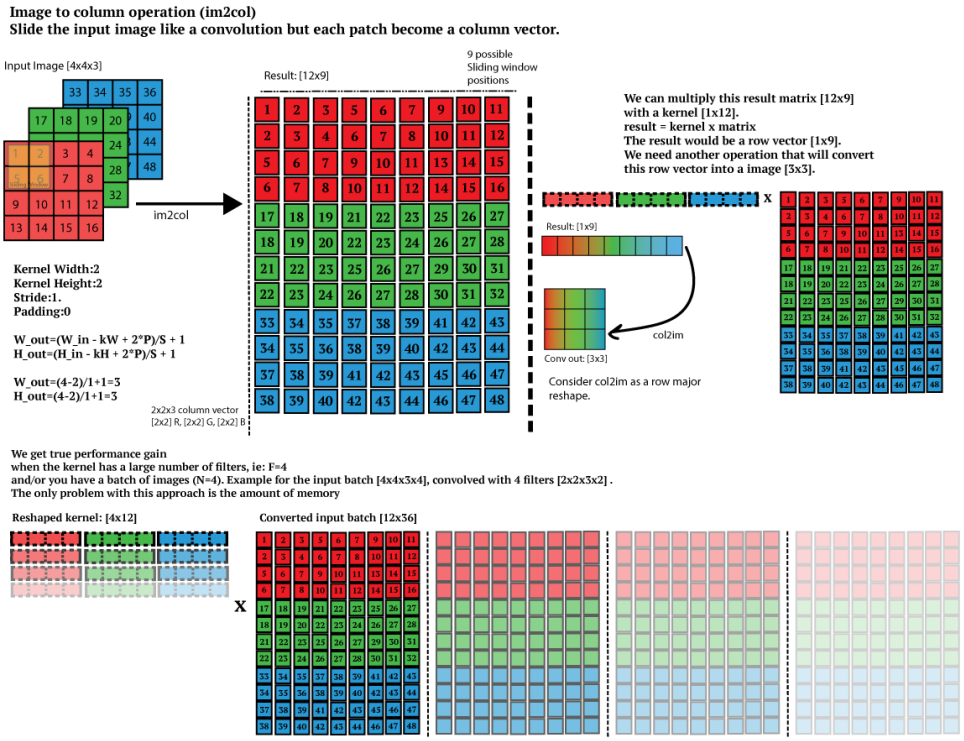

Figure 1.3: im2col algorithm scheme using a 2×2 filter on a image with 3 channels. At the end of the im2col algorithm the GEMM is performed between weights and input image.

is just a simple transformation which flats the original input into a bigger matrix where each column carries all the elements which have to be multiplied for the filter mask into a single step[17]. In this way we can immediately apply our GEMM algorithm on the full image. In Fig. 1.3 the main scheme of this algorithm is reported. This kind of algorithm certainly optimize the computation efficiency of the GEMM product but in payback we have to store a lot of memory for the input re-organization.

Using the mathematical theory behind the problem a third idea can arise using the well known Convolution Theorem: the Fourier transformation of our functions (that in this case are given by the input image and the weights kernel) can be reinterpreted into a simple matrix product in the frequency space. This is certainly the most "physical" approach to solve this problem and probably the easier one since the Fourier transformation is a well-known optimized algorithm and many efficient implementations are already provided in literature. One of the most efficient one is provided by the FFTW (*Fast Fourier Transform in the West*) library [11]: the FFTW3 is an open source C subroutine library for computing the discrete Fourier transform (DFT) in multiple dimensions without constrains in input sizes or data types. The library is not only accurate in the computation but it also provide an efficient parallel version for multi-threading applications.

A further implementation kind is given by linear algebra considerations (very closed to numerical considerations) and it is called Coppersmith-Winograd algorithm. This algorithm was designed to optimize the matrix product and in particular to reduce the computational cost of its operations. Suppose we have an input image given by just 4 elements and a filter mask with size equal to 3:

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

[17] We work under the assumption that the weights matrix is already a flatten array and thus each row of the weights matrix represents the full mask.

we can now using the told above im2col algorithm and thus reshape our input image and weights into

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

given this data we can simply compute the output as the matrix product of this two matrices. The Winograd algorithm rewrites this computation as follow:

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

where

$$m1 = (d0 - d2)g0 \qquad m2 = (d1 + d2)\frac{g0 + g1 + g2}{2}$$

$$m4 = (d1 - d3)g2 \qquad m3 = (d2 - d1)\frac{g0 - g1 + g2}{2}$$

where we can easily notice that the two fractions in $m2$ and $m3$ involve only weight quantities and thus they have to be computed only one time for each filter (at each step). Moreover we have to manage 4 ADD and 4 MUL operations to calculate the m_i quantities and 4 other ADD to compute the result. In doing normal matrix products we have to do 6 MUL operations instead of 4. This is reducing computationally expensive MUL operations by a factor of 1.5x which is very significant[18]. In this simple example we use a so-called $F(4, 3)$, i.e image of size 4 and kernel of size 3 which gives us 2 convolutions. More general formulations are $F(m \times m, r \times r)$ and if we use an image of size 4×4 and a kernel of size 3×3 we can compare the 16 MULs of the Winograd algorithm against the 36 MULs which are required by the normal matrix product (2.25x). The Winograd efficiency was widely proofed for Convolutional network models, especially when the kernel size is small. In our Byron library we provide its implementation for kernel sizes equal to 3 since the numerical generalization is not straight-forward[19].

To test which algorithm could be more appropriated for Neural Network models we tested their computational time efficiency on different random images. The tests were performed on a classical bioinformatics server (128 GB RAM memory and 2 CPU E5-2620, with 8 cores each) and we considered only kernel sizes equal to 3 (Winograd constrain) varying the input dimensions and the number of filters. In Fig. 1.4 we show the result of our simulations using the im2col values as reference[20].

In all our simulations we found a visible speedup using the Winograd algorithm against the other two algorithms: for small dimensions we obtain more than 5x against the im2col and 25x against the fftw implementation. The worst algorithm is certainly the fftw one which, despite the efficient FFTW3 parallel-library, is always more than 5 times slower than the reference. However, it is interesting notice how the fftw implementation is able to reach the best performances when the dimensions are proportional to powers of 2, as expected from the mathematical theory behind the Discrete Fourier Transformation.

[18] A multiplication takes 7 clock cycles in a normal CPU while an add takes only 3 clock cycles.

[19] We would also highlight that this formulation is valid only if we consider unitary strides.

[20] The im2col algorithm can be found in the major part of Neural Network library and it is also the only convolution function implemented in the darknet library, which is a sort of reference for our work.

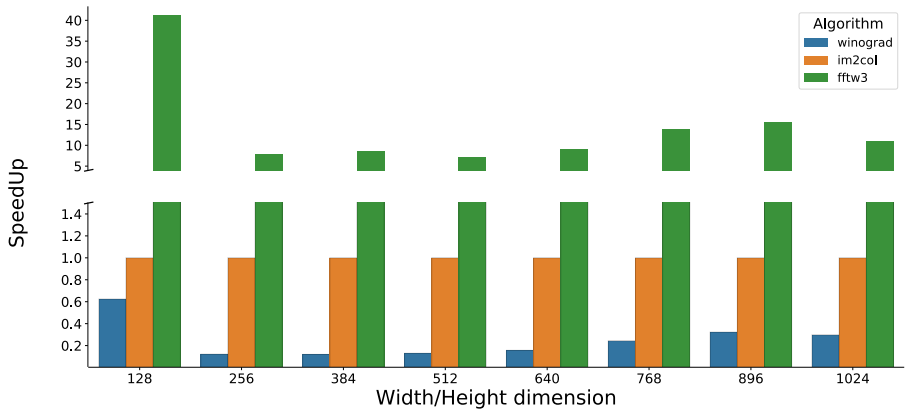

Figure 1.4: Time performances of different convolution algorithms: im2col (orange, reference), FFTW3 (green, fast Fourier transformation using the FFTW3 library) and Winograd (blue). The values are normalized according to the im2col results since it is the most common convolution algorithm. The tests were performed on different input sizes (width/height), keeping fixed the number of channels and the number of filters. The tests were performed using a C++ implementation of the three methods.

We can conclude that the Winograd algorithm is certainly the best choice when we have to perform a 2D convolution. The payback of this method is given by the rigid constrains related to the mask sizes and strides: when it is possible it remains the best solution but in all the other cases the im2col implementation is a relatively good alternative. The efficiency of Byron library follows the efficiency of the Winograd algorithm since the major part of layers in modern deep learning Neural Network models are Convolutional layers with size equal to 3 and unitary stride.

1.1.5 Pooling function

Output Neural Network feature maps often suffer of sensitivity on feature location in the input. One possible approach to overcome this problem is to down sample the feature maps making the resulting feature map more robust to changes in the position. Pooling functions perform this kind of down sample and they reduce the spatial dimension (but not depth) of the input. Their use represents an important computational performance improver tool (less feature, less operations) and a useful dimensionality reduction method. The reduction of feature quantity can also prevent over-fitting problems and it improves the classification performances.

Pooling layers are intrinsically related to Convolutional layers. The analogy lives in the filter mapping procedure which produces the output in both methods. While in the Convolutional layer we map a filter over the input signal and we apply a multiplication of the layer weights and the signal values, in the pooling layer we simply change the filter function keeping the same filter mapping procedure (see section 1.1.4 for more informations). The input parameters of the method are the same of the Convolutional one: the input dimensions, the kernel size and (optional) the stride value.

The most common pooling layers are the Average Pool and the Maximum Pool. The Average Pool layer performs a down sampling on the batch of images. It slides a 2D kernel of arbitrary size over the image and the output is the mean value of the pixels inside the kernel. In Fig. 1.5 are shown some results obtained by performing an average pool with different kernel sizes. Also in this case this test was obtained using our NumPyNet library.

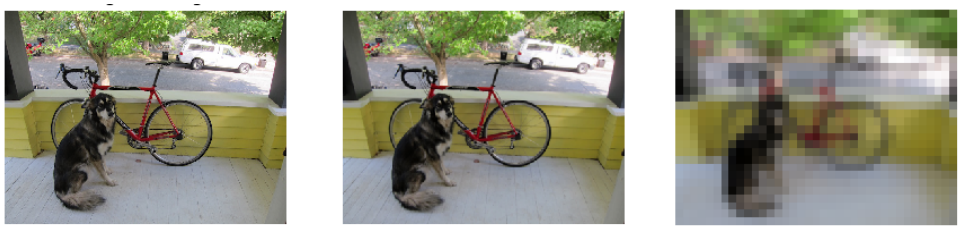

Figure 1.5: Average Pool functions applied on a testing image. **(left)** The original image. **(center)** Average Pool output obtained with a kernel mask (3×3). **(right)** Average Pool output obtained with a kernel mask (30×30).

If in the Convolutional layers a key role was played by the matrix product, in the Pooling layers we have to carefully manage the mapping operations to obtain optimal results. In particular we would to show the efficient implementation provided into NumPyNet.

In the previous sections we introduced the im2col algorithm which is an efficient method to re-organize the input data. The same algorithm can also be applied for Pooling layers and thus evaluate the Pooling function (avg, max, etc.) on each row of the re-arranged matrix. The implementation of the im2col algorithm in Python requires the evaluation of multiple indexes using complex formulas. Since the NumPyNet library was founded on the Numpy package we can provide an alternative implementation using the view functionality of the library. A view of a given array is simply another way of viewing its data: technically that means that the data of both objects is shared and thus no copies are created. In particular we can use the deeper functions of the Numpy package to create a re-organization of our data according to the desired output[21]. In the following code we show our implementation of the Average Pooling layer:

Listing 1.1: 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)

```

[21] The same technique was also used for the implementation of the Convolutional layer in the NumPyNet library.

```
25      subs = np.lib.stride_tricks.as_strided(input, view_shape, strides=
        strides)
26      # returns a view with shape = (batch, out_w, out_h, out_c, kx, ky)
27      return subs
28
29  def forward(self, input):
30
31      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 key role in this implementation is played by the _asStride function: it returns a view of the original array in which all the masks are organized into a single list. Using this data re-arrangement we can easily compute the desired pooling function (average in this example) according to the appropriated axes. We would stress that no copies are produced during this computation and thus we can obtain a faster execution than other possible implementations (e.g im2col).

1.1.6 BatchNorm function

A common practice before the training of a Neural Network model is to apply some pre-processing to the input patterns. A classical example is the normalization of training set, i.e it resembles a normal distribution with zero mean and unitary variance. The initial preprocessing is useful to prevent the early saturation of non-linear activation functions (see section 1.1.3). Moreover in this case we can ensure that all inputs are in the same range of values.

In a deep neural network architecture we can find the same problem also into the intermediate layers because the distribution of the activations is constantly changing during training. This behavior produces a slowdown in the training convergence because each layer have to adapt itself to a new distribution of data in every training step (or *epoch*). This problem is also called *internal covariate shift*.

A second problem arises from the heterogeneity of available input data. If we tune the model parameters according to a given set of data, which inevitably will be limited, we can meet problems during the generalization, i.e the validation of our model using new data, to new samples if they belongs to an equivalent but deformed distribution: this kind of problem passes under the name of *over-fitting*. A classical example is given by the image detection: if we train a Neural Network model using gray-scale images we can find generalization problems using colored images. This problem can be solve using regularization techniques.

BatchNorm function (Batch Normalization) allows to overcome these problems with a continuous rescaling of the Neural Network intermediate values during the training[22] [16]. In this way we can ensure more stability of the extracted features [18] during the training and a faster convergence.

[22] The input data to feed the Neural Network model are commonly packed into a series of *batches*, i.e small subsets of data. The BatchNorm function takes its name from this nomenclature and it processes each batch independently.

In particular, the method processes the input of a given layer in order to fight the internal covariate shift problem removing the batch mean and normalizing by the batch variance:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad {\sigma_B}^2 = \frac{1}{m-1}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

where m represents the batch-size and x_i is the value of the pixel x in the i-th image of the batch ($\in [0, m]$). Thus the input data becomes:

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{{\sigma_B}^2 + \epsilon}}$$

where we add an extra ϵ in the denominator for numerical stability[23]. After this common rescaling we also a apply a scaling-shift to previous results:

$$y_i = \gamma\hat{x_i} + \beta$$

where the γ and β coefficients are left as variables to be tuned during the training (they are learned during training). The updating rule of the function parameters (γ and β) is given by the derivative of the previous functions:

$$\delta\beta = \sum_{i=1}^{m} {\delta_i}^l \qquad \delta\gamma = \sum_{i=0}^{m} {\delta_i}^l \cdot \mu_B$$

where δ^l is the error passed from the next layer of the network structure. To complete the error propagation we have also compute the derivative of the BatchNorm function output:

$${\delta_i}^{l-1} = \frac{m \cdot \delta\hat{x_i} - \sum_{j=1}^{m}\delta\hat{x_i} - \hat{x_i}\cdot\sum_{j=1}^{m}\delta\hat{x_i}\cdot\hat{x_i}}{m\cdot\sqrt{{\sigma_B}^2 + \epsilon}}$$

Since the BatchNorm function is became a sort of standard into a deep learning models, an efficient implementation of this algorithm is essential to achieve the best computational performances. We have also to take into account that the batch-normalization procedure is commonly performed after a fully-connected layer or a convolutional one. Thus the best performances could be obtained merging the two functionality as much as possible as suggested in [3].

The Byron library was inspired by the *darknet* library provided by Redmon J. et al. and by its many branches. Despite in each implementation we can find the BatchNorm function, aware of the author, in any version we can find a right implementation of this function as standalone method. We have already highlighted that this normalization function can be efficiently joined to other function to increase the computational performances but in these case we have to different manage the dimensions of the involved arrays. A standalone implementation of the BatchNorm function required a rearrangement of its functions and it was provided into the Byron library. This was one of the various improvements provided by Byron against the other *darknet*-like libraries.

Other common regularization techniques are given by the regularization of the neuron outputs with penalty loss functions. Classical examples are given by the L1 (Laplacian) and L2 (Gaussian) penalties. Both these functions are implemented either in NumPyNet and Byron but for sake of brevity we will not discuss about them.

[23] The floating point numbers into a computer have finite precision and the variance can underflow bringing to infinite values in the BatchNorm equation.

1.1.7 Dropout function

Many times along this work we have been talked about the *over-fitting* problem. The over-fitting problems arise when the complexity of our model becomes too high regard the amount of available data, i.e when the number of parameters of our model is comparable to the number of available data. A classical example is given by the polynomial fitting problem. Given an initial set of N data points we can always find a polynomial curve of degree equal to $N-1$ which can perfectly fit our data. In this case the model flexibility is minimum and new additional data points difficulty lies on the same curve. In other words we tuned each model parameter according to the given data set but we completely lose the possibility of generalization.

In Neural Network models we have to manage a large quantities of parameters and it is quite easy to stumble on this problem. Possible workaround could be given by the regularization techniques told in the previous section (ref. 1.1.6 for further informations) or by a Dropout function. This function simply dropping out some neuron units into a Neural Network during the training phase. Ignore some neurons means that they will not be considered during a particular (single) forward/backward step. So, given a set of neurons we have a probability p to keep the neuron and $1-p$ to remove it. In this way we can reduce the co-dependency of nearest neurons inside the network and so reduce the possibility of over-fitting.

The above description bring us to a straightforward implementation of the algorithm into the NumPyNet library (ref. 1.2).

Listing 1.2: NumPyNet version of Dropout function

```
import numpy as np

class Dropout_layer(object):

  def __init__(self, prob):

    self.probability = prob
    self.scale = 1. / (1. - prob) if prob != 1 else 1.

    self.out_shape = None
    self.output, self.delta = (None, None)

  def forward(self, input):

    self.out_shape = input.shape

    self.rnd = np.random.uniform(low=0., high=1., size=self.out_shape) <
    self.probability
    self.output = self.rnd * input * self.scale
    self.delta = np.zeros(shape=input.shape)

  def backward(self, delta=None):

    if delta is not None:
      self.delta = self.rnd * self.delta * self.scale
      delta[:] = self.delta.copy()
```

The above code numerically reproduce the theoretical formulation given. After the initialization of the private object variables, the forward function generates a set of random positions and apply them (if they are less than the given probability) to the output: these positions will be turned off and the others will be multiply by a scale probability factor to increase their importance. The backward function simply invert the transformation on the back-propagated gradient delta.

Despite this straightforward implementation we have to carefully manage some crucial

points into the C++ equivalent. The Byron library works into a single parallel region so after the (sequential) initialization of the layer object the forward/backward phases are evaluated by all the available threads in parallel. This bring us to a standard problem in multi-threading programming: the generation of independent random numbers among threads. Inside a parallel region all the declared variables are (by definition) shared among all the available threads. Thus, if we simply create a random number generator we have to face on the thread-concurrency. As consequence the random number generated will not be independent but (most probably[24]) repeated by each thread. The simple workaround implemented into the Byron library is given by assigning a random number generator to each thread (with its own seed and indexed by the thread ID). In this way we can ensure a totally independence of the random numbers generated during the forward phase (ref. on-line).

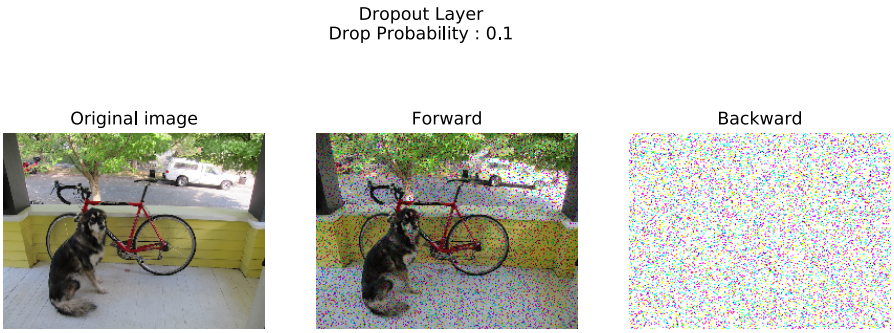

Figure 1.6: Dropout function applied on a testing image. The 10% of image pixels are turned off by the forward function. The corresponding gradient is back-propagated only on the previously activated pixels.

As visualization example we can use our simple test image and apply our transformation (see Fig.1.6). Our input image shows many pixel turned off according to the given probability, as expected. On the other hand, the backward output turns on only the same pixel[25].

A usage example of this functions is provided into the NumPyNet examples: in those simple examples we can easily compare the learning performances of standard neural network models with and without the Dropout function on classical datasets.

1.1.8 Shortcut connections

The harder becomes the problem to solve and the deeper[26] will be the Neural Network model created to solve it. The payback of these deep network structures is a reduction in accuracy after reaching a maximum, the so-called *degradation problem*. This accuracy reduction does not arise from over-fitting problems but it is due to numerical instabilities (*vanishing gradient* - as the gradient is back-propagated to earlier layers, repeated multiplications may make the gradient very small) and troubles related to the data dimensionality (called *curse of dimensionality*). Despite Neural Network could be defined as universal function approximators, adding numerous layers and thus parameters, the result in accuracy does not grow proportionally. With simple empirical examples we can easily see how the accuracy starts to saturate (and eventually degrade) with an increasing number of

[24] The deterministic generation of random number is hard to reproduce into a parallel environment despite the seed initialization. The "probability" of repeating the same sequence is related to the affinity of each thread to the given process.

[25] For visualization purposes we manually set the gradient to a uniform value.

[26] The deep of a Neural Network model is related to the number of layers which made it.

layers. Those problems poses a limit to the number of layers usable on a Neural Network model and seem that the shallower networks learn better than their deeper counterparts. Keeping this results in mind we can think about a strategy to skip these "extra" layers.

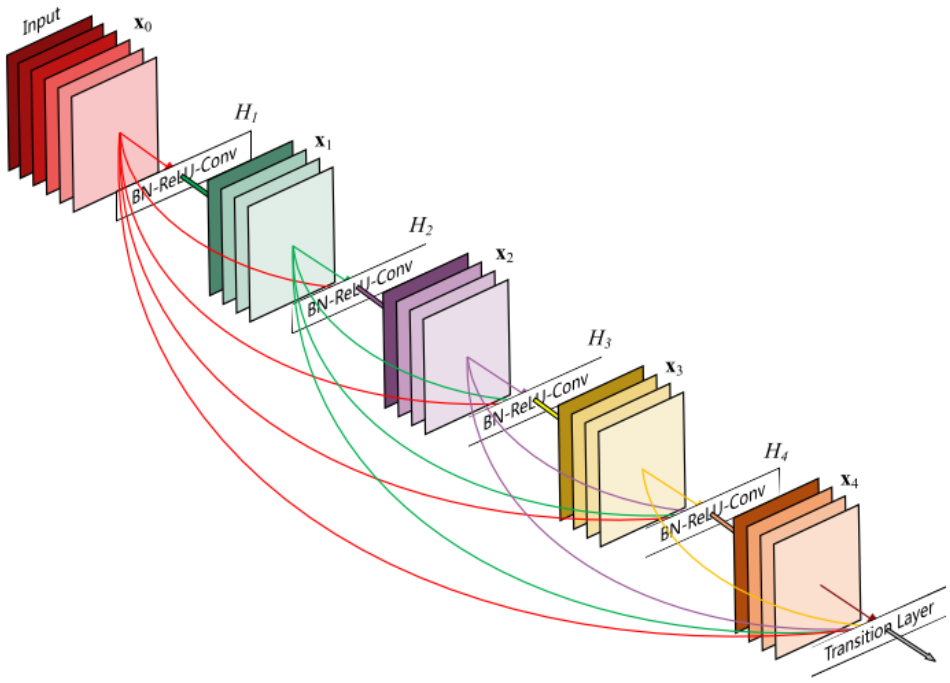

Figure 1.7: Scheme of shortcut connections into a deep learning model. Each colored line connects the previous layer block to the following one. The output combination can be customized but the most used one is a simple linear combination of them. A particular attention must be payed with the dimensions management.

We can obtain a simple solution to this problem making extra connections between layers called shortcuts or residuals. A shortcut is a link between two distant layers without involving the set of layers between them, a so-called "identity shortcut connection". A graphical example is show in Fig. 1.7. The authors of [13] argue that stacking layers should not degrade the network performance, because we could simply stack identity mappings (layer that does not do anything) upon the current network, and the resulting architecture would perform the same. In the original paper, the shortcut connections perform an operation like:

$$H(x) = F(x_1) + x_2$$

where $F(x)$ is the output of the previous block and x is the output of the current block. The function F generalizes the combination of these two values[27].

The introduction of these extra connections bring us to the ResNet (Residual Neural Network) models era in which a key role was played by the object detection models. A wide

[27] In our implementations we choose to generalize this formula as

$$H(x) = \alpha x_1 + \beta x_2$$

range of modern deep learning architectures use this kind of connections and in this way they involve a large number of layers: famous examples of this kind are the VGG models and the ResNets. We have done a large use of this connections also in the models described in the next sections, either for object detection purposes (ref. 1.3), Super Resolution (ref. 1.2) and mostly for our segmentation (ref. 1.4) applications. This kind of functions are becoming so popular into the modern deep learning models that more and more often we describe a model according to its *residual blocks*, i.e the layer ensemble between two shortcut connection.

From a computational point of view the implementation of this kind of "layers" is straightforward in Python (and thus in our NumPyNet): we can easily implement a network structure as a list of object and thus a shortcut connection simply combine the output of two elements of it. We met more problems when we translated this idea into C++. The C++ language is more rigid with the data type involved in each operation and we have to carefully manage the "signature" (list of input arguments) of each function. In this way we can not simply implement a list of different object types as a network structure.

A possible solution can reached using the object inheritance: we can create a single Base_layer object and specialize it according to our needs. This is certainly the most C++-like solution but it requires many checks (if statements) at execution time. An other (more modern) solution is provided by the new (standard) data types provided by the C++17: in particular we refer to the variant objects. A variant is a template union data-type which allow to combine and reinterpret different data types into a single object. The most important consequence of the use of this kind of data-type is that we can easily jump to one type to an other using constexpr statement which (by definition) are solved at compile time. Besides the particulars involved into this kind of implementation is important to notice that the difference between the two solution is the same between run-time and compile-time: if we perform computation at compile-time we will not re-execute when the code runs and thus we can reach better time performances. The Byron library widely use templates and with the support of the C++17 standard a large part of costly operations are execute one-for-all at compile time[28].

Using variant object and templates we can easily implement a shortcut connection also in C++ as can be seen on the on-line version of the code (ref. on-line).

1.1.9 Pixel Shuffle

Pixel Shuffle layer is one of the most recent layer type introduced in modern deep learning Neural Network. Its application is closely related to the single-image super-resolution (SISR) research, i.e the ensemble techniques which aim at restoring a high-resolution image from a single low-resolution one (see section 1.2 for further details).

The first SISR Neural Networks start with a preprocessing of low-resolution images in input with a bi-cubic up-sampling. Then the image, with the same dimensions of the desired output, feeds the model which aim to increase the resolution and fix its details. In this way the amount of parameters and moreover the computation required by the training section increase (by a factor equal to the square of the desired up-sampling scale), despite the required image processing is smaller. To overcome this problem a Pixel Shuffle transformation, also known as *sub-pixel convolution*, was introduced [27]: in this work the authors proofed the equivalence between a regular transpose convolution, i.e the previous standard transformation to enlarge the input dimensions, and the sub-pixel convolution transformation without losing any information. The Pixel Shuffle transformation reorganize the low-resolution image channels to obtain a bigger image with few channels. An example of this transformation is shown in Fig. 1.8.

[28] We provide also an efficient retro-compatibility for "old-standard users" with a custom implementation

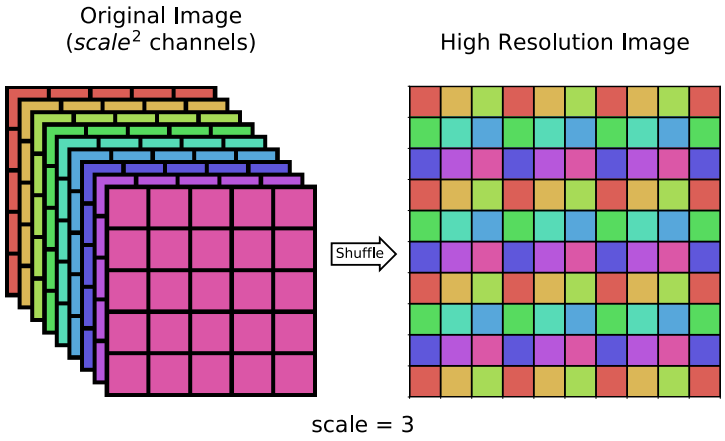

Figure 1.8: Pixel Shuffle transformation. On the left the input image with $scale^2$ ($:= 9$) channels. On the right the result of Pixel Shuffle transformation. Since the number of channels is perfect square the output is a single channel image with the rearrangement of the original ones.

Pixel Shuffle rearranges the elements of the input tensor expressed as $H \times W \times C^2$ to form a $scale \cdot H \times scale \cdot W \times C$ tensor. This can be very useful after a convolution process, in which the number of filters chosen drastically increase the number of channels, to "invert" the transformation like a sort of *deconvolution* function.

The main gain in using this transformation is the increment of computational efficiency of the Neural Network model. The introduction of Pixel Shuffle transformation in the Neural Network tail, i.e after a sequence of small processing steps which increase the number of features, reorganize the set of features into a single bigger image, i.e the desired output in a SISR application. The feature processing steps, which generally are faced on with convolutional layers, can be performed with smaller images in input and thus can be obtained faster since the up-scaling task will be performed by a single Pixel Shuffle transformation.

Despite this transformation has became a standard in super-resolution applications and thus it can be found into the most common deep learning libraries (e.g *Pytorch* and *Tensorflow*) a C++ implementation is hard to find. Moreover, each library implements the transformation following its own data organization[29]. For this reason we proposed in our libraries a dynamic version of the algorithm in C++ able to perform both versions of the algorithm.

The algorithmic implementation of the pixel-shuffle transformation is essentially a re-indexing of the input values. While in a C++ implementation of the algorithm we could obtain the desired result inside a sequence of nested for loops playing with the loop indexes, for an efficient Python version we used a sequence of transposition and reshaping to rearrange the input values. The following snippet shows the NumPyNet version of this algorithm.

Listing 1.3: NumPyNet version of Pixel-Shuffle function

```
1 import numpy as np
2
3 class Shuffler_layer(object):
```

of variant objects.

[29] The main difference between *Pytorch* and *Tensorflow* is related to the storage organization of the image. *Tensorflow* has a "standard" input assessment as $H \times W \times C$. *Pytorch* has a so-called channel-first implementation and so the input tensor is organized as $C \times H \times W$.

```
4
5   def __init__(self, scale):
6
7     self.scale = scale
8     self.scale_step = scale * scale
9
10    self.batch, self.w, self.h, self.c = (0, 0, 0, 0)
11
12    self.output, self.delta = (None, None)
13
14  def _phase_shift(self, input, scale):
15    b, w, h, c = input.shape
16    X = input.transpose(1, 2, 3, 0).reshape(w, h, scale, scale, b)
17    X = np.concatenate(X, axis=1)
18    X = np.concatenate(X, axis=1)
19    X = X.transpose(2, 0, 1)
20    return np.reshape(X, (b, w * scale, h * scale, 1))
21
22  def _reverse(self, delta, scale):
23    # This function apply numpy.split as a reverse function to numpy.
      concatenate
24    # along the same axis also
25
26    delta = delta.transpose(1, 2, 0)
27
28    delta = np.asarray(np.split(delta, self.h, axis=1))
29    delta = np.asarray(np.split(delta, self.w, axis=1))
30    delta = delta.reshape(self.w, self.h, scale * scale, self.batch)
31
32    # It returns an output of the correct shape (batch, in_w, in_h, scale
      **2)
33    # for the concatenate in the backward function
34    return delta.transpose(3, 0, 1, 2)
35
36  def forward(self, input):
37
38    self.batch, self.w, self.h, self.c = input.shape
39
40    channel_output = self.c // self.scale_step # out_c
41
42    # The function phase shift receives only in_c // out_c channels at a
      time
43    # the concatenate stitches together every output of the function.
44
45    self.output = np.concatenate([self._phase_shift(input[:, :, :, range(i,
       self.c, channel_output)], self.scale)
46                                  for i in range(channel_output)], axis=3)
47
48    self.delta = np.zeros(shape=self.out_shape, dtype=float)
49
50  def backward(self, delta):
51
52    channel_out = self.c // self.scale_step # out_c
53
54    # I apply the reverse function only for a single channel
55    X = np.concatenate([self._reverse(self.delta[:, :, :, i], self.scale)
56                                      for i in range(channel_out)], axis=3)
57
58
59    # The 'reverse' concatenate actually put the correct channels together
      but in a
60    # weird order, so this part sorts the 'layers' correctly
61    idx = sum([list(range(i, self.c, channel_out)) for i in range(
```

```
    channel_out)], [])
62    idx = np.argsort(idx)
63
64    delta[:] = X[:, :, :, idx]
```

The two functions _phase_shift and _reverse[30] produce the re-arrangement of the indexes according to the pixel-shuffle transformation and its inversion[31] In the forward function we apply the _phase_shift to the sequence of channels (in the right order) and then we concatenate the results into a single tensor (output). The backward function instead needs a re-ordering of channel sequence after the concatenation.

As told above, in the C++ implementation provided into the Byron library we can compute the desired re-indexing using a series of nested for loops. An equivalent solution can be obtained also by the contraction of the loops into a single one using divisions to obtain the right indexes. This solution was taken in count into the first version of the library but the amount of required divisions weights on the computational performances. The division operations are the most computationally expensive operations in terms of CPU clock-time. The old versions of OpenMP multi-threading library forced the users to spend time in the evaluation of the "loop-contraction" to obtain the better performances by a single parallel for loop. The new features of OpenMP library provide the very powerful *collapse* keyword which performs an automatic loop-contraction. The keyword can be applied only with a series of independent and perfectly nested[32] for loops which is exactly our case. Moreover we have not to take in care any thread concurrency trouble since the iterations, as the output indexes, are totally independent. We widely used the collapse keyword in the Byron library to simplify the code and the function evaluation but the Pixel-Shuffle case is one of the most efficient one, since we could collapse six nested loops[33] (ref. on-line.

1.1.10 Cost function

A machine learning algorithm is used to minimize or maximize a cost function. In other words when we implement a machine learning algorithm we want to know how good is our result according to prior knowledge about the desired results. So we have to establish a function ables to represent the error of our model. This kind of function are commonly called *error functions* or *loss functions* or just simply *cost function*. In the previous sections we have shown many algorithms used into a Neural Network model and we have talked about how to update the functional parameters according to the evaluated error. This error is provided by the cost function.

The cost function represents the final output of our Neural Network model so it is reasonable to talk about it at the end of this chapter. There are many kinds of loss functions and there is not a particular one able to works with all kinds of data. So we have to pay attention to chose the right one in our problems. In particular we have to take in count the possible presence of outliers, the structure of our model, the computational efficiency of our algorithm and most of all the number of classes that we want to predict. Broadly, we can classify the loss functions into two major categories: the classification losses and the regression losses. In the first case we want to predict a finite number of categorical values (classes). In the second case the prediction is performed on a series of continuous values. Since in this work we are focusing only on classification problems we will only talk about the first case.

[30] These function are "private" function of the object class.

[31] During the back-propagation, in fact, we have to apply the reverse transformation to the gradient.

[32] Two for loops are perfectly nested if there are not other code lines between them.

[33] In the Pixel-Shuffle we have to loop over batch, width, height, channels plus a couple of loops over the scale factor that we want to apply. In total we have to manage six dimensions that can be easily collapsed into a single one given by their product.

The most common cost function is given by the *Mean Square Error* (MSE) or *L2 loss* (very closed to the regularization function hinted at the end of 1.1.6). Its mathematical formulation is quite simple and it is given by

$$MSE = \frac{\sum_{i=1}^{N} (y - t)^2}{N}$$

where we follow the nomenclature given in 1.1.1 and N is the number of output which is equivalent to the number of classes. It is one of the most used cost function due to its simplicity either from a mathematical either from a numerical point of view. The possible range of values ranged from 0 to ∞. With MSE function the predictions which are far away from actual values are heavily penalized, due to the squaring.

A slight different function is given by the *Mean Absolute Error* (MAE) or *L1 loss* in which we replace the squaring with a module of the error.

$$MAE = \frac{\sum_{i=1}^{N} |(y - t)|}{N}$$

With MAE we loose the information about the error direction (preserved by the squaring in MSE) and just simply evaluate the absolute value of it.

The main differences between these two functions can be summarized as follow: using the MSE function we can easily solve the problem but the MAE function is more robust against possible outliers. Despite both functions reach the minimum in a perfect classification configuration (error equal to zero), in presence of outliers we have to manage with large differences in the numerator of the function. With large differences, the square values are greater than the absolute values but while the MSE tries to adjust its performance to minimize those cases, the other samples pay the higher cost.

A problem related to the MAE function arises during the gradient evaluation. Its gradient, in fact, is the same throughout, which means that we will have large gradient values also with small differences which is a worse configuration during the training. A simple possible workaround is to introduce a shrinking parameter, given by a dynamic learning rate, when we move closer to the minimum.

When we have to manage multi-class problems there are other common cost functions based on likelihood scores. The simpler one is the *Cross Entropy loss* or *Log loss*:

$$CrossEntropyLoss = -(y \cdot \log(t) + (1 - y) \cdot \log(1 - t))$$

This function just multiply the log of the actual predicted probability by the ground truth class. In this way when we have two classes (e.g $t \in [0, 1]$) we can alternatively nullify the two parts of the function[34]. In this way the loss function heavily penalizes the predictions that are confident but wrong. This function works with binary classification problems where the output classes are binned in $[0, 1]$. For this reason the output of the model must be constrained into the $[0, 1]$ domain and thus a proper activation function should be provided. Classically this loss function is used jointly to the sigmoid activation (ref. 1.1.3) which constrains the output of the model in the desired interval. For this reason in our implementation of the algorithm we chose to merge the sigmoid function and the Log Loss function into a single object[35].

A last duty to mention loss function is the extension of the Log loss to multiple classes, the so-called *Categorical Cross Entropy Loss*.

[34] When the actual label is equal to 1, i.e $y = 1$, the second half of the Log Loss function disappears whereas in case of actual label is equal to 0 the first half is null.

[35] We also try to prevent wrong uses of this loss function for laypersons. This implementation was already suggested by the *darknet* library so we simply propagate it in our implementations.

$$CategoricalCrossEntropyLoss = -\sum i = 1^{N} (y \cdot \log(t))$$

This function generalized the previous one for multiple-classes, i.e for problems where the correct output can be only one. The loss compare the distribution of the predictions, i.e output of the model, with the prior known distribution. In this way only the probability of the true class will be 1 and all the other classes will be set to 0. Also in this case we have to pay attention to the output of our model which is intended as a probability value ranging in $[0, 1]$. In particular this function commonly works jointly to a softmax activation function. As in the previous case we chose to implements this loss function in a separated object associated to the softmax transformation.

Many other loss function can be mentioned to overcome different kind of problems. The list of presented loss function was related to the implementation of the *darknet*-like library which are ported also into the NumPyNet and Byron libraries, i.e either in Python and C++. NumPyNet and Byron libraries also provided a wider list of loss functions to improve the usability of them and improve their computation (and fixed some *darknet* issues). A full list of available loss functions can be found in the on-line version of the libraries with a list of easily visual examples.

A further improvements was given from a numerical point-of-view: many mathematical formulas needs expensive math operations as logarithms and trigonometric functions. An efficient (but approximated) math formulas was implemented both in the C++ and Python to reach faster computational performances. These numerical math operations are widely used into the Byron library to increase the performances despite their used can be turned off by user at compile time in Byron. The full set of functions, in fact, is enclosed into a macro definition (__fmath__) that can be enabled at compile-time.

A classical example of this faster math operation is given by the *fast inverse square root* algorithm, firstly introduced in 1999 in the source code of *Quake III Arena*, a first-person shooter video game. The method is based on a Newton algorithm which can be stopped at the desired precision order: less precision is associated to faster execution, obviously. In our fast math implementation we provide a set of Newton algorithms associated to the most common mathematical operations, like exp, log, sqrt and so on. We tested these implementations against the common standards (Numpy package for Python and std:: for C++) and we compare the time execution performances (we required a precision of at least $10^{-}4$). The obtained results are shown in Fig. 1.9 where we normalized the execution time taking Numpy implementation as reference.

As can be see all the results obtained by the fast math algorithms are faster or at least equals to the standard ones. The C++ version of the fast math is certainly the better choice for an efficient implementation of the algorithms in all the cases and it is interesting to notice how some functions (pow2 and log10) are drastically slower in C++ than in Python, despite the intrinsic overhead of the Python language. This is probably due to particularly optimizations performed by the Numpy package in the implementation of these special cases: if we compare those functions to the general ones (pow and log), in fact, the results confirm the efficiency of the C++ language.

These results highlight the importance of code testing before release it: we have to pay always attention in writing a code and query also the standard choices.

1.2 Super Resolution

The Super Resolution (SR) is a slight novel technique based on Neural Network models which aims to improve the spatial resolution of a given image[36].

[36] The best-known "implementation" of Super Resolution concerns the microscopy super-resolution. In this work we are focusing on algorithms and numerical implementation so we will talk about the numerical

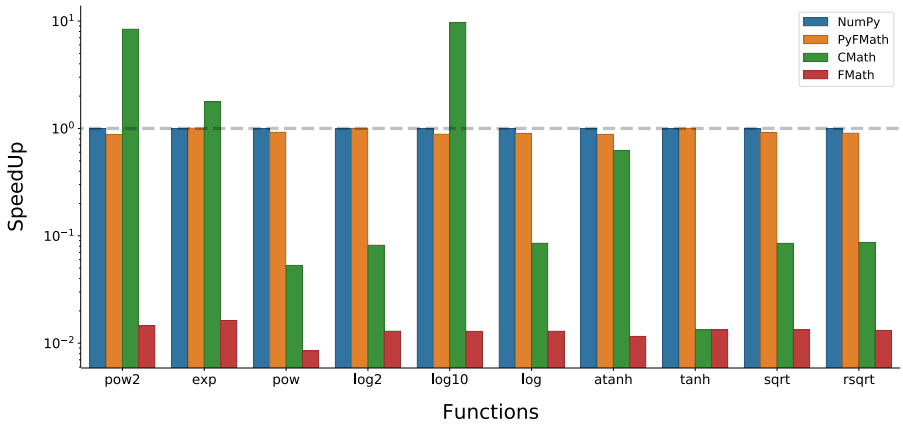

Figure 1.9: Time performances of standard mathematical operations implemented through Newton approximations. We compare the results obtained with the Numpy library (blue, reference) and the standard C++ library (CMath) to their equivalent into the custom FMath version. In the comparison we have to take in mind that the Numpy library is based on a C++ wrap and that the Python version of the FMath is written in pure Python language. In all the cases the FMath version of the functions performs better or at-least-equal to the standard one.

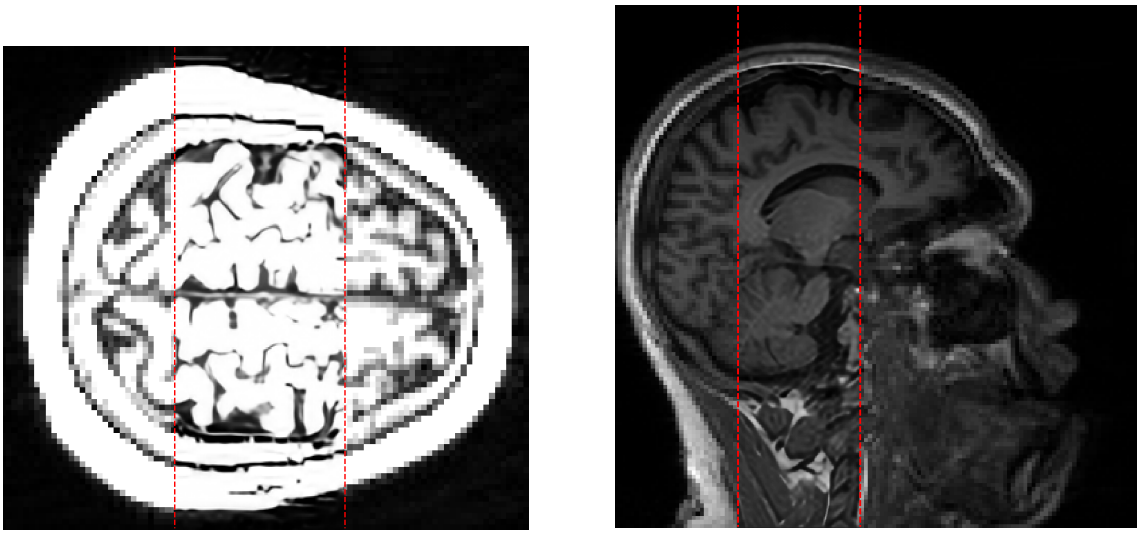

Figure 1.10: Single Image Super Resolution. Between the red lines the super resolved version of the image.

The first SR methods on digital images estimate the high frequency informations of the images starting from a series of low-resolution (LR) patches and their high-resolution (HR) counterpart. These patches (ROIs of the LR image commonly smaller than 50×50) were extracted after an edge enhancement procedure or a simple 2D Fourier transform which extract the high frequency informations. Collecting these patches an "association dictionary" between LR and HR was created. This dictionary was so used to learn the correct association between the LR e HR counterpart and then applied on new images. The considered images could also be of the same dimensions in these firstly applications, i.e the purpose was only to improve the spatial resolution of the image without changing the sampling step.

The idea of use neural network models and in particular convolution functions to face on this problem was born in 2014 at the Engineering University of Honk Kong due to the large popularity of these models during that years. The increasing computational power allowed to create automatic models able to learn the LR-HR association without any dictionary. In this year arise the SRCNN model [10], a three-layer neural network able to learn a large ensemble of features to reproduce the desired association. The first layer aimed to extract the LR patches from the input image; the second layer produce the association between the LR patches and the tuned HR ones; the last layer reorganized the HR patches ensemble produced into a single HR image, i.e the output.

From this starting implementation many improvements was performed in this research field but the fundamental idea is not changed. Modern models simply have a greater number of layers, due to the increasing computational power availability, and they used appropriate workaround to overcome the (large-)parameters tuning problem.

In the next sections we will show the super resolution technique step-by-step starting from the image pre-processing until the most modern algorithmic solutions. At the end of this chapter the NumPyNet and Byron implementation of some modern models will be presented and applied over biomedical images.

1.2.1 Resampling

Up to now we have talked about neural network models as classification algorithms. In the SR problem we have no classes but the desired output is a image. This behavior is often hard to digest but it does not change anything about the previous consideration. The only change will be related to the size of the neural network and its amount of parameters that could drastically increase due to the larger output required. Lets start from the beginning: to feed a super-resolution model we have to use a series of prior-known LR-HR image association. In the real life we always have a series of images, typically LR images, and we want to enlarge the resolution of them, i.e enlarge the spatial dimensions of the input image, to better see some particulars or just to create an output without artifacts or evident pixel grains. If we consider these series of images as the HR one we can easily down-sample them without particular troubles[37]. This re-sampling will introduce a aliasing factor that our model should learn to nullify. The number of model parameters is typically around the 10^7 so if we introduce any filtering process (degradation) in the input image the model will be able to overcome also these problems.

Starting from these considerations we can down-sample our images by a desired scale factor: common scale factor are between 2 and 8 and in this work we will refer to a scaling equal to 4. A crucial role is played by the re-sampling (or down-sampling) algorithm chosen for the artificial image degradation. Any down-sampling algorithm, in fact, loose part of the original information by definition. Thus we can facilitate the learning choosing

counterpart of this technique, totally ignoring the original "hardware" version.

[37] Ignoring particular cases the hardest step is always to enlarge the image resolution and not the inverse step.

a lossless one but in this way we will loose in generalization (the model will not learn how to overcome some cases), or we can apply a drastic down-sample technique and achieve better performances later.

The simpler down-sampling algorithm is given by a *nearest interpolation*. This algorithm pass a kernel mask over the image and it substitutes each pixel mask to their average[38]. This procedure can be achieved using a *Pooling* algorithm (in particular an AveragePooling) (ref. 1.1.5 for further informations) for the down-sample or we can use an UpSample layer. The UpSample function is commonly related to GAN (Generative Adversarial Networks) models in which we have to provide a series of artificial images to a given Neural Network but it is a function which can be introduce inside a Neural Network model to rescale the number of features. We mention it in this section since it is not intrinsically related to a Neural Network model but it could be use as image processing technique.

We provide an implementation of this algorithm either in NumPyNet either in Byron library using different techniques. The UpSample function inside a Neural Network model has to provide both up- and down- sampling technique since one is used in the forward function and its inverse during the back-propagation. To achieve this function in NumPyNet we can use a series of reshapes and striding on the input matrix as shown in the following snippet.

Listing 1.4: NumPyNet version of Upsampling function

```
import numpy as np
from numpy.lib.stride_tricks import as_strided

class Upsample_layer(object):

  def __init__(self, stride=(2, 2), scale=1., **kwargs):

    self.scale = float(scale)
    self.stride = stride

    if not hasattr(self.stride, '__iter__'):
      self.stride = (int(stride), int(stride))

    assert len(self.stride) == 2

    if self.stride[0] < 0 and self.stride[1] < 0: # downsample
      self.stride = (-self.stride[0], -self.stride[1])
      self.reverse = True

    elif self.stride[0] > 0 and self.stride[1] > 0: # upsample
      self.reverse = False

    else:
      raise NotImplementedError('Mixture upsample/downsample are not yet
    implemented')

    self.output, self.delta = (None, None)

  def _downsample (self, input):
    batch, w, h, c = input.shape
    scale_w = w // self.stride[0]
    scale_h = h // self.stride[1]

    return input.reshape(batch, scale_w, self.stride[0], scale_h, self.
    stride[1], c).mean(axis=(2, 4))

  def _upsample (self, input):
```

[38] The inverse (up-sampling) interpolation simply replicates each pixel in each dimension by a number equal to the scale factor.

```
36      batch, w,  h,  c  = input.shape      # number of rows/columns
37      b,     ws, hs, cs = input.strides    # row/column strides
38
39      x = as_strided(input, (batch, w, self.stride[0], h, self.stride[1], c),
         (b, ws, 0, hs, 0, cs)) # view a as larger 4D array
40      return x.reshape(batch, w * self.stride[0], h * self.stride[1], c)
                                        # create new 2D array
41
42    def forward(self, input):
43      self.batch, self.w, self.h, self.c = input.shape
44
45      if self.reverse: # Downsample
46        self.output = self._downsample(input) * self.scale
47
48      else:            # Upsample
49        self.output = self._upsample(input) * self.scale
50
51      self.delta = np.zeros(shape=input.shape, dtype=float)
52
53    def backward(self, delta):
54      if self.reverse: # Upsample
55        delta[:] = self._upsample(self.delta) * (1. / self.scale)
56
57      else:            # Downsample
58        delta[:] = self._downsample(self.delta) * (1. / self.scale)
```

Thus the down-sampling algorithm is obtained reshaping the input array according the two scale factors (strides in the code) along the two dimensions and computing the mean along these axes. Instead the up-sample function use the stride functionality of the Numpy array to rearrange and replicate the value of each pixel in a mask of size strides×strides.

The same functionality can be obtained in the C++ version of the code provided by the Byron library in which we compute the right indexes along a nested sequence of for loops (ref. on-line). We have to take in care the summation reduction provided by the down-sampling according to the thread concurrency: in this case we can not generalize the loop collapsing to the full set of lops but we have to separately manage the summation in a sequential section.

A more sophisticated interpolation algorithm, which reduce the loosing informations, is provided by the *bicubic interpolation*. The re-sampling algorithm interpolate the information provided by the nearest pixels using a cubic function. Given a pixel, the interpolation function evaluates the 4 pixels around it applying a filter given by the equation:

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

where x identify each pixel below the filter. Commonly 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[39]. Despite this function was also implemented in the most common library in Python we provide an efficient multi-threading implementation in the Byron library.

Equivalent performances could be achieved using a generalized version of the bicubic filter which use the 8 positions mask around each pixel, the so called Lanczos filter. Also this function was provided into the Byron library.

To better understand the told above functions we can consider their application on the simple image given in Fig.1.11.

[39] In this case the filter is also called Catmull-Rom filter.

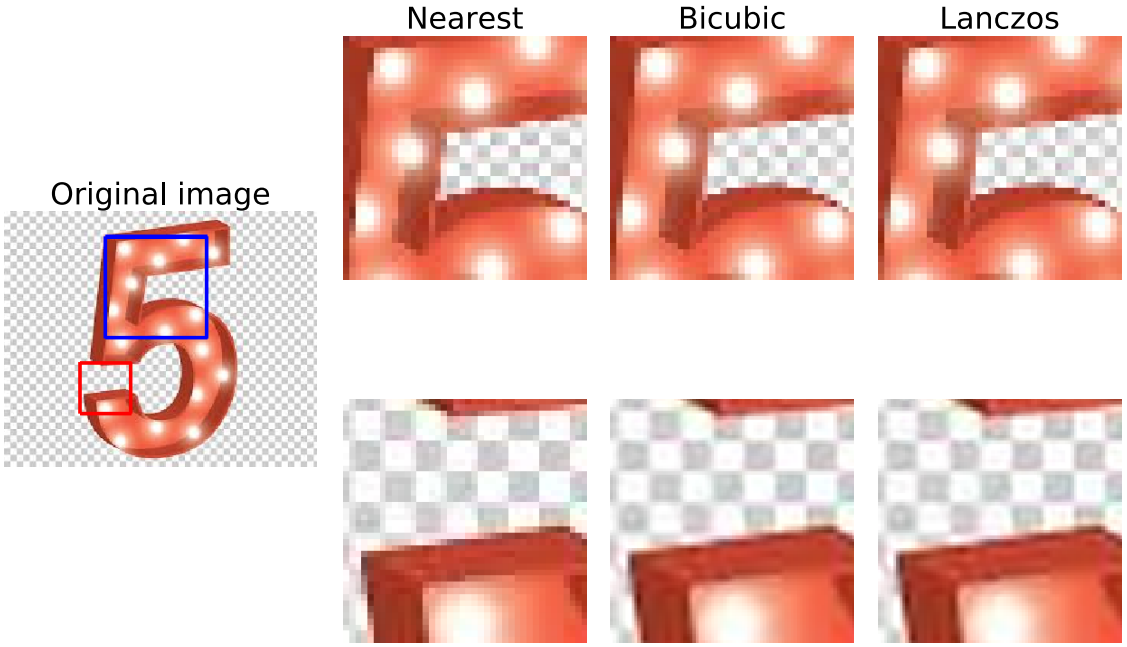

Figure 1.11: Re-sampling image example. **left**) The original image. **up right**) The down-sampled blue-ROIs using different interpolation algorithm (Nearest, Bicubic and Lanczos, respectively). We use a scale factor equal to 2 (half size in down-sampling and double size in up-sampling). The Lanczos interpolation is the lossless algorithm but from a qualitative point-of-view the result is the quite the same of the bicubic one. **down right**) The up-sampled red-ROIs using the same interpolation algorithm of the upper row. Also in the Up-sampling the Lanczos and bicubic algorithms produces equal qualitative results. The Nearest algorithm produces the worse results in both up- down- cases.

In the figure the three algorithms were applied over the same image to highlight the differences against the down-sampling and up-sampling. The nearest interpolation algorithm produces always the worse results both in up-sampling and down-sampling. In the bicubic and Lanczos down-sampling we can better appreciate the "preservation" of the line shapes that are lost using the Nearest algorithm. The result obtained by bicubic and Lanczos are quite similar in both cases but the computational cost of the Lanczos algorithm is greater than the bicubic one. This is the reason why the bicubic interpolation is the most used technique for image resizing with a balance between computational cost and qualitative results. In our implementation of SR algorithms we chose to use the bicubic interpolation for those reasons.

The aim of SR algorithm is to overcome these results and obtain a better quality image either from an optical point-of-view either from a mathematical one. Until now we are considering the quality of the digital image only from a qualitative point-of-view. In the next section we will introduce some useful mathematical scoring to numerically evaluate the image quality.

1.2.2 Image Quality

The most common image quality evaluator is given by our eyes. This is true also for SR problems: the final purpose still remain to obtain images that are better visible for human eyes, the so called *visual loss*. We can however provide some mathematical formulas which allows to quantitative evaluate the image quality. In both cases we need to establish a relation between the original image and the produced one. Thus we can formulate a quality score only with a reference image. In SR problems, or more in general in up-sampling problems, we can compare the original HR image with the image obtained by the output of our model. In this way our quality score will be a measure of similarity between the two images.

The simple similarity score can be obtained evaluating the peak-signal-to-noise-ratio (PSNR). This quantity is commonly used to establish the compression lossless of an image and it can be computed as

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

where $\max(I)$ is the maximum value which can be taken by a pixel in the image (in general it will be 1 or 255 depending on the image format chosen) and MSE is the Mean Square Error (ref. 1.1.10) between the original image and the reconstructed one. The MSE for an image can be computed as:

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

where W, H are width and height of the two images and I, K are the original and reconstructed images, respectively.

In other words the PSNR is the maximum power of the signal over the background noise. It is expressed in decibel (dB) because the image values ranging in a wide interval and the logarithmic function rearrange the domain. Thus we can conclude that high PSNR values are associated to a good reconstruction of the original image.

The PSNR is probably the most common quality score [15] but it does not always related to a qualitative visual quality. Despite it is commonly used as loss function for SR models.

	Nearest	Bicubic	Lanczos
PSNR	25.118	27.254	26.566
SSIM	0.847	0.894	0.871

Table 1.2: Image quality scores: PSNR (peak-signal-to-noise-ratio) and SSIM (Structural SIMilarity index). The values are computed on the image shown in Fig. 1.11. The original image was down-sampled using a Lanczos algorithm and then re-up-sampled using three different algorithms: nearest, bicubic and Lanczos interpolations. For each interpolation algorithm the PSNR and SSIM was evaluated. As expected the highest scores were obtained using the bicubic algorithm while the worst reconstruction is performed by the nearest algorithm.

Considering the series of images shown in Fig. 1.11 we can evaluate the PSNR score starting from a down-sampled image. Taking the down-sampled image obtained with the Lanczos algorithm we can compare the original image with their up-sampled version given by the three methods (ref. Tab. 1.2). As expected, the lowest PSNR value is achieved by the nearest interpolation method while the best performances are obtained by the bicubic algorithm. This confirm the wider use of bicubic method in image processing applications. Moreover we have to take in account that an increment of 0.25 in PSNR value correspond to a visible improvement for human eyes.

A more advanced quality score, commonly used in super resolution image evaluation, is given by the *Structural SIMilarity index* (SSIM). The SSIM aims to mathematically evaluate the structural similarity between two images taking into account also the visible improvement seen by human eyes. The SSIM function can be expressed as

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

where N is the number of arbitrary patches which divide the image[40] For each patch the SSIM is computed as

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{({\mu_x}^2 + {\mu_y}^2 + c_1)({\sigma_x}^2 + {\sigma_y}^2 + c_2)}$$

where μ and σ are the means and variances of the images, respectively, and σ_{xy} represents the covariance. The c_1 and c_2 parameters are fixed to avoid mathematical divergence. Also in this case higher value of SSIM corresponds to high similarity between the original image and the reconstructed one.

Based on the previous equation we can highlight a link with the pooling functions discussed in 1.1.5. Also in this case, in fact, we works with a window/kernel moved along the image which applies a mathematical function on the underlying pixels. This equivalence suggests an easy implementation of this method with slight modifications of the previous code.

The evaluation of SSIM quality score on the previous up-sampled images (ref. Fig. 1.11 and Tab. 1.2) confirms the results obtained by the PSNR. Also in this case the worst reconstruction is obtained by the nearest algorithm while the highest SSIM is obtained by the bicubic algorithm. The gap between SSIM values is smaller than PSNR ones but this is due to the different domains of the two functions.

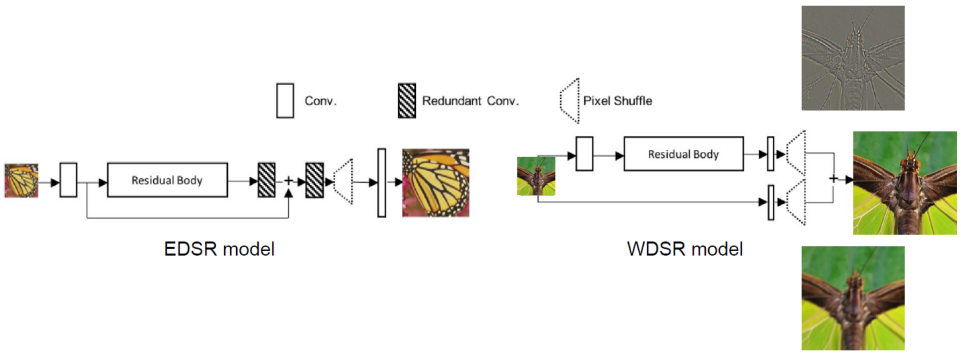

Figure 1.12: Super Resolution models analyzed in this word. **(left)** EDSR model. The model is a modified version of the ResNet architecture designed for SISR applications. The architecture is made by a sequential CNN framework which processes the input image. The EDSR has more than 43 billion of parameters in total. **(right)** WDSR model. The model is the updated version of the EDSR one. The model optimizes its numerical efficiency using a different approach in the analysis of low- and high-frequency components in the input image. the WDSR has slight more than 3.5 billion of parameters, less than 10% of the EDSR model.

1.2.3 Super Resolution Models

There were different kind of models proposed for image Super Resolution purposes but in this work we focused only on two of them. Both are based on deep learning Neural Network models and they became famous in the research community since they both won the last NTIRE editions, 2017 and 2018 respectively.

Layer	Channels input/output	Filter dimensions	Number of Parameters
Conv. input	3/256	3×3	6912
Conv. (residual block)	256/256	3×3	589824
conv. (pre-shuffle)	256/256	3×3	589824
Conv. (upsample block)	256/1024	3×3	2359296
Conv. output	256/3	3×3	6912

Table 1.3: EDSR model scheme summary. We highlight the number of parameters of each macro-block. The total number of parameters of this model is given by the sum of the values in the last column (more than 3 million of parameters).

The first model is called EDSR (*Enhanced Deep Super Resolution*) and was firstly proposed at the NTIRE challenge in 2017 []. The EDSR model structure could be broadly summarized as an updated version of the SRResNet model which is already a modified version of the classical ResNet (standard CNN based on multiple residual blocks). The major updates concern a series of optimization to improve the training speed and the quality of the output image. In particular, the batch normalization steps are removed to improve the algorithm speed: it was proofed that in low-level vision tasks as the super resolution one, i.e without complex evaluations as object detection, a wide and dynamic range of outputs can be useful []. A scheme of the EDSR architecture is shown in Fig. 1.12(a) and the full set of parameters are reported in Tab. 1.3: the EDSR model has more than 43 billion of parameters in total.

[40] Patch dimensions commonly used are 11×11 or 8×8.

A first convolutional layer takes the LR image which is processed using 256 filters. Then a set of 32 residual blocks (convolution with 256 filters + ReLU activation + convolution with 256 filters + linear combination of the output with the input) process the feature map. The tail of the architecture is made by an up-sample block which re-organize the pixels using a series of convolution and pixel-shuffle functions. The up-sampling follows the scale factor imposed: the model increases the spatial resolution of the image by a fixed scale factor (x2 and x4 in our applications) and each pixel-shuffle application is equivalent to a x2 in the output sizes[41].

The first convolutional layer extracts the low frequency components of the input image which will be combined to the output of the residual blocks at the end of the model. The residual blocks with their relative convolutional layers extract the feature map and the high frequency informations into the LR image: in this way the low- and high-frequency components are "independently" analyzed by the model and then re-combined in the output. The last set of up-sampling blocks simply reshape and reorganize the extracted informations according to the desired sizes.

The large amount of filters of the up-sampling blocks and the input dimensions drastically affect the computational performances of the model: we numerically evaluated that the most time spent by the processing is related to the tail of the model and thus to the up-sampling blocks.

The second analyzed and implemented model is the WDSR (*Wide Deep Super Resolution*) model which won the NTIRE challenge in 2018 []. The WDSR model is a modified version of the EDSR one. The improvements principally concern two aspects: the network structure and the residual blocks.

As shown in Fig. 1.12(b), the WDSR simplifies the network architecture removing the convolutional layers after the pixel-shuffle ones. Moreover, if the EDSR applies a x2 up-sampling every pixel-shuffle layer, in the WDSR a single pixel-shuffle function performs a x4 up-sampling. This update drastically reduce the computational time and the amount of parameters. Furthermore, the combination between low- and high- frequency components in this case are processed separately (two different branches) and only at the end they are re-combined (ref. Fig. 1.12(b)).

Layer	Channels input/output	Filter dimensions	Number of Parameters
Conv. input 1	3/32	3×3	864
Conv. 1 (residual block)	32/192	3×3	55296
conv. 2 (residual block)	192/32	3×3	55296
Conv. (pre-shuffle)	32/48	3×3	13824
Conv. input 2 (pre-shuffle)	3/48	5×5	3600

Table 1.4: WDSR model scheme summary. We highlight the number of parameters of each macro-block. The total number of parameters of this model is given by the sum of the values in the last column ($\sim$ 100K parameters, less than 1/10 of EDSR model).

The WDSR also changes the residual block structure: the ReLU activations tends to block the information flow from the first layers [] and in super resolution structures is important to prevent it since they contain the low-frequency components of the image. To overcome this problem without increasing the number of parameters the WDSR proposes the so-called "passage enlargement", i.e the reduction in the number of channels in input and the corresponding enlargement of the output channels before the ReLU activation. This optimization allows to increase the number of channels to be activated and thus a better

[41] It is straightforward that adding multiple up-sampling blocks and thus pixel-shuffle functions we can train the model according to every desired upscale.

information flux along the network keeping the required non-linearity. The number of parameters is however constant because there is only a re-arrangement of the input/output parameters. The list of network parameters are reported in Tab.1.4: the WDSR has slight more than 3.5 billion of parameters, less than 10% of the EDSR model. This confirms the computational efficiency of the WDSR against the EDSR one.

In this work we used pre-trained models so we could not change the network structure or change their learning weights. For this reason we could use only a x2, x4 EDSR model and a x4 WDSR model. The weights were converted to the Byron format and our custom implementation of the network used for the applications. We would stress that our could be the first C++ implementation of these models and probably the first optimized version for CPUs environment[42].

1.2.4 DIV2K dataset

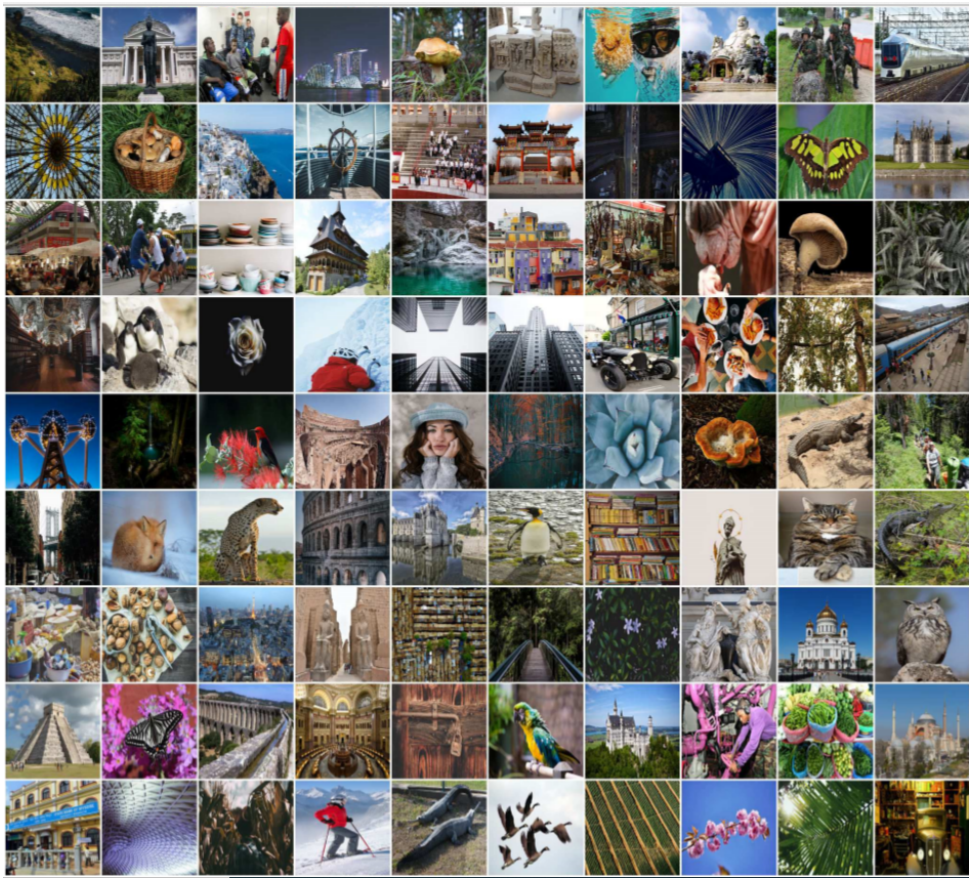

Figure 1.13: DIV2K validation set examples.

In our super resolution applications we used as training set the images provided by the DIV2K (*DIVerse 2K resolution high quality images*) dataset [2]. This dataset was appositely created for the 2017 NTIRE challenge (*New Trends in Image Restoration and Enhancement*). The NTIRE challenge is an international competition which aims to monitoring the state-of-art in digital image processing and image analysis and it takes place

[42] We have to mention also that the public available implementation of these models are developed only in Tensorflow and PyTorch but the major part of them does not work in CPU environments without heavy modifications.

at the CVPR (*Computer Vision and Pattern Recognition*) conference every year. One of the most important monitored task is the super resolution research progress. Thus, every year, many research groups propose new super resolution models, mostly based on neural network models, to improve the state-of-art results on this research field. The challenge is won by the model which performs the higher PSNR value over a validation set extracted on the DIV2K dataset. For these reason the DIV2K dataset is considered as a standard for super resolution applications.

The dataset contains 800 high-resolution images as training set and their corresponding low-resolution ones, obtained by different down-sampling methods and different scale factors (2, 3, and 4). A second set of 100 high-resolution images makes the test set on which the model can evaluate its accuracy: also this second set of images have their low-resolution counterpart. Finally, a third group of 100 images constitutes the validation set, i.e they are blinded images without their corresponding high resolution counterpart, and they are used to evaluate the results of the models in race.

All the 1000 images are 2K resolution, i.e width and height dimensions must have at least 2K pixels. The images are collected paying particular attention to the quality, diversity of sources (web sites and cameras) and contents. The DIV2K images, in fact, collect a large diversity of contents, ranging from people, handmade objects and environments (cities, villages) to natural sceneries (including underwater and dim light conditions) and flora and fauna. In each image we can find more or less complex shapes, geometries and also some words. We would stress that no one bio-medical image is contented in the dataset since it is very difficult obtain high quality images of this kind (let alone the problems about copyrights and releases).

In our SR applications we used pre-trained[43] neural network models on the DIV2K and we tested their performances over NMR (Nuclear Magnetic Resonance) images. The models have never seen this kind of images but during the training they learned a large quantity of shapes that can be "found" also in bio-medical images. The bio-medical images were provided by the collaboration with the MRPM group of the Physics Department of the University of Bologna and the Bellaria hospital of Bologna. We thank the volunteers who perform the NMR acquisitions and shared their data.

1.2.5 Results

1.3 Object Detection

Object detection is one of the larger deep learning sub-discipline, especially when we talk about Neural Network models. This kind of problems aim to identify single or multiple objects into a picture or video stream. The possible applications of these tools are everywhere these days and they involve object tracking, video surveillance, pedestrian detection, anomaly detection, people counting, self-driving cars or face detection, the list goes on.

There are many machine learning and deep learning techniques and algorithms proposed during the years and each one has its pros and cons. The most prominent and moder techniques involves the use of very deep Neural Network models with a huge amount of parameters to tune. The most famous one are probably the Faster R-CNN (*Faster Region Convolutional Neural Network*) [25] and their "evolution" given by the YOLO (*You Only Look Once*) model [22, 23, 24].

The R-CNN models are one of the state-of-art CNN-based deep learning object detection model and their evolution into Fast R-CNN tries to improve the speed on object detection. The standard approach for object detection is based on moving a *sliding win-*

[43] The developed models were not re-trained due to limited time and low computational architectures available.

dow to search in every position of the image the looking for objects. However, the intrinsic problem of these kind of approach is in the dimension of the window and in the large computation required to map with multiple window sizes the full image. Moreover, different objects or even the same kind of objects could have different aspect ratios and sizes in relation to the position of the camera which captured the image or to their distances. R-CNN models try to overcome these problems generating about 2k region proposals, i.e bounding boxes, and applying to each one a image classification using standard CNN. Finally, each detected region can be refined using a regression approach.

A Faster R-CNN model is based on the same idea but, instead of feeding the bounding boxes to the CNN, it feeds the input image to the CNN to generate a convolutional feature map. Starting from this feature map we can easier identify the region of proposals (Region Proposal Network) and warp them into squares. The list of these regions are then reshaped using a Polling layer and processed by a fully connected layer. The advantages of Faster R-CNN are thus visible: we do not need to feed 2k region proposals to the CNN every time but the feature map is generate once per image using the convolution operation. In this way we can also separate the feature map creation to the selective search algorithm.

A key role is played by the *anchor* concept: an *anchor* is essentially a box and it identify the shape of a portion of the input image at different scale level. The CNN feature map feeds the Region Proposals Network which uses a sliding window over it generating k anchor boxes. These boxes are certainly fewer than the 2k previous cited windows.

A breakthrough idea on the real-time object detection was the introduction of the YOLO model. The model was developed by Redmon et al. at Washington University and it is probably the state-of-art on object detection, especially for its very incredible speed (it can reach 45 FPS on modern GPUs!). Certainly it is the faster method public available but its popularity is due also to its innovative strategy in object detection. Despite all the other algorithms use regions to localize the object into the image, the YOLO network does not look at the complete image but only on a parts of it which has the higher probability to contain an object. In YOLO a single CNN predicts the bounding boxes and the class probabilities of them. YOLO slit a single image into a $S \times S$ grid and on each grid m bounding boxes are taken. For each of them, the CNN outputs a class probability and offset values. Finally these bounding boxes are filtered according to their probability and a chosen threshold.

One of the most bigger limitation of this model is that it struggles with small objects. This is due to the spatial constraints of the algorithm. Fortunately, in the previous section we have already discussed on how we can overcome this kind of problem using Super Resolution. In the next section we will discuss about further characteristics of the YOLO model and about its implementation into the Byron library and its efficiency against the original implementation. Finally we will join the efficiency of the previous Super Resolution models to the performances of our custom implementation of YOLO.

1.3.1 Yolo architecture

The YOLO Neural Network architecture was firstly published in the 2015 but from the first version many improvements were performed and now we have the third revision of it. We do not want to recall the history of this model so we will discuss only about the YOLOv3 model (for sake of simplicity we will call it just YOLO).

YOLO is a deep Neural Network model with more than 100 layers and more than 62 billion of parameters. The first versions of YOLO are based on a Darknet-19 architecture (19-layer network followed by 11 more layers for object detection). In the last release of YOLO model the first part of the network structure is used for the feature map extraction

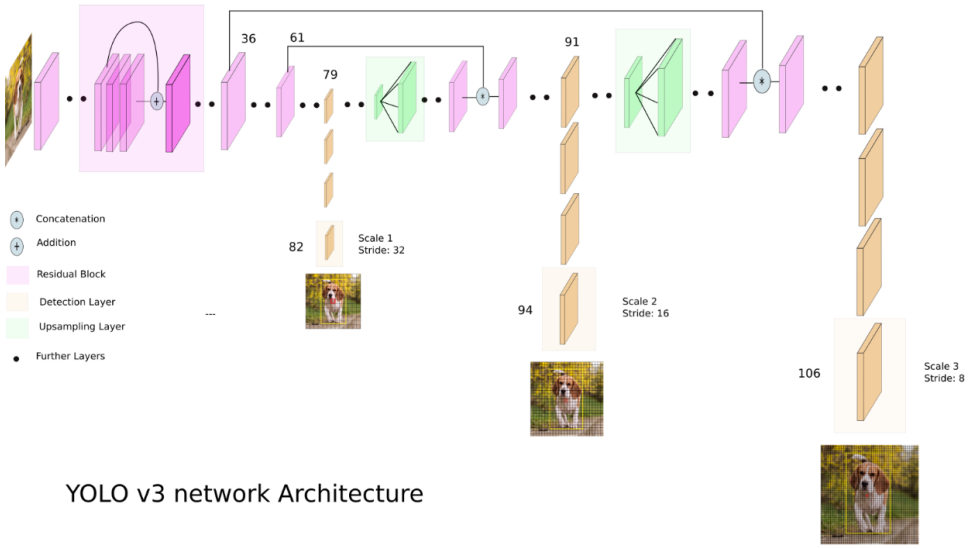

Figure 1.14: Yolo Neural Network scheme.

and it is essentially a modified version of the Darknet-53 model, i.e the update version of the previous model, with more layers and parameters. This improvements increase the classification performances but it throwbacks a reduction in computational performances[44]. This improvement could be done also thanks to the introduction of multiple residual blocks which, as discussed in the previous sections (ref. 1.1.8) allows to increase the deep of the model without losing performances.

YOLO performs the object detection using a multi-scale approach: three different scales were taken into account during the training section and it increases the classification performances of the model. The network structure can be broadly summarize as a simple CNN and its output is generated by applying a series of three different detection kernel 1×1 kernel on the feature map. Moreover, this detection was performed in three different places in the network, i.e three YOLO detection layers are distributed along the network structure. The shape of the detection kernel is $1 \times 1 \times (B \times (5 + C))$, where B is the number of bounding boxes a cell on the feature map can predict and C is the number of classes. The fixed number ("5") is given by 4 bounding box attributes plus one object confidence coefficient (the so-called *objectness* into the code). In our applications we used the COCO dataset (see next sections, **??**) and thus we fixed the values of B and C to 3 and 80, respectively (thus the kernel size is equal to $1 \times 1 \times 255$). We would stress that the three scale detections are equivalent to three level of down-sampling of the original image (or better the feature map), respectively equal to 32, 16 and 8.

The input image is down sampled using the first 81 layer and only the 82nd layer performs the first detection[45]. Then the feature map produced by the 79th layer is subjected to a few convolutional layers before being up sampled by 2x to a 26×26. The up-sampling is performed by a previously discussed Upsample function (ref. **??**). The feature map is then concatenated with the one produced by the 61st layer and processed by a second series of convolutions until the 94th layer performs the second detection. A third (similar) procedure is performed again until the end of the architecture (106th layer) where the final $52 \times 52 \times 255$ feature map is produced as output. The first detection layer is responsible for detecting larger objects while the second two analyzes smaller regions: a comparative analysis of these three different scale results improves the detection performances and help to filter false positive cases.

[44] For the record, the older YOLO version are faster than the last release but less accurate.
[45] Considering an input image of size 416×416 the resulting feature map would be of size 13×13.

The introduction of three different detection layers improves the issues of detection small objects in comparison to the previous versions but it remains a crucial limit of the model. Moreover, the up-sampling layers connected with the previous layers (shortcut) help to preserve the fine grained features and thus the identification of small objects into the image.

The model uses a total of 9 anchor boxes with three scale per each. The anchors have to be computed before the training phase on the training dataset: the author suggests to use a K-Means clustering for this purpose. The first three anchors will be associated to the first (larger scale) detection layer and so on along all the structure. Taking into account an image of 416×416 as example, the number of predicted boxes will be 10'647 (which is 10x the number of boxes predicted by the previous version of the model).

A further innovative improvement was given by the loss function used to train the model. The loss computation for true positive identification has to take into account that multiple bounding boxes per grid cell are performed and thus we have to filter them. In other words we want to preserve only the bounding boxes "responsible" for the object. This can be achieved using the highest IoU (*Intersection Over Union*) with the ground truth. YOLO uses a modified MSE error between the predictions and the ground truth. In particular the loss function is composed by three terms: the classification loss, the localization loss and the confidence loss.

The classification loss quantify the error of detection and it is given by

$$\mathcal{L}_1 = \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

where $\mathbb{1}_i^{\text{obj}}$ is equal to 1 if an object appears in cell i, $p_i(c)$ is the output of the model and $\hat{p}_i(c)$ denotes the conditional class probability for class c in cell i.

The localization loss measures the errors in the predicted boundary box locations and sizes: in this way we can filter only the boxes responsible for detecting the object.

$$\mathcal{L}_2 = \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_i^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2\right] + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_i^{\text{obj}} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2\right]$$

where $\mathbb{1}_i^{\text{obj}}$ is equal to 1 if jth boundary box in cell i is responsible for detecting the object, λ_{coord} increase the weight for the loss in the boundary box coordinates[46] and (x, y, w, h) are the boundary box coordinates.

The confidence loss quantifies if an object is detected into the founded box (*objecteness*), i.e

$$\mathcal{L}_2 = \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_i^{\text{obj}} \left(C_i - \hat{C}_i\right)^2$$

where $\hat{C}_i$ is the box confidence score of the box j in cell i. If the object is not detected into the box, the confidence loss is computed as:

$$\mathcal{L}_2 = \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_i^{\text{obj}} \left(C_i - \hat{C}_i\right)^2$$

[46] The default value used in the model is 5.

where λ_{noobj} weights down the loss when detecting background (most boxes do not contain any objects and in the training images a large amount of pixels are occupied by background)[47].

The final loss is given by the sum of these three contributions

$$\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3$$

To further improve the detection performances we have to remove duplicate detections. This is performed by YOLO model applying a non-maximal suppression to remove duplicates with lower confidence. Thus, the method sorts the predictions according to the confidence scores and starting from the top scorer it filters the predictions with the same class and a IoU score greater than a given threshold. In this way we tune the bounding boxes to be as much fit as possible to the object shape.

1.3.2 COCO dataset

Figure 1.15: COCO validation set examples.

The first issue to take into account when we want to train an object detection model is certainly to provide a good training set. The dataset has to include multiple and different prospective of the searching object and all these images has to be manually annotated (ground truth for a supervised learning). To train a robust classifier, we need to provide a lot of pictures to our model since the model has a lot of parameters to be tune. So the training samples should have different backgrounds, random object and varying lighting conditions. The set of training images could not be made by high quality images but the most important required features is certainly the heterogeneity of data.

During a training section we have also to take in count that a part of the available data has to "discard" and used as test set so the number of sample has to be sufficient for both steps. The YOLO model has more than 62 billion of parameters to be tuned and a sufficient number of annotated samples to train it is hard to produce. Fortunately, there are different public available datasets designed to face on object detection training problem. One of the most popular one is the COCO dataset.

COCO dataset is a large-scale open source dataset designed for multiple deep learning training tasks. In particular we can find a large number of images manually annotated useful for object detection, segmentation and captioning. The dataset is continually updated and quite every year a new version is released.

[47] The default value used in the model is 0.5.

The intrinsic limitation of the dataset is given by the available classes: COCO includes 80 different object classes concerning general purpose objects, starting from different animals to everyday objects and transports. This limits the possible applications but it remains a very useful tool for testing new models[48]. The dataset includes more than 300k images in which more than 200k are already labeled. Certainly the unlabeled ones could be used as test set for a visual estimation of performances[49].

In our applications we were focused on people detection and this category is already included into the available ones so we considered the COCO dataset an optimal solution for our purposes.

The YOLO network was training on these images using different scale dimensions: the images are fed to the network with sizes ranging from 320×320 to 608×608 with increments of 32[50]. This variability helps the sensibility of network (convolutional) filters to the details of the image. Moreover, it helps the detection to identify the object at different scale levels. We would stress that it does not put a limit into the input dimensions since the filter weights are independent to them. However, our tests highlight that the best results are obtained rescaling the image to 608×608.

The original implementation of the YOLO model (provided by Redmon J. in his webpage) provides a pre-trained version of the model to the COCO dataset. For our applications we do not re-trained the model[51], but we converted the available weights to the Byron format.

1.3.3 Results

1.4 Image Segmentation

In the previous section we have discussed about the object classification and object detection problems (ref. 1.3). Now we want to go deeper on this topic and extract the

[48] COCO dataset is considered as a sort of standard in object detection applications and every new proposed model provides its performances against it.

[49] The object detection problem is a considered an hard task for computer vision application but it is a straightforward task for human eyes.

[50] The increment value chosen is exact the down-sampling factor performed by the architecture.

[51] The training of YOLO model requires a lot of time and computational resources. All this work of thesis was performed using a cluster machine shared among many users and thus it was impossible to dedicate the full computational resources to a single application.

exact pixels which belong to an object into a given image. This kind of problem is called Image Segmentation, i.e give a label to each pixel of the input image.

Image segmentation is a typical task in many research fields and could be used for different purposes. Informations about pixel-wise position of objects inside an image could be used for extract object shapes from the image or to simplify and/or change the representation of an image into something more meaningful and easier to understand. This is an hot topic especially for self-driving car applications in which we have to find the exact shapes of object to better estimate their perspective position. Moreover, all these applications require fast algorithm as much as possible closed to real-time.

This kind of task can be performed using a pipeline of image processing functions or by training a neural network model. In the first case we have to stack a series of function to process the input image: it has to filters and extracts the useful informations about the searched object but most of all it has to be as most general as possible to face on the common heterogeneity of samples. In the second case we leave to the neural network model parameters the searching of optimal combination of function but we have to provide a supervised input pattern, i.e a combination of input and annotated pixel-wise mask of each image. The image annotation is one of the most hardest and boring step of image segmentation and for these reasons is very hard to find public dataset usable.

In this chapter we introduce a particular neural network model commonly used in image segmentation problems and we will describe its characteristics and performances. We applied this model to a novel dataset of CT images. The dataset annotation was performed by a custom semi-supervised pipeline of image processing and the neural network model was trained and tested on this dataset. The original data are taken from here and the corresponding annotations are released on here.

1.4.1 U-Net model

U-Net neural network model is one of the state-of-art model in image segmentation. It was firstly developed for biomedical image segmentation but it shew its efficiency also in different application tasks and different research topics. Its backbone is intrinsically a "common" CNN but the structure can be divided into two macro paths. The first path of the model is a contraction path (or *encoder*) while the second path is an expansion path (or *decoder*). The first set of layers in the model, in fact, are a sequence of convolutional and pooling layers which aim to extract features and reduce the dimensionality of the input in the same way as an encoder convert a signal to a smaller range of values. The extracted features are then processed by the decoder, i.e a second set of convolutional and up-sampling layers, to reconstruct the feature map size and the segmentation mask. An illustrative representation of the model structure is provided in Fig. 1.16.

We have already discussed about the functionality of each layers in the previous sections and also in this kind of model a key role was performed by the shortcut connections. The decoding path tends to lose some of the higher level features the encoder learned: using shortcut connections the output of the encoding layers are passed directly to the decoding layers so that all the important pieces of information can be preserved.

In the previous sections we have also described the common loss functions used to train Neural Network models. Considering the "simple" segmentation of an object from its background, the ground truth mask, i.e the "label" of the input image, would be a binary matrix. In these cases a valid loss function (also used in our applications) could be the *binary cross-entropy* (ref. 1.1.10).

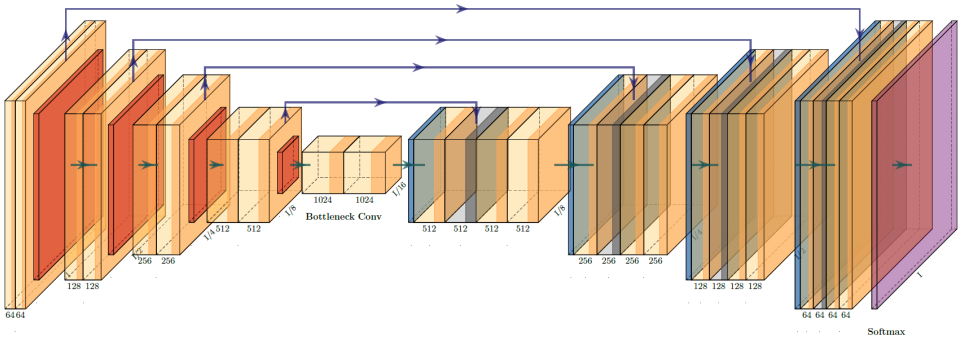

Figure 1.16: U-Net model scheme. The first part of the structure represents the encoder while the tail of the model is the decoder part. The model name is given by the numerous shortcut connections which link the encoder layers to the decoder ones: if we contract the long-range connections the global structure acquire a U form. The figure was generated using the PlotNeuralNet package of H. Iqbal.

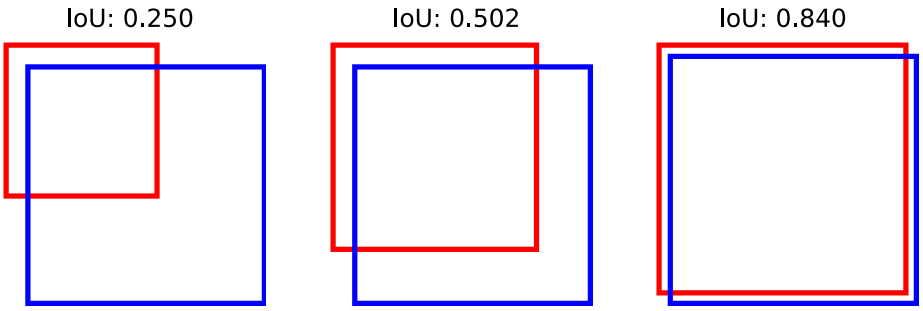

Figure 1.17: IoU score example. The IoU score is computed as the area intersection of the two boxes over their union. Starting from the left we can see an increment of the overlap between the two boxes related to an increment in their IoU scores.

A word of caution must be spent about the metrics to evaluate the performances of our model. Standard metrics, as the *accuracy*[52], are not good measures to face on the segmentation problem. If we want to identify and segment an object into an image we can reasonably assume that the number of pixels concerning the object would be very few against the number of pixels related to the background. Thus the told above binary mask would be a matrix with a large amount of zeros and only few ones. In this case the standard metric functions have to consider an unbalanced number of samples: if the model outputs a matrix of all zeros the accuracy of it will be high despite the informative values are only the few pixel equals to one. A possible solution to overcome this problem is given by the *mean IoU score* which measures the average IoU between the output mask and the binary ground truth:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

The efficiency and meaning of this score can be visible in Fig. 1.17.

1.4.2 Femur CT Dataset

1.4.3 Results

1.5 Replicated Focusing Belief Propagation

Until now we are talking about neural networks based on the standard update rule of backpropagation. Other learning rule for weight updates were proposed and the choice of the best one it is a still open-problem. The final purpose is to obtain a feasible learning rule able to model the biological learning of the human brain.

The learning problem could be faced on through statistical mechanic models joined with the so called Large Deviation Theory. In general the learning problem can be split in two sub-parts: the classification problem and the generalization one. The first aims to completely store a pattern sample, i.e a prior known ensemble of input-output associations (*perfect learning*). The second one corresponds to compute a discriminant function based on a set of features of the input which guarantees a unique association of a pattern.

From a statistical point-of-view many Neural Network models have been proposed and the most promising seem to be models based on spin-glasses. Starting from a balanced distribution of the system, generally based on Boltzmann distribution, and under proper conditions, we can proof that the classification problem became a NP-complete computational problem. A wide range of heuristic solution to that type of problem were proposed.

In this section we show one these algorithms developed by Zecchina et al. [4] and called *Replicated Focusing Belief Propagation* (rFBP). The theoretical background of the algorithm is beyond the scope of this thesis so we focus on its numerical implementation and optimization.

Moreover, despite their proofed theoretical efficiency, the applications on real data are still few. Thus we show the application of the optimized version of the rFBP algorithm on the Genome Wide Association (GWA) data provided by the European COMPARE project. This work was also presented on the CCS-Italy (Conference of Complex System) of the 2019 [9].

1.5.1 Algorithm Optimization

The rFBP algorithm is a learning algorithm model developed to justify the learning process of a binary neural network framework. The model is based on a spin-glass distribution of

[52] The accuracy measures the number of true positives + false negatives outputs on the total number of predictions.

neurons put on a fully connected neural network architecture. In this way each neuron is identified by a spin and so only binary weights (-1 and 1) can be assumed by each entry. The learning rule which controls the weight updates is given by the Belief Propagation method.

A first implementation of the algorithm was proposed in the original paper [4] jointly with an open-source Github repository. The original version was written in Julia language and despite it is a quite efficient implementation the Julia programming language stays on difficult and far from many users. To broaden the scope and use of the method a C++ implementation was developed with a jointly *Cython* wrap for Python users. The C++ language guarantees also better computational performances against the Julia implementation. This implementation is optimized for parallel computing and is endowed with a newly written C++ library called *Scorer* (see Appendix D for further details), which is able to compute a large number of statistical measurements based on a hierarchical graph scheme. With this optimized implementation we believe we can encourage researchers to approach these alternative algorithms and to use them more frequently in real context.

Like the Julia implementation also the C++ one provides the entire rFBP framework in a single library callable via a command line interface. The library widely uses template method to perform dynamic specialization of the methods between two magnetization version of the algorithm. The main categories of objects needed by the algorithm are wrapped in handy C++ objects easy to use also from the Python interface. A further optimization is given by the reduction of the number of available functions: in the original implementation a large amount of small functions are used to perform a single complex computation step enlarging the amount of call stack; in the C++ implementation the main functions are re-written with the minimum quantity of functions to ease the vectorization of the code.

The full rFBP library is released under MIT license and it is open-source on Github [8]. The on-line repository provides also a full list of installation instructions which could be performed via *CMake* or *Makefile*. The continuous integration of the project is guaranteed in every operative system using *Travis CI* and *Appveyor CI* which test more than 15 different C++ compilers and environments.

To encourage the Machine Learning community in the use of this kind of methods we provide a Python version based on a *Cython* wrap of the C++ objects. This wrap guarantees also a good integration with the other common Machine Learning tools provided in the *scikit-learn* Python package; in this way we can use the rFBP algorithm as equivalent in other pipelines. Like other Machine Learning algorithm also the rFBP one depends on many parameters, i.e its hyper-parameters, which has to be tuned according to the given problem. The Python wrap of the library was written also according the *scikit-optimize* Python package to allow an easy hyper-parameters optimization using the already implemented classical methods.

1.5.2 SNP classification

The few available applications of the rFBP algorithm to real data are amenable to two aspects: I) learning technique; II) algorithm implementation. The first one is related to the intrinsic definition of the algorithm which is designed to reach a complete memorization of the training dataset; in the other Machine Learning processes we normally want to avoid this kind of results since it could bring to *over-fitting* problems. The second one is given by the binary values involved in each step of the algorithm which intrinsically limit the possible applications[53].

[53] The Neural Network weights can assume only binary values since they model up/down spins. Moreover also the input is required to be a spin configuration and thus binary. The common Machine Learning problems involve floating-point values as input pattern.

Classification problems which involved only binary quantities are quite small but the GWA is one of them. In the GWA we have a series of genome data belonging to different classes as input. A genome is the ensemble of genes of an organism and each gene is identified by a series of nucleotides with 4 possible values (G, guanine; C, cytosine; A, adenine; T, thymine). The comparison between a reference (healthy) genome and an infected one highlights the biological mutation related to the understudy disease. This mutation are the so-called SNPs (Single Nucleotide Polymorphisms). So we can identify a genome as a sequence of its mutation in relation to a reference genome, i.e a sequence of two possible values given by the on/off of the mutation in each nucleotide.

The COMPARE project aims to develop new methods to avoid the genetic disease transmission. In this project plays a crucial role the *Source Attribution*, i.e the classification of a given disease based on the list of its mutation.

We tested the rfBP on 210 Salmonella enterica genome sequences, $4857450\,bp$ (base pairs) long, living inside animals. Our early goal was to discriminate those bacteria living in pigs (159 samples) with respect to all the others animals (51 samples).

First of all we filter our data removing from each genome a base if it is not mutated in each sample. In this way we reduce the number of bases to $8189\,bp$. A graphical representation of these samples is given in Fig. **??**. The dataset was divided in training and test sets using a stratified cross-validation procedure to guarantee a proportional subdivision of the samples into the two classes. The algorithm hyper-parameters was tuned on the training set based on the performances obtained using a internal stratified 10-fold cross-validation: in each fold the training was performed by a given sequence of hyper-parameters and the performances evaluated on the corresponding test set; the hyper-parameters configuration which obtains the best performances on the full training set was chosen as best configuration. The performances evaluation was performed using the custom *Scorer* library. Considering the unbalanced sample quantities the Matthews Correlation Coefficient (MCC) is chosen as good scorer indicator for the evaluation.

With the tuned hyper-parameters we performed the training of rFBP algorithm on different percentage of the training set: 25%, 45%, 65% and 85%. In the same way we train also a list of the most common Machine Learning classifiers: single perceptron with floating-point weights (Perc); standard Neural Network with gradient descent as updating rule (MLP); support vector machine with linear kernel (lSVM); support vector machine with radial kernel (rSVM); linear discriminant analysis (LDA); decision tree (DT); random forest (RF); k-nearest neighbors with 2-clusters (kNN); Guassian process (GP); diag-quadratic discriminant analysis (GNB); Bernoulli naive bayes (BNB); AdaBoost (AdaB). For each training percentage we perform the optimization of the hyper-parameters of each classifier with the same number of optimization steps. In Fig. **??** the accuracies and MCC results are shown, respectively.

From this analysis we can conclude that the rFBP algorithm shows comparable performances with the other classifiers. These performances globally grow with the training set size but only the rFBP is able to reach a "perfect learning" configuration, i.e accuracy of 100% and MCC=1. We have also noticed that the rFBP classifier and the GNB are the only two algorithms which qualitatively does not show performances saturation on their training.

A second analysis was performed on the data distribution using a multiple χ^2-test. Starting from the whole set of genomes we can compute the contingency-matrix of the two classes[54]. The χ^2-test was performed on the full set of $8189\,bp$ and so the extracted *p-values* were corrected according multiple-tests. Using the Šidák [28] correction method and by the definition of significant threshold of 0.05 we found 1103 significant bases. An

[54] The contingency-matrix displays the (multivariate) frequency distribution of the variables. Each row will count the number of hosts with/without the SNPs. Each column will identify a class.

analogous χ^2-test was performed on the rFBP weights to identify a putative

1.5.3 Results

Bibliography

[1] 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.

[2] 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.

[3] AlexeyAB. darknet. https://github.com/AlexeyAB/darknet, 2019.

[4] C. Baldassi, C. Borgs, J. T. Chayes, A. Ingrosso, C. Lucibello, L. Saglietti, and R. Zecchina. Unreasonable effectiveness of learning neural networks: From accessible states and robust ensembles to basic algorithmic schemes. *Proceedings of the National Academy of Sciences*, 113(48):E7655–E7662, 2016.

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

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

[7] 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.

[8] N. Curti and D. Dall'Olio. Replicated focusing belief propagation. https://github.com/Nico-Curti/rFBP, 2019.

[9] D. Dall'Olio, N. Curti, G. Castellani, A. Bazzani, and D. Remondini. C++ implementation, optimization and application of the focusing belief propagation algorithm, 2019.

[10] C. Dong, C. Change Loy, K. He, and X. Tang. Image super-resolution using deep convolutional networks. *arXiv e-prints*, page arXiv:1501.00092, Dec 2014.

[11] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] 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 *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[14] A. Hodgkin and A. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Bltn Mathcal Biology*, 1990.

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

[16] 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.

[17] 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.

[18] Y. Lecun, L. Bottou, G. Orr, and K.-R. Müller. Efficient backprop. 08 2000.

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

[20] T. Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed <today>].

[21] 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.

[22] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection, 2015.

[23] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger, 2016.

[24] J. Redmon and A. Farhadi. Yolov3: An incremental improvement, 2018.

[25] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2015.

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

[27] 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, Sep 2016.

[28] Z. Sidak. Rectangular confidence regions for the means of multivariate normal distributions. *Journal of the American Statistical Association*, 62(318):626–633, 1967.