Alma Mater Studiorum · Università di Bologna

Dipartimento di Fisica ed Astronomia
Dottorato di Ricerca in Fisica Applicata - 32° ciclo

Implementation and optimization of algorithms in Biomedical Big Data Analytics

Supervisore:

Prof. Daniel Remondini

Co-Supervisori:

Prof. Gastone Castellani
Prof. Armando Bazzani

Presentata da:

Nico Curti

Esame finale anno 2020

"*No one know nothing,*
everyone know something,
but something is nothing to someone,
while
something is important to everybody"

Daudi, Manyara

Abstract

Big Data Analytics poses many challenges to the research community who has to handle several computational problems related to the vast amount of data. An increasing interest involves Biomedical data, aiming to get the so-called "personalized medicine", where therapy plans are designed on the specific genotype and phenotype of an individual patient and algorithm optimization plays a key role to this purpose. In this work we discuss about several topics related to Biomedical Big Data Analytics, with a special attention to numerical issues and algorithmic solutions related to them. We introduce a novel feature selection algorithm tailored on *omics* datasets, proving its efficiency on synthetic and real high-throughput genomic datasets. The proposed algorithm is a supervised signature identification method based on a bottom-up combinatorial approach that exploits the discriminant power of all variable pairs. We tested our algorithm against other state-of-art methods obtaining better or comparable results.

We also implemented and optimized different types of deep learning models, testing their efficiency on biomedical image processing tasks. Three novel frameworks for deep learning neural network models development are discussed and used to describe the numerical improvements proposed on various topics. In the first implementation we optimize two Super Resolution models showing their results on NMR images and proving their efficiency in generalization tasks without a retraining. The second optimization involves a state-of-art Object Detection neural network architecture, obtaining a significant speedup in computational performance. We also highlight how Super Resolution models are able to overcome object detection issues and, therefore, increase detection performances. In the third application we discuss about femur head segmentation problem on CT images: a semi-automatic pipeline for the image annotation is proposed and a deep learning neural network model trained on these images.

The last section of this work involves the implementation of a novel biomedical database obtained by the harmonization of multiple data sources, that provides network-like relationship between biomedical entities. Data related to diseases, symptoms and other biological relates were mined using web-scraping methods, and a novel natural language processing pipeline was designed to maximize the overlap between the different data sources involved in this project. We describe the key steps which lead us to this network-of-networks database and we discuss its potential application in biomedical research.

Contents

Introduction

Big Data: these two words are at the heart of many contemporary researches. Nevertheless, it is yet a loose term and no exhaustive description has been provided. We commonly associate this term to the description of data generated from several machines and used to describe very complex systems. We can find Big Data associated to multiple kinds of modern researches which use this term to highlight the complexity of their projects. The use of Big Data, in fact, is closely related to the Complexity term (intended with its physical meaning) and to the major part of Artificial Intelligence researches, since they seem to be the only way to overcome these problems. As anticipated, it is difficult to find out a satisfactory definition about Big Data and the common sense tends to name them simply as a vast amount of data. However, this is just a broadly description of them and it simplifies too much their usage and power. We can find Big Data in more applications and fields than we usually think and a prominent research field is the Biomedical one.

Biomedical data are growing both in size and breath of possible usages. This growth is driven by the development of newer and cheaper technologies for data acquisition, which enlarge the availability of them. At the same time, also the computational power is increasing and we can take advantage of more efficient and complex algorithms and pipelines for the analysis of a such amount of data. Unfortunately, this second growth is not enough fast to tackle these problems and the development of novel techniques of processing and, moreover, algorithms able to extract informative portions of data is essential in the so-called Big Data Analytics. This is even more true when we talk about Biomedical Big Data, i.e data related to health-care studies, which aim to identify the variable responsible for more or less complex diseases or to give a description of biological processes. In addition to the novel *Next Generation Sequencing* (NGS) technologies related to the analysis of biological structures like DNA and RNA, the so-called *omics* researches involve a large part of the contemporary biomedical researches. The term *omic* data, also in this case, refers to the wide range of biological studies ending in -omics, like *genomics*, *proteomics* or *metabolomics* which aim to describe and quantify biological processes at different scale levels. The analysis of these kinds of data poses many challenges to the research community, especially for the vast amount of variables involved. In general, biological research field is used to analyze only few samples compared to the number of variables involved: this is exactly the opposite behavior of common statistical analyses and, moreover, of physics research. The ability to extract information and reduce the problem dimensionality is crucial to address these problems.

More complex analyses related to high dimensional problems are the image processing ones. Biomedical imaging is another of the most prominent kind of analysis for the development of novel medical treatments. The many differences between acquisition methods and data structures/characteristics for different imaging modalities create a zoo of possible studies and analyses. At the same time, the dimensionality of the involved images requires an adequate computational effort. These characteristics satisfy all the requirements posed by the modern deep learning training. It is not always possible to create an appropriate mathematical model to describe the underlying dataset and in many cases we need to handle more general applications. Standard machine learning methods can not keep up

such requirements and they are giving way to deep learning models. In many applications these models are used as black-boxes and their complexity does not always allow a complete understanding about their learning. Nevertheless, their efficiency is overcoming standard methods in a vast amount of applications and they are the only tools which give the semblance of an artificial intelligence.

All these data and analyses involve multiple scientific researches which, driven by them, are becoming more accurate but, at the same time, also more specialized. With a such heterogeneity of data, we can handle very useful analyses of any biological compound with a payback of a loss about the system complexity and interactions. The absence of a standardized system for sharing biomedical information contributes to the difficulty about merging results provided by different studies. Several European projects about health-care research has been financed aiming to develop an harmonization between biomedical data sources. The principal issues about this topic are related to a non rigid nomenclature of medical keywords and data formats. Relational databases have efficiently driven Big Data research up to now, but the increasing demand of non-trivial connections between different kinds of entries is paving the way to different kinds of approaches and data management. At the same time, also the research about novel natural language processing methods are becoming very popular in these applications.

This work of thesis starts from these multiple issues about Big Data and it focus on different Biomedical topics. In each chapter we are going to handle a different aspect of Big Data research and a different kind of data. For each topic we try to offer a balanced description between the mathematical/theoretical background and numerical issues/solutions of it. The main focus of each application remains its algorithmic description and the numerical solutions developed to handle the underlying problem.

We start our trip across Biomedical Big Data introducing a novel feature selection algorithm. The proposed algorithm is tailored for gene expression analyses and in the various sections of the first chapter we provide a description of all its pros and cons. The algorithm was already used in earlier scientific publications but, for the first time, a deeper analysis of all its characteristics either from a numerical either from a algorithmic point-of-views is provided. The algorithm has undergone also an intensive optimization procedure to make it able to handle Big Data problems in a reasonable computational time. We test our method against a custom toy model and later we compare its efficiency against state-of-art equivalent models and data. We also show some applications of it to different kinds of data, starting from gene expression datasets, passing to protein expression levels, up to other types of non-biological data, proving its efficiency in all these topics. Within the limits of our knowledge about biological processes, we provide also an interpretation about the obtained results where it is possible.

Then, we move to more numerical expensive analyses with the help of modern deep learning models. Starting from a brief introduction about neural network models we look at the different functions/layers included in the later discussed models. For each of them we give a theoretical explanation about the mathematical functions and, also in this case, we deeply focus on their numerical implementation. Three custom libraries are introduced, developed by the author of the thesis, showing the results obtained by them with other state-of-art implementations. We use deep learning neural network models to handle different kind of image processing analyses with a particular attention to biomedical images. As previously discussed, there are multiple image sources in the biomedical field and in our applications we use NMR and CT images. Implementing the most recent Super Resolution algorithms we show their application on NMR images, proving how they can help to increase image quality and how they can be also used to improve object detection tasks. Other kinds of applications are also shown to prove the versatility of such methods in several biomedical tasks.

We end our discussion introducing a novel database, related to the most common diseases, their symptoms and other biomolecular entities, obtained by the harmonization of publicly available datasets. A global description about Big Data sources and how we can handle problems related to the data extraction is discussed before our pipeline of processing. A key role in our work is played by natural language processing methods and, thus, starting from a brief introduction about them we focus on the developed pipeline. The work concern the merging of multiple datasets into a single network structure able to manage the interactions between different biomedical compounds. The network-of-networks structure generated during this project allows a wider overview of several diseases, pointing out their association to genes, drugs and other biological data. We also discuss about how this large amount of information can be managed using modern database languages and about the chosen strategy to share our results to the scientific community.

For sake of brevity, not all the developed projects are discussed and some of the remaining ones are bounded in the Appendix of this text. However, the principal contributions of this work are related to the developed codes. All the codes described in this work are, in fact, publicly available on-line on Github (https://github.com/Nico-Curti/). We have paid special attention to the development of our codes, carefully managing their testing and availability. Each code has been enriched by an adequate on-line documentation, either about its usage either about its installation and performances. A small part of the codes has been written in pure-Python, while the major part has been written in C++: for this reason a continuous integration of them is essential to ensure their usability. We remark that also the current text is publicly available on Github as Latex code, and to ease its reading and its hyper-link connections, we have converted it also into a Gitbook version available at https://nico-curti2.gitbook.io/phd-thesis/.

Chapter 1

Feature Selection - DNetPRO algorithm

After the end of the Human Genome Project (HGP, 2003) [63] there were growing interest on biological data and their analysis. At the same time, the availability of this type of data increased exponentially with the technological improvement of data extractors (High-Throughput technologies) [74] and the lower production costs. These are the main factors that allow us to go into the new scientific era of Big Data. Biological Big Data works with very large and complex datasets which are typically impossible to store, handle and analyze using standard computers and techniques [53]. Just think that we need around 140 GB for the storage of the DNA of a single person and an Array Express, a compendium of public gene expression data, has more than 1.3 million of genomes which have been collected in more than 45 000 experiments [38]. Since the number of available data is getting greater, we need to design several storage databases to organize, classify and, moreover, extract information from them. The Bioinformatics European Institute (EBI) at Hinxton (UK), which is part of the European Laboratory of Biological Molecular and one of the biggest repositories of biological data, stores 20 petabytes of genomic data and proteomics back-ups. The amount of the genomics data is only 2 petabytes, and it doubles every year: it is not worth to remark that these quantities represent about a tenth of data stored by CERN of Ginevra [61]. In contrary, the ability of processing data and the computational techniques of analysis do not grow in the same way. Therefore, the gap between the growth of available data and our ability to work with them is getting bigger.

From a computational point-of-view, the Bioinformatics new-science is looking for new methods to analyze these large amount of data. Common Machine Learning methods, i.e computational algorithms able to find significant patterns into large quantities of data, need to be optimized and modified to increase their computational and statistical performances. To improve the computational time, we need to extend existing methods and algorithms, and to develop new dimensionality reduction techniques. In Machine Learning, in fact, as the dimensionality of the data increases, the amount of data required to perform a reliable analysis grows exponentially[1]. Dimensionality reduction techniques are methods able to find the more significant variables of a given problem or a combination of them, where "significant" means that these few variables (or features) preserve the information about the problem as much as possible. High-dimensional omics data (e.g. transcriptomics through microarray or NGS, epigenomics, SNP profiling, proteomics and metabolomics, but also metagenomics of gut microbiota) pose enormous challenges on how to extract useful information from them. One of the prominent problems is to extract low-dimensional sets of variables – signatures – for classification and diagnostic purposes, or to better stratify

[1] High dimensional data tends to become very sparse and as result it is hard to perform robust statistical evaluation on it. This phenomena is commonly called "curse of dimensionality" [9].

patients for personalized intervention strategies based on their molecular profile [79, 14, 51, 6].

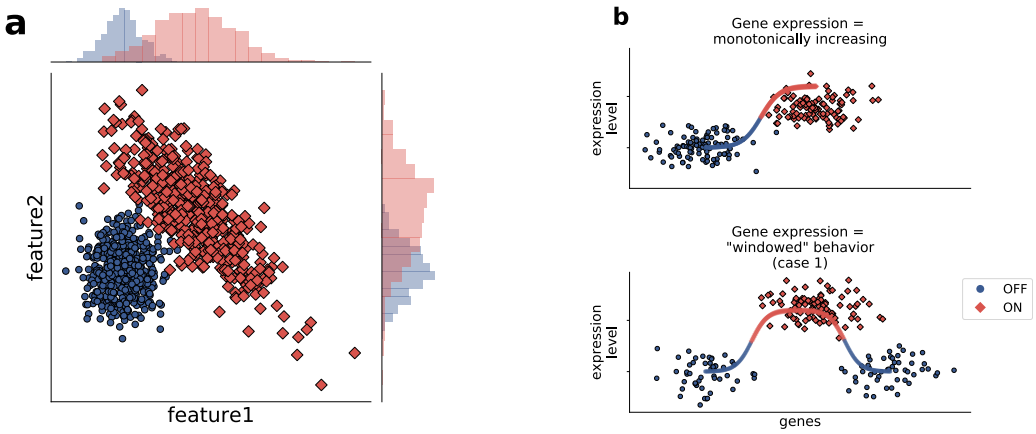


Figure 1.1: (**a**) An example in which single-parameter classification fails in predicting higher-dimension classification performance. Both parameters (*feature1* and *feature2*) badly classify in 1-D, but they have a very good performance in 2D. Moreover, classification can be easily interpreted in terms of relative higher/lower expression of both probes. (**b**) Activity of a biological feature (e.g. a gene) as a function of its expression level: top) monotonically increasing, often also discretized to an on/off state; center, bottom) "windowed" behavior, in which there are two or more activity states that do not depend monotonically on expression level. X axis: expression level, Y axis, biological state (arbitrary scales).

Many approaches are used to face such problems [40], as Elastic Net [48], Support Vector Machine, K-nearest Neighbor, Neural Network and Random Forest [67]. Some methods select variables using single-variable scoring methods [33, 45] (e.g. Student's t test for a two-class comparison), while others search for projections in lower-dimensional variable spaces, but all these approaches could fail even in simple 2-dimensional situations (Fig. 1.1). As shown in Fig. 1.1 (a), both variables perform poorly taken singularly, but their performance becomes optimal taking them together (in terms of linear separation of the two classes).

It is known that complex separation surfaces characterize classification tasks associated to image and speech recognition, for which Deep Networks are used successfully in recent times, but in many cases biological data, such as gene or protein expression, are more likely characterized by an up/down-regulation behavior (as shown in Fig. 1.1 (b) top), while more complex behaviors (e.g. a "windowed" optimal range of activity, Fig. 1.1 (b) bottom) are much less likely. Discriminant-based methods (and logistic regression methods alike) can very likely provide good classification performances in these cases, if applied in at least two-dimensional spaces. The "linearity" of these methods (that generate very simple class separation surfaces, i.e. linear or quadratic) also ensures that a "buildup" of a signature based on lower-dimensional signatures can be done.

These considerations are relevant in particular for microarray data, where we face on few samples compared to a huge amount of variables (gene probes). This kind of problem, often called "large N, small S" problem (where N is the number of features, i.e variables, and S is the number of samples), tends to be prone to overfitting[2] and they are classified to ill-posed. The difficulty on the features extraction can also increase due to noisy variables that can drastically affect the machine learning algorithm. It is often

[2] A solution to a problem is classified as "overfitted" if small fluctuations on the data variance produces classification errors. This problem arises when the model perfectly fits a small training set, but it is not able to generalize to a large amount of test samples (generalization).

difficult to discriminate between noisy and significant variables, even more as the number of variables increases.

In this thesis we propose a new method of feature selection - DNetPRO, *Discriminant Analysis with Network PROcessing* - developed to outperform the problems mentioned above. Our method is designed to gene-expression data analysis and it was tested against the most common feature selection techniques. The method was already applied on gene-expression datasets, but my work focused on its benchmarking and optimization for Big Data applications. The pipeline is composed by several steps and only a part of them were designed for biological application: this allows us to apply (part of) the same method also on different topics with good results (see Appendix for further information about the analyses on non-biological data).

1.1 DNetPRO algorithm

The DNetPRO algorithm produces multivariate signatures starting from all the couples of variables analyzed by a Discriminant Analysis. For this reason, it can be classified as a combinatorial method and the computational time for the exploration of variables' space is proportional to the square of the number of the underlying variables (ranging from 10^3 to 10^5 in a typical high-throughput omics study). This behavior allows to overcome some of the limits of single-feature selection methods and it provides a hard-thresholding approach compared to projection-based variables selection methods. The combinatorial evaluation is the most time-expensive step of the algorithm and it needs an accurate algorithmic implementation for Big Data applications (see the next section for further information about the algorithmic implementation strategy). A summary of the algorithm is shown in 1.

Data: Data matrix (N, S)
Result: List of putative signatures
Divide the data into training and test by an Hold-Out method;
for *couple* $\leftarrow$ *(feature_1, feature_2)* $\in$ *Couples* **do**
  Leave-One-Out cross validation;
  Score estimation using a Classifier;
end
Sorting of the couples in ascending order according to their score;
Threshold over the couples score (Kbest couples);
for *component* $\in$ *connected_components* **do**
  if *reduction* **then**
    Iteratively pendant node remotion;
  else
    S
  end
  ignature evaluation using a Classifier;
end

Algorithm 1: DNetPRO algorithm for Feature Selection.

So, given an initial dataset, with S *samples* (e.g. cells or patients) each one described by N observations (our *variables*, e.g. gene or protein expression profiles), the signature identification can be summarized with the following steps:

- separation of available data into a training and test sets (e.g. 33/66, or 20/80);
- estimation of the classification performance on the training set of all $S(S-1)/2$ variable couples through a computationally fast and reproducible cross-validation procedure (leave-one-out cross validation was chosen);

- selection of top-performing couples through a hard-thresholding procedure. The performance of each couple constitutes a *weighted link* of a network, where the nodes are the variables connected at least through one link;
- every *connected component* which composes the network identifies a putative signature.
- (optional) in order to reduce the size of an identified signature, the pendant nodes of the network (i.e. nodes with degree equal to one) can be removed, in a single step or recursively up to the core network (i.e. a network with all nodes with at least two links).
- all signatures are evaluated onto the test set to estimate their performances.
- a further cross validation step is performed (with a further dataset splitting into test and validation sets) to identify the best performing signature.

We would stress that this method is completely independent to the choose of the classification algorithm, but, from a biological point-of-view, a simple one is preferable to keep an easy interpretability of the results. The geometrical simplicity of the resulting class-separation surfaces, in fact, allows an easier interpretation of the results, as compared to very powerful, but black-box, methods like nonlinear-kernel SVM or Neural Networks. These are the reasons which lead us to use very simple classifier methods in our biological applications as diag-quadratic Discriminant Analysis or Quadratic Discriminant Analysis (Appendix A for more information about the mathematical background and their respectively implementations). Both these methods allow fast computation and an easy interpretation of the results. A linear separation might not be common in some classification problems (e.g. image classification), but it is very likely in biological systems, where many responses to perturbation consist in an increase or decrease of variable values (e.g. expression of genes or proteins, see Fig. 1.1 (b)). This assumption is very plausible for biological data, since genes are in general up- or down-regulated in order to modify their activity and protein and metabolites most of the times respond consequently.

A second direct gain by the couples evaluation is related to the network structure: the DNetPRO network signatures allow a hierarchical ranking of the features according to their centrality compared to other methods. The underlying network structure of the signature could suggest further methods to improve its dimensionality reduction based on network topological properties to fit real application needs, and it could help to evaluate the cooperation of variables for the class identification.

In the end, we remark that our signatures have a purely statistical relevance by being generated with a purpose of maximal classification performance, but sometimes the selected features (e.g. genes, DNA loci, metabolites) can be of clinical and biological interest, helping to improve knowledge on the mechanism associated to the studied phenomenon [5, 79, 12, 86].

1.2 Synthetic dataset benchmark

Standard feature selection algorithms test single-variable performances. Starting from the ranked variables according to their scores, a signature is obtained selecting the top scorer ones following an iterative addition of variables until a desired output score is reached. These methods-like are called K-best algorithms and they filter the number of variables without any constrain on their mutual interaction or correlation. The proposed DNetPRO algorithm tries to extract the more statistically significant variables considering the interaction between them, i.e the combination of variable-pairs. Thus, while the K-best

algorithms scale according to the number of variables, the DNetPRO algorithm is more computational expensive and its usage can be justify only if its efficiency is proved.

We developed a toy model simulation to compare the performances of a standard K-best algorithm with the DNetPRO, considering either the number of samples and the number of variables. Since the DNetPRO algorithm was designed to gene expression applications, our toy model should consider a large number of variables with only a relative small number of samples. To simulate a so like synthetic dataset, we used a toy model generator provided by the scikit-learn Python package. The model generator allows to set a precise number of classes, distinguishing between *informative features*, i.e. variables which easily separate the class populations, and *redundant features*, i.e. variables which represent noise in our problem. The number of informative features should be realistically small compared to the noise, so in our simulations we chose to introduce a maximum of 1% informative features in each simulation.

We randomly generated data from Gaussian distributions with an increasing number of samples and variables, i.e dimensions. In each simulation we split the number of samples in training and test sets (Hold-Out method, with 2/3 of data as training and 1/3 as test) and we applied the DNetPRO algorithm. From each simulation we tested the extracted signatures on the test set, keeping the best performing one. On the same data-subdivision we applied the K-best algorithm, filtering the same number of variables of the DNetPRO best signature, i.e K equal to the number of nodes in the DNetPRO best signature. In this way, we could compare the performances obtained on the test set by the two methods. We would highlight that, in general, there is not a stop criteria for the K-best algorithm, so the number of variables selected could be smaller or greater than the number of DNetPRO signature nodes. However, we can reasonably assume that, according to the K-best interpretation, the selected features should be the most performing ones, and the addition of more variables should introduce only a small quantity of noise. In Fig. 1.2 we show the results obtained in our simulations, keeping fixed the number of variables/samples and varying the number of samples/variables (Fig. 1.2 (a) and Fig. 1.2 (b), respectively).

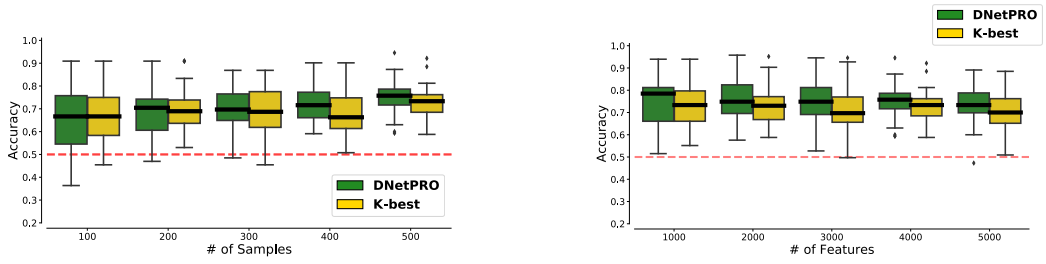


Figure 1.2: Synthetic dataset simulation. Comparison of accuracy performances obtained by the DNetPRO algorithm and the K-best algorithm. **(a - left)** Performances obtained in function of the number of samples, keeping fixed the number of variables. **(b - right)** Performances obtained in function of the number of variables, keeping fixed the number of samples.

For the same number of variables (Fig. 1.2 (a)) we noticed as the two methods perform quite similar but the DNetPRO is able to reach better performances as the number of samples increase. This trend can be explained also in statistical terms: with small samples the variability of our (random) data is large and the performance distributions are more unstable. With a greater number of samples, the variances of our classes are reduced, and the statistical quantities involved in the computation of the discriminant curve can be evaluated with more accuracy. As the number of samples increase, the statistical evaluation of variables becomes easier and the correspondence between the top scorer variables and the true-informative ones increases. In few sample cases, the quantity of noise is big and

in a high dimensional space is hard to find the most informative directions: noise variables can reach higher performances than the informative ones in these cases.

Despite of the simplicity of our toy model, the DNetPRO is able to highlight its efficiency in terms of performances against the single-feature method. A slight different behavior is shown by varying the number of variables and keeping fixed the number of samples (Fig. 1.2 (b)). In this case we noticed that the median accuracy (black line in the plot) of the DNetPRO algorithm always outperforms the K-best one. With a small number of variables (left part of the plot) the K-best algorithm performances are more stable and, only from a statistical point-of-view, we can prove the efficiency of the DNetPRO algorithm (the median of the distribution is still higher compared to the K-best one). As the number of variables increase, also the efficiency of the DNetPRO algorithm increases until it exceeds the K-best results (and its distribution is narrowed). We reached this situation quite faster in our simulation since we constrained our toy model with a forced unbalance between the number of samples and variables, i.e the so-called ill-posed problems. The DNetPRO was designed to work in these situations and it is able to reach high accuracy results also in critical ill-posed problems. The pair-variables evaluation could be helpful to find good variables which are penalized in the single score ranking, but which can prove a good performance-interaction with the others. In these cases, the DNetPRO results could be helpful also to understand the variable interactions, due to the network structure of the signature which can bring to deeper considerations on the fine grain cooperation of variables in a real problem context.

This kind of toy model is considered as a standard for feature selection testing, but it puts several disadvantages for the DNetPRO evaluation. We started our discussion about the DNetPRO taking into account the two distributions of data showed in Fig. 1.1 (a). The DNetPRO algorithm was designed to face that kind of situations in better way. The limits of our algorithm are so bounded to the sample distributions: if the informative variables are totally independent one from each others, the couples evaluation does not guaranteed the best approach to the problem. Considering the signatures extracted by the DNetPRO algorithm we noticed this kind of behavior: the core of our signatures was principally composed by informative variables (which were manually introduced so easily traced) into a star-network structure.

We have to face also the problem of multiple putative (disjointed) signatures: the DNetPRO algorithm takes into account only the connected components with the highest score as putative signature. If the informative variables are disjointed, the corresponding star-networks will be disjointed.This means that we have to enlarge the amount of nodes in our signature.

We evaluated both these situations in our toy model simulations. In the first case, we introduced only two informative variables obtained by a sampling of the distributions showed in Fig. 1.1 (a). In all our simulations, the DNetPRO algorithm was able to identify the couple of these variables as the best putative signature. At the same time the K-best algorithm find with more difficulty those variables, especially when the number of variables become greater. Considering the distribution of single-variable scores, in fact, we could notice as the informative variables, despite they were manually introduced, were not always the top scoring ones: in large dimensional spaces also noisy-variables produced high(er) performances.

Using the same sample distributions for informative features, we manually introduced multiple couples in our dataset. As expected the DNetPRO algorithm is not able to identify into a single connected component, i.e a single putative signature, the full set of informative variables, while the K-best algorithm easily find them in the top scoring ranking. To guarantee the full set of informative features into the DNetPRO signature, we had to enlarge the number of nodes and thus we had to introduce multiple noisy-variables. This

behavior highlights the limits of the DNetPRO algorithm and the need of a (optional) filtering procedure to face these critical cases[3].

1.3 Algorithm implementation

The DNetPRO algorithm is made by a sequence of different steps which have to be performed sequentially for a signature extraction. To this purpose, each step can be optimized independently by using the full set of available computational resources[4]. In this section we analyze each part of the pipeline, focusing on the optimization strategies used for the algorithm implementation.

The full code is open source and available at [17]. The code installation is automatically tested using Travis CI (for Linux and MacOS environments) and Appveyor CI (for Windows environments) at each update (commit). The installation can be performed using CMake and a full set of instructions can be found in the on-line project documentation.

The Python version of the algorithm (see next sections) can be installed via setup.py and its compilable parts built via CMake. The Python installer provides also the full list of project dependencies, which will be automatically installed. A full list of example scripts and utilities to obtain the results showed in the next sections can be found in the Github repository. We provide also the complete benchmark pipeline used in our simulations and able to run on cluster environment using a SnakeMake version of it (see next sections).

1.3.1 Combinatorial algorithm

The most computational time-expensive step of the algorithm is certainly the couples evaluation. From a computation point-of-view this step requires $(O(N^2))$ operations for the full set of combinations. Since we want to perform also an internal Leave-One-Out cross validation for the couple performances estimation, we have to add a $(O(S-1))$ to the algorithmic complexity. Let's focused on some preliminary considerations before discuss about the proposed implementation:

- **Performance:** we aim to apply our method on large datasets and thus we have to take care about the time-performances of this step (identified as bottleneck). To reduce as much as possible the call stack inside our code, we should perform the entire code with the smaller number of functions as possibly. Moreover, we have to simplify for loops and take care about the automatic code vectorization performed by the optimizer at compile time (SIMD, *Single Instruction Multiple Data*). A further optimization step to take into account is related to the cache accesses: the use of custom objects inside the code should benefit from cache accesses (AoS vs SoA, *Array of Structure* vs *Structure of Arrays*).

- **Interdependence:** the performances evaluation is a set of completely independent computational processes and it can be faced on as N^2 separately tasks. Thus, it can be easily parallelizable to increase speed performance.

- **Simplify:** the use of a simple classifier for performances evaluation simplifies the computation and the storage of the relevant statistical quantities. In the discussed

[3] In the algorithm description we discussed about the possibility of removing pendant nodes as optional filtering procedure. The optional step can help but not completely solve the above problem: if there are two disjointed signatures, we have to enlarge the number of nodes and create a connection between them, but this connection would be probably due to a noisy variable. The pendant node remotion can help to reduce the amount of nodes, but links which connect the two components would be preserved.

[4] Further optimization can be performed in a cross validation environment and they will be discussed later in this section.

implementation, we focused on a Diag-Quadratic classifier (see Appendix A for further information) in which only means and variances of data play a role in its computation.

- **Cross Validation:** the use of a Leave-One-Out cross validation allows to perform substantially optimizations in the statistical quantities evaluations across the folds (see discussion in Appendix A - Numerical Implementation).

- **Numerical stability:** we have also to take in care about the numerical stability of the statistical quantities, since we are working under the assumption of a reasonable small number of samples compared to the amount of variables. This hypothesis affects the variance estimation: the chose of a numerically stable formula for this quantity plays a crucial role for the computation, because the classifier score has to be normalized by it.

With these ideas in mind, we have written a C++ code ables to optimize this step in a multi-threading environment, aiming to test its scalability over multi-cores machine.

Starting from the first discussed point, we chose to implement the full code inside a single main function, with the help of only a single SoA custom object and one external function (*sorting algorithm* discussed in the next section). This allows us to implement the code inside a single parallel section, reducing the time of thread spawning. We chose to import the data from file in sequential mode, since the I/O is not (particularly) affected by parallel optimizations (in our simulations).

Following the instructions suggested in Appendix A - Numerical Implementation, we compute the statistical quantities on the full set of data before starting the couples evaluation. Taking a look to the variance equation

$$\sigma^2 = \frac{\sum_{i=1}^{S}(x_i - \mu)^2}{S-1} = \frac{\sum_{i=1}^{S}(x_i{}^2)}{S-1} - \mu^2 \tag{1.1}$$

we can see that the first equation involves the mean computation as a simple sum of elements, using a large number of subtractions that are numerically unstable for data outliers (moreover because they are elevated to square). The better choice, in this case, is given by the second formula, that allows to compute both quantities in the formula inside a single parallel loop[5]. At each cross validation, we use the two pre-computed sums of variables, removing the only data points excluded by the Leave-One-Out. Another precaution to take in care is to add a small epsilon to the variance before its usage in the denominator of the classifier function to prevent numerical underflow.

The set of pair variables can be obtained only by two nested for loops in C++ and a naive optimization can be simply obtained by reducing the number of iterations following the triangular indexes of the full matrix (by definition the score of the couple (i, j) is equal to the score of (j, i)). This precaution easily allows the parallelization of the external loop and drastically reduce the number of iterations, but it also creates a link between the two iteration variables. The new release of OpenMP libraries [28][6] (from OpenMP 4.5) introduces a new *keyword* in the language, that allows the collapsing of nested for loops into a single one (whose number of iterations is given by the product of the single dimensions), in the only exception of a completely independence of iteration variables. So,

[5] To facilitate the SIMD optimization the code is written using only float (single precision) and integer variables. This precaution takes in care the register alignment inside the loops and it facilitates the compiler-optimizer.

[6] The OpenMP library is the most common non-standard library for C++ multi-threading applications.

the best strategy to use in this case is to perform the full set of N^2 iterations with a single collapse clause in the external loop[7].

Listing 1.1: Python parallel couples evaluation algorithm

```
import pandas as pd
import itertools
import multiprocessing
from functools import partial

from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import LeaveOneOut, cross_val_score

def couple_evaluation (couple, data, labels):
  f1, f2 = couple

  samples = data.iloc[[f1, f2]]
  score = cross_val_score(GaussianNB(), samples.T, labels,
                          cv=LeaveOneOut(), n_jobs=1).mean() # nested
    parallel loops are not allowed

  return (f1, f2, score)

def read_data (filename):
  data = pd.read_csv(filename, sep='\t', header=0)
  labels = data.columns.astype('float').astype('int')
  data.columns = labels

  return (data, labels)

if __name__ == '__main__':

  filename = 'data.txt'

  data, labels = read_data(filename)

  Nfeature, Nsample = data.shape

  couples = itertools.combinations(range(0, Nfeature), 2)
  couples_eval = partial(couple_evaluation, data=data, labels=labels)

  nth = multiprocessing.cpu_count()

  with multiprocessing.Pool(nth) as pool:
    score = zip(*pool.map(couple_eval, couples))
```

In this section we provide an "equivalent" Python implementation with the use of common machine learning libraries and parallel settings (ref. 1.1). In the next sections we will discuss about the computational performances of this naive implementation compared to the C++ version discussed above.

1.3.2 Pair sorting

The sorting algorithm starts at the end of variable couple evaluation and it re-orders the variable-pairs in ascending order to ease the next steps of signature identification[8]. This

[7] Obviously the iterations where the inner loop variable is lower than the outer one will be skipped by an if condition.

[8] Talking about performances, in some cases the simple accuracy is useless, especially when we are working with unbalanced population classes. In this case we can use a statistical score which takes in count the balancing between right sample classifications and classes (e.g Matthews Correlation coefficient, MCC). The developed code evaluates either the global accuracy of classification either the MCC and, with slight changes, it allows to perform the pair re-ordering according to the desired score. Since in the

step is performed in the same code (and same parallel section) introduced in the previous section, but it deserves an own topic for a better focus on the parallelization strategy chosen. There are many parallel implementations of sorting algorithms and, to reach the best performances, we have to chose the more appropriate one.

Serial version of sorting algorithms can be found in the major part of the programming languages (Python and C++ included). Also the naive versions of this algorithm are quite optimized and they perform the computation with an algorithmic complexity equal to $(O(N\dot{\log}(N)))$[9]. In this case we do not need to re-invent any sorting technique, but we have to insert as well as possible this algorithm into our parallel section, using the variable format chosen for couple performances storage. Since we work with SoA objects, we need to re-order all the structure arrays in the same way. We can not use a simple sort function, but we can compute the set of indexes that allows the reordering of the arrays, the so-called argsort method. To rearrange the indexes according to a given array of values, we use templates in C++.

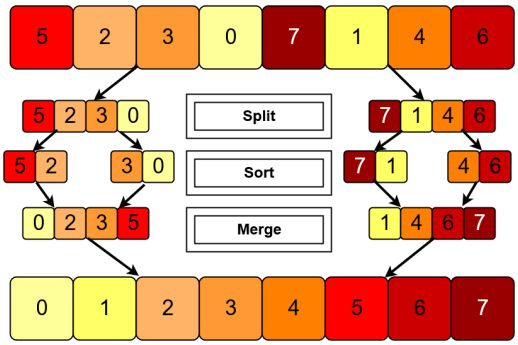


Figure 1.3: Parallel merge-sort algorithm scheme (DAG). Starting from the original array, the master thread splits the work (sub-arrays) along two slave threads (split step in the graph). The split recursion is applied up to a required size of sub-arrays is reached. Each slave-thread applies a sort function (sort step in the graph). Then, the full array is recombined following back the thread recursion and applying an inplace-merge function (merge step in the graph).

As parallelization strategy we can yet invoke the new *keywords* of the OpenMP library, applying a *divide-and-conquer* scheme, using a tree of independent tasks[10]. Using the maximum power of two of the available threads, we split the computation in equal size sub-arrays and we perform independent argsorts. Then, going backward to the subdivisions at each step, we merge the sub-arrays two-by-two up to the master thread (ref Fig. 1.3).

1.3.3 Network signature

After the rearrangement of feature pairs in ascending order, we can start to create the variable network and looking for its connected components as putative signatures. Each feature represents a node in the network and a given pair is a connection between them (link). Since the full storage of the network would require a matrix $(N \times N)$, we need to chose a better strategy for the processing[11].

---

next section we will discuss about the application of the DNetPRO algorithm to real data using only the classification accuracy as score, we will focus only on it in the next sections.

[9] We are considering only un-stable sorting, in which the preserving order of equivalent elements in the array is not guaranteed.

[10] Tasks in OpenMP are code blocks that the compiler wraps up and makes available to be executed in parallel.

[11] We are working in the hypothesis of very large N.

The ordered set of couples computed in the previous section represents a so-called *COO sparse matrix* (Coordinate Format sparse matrix) and we can reasonably assume that the desired signature will be composed by the top ranking of them. So, in the first step we cut a reasonable percentage of available pairs, focusing only on them.

Moreover, we are interested in a small set of variables unknown at prior. Loading all the node pairs into the same graph can slow down the computation. An iterative method (with stop criteria) can perform better and only in worst cases the full set of pairs will be loaded.

Since the described algorithm does not require particular performances efficiency, the main code used in our simulations was written in pure Python. A C++ implementation of the same algorithm was developed with the help of the Boost Graph Library [84] (BGL), but to not overweight the code installation, it was reserved just for a style exercise. In this section we discuss about this second version and also about the strategies chosen to implement an efficient version of it. This version of the algorithm was also used, as stand alone method, for other applications that are discussed in the Appendix of this work.

BGL is a very wide framework for graph analyses based on template structures. The library efficiency discourages users to re-implement the same algorithms and, for the current purpose, it was resulted more than sufficient. Starting from the top scorer feature pairs, we progressively add each couple of nodes to an empty graph. At each iteration step, the number of connected components is evaluated up to a desired number of nodes (greater or equal) is not reached[12]. Two degrees of freedom are left to the user: in order, pruning and merging. The first one performs an iterative remotion of nodes with degree equal (or lower) than 1, i.e pendant nodes, until the graph core is not filtered. The merging clause chooses between the biggest connected component, or the set of all the disjointed connect components, as putative signature. The output of merging step determines the number of nodes in the graph which have to be considered by the stop criteria.

A crucial role in the algorithm optimization is played by the chosen of the BGL graph structure. Since the two degrees of freedom imply a continuous rearrangement of the graph nodes, we have chosen to apply a filter mask over the main graph structure that highlights the only parts of interest. This can be done using the boost :: filtered_graph object of BGL. In 1.2 the C++ snippet is shown.

Listing 1.2: DNetPRO signature extraction

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/connected_components.hpp>
#include <boost/graph/filtered_graph.hpp>
#include <boost/function.hpp>
#include <boost/graph/iteration_macros.hpp>

typedef typename boost :: adjacency_list< boost :: vecS, boost :: vecS,
    boost :: undirectedS, boost :: property< boost :: vertex_color_t, int
    >, boost :: property < boost :: edge_index_t, int > > Graph;
using V = Graph :: vertex_descriptor;
using Filtered = boost :: filtered_graph < Graph, boost :: keep_all, boost
    :: function < bool(V) > >;


std :: vector < int > FeatureSelection (int ** couples, const int &
    min_size, bool pruning=true, bool merging=true)
{
  Graph G;
  std :: set < V > removed_set;
```

[12] This procedure is quite similar to put a threshold value on the couple performances or to highlight inside the full network the components linked by weights greater than a given value.

```
16  Filtered Signature (G, boost :: keep_all {}, [] (V v) {return removed_set
      .end() == removed_set.find(v);});
17
18  int L = 0, leave, Ncomp, i = 0;
19
20  while ( true ){
21
22    boost :: add_edge (couples[i][0], couples[i][1], G);
23
24    while ( pruning ){
25
26      leave = 0;
27      BGL_FORALL_VERTICES (v, Signature, Filtered);
28        if ( boost :: in_degree (v, Signature) < 2 ){
29          removed_set.insert (v);
30          ++ leave;
31        }
32
33      if ( leave == 0 )
34        break;
35    }
36
37    if ( num_vertices (G) - removed_set.size() ){
38
39      components.resize (num_vertices (G));
40
41      Ncomp = boost :: connected_components (Signature, &components[0]);
42
43      if ( merging ){
44
45        BGL_FORALL_VERTICES (v, Signature, Filtered)
46          if ( boost :: in_degree(v, Signature) )
47            core.push_back ( static_cast < int >(v) );
48      }
49      else {
50
51        std :: map < int, int > size;
52        for ( auto && comp : components ) ++ size[comp];
53
54        auto max_key = std :: max_element (std :: begin(size), std :: end(
      size),
55                                           [] (const decltype(size) ::
      value_type && p1, const decltype(size) :: value_type && p2)
56                                           { return p1.second < p2.second;
      })->first;
57
58        BGL_FORALL_VERTICES (v, Signature, Filtered)
59          if ( components[v] == max_key )
60            core.push_back( static_cast < int >(v) );
61      }
62
63      components.resize (0);
64      L = static_cast < int >(core.size());
65    }
66
67    removed_set.clear();
68
69    if ( L >= min_size ) break;
70
71    ++ i;
72
73    core.resize (0);
74  }
```

```
75
76   return core;
77 }
```

In the above description, it should be clear that, given any set of ordered (in ascending order) couples of variables, this algorithm allows to extract the core network, independently by the procedure which generate them. So, it can be used as dimensionality reduction algorithm of general purpose network structures. An example of this kind of application is reported in Appendix B - Venice Road Network in which we summarize the results published in [64, 27].

1.3.4 DNetPRO in Python

Up to now we have focused on the algorithm performances, leaving out the usability of the DNetPRO algorithm for the (research) community. Despite the C++ is one of the most efficient and older programming language[13], the number of Python-users is growing in the last years. Python is becoming leader in scientific research publications and the major part of Machine Learning analyses are performed using Python libraries (in particular scikit-learn library). So, we have to reach a compromise between performances and usability of the new codes and a reasonable solution is given by the Cython [8] language.

Cython "language"[14] allows an easy interface between C++ codes and Python language. With a relatively simple wrapping of the C++ functions, they can be used inside a pure Python code, preserving as much as possible the computational performances of a pure C++ version. In this way, we have written a simple Python object which performs the full set of DNetPRO steps and, moreover, which is compatible with the functions provided by other machine learning libraries.

With these purposes we have chosen to operate a "double wrap" of the C++ functions to separate as much as possible the C++ components from Python[15]. The Python object was written considering a full compatibility with the scikit-learn library to allow the usage of the DNetPRO feature selection method as an alternative component of other Machine Learning pipelines.

1.3.5 DNetPRO in Snakemake

The starting (silent) hypothesis done up to now is that we want to run the DNetPRO algorithm on a single dataset (or better on a single Hold-Out subdivision of data). On this configuration it is legal to stress as much as possible the available computational resources and parallelize each step of the algorithm.

If we want to use our algorithm into a larger pipeline, in which we compare the results obtained over a Cross-Validation, we have to re-think about the parallelization done. In this case, each fold of the cross validation can be interpreted as an independent task and, following the main programming rule *"parallelize the outer, vectorize the inner"*, we should spawn a thread for each fold and perform the couple evaluation in sequential mode.

[13] Still in common use in scientific research groups.

[14] It is not a real programming language since it is based on Python. However, it has its own syntax and keywords which are different either from Python and C++. It also needs a compiler to run and it is certainly different from Python.

[15] Cython wraps are very powerful tools for C++ integration into Python code but, by experience, they are difficult to manage by pure-Python-users. A simple workaround is to perform a first wrap of the C++ functions inside a Cython object, adding a second wrap of it into a pure-Python class. This two-steps wrap certainly gets worse the computational performances, but it allows a complete separation between the compiled part of the code (Cython) and the interpreted (Python) one. Moreover, we can leave back all the checks on input parameters of the C++ function since they can be performed at run time by the Python wrap.

Certainly, the optimal solution would be to separate our jobs across a wide range of interconnected computers, performing the same computation in parallel, but it would required to implement our hybrid (C++ and Python) pipeline into a Message Passing Interface (MPI) environment.

The easier solution to overcome all these problems can be obtained using a set of SnakeMake [52] rules. SnakeMake is an intermediate language between Python and Make. Its syntax is almost like the Make language, but with the help of the easier and powerful Python functions. It is widely used in bioinformatics pipeline parallelization, since it can be easily applied over single or multi-cluster environments (master-slave scheme) with a simple change of command line.

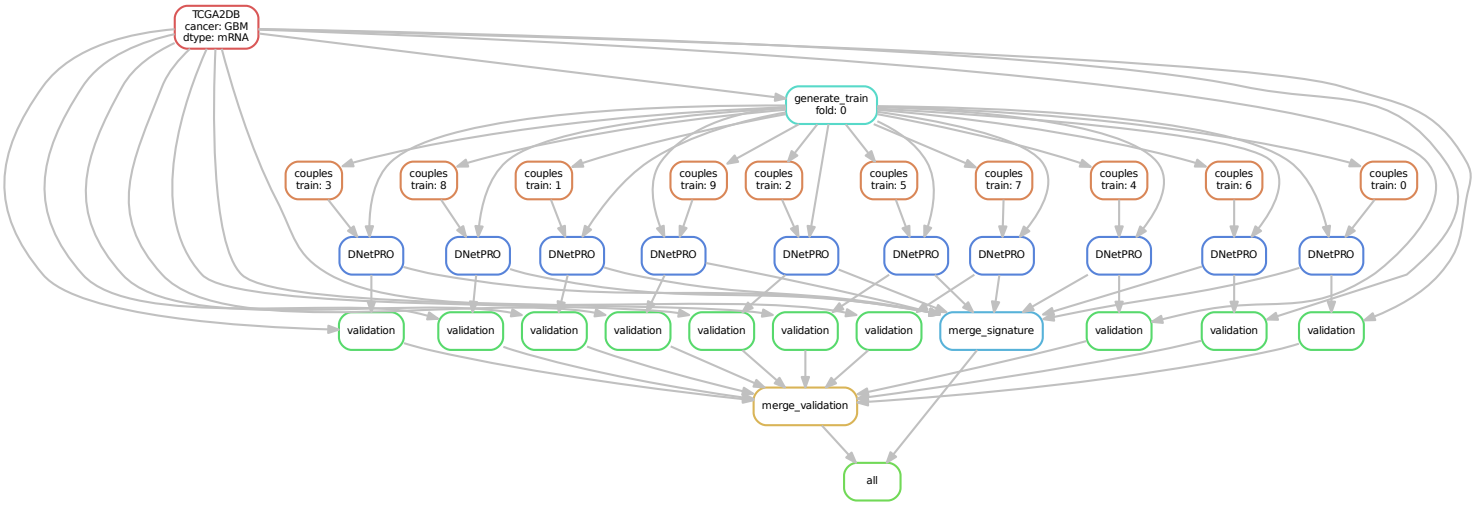


Figure 1.4: Example of DNetPRO pipeline on a single cross validation. It is highlighted the independence of each fold from each other. This scheme shows a possible distribution of the jobs on a multi-threading architecture or for a distributed computing architecture. The second case allows further parallelizations (hidden in the graph) applied to each internal step (e.g. the evaluation of each pair of genes).

So, to improve the scalability of our algorithm we implemented the benchmark pipeline scheme using Snakemake rules and a work-flow example for a single cross-validation is shown in Fig. 1.4. In this case, each step of Fig. 1.4 can be performed by a different computer-unit, preserving the multi-threading steps with a maximum scalability and possibility to enlarge the problem size (number of variables).

1.3.6 Time performance

As described in the above sections, the DNetPRO is a combinatorial algorithm and thus it requires a particular accuracy in the code implementation to optimize as much as possible the computational performances. The theoretical optimization strategies, described up to now, have to be proved by quantitative measures.

We tested the computational performances of our Cython (C++ wrap) implementation against the pure Python (naive) implementation showed in 1.1. The time evaluation was performed using the timing Python package in which we can easily simulate multiple runs of a given algorithm[16]. In our simulations, we monitored the three main parameters related

[16] We would stress that we can use the timing Python package only because we provided a Cython wrap

to the algorithm efficiency: the number of samples, the number of variables and (as we provided a parallel multi-threading implementation) the number of threads used. For each combination of parameters, we performed 30 runs of the both algorithms and we extracted the minimum execution time. The tests were performed on a classical bioinformatics server (128 GB RAM memory and 2 CPU E5-2620, with 8 cores each). The obtained results are shown in Fig. 1.5. In each plot, we fixed two variables and we evaluated the remaining one.

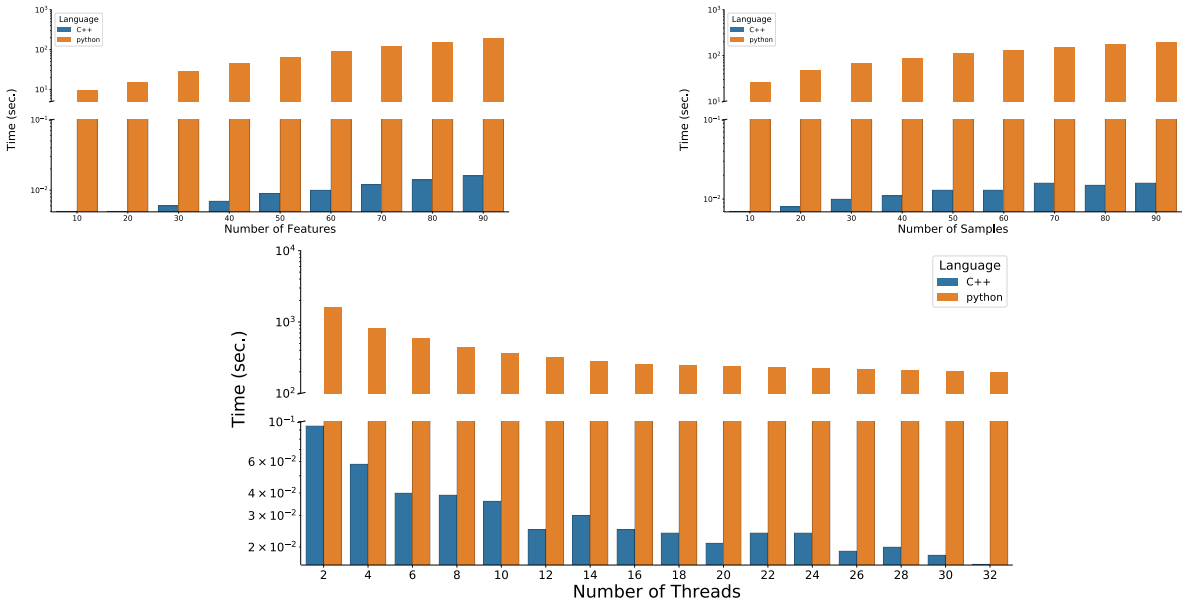


Figure 1.5: Execution time of DNetPRO algorithm. We compare the execution time between pure-Python (orange) and Cython (blue, C++ wrap) implementations. **(a - left)** Execution time in function of the number of variables (the number of samples and the number of threads are kept fixed). **(b - right)** Execution time in function of the number of samples (the number of variables and the number of threads are kept fixed). **(c - bottom)** Execution time in function of the number of threads (the number of variables and the number of samples are kept fixed).

In all our simulations, the efficiency of the (optimized) Cython version is easily visible and the gap between the two implementations reached more than 10^4 seconds. On the other hand, it is important to highlight the scalability of the codes against the various parameters. While the code performances scale quite well with the number of features (Fig. 1.5 (a)) in both the implementations, we have a different trend varying the number of samples (Fig. 1.5 (b)): the Cython trend starts to saturate almost immediately, while the computational time of the Python implementation continues to grow. This behavior highlights the results of the optimizations performed on the Cython version which allows the application of the DNetPRO algorithm also to larger datasets without loosing performances. An opposite behavior is found monitoring the number of threads (ref Fig. 1.5 (c)): the Python version scales quite well with the number of threads[17], while the Cython trend is more unstable. This behavior is probably due to a non-optimal scheduling in the parallel section: the work is not equally distributed along the available threads and it penalizes

of our DNetPRO algorithm implementation. We would also highlight that, albeit minimal, the Python superstructure penalizes the computational performances and the best results can be obtained using the pure C++ version of the code.

[17] The optimal result should be a linear scalability with the number of threads but it is always difficult to reach this efficiency. Thus, a reasonable good result is given by a progressive decrease, increasing the number of threads.

the code efficiency, creating a bottleneck related to the slowest thread. The above results are computed considering a number of features equal to 90 and, thus, the parallel section distributes the 180 ($N \times N$) iterations along the available threads: when the number of iterations is proportional to the number of threads used (12, 20 and 30 in our case), we have a maximization of the time performances. Despite of this, the computational efficiency of the Cython implementation is so much better than the Python one that its usage is indisputable.

1.4 Benchmark of DNetPRO algorithm

Up to now we have been talked about the DNetPRO algorithm from a theoretical and numerical points-of-view. Starting from this section, we discuss about the application of this algorithm on real biological datasets (see Appendix B - Venice Road Network for results obtained on non-biological data types).

Previous versions of the DNetPRO algorithm have been already applied on biological data [5, 79, 12, 86], but in this work we want to introduce a wide range benchmark of it. In the following sections we are going to describe the results obtained on the Synapse dataset and published in [26].

1.4.1 Synapse dataset

As benchmark dataset was chosen the core sets extracted from The Cancer Genome Atlas (accession number syn300013, doi:10.7303/syn300013) (*Synapse dataset* in the following), used in a previous study [89] which aimed at quantifying the role of different omics data types (e.g. mRNA and miRNA microarray data, protein levels measured with Reverse Phase Protein Array - RPPA) via different state-of-the-art classification methods. This allowed us to compare our results to a large set of commonly used classification methods, by using their performance validation pipeline (accession number syn1710282, doi:10.7303/syn1710282).

The Synapse dataset is composed by four tumors datasets: kidney renal clear cell carcinoma (KIRC), glioblastoma multiforme (GBM), ovarian serous cystadenocarcinoma (OV) and lung squamous cell carcinoma (LUSC). For each cancer type we applied the DNetPRO algorithm on mRNA, miRNA and RPPA data and we compare the performances results with the Yuan et al. ones.

The summary description of the datasets used is reported in the Tab. 1.1.

Each tumor dataset was pre-processed by adding a zero-mean Gaussian random noise ($\sigma = 10^{-4}$)) to remove the possible null values in the database, which could produce numerical errors in the distances evaluation between genes. Then, we randomly split each dataset in training and test sets with a stratified (i.e. balanced for class sample ratio) 10-fold procedure: with the stratification we are reasonably sure that each training-set is a good representative of the whole sample set. The choice of a 10-fold splitting is aimed to reproduce the analysis pipeline presented by Yuan et al. with an analogous cross-validation procedure. Since we don't have exact details of their data splitting, the cross validation was repeated 100 times, for a total of 1000 training procedures for each tumor (OV, LUSC, KIRC, GB) and data type (mRNA, miRNA, RPPA). Each training procedure led to the extraction of multiple signatures.

We chose threshold values in order to obtain a resulting number of variables (network nodes) in the order of $10^2 - 10^3$, and identified all connected components of the network as signatures. If more than one component existed, each one was considered as a different signature.

The final multidimensional signatures were tested by a Discriminant Analysis with a

Cancer	mRNA	miRNA	Protein	Number of samples
GBM	AgilentG4502A	H-miRNA_8x15k	RPPA	
	17814	533	[a]	210
KIRC	HiseV2	GA+Hiseq	RPPA	
	20530	1045	166	243
OV	AgilentG4502A	H-miRNA_8x15k	RPPA	
	17814	798	165	379
LUSC	HiseqV2	GA+Hiseq	RPPA	
	20530	1045	174	121

Table 1.1: In the first row platforms are reported and the second shows the dimension of dataset as number of probes. AgilentG4502A: Agilent 244K Custom Gene Expression G4502A; HiseqV2: Illumina HiSeq 2000 RNA Sequencing V2; H-miRNA_8x15K: Agilent 8 × 15K Human miRNA-specific microarray platform; GA+Hiseq: Illumina Genome Analyzer/HiSeq 2000 miRNA sequencing platform; RPPA: MD Anderson reverse phase protein array. The last column shows the number of sample.
[a] Missing data-type for that cancer type.

diag-quadratic distance, to avoid possible problems about covariance matrix inversion (as for the Mahalanobis distance).

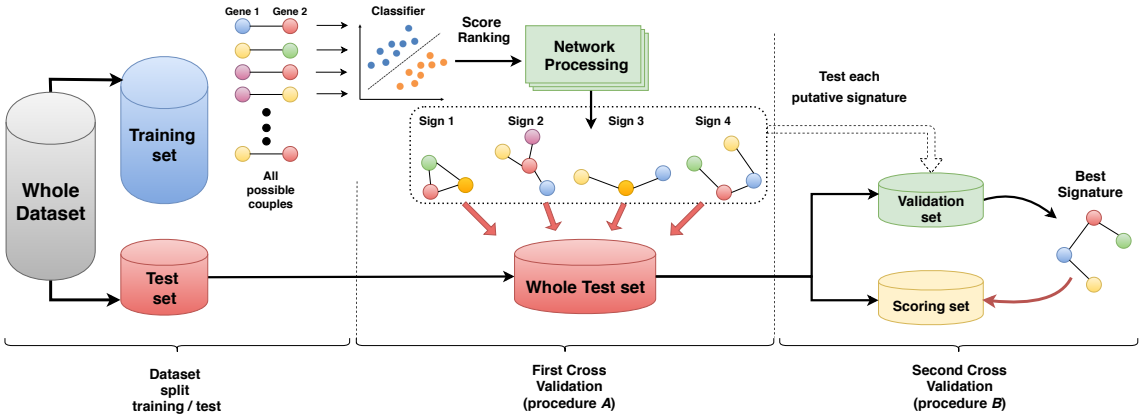


Figure 1.6: Scheme of DNetPRO algorithm. On the "training set", all possible couples of variables are used for Discriminant Analysis, generating the fully connected network weighted by classification performance. Thresholding ranked couples, several signatures can result (as connected components) and their performance is evaluated on the "whole test set" (procedure A). A unique best signature can be identified on a "validation set" and tested in a "scoring set", obtained by further splitting the "whole test set" (procedure B).

We remark that DNetPRO can provide more than one signature as a final outcome, given by all the connected components found in the variable network, or a unique top-performing signature can be obtained by a further cross-validation step (procedure A and procedure B in Fig. 1.6, respectively).

In the single cross validation configuration (procedure A in Fig. 1.6), the best signature was extracted as the one reaching the highest accuracy score during the training step. This best signature was then tested over the available test set.

When also the second cross validation was used (procedure B in Fig. 1.6) the best signature wasted as the most performing over a subset of the whole test set (*validation set*), and the final performance was evaluated on the remaining *scoring set*.

To compare our results with the work of Yuan et al., we used the AUC (*Area Under the Curve*) score, that they provided in the paper as the result of their analyses. The distribution of our results could be compared to the single score value given in the other work.

1.4.2 mRNA dataset

We applied both training procedure (ref. Fig. 1.6) on the mRNA dataset. The results are shown, as distribution of AUC (Area under the curve) score, in Fig. 1.7 (a) for the best signatures obtained with procedure A (corresponding to the validation approach used in [89]), while results with the full cross-validation procedure B are shown in Fig. 1.7 (b).

As expected, performances decrease with the introduction of the second cross validation step, but the values remain quite stable showing the robustness of the extracted signatures, and we remark that the validation procedure used in the reference paper by Yuan et al. resembles our approach without the second validation step.

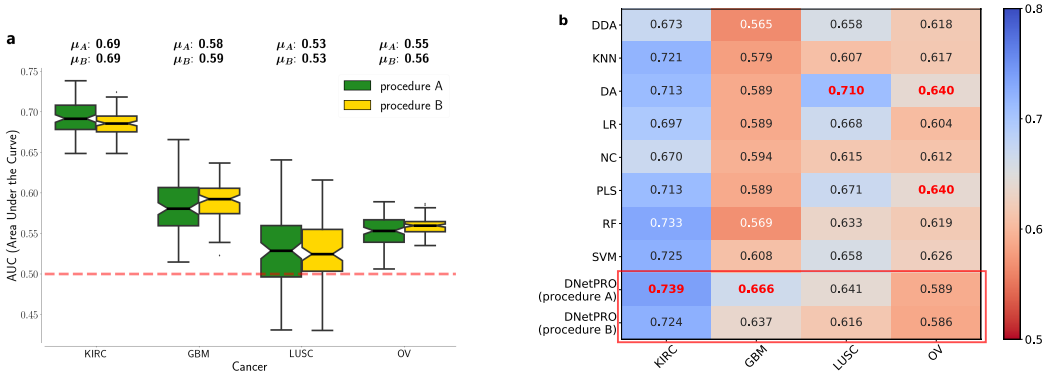


Figure 1.7: Results obtained by the DNetPRO algorithm pipeline on four mRNA tumor datasets, as from the Synapse database [89]. **(a)** Distributions of AUC values for the tumor datasets. Green boxplots: results using procedure A as described in Fig. 1.6; yellow boxplots: results obtained using procedure B. **(b)** Comparison of DNetPRO results with the methods used in the paper of Yuan et al.: max AUC values obtained over the 10-Fold cross-validation procedure.

All results are comparable (LUSC) or better (KIRC, GBM) than the results reported in [89], except for the OV dataset, also with the more conservative approach involving a further cross-validation step. The size of the extracted signatures is quite constant, and smaller than 500 genes in each pipeline execution.

To test the robustness of our method, since each cross-validation procedure may generate different signatures, we measured the overlap of the genes belonging to each mRNA signature over 100 simulations with different training-test data splitting. We observed an average overlap ranging from 40% to 60%, with a smaller group of genes found across all the 100 cross-validation iterations.

In this application the DNetPRO algorithm has several advantages: easy scalability on parallel architectures, simple signature interpretation allowing a valuable application in a biomedical context and a significant robustness in a highly noisy environment such as genomics measurements.

1.4.3 miRNA and RPPA dataset

The same analysis pipeline presented in the paper about gene expression data from the Synapse dataset is applied in this Section also to the miRNA and RPPA datasets, with the results presented in Fig. 1.8.

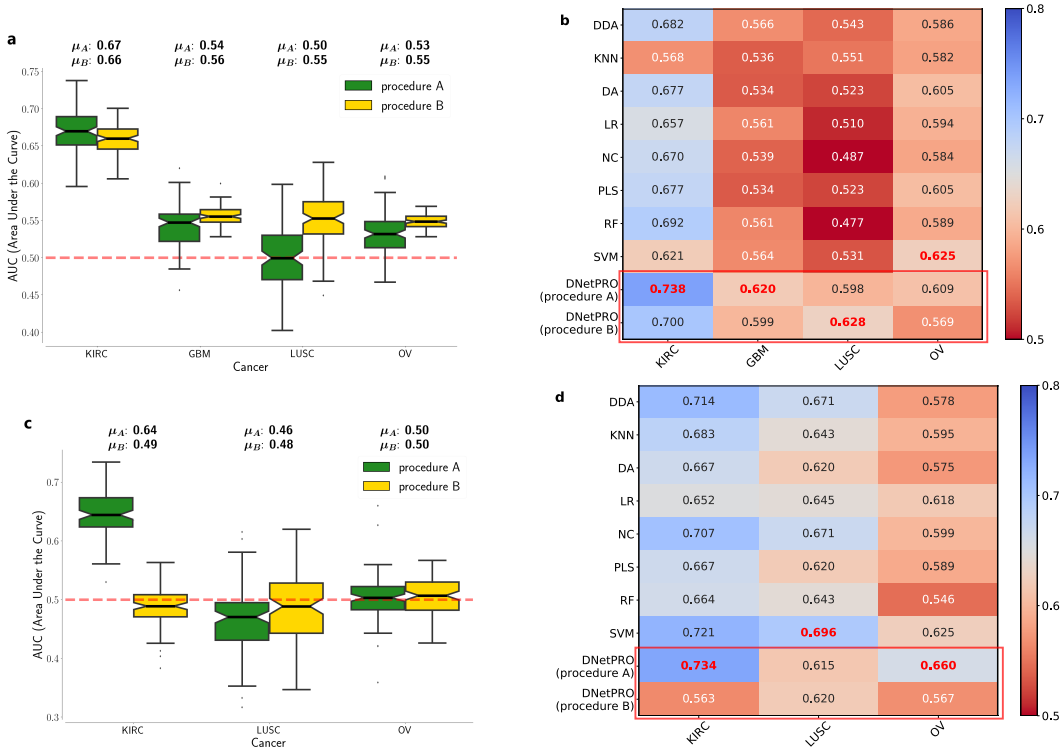


Figure 1.8: Results obtained by the DNetPRO algorithm pipeline on the four Synapse miRNA and RPPA tumors datasets. **(a, c)** Distributions of AUC scores obtained over the four datasets. Green box-plots: results using procedure A of DnetPRO; yellow box-plots: results obtained using procedure B. **(b, d)** Comparison of DNetPRO with the methods used in [89]. The reported values are the max AUC values obtained over the 10-Fold cross-validation procedure.

The results obtained on the miRNA dataset (Fig. 1.8 (a, b)) are comparable to the reference, while for the RPPA dataset only the LUSC tumor shows AUC values comparable with the others. Moving from the procedure A to the procedure B, i.e. adding a second cross-validation step, the RPPA performances drastically decrease for the KIRC and OV, while their remain quite stable for the LUSC dataset. The same behavior is shown in the miRNA datasets in which however both performances are still comparable or better (KIRC, GBM, LUSC) than the reference ones.

The obtained results show the efficiency of the DNetPRO algorithm also in applications very far from the mRNA one. Despite the good performances we have to better clarify the biological background of these data: in the mRNA dataset we hypothesized a monotonic behavior of genes (up- or down-regulation of gene expression level) and this model is very likely. An analogous model for the miRNA data has not yet been demonstrated. This behavior is even more true when we consider RPPA data. RPPA data are often affected by a wide series of experimental difficulties about data interpretation. In biological applications we have to consider often the technique used to acquire data: different experiments can introduce different noise sources which can affect our model performances and thus the DNetPRO application and interpretation.

We conclude this section by remarking that the signatures obtained by the DNetPRO

algorithm aim to identify a set of significant statistical variables. The network structure of the signature is designed to simplify a possible interpretation of the variables involved, but no prior biological knowledge is taken into account in our algorithm. Therefore, a biological interpretation of the DNetPRO signature could be proved only occasionally.

1.4.4 Couple ranking

Since the number of variable pairs is typically very large (e.g. 10^8 pairs with gene expression microarrays containing about 10^4 probes), many of them may achieve the same performance, since the possible values are integer number typically in a limited range (corresponding to the number of available samples, $10^2 - 10^3$ in many cases). Therefore, the pair ranking is characterized by multiple "plateaus" (Fig. 1.9 (a)), and the selection of variable pairs, based on a hard thresholding procedure, is highly influenced by this behavior. Monitoring this trend we can notice that only a few number of pairs belong to the first performance chunks and, while the performances decrease, multiple pairs (and features) appear, as it can be seen in Fig. 1.9 (a).

This kind of trend highlights the difficulty on finding informative features inside the huge noise of other variables and it gives us a constrain in the developing of a realistic biological toy model (ref. previous sections). Moreover, it confirms that a putative signature could be made by only a few central genes, at least weakly connected with other noisy nodes.

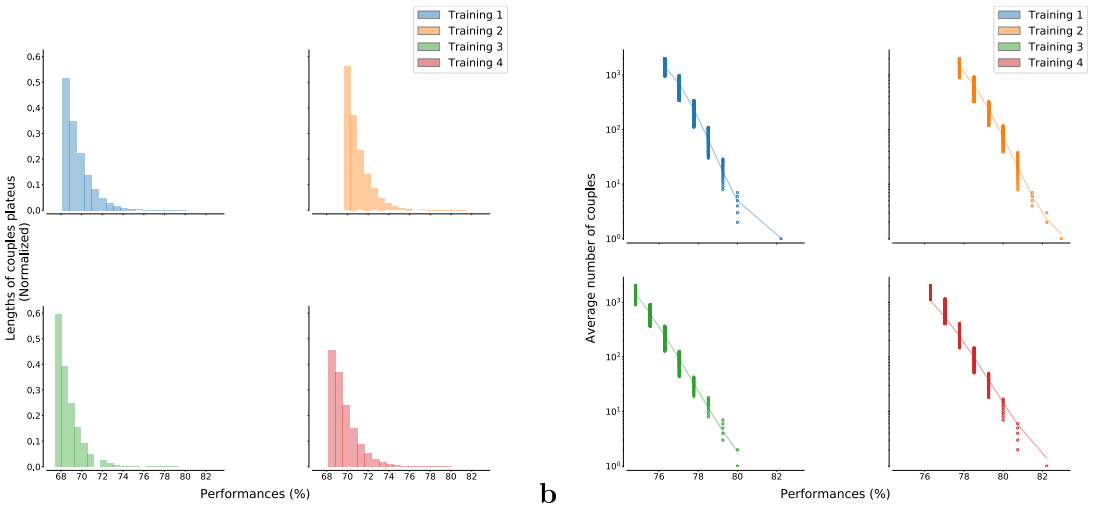


Figure 1.9: Analysis of ranked pairs distributions according to the performance score obtained in the training step. (**a**) The distribution of plateau lengths is approximately exponential. (**b**) Average number of pairs with the same score value: this behavior is typical in ranking order distribution and it can be fitted by the relation $f(x) = A(M+1-r)^b/r^a$ as shown in [60], where r is the rank value, M its maximum value, A a normalization constant and (a, b) two fitting exponents.

As in other cases of ranked values [60], we can fit these ranking distributions with a combination of power-law functions, obtaining a good agreement with experimental points (Fig. 1.9 (b)).

We also observed that *star*-networks frequently appear with one variable highly connected to many others which are only connected with it. This happens when a variable has a strong discriminating power, to which other possibly less relevant variables get linked for noisy fluctuations.

As stated before, we suggest that these variables (pendant nodes in the *star*-network) can be removed from the signature without significantly affecting its performance. The

procedure can be applied for one single step (in order to remove pending nodes from a star configuration) or it can be applied recursively, until the signature becomes constituted only by the 2-core network (i.e. with all nodes having degree ≥ 2). Empirical analysis performed on real data has shown that the removal of these variables does not affect significantly the signature performance and in the meanwhile it allows a significant reduction of its dimensionality. Since there is no clear theoretical explanation of this behavior, we suggest to introduce this step only optionally, since it is not easy to quantify the risk of loosing relevant information from the removed variables. The underlying idea is that the more connected are the nodes, the more the variables in the signature "work well" together, a plausible hypothesis given by the linear sample separation surface provided by the Discriminant classifier. Moreover, the network structure of the signature suggests further considerations about the relevance of a variable as a function of its role in the network (e.g. node centrality such as degree or betweenness centrality).

1.4.5 Characterization of signature overlap

In the analysis of the Synapse dataset we used a complex pipeline of cross-validation (ref Fig. 1.6) to obtain a sufficient statistics. The DNetPRO algorithm was instead designed to work on a single dataset, since the signature extraction can involve different variables for different data subdivisions. In our application, we divided the dataset into a training-test subdivision and the signature were extracted along a 10-fold cross-validation over the training set. This kind of setup could produce 10 totally different signatures, in the worst case. Moreover, we replicated our simulation for 100 repetitions and thus a set of 1000 totally independent signatures were extracted.

Starting from this large amount of variables, we evaluated the robustness of the DNetPRO algorithm in the variable identification, studying the overlap between the obtained signatures. From a statistical point-of-view, it is quite unlikely that the same set of variables were included into all the extracted signatures, especially on this application in which variable roles are assumed by genes. On the other hand, the overlap of these signatures could highlight a statistical significance of some variables, and thus genes related to the understudied tumors.

As case study, we analyzed only the KIRC mRNA dataset, in which the extracted signatures ranged from 4 to 650 genes ($\mu = 382$ genes). For each gene we counted its occurrences along the 1000 signatures. The same analysis was performed taking into account the signatures generated using the K-best score variables (ref. 1.2 for further information) and a random features extraction (null model). In Fig. 1.10 gene distributions obtained by the three methods are shown.

Both DNetPRO and K-best feature extraction algorithms identified a core set of genes common to all the signatures. The random feature extraction method, instead, is not even comparable with the others and it simply represents a null model.

The K-best algorithm appears more stable than the DNetPRO algorithm and it is easier to find the same genes along the extracted signatures. This behavior could be associated to the problems highlighted also in the toy model simulations (ref. 1.2): the DNetPRO algorithm is able to identify only one signature, but the informative features (genes) could not co-operate in the same network-signature and thus they could be discarded. The DNetPRO signatures were, in fact, very small compared to the number of variables, and thus only small network components were extracted, which were very closed to star-networks. Despite the discrepancy between the signatures we have a core of 18 genes which occurs in at least the 95% of both method-signatures (8 of them are common in the 99%).

The common genes were mapped on public databases (TISIDB [77] and Oncotarget [69]), which link tumors to related genes. We found 14/18 genes as informative probes for the KIRC tumor in the TISIDB and 7 of them were also found in the Oncotarget

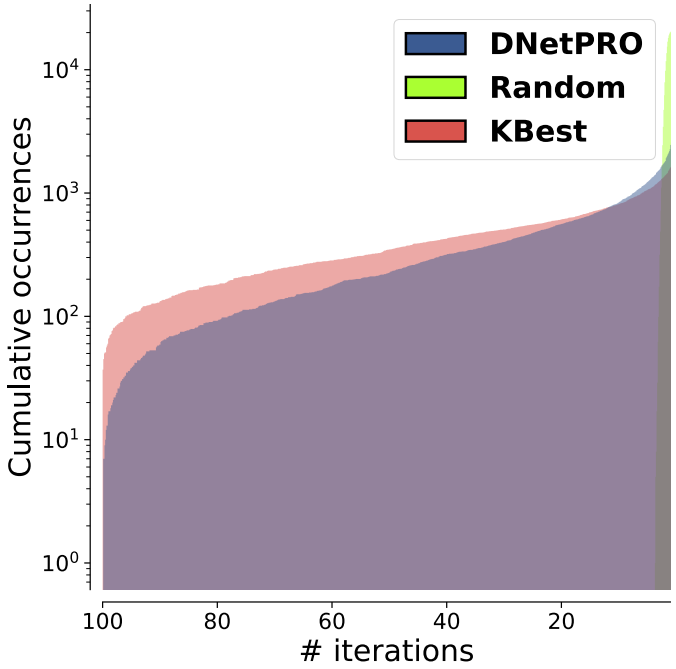


Figure 1.10: Signatures overlap obtained in the KIRC mRNA datasets. For each gene we counted its occurrences along 1000 signatures obtained by three different methods. In blue we represent the distribution of the overlap of the 1000 signatures obtained by the DNetPRO algorithm applied to the Synapse dataset. In red we represent the distribution of the overlap of the 1000 K-best variables extracted from the Synapse pipeline: the number of genes (K) is the same of the corresponding DNetPRO signature in each extraction. In yellow we represent the distribution of 1000 random signatures: a random sampling could be interpreted as null model.

database[18]. Taking into account the core set of 8 genes, we found 3 of them on Oncotarget database and 7 of them on the TISIDB. The only exception was given by the LOC388796 gene, which has not been found in any database.

1.5 Cytokinome dataset

Increasing evidence suggests that inflammation is involved in Alzheimer's disease (AD) pathogenesis. Elevated peripheral levels of different cytokines and chemokines in subjects affected by AD compared with healthy control (CTL) have emphasized the role of peripheral inflammation in the disease. Thus, these proteins can represent specific factors of disease development and progression. Considering the cross-talking between the central nervous system and the periphery, the inflammatory analytes may provide utility as biomarkers to identify AD at earlier stages, in particular for the diagnosis of Mild Cognitive Impairment (MCI), a condition at risk of development of dementia. AD is a major neurocognitive disorder and the most common cause of dementia in the old age, accounting for 60% to 80% of all causes. During the past decade, a conceptual shift occurred in the field of AD considering the disease as a continuum. In this context, there is an urgent need for biomarkers identification able to accurately detect AD in an early stage, before the appearance of neurologic signs. An early diagnosis can hopefully lead to a better and more effective treatment, which could potentially limit neuronal damage and prevent the development of overt AD. An emerging field in the study of neuroinflammation is the sex-related differences: in the last years, gender studies have been increasingly developed with the aim to adopt gender differences as a key to interpretation many diseases, including neurodegenerative diseases.

Experimental data showed that many mechanisms are involved in AD pathogenesis including neuroinflammation. The dysregulation of cytokines and chemokines is a central feature in the development of neuroinflammation, neurodegeneration, and demyelination both in the central and peripheral nervous systems. Among many chemokines and cytokines, pro-inflammatory IFNα2, TNFα, and IL-1α are described as heterogeneously implicated in AD pathogenesis.

The interactive network of cytokines/chemokines, defined as "cytokinome", is extremely complex. Using the DNetPRO algorithm as statistical feature selection method, we might discriminate the groups and propose a useful tool to follow the progression and evolution of AD from its early stages, also in light of gender differences. With this study, we aimed first at the identification of a potential proteins profile able to discriminate AD, MCI and CTL and, therefore identify a potential early and easy to get a diagnostic marker of subjects at risk.

Further information about this work can be found in the original paper [10].

1.5.1 Dataset

In this case-control observational study, we evaluated 289 old-age subjects referred to our Geriatric Memory Clinic. The dataset comprises 189 female and 100 male individuals with a mean age of 78.6 ($\pm$7.5) years. The date were provided by the co-authors of this project at the Institute of Gerontology and Geriatrics at the University of Perugia (Department of Medicine). For each patient a set of 26 cytokine expression level were computed with the additional information about subject sex, age and diagnosis label (AD, MCI or CTL). Of

[18] The list of genes in the TISIDB cover "only" 988 genes. From our list we have only one gene which was found in the Oncotarget database and not in the TISIDB. This gene misses in the TISIDB so we can not evaluate its importance.

the 289 enrolled subjects, the whole set of cytokines was available for 284 subjects (98%), specifically 87/88 CTL (99%), 70/73 MCI (96%), 127/129 AD (98%).

To approximate normal distribution, plasma cytokines and chemokines were log-transformed for data analyses. For the analysis of single cytokines with respect to the CTL, MCI and AD group, we designed a linear model analysis, with the value of each cytokine as a linear combination of the subject group (with CTL samples as the baseline, and MCI, AD as conditions), age and sex, as factors (the formula representation would be "cytokine $\sim$ group + sex + age"). The last two were included as possible confounding factors, even if the analyses revealed that their role for each cytokine is marginal. Only IFNα2, IL-1α, and MCP-1 differed among groups after correction for age and sex. A threshold $p < 0.05$ was considered for significance at all levels (group, sex or age).

Then we applied the DNetPRO algorithm looking for a signature capable of discriminating between CTL and AD: to this purpose, we performed a Hold-Out cross-validation procedure to identify the cytokine signature, considering 2/3 of samples to train the model and then we tested the signature performance on the remaining 1/3 of the total samples. In this analysis we did not separate male from female samples, to avoid the bias given by the uneven number of samples in these two groups, and since previous analysis at a single-cytokine level did not find significant differences due to sex. Then, we classified MCI samples with the CTL-AD signature obtained in the previous step, that allowed labeling MCI samples as CTL or non-CTL.

1.5.2 Results

The best signature identified to discriminate between CTL and AD subjects is composed of three cytokines, IFNα2, TNFα, and IL-1α. Its total accuracy on the CTL-AD test set is 65.27% (with 61% CTL and 66% AD correctly classified). The sensitivity/specificity values for classification is reported in Tab. 1.2.

	Accuracy	**Sensitivity**	**Specificity**
	AD vs. CTL	AD	CTL
Men	16/25	8/12	8/13
	(64.00%)	(66.67%)	(61.54%)
Women	33/48	27/38	6/10
	(68.75%)	(71.05%)	(60.00%)
Total	47/72	36/54	11/18
	(65.24%)	(66.66%)	(61.11%)

Prediction	**Sensitivity**	**Specificity**
MCI as non-CTL	MCI	CTL
15/26	15/26	24/36
(57.69%)	(57.69%)	(66.67%)
41/47	41/47	23/51
(87.23%)	(87.23%)	(45.09%)
62/73	62/73	36/87
(84.93%)	(84.93%)	(41.38%)

Table 1.2: The sensitivity/specificity values for AD vs CTL classification by the 3-protein signature, for the total sample dataset and stratified by sex. The second table shows the result of predictions of MCI samples with the same signature as non-CTL samples. In this case the sensitivity and specificity were computed in relation to the CTL in the training set.

Applying this signature to classify MCI vs CTL samples, it correctly predicted 84.93% of MCI as "non-CTL". Two cytokines from the signature, IFNα2 and IL-1α, showed a significant difference between groups also at a single cytokine level in previous analyses. We plotted them as a representative in all population and stratified the scatter plots by sex (ref Fig. 1.11 A, B). The CTL group resulted better separated from MCI and AD in women as compared with men. The trajectory of the subject groups moves from CLT to AD, and interestingly the identified signature is able to differentiate MCI from CLT better than from AD. This is a promising result since it seems more useful to recognize MCI from CLT than full-blown AD from CLT. Probably the poor sensibility in detecting

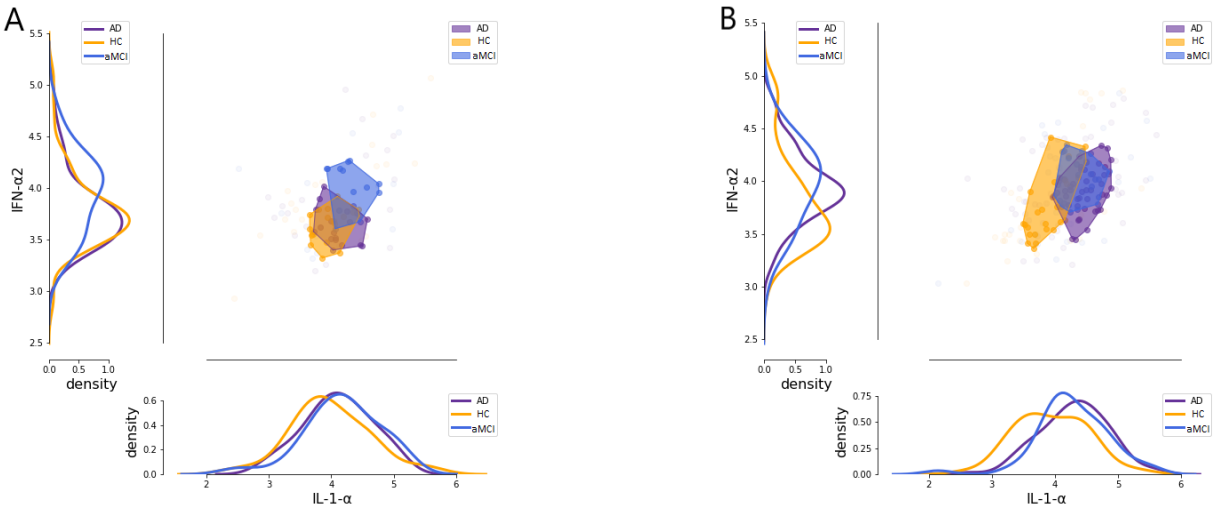


Figure 1.11: **(A)** Scatter plot of IL-1α and IFN-α, and distribution plot for the single cytokines along the axes, stratified by diagnostic group (AD, CTL, and MCI) in **males**. In this case, the HC group is less separated from MCI and AD. **(B)** Scatter plot of IL-1α and IFN-α, and distribution plot for the single cytokines along the axes, stratified by diagnostic group (AD, CTL, and MCI) in **females**. In this case, the HC group is well separated from MCI and AD.

AD could be linked to the disease evolution that makes nebulous and vague the cytokine pattern in the brain of these patients, as confirmed from several studies that found both up-regulation and down-regulation of many cytokines in AD cerebral samples. This fact could be more accentuated in our population of old age subjects in which markers of aging are often mixed with those of dementia.

In this study we show that: 1) an easy to get cytokines signature composed of three molecules - IFNα2, TNFα, and IL-1α - is able to discriminate the studied groups; 2) the combination of IFNα2 and IL-1α able to distinguish CTL from MCI and AD better in women than in men. Sex (referred to biological differences) and gender (psychosocial and cultural differences) affect human brain biology throughout individual lifespans, affecting male and female cognitive functions differently. Epidemiological studies show that women have a higher risk of AD as well as a higher dementia prevalence, particularly in the old age, as compared with men.

In conclusion, the identified cytokinome signature shows a good accuracy in differentiating MCI from CTL, especially in female. Understanding sex differences will help to define individualized preventive and treatment interventions for AD.

1.6 Bovine Dataset

Paratuberculosis or Johne's disease (JD) in cattle is a chronic granulomatous gastroenteritis caused by infection with *Mycobacterium avium subspecies paratuberculosis* (MAP). JD is not treatable; therefore the early identification and isolation of infected animals is a key point to reduce its incidence worldwide. In this work DNetPRO algorithm was applied to RNAseq experimental data of 5 cattle positive to MAP infection compared to 5 negative uninfected controls. The purpose was to find a small set of differentially expressed genes able to discriminate between infected animals in a pre-clinical phase. Results of the DNetPRO algorithm identified a small set of 10 transcripts that differentiate between potentially infected, but clinically healthy, animals belonging to paratuberculosis positive herds and negative unexposed animals. Furthermore, the same set of 10 transcripts differentiate neg-

ative unexposed animals from positive animals based on the results of the ELISA test[19] for bovine paratuberculosis and fecal culture. Within the 10 transcripts that together had good discriminative potential, 5 (TRPV4, RIC8B, IL5RA, ERF and CDC40) show significant differential expression between the three groups while the remaining 5 transcripts (RDM1, EPHX1, STAU1, TLE1, ASB8) did not show a significant differences in at least one of the pairwise comparisons. In conclusion, the discriminant analysis described here identified a set of 10 genes that discriminate between the exposed and sero-converted animals. When tested in a larger cohort, these finding lead the possible use of RNA expression analysis as new diagnostic test for paratuberculosis. Such a signature could allow early interventions to reduce the sanitary and economic burden, and to reduce the risk of infection spreading.

In the next sections a description of the dataset and of main DNetPRO results will be discussed. Further information can be found in the original paper [59].

1.6.1 Dataset

Paratuberculosis or Johne's disease (JD) in cattle is a chronic granulomatous gastroenteritis caused by infection with *Mycobacterium avium subspecies paratuberculosis* (MAP). JD is present worldwide, is a welfare issue and causes significant economic losses. Cattle are usually infected as young calves but typically do not show clinical signs before 24 months of age, however not all infected animals progress to clinical disease. JD is not treatable, therefore the early identification and isolation of infected animals, before they start shedding the bacteria, is a key point to reduce its incidence in cattle herds worldwide. In addition, an association between MAP and Crohn's disease (CD) in humans has been suggested and intensively explored. Given the economic losses and welfare concerns for livestock, and possible human health risk, the research interest in JD has been driven by the substantial difficulty in early diagnosis of infected animals and the exploration of potentially new diagnostic techniques.

The dataset used in this work was previously discussed and generated by some of the authors of the original paper. In detail, the dataset used comprised 15036 transcripts from 15 samples, classified as "serologically negative non exposed cows/healthy" (5 samples, labeled as NN), "serologically negative exposed cows/ infected" (5 samples, NP) and "serologically positive cows/clinical" (5 samples, PP). Only transcripts with non-zero measures for all samples were considered, reducing the dataset to 13529 transcripts.

All data generated or analyzed during this study is available upon request, furthermore all transcript counts per sample are given as supplementary information files of the original paper.

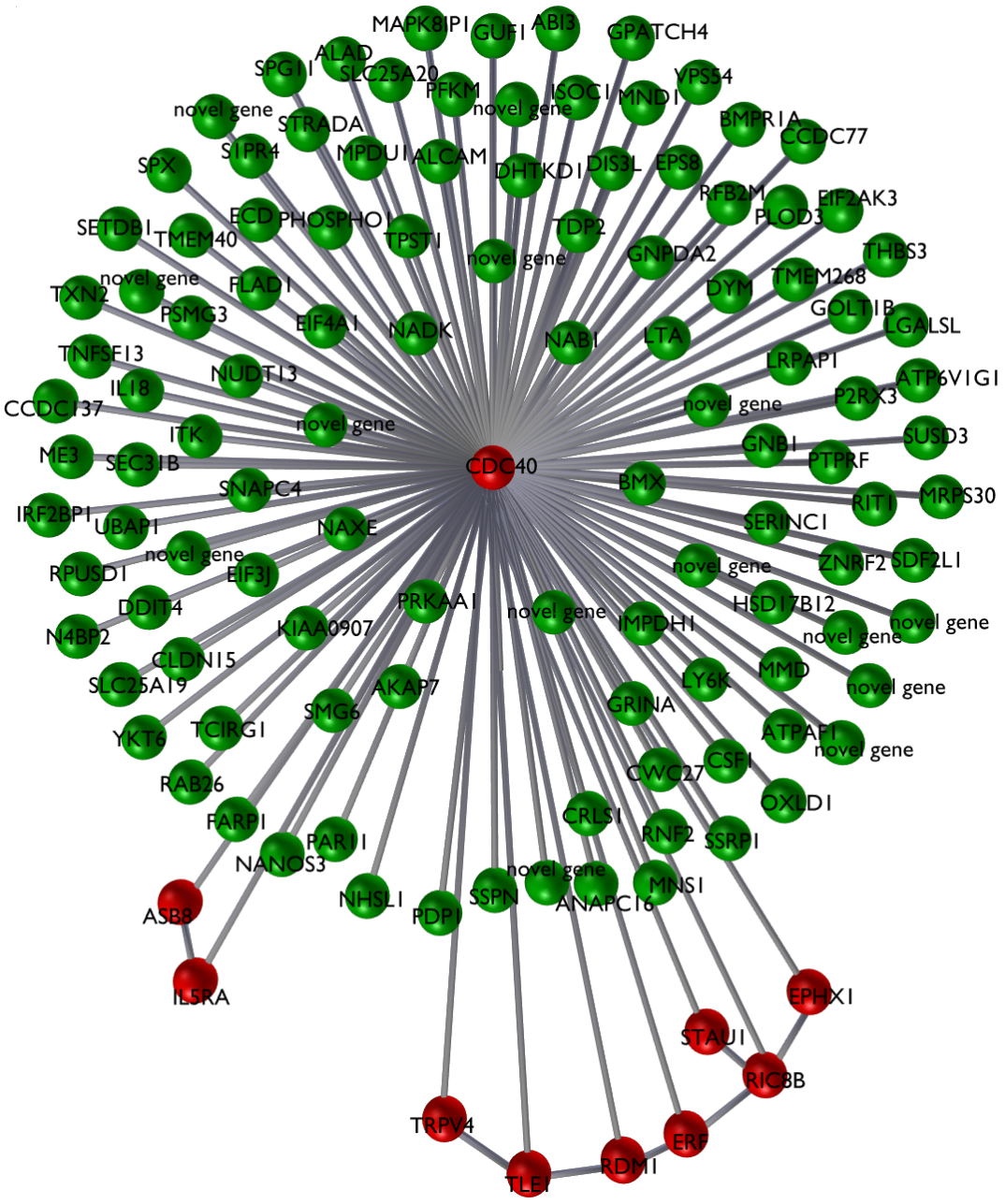
1.6.2 Bovine Signature

In the context of high-throughput data analysis, a challenge is the search for an optimal choice of variables (a "signature") to classify groups of samples or regress trends with optimal performance and minimum dimensionality. Usually high-throughput omics data (e.g. transcriptomics, ge-nomics, methylomics) provide datasets with few tens to hundreds of samples, and often 1000 times larger numbers of variables. The objective of dimensionality reduction through the choice of an optimal signature is twofold: 1) the identification of relevant variables, that should separate the signal from the noise (i.e. variables not significantly associated to, or descriptive of the studied process); 2) in a practical context, it is important to establish future diagnostic criteria that can be implemented in cheap and simple toolkits, such as PCR cards or dedicated microarray chips, that usually test a small number of transcripts (ranging from tens to hundreds, at most). The quantity of

[19] The enzyme-linked immunosorbent assay. It is a common diagnostic tool as well as a quality control check in various bio-medical industries and in medicine.

samples compared to the available features of this work, join with the final purposes of this kind of analysis, set the well-known ill-posed problem conditions for which the DNetPRO algorithm was thought.

Since the number of sample is drastically small no robust cross-validation procedure can be applied. So we focused on the identification of a putative gene-signature able to discriminate between NN and NP samples, leaving the PP data as validation set. In this case we hypothesize that PP samples will be classified more closely with NP sample rather than NN as exposed, possibly infected samples, should be more similar to positive samples, than to negative controls.



a

Starting from the top-performing couples of transcripts, we obtained an initial signature of 123 different transcripts (Fig 1.13 (a), all the nodes), capable to correctly classify 4 out of the 5 NN samples (80%) and all 5 NP samples (100% performance). The average

[20] The figure was generated using a custom network visualizer described in Appendix C - BlendNet.

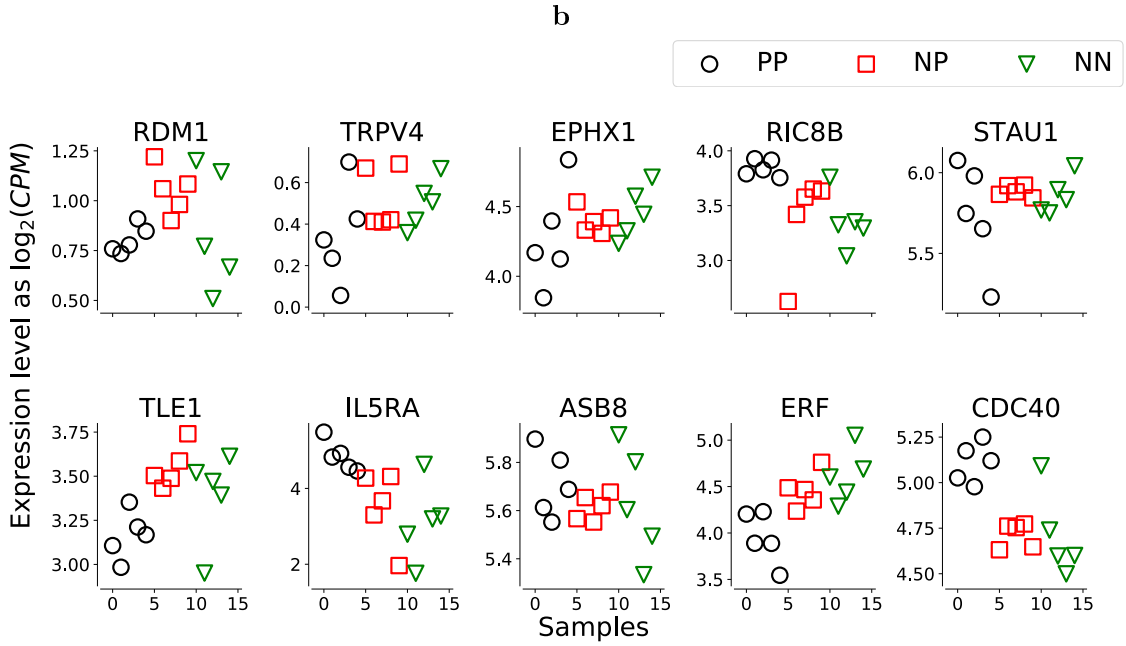


Figure 1.13: (**a**) Plot of the 123-transcript network, with a detail of the 10-probe signature (red nodes)[20]. (**b**) Transcript levels for the 10 genes belonging to the classification signature identified by the combinatorial discriminant analysis (CDA). Some transcripts (EPHX1, RIC8B, IL5RA, ERF, CDC40) show a clear trend between 5 animals serologically positive to the ELISA test for MAP (PP), 5 exposed serologically negative (NP) and 5 serologically negative unexposed control animals (NN).

performance was therefore 90% with Matthews correlation coefficient MCC = 0.82. Processing the 123-transcript network by removing all pendant nodes (i.e. removing all single transcripts belonging to only one best-performing couple) we obtained a final signature with 10 transcripts with a 100% performance classifying all NN and NP samples (Fig 1.13 (a), only red nodes). As it can be seen, many nodes are directly connected to the central node (belonging to the 10-transcript signature), while only the 10 transcripts of the signature are also connected between each other.

Principal Component Analysis of the 10-transcript signature showed that in many cases there was a progressive increase or decrease in the transcript levels when passing from a healthy (NN) sample to a positive (PP) sample, passing through the infected (NP) sample class. Fig 1.13 (b) shows the expression levels of the transcripts belonging to the signature for all samples.

To further validate the goodness of the signature, we generated 10000 different signatures with 10 randomly chosen transcripts, and then applied a Leave-One-Out cross validation procedure to classify all 15 samples with these signatures. Comparing the performance of the random signatures with the true 10-transcript signature, only 50 of these signatures (corresponding to 0.5% of the random signature distribution) produced better performance than our signature in terms of classification performance, confirming its high significance.

We even characterized the possible biological role of the signature genes, among the significantly differentially expressed genes, the cell division cycle 40 gene (CDC40) showed the smallest fold change between classes. However in the identified signature the CDC40 gene is the most central node associated with the health status of the animals related to JD. CDC40 was also under expressed in the NP and PP groups, compared with the NN

group and it has been shown to be involved in clathrin medated endcytosis from a biological point-of-view. Clathrin is the best characterized coat protein involved in the endocytosis process, specifically in receptor-mediated-internalization. *Mycobacterium paratuberculosis* enters the host macrophages, its primary target cell, and manages to survive within their phagosome. It is possible that the under-expression of CDC40 in infected and sick animals compared to unexposed animals may be associate with down regulation of macrophage genes post mycobacterial invasion, facilitating the survival of the pathogen with the host target cell.

Interestingly within the set of 10 discriminating transcripts, in addition to CDC40, others show links with immune response mechanisms, these include IL5RA, ERF and TRPV4. These genes potentially have functions related to the biology of progression of JD. Also for the other genes of the final 10-transcript signature a possible biological interpretation related to JD was given (see the original paper for further descriptions).

In conclusion, the DNetPRO algorithm identified a set of 10 genes, the expression levels of which could discriminate between the exposed and sero-converted animals. These finding lead the possible use of RNA expression analysis as new diagnostic test for JD. In particular the approach may be able to identify infected animals prior to sero-conversion, prior to a positive ELISA test result. However, further tests for specificity and validation in a larger cohort are required.

Chapter 2

Deep Learning - Neural Network algorithms

In the first chapter we have discussed about the difficulties on extracting information from a huge amount of data, and we have proposed a novel feature selection algorithm to face these problems. Those kind of applications go under the wide research field of Machine Learning. Machine learning algorithms are closely related to a statistical interpretation of the available data. With the increasing availability of computational power and data it is not always possible to tune and build an accurate model able to describe the heterogeneity of our samples. Many everyday problems involve very complex tasks, and we are interested on models able to solve many tasks at the same time. From a machine learning point-of-view this can be achieved building pipelines, i.e work-flows made by multiple steps of processing, which aim to simulate as much close as possible the human intelligence. This leads us into the Deep Learning research field, in which very computational expensive models have been built to face general purpose problems, often related to real time applications.

The description of a deep learning model is quite often given by a Neural Network architecture, i.e a more or less complex pipeline of functions which takes in input a sample and it applies a series of transformations and filters to obtain the required result. All these pipelines are very computational expensive and they require appropriate optimization strategies.

In this chapter we introduce some of the most common functions related to deep learning applications, giving a very fast mathematical explanation of them and carefully focusing on their numerical issues and solutions. We start from an introduction about general Neural Network models up to some of modern deep learning models, involving object detection, image segmentation and image super resolution. In particular, we describe two custom libraries (NumPyNet and Byron) developed by the author of this thesis, for educational and analytical purposes, respectively. Both libraries are released with MIT license and the codes are publicly available on my Github page (Byron and NumPyNet). These libraries have already used in several applications and in the last sections we show some of the obtained results[1].

In the last section of this chapter we introduce a different kind of Neural Network model, the Replicated Focusing Belief Propagation (rFBP) model. This model has solid physical and statistical bases and we discuss about its novel optimized implementation, available on my Github (Replicated Focusing Belief Propagation) and released under MIT license. This model differs from standard deep learning neural networks changing the updating rule and we show its first application on real data.

[1] Both NumPyNet and Byron libraries have been developed with the collaboration of master degree students and several thesis have their applications as core arguments.

2.1 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 using 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 (validation); otherwise we simply have an extrapolation of our data (prediction).

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 have became the multiple open-source Python libraries available on-line as Tensorflow [1], Pytorch [68] and Caffe [50]. 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 should be mentioned the darknet project of Redmon J. et al. which has created a sort of standard in object detection applications using a pure Ansi-C library[2].

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 implementation. The numerical aspects will be traced following two libraries developed by the author: NumPyNet library [20] and Byron library [21].

NumPyNet is 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 [66] (a base 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 cited in them. 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 are certainly efficient from a computational point-of-view and the numerous wrappers (like *Keras* library) guarantee an extremely simple user interface. On the other hand, the deeper functionalities of the code and the implementation strategies used are unavoidably hidden behind tons of code lines. In this way the user can perform complex computational tasks using the library as black-box package. NumPyNet wants 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 allows us to give a first numerical analysis of the model functions and, moreover, to show the results of each function on an image to better un-

[2] 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 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 libraries developed in this work are all inspired on it. The large part of our work has been related to a deep optimization of this library either in terms of functionality and issues either in terms of computational performances.

derstand the effects of their applications on real data[3]. Each NumPyNet function was tested against the equivalent Tensorflow implementation, using an automatic testing routine through PyTest [65]. The full code is open-source on the Github page of the project. Its installation is guaranteed by a continuous integration framework of the code through Travis CI for Unix environments and Appveyor CI for Windows OS. The library supports Python versions ≥ 2.6[4].

As term of comparison we discuss the more sophisticated implementation given by the Byron library. Byron (*Build YouR Own Neural network*) library is written in pure C++ with the support of the modern standard C++17. We deeply use the C++17 functionality to reach the better performances and flexibility of our code. What makes Byron an efficient alternative to the competition is the complete multi-threading environment in which it works. Despite the most common Neural Network libraries are optimized for GPU environments, there are only few implementations which exploit the fully functionality of a multiple CPUs architecture. This gap discourage multiple research groups on the usage of such computational intensive models in their applications. Byron works in a fully parallel section in which each single computational function is performed using the full set of available cores. To further reduce the time of thread spawning, and so optimize as much as possible the code performances, the library works using a single parallel section which is opened at the beginning of the computation and closed at the end[5].

The Byron library is released under MIT license and publicly available on the Github page of the project. The project includes a list of common examples like object detection, super resolution, segmentation, 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 about its installation is provided; the installation can be done using CMake, Make or Docker and the Python version is available with a simple setup.py. The testing of each function is performed using Pytest framework against the NumPyNet implementation (faster and lighter to import than Tensorflow).

We use Byron library as term of comparison with other common libraries used for Neural Network models and for each function we have tested 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 will introduce the fundamental Neural Network model, the so-called *Simple Perceptron*. From the simplest model we will add complexity and layers to overcome the relative problems (mathematical and numerical), introducing the main functionalities of the modern Neural Network architectures.

2.1.1 Simple Perceptron

The fundamental unit of each Neural Network model is the *simple Perceptron* (or single neuron). The *Perceptron* is the simpler mathematical model of a biological neuron and it is based on the Rosenblatt [76] 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 [46] (H-H model), we have an action potential, i.e the output of the neuron, given by

[3] 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.

[4] The library provides also an Image object to load and process images. The object is based on OpenCV API [11]. OpenCV does not yet support Python versions 2.7 and 3.3 so the whole NumPyNet package does not work on these two versions of Python. You can just exclude the Image script from the package or use a novel wrap based on different library (e.g Pillow).

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

$$y = \sigma\left(\sum_{i=1}^{N} w_i x_i + w_0\right) \tag{2.1}$$

where σ is the activation function, w_i are the synaptic weights and x_i are the inputs. The w_0 coefficient identifies the bias of the linear combination and it is left as parameter to be tuned 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

$$w_i(\tau + 1) = w_i(\tau) + \gamma(t - y)x \tag{2.2}$$

where γ is the gain or step size ($\gamma \in [0, 1]$) and t is the desired output. In other words we have to compute firstly the difference between the current output and the desired one, i.e the error or cost function or loss function[6], 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[7] by the *Perceptron*[8]. Moreover, we can manage only two classes since an hyper-plane divides 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 we can treat the problem using a continuous cost function. In this case we define it as

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N} (t_i - y_i)^2 \tag{2.3}$$

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

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w} \tag{2.4}$$

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

$$\Delta\mathbf{w_i} = -\gamma\frac{\partial E}{\partial w_i} = -\gamma\sum_{i=1}^{N} (t_i - y_i)(-x_i) \tag{2.5}$$

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

[6] 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 sections.

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

[8] A classical example of learning problem is given by the XOR logical function. Since the XOR output is not linearly separable the Perceptron could not converge.

To implement this kind of model into a pure Python application we do not need extra libraries, but we can just use the native keywords of the language. A possible implementation of this model was developed and release in an on-line gist. In this simple snippet we examine the functionality of the Simple Perceptron model across different logical functions and we proved its fast convergence on linear separable datasets[9]. An equivalent C++ implementation of the model is also released and it can be found in this other gist.

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

2.1.2 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 interactions, in which the output of a neuron feeds-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

$$y = \sigma\left(X \cdot W + W_0\right) \tag{2.6}$$

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

$$\delta W = \delta W + X^T \cdot \left(\frac{\partial f(y)}{\partial y} \cdot \delta^l\right) \qquad \delta W_0 = \sum_{i=0}^{m} \frac{\partial f(y)}{\partial y_i} \cdot \delta_i^l \tag{2.7}$$

where, also in this case, we simply pass to the matrix formalism and we convert the discrete 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[10].

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

Fully connected Neural Networks overcome the above told *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 is 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 are lost in the dimensionality reduction performed by the hidden units.

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

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

- 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 information 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 into account 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 problem and its algorithmic complexity can be hardly reduced under $O(N^3)$[11]. 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 it leads 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 is given by a series of sequential index locations (the row elements are 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 a 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 are given by a discontinuous portion of memories (the column elements are 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 have a *cache-miss* at each iteration[12].

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 issue is to perform a transposition of the second matrix to obtain a row-row matrix product[13]. 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[14].

Following back to our Neural Network implementation we can obtain the output values using the above technique. Moreover, we can assume from the beginning that the transposition of weights matrix and so remove the $O(N^2)$ calculus from the matrix product.

[11] The complexity is often given in the assumption of only square matrices ($N \times N$) involved in the computation. For no-square matrix the algorithmic 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 implementations of the algorithm are able to reduce the algorithmic complexity (e.g Strassen algorithm) but neither implementation is able to overcome the $O(N^{2.7})$ complexity up-to-now.

[12] 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).

[13] 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.

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

This simple (but carefully studied) optimization allows 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.

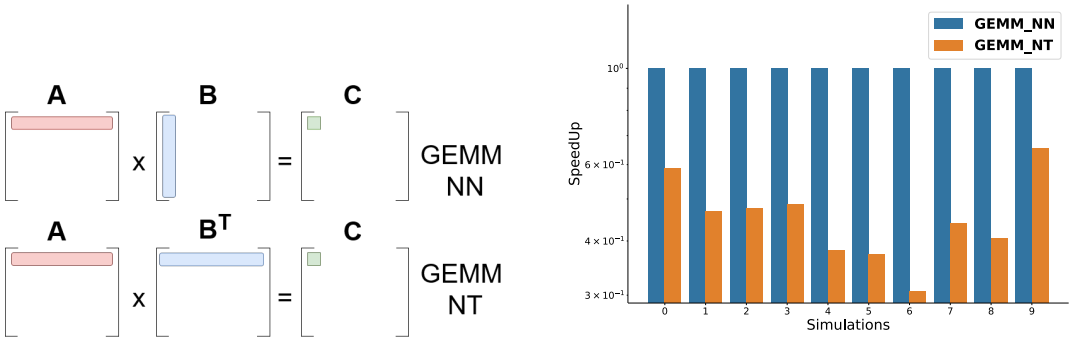


Figure 2.1: GEMM algorithms time performances. GEMM NN: matrix multiplication considering both the matrices in "normal" format, i.e $A \cdot B$. GEMM NT: matrix multiplication considering the first matrix in "normal" format and the second one transposed, i.e $A \cdot B^T$. We perform 100 tests of 1K 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 layer we implement both the matrix product cases to compare their performance results. We tested the two implementations inside Python using the einsum function provided by the Numpy package. In particular, we evaluated the time-performances over 1000 applications of the two GEMM (*Generalized Matrix Multiplication*) functions (GEMM NN, i.e considering both matrices with "normal" shapes; GEMM NT, i.e considering the first matrix as "normal" and the second transposed) considering matrices of shapes (100×100). We performed 500 run and we saved the minimum time obtained over 10 realizations. In Fig. 2.1 we show the results rescaled by the mean time (over the 500 realizations) of the GEMM NN algorithm (reference). As can be seen in Fig. 2.1, the speedup of the GEMM NT matrix is evident and it is always faster than GEMM NN algorithm, with a maximum of $3.2\times$ in the speedup.

In the Byron library 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 a faster GEMM algorithm (especially for dimensions proportional to powers of 2, which are very common in neural network models).

2.1.3 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[15] can be computed as the dot product of the input and neuron weights (see previous section); in this case the output values range from $-inf$ to $+inf$ and they are equivalent to 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 pattern data. From a biological point-of-view the activation functions model the on/off state of a neuron in the output decision process.

[15] 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	$f(x) = \frac{x}{\|x\|+1}$	$f'(x) = \frac{1}{(\|f(x)\|+1)^2}$
Elliot	$f(x) = \frac{\frac{1}{2}*S*x}{1+\|x*S\|} + \frac{1}{2}$	$f'(x) = \frac{\frac{1}{2}*S}{(1+\|f(x)*S\|)^2}$
SymmElliot	$f(x) = \frac{S*x}{1+\|x*S\|}$	$f'(x) = \frac{S}{(1+\|f(x)*S\|)^2}$

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

Many activation functions have been proposed during the years and each one has its characteristics, but not an appropriated application field. The best activation function to use in a given situation (to a particular problem) is still an open question. Each one has its pros and cons in some situations, so each Neural Network library implements a wide range of them and it leaves to the user to perform his own tests. In Tab. 2.1 we show the list of activation functions implemented in our NumPyNet and Byron 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 back-propagation of the error gradients.

As can be seen in Tab. 2.1 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 test image. This can be easily done using the example scripts inserted into our NumPyNet library. In Fig. 2.2 the effects of the previously described 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 have been rescaled $\in [-1, 1]$ before the input to the functions.

Figure 2.2: 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.

From the results showed in Fig. 2.2 we can better appreciate the differences between the several mathematical formulas: a simple Logistic function does not produce evident effects on the test image, while a Relu activation tends to overshadow the image pixels. This feature of the Relu activation function is very useful in Neural Network model and it also determines 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 function is, in fact, the most used into the modern Neural Network models. Its diffusion is imputed to its numerical efficiency and to the benefits it brings [37]:

- 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 modify 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 information. The possibility to have null values in output (ref Tab. 2.1) allows a better representation of the dimensions inside the network.

- Sparsity: sparsity representation of data is exponentially efficient in comparison to dense one, where the exponential power is given by the number of no-null features [37].

- 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 have chosen to use Relu activation function in the major part of the layers.

2.1.4 Convolution function

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

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{2.8}$$

In signal processing, this operation is also called *crossing correlation* ad it is equivalent to the *autocorrelation* function computed at 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 have a 2D discrete version of the formula given by:

$$\begin{aligned} C &= k * I \\ C[i,j] &= \sum_{u=-N}^{N} \sum_{v=-M}^{M} k[u,v] \cdot I[i-u, j-v] \end{aligned} \tag{2.9}$$

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

The use of CNN in modern image analyses 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[16]. Moreover, if we consider detection problems, i.e the problem of detecting a set of features (or an object) into a larger pattern, we want a system ables to recognize the object regardless of where it appears into the picture. In other words, we want that our model would be independent by simple translations.

Both the above problems can be overcame 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 a signal through the application of relevant filters.

The main parameters of this function are given by the input dimensions and the filter/kernel dimensions, i.e the number of weights 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 Networks) we can sophisticate it, playing with the possible movements of the filter mask. In particular, aside the kernel mask-size, we can 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 further reduce the output dimensions.

Given this theoretical background we can implement the convolution function in many different ways, using different mathematical approaches: a study about 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 function as described in the above equation. This version is certainly the easier to implement, but its computational performances are so worst that, for sake of brevity, we excluded it from our tests[17].

[16] 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.

[17] 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.

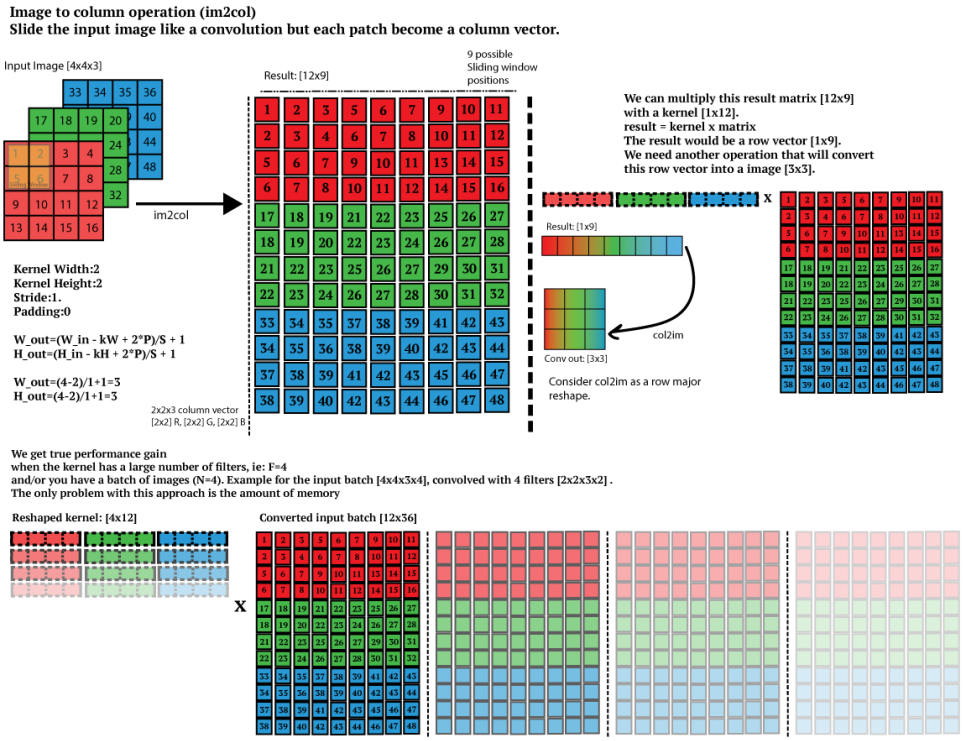


Figure 2.3: im2col algorithm scheme using a 2×2 filter on a image with 3 channels. At the end of the im2col algorithm the GEMM is performed between weights and input image.

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 several matrix products. We can re-arrange the involved data to optimize this computation and evaluate a single matrix product: this re-arrangement is called im2col (or im2row) algorithm. The algorithm 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[18]. In this way we can immediately apply our GEMM algorithm on the full image. In Fig. 2.3 the main scheme of this algorithm is reported. This algorithm optimizes the computational efficiency of the GEMM product but we have to store a lot of memory for the input re-organization in payback.

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, with several efficient implementations provided in literature. One of the most efficient one is provided by the FFTW (*Fast Fourier Transform in the West*) library [35]: FFTW3 is an open source Ansi-C subroutine library for computing the Discrete Fourier Transform (DFT) in multiple dimensions, without constraints in input sizes or data types. The library is not only computationally accurate, but it also provides an efficient parallel version for multi-threading applications.

A further implementation kind is given by linear algebra considerations (very closed to

[18] 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.

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:

$$\text{img} = \begin{bmatrix} d0 & d1 & d2 & d3 \end{bmatrix} \qquad \text{weights} = \begin{bmatrix} g0 & g1 & g2 \end{bmatrix} \tag{2.10}$$

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

$$\text{img} = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix}, \qquad \text{weights} = \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} \tag{2.11}$$

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:

$$\text{output} = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1 + m2 + m3 \\ m2 - m3 - m4 \end{bmatrix} \tag{2.12}$$

where

$$\begin{aligned} m1 &= (d0 - d2)g0 & m2 &= (d1 + d2)\frac{g0 + g1 + g2}{2} \\ m4 &= (d1 - d3)g2 & m3 &= (d2 - d1)\frac{g0 - g1 + g2}{2} \end{aligned} \tag{2.13}$$

where we can easily notice that the two fractions in $m2$ and $m3$ involve only weight quantities and thus they could 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: the reduction of computational expensive MUL operations by a factor 1.5x is very significant[19]. 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 has been widely proved for CNNs, 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 straightforward[20].

We tested the computational-time of each algorithm 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 input dimensions and number of filters. In Fig. 2.4 we show the results of our simulations using the im2col values as reference[21].

In all our simulations we found a visible speedup using the Winograd algorithm against the other two algorithms: for small dimensions we obtained more than 5x against the im2col and 25x against the fftw implementation. The worst algorithm is certainly the fftw

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

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

[21] 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 reference for our work.

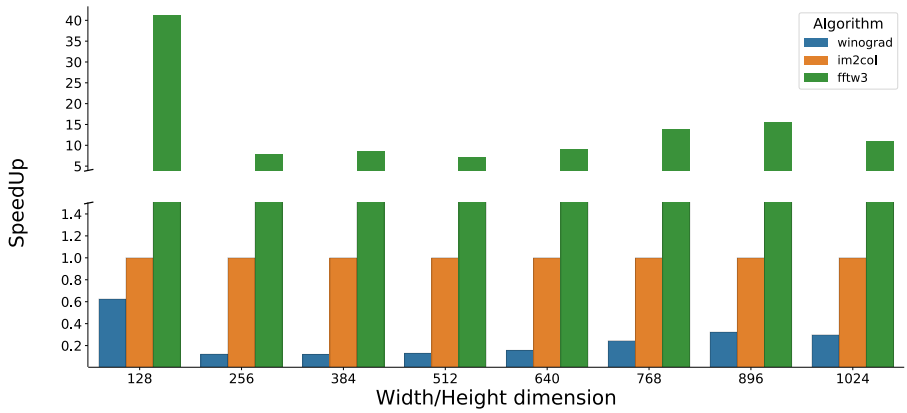


Figure 2.4: Time performances of different convolution algorithms: im2col (orange, reference), FFTW3 (green, fast Fourier transformation using the FFTW3 library) and Winograd (blue). The values are normalized according to the im2col results since it is the most common convolution algorithm. The tests 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.

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.

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 constraints 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 sizes equal to 3 and unitary strides.

2.1.5 Pooling function

Output Neural Network feature maps often suffer of sensitivity about features 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-sampling and they reduce the spatial dimension (but not depth) of the input. Their use represents an important computational performance improver (less feature, less operations) and a useful dimensionality reduction method. The reduction of features 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 2.1.4 for more information). The method parameters 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 underlying

the kernel. In Fig. 2.5 are shown some results obtained by an average pooling, with different kernel sizes. Also in this case the test was obtained using our NumPyNet library.

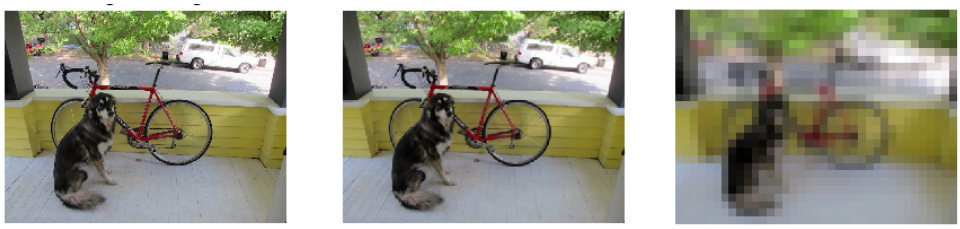


Figure 2.5: Average Pool functions applied on a testing image. **(left)** The original image. **(center)** Average Pool output obtained with a kernel mask (3×3). **(right)** Average Pool output obtained with a kernel mask (30×30).

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 will discuss about the optimized implementation provided into NumPyNet.

In the previous sections we introduced the im2col algorithm which is an efficient method to reorganize the input data. The same algorithm can also be applied for Pooling layers, evaluating the Pooling function (avg, max, etc.) on each row of the rearranged matrix. The implementation of the im2col algorithm in Python requires the evaluation of multiple indexes using complex formulas. Since the NumPyNet 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 it means that the data of both objects (original array and the viewed one) are shared and thus no copies are created. In particular, we can use the deeper functions of the Numpy package to create a reorganization of our data according to the desired output[22]. In the following code we show our implementation of the Average Pooling layer:

Listing 2.1: NumPyNet version of AvgPool function

```
import numpy as np

class Avgpool_layer(object):

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

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

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

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

    # Shape of the final view
```

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

```
20      view_shape = (batch, 1 + (w - kx)//st1, 1 + (h - ky)//st2) + input.
        shape[3:] + (kx, ky)
21
22      # strides of the final view
23      strides = (batch_stride, st1 * s0, st2 * s1) + input.strides[3:] + (s0,
         s1)
24
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 snippet 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 rearrangement we can easily compute the desired pooling function (average in this example) according to the appropriate axis. 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).

2.1.6 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 2.1.3). Moreover, in this way 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 constantly changes during training. This behavior produces a slowdown in the training convergence because each layer has 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 is limited, we can meet problems during the generalization phase, i.e the validation of our model using new data, to new samples if they belong 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 task: if we train a Neural Network model using gray-scale images, we can find generalization issues 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[23] [49]. In this way we can ensure more stability to the extracted features [54] during the training and a faster convergence.

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

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad {\sigma_B}^2 = \frac{1}{m-1}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2.14}$$

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:

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{{\sigma_B}^2 + \epsilon}} \tag{2.15}$$

where we add an extra ϵ in the denominator for numerical stability[24]. After this common rescaling, we apply a shift-scaling to the previous results:

$$y_i = \gamma\hat{x_i} + \beta \tag{2.16}$$

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:

$$\delta\beta = \sum_{i=1}^{m} {\delta_i}^l \qquad \delta\gamma = \sum_{i=0}^{m} {\delta_i}^l \cdot \mu_B \tag{2.17}$$

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

$${\delta_i}^{l-1} = \frac{m \cdot \delta\hat{x_i} - \sum_{j=1}^{m}\delta\hat{x_i} - \hat{x_i}\cdot\sum_{j=1}^{m}\delta\hat{x_i}\cdot\hat{x_i}}{m\cdot\sqrt{{\sigma_B}^2 + \epsilon}} \tag{2.18}$$

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

The Byron library is 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 not 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 differentially manage the dimensions of the involved arrays. A standalone implementation of the BatchNorm function required a rearrangement of its functions and it has provided into our Byron and NumPyNet libraries. This was one of the various improvements provided by Byron against other darknet-like libraries.

[23] 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.

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

Other common regularization techniques are given by the regularization of neuron outputs with penalty loss functions. Classical examples are given by 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.

2.1.7 Dropout function

Many times along this work we have talked about the *over-fitting* problem. 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 fits our data. In this case the model flexibility is minimum and new additional data points difficulty lie on the same curve. In other words, we have tuned each model parameter according to the given dataset, but we have completely lost the possibility of generalization.

In Neural Network models we have to manage a large quantity of parameters and it is quite easy to stumble on this problem. Possible workaround could be given by the regularization techniques described in the previous section (ref. 2.1.6 for further information) or by a Dropout function. This second function simply drops out some neuron units into a Neural Network during the training phase. Ignore some neurons means that they will not be considered during a (single) forward/backward step. So, given a set of neurons, we have a probability p to update (keep) the neuron and $1-p$ to ignore (remove) it. In this way, we can reduce the co-dependency of nearest neurons inside the network and reduce the possibility of over-fitting.

The above description lead us to a straightforward implementation of the algorithm into the NumPyNet library (ref. 2.2).

Listing 2.2: 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 reproduces the theoretical formulation given. After the initialization of the private object variables, the forward function generates a set of random

positions and it applies 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 inverts 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 lead 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 construction) shared among all the available threads. Thus, if we simply create a random number generator we have to face the thread-concurrency. As consequence, the random number generated will not be independent but (most probably[25]) repeated by each thread. The simpler workaround, implemented into the Byron library, is given by assigning a random number generator to each thread (with its own seed initialized 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).

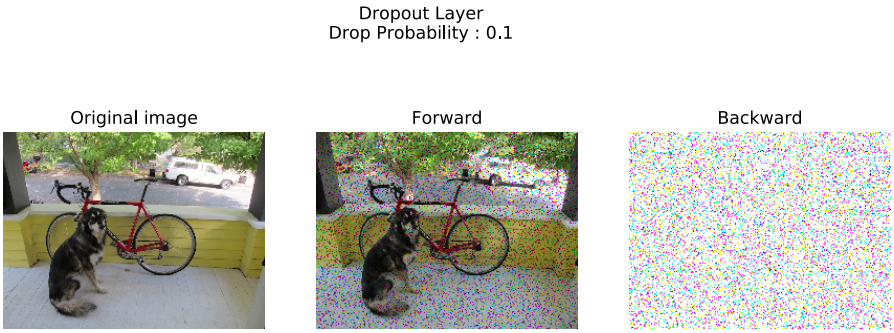


Figure 2.6: Dropout function applied on a testing image. The 10% of image pixels are turned off by the forward function. The output of the back-propagation is computed considering a uniform (white pixels) image: in this way we can notice that only the previously activated pixels allow the gradient passing. In this way the dropout function allows to update only a part of the model parameters (turned on pixels).

As visualization example, we can use our simple test image and apply the dropout transformation (see Fig.2.6). Our input image shows many pixels turned off according to the given probability, as expected. On the other hand, the backward output turns on only the same pixel: for visualization purposes we manually set the gradient to a uniform value.

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

2.1.8 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 Neural Network structures is a reduction in accuracy after reaching a maximum, the so-called *degradation problem*. This accuracy reduction does not arise from an over-fitting problem but it is due to numerical

[25] 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.

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

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

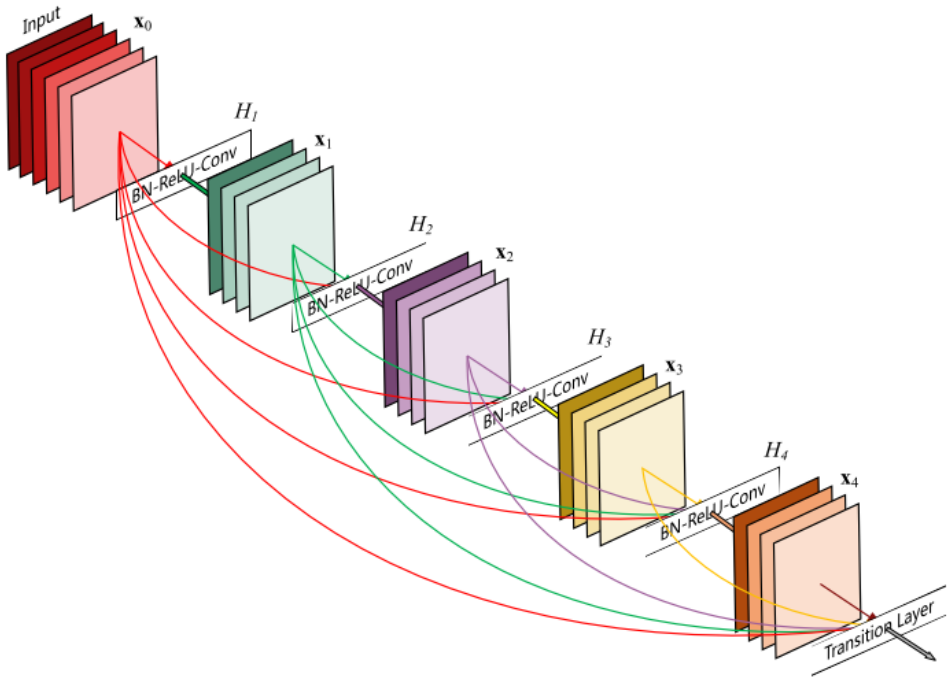


Figure 2.7: Scheme of shortcut connections into a deep learning model. Each colored line connects the previous layer block to the following one. The output combination can be customized but the most used one is a simple linear combination of them. A particular attention must be paid with the dimensions management.

We can obtain a simple solution to this issue 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. 2.7. The authors of [43] 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:

$$H(x) = F(x_1) + x_2 \tag{2.19}$$

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 leads to the ResNet (Residual Neural Network) models era, in which a key role is played by the object detection models. A wide range of modern deep learning architectures uses this kind of connections and in this way they can reach a large number of layers: famous examples about them are the VGG models and the ResNets. We have done a large use of these connections also in the models described in the next sections, either for object detection purposes (ref. 2.3), Super Resolution (ref. 2.2) and mostly in our image segmentation (ref. 2.4) application. This kind of functions are becoming so popular into 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 connections.

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 objects in which a shortcut connection simply combines the output of two of its elements. We met more problems when we translated this idea into C++. The C++ language is more rigid with the data types 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 be reached using 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 allows to combine and reinterpret different data types into a single object. The most important consequence in the usage of this kind of data type is that we can easily jump to one type to an other using constexpr statements, which (by definition) are solved at compile time. Besides the particulars involved into this kind of implementation, it is important to notice that the difference between the two solution is the same between compile-time and run-time, i.e one-for-all against at-every-run. The Byron library widely uses 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 objects and templates we can easily implement a shortcut connection also in C++ as can be seen on the on-line version of the code.

2.1.9 Pixel Shuffling

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 to restore a high-resolution image from a single low-resolution one (see section 2.2 for further details).

The first SISR Neural Networks started 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, fed the model which aimed to increase the resolution and fixed its details. In this way the amount of parameters, and moreover the computation required by the training section, increased (by a factor equal to the square of the desired up-sampling scale), despite the required image processing was smaller. To overcome this problem a

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

$$H(x) = \alpha x_1 + \beta x_2 \tag{2.20}$$

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

Pixel Shuffle transformation, also known as *sub-pixel convolution*, was introduced [82]: in this work the authors proved 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 reorganizes the low-resolution image channels to obtain a bigger image with few channels. An example of this transformation is shown in Fig. 2.8.

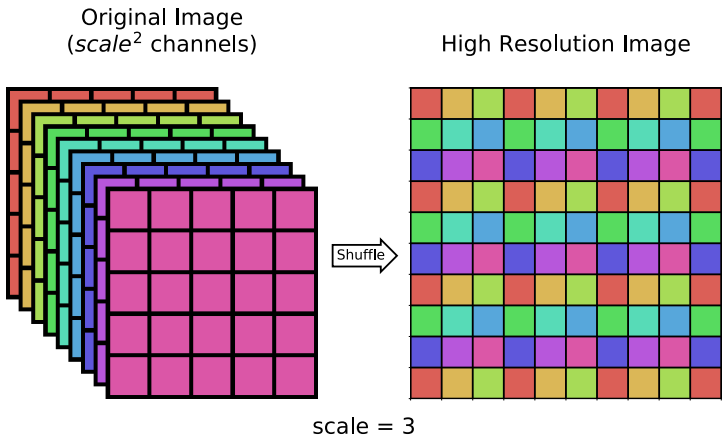


Figure 2.8: Pixel Shuffle transformation. On the left the input image with $scale^2$ ($:= 9$) channels. On the right the result of Pixel Shuffle transformation. Since the number of channels is perfect square the output is a single channel image with the rearrangement of the original ones.

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 increases 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, reorganizes the set of features into a single bigger image, i.e the desired output in a SISR application. The features processing steps, which generally are faced 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 ables 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 need to use a sequence of transposition and reshaping to rearrange the input values. The following snippet shows the NumPyNet version of this

[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 $C \times H \times W$.

algorithm.

Listing 2.3: NumPyNet version of Pixel-Shuffle function

```
import numpy as np

class Shuffler_layer(object):

  def __init__(self, scale):

    self.scale = scale
    self.scale_step = scale * scale

    self.batch, self.w, self.h, self.c = (0, 0, 0, 0)

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

  def _phase_shift(self, input, scale):
    b, w, h, c = input.shape
    X = input.transpose(1, 2, 3, 0).reshape(w, h, scale, scale, b)
    X = np.concatenate(X, axis=1)
    X = np.concatenate(X, axis=1)
    X = X.transpose(2, 0, 1)
    return np.reshape(X, (b, w * scale, h * scale, 1))

  def _reverse(self, delta, scale):
    # This function apply numpy.split as a reverse function to numpy.concatenate
    # along the same axis also

    delta = delta.transpose(1, 2, 0)

    delta = np.asarray(np.split(delta, self.h, axis=1))
    delta = np.asarray(np.split(delta, self.w, axis=1))
    delta = delta.reshape(self.w, self.h, scale * scale, self.batch)

    # It returns an output of the correct shape (batch, in_w, in_h, scale**2)
    # for the concatenate in the backward function
    return delta.transpose(3, 0, 1, 2)

  def forward(self, input):

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

    channel_output = self.c // self.scale_step # out_c

    # The function phase shift receives only in_c // out_c channels at a time
    # the concatenate stitches together every output of the function.

    self.output = np.concatenate([self._phase_shift(input[:, :, :, range(i, self.c, channel_output)], self.scale)
                                  for i in range(channel_output)], axis=3)

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

  def backward(self, delta):

    channel_out = self.c // self.scale_step # out_c

    # I apply the reverse function only for a single channel
    X = np.concatenate([self._reverse(self.delta[:, :, :, i], self.scale)
                                      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 rearrangement 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 reordering of channel sequences 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 into account in 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 ones in terms of CPU clock-cycles. The old versions of OpenMP multi-threading library forced the users to spend time in the evaluation of "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.

2.1.10 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 functions 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. We have to pay attention to chose the right one in our problems. In particular, we have to take into account 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

[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 factors 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.

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.

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 2.1.6). Its mathematical formulation is quite simple and it is given by

$$MSE = \frac{\sum_{i=1}^{N} (y - t)^2}{N} \tag{2.21}$$

where we follow the nomenclature given in 2.1.1 and N is the number of outputs, 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 and numerical point-of-view. The possible output values range 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.

$$MAE = \frac{\sum_{i=1}^{N} |(y - t)|}{N} \tag{2.22}$$

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 training. A simple possible workaround is given by the introduction of a shrinking parameter, given by a dynamic learning rate, when we move closer to the minimum.

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

$$CrossEntropyLoss = -(y \cdot \log(t) + (1 - y) \cdot \log(1 - t)) \tag{2.23}$$

This function just multiplies 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. 2.1.3) which constrains the output of the model in the desired interval. For this reason

[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 the actual label is equal to 0 the first half is null.

in our implementation of the algorithm we have chosen 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*.

$$CategoricalCrossEntropyLoss = -\sum i = 1^{N} (y \cdot \log(t)) \tag{2.24}$$

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 have chosen to implement this loss function in a separated object associated to the softmax transformation.

Many other loss functions can be mentioned to overcome different kind of problems. The list of presented loss functions is related to the implementation of the darknet-like libraries which are ported also into the NumPyNet and Byron libraries, i.e either in Python and C++. NumPyNet and Byron libraries provide an optimized version of these functions (fixing also some darknet issues) and they include also other kind of loss functions to improve the library usability. 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 improvement has been performed from a numerical point-of-view: many mathematical formulas need expensive math operations as logarithms and trigonometric functions. An efficient (but approximated) math formula has been implemented both in C++ and Python to reach faster computational performances. These numerical math operations are widely used into the Byron library to increase computational performances, despite their usage can be turned off by user at compile time. The full set of functions, in fact, is enclosed into a macro definition (__fmath__) that can be enabled/disabled 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 related 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 compared their execution-time (we required a precision of at least $10^{-}4$). The obtained results are shown in Fig. 2.9, where we normalized the execution-time, keeping Numpy implementation as reference.

As can be seen, 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 optimized implementation of the algorithms in all the cases. 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 particular 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.

[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.

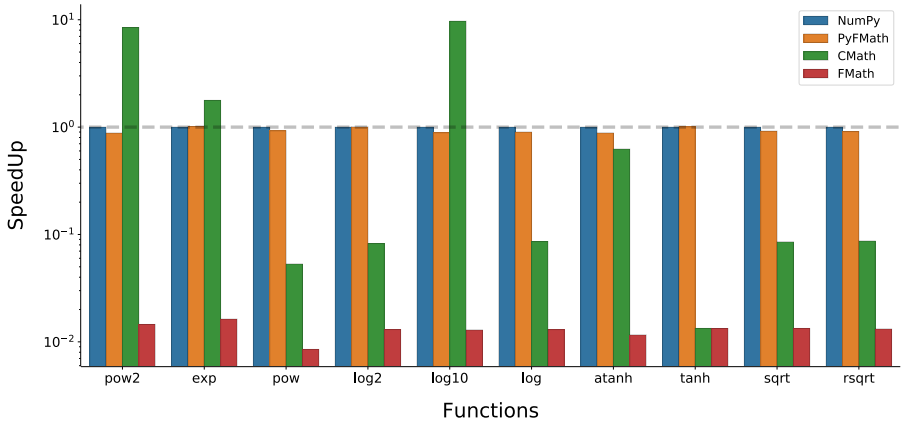


Figure 2.9: Time performances of standard mathematical operations implemented using Newton approximations. We compare the results obtained with the Numpy library (blue, reference) and the standard C++ library (CMath, green) to the respective functions implemented in our PyFMath (orange) and FMath (red). In the comparison we have to keeping 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.

2.2 Super Resolution

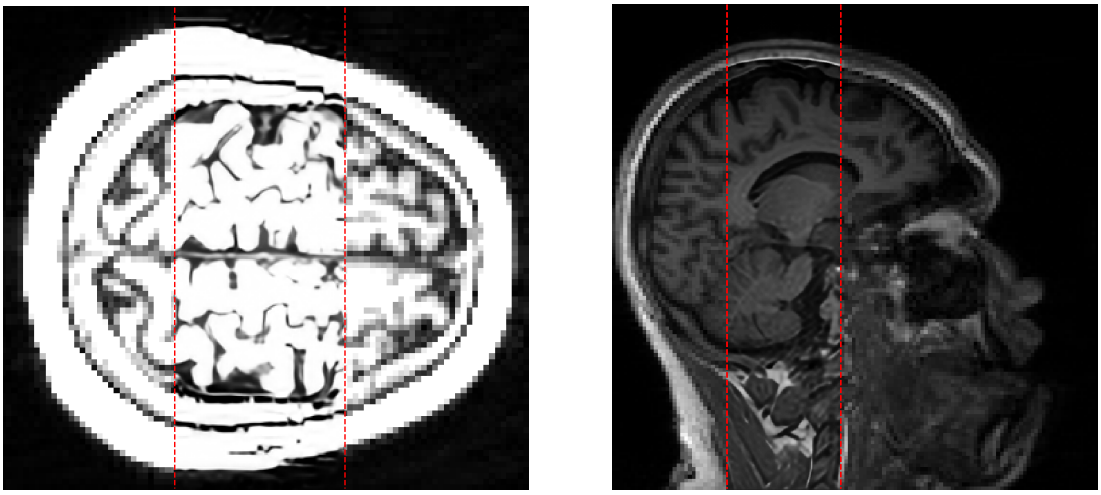


Figure 2.10: Single Image Super Resolution. Between the red lines the super resolved version of the original image.

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].

The first SR methods on digital images estimate the high frequency information of the images, starting from a series of low-resolution (LR) patches and their high-resolution (HR) counterparts. These patches (ROIs of the LR image commonly smaller than 50×50) were extracted after an edge enhancement procedure or a simple 2D Fourier transformation, which extracts the high frequency information. Collecting these patches an "association

[36] The best-known "implementation" of Super Resolution concerns the microscopy super-resolution. In this work we are focusing on algorithms and numerical implementations so we will talk about the numerical counterpart of this technique, totally ignoring the original "hardware" version.

dictionary" between LR and HR was created. This dictionary was used to learn the correct associations between LR and HR and then applied on new images. The images considered were 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 this problem was born in 2014 at the Engineering University of Honk Kong, due to the large popularity of these models during those years. The increasing computational power allowed to create automatic models able to learn the LR-HR associations without any dictionary. In this year the SRCNN model [32] arises, a three-layer neural network able to learn a large ensemble of features to reproduce the desired associations. The first layer aimed to extract the LR patches from the input image; the second layer produces 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 use appropriated 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 up to the most modern algorithmic solutions. At the end of this chapter the NumPyNet and Byron implementations of some modern models will be presented and applied over biomedical images.

2.2.1 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 an image. This behavior is often hard to digest, but it does not change anything about the previous considerations. The only change is related to the size of the neural network and its amount of parameters, that could drastically increase due to the larger output required. Let start from the beginning: to feed a super-resolution model we have to use a series of prior-known LR-HR image associations. In the real life, we have always a series of images, typically LR images, and we want to improve their resolution, 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 ones, we can easily down-sample them without particular troubles[37]. This re-sampling could introduce an 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 should be able to overcome also these issues.

Starting from these considerations, we can down-sample our images by a desired scale factor: common scale factors are between 2 and 8 and in this work we will refer to a scaling factor 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, looses part of the original information by definition. Thus, we can facilitate the learning choosing 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 neighbors interpolation*. This algorithm pass a kernel mask over the image and it substitutes each pixel mask to

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

their average[38]. This procedure can be achieved using a *Pooling* algorithm (in particular an AveragePooling) (ref. 2.1.5 for further information) 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 introduced 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 techniques, 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 reshaping and striding on the input matrix as shown in the following snippet.

Listing 2.4: 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):
    batch, w,  h,  c  = input.shape      # number of rows/columns
    b,     ws, hs, cs = input.strides    # row/column strides

    x = as_strided(input, (batch, w, self.stride[0], h, self.stride[1], c),
```

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

```
      (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 uses 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 loops but we have to separately manage the summation in a sequential section.

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

$$k(x) = \frac{1}{6}\begin{cases} (12-9B-6C)|x|^3 + (-18+12B+6C)|x|^2 + (6-2B) & \text{if} |x| < 1 \\ (-B-6C)|x|^3 + (6B+30C)|x|^2 + (-12B-48C)|x| + (8B+24C) & \text{if} 1 \leq |x| < 2 \\ 0 & \text{otherwise} \end{cases} \tag{2.25}$$

where x identifies each pixel below the filter. Common values used for the filter parameters are $B = 0$ and $C = 0.75$ (used by OpenCV library) or $B = 0$ and $C = 0.5$ used by Matlab[39]. Despite this function is also implemented in the most common Python libraries, we provide an efficient multi-threading implementation in the Byron library.

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

To better understand the told above functions, we can consider their application on the simple image given in Fig.2.11.

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

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

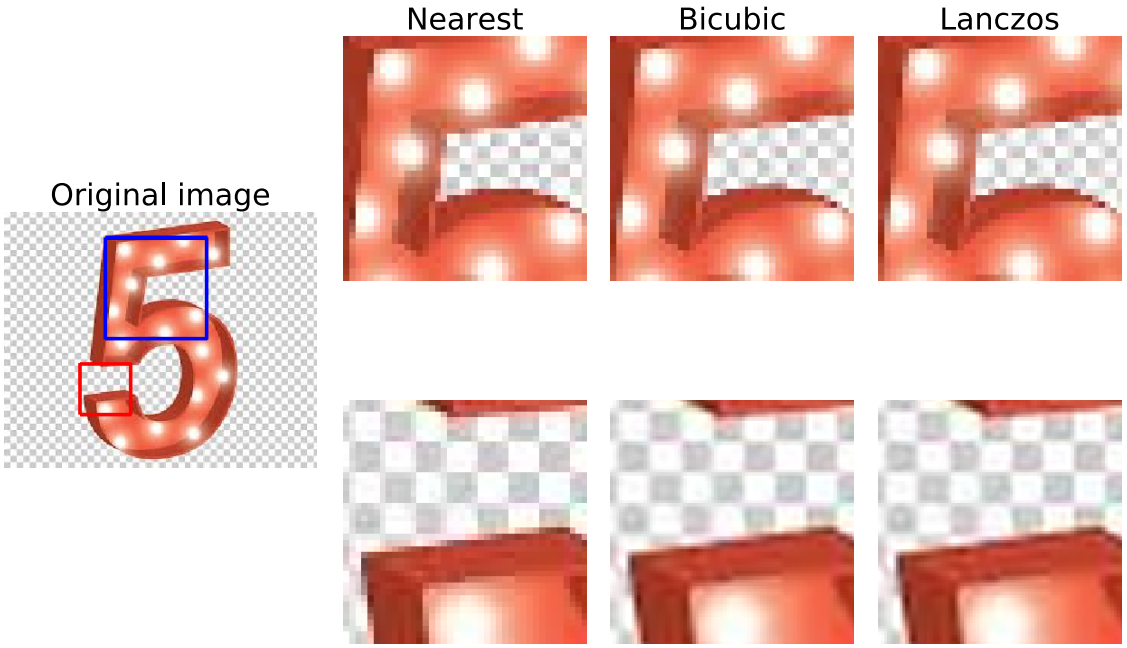


Figure 2.11: Re-sampling image example. **(left)** The original image. **(up right)** The down-sampled blue-ROIs using different interpolation algorithms (Nearest, Bicubic and Lanczos, respectively). We use a scale factor equal to 2 (half size in down-sampling and double size in up-sampling). The Lanczos interpolation is the lossless algorithm, but from a qualitative point-of-view the result are quite the same of the bi-cubic ones. **(down right)** The up-sampled red-ROIs using the same interpolation algorithm of the upper row. Also in the Up-sampling the Lanczos and bi-cubic algorithms produce equal qualitative results. The Nearest algorithm produces the worse results both in up- and down- cases.

produces always the worse results both in up-sampling and down-sampling. In the bi-cubic 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 bi-cubic and Lanczos are quite similar in both cases, but the computational cost of the Lanczos algorithm is greater than the bi-cubic one. This is the reason why the bi-cubic interpolation is the most used technique for image resizing, with a balance between computational cost and qualitative result. In our implementation of SR algorithms we chose to use the bi-cubic interpolation for those reasons.

The main aim of SR algorithms is to overcome these results obtaining 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 sections we will introduce some useful mathematical scoring to numerically evaluate the image quality.

2.2.2 Image Quality

The most powerful image quality evaluator is given by our eyes. This is true also for SR problems: the final purpose still remains to obtain images that are better visible for human eyes, the so called *visual loss*. We can however provide some mathematical formulas which allow to quantitative evaluate the image quality. In both cases we need to establish a relationship 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 would be a measure of similarity between the two images.

The simpler 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

$$PSNR = 20 \cdot \log_{10} \left(\frac{\max(I)}{\sqrt{(MSE)}} \right) \tag{2.26}$$

where $\max(I)$ is the maximum value which can be taken by a pixel in the image (in general it should be 1 or 255 depending on the image format chosen) and MSE is the Mean Square Error (ref. 2.1.10) between the original image and the reconstructed one. The MSE for an image can be computed as:

$$MSE = \frac{1}{WH} \sum_{i=1}^{W} \sum_{j=1}^{H} (I(i,j) - K(i,j))^2 \tag{2.27}$$

where W, H are width and height of the two images and I, K are the original and reconstructed image, 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 [47], but it is not always related to a qualitative visual quality. Despite it is commonly used as loss function for SR models.

Considering the series of images shown in Fig. 2.11 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. 2.2). As expected, the lowest PSNR value is achieved

	Nearest	Bicubic	Lanczos
PSNR	25.118	27.254	26.566
SSIM	0.847	0.894	0.871

Table 2.2: Image quality scores: PSNR (peak-signal-to-noise-ratio) and SSIM (Structural SIMilarity index). The values are computed on the image shown in Fig. 2.11. The original image was down-sampled using a Lanczos algorithm and then re-up-sampled using three different algorithms: nearest, bi-cubic and Lanczos interpolations. For each interpolation algorithm the PSNR and SSIM was evaluated. As expected the highest scores were obtained using the bi-cubic algorithm, while the worst reconstruction is performed by the nearest algorithm.

by the nearest interpolation method, while the best performances are obtained by the bi-cubic algorithm. This confirms the wider use of the bi-cubic method in image processing applications. Moreover, we have to take in account that an increment of 0.25 in PSNR value corresponds 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 improvements seen by human eyes. The SSIM function can be expressed as

$$SSIM(I,K) = \frac{1}{N}\sum_{i=1}^{N} SSIM(x_i, y_i) \tag{2.28}$$

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

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{({\mu_x}^2 + {\mu_y}^2 + c_1)({\sigma_x}^2 + {\sigma_y}^2 + c_2)} \tag{2.29}$$

where μ and σ are the mean and variance of the images, respectively, and σ_{xy} represents the covariance. The c_1 and c_2 parameters are fixed to avoid mathematical divergences. Also in this case, higher values of SSIM correspond to high an similarity between the original image and the reconstructed one.

Based on the previous equation, we can highlight a link with the pooling function discussed in 2.1.5. Also in this case, in fact, we work 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. 2.11 and Tab. 2.2) 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 bi-cubic algorithm. The gap between SSIM values is smaller than PSNR ones, but this is due to the different domains of the two functions.

2.2.3 Super Resolution Models

There were different kinds 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

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

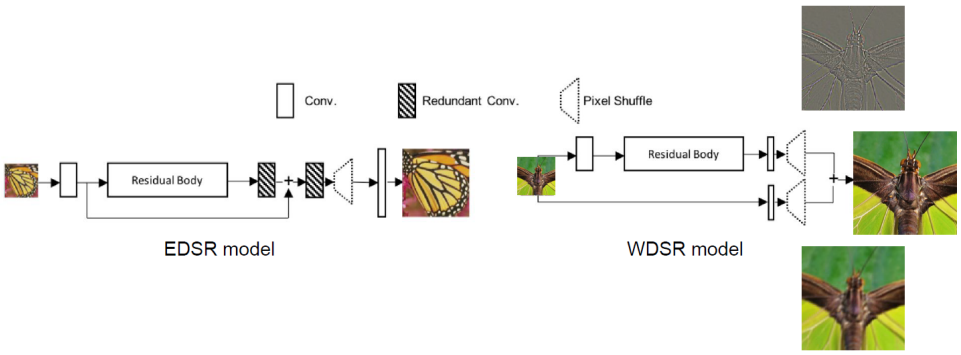


Figure 2.12: Super Resolution models analyzed in this work. **(left)** EDSR model. The model is a modified version of the ResNet architecture designed for SISR applications. The architecture is made by a sequential CNN framework, which processes the input image. The EDSR has more than 43 million of parameters 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 million of parameters, less than 10% of the EDSR model.

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 2.3: 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 [2]. 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 optimizations to improve the training speed and the quality of the output image. In particular, the batch normalization steps are removed to improve the speed of the algorithm: it was proved 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 [56]. A scheme of the EDSR architecture is shown in Fig. 2.12 (a) and the full set of parameters are reported in Tab. 2.3: the EDSR model has more than 43 million of parameters in total.

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) processes the feature map. The tail of the architecture is made by an up-sample block which re-organizes 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 information from 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 information 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 [88]. 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. 2.12 (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 reduces the computational time and the amount of parameters. Furthermore, the combination of 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. 2.12 (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 2.4: 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 activation tends to block the information flow from the first layers [78] and it is important to prevent it in super resolution structures, 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 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.2.4: the WDSR has slight more than 3.5 million of parameters, less than 10% of the EDSR model. This confirms the computational efficiency of the WDSR against the EDSR one.

[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.

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 environments[42].

2.2.4 DIV2K dataset

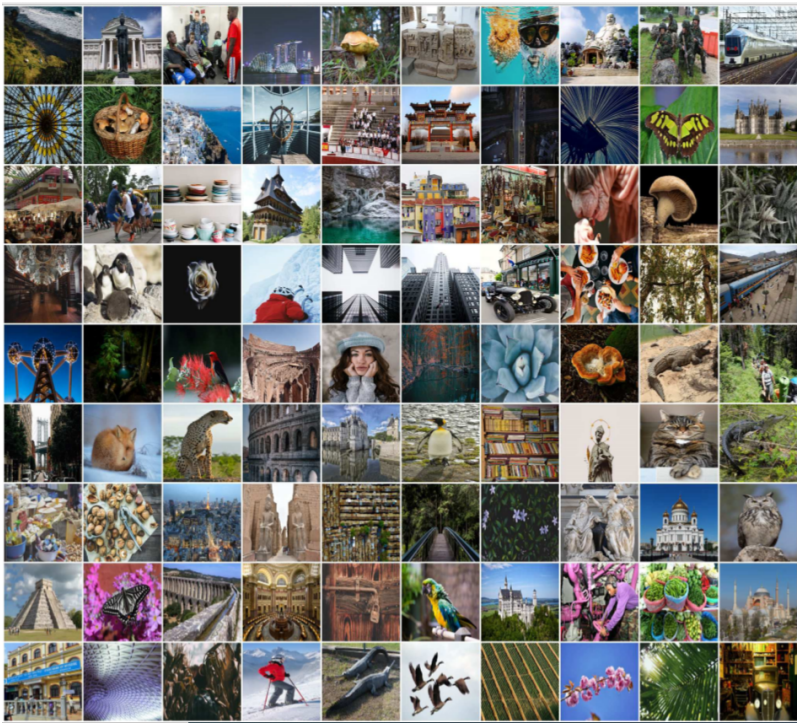


Figure 2.13: DIV2K validation set examples.

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 monitor the state-of-art in digital image processing and image analysis and it takes place 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 topic. The model which performs the higher PSNR value over a validation set extracted on the DIV2K dataset wins the challenge. For these reasons 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

[42] We have to mention also that the publicly available implementations 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.

the model can evaluate its accuracy: also this second set of images has 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 samples, i.e width and height dimensions must have at least 2×10^3 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), 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 to 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 on 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.

2.2.5 Results

As discussed in the previous sections we implemented the EDSR and WDSR models into our custom Byron library, but we did not re-trained the models. The weights used in this work were taken from the official implementations of the models, publicly available on the corresponding Github pages(EDSR and WDSR).

First of all we tested our implementation in terms of execution time. The official implementations are written using Tensorflow and PyTorch frameworks and they are usable only with a GPU support. Thus, no tests were performed in relation to them, but only between the two models inside the same Byron framework.

We started our numerical tests comparing the efficiency of the two models, keeping fixed the input sizes. In this way we could reproduce what has already proved by the original papers, i.e the numerical efficiency of the WDSR model against its predecessor EDSR one. In particular over 100 runs we could easily prove that the WDSR model is more than 10x faster than EDSR, due to the discussed properties explained in the previous section.

A second analysis was performed on the performance efficiency of both the models over a validation set of images. We evaluated the two quality scores described in the above sections (PSNR and SSIM) over the validation set provided by the DIV2K dataset. The full validation set comprises 60 images and we compared the efficiency of the two models against the standard up-sampling technique given by the bi-cubic algorithm. In Fig. 2.14 we show the score distributions obtained using the three methods over these 60 images.

As can be seen by the two plots in Fig. 2.14 the quality improvement given by the Super Resolution methods against the bi-cubic algorithm is evident. On the other hand, the gap between the two Super Resolution models is relatively small: the EDSR model performs statistically better than WDSR, but we have also to take into account that the

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

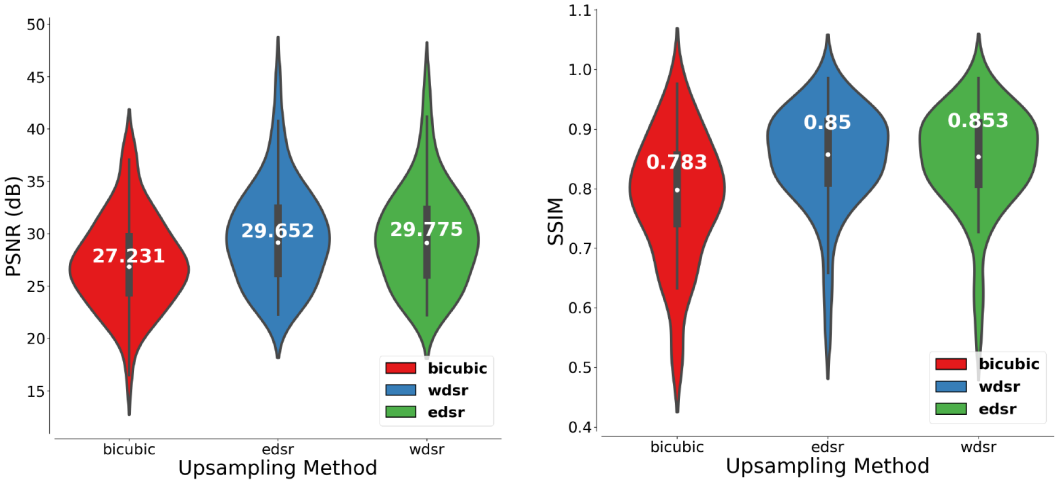


Figure 2.14: Comparison of performances between the bi-cubic up-sampling and EDSR and WDSR Super Resolution models on the DIV2K validation set. The performances are obtained down-sampling the input images and then re-up-sampling them according to the desired scale factor: the chosen scale factor is 4x. **(left)** PSNR score on the 60 validation images. **(right)** SSIM score on the 60 validation images.

WDSR model has less than 10% of the EDSR parameters. Moreover we have to consider the combination between performances and execution time: in this case the WDSR is certainly the best choice for Super Resolution applications.

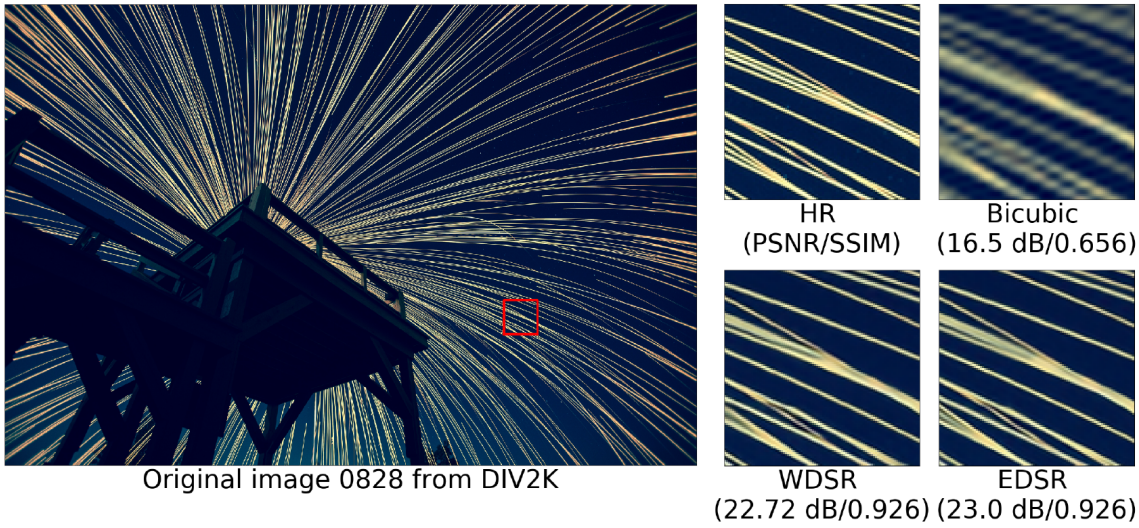


Figure 2.15: Super Resolution visual example extracted from the DIV2K validation set. The quality score in terms of PSNR and SSIM are compared between a standard bi-cubic up-sampling and the EDSR and WDSR models.

A visual proof of our results is shown in the Figures 2.15, 2.16 and 2.17: as discussed in the previous sections, the visual comparison is certainly the more accurate score for super resolution applications. As can be seen in Fig. 2.15 and Fig. 2.16 the two models have perfectly learned how to reconstruct the complex line shapes of the input image. At the same time they have also learned how to reconstruct words and different kinds of textures. These results highlight either the efficiency of the two models, either the importance of the training set for this kind of applications: the DIV2K dataset has a wide heterogeneity of different textures inside its images and, thus, the model is able to perfectly reconstruct a

Figure 2.16: Super Resolution visual example extracted from the DIV2K validation set. The quality score in terms of PSNR and SSIM are compared between a standard bi-cubic up-sampling and the EDSR and WDSR models.

Figure 2.17: Super Resolution visual example extracted from the DIV2K validation set. The quality score in terms of PSNR and SSIM are compared between a standard bi-cubic up-sampling and the EDSR and WDSR models.

huge amount of different shapes after the training section.

The obtained results encouraged us to test the efficiency of the two models also with different kind of images. In particular we tested their quality score performances on NMR images of human brain. The images were provided by the Bellaria Hospital, but due to privacy constrains we can show the results only on a single set of them[44].

We used a series of T_1 weighted NMR images sampled with a spatial frequency of 1 mm in each direction (x, y, z with a resolution of 256×256 pixels). The images were down-sampled to 128×128 (2x down-sampling) and to 64×64 (4x down-sampling). Then, both the down-sampled series were re-up-sampled to the starting dimensions using the EDSR (2x) and WDSR (4x) models. Also in this case the results were compared to a standard bi-cubic up-sampling algorithm. The data acquisition included 176 slices and each one was independently processed. The results obtained by the 2x and 4x up-sampling are shown in Fig. 2.18 and Fig. 2.19, respectively.

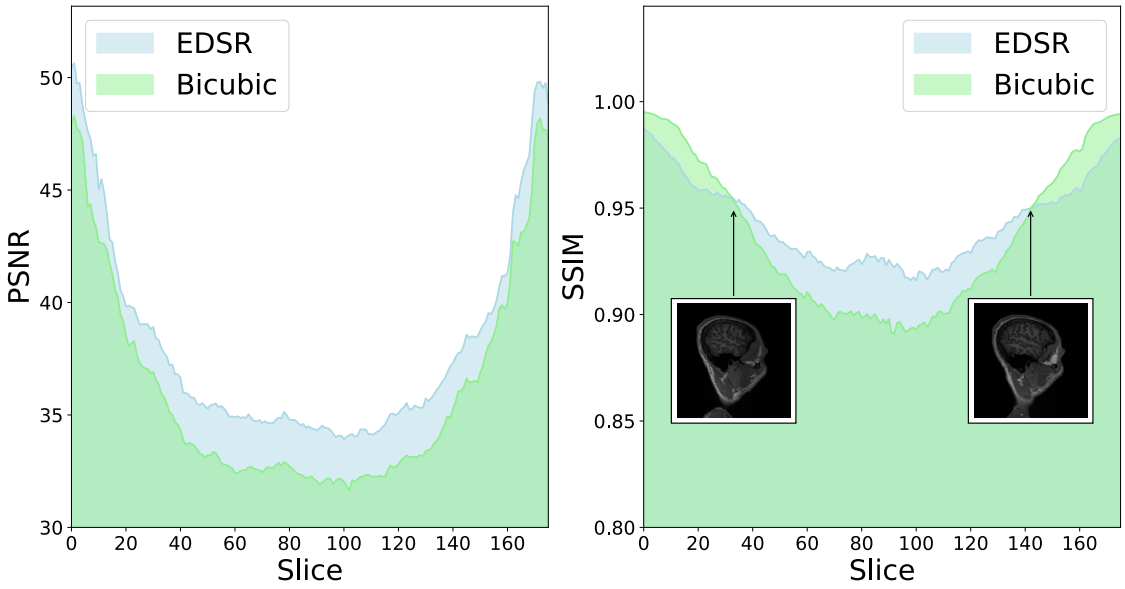


Figure 2.18: PSNR (left) and SSIM (right) quality scores obtained by the EDSR model on the 2x NMR slices of the human brain. We compared the efficiency of the EDSR model to the results obtained by a standard bi-cubic up-sampling. The Super Resolution model is able to better reconstruct the brain shapes and textures, obtaining a higher image quality score in the major part of the slices. The critical points, i.e where the bi-cubic up-sampling performs better than the super resolution algorithm, are highlighted in the plot and they correspond to the less informative area of the brain.

In both the cases the Super Resolution models over-performed the bi-cubic algorithm in the major part of the slices. The only exception is given by the 2x results where there are a set of slice in which the bi-cubic efficiency is higher than the super resolution one in terms of SSIM quality score. As can be seen in Fig. 2.18 the efficiency of the EDSR model decreases in the first and last parts of the acquisition: the corresponding slices are highlighted in the plot and we can easily notice how they correspond to the less informative portions of the brain. The most central (and thus informative from a bio-medical point-of-view) part is better reconstructed by the Super Resolution models. We would stress that an increment of 0.25 in the PSNR score is considered visible by naked eyes. The images showed at the beginning of this section (ref. Fig. 2.10) were obtained using the WDSR model over our images and they visibly highlight the efficiency of our Super Resolution

[44] I'm the "patient" in this acquisition.

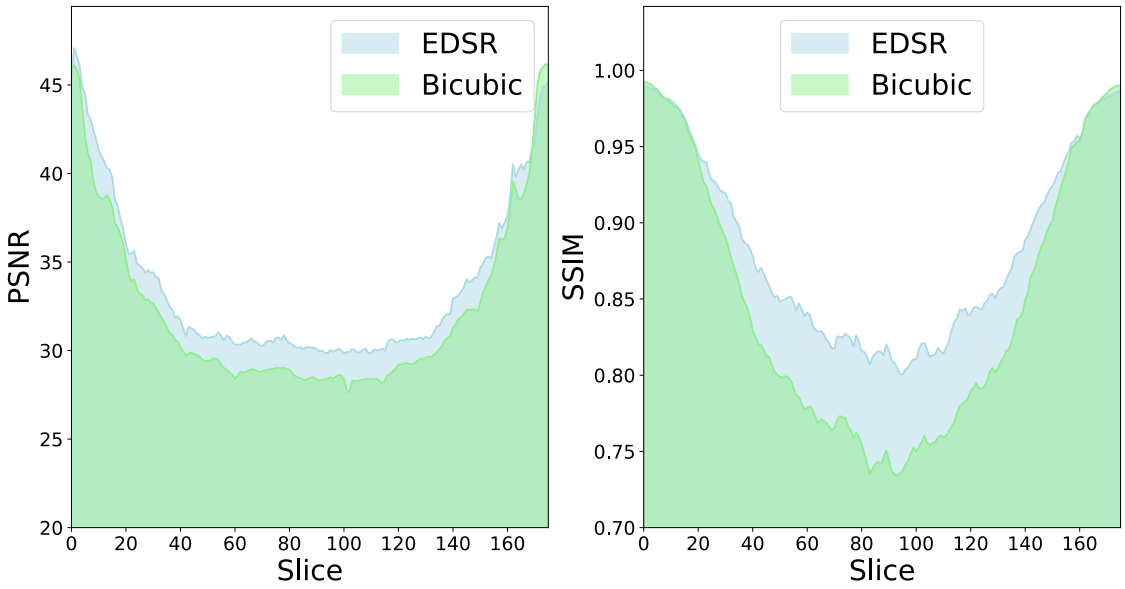


Figure 2.19: PSNR (left) and SSIM (right) quality scores obtained by the WDSR model on the 4x NMR slices of the human brain. We compared the efficiency of the WDSR model with the results obtained by a standard bi-cubic up-sampling. The Super Resolution model is able to better reconstruct the brain shapes and textures, obtaining a higher image quality score in the major part of the slices. The critical points, i.e where the bi-cubic up-sampling performs better than the super resolution algorithm, also in this case correspond to the less informative area of the brain.

models.

In conclusion this work proved how we can obtain good results without need to re-train a Neural Network model. The presented models are, in fact, able to generalize the learning patterns and textures also to different image kinds. The PSRN and SSIM performances obtained by the NMR image reconstructions are also in agreement with the results obtained on the DIV2K validation set and they confirm the goodness of the DIV2K dataset as training set for Super Resolution applications. Further analyses are still in work in progress and these results encourage us to test our trained models also on wider bio-medical datasets.

2.3 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 and the list goes on.

There are many machine learning and deep learning techniques proposed during the years about this topic and each one has its own pros and cons. The most prominent and moder techniques involve the use of very deep Neural Network models, with a huge amount of parameters to tune. The most famous ones are probably the Faster R-CNN (*Faster Region Convolutional Neural Network*) [73] and its "evolution" into the YOLO (*You Only Look Once*) model [70, 71, 72].

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 their speed. The

standard approach for object detection is based on moving a *sliding window* to search in every position of the image the objects. However, the intrinsic problems of these kinds of methods are the window dimensions and the large computation required to map with multiple window sizes the full image. 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 2 000 region proposals, i.e bounding boxes, and applying to each one a image classification procedure, using a 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 2 000 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 on these models is given by the *anchor* concept: an *anchor* is essentially a box and it identifies the shape of a portion of the input image at different scale levels. 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 previous cited 2 000 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 publicly available, but its popularity is also due 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 slits 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 sections 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, considering its efficiency against the original implementation. Finally, we will join the efficiency of the previous Super Resolution models to the performances of our optimized implementation of YOLO.

2.3.1 Yolo architecture

YOLO Neural Network architecture was firstly published in the 2015, but from the first version many improvements have been performed and now we have its third version. 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 million of parameters. The first version of YOLO was based on a Darknet-19 architecture

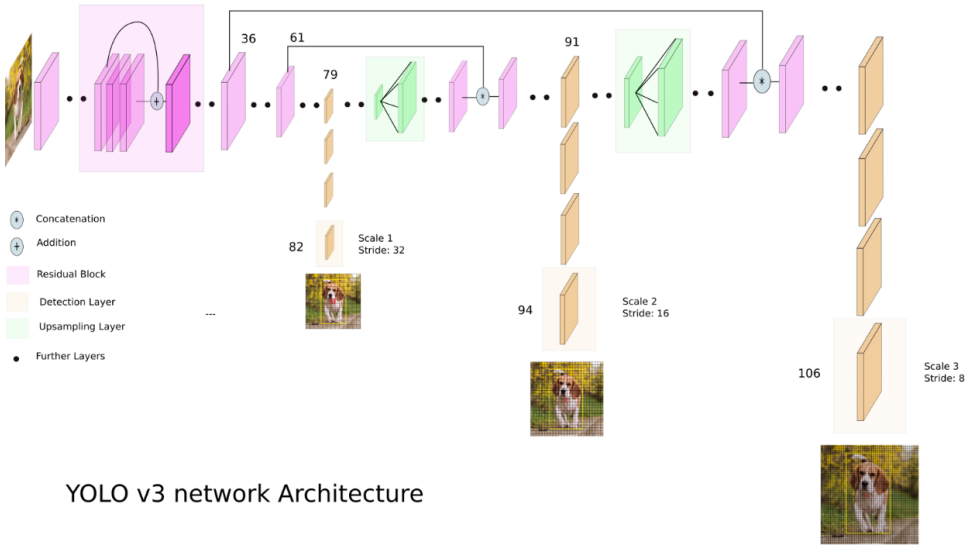


Figure 2.20: Yolo Neural Network scheme.

(19-layers Neural Network followed by 11 more layers for object detection). In the last release of YOLO, the first part of the network structure is used for the feature map extraction and it is essentially a modified version of the Darknet-53 model, i.e the updated version of the previous model, with more layers and parameters. These improvements increase the classification performances, but it throwbacks a reduction in computational performances[45]. These improvements could be done also thank to the introduction of multiple residual blocks which, as discussed in the previous sections (ref. 2.1.8), allow to increase the deep of the model without losing performances.

YOLO performs object detection using a multi-scale approach: three different scales are taken into account during the training section to improve classification performances. The network structure can be broadly summarized as a simple CNN and its output is generated by applying a series of three different detection 1×1 kernels on the feature map. Moreover, this detection is performed in three different places into network, i.e three YOLO detection layers are distributed along the network structure. The detection kernel shape is $1 \times 1 \times (B \times (5 + C))$, where B is the number of bounding boxes which a cell in the feature map can predict and C is the number of classes. The fixed number ("5") is given by 4 bounding boxes attributes plus 1 object confidence coefficient (the so-called *objectness* into the code). In our applications we have used the COCO dataset (see next sections, 2.3.2) and thus we have fixed the value 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 levels 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[46]. Then the feature map produced by the 79th layer is subjected to few convolutional layers before being 2x up-sampled to a 26×26. The up-sampling is performed by a previously discussed UpSample function/layer (ref. 2.2.1). The feature map is then concatenated with the one produced by the 61st layer and it is processed by a second series of convolutions up to the 94th layer which performs the second detection. A third (similar) procedure is performed again up to 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 analyze smaller

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

regions: a comparative analysis of these three different scale results improves the detection performance and it helps to filter false positive detections.

The introduction of three different detection layers improves the detection of the 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 the small objects into the image.

The model uses a total of 9 anchor boxes with three scales per each. Anchors have to be computed before the training phase on the dataset: the author suggests to use a K-Means clustering for this purpose. The first three anchors are 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 is 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: thus, we have to filter them. In other words we want to preserve only bounding boxes "responsible" for true objects. This can be achieved using the highest IoU (*Intersection Over Union*) with the ground truth. YOLO uses a modified version of MSE error between predictions and ground truths. In particular, the loss function is composed by three terms: classification loss, localization loss and confidence loss.

The classification loss quantifies the detection error and it is given by

$$\mathcal{L}_1 = \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c)\right)^2 \tag{2.30}$$

where $\mathbb{1}_i^{\text{obj}}$ is equal to 1 if an object appears in the 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 predicted boundary box locations and sizes: in this way we filter only the boxes responsible for detecting the object.

$$\begin{aligned} \mathcal{L}_2 = \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_i^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2\right] + \\ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_i^{\text{obj}} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2\right] \end{aligned} \tag{2.31}$$

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} increases the weight for the loss in the boundary box coordinates[47] and (x, y, w, h) are the boundary box coordinates.

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

$$\mathcal{L}_2 = \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_i^{\text{obj}} \left(C_i - \hat{C}_i\right)^2 \tag{2.32}$$

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:

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

$$\mathcal{L}_2 = \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_i^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \tag{2.33}$$

where λ_{noobj} weights down the loss when detecting background (most boxes do not contain any object and in the training images a large amount of pixels are occupied by background)[48].

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

$$\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3 \tag{2.34}$$

To further improve detection performances we have to remove duplicate detections. This is performed by YOLO applying a non-maximal suppression to remove the duplicates with lower confidence. The method sorts the predictions, according to the confidence scores, and, starting from the top scorer, it filters them 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 shapes.

2.3.2 COCO dataset

Figure 2.21: 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 objects and all these images have 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 objects and varying lighting conditions. The set of training images could not be made by high quality images, but it needs a wide heterogeneity of data.

During a training section we have also to take into account that a part of the available data has to be "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 million of parameters to be tuned and a sufficient number of annotated samples to train it is hard to produce. Fortunately, there are different publicly available datasets designed to face object detection training problems. One of the most popular one is the COCO dataset.

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

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 a new version is released quite every year.

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[49]. The dataset includes more than 300 000 images in which more than 200 000 are already labeled. Certainly, the unlabeled ones could be used as test set for a visual estimation of performances[50]. In our applications we have 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[51]. This variability helps the sensibility of network (convolutional) filters to the details of the image. Moreover, it helps the detection in identifying 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 on the COCO dataset. For our applications we did not re-train the model[52], but we converted the available weights to the Byron format.

2.3.3 Results

The original implementation of the YOLO model was provided by Redmon et al. and it is publicly available in his official web page of the darknet project. The code is written in Ansi-C and, only thank to the many branches developed by the Github community, it can be compiled on all the OS. The Ansi-C language is a very low-level programming language and it is hard to obtain better performances rewriting the code. This guarantees its supremacy in terms of speed in the research community. The code is particularly optimized for GPU applications: the darknet library provides an efficient CUDA support and it can be optimally used only with NVIDIA GPUs.

The proposed Byron library has been developed following the backbone and innovative ideas provided by the darknet project. The main difference between them is the programming language chosen: Byron is written in pure C++, a "higher"-level programming language. Generally, we can not obtain better computational performances using C++ in relation to an Ansi-C implementation. However, the C++ language is more popular than Ansi-C and more easy to write and modify. The second main difference of Byron is related to the target computational environment: it is designed and optimized to reach the better performances on a single or multiple CPUs architecture. In this way we aim to enlarge also the usability of our code. Many research groups, in fact, have very powerful server grade machines, without a GPU support and it is hard for them to get close to the deep learning applications. An emblematic case is given by the bioinformatics research, in which a large

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

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

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

[52] 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.

amount of money are spent to buy efficient server grade machines to process large DNA datasets which are commonly processed using only CPUs support.

In the previous sections we have discussed about different kinds of optimization related to the various (possible) components of a Neural Network model. All these optimizations were implemented in the Byron library to reach the best performances. Moreover, studying the Redmon et al. implementation, many issues were found in the darknet project, especially related to the multi-threading support and thread concurrency. Byron library widely uses OpenMP features, paying attention to thread concurrencies, minimizing the time for thread spawning. In Byron a single parallel section is open at the beginning of the processing and it is closed at the end, with a carefully management of the threads along all the network structure.

In view of these considerations, a first test was performed to compare the Byron efficiency against the darknet one, in terms of time-performances. To compare the results, we implemented the same YOLO model into our custom Byron framework and we compared its time efficiency against the original implementation. Tests were performed turning off the multi-threading support, since the darknet implementation uses it only in the GEMM steps. We performed 5 independent simulations using the same input image to test the time stability of both implementations. Each simulation performed 100 runs of both algorithms. The results are shown in Fig. 2.22, where we have normalized our times in relation to the darknet ones (reference).

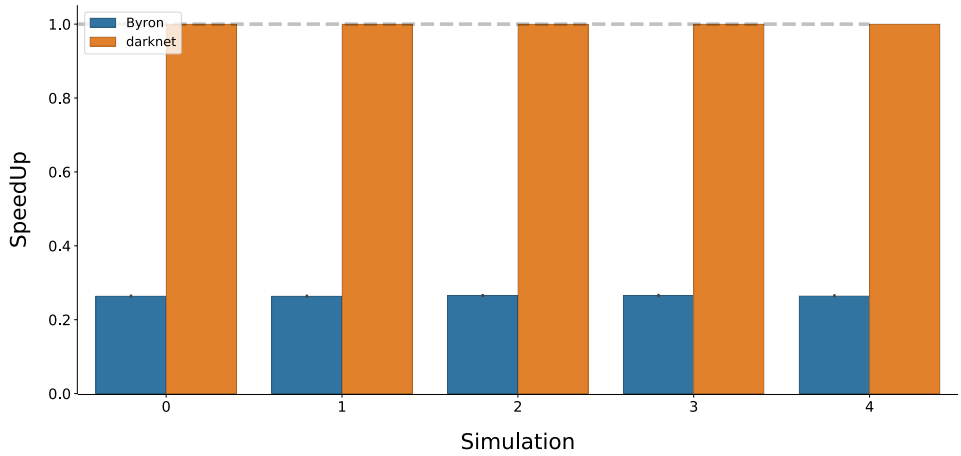


Figure 2.22: Comparison of time performances between the Byron and darknet implementations of YOLO model. Simulations were performed keeping fixed the input image sizes and without any multi-threading support. Each simulation includes 100 runs of both the algorithms. The Byron version is approximately 3.8x faster than darknet in all the simulations.

Both implementations are quite stable across the simulations and our measures show a very tight variability. The differences in time performances are evident and we can summarize them with a 3.8x speedup obtained by Byron against darknet. The multiple optimizations discussed and used by Byron are proved by numerical results and they highlight the efficiency of our implementation against the state-of-art. We would stress that, using our version of the model into a server grade machine (128 GB RAM memory and 2 CPU E5-2620, with 8 cores each), YOLO can process 416×416 images in real-time (less than a second), while darknet can reach the same performances only with a GPU support.

Once the efficiency has been proved, we "update" the YOLO model to overcome its issues. In particular, we have discussed in the previous section (ref. 2.3.1) that the biggest issue related to YOLO concerns the detection of small objects. Despite the model is

incredibly efficient in object detection also with low quality images, there is a sort of limit in the number of pixels needed for object identification. This kind of problems are particularly critical in people counting tasks, and moreover in crowd counting applications. YOLO is able to identify the major part of persons into an image, but it decreases its efficiency when they partly overlap or they are far from the camera (and thus at low resolution).

We had the opportunity to empirically verify its limit, working on a people tracking project for real-time applications. The project was developed in collaboration with the Complex Systems (*PhySyCom*) group of the University of Bologna, with the support of Canon Inc., Telecom Italia and Fabbrica Digitale, and it aims to detect and track people, using video camera devices. The experiments were executed around the streets of Venice city, with the support of the Venice City Council. Using our custom implementation of YOLO we were able to detect the major part of persons, but we lost efficiency when the people flow increased or when we face with open space area (a crucial point was Piazza San Marco).

In the previous section we have largely discussed about the efficiency of Super Resolution techniques to improve the image quality, so it stands to reason that their application will be helpful to overcome the told above issue. We applied the previously described EDSR model to critical images, i.e where YOLO did not perfectly detect all the people in the picture. For privacy reasons, we can not show the results obtained on Venice data, but we can show a simple example to prove our combination of models. The example is shown in Fig. 2.23.

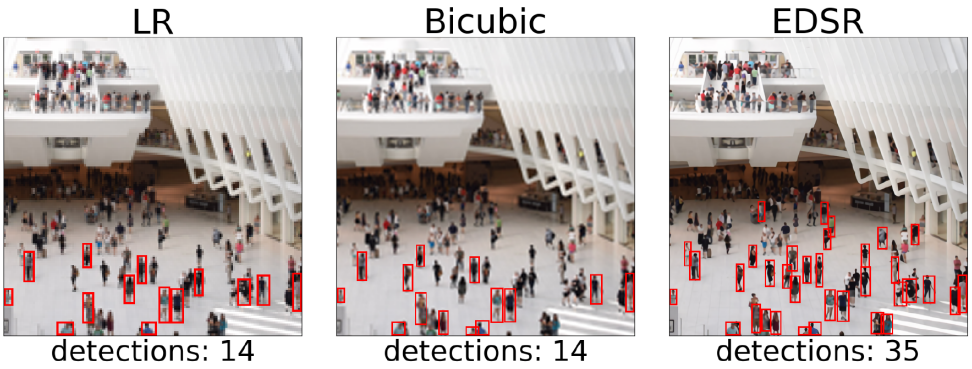


Figure 2.23: YOLO people detections on a image ROI. **(left)** The original ROI and its corresponding detections. **(center)** Up-sampling of the original ROI using a bi-cubic algorithm and its corresponding detections. **(right)** Up-sampling of the original ROI using the EDSR model and its corresponding detections. The use of Super Resolution model is able to improve the YOLO detection of small persons of more than 200%. YOLO is not still able to detect the smaller (far) persons.

On the first image (left of Fig. 2.23) we show only a small ROI of a (larger) input image, where YOLO is able to find only few people. We would remark that the detected people are all in the bottom part of the image, where person "sizes" are bigger. Using a standard bi-cubic up-sampling (center of Fig. 2.23) detection performances are the same, proving as standard up-sampling methods are not appropriate to overcome this task. The application of EDSR model (right of Fig. 2.23) is able to improve the quality of the image and ease the YOLO work. In this case the detection is more than twice of the previous case. The issue remains for the top part of the image, where only few pixels identify a person. Set out to test the limit of this model combination, we have extracted a further ROI from it, selecting only the top part of the image. The results are shown in Fig. 2.24.

The task in this case is certainly harder and also human eyes hardly count the number of persons into the image. With the raw image, YOLO is not able to find anything and

Figure 2.24: YOLO people detections on a ROI of the previous image ROI. **(left)** The original ROI and its corresponding detections. **(center)** Up-sampling of the original ROI using a bi-cubic algorithm and its corresponding detections. **(right)** Up-sampling of the original ROI using the EDSR model and its corresponding detections. Without the Super Resolution application the YOLO model is not able to recognize any person. The bi-cubic up-sampling allows the detection of only 2 persons against the 7 obtained by the use of EDSR model.

also with the bi-cubic up-sampling only 1 person is recognized by the model. With the EDSR pre-processing, the detection performance increases and 7 persons are recognized. Certainly, the people count is underestimated, but Super Resolution pre-processing seems to be the only available solution to improve YOLO performances on these critical cases.

2.4 Image Segmentation

In the previous section we have discussed about object classification and object detection problems (ref. 2.3). Now we want to go deeper on this topic, aiming to extract the exact pixels belonging to an object into a given picture. 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. Information about pixel-wise positions of an object into a picture could be used to extract object shapes 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, where we have to find the exact object shapes to better estimate their perspective position. All these applications require algorithms fast as much as possible, closed to real-time.

We can face these problems using image processing pipelines or training a Neural Network model. In the first case, we have to stack a series of functions to process the input image: the pipeline has to filter and extract the useful information about the searched object, but most of all it has to be as most general as possible to face common heterogeneity of image samples. In the second case, we leave to the Neural Network model parameters the search of the optimal functions combination, but we have to provide a supervised input pattern made by several samples, i.e a combination of inputs and annotated pixel-wise masks of each image. Image annotation is one of the most hardest and boring steps of image segmentation and for these reasons it is very hard to find public dataset usable.

In this chapter we introduce a Neural Network model commonly used in image segmentation problems, describing its characteristics and performances. We applied this model to a novel dataset of CT images. The dataset annotation has been performed using a custom semi-supervised pipeline of image processing developed by the author and the Neural Network model was trained and tested on this dataset. The original data were taken from here [42] and the corresponding annotations are released on here.

2.4.1 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 has shown its efficiency also in different tasks and 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 one is an expansion path (or *decoder*). The first set of layers, in fact, are a sequence of convolutional and pooling layers, which aim to extract features and reduce the input dimensionality, in the same way as an encoder converts 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 shown in Fig. 2.25.

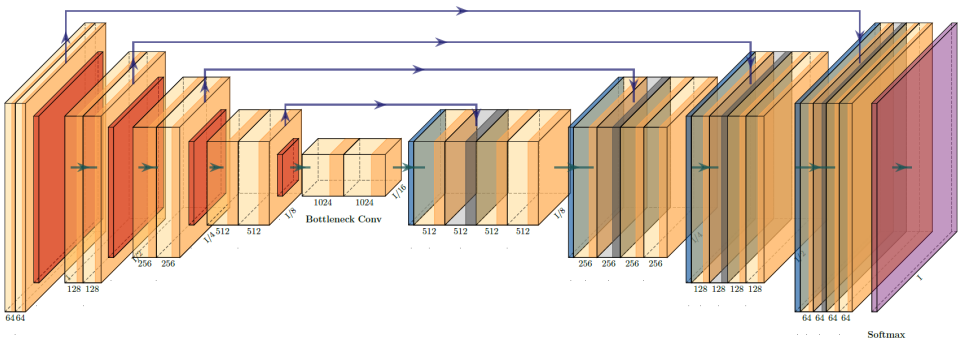


Figure 2.25: U-Net model scheme. The first part of the structure represents the encoder, while the tail of the model is the decoder part. The model name is given by the numerous shortcut connections which link the encoder layers to the decoder ones: if we contract the long-range connections the global structure acquire a U form. The figure was generated using the PlotNeuralNet package of H. Iqbal.

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

In the previous sections, we have 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, it would be a binary matrix. In these cases a valid loss function (also used in our applications) is the *binary cross-entropy* (ref. 2.1.10).

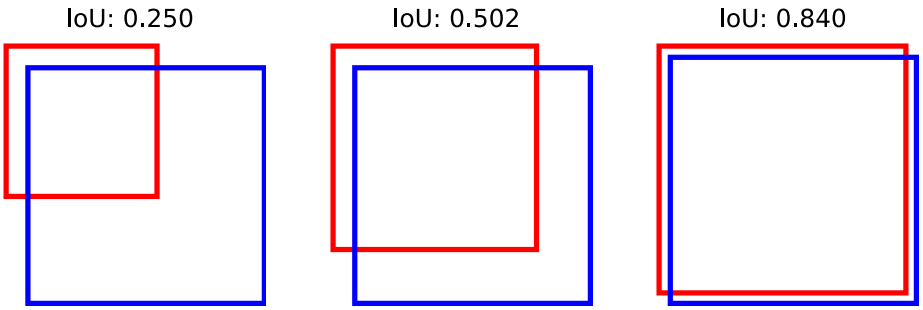


Figure 2.26: IoU score example. The IoU score is computed as the area of the intersection of the two boxes over their union. Starting from the left we can see an increment of the overlap between the two boxes related to an increment in their IoU score.

A word of caution must be spent about the metrics for the performance evaluations of our model. Standard metrics, as the *accuracy*[53], are not good measures to face segmentation problems. If we want to find and segment an object into a picture, 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 (background) and only few ones (object). In this case the standard metric functions have to consider an unbalanced number of samples: if the model outputs a matrix of all zeros, its accuracy would be high despite the informative values are only the few pixels equal to one. A possible solution to overcome this issue is given by the *mean IoU score* (ref. 2.3.1 for information about IoU), which measures the average IoU between the output mask and the binary ground truth:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}} \tag{2.35}$$

The efficiency and meaning of this score can be visible in Fig. 2.26.

2.4.2 Femur CT Dataset

The training of a deep Neural Network model as the U-Net requires a large set of images with corresponding labels. We developed some experiments on automatic segmentation into a (work in progress) project commissioned by the Rizzoli Hospital of Bologna. The project aims to develop an automatic pipeline of image processing to extract the 3D femur

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

structure starting from CT (*Computer Tomography*) images. In particular, the crucial point is to improve femur head identification and segmentation, trying to discriminate this part of the bone from the articular cartilage and, moreover, from the acetabular fossa. The project was developed in collaboration with the Engineering group of the professor M. Viceconti of the Department of Industrial Engineering and it aims to study the osteoporosis syndrome and its consequences.

In this work no data have been provided by the Rizzoli Hospital and it is hard to find annotated biomedical (public) images on-line, especially about the region of our interest. We found only few samples of femur CT images (4 patients) and they are certainly not enough for an accurate training of the model. Thus, we applied a huge data augmentation pre-processing: each image was randomly rotated, shifted and mirrored. Moreover, we had to face on the problem of data annotation, which is always a difficult and time expensive task: we did not have accurate medical annotations, so we had to perform them by ourself.

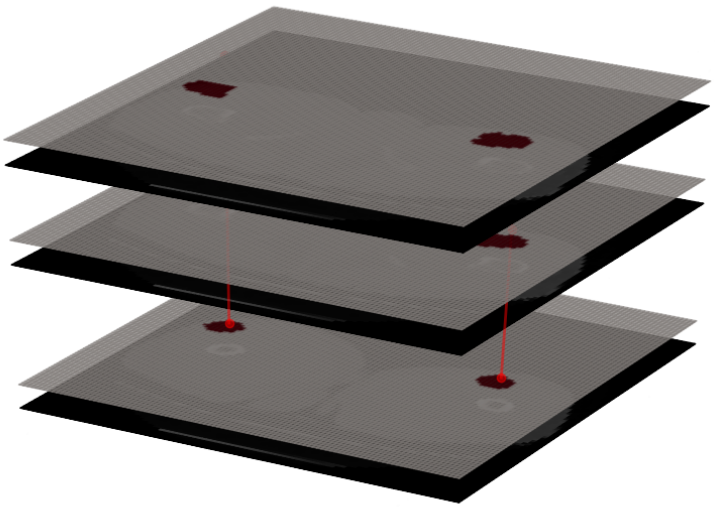


Figure 2.27: Naive segmentation pipeline applied to a series of CT slices. The thresholding algorithm combined with morphological operations allows to obtained a naive segmentation of the femur bone. The centroid of the segmented connected components is used to filter the false positive results. This pipeline was used to simplify the annotation procedure of the CT dataset.

The annotation was performed using a semi-automatic approach. We developed a custom image processing pipeline, applying a combination of thresholding and morphological operations to extract as better as possible the bone structure from each CT frame (identified as a pixels connected component). The thresholding operation produced many false positives into a single image which had to be filtered. We could (reasonably) assume that the femur position did not change between two following images. Thus, once the multiple pixel connected components were identified, we filtered them according to their relative position into the image: each group of pixels obtained by thresholding had its own centroid, which it remained quite the same also into the next slice (ref. Fig.2.27). An interpolation of these components was performed to filter only the femur parts. This method worked quite good when we considered slices far from the femur head: when the acetabular fossa became very close to the femur head, the two components were not divided. An example of this kind of issues is shown in Fig. 2.28.

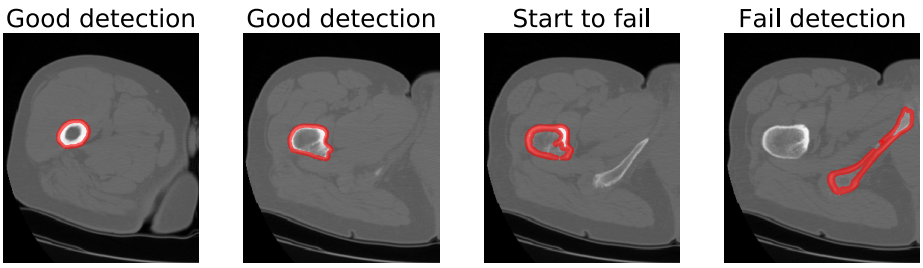


Figure 2.28: Example of automatic segmentation using custom image processing pipeline. Starting from the bottom of femur bone the detection seems good but when the method starts to fail the failure is propagated to the next slices. The method is too naive to perform a good segmentation on the full set of slices. However, it can be useful to reduce the quantity of slices to annotate manually.

This image processing is just a naive approach and it could not solve the full segmentation task, but it can be considered as a good preliminary tool to produce annotated images. With it we reduced the amount of required annotations by more than 50%. The other part of the images were manually annotated. The manual annotation was performed without any medical background and thus we can not ensure the goodness of our results. This work only aims to proof the possibility of using deep learning techniques to face segmentation problems.

Following this approach we have been able to annotate 104 CT images randomly sampled from the 4 patient slices. In particular, we have extracted 40 slices from a single patient and 96 from the remaining three. In this way we could use the 96 images as training set (applying the told above image augmentation pipeline) and the 40 remaining slices as test set. We chose to use a single patient slices as test set because with the output generated by the U-Net model we want to reconstruct the (approximated) 3D structure of the femur bone. The 3D reconstruction is still in work in progress and we will not discuss about it in the obtained results.

2.4.3 Results

We implemented the U-Net model and data augmentation pipeline using Tensorflow framework. We did not use our Byron or NumPyNet libraries since the training section is very computational expensive and in this project we had the possibility to use a NVidia GeForce RTX 2080 Ti, which can be easily managed using a Tensorflow implementation[54]. The training performances in terms of loss (*binary CrossEntropy*) and accuracy (we have already mentioned that it is not a good estimator in segmentation tasks, but it is "required" in standard training plot) are shown in Fig. 2.29.

As can be seen in the left plot of Fig. 2.29 the binary cross-entropy loss tends to saturate just after the 40th training epoch and in the same way also the accuracy score reaches its plateau (notice that the starting value of the accuracy score is more than 93% and it proves the incompatibility of this metric for segmentation problems).

Using the weights obtained by the training step we validated our model on the 40 test images. We fed our Neural Network model with each CT slice and we filtered the output[55] using a thresholding of 10^{-2}, i.e values less or equal to the threshold were turned off. From each slice the IoU score was computed taking the corresponding ground truth, i.e

[54] We thank the *PhySyCom* group of the Bologna University for its support on this project and for the availability of its computational resources.

[55] The model output is a floating point images with values ranging from 0 to 1. To compare the output with a binary mask we have to apply a thresholding procedure to binarize the image.

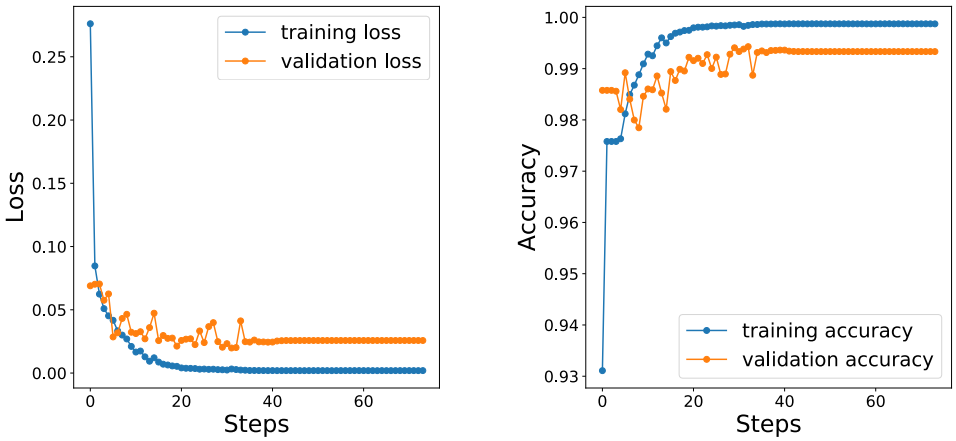


Figure 2.29: U-Net training scores in terms of loss (binary cross-entropy) and accuracy score. After approximately only 40 epochs both the measures reached their plateaus. In the same way also the validation score (computed over the test set) saturates.

the binary mask extracted with our semi-automatic (and not medically accurate) pipeline. In Fig. 2.30 we show the distribution of IoU score over the 40 test images.

The major part of the test slices obtained a IoU score greater then 0.8 which corresponds to a good agreement between U-Net output binary mask and the corresponding ground truth. Only a 20% (10/40 slices) of the test slices have shown a IoU score less than 0.8 and thus a binary mask quite different from the desired output. In Fig. 2.31 we show some of the "good" results obtained using our trained model.

Despite the first slice showed in Fig. 2.31 could be easily segmented also by our custom image processing pipeline (the bone extraction in this case is quite easy) the second two slices show more issues: it is hard to discriminate between femur head and acetabular fossa when the two components are so much close each other. In all these cases the U-Net model is able to discriminate between the two bones with a good agreement with our naive ground truth. The model still produces some false positive segmentations in these cases: the output could be corrected reapplying our image processing pipeline and thus filtering the bone identifications in disagreement with the connected components centroids obtained by the previous slice. To completely proof our results we need of more data and certainly more annotated slices, but these preliminary results encourage us to use Neural Network models, as U-Net, to face also this task.

2.5 Replicated Focusing Belief Propagation

Up now we have implicitly talked about Neural Network models based on the standard updating rule of back-propagation. Other learning rules for weight updates have been proposed and the choice of the best one it is still an open problem. The final purpose is to obtain a feasible learning rule ables to model the biological learning of the human brain.

The learning problem could be faced through statistical mechanic models joined with the so-called Large Deviation Theory. In general, the learning problem can be split into 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 ones seem to be the spin-glass models based. Starting from a

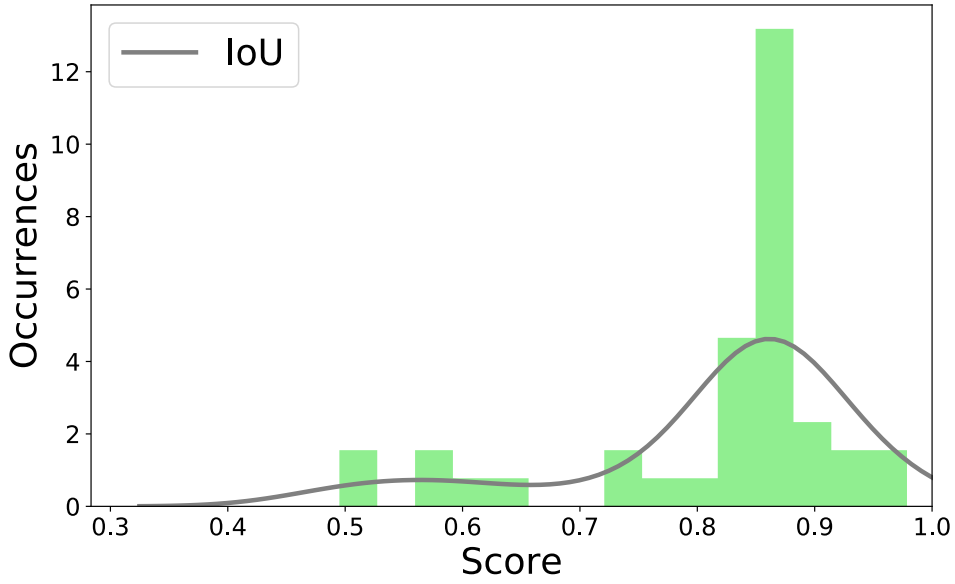


Figure 2.30: IoU (*Intersection over Union*) distribution obtained on the test set. The IoU score quantifies the agreement between U-Net output binary mask and the corresponding ground truth. A perfect match corresponds to an IoU score equal to 1 and a completely disagreement is given by a null value of IoU score. The 80% of the test set has obtained a IoU score greater than 0.8 and thus a good correspondence between our results and the ground truth. We would stress that the ground truth was obtained applying a custom semi-automatic image processing pipeline which has not validated from a biomedical point-of-view.

balanced distribution of the system, generally based on Boltzmann distribution, and under proper conditions, we can prove that the classification problem becomes a NP-complete computational problem. A wide range of heuristic solutions to that type of problems were proposed.

In this section we show one of 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 proved theoretical efficiency, the applications on real data are still fews. Thus, we show the application of the optimized version of the rFBP algorithm on a Genome Wide Association (GWA) dataset provided by the European COMPARE project. This work was also presented on the 2019 CCS-Italy (Conference of Complex System) [29].

2.5.1 Algorithm Optimization

The rFBP algorithm is a learning algorithm developed to justify the learning process of a binary neural network framework. The model is based on a spin-glass distribution of 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 of the code was written in Julia language and, despite it is a quite efficient implementation, the Julia programming

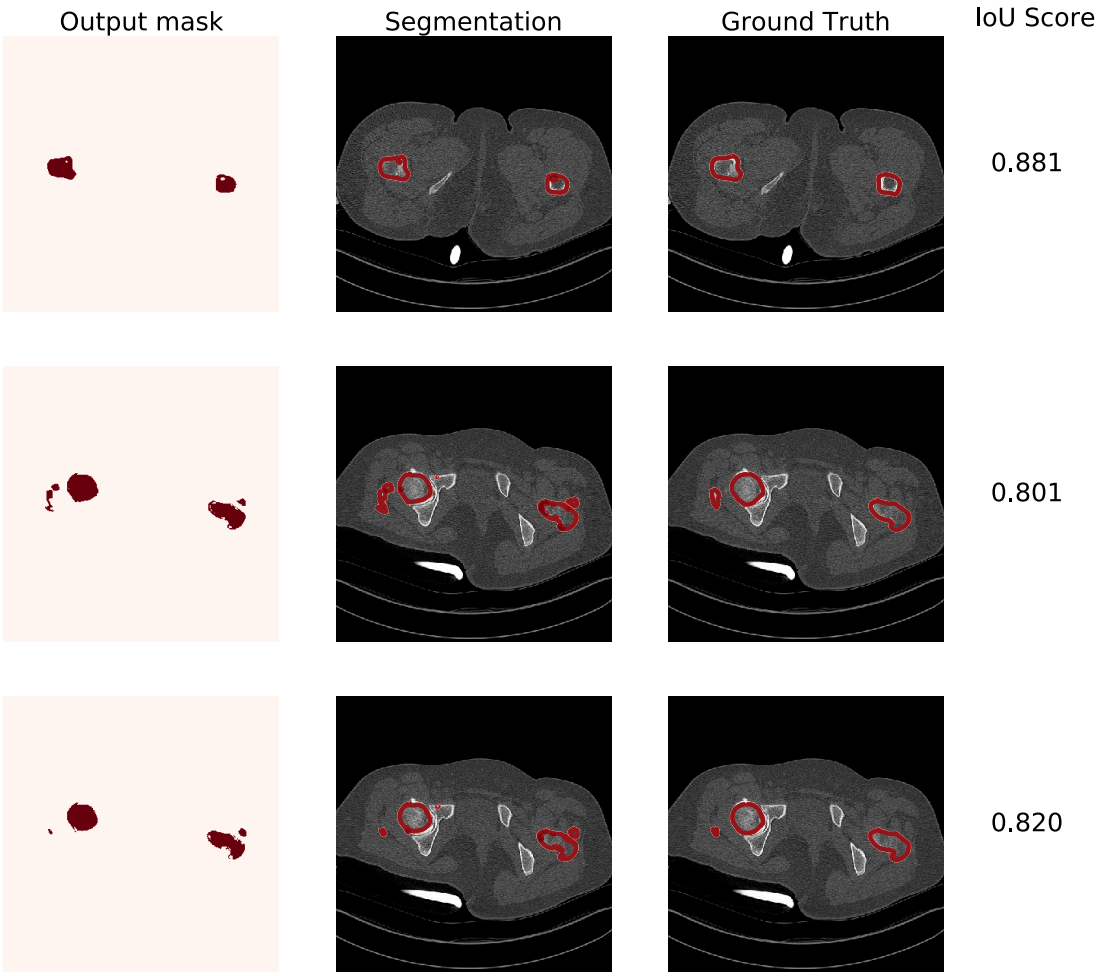


Figure 2.31: Output mask of trained U-Net model and corresponding ground-truth and IoU score. **(first column)** U-Net model output after a thresholding equal to 10^{-2}. **(second column)** Superposition of the original image with the generated binary mask. **(third column)** Corresponding ground truth of the CT slice. **(fourth column)** IoU (Intersection Over Union) score between the model output and ground truth slice.

language stays on difficult and far from many users. To broaden the scope and usage of the method, a C++ implementation was developed jointly with a Cython wrap for Python users. The C++ language guarantees better computational performances against the Julia implementation and the Python version enlarges its usability. This implementation is optimized for parallel computing and is endowed with a custom 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 try to encourage researchers to approach these alternative algorithms and to use them more frequently on real contexts.

As 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 syntaxes to perform dynamic specializations of the methods between two magnetization versions of the algorithm. The main object categories needed by the algorithm are wrapped into handy C++ objects, easy to use also from the Python interface. A further optimization is given by the reduction of the number of the 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, minimizing the call stack to ease the vectorization of the code.

The full rFBP library is released under MIT license and it is open-source on Github [22]. 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.

The Python wrap guarantees also a good integration with the other common Machine Learning tools provided by the scikit-learn Python package; in this way we can use the rFBP algorithm as equivalent alternative also in other pipelines. Like other Machine Learning algorithm also the rFBP one depends on many parameters, i.e its hyper-parameters, which have to be tuned according to the given problem. The Python wrap of the library was written according to the scikit-optimize Python package to allow an easy hyper-parameters optimization, using the already implemented classical methods.

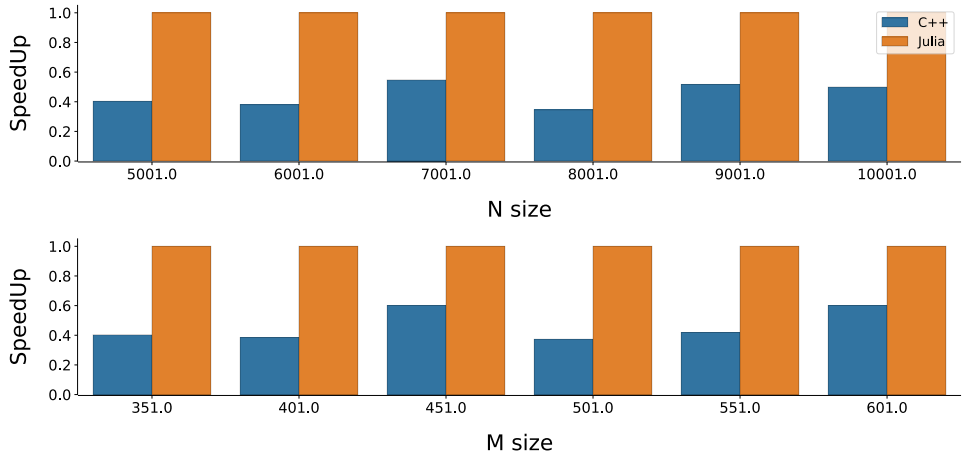


Figure 2.32: Comparison of time performances between the two available implementations. In orange the execution time of the Julia implementation (reference) provided by the original paper of Baldassi et al. In blue the execution time of our Cython version provided in the rFBP package. The simulations were performed varying the input dimension sizes (number of samples, M, and number of variables, N). For each input configuration 100 runs of both algorithms were performed and the results were normalized by the Julia implementation result. In these cases we fixed the magnetization to **MagP64**.

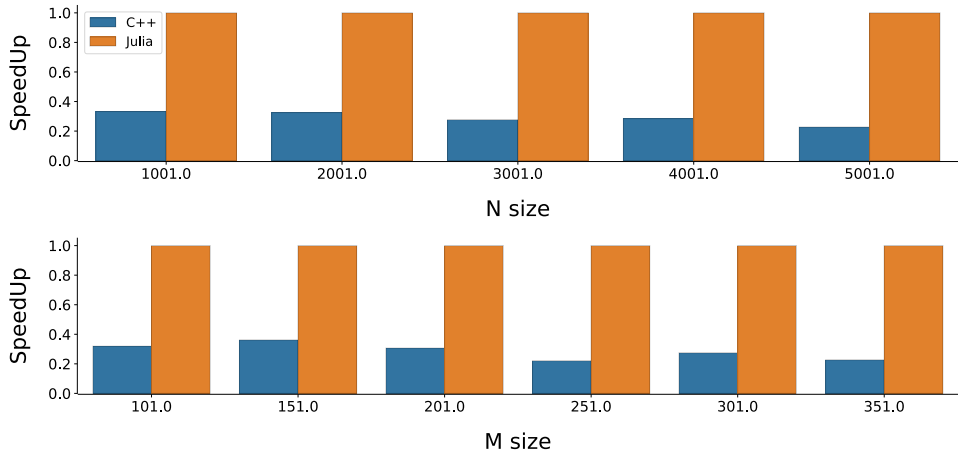


Figure 2.33: Comparison of time performances between the two available implementations. In orange the execution time of the Julia implementation (reference) provided by the original paper of Baldassi et al. In blue the execution time of our Cython version provided in the rFBP package. The simulations were performed varying the input dimension sizes (number of samples, M, and number of variables, N). For each input configuration 100 runs of both algorithms were performed and the results were normalized by the Julia implementation result. In these cases we fixed the magnetization to **MagT64**.

We firstly test the computational efficiency of our implementation against the original Julia one. The tests were performed comparing our Cython version of the code (and thus with a slight overhead given by the Python interpreter) and the Julia implementation as reference. Varying the dimension sizes (number of samples, M, and number of variables, N) we tested the time efficiency over 100 runs of both the algorithms. We divided our simulation according to the two possible types of magnetizations (MagP64 and MagT64 as described by the original implementation available here) and the obtained results are shown in Fig. 2.32 and Fig. 2.33, respectively.

As can be seen by the two simulations our implementation (blue bars in the Figures) always overcomes the time performances of the original one (orange bars in the Figures), taken as reference in the plot. However, we can not guarantee a perfect parallel execution of our version: also with multi-threading support the scalability of our implementation does not follow a linear trend with the number of available cores. In our simulation, in fact, we used 32 cores against the single thread execution of the Julia implementation but we gained only a 4x and 2x of speedup for MagT64 and MagP64, respectively. The network training is a sequential process by definition and thus it is hard to obtain a relevant speedup using a parallel implementation. In this case it is probably jointed to a not perfect parallelization strategy which bring to a not efficient scalability of our version. However, the improvements performed to the code allow us to use this algorithm with bigger dataset sizes.

2.5.2 SNP classification

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 limits the possible

applications[56].

Classification problems which involve 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 underlying disease. In this contest the mutations can be classified as SNPs (Single Nucleotide Polymorphisms). The biological classification of possible gene-mutations is certainly more complex than this rough description, but for the purposes of this work we can simply consider all kinds of variations as polymorphisms[57]. Thus, we can identify a genome as a sequence of its polymorphisms in relation to a reference one, i.e a sequence of two possible values given by the on/off of nucleotide mutation.

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 polymorphisms.

We tested the rFBP on 210 Salmonella enterica genome sequences, $4857450\,bp$ (base pairs) long, living inside animals. Our early goal was to discriminate bacteria which lives in pigs (159 samples) against to all the other animals (51 samples).

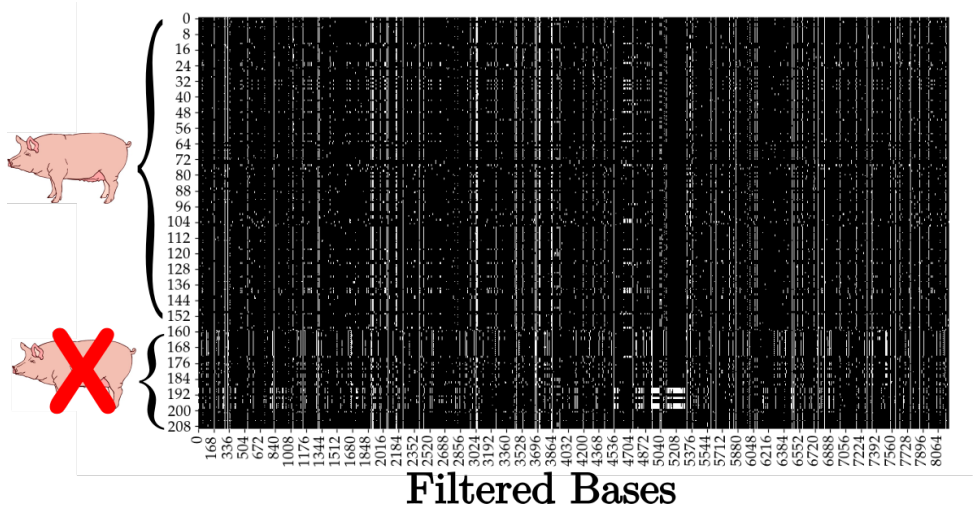


Figure 2.34: SNP sequences of Salmonella enterica samples used in this work. The x-axis shows the genome bases and the y-axis the corresponding samples (210 samples in total). The black dots are related to a base without polymorphisms (in relation to the genome reference), while the white dots are the polymorphisms (SNPs) identified. The first 159 rows contain the genome sequences related to pigs, i.e the sequences obtained by a pig which host the bacteria, and the following ones contain sequences of other animals. Also with naked eyes we can see the differences between the two data types.

First of all, we filtered our data removing from each genome a base if it showed a polymorphism in all the samples. In this way we reduced the number of bases to $8189\,bp$. A graphical representation of these samples is given in Fig. 2.34. 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 hyper-parameters of the

[56] 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 and it is not straightforward their conversion to binary values without loosing information.

[57] I apologize to the expert readers and biologists.

algorithm were tuned on the training set in relation to the performances obtained using an internal stratified 10-fold cross-validation: in each fold the training was performed using a sequence of hyper-parameters and the performances evaluated on the corresponding test set; the hyper-parameters configuration which obtained the best performances on the full training set was chosen as the best one. We used our custom Scorer library for the performances evaluation. Considering the unbalanced sample quantities, the Matthews Correlation Coefficient (MCC) was chosen as good score indicator for the evaluation.

2.5.3 Results

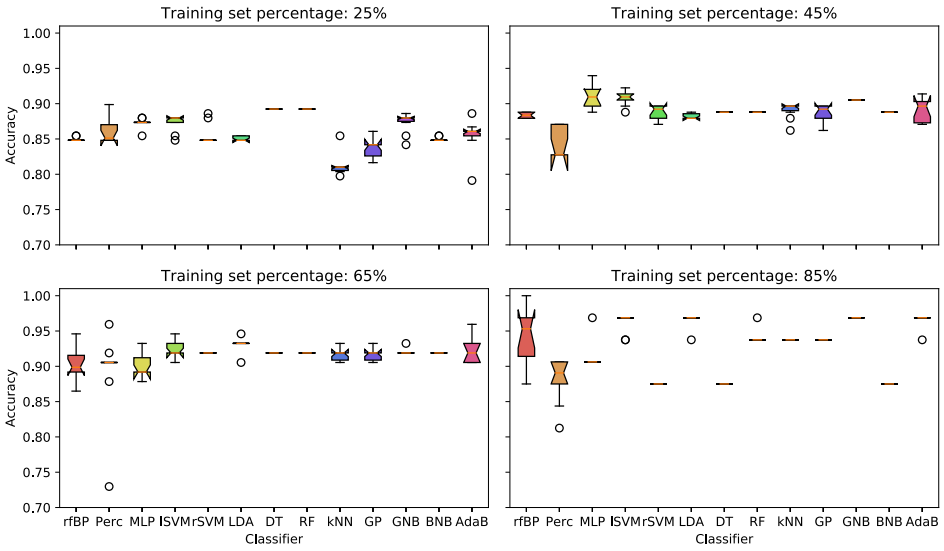


Figure 2.35: Accuracy score obtained on the validation set varying the training set size. We compared the trends of the whole set of classification algorithms used.

With the tuned hyper-parameters we performed the training of rFBP algorithm on different percentages of the training set: 25%, 45%, 65% and 85%. In the same way we trained also a list of the most common Machine Learning classifiers: simple 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 performed the optimization of the hyper-parameters of each classifier with the same number of optimization steps. In Fig. 2.35 2.36 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 was 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 were the only two algorithms which qualitatively does not show performance 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

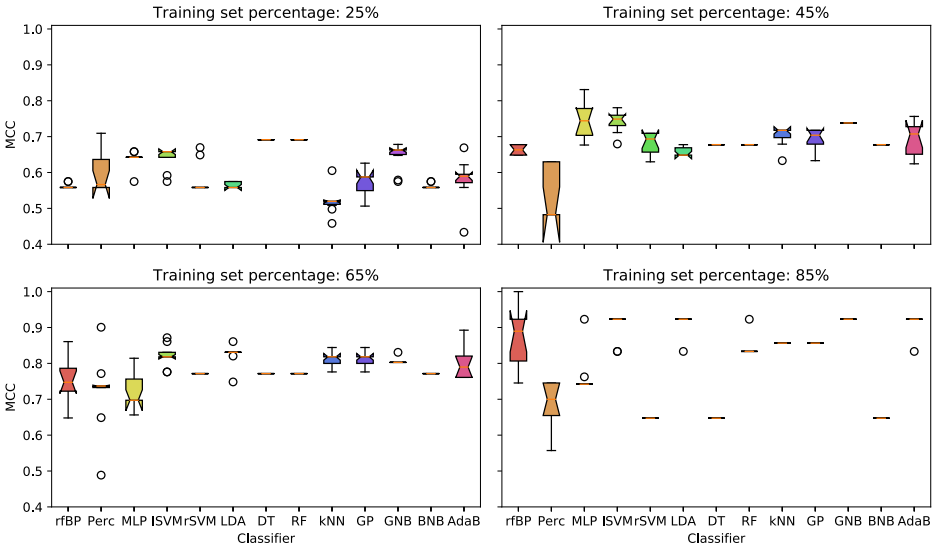


Figure 2.36: Matthews Correlation Coefficient (MCC) score obtained on the validation set varying the training set size. We compared the trends of the whole set of classification algorithms used.

two classes[58]. 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 [83] correction method and by the definition of significant threshold of 0.05 we found 1103 significant bases. An analogous χ^2-test was performed on the rFBP weights to identify a putative correlation between a set of weights and mutated bases. This second χ^2-test was performed only on the simulation which involved the 85% of data as training set because it was the case in which the rFBP algorithm shows the better performances than the other classifiers. We firstly defined a base as significant if its corresponding p-value was less or equal than 0.05: in this way we could associate to each base a numerical weight of 0 if it was not significant an $+1$ or -1 if it was, where $+1$ identified the pig class and -1 the other one. The set of weights defined following these instruction could be associated to the "ideal set". In this way we could ensure that if the corresponding rFBP weights were equal to $+1$ in all the significant positions (and thus in all the significant bases) for the pig class, the model output would be $+1$ and -1 in the opposite case. This mechanism follows the Simple Perceptron algorithm scheme (ref. 2.1.1) in which each weight is associated to a given entry of the input samples. The rFBP algorithm follows the same rules with an activation function given by the Heaviside Θ and it changes only the updating rule. Moreover, following this method we could ensure that only the 1103 significant bases extracted were associated to a not null weight.

We took into account the 10 weights set extracted by the 10-Fold cross validation performed to extract the previous results. From these 10 sets we extracted the representative one using a simple average of their values: each weight entry was computed as the mean of the 10 weight realizations. In this way each weight entry was converted to a floating point number and we can easily extract the set of weights perfectly equal to ± 1. From our analyses 5201 weights were consistently equal to ± 1 in all the simulations, i.e the algorithm assigned to 5201 weights always the same value. In this way we could consider these weight

[58] The contingency-matrix displays the (multivariate) frequency distribution of variables. Each row counts the number of hosts with/without the SNPs. Each column identifies a class.

entries as the significant positions identified by the rFBP algorithm.

These rFBP significant weight set could be compared to the χ^2 ideal set. From this comparison we noticed a good agreement between the two sets: the major part of the significant bases for the χ^2 multiple test could be found also in the significant weights identified by the 10 realizations of the rFBP algorithm. In particular, we found that 838/841 bases were significant for both the methods. The rFBP algorithm correctly identified 838/848 significant bases related to the -1 class and only 3/255 bases related to the $+1$ class.

In conclusion, we could prove that the rFBP algorithm is able to identify the major part of the significant polymorphisms in the training set. However, the use of the only training set to extract the significant weights certainly penalized the rFBP algorithm and a second simulation (without prediction purposes) was performed considering the full set of data, i.e 10 realizations without cross-validation. In this second case the rFBP significant weights correctly identified 702/1103 where 696/848 were related to the -1 class and 6/255 to the $+1$ class. In both cases we could conclude that the dataset did not contain enough information for the $+1$ class identification for the rFBP algorithm.

Following the above results, a final training was performed using only the significant bases identified by the rFBP algorithm and only the significant bases extracted by the χ^2 multiple test, using the full set of available classifiers. We noticed how the performances of all the classifiers are significantly better using the bases extracted by the rFBP algorithm (always over the 87% of accuracy) than the results obtained considering the χ^2 significant bases (only few classifier were able to obtain more than 85% of accuracy).

We conclude that our results highlight the efficiency of the rFBP algorithm for genome analyses and SNPs classification problems. Moreover we could propose also the rFBP algorithm as a valid feature selection alternative to classical statistical tests. These results also encourage us to further investigate about the biological meaning of the significant bases identified.

Chapter 3

Biomedical Big Data - CHIMeRA project

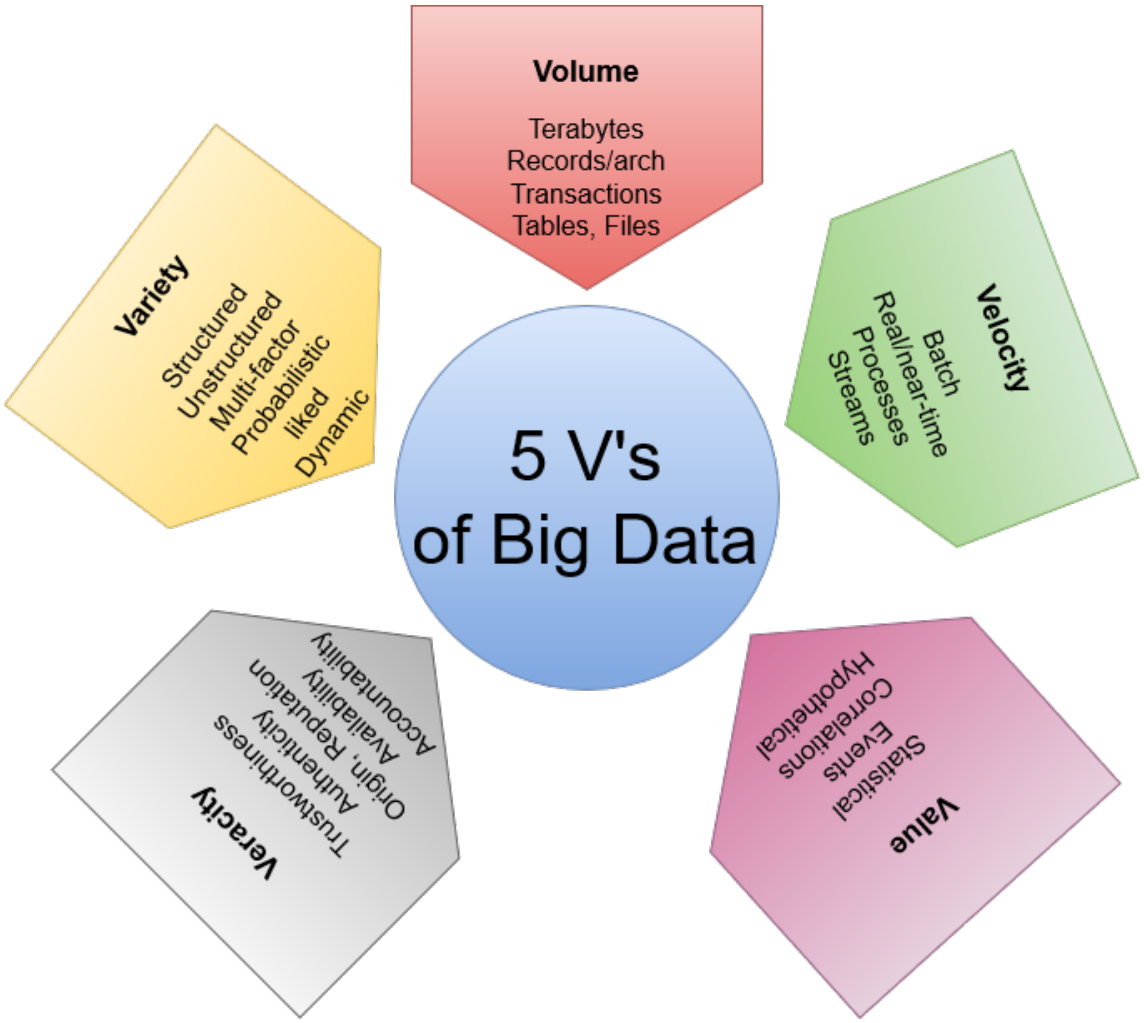


Figure 3.1: Big Data 5 V's

Every second a large quantity of data are produced and shared along the Internet and web-pages. Data are collected by social networks, chat messages, video streaming and images. Everyone, in fact, can easily create new data sources and share or put them in

Internet pages. The growth of these data is not limited to multimedia data, but it involves many different fields. This is one of the most important features of the contemporary time, the so-called Big Data era: this huge volume of data is making a new field in data processing, called Big Data Analytics, that nowadays is positioned among the top ten strategic technologies (Gartner Research, 2012).

It is still difficult to provide a definition of what exactly are Big Data and we can find many slight different nomenclatures and categories which aim to formulate its explanation. Moreover, Big Data does not define a particular data type, but more than we normally think sources can be labeled as it. The *International Journal of Computer Applications* defined them as "$[\cdots]$ a collection of large and complex datasets that can not be processed and analyzed using traditional computing techniques. It is not a single technique or a tool, rather it involves many areas of business and technology". This definition involves many aspects of Big Data processing, but it does not provide any definition about their nature. Moreover, it is easy to identify them as "big" and thus difficult to analyze, but they are around us every day and just using the Internet connection every smart-phone or laptop can extract and visualize our web queries. So could not be properly correct to define them in this way. However, it is sure that standard computing techniques have to be reviewed to face this vast amount of data and a even more important attention has to be paid on the algorithmic implementations.

While a global definition of Big Data is evidently difficult, we can provide a description of them using some of their "essential" features. One of the most common and used set of labels for this purpose is given by the so-called 5 V's of Big Data: volume, velocity, variety, veracity and value. Despite the first twos are quite obvious (Big Data are certainly *big* in volume and they are produced very *fast*), the remaining three need a particular attention. We have already treated problems related to the volume of data (ref. Chapter1) and the need of very fast processing and algorithmic optimizations (ref. Chapter2), so in this chapter we want to focus on the remaining three characteristics of Big Data Analytics.

As pre-announced, there are many different sources able to provide data and this feature describes the extremely heterogeneity and variety of them. We can, however, broadly classify this variety into three classes: *structured*, *semi-structured* and *unstructured* data. A dataset is *structured* if we can easily manage the information in it or, in other words, if it is described using a standard data format (it is "quearable"). On the other side, we have the completely *unstructured* datasets, where data are disorganized and we need one or multiple pre-processing steps before handle them. The intermediate format is given by the *semi-structured* data, in which only a part of them could be handled with standard techniques or we can lead them to their structured version. The organization of data has been a crucial task in this work of thesis and we will return on this topic in the next sections.

The fourth essential characteristic of Big Data is their *veracity*, due to data inconsistency and incompleteness. Data are shared very fast using Internet and we can find some ambiguities and/or deceptions between different data sources. If we want to merge and aggregate different kinds of information (harmonization), we have to face this kind of problems. The final task of every Big Data Analytic application is, in fact, to process large quantities of data and obtain a unique answer to a problem, which can not vary in relation to the portion of samples or datasets used.

The last, and probably most important feature, is certainly their *value*: it is good to have access to several data, but unless we can turn them into valuable information they are useless. In this vast amount of data only a small part of them can be considered as informative, and it is always hard to extract the informative core. Moreover, we have to take into account also the difficulties about the management of these data and their more or less complex structure. However, also in this case, it is hard to generalize this property

to all data stored in Internet: every day we see a large quantity of useless information in the Web and it is hard to figure out that some of them can be useful for research applications. A key role is played by the *questions* which we ask: for every data source, there is always an appropriate question which can be answer using it and which can give a value to it, and vice versa. In this way also the seemingly useless datasets can acquire importance for an appropriate research project.

In this chapter we are going to discuss about the latest project developed during my PhD and which is still in work in progress: the CHIMeRA (*Complex Human Interactions in MEdical Records and Atlases*) project. The project is an extension of a task of the INFN FiloBlu project (ref. next sections and Appendix E for further information about the FiloBlu project) which financed my last PhD year. CHIMeRA aims to create a unified database of biomedical records, using Natural Language processing techniques. Its final purpose is to merge multiple data sources available on-line into a single network structure, which highlights the relevant interactions between biomedical information, i.e starting from diseases to the biological agents and compounds involved into their causes and consequences. The realization of the first version of CHIMeRA has required a lot of time and the development of novel pipelines of data processing. The project does not still achieve its conclusions, but in this chapter we are going to cross through the main key points which allowed its construction.

3.1 The CHIMeRA project

The increasing availability of large-scale biomedical literature under the form of public on-line databases has opened the door to a whole new understanding of multi-level associations between genomics, protein interactions and metabolic pathways for human diseases. Many structures and resources aiming to such type of analyses have been built, with the purpose of disentangling the complex relationships between various aspects of the human system, relating to diseases [90, 44, 55]. Such structures, while allowing to study disease-to-some-other-omic associations, may not be sufficient when trying to bridge the gap of interpreting results and concepts proofing clinical studies, when many types of data are involved. Looking for causation of diseases across different omics has also became a major challenge, with the aim of expanding etiology and obtaining insights on pathogenesis [57]. This task may prove to be particularly hard when dealing with medical ontology strings, coming from different sources. Information of this type are usually provided by brief sentences and periphrases, while synonyms may occur to describe the same concepts, causing different data source to provide different relationships for similar instances. Text mining and string processing is becoming a required step when trying to exploit medical ontology as a bridge to diffuse information.

All these data come from different kinds of studies, performed by independent research groups, who want to prove their theory about a particular aspect of biological agent interactions. Modern biological analyses perform very capillary studies on biomedical agents, deeply studying the relationships between them, but loosing information about what there is around them. This approach is extremely efficient for the detection of the minimal causal agents of a problem, but it tends to loose its global and complex[1] behavior. This is the starting point of complex systems, i.e systems composed by multiple components with a mutual interactions between them. The study of an individual aspect, in fact, could give us only a partial overview of the system, but we have to take into account the interactions between its multiple components for a global description.

Network structures are acquiring even more importance on this kind of studies. Complex System and System Biology researches have proposed multiple models about the

[1] From a physical point-of-view.

dynamical and evolutionary interactions of the human system agents, aiming to study the hidden relationships between them using graph models. A network structure, in fact, is able to highlight and quantify non-trivial correlations between system components. The mathematical definition of network structures is given by the Graph Theory. We define a network/graph as a pair $G = (V, E)$, where V is a set of elements called nodes (or vertexes) and E is the set of their pairwise associations (links or edges). The total number of graph nodes (or cardinality of the graph) is denoted by N and it defines the order of the graph. The graph dimension is given by the number of its edges (m). We define a graph as *complete graph* if it has all its possible edges ($m = N \times N$). A network made by nodes of the same type could be described via its adjacency matrix, i.e a matrix ($N \times N$) in which each row/column identifies a node and each link e_{ij} quantifies the importance of the interaction between the node v_i and the node v_j. We define the importance of a node into the graph using the number of its connections: this is a classical *centrality measure* and it is called node's *degree* centrality. Starting from these definitions, we can enrich our model combining multiple network structure: given two graphs $G(V, E)$ and $G'(V', E')$, we define their combination as a new graph, where its nodes are given by the intersection of $V \cap V'$ and its edges are given by $E \cap E'$. If $V \cap V' = \emptyset$ the two graphs are *disjointed*; in contrary, if $V' \subseteq V$ and $E' \subseteq E$ then G' is a subgraph of G. Combining multiple graphs together, possibly including nodes of different types, we obtain a network-of-networks structure ables to map a wide range of interactions from multiple sets of elements. We will describe its properties later.

In real data applications, we can often reasonably assume that a wide amount of matrix entries are null, i.e the interaction between the involved agents is quite sparse, and we can use important properties related to sparse matrices to manage our network. However, when the amount of data increases, also the management of a such sparse matrix could be difficult. More efficient solutions are provided by modern Database formats and languages (e.g MySQL, SQLite, InfluxDB, $\cdots$), which store all the information into a binary format and they allow to submit queries to extract the desired portion of data. A global visualization of these huge amount of data is, in fact, without practical-sense and none valuable information can be extracted from the global representation of the system. The most important feature of network model is, in fact, the definition of a hierarchy of interactions: the relationship between two nodes is given by the amount of connections which links them or, in other words, by their paths. Starting from a node, its nearest neighbors are given by the set of nodes connected to it: re-iterating this concept we can explore all the network structure[2]. In this way we can study the interactions of each node at different precision orders and causalities.

In light of these considerations, we started to develop the CHIMeRA project (*Complex Human Interactions in MEdical Records and Atlases*), in which we aim to merge state-of-art studies and databases about biomedical researches into a unified network-of-networks structure. A key role on our network structure is played by diseases: the major part of biomedical researches are focused on causes and consequences of a given disease, involving the corresponding databases to store the interactions between them and other biological factors. Diseases are also the most bigger manifestations of biological malfunctions and a large part of the biomedical researches are financed on their study, looking for their fine grain causes. Thus, a disease could be a valid "bridge" between multiple data sources: merging disease-nodes derived from different datasets we can provide a unique structure which hosts all the information.

The crucial point of this project has been, in fact, the merging of different kinds of information provided by multiple distinct data structures. As told above, the major part

[2] We assume that our network structure does not have isolated nodes and it has only undirected connections.

of scientific researches have focused on a partial aspect of the problem and they provide an independent result from the others, reducing the possibility of interactions between the outputs. We tend to spend a lot of computational power to visualize the results using web pages and on-line services, but they drastically affects the real usage of these information. The CHIMeRA project began from these independent sources, aiming to maximize their overlap and, thus, the communications between them.

We have to pay a final attention about the format of these data: in physics we are friendly with numerical data, but in these contexts we have to work with words and text strings. The told above databases include only the outputs of various researches and the "interpretations" of the analyzed numerical data. For example, if a numerical significant correlation was found between a disease and a gene we would find an association between them into a database (modeled as a link in our network). The only information available into this database is a link between the two words, the disease name and the gene name, without any numeric value. While numbers have a unique representation (the number 42 is always 42[3]) we can use multiple periphrases, i.e set of strings, to identify the same concept. The biomedical community, in fact, has not yet provided a (public[4]) unified standard for disease identification or, at least, it has not yet provided a rigid standard as for other kinds of data as genes or SNPs. So, if the diseases could be an efficient way to link together multiple data sources, they throwback an extreme variability in their nomenclature. The CHIMeRA project has tried to overcome this issue, using a Natural Language Processing (NLP) approach.

In the next sections we are going to discuss about the multiple steps which lead us to the formulation of our unified CHIMeRA database. We will start from the preliminary studies performed in collaboration with the INFN FiloBlu project, which allow the creation of the SymptomsNet structure, i.e a "smaller" network based only on Italian words which links diseases to their related symptoms. Then we will briefly introduce the most common NLP techniques, also used into the CHIMeRA pipeline and, finally, we will show the main developed features of our CHIMeRA network.

3.2 How to find the data - Web Scraping

The INFN FiloBlu project was developed by the collaboration between the Physics Department of the University of Bologna and the INFN group of the Sapienza University of Rome. The project aims to implement a NLP pipeline to process messages with medical theme, helping doctor-patient interactions. Domiciliary care for oncology patients is preferred due to cheaper costs than hospitalization, and a more comfortable living for them. To successfully follow therapies during domiciliary care, the patient is in constant contact with health-care professionals and he is frequently monitored. Patients are interested in an actively collaboration to the management of their health and they are willing to use also ICT technologies. The FiloBlu project meets the citizens' needs developing a tool to optimize the efficiency and the effectiveness of care processes, developing two APPs (patient and medical sides) to support doctor-patient communication. The final purpose of the project is to process doctor-patient chat messages (using an interface similar to "WhatsApp"), computing from them a score related to the patient state. The APPs are equipped with features specifically designed for health-care applications and using a Natural Language Processing pipeline on the text messages they compute an "attention" score for each text message. The "attention" score is then used to rank the patients' messages on the medical-side APP, prioritizing (potential) critical situations.

[3] And it is certainly the right answer!

[4] For sake of completeness we have to mention the MedDRA database which is a pay-to-use repository of these information.

FiloBlu was financed by POR-FESR project in Lazio region in collaboration with the Sant'Andrea Hospital of Rome, so the project was developed only for Italian language communications. This constraint drastically affects the data availability, which are very hard to find on-line. Text message analysis concerns the evaluation of critical keywords and medical terms, so we faced this problem generating a diseases ontology. In particular, we are interesting in the relation between symptoms, diseases and their mutual interactions for the realization of our score function. More details about the pipeline used for the message processing are given in Appendix E - Neural Network as a Service.

The English is becoming the predominant language in the research community and it is really hard to find (enough) data in other languages: everyone who wants to share his data via Internet has to provide them in English if he wants to increase its visibility and availability. The Italian constraint posed by the project, drastically limits the data sources and no public databases were found. We would stress that as "database" we consider a publicly available set of structured data, which can be downloaded and easily used.

Surfing on Internet many web pages can be found about diseases and their interactions with symptoms and causes, the so-called *on-line doctor*[5] (or Medical Services) pages. An on-line doctor is a querable Internet service which allows user-auto-diagnosis based on the information inserted. The reliability of the information stored in these tools is only partially guaranteed by the service provider and, thus, it can not be considered as a scientific method for medical diagnosis. However, the amount of information collected by these applications is very interesting, and it can be used to simulate reasonable medical queries, needed by our project. Also in this case, it is important to notice that, despite the availability of these public information, the data are structured according to the web page needs and, moreover, there is not an immediate download availability of the raw data.

So, how can we obtain these useful information and re-organize them into a structured data format? The answer is given by the web-scraping techniques. With the term web-scraping we identify the wide set of algorithms developed to extract information from a website, or, more in general, from the Internet: while web-scraping can be done also manually, with this term we typically refer to automated methods. All the Internet pages are intrinsically pieces of codes written in different programming languages (HTML, PHP, $\cdot$). The major part of websites are written in HTML, an extreme verbose language, with more or less JavaScript supports. The way chosen to write a code and to reach the desired output is always left to the programmer: in these way we do not have a rigid standard (excepted by the programming language constraints) and in each website underlies a potential completely different ensemble of code lines. Thus, the realization of a web-scraper poses several issues to the programmer, who has to find underlying patterns inside the web page to get the information stored. In other words, the web-scraping technique is an emblematic example of Big Data Analytics algorithm, since it aims to extract a *value* from a large amount of *unstructured* information (raw website code).

A web-scraping algorithm is made by a series of multiple steps, which have to be performed automatically (without human overview). First of all, the algorithm has to recognize unique website structures: we can broadly summarize this task as the parsing of the underlying HTML code. Inside the large amount of code lines[6] are stored the useful information for our application. So, the algorithm should be able to detect relevant and interesting parts and filter them. Then it can easily reorganize the information into a usable data format and save the result.

There are multiple ways in which all these tasks could be addressed, and multiple open source libraries provide user friendly interfaces for the creation of own web-scraper. The

[5] Famous English applications are SteadyMD, MDLIVE, Sherpaa, LiveHealth Online and so on. Each service provides slight different information and the choice of the best one vary according to the user needs.

[6] Very large if we consider a pure HTML web page.

most common one (and also used in our applications) is the BeautifulSoup [75] Python package. This package provides a very powerful Python library designed to navigate and read website source codes. The integration of this library with other pre- and post-processing techniques allows the extraction of the desired information from a website and, moreover, their reorganization into a structured data format.

3.3 SymptomsNet

Find relationships between symptoms and diseases, and their reflections on system-wise perspectives such as genomics and metabolomics, still remains a crucial issue for medical research, but nonetheless an open one. The relation between symptoms and diseases can be used to see analogies and co-occurrences of different pathologies, including morbidity and co-morbidity. The construction of a unique and consistent database of these kinds of data is an open problem for the research community and a crucial task for many actual projects. The main problems arise from the complexity and heterogeneity of the available data and from the many nomenclatures used by different public databases. In many cases it is not so clear how to infer associations between symptoms and diseases, and, in addition, different data sources provide different connections. These information are stored as sentences and periods of variable length and we have to face the problem about different synonyms and periphrases used to describe same concepts.

In our work, we used large-scale public on-line databases to construct a bipartite network of human symptoms-diseases. A bipartite network (or *bigraph*) is a graph whose nodes can be divided into two disjoint and independent sets: the underlying adjacent matrix is rectangular ($N \times M$) and it describes the connections between N elements of the first type and M elements of the second one. We can always lead back to a square matrix ($N \cdot M \times N \cdot M$) using zero blocks for intra-group connections. We used common tools of natural language processing (see next sections for further informations about them) to clean and standardize data, to maximize the overlap between different data sources. After its construction, this network has been used to establish a score of different words based on node centrality measures. This complex map of associations can be used, also, to link other data sources and enrich the disease descriptions from other biomedical points-of-view.

Many on-line databases offer auto-diagnosis tools and search engine in which the user can insert a list of symptoms or diseases obtaining back the corresponding "diagnoses". While many international databases are quite consistent and supported by medical/biological research groups, the available data in Italian language are quite scarce.

Using the Italian version of the few public *on-line doctor* websites found, we obtained the needed information. We applied a set of custom web-scraping pipelines to several web pages to extract medical information, mainly focusing on sites which highlight relationships between symptoms and diseases. We would stress that the extremely variability of websites requires an equally varied set of web-scraping algorithms. Thus, for each web page taken into account a relative web-scraper was developed. As discussed above the Italian data sources are fewer than the English ones, so only three web pages have been taken into account in our analysis: My PersonalTrainer[7], SaniHelp[8] and Sapere.it[9]. All these three sites provide an organized series of tables which associate a disease to its corresponding symptoms, so they are easily to treat with web-scraping algorithms. These databases are not reliable from a scientific point-of-view and their vulnerabilities are shown also by a non-rigid labeling of the two classes: in multiple cases we found a disease as symptom of a different one, without a perfect agreement between the three data sources. Possible issues

[7] Arnoldo Mondadori Editore S.p.A.
[8] Terms and conditions available here.
[9] De Agostini Group.

related to an incorrect disease information could not be attributed to our web-scraping pipeline, but they should be already present into the original data which, we want remark it, they are the only Italian datasets publicly available and found.

The data extracted from the three websites cover a wide range of possible diseases and from each of them we obtained a network with a few thousand nodes, our SymptomsNet. The overlap of single words contained in "disease-sentences" was quite low, so a pre-processing was needed. Nodes were processed by standard natural language processing techniques, extracting word stems to maximize the overlap between sources. If two diseases showed different symptoms, we decided to concatenate the list of edges to not loose information.

The processed outputs create a network with 2 285 nodes and more than 29 000 links (only the 1% of the total number of possible links). The final SymptomsNet is reported in Fig.3.2, where node sizes are proportional to the number of their connections (Tab.3.1 for the top ranking links).

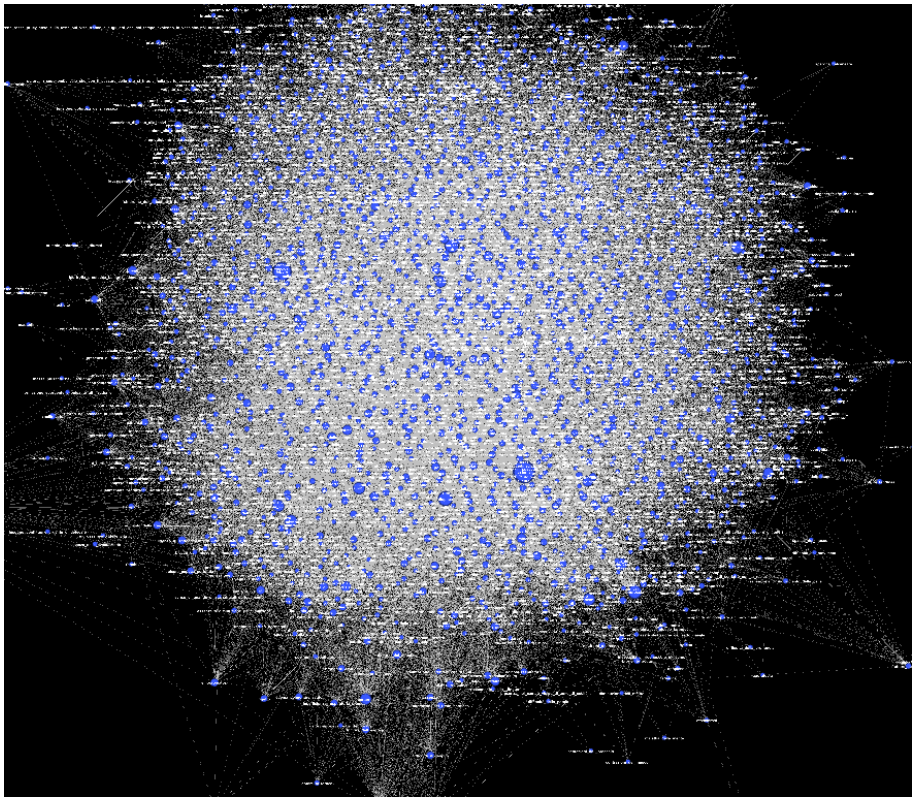


Figure 3.2: Symptoms-disease network generated by merging of three public Italian webpages of auto-diagnosis search engine. The network connects symptom and disease words according to the information found in the web sites. The network comprises 2 285 nodes and more than 29k links. Node sizes are proportional to their centrality (number of connections or degree score). In this way the most common symptoms/diseases are represented as the biggest nodes.

In this simple example we can already notice as the most central (big) nodes are associated to the most common symptoms-diseases, as expected. This result can be already interpreted as a validation of the performed processing. We can notice from Tab.3.1 that also in the top ranking nodes we find some diseases and related synonyms: this could be an issue for the network structure, since it means that the developed processing pipeline is not able to merge together different words with equal meaning. However, the project purpose was to create a reasonably good diseases ontology and this issue could be turned

Disease/Symptom	degree
Astenia	384
Febbre	313
Dispnea	225
Nausea	222
Anoressia	201
Ematemesi	193
Vomito	182
Debolezza	176
Affaticamento	176
Esaurimento	172
Mancanza Forze	168
Edema	158

Table 3.1: Top ranking links in SymptomsNet. We can notice "periphrases/synonyms" associated to same symptoms, as *Debolezza* and *Mancanza Forze* which are left to increase the heterogeneity of samples in the FiloBlu project.

to a strength of our applications: it proves that we have an agreement between different databases (synonyms have comparable degree score and thus same importance) and it highlights the variety of mined terms (different names which identify the same disease). In fact, this kind of occurrences allow to consider a wide range of possible synonyms in the score attribution, enforcing the text analysis required by the FiloBlu project: the node degree can be used as weight (1/degree) for text words, obtaining a simple score for the message given by the sum of the mapped keywords.

We conclude that from this very simple and preliminary work we are able to propose a novel symptoms-diseases network based on Italian public databases and, far as the author knows, no other equivalent results are reported in literature. This work allowed also the realization of a novel database obtained by the union of publicly available data. The extracted centrality measures can be used as weights for the corresponding symptoms/diseases and a valid input to model words frequency/importance in text analyses.

SymptomsNet is based on a bipartite graph which associates disease nodes to symptom ones. These results highlight the potentiality of such structures and they leaded us to further investigate about them and their creation. In particular, reiterating the same procedure we could be able to join together different bipartite graphs obtaining a network-of-networks structure which stores multiple types of information. This is the main idea behind the CHIMeRA project. To this purpose we have to manage more reliable data sources and improve our natural language processing pipeline to increase dataset overlaps. All these tasks can be easier performed using English words and validated databases. In the next sections, we are going to discuss about what natural language processing means in modern researches and we will describe the pipeline and databases used in the development of the CHIMeRA network-of-networks.

3.4 Natural Language Processing

Natural Language Processing (NLP) is a quite novel research field driven by the increasing availability of textual data (ref. Fig. 3.3). As told in the previous sections the incoming of Internet world exponentially increases the amount of data shared by people, and the major part of them are textual data, i.e data composed by words, phrases and,

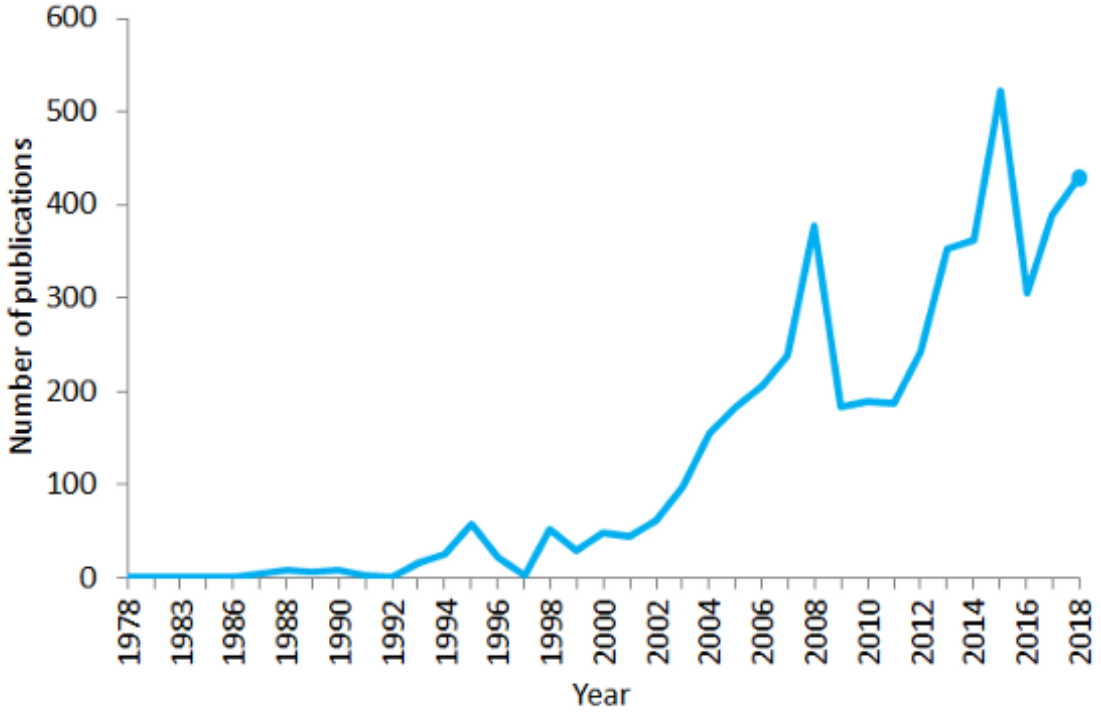


Figure 3.3: Number of publications containing the sentence "natural language processing" in PubMed in the period 1978–2018. As of 2018, PubMed comprised more than 29 million citations for biomedical literature.

more in general, texts. The NLP combines together techniques coming from linguistic, computer science, information theory and artificial intelligence researches. It concerns the interactions between human languages and computers or, in other words, it studies how a computer can analyze a huge amount of natural language data, extracting numerical information from them. This is a very hard task to perform since it is not straightforward to teach to a machine how humans communicate between them. A key role is played by artificial intelligence researches which develop new algorithmic techniques to face these problems.

Most of the modern NLP techniques are based on a Machine Learning approach: thus we can find statistical methods and deep learning algorithms which aim to solve these tasks. A first step to perform is the conversion of the human speech into a machine readable input; audio signals are so converted into string texts and only at this point the input can be analyzed from the machine. Applying this work-flow in forward and reverse mode we can perform a communication between a human and machines, and vice versa. In this section we will ignore how the conversion from human voice to numerical inputs could be performed and its related problems and solutions, focusing on the latest part of this pipeline, i.e in the description of the common techniques used to convert a string text into numeric values. This is also the case related to our CHIMeRA project, in which we have a huge amount of names and strings associated to medical terms and we want to standardize them increasing their overlap.

First of all, we have to take care that each human language has its own characteristics and thus it is hard to create a pipeline ables to process all the languages at the same time, while it is easier to tune an algorithm on a particular language. In our work we focused on Italian (SymptomsNet) and English (CHIMeRA Network) languages. Since SymptomsNet project has been developed as simple proof of concepts, the developed Italian pipeline

was really naive and, for sake of brevity, we will focus only on the CHIMeRA pipeline, i.e the English one. We would stress that in our application we are not interested on the understanding of words meaning, but we want to minimize the word heterogeneity, maximizing their overlap. Thus, we have ignored the semantic strings meaning and we have focused only on their syntaxes.

The syntax is the set of rules, principles and processes that govern the structure of sentences in a given language. We can create groups of words applying grammatical rules: grammatical rules have to be converted into algorithms which take in input a word and give in output a processed version of it. In this case there is not a numerical output, but just a reorganization of string letters and words. The most common techniques involved in syntactic analysis are:

- **Sentence breaking:** it divides a continuous text into sentences placing boundaries.
- **Word segmentation (tokenization):** it splits a large set of continuous text into units.
- **Parsing:** it provides the grammatical analysis of the provided sentence.
- **Morphological segmentation:** it splits words into individual units called morphemes.
- **Part-of-speech tagging:** it finds the grammatical parts of speech for every word.
- **Lemmatization:** it reduces the inflectional forms of a word into a single form.
- **Stemming:** it cuts the inflected words to their root form.

All these algorithms are very similar each other, so to better understand their functionality is useful an example. Let start from a useless text taken from the NLP Wikipedia web-page:

Listing 3.1: Original text

```
text = "Natural language processing (NLP) is a subfield of linguistics,
    computer science, information engineering, and artificial intelligence
    concerned with the interactions between computers and human (natural)
    languages, in particular how to program computers to process and
    analyze large amounts of natural language data. Challenges in natural
    language processing frequently involve speech recognition, natural
    language understanding, and natural language generation."
```

First of all we notice that the text is made by two sentences, that can be broken using a *sentence breaking* algorithm. In this way, we obtain a list of two strings given by

Listing 3.2: Sentence breaking

```
sentence_1 = "Natural language processing (NLP) is a subfield of
    linguistics, computer science, information engineering, and artificial
    intelligence concerned with the interactions between computers and
    human (natural) languages, in particular how to program computers to
    process and analyze large amounts of natural language data."

sentence_2 = "Challenges in natural language processing frequently involve
    speech recognition, natural language understanding, and natural
    language generation."
```

Now, we can divide each sentence into its set of words, using a word *tokenization*. Focusing only on the first sentence, we obtain in output:

Listing 3.3: Tokenization

```
tokens = ['Natural', 'language', 'processing', '(', 'NLP', ')', 'is', 'a',
    'subfield', 'of', 'linguistics', ',', 'computer', 'science', ',', '
    information', 'engineering', ',', 'and', 'artificial', 'intelligence',
    'concerned', 'with', 'the', 'interactions', 'between', 'computers', '
    and', 'human', '(', 'natural', ')', 'languages', ',', 'in', 'particular
    ', 'how', 'to', 'program', 'computers', 'to', 'process', 'and', '
    analyze', 'large', 'amounts', 'of', 'natural', 'language', 'data', '.']
```

There are multiple useless tokens in the processed list and we can filter them using a type of *part-of-speech tagging* algorithm, which removes the so-called *stop words* and punctuations. In our example our list of tokens becomes

Listing 3.4: Filtering stop-words and punctuations

```
tokens = ['Natural', 'language', 'processing', 'NLP', 'subfield', '
    linguistics', 'computer', 'science', 'information', 'engineering', '
    artificial', 'intelligence', 'concerned', 'interactions', 'computers','
    human', 'natural', 'languages', 'particular', 'program', 'computers', '
    process', 'analyze', 'large', 'amounts', 'natural', 'language', 'data']
```

A final processing could be given by a *stemming* algorithm, which extracts the root form of each word. Using a stemmer on the previous set of words we obtain

Listing 3.5: Stemming

```
tokens = ['natur', 'languag', 'process', 'nlp', 'subfield', 'linguist', '
    comput', 'scienc', 'inform', 'engin', 'artifici', 'intellig', 'concern'
    , 'interact', 'comput', 'human', 'natur', 'languag', 'particular', '
    program', 'comput', 'process', 'analyz', 'larg', 'amount', 'natur', '
    languag', 'data']
```

As can be seen by this example, the stemming algorithm converts in lower case each letter of each word and it removes the inflections from each of them. This is a very naive example, but we can already notice as our processing allows to merge multiple words together. In the original sentence we have the word "*Natural*" (with capital letter) and two occurrences of "*natural*" (lower case). Moreover, we have three occurrences of the "*computer*" word, but only two of them are in singular form. The tokenization + stemming processing allows to compare different word forms making them compatible.

Combinations of these algorithms can be found in everyday applications, starting from email assistants or website chat box, to the more advanced sentiment analyses and fake news identifiers [85, 30, 80, 91]. NLP pipelines are used also in biomedical applications and modern multinational companies like Amazon, IBM or Google are financing different kinds of research on this topic. Amazon Comprehend Medical is a NLP service developed by Amazon to extract disease conditions, medications and treatment outcomes from patient notes, electronic health records and other clinical trial reports. At the same time, also companies like Yahoo and Google base their filters and email classifiers on NLP algorithms to stop email-spam. Also the hot topic of these last years about the fake news is faced using NLP pipelines and the NLP Group at MIT is developing new tools to determine if a source is accurate or politically biased based on text analyses.

In our applications we built a custom pipeline based on a part of the described above functions. In the following sections we will describe in detail our pipeline: we would stress that the efficiency of our pipeline could not be generalized to other datasets, since our purpose was to obtain the best result for our application. In other words, we had fine-tuned our pipeline based on the data used in this project. Moreover, we have to clarify that our pipeline is not fully-automatic, but it was made according to a semi-supervised approach: we customized the work-flow following the encountered issues.

3.5 CHIMeRA datasets

We have seen how we can extract useful information also from unstructured databases using a web-scraping pipeline. The *on-line doctor* web pages could be very useful for a toy model application like the SymptomsNet, but if we want to produce scientific relevant results, we have to take care to the validity of data. Since English datasets availability is easier than the Italian ones, we moved to more "robust" databases.

As told in the previous sections, there are a lot of studies performed on disease associations to other biomedical agents and in many cases the resulting datasets are publicly available on Internet. This is the case of DisGeNET [34] and DrugBank [58] datasets, which contain relationships between a large number of diseases with genes/SNPs and drugs (and other information), respectively. DisGenet web-page allows to download the dataset already stored into a well structured network format (sparse adjacency matrix, with 210 498 associations between 117 337 SNPs, 10 358 diseases and 17 549 genes), while DrugBank poses more issues to the treatment of data: DrugBank was designed to provide a large set of information related to each drug using its own website and thus it needs a huge preprocessing of the JSON dataset structure to highlight all the possible network associations (14 812 drugs, 649 metabolite pathways, 3 256 gene targets, 40 SNP targets, and 532 food interactions). Using DisGenet we can connect diseases to their related genes and SNPs. From the reviewed format of DrugBank, instead, we can link each disease to the associated drugs. Associated to each drug we have also a list of gene and SNP targets, which can be merged to the information provided by DisGenet. Moreover, we can insert also food interactions, metabolite pathways and drug interactions (synergies or not) extracted from DrugBank. We would stress that, despite the trivial overlaps between the same data types (genes, diseases and SNPs up to now), just using the rearrangement of these pairs of databases into a network structure, we can already provide a possible extrapolation of the underlying information, using the paths between nodes. Starting from a disease inside DisGenet, using a single-database approach we can study "causality" relationships with the connected genes or SNPs. Using a multiple-databases (or a network-of-networks structure) approach, we can map that disease to other kinds of information like drugs, foods and metabolite pathways. The purpose of a such network-of-networks structure is to unveil relationships hidden by the underwhelming overlap between single-type information across different databases. The set of different information merged can be exploited for applications such as wide-scale drug effect evaluation and design addressing general diagnostic questions for systems medicine and diseases etiology expansion. In other words, a network-of-networks structure allows the inference of the missing connections using node contraction. A full list of the information collected by our web-scraping and rearrangement pipelines into the CHIMeRA database is shown in Tab. 3.2.

To enlarge our disease information we looked at other on-line data sources. A very interesting database is given by HMDB [87] (*Human Metabolite Data Bank*), which comprises a vast amount of metabolites and metabolite-pathways with the associated drugs and diseases (114 003 metabolite entries, with chemical taxonomies and ∼25 000 human metabolic and disease pathways[10]). The interconnections with the previous discussed datasets are straightforward, but in this case the data are not publicly available and we needed to apply a web-scraping algorithm to get its information. An analogous procedure was applied to extract the data stored into RXList database. RXList is an on-line website very similar

[10] The human metabolite-pathways can be divided into different types according to the informations stored in the HMDB dataset. The interactions between HMDB and DrugBank is already established through a vast series of hyper-links which connect them using metabolites and metabolite-pathways information. In this way we mapped also the information related to metabolite-pathway types to DrugBank dataset, obtaining a finer grain nomenclature and classification of these data. These information can be used to improve our disease description. In Tab.3.2 is shown only the aggregated data.

	disease	drug	food	gene	metabolite	phenotype	SNP	metabolic pathway
disease	CTD RXList SNAP	RXList	×	DisGeNET	HMDB	CTD	DisGeNET	HMDB
drug	RXList	DrugBank	DrugBank	×	×	×	×	DrugBank
food	×	DrugBank	×	×	×	×	×	×
gene	DisGeNET	×	×	×	×	×	×	×
metabolite	HMDB	×	×	×	×	×	×	HMDB
phenotype	CTD	×	×	×	×	×	×	×
SNP	DisGeNET	×	×	×	×	×	×	×
metabolic pathway	HMDB	DrugBank	×	×	HMDB	×	×	×
# nodes	63974	35161	532	18799	114100	13214	117337	1329

Table 3.2: Description of the data mined by the CHIMeRA project before merging. The datasets were collected using custom web-scraping pipelines and by a rearrangement of the public data. For each pair of data types we report the list of datasets used to evaluate the interaction.

to the previous discussed auto-diagnosis tools, where we can find associations between diseases, drugs and other several pathogenic associations. In this case we have a further distinction between diseases: we have diseases related to drugs and diseases connected to other caused-diseases. We have taken care of this kind of associations using directional links[11]. We remark that each web-scraping pipeline has been customized according to a precise website, so for each analyzed case a different code has been developed to address the data extraction.

All these information can enrich our database and the description of a given disease, but we have to face the problem of data merging. As previously discussed, we do not have a unique nomenclature for diseases and we found analogous names (periphrases or synonyms) which identify the same concept (disease). A useful tool to overcome these issues is given by a synonym dictionary: a powerful example is the CTD [39] (*Comparative Toxicogenomics Database*, 7 212 diseases with mapped synonyms and 4 340 diseases with related phenotypes) database. Using CTD jointly with SNAP [92] (*Stanford Large Network Dataset Collection*, 8 803 disease terms with related synonyms) database we could enlarge the number of synonyms associated to each disease name.

We remember that the crucial point of our merging procedure is given by the disease nodes, since they are the node types shared along (almost) all the databases. The help given by the synonym dictionaries increases the overlap between the mined datasets, but we chose to maximize it using a further NLP pipeline. We began our pipeline using a word *standardization*, i.e converting all words into their lower case formats and replacing all punctuation characters with a unique one[12]. Then, we noticed that a not negligible part of

[11] For sake of clarity, we encountered the same discrimination also into DrugBank dataset, in which we had intra-drug connections related to synergies or not in the use of multiple drugs together.

[12] An unexpected issue arise in this step: different databases use different enumeration systems. In some entries we found disease names associated to numbers which identify their multiple types. An example could be "Polyendocrine Autoimmune Syndrome type 1" but at the same time in a second database the same disease could be represented by "polyendocrine autoimmune TYPE I". Despite the global differences between the two names, given in this case by upper- and lower-cases of some letters and the deletion of some words, a very critical odds is the enumeration style. The performances of our pipeline dramatically increased using a roman_number_converter algorithm.

words involved into disease names was useless for the description: words like "syndrome", "disease", "disorder", "deficiency", $\cdots$ are not informative and they can be ignored (filtered). Then, we split disease names into a series of token according to the list of words which compose them (*tokenization*) and sort them.

To further increase the overlap we transformed inflected words to their root form, using a *stemming* algorithm: the stemmer strength has to be tuned according to the desired result. A first processing was performed using a Lancaster stemmer (more aggressive). If the resulting output was too short to be compared with other names, the starting token was processed by a Porter Snowball stemmer (less aggressive). The choice of the stemmer algorithm is a very crucial task for NLP, because, using it, we irreversibly loose information. Other processing steps were performed for critical cases encountered during the analysis: these steps constrain our pipeline and they tuned it for the underlying application.

The work-flow output includes multiple false-positive matches: the pipeline performs a brute force processing and some information lost along the steps could be significants. These cases lead to having multiple processed (same) names belonging to several (different) diseases: an example is shown in Fig. 3.4. Considering the original name and the processed one (pipeline output), we merged two names using a score match. This can be achieved introducing the standard word metrics: a common distance between words can be evaluated using the *Levenshtein Distance* which follows the equation

$$d_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0 \\ \min \begin{cases} d_{a,b}(i-1,j) + 1 \\ d_{a,b}(i,j-1) + 1 \\ d_{a,b}(i-1,j-1) + 1_{(a \neq b)} \end{cases} & \text{otherwise} \end{cases} \tag{3.1}$$

where a and b are two strings of length $|a|$ and $|b|$ respectively. The *indicator function* $1_{(a \neq b)}$ is equal to 0 when $a_i = b_j$ and 1 otherwise. In this way the Levenshtein distance between a and b evaluates the distance between the first i characters of a and the first j characters of b. Despite the apparently complexity of the mathematical equation, the *Levenshtein Distance* is a particular case of the more general *Edit Distance*, i.e a way to quantify how dissimilar two strings are to another by counting the minimum number of operations required to transform one string into the other. Also in this case an example could be more explanatory: given the two strings "*kitten*" and "*sitting*", their *Levenshtein distance* is equal to 3, in fact

1. **k**itten $\rightarrow$ **s**itten (substitute "s" for "k")
2. sitt**e**n $\rightarrow$ sitt**i**n (substitute "i" for "e")
3. sittin $\rightarrow$ sittin**g** (insert "g" at the end)

Using the Levenshtein formula we evaluated the distances between two original names and we associated the disease to the higher scorer. A summary scheme of our pipeline is shown in Fig. 3.4.

The described NLP pipeline further increases the database overlaps (e.g CTD-SNAP 24.17%; DisGenet-RXList 19.78%). We manually supervised the merging procedure taking care to reduce the false positive percentage. In some cases, the overlap percentage remained low also after the application of our pipeline (e.g. RXList-HMDB 8.03%; SNAP-HMDB 0.39%). Different data sources could be focused on different types of information and it is therefore reasonable to assume that, in some cases, the overlap is low. We supervised these critical cases with a manual check and we demonstrated our hypothesis. This behavior supports our choice of the databases: they include complementary information, which could improve the informative power of our structure. At the same time, this result also

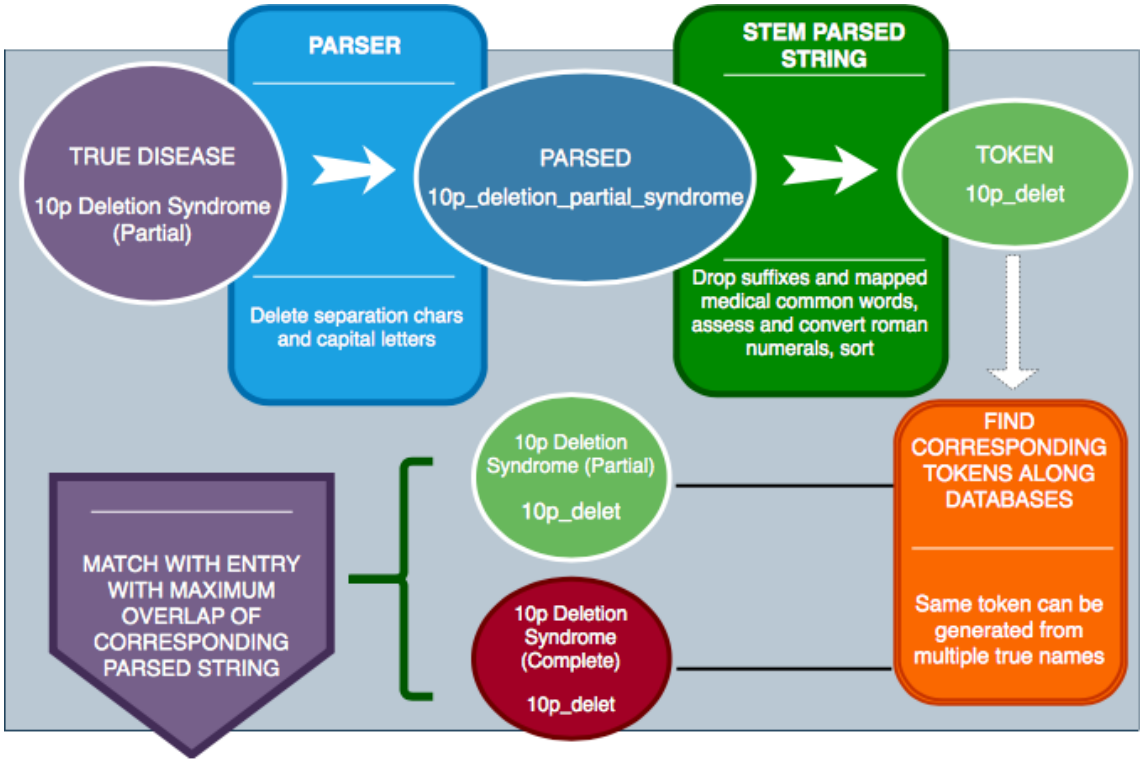


Figure 3.4: Scheme of the NLP pipeline developed in the CHIMeRA project. The disease words are processed in multiple step as showed in the example.

proves the efficiency of our pipeline, confirming that the union of multiple data sources can effectively enlarge our knowledge about biomedical agents.

The output of our merging procedure allows the realization of the CHIMeRA network-of-networks, i.e a network with more than 3.6×10^5 nodes and more than 3.8×10^7 links (ref. Fig.3.5). In our resulting structure we have 7 node types: disease (63 974), drug (35 161), gene (18 799), SNP (117 337), metabolite (114 100), phenotype (13 214), metabolite-pathway (1 329) and food (532). The full network adjacency matrix is still a block matrix, i.e we do not have all the combinations of information in our database. An emblematic case is given by food nodes: we have food information only into DrugBank and thus they would be connected only with drug types. On the other hand, our network architecture could be easily improved adding new data sources: food nodes are pendant nodes that could be easily connected to other kinds of data, introducing new node types or just filling the available blocks. CHIMeRA is still a work in progress project so we are still looking for improvements and new databases to add.

3.6 CHIMeRA analyses

The large amount of information provided by CHIMeRA network has to be analyzed to prove its efficiency. A preliminary analysis was performed evaluating the nodes degree centrality (the number of links associated to each node). The degree centrality is the simpler measure to quantify the importance of a node into a network: since our network-of-networks structure includes multiple node types we monitored it for each of them[13].

[13] We have chosen the degree centrality rather than other standard measures due to its numerical-simplicity/informative ratio. CHIMeRA network-of-networks includes a large amount of nodes, so the algorithmic complexity drastically affects the time performance. Degree centrality is given by summing the in- out-connections and thus it is the faster solution also with large matrix as in our case.

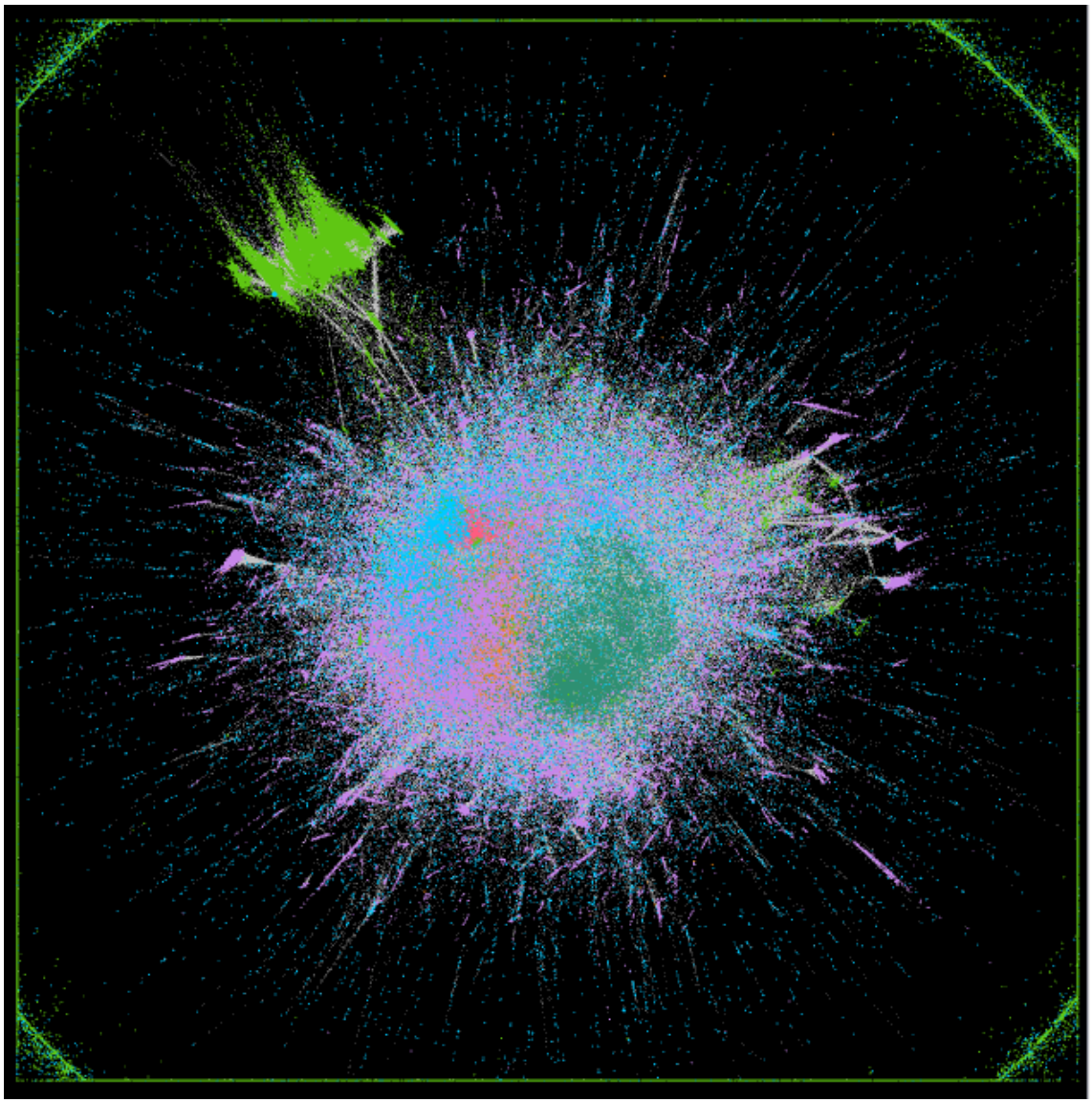


Figure 3.5: Graphical rendering of the first version of CHIMeRA network. The visualization was performed before the inclusion of DrugBank dataset. For computational issues we have not performed newer image of the global structure. In the image we represented disease nodes (azure), gene nodes (orange), SNP nodes (purple), metabolite nodes (light green), drug nodes (pink) and phenotype nodes (dark green). The visualization was obtained by the *Atlas layout* provided by Gephi.

	mean	std	min	25%	50%	75%	max
global degree	121.44	784.86	1	1	2	7	108147
disease	18.54	109.25	0	0	1	3	22911
drug	93.59	737.60	0	0	0	0	17750
food	0.03	0.32	0	0	0	0	12
gene	1.96	38.81	0	0	0	0	5605
metabolite	0.37	154.73	0	0	0	0	85236
phenotype	5.50	100.09	0	0	0	0	9732
SNP	0.65	21.96	0	0	0	0	4866
metabolic pathway	0.09	2.81	0	0	0	0	594
disease pathway	0.05	1.27	0	0	0	0	283
drug-action pathway	0.14	6.79	0	0	0	0	1006
drug-metabolism pathway	6×10^{-3}	0.28	0	0	0	0	60
signaling pathway	4×10^{-3}	0.29	0	0	0	0	49
physiological pathway	1×10^{-3}	0.07	0	0	0	0	13
macro pathway	0.48	106.06	0	0	0	1	59993

Table 3.3: Statistics of degree connections related to each node type stored in the CHIMeRA database. For each node type the average, standard deviation, minimum value, maximum value and percentiles of the degree distribution are reported. The first row shows the aggregated value of degree scores.

In this way we can perform a preliminary overview of the full set of information in the network. The results obtained by this test on degree centrality are shown in Tab. 3.3.

For each node type we computed the average number of connections and the most important parameters of each distribution (minimum, maximum, average, standard deviations and percentiles). This preliminary analysis confirms what we have already discussed during the creation of the CHIMeRA structure and thus that the minimum number of connections is just 1 (ref. min row in Tab 3.3): disease nodes are the core of our network-of-networks model but they do not have a direct link with food nodes; at the same time, drug nodes, which are connected to the several other types do not include all of them. Using the finer grain distinction between the metabolite-pathways (obtained by the information provided by HMDB) we noticed that the major part of their connections concern the *macro pathway* category as expected: *macro pathway*s identify the more general category in our nomenclature, including biological processes like apoptosis, DNA replication fork and phosphatidylethanolamine biosynthesis. A better visualization of the network structure could be done counting the average number of connections between each node group, i.e the block matrix visualization of the underlying bipartite-graphs. In Fig. 3.6 is shown this block matrix representation.

In Fig. 3.6 we can better appreciate the connections between the available information, visualizing and quantifying them. As expected, the only node type which is connected to all the others is the *disease* one: the only exception is given by *food* nodes for the previously discussed reasons. Our network-of-networks structure is very sparse and we do not have direct information of the major part of combinations (null blocks). The only two node types which show a reasonably good interaction with other blocks are the *disease* and *drug* ones[14].

[14] For sake of clarity we have to highlight also the two diagonal blocks in our matrix, given by these two node types. They arise from the synonyms and related causes in the first case (information provided by synonym dictionaries and RXList), while in the second case they highlight the synergies or not (information given by DrugBank). A network adjacency matrix tends to nullify diagonal information and node self-loops to prevent numerical issues and to increase the amount of mathematical theorems for its analysis.

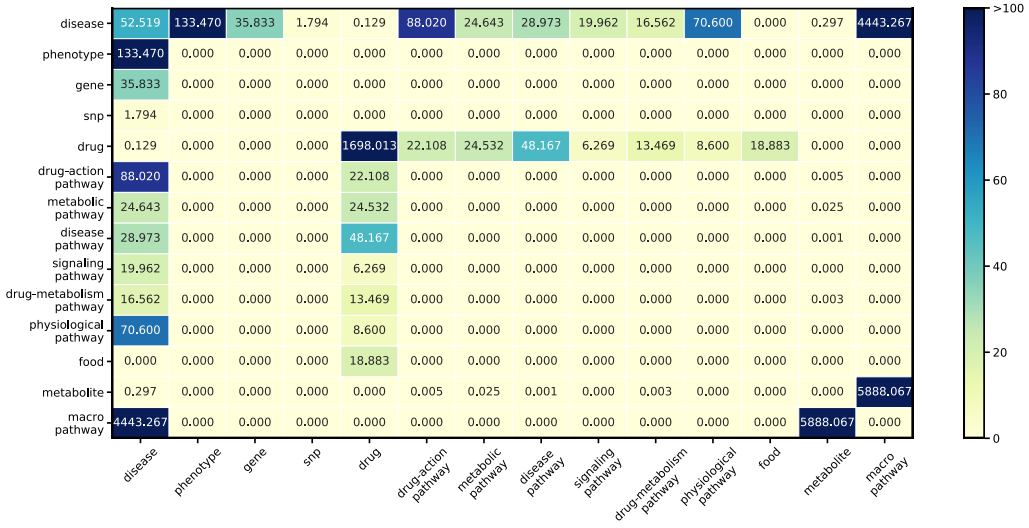


Figure 3.6: Block matrix representation of the CHIMeRA network. We computed the average number of connections between each node group and the bipartite-graphs structure is highlighted.

The sparsity of CHIMeRA network-of-networks highlights all the pros and cons of our work. More a matrix is sparse and more its management could be efficient from a numerical point-of-view: we are able to use a wide range of algorithms developed and tuned by sparse algebra; also the memory occupation could be optimized and it is a crucial task when we work with such a big quantity of data. At the same time, it highlights the potentiality of our work: each block connection derives from a single database evaluation (in first approximation), and we can reasonably assume that each block represents a possible output of a query performed on that database. In our global database we have the union of all these information and using at least the 2nd neighbors of a node (the nearest neighbors of each nearest neighbor of a node) we could obtain a mapping of all available information about it. Moving along the matrix blocks, in fact, we can start from a gene and see only the associated diseases, which is comparable to a single-database query; since each disease is connected to all the other node types, the 2nd neighbors of our gene give a panoramic overview of all the biomedical agents associated to it. This process is equivalent to an inference of missing blocks: if a gene is connected only to disease types, since all the other blocks are null, we can infer missing blocks using the links provided by disease nodes. This is the real power of a such network-of-networks structure. We would stress that the inference procedure could lead to incorrect biological associations, since it represents only an hypothesis unbacked up by data. However, using more database sources we can easily integrate missing information using the developed processing pipeline and, thus, increase the reliability of our hypothesis.

To further investigate the information provided by our network using the computed degree scores, we evaluated degree distributions of each node type. In Fig. 3.7 we show the degree distributions obtained considering the different node types individually.

All the distributions showed in Fig. 3.7 have long tails, but we cut them for visualization purposes. As expected and already highlighted by the previous analyses, the major part of node types have a not negligible amount of nodes with very low connectivity. Many node types have also isolated elements, i.e nodes with 0 degree. This behavior could be due to

We would stress that the showed matrix is not the adjacency matrix of CHIMeRA network but it is an aggregate representation of it. Thus, in our network each node has connections only with other nodes and no self-loops are present.

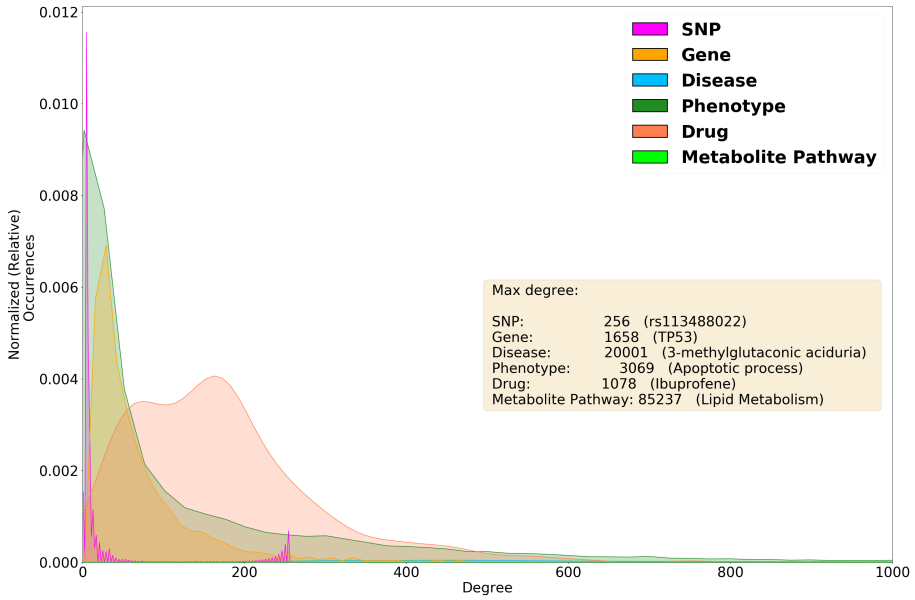


Figure 3.7: CHIMeRA degree distributions for different node types. The plot is cut for visualization purposes. The maximum degree node information are highlighted in the box.

two possible causes: 1) our pipeline tends to remove some information and, thus, some true positive associations; 2) there are some missing information in the original databases that could not be overcome by our merging. Since the most part of isolated nodes are given by *metabolites* (ref. Fig. 3.5 in which the green dots around the plot are isolated metabolite nodes) we checked into HMDB the origin of these issues. In a not negligible number of cases, HMDB does not provide a disease association to a given metabolite and it proves our second hypothesis, preserving the efficiency of our pipeline.

A more interesting result is obtained considering the most central nodes, i.e the node related to the maximum degree score. This information could be used also as validity check of the structure. As in the SymptomsNet case (ref. 3.3), we expect a reasonable interpretation of the most central nodes. These results are shown in the yellow box of Fig. 3.7.

The most central node for the SNP node type is rs113488022, well known polymorphism, validated by more than 70 public researches. This SNP is related to a wide range of cancer diseases and its clinical significance has been proved in different studies. It is always hard to discuss about the more or less importance of a SNP compared to the others, but its relation to several cancer types confirms its centrality score in our network structure. A more easily interpretable result is given by the most central gene, the TP53. TP53 is a crucial gene in many tumor diseases and its importance is well mirrored in our network structure. The major part of diseases inserted into public databases are related to tumor researches, and thus it was quite obvious to obtain n high centrality of this gene in our network. For the same reasons also the most central node for the *phenotype* type should be a tumor related characteristic: as expected, the most central node in this case is given by the apoptotic process, i.e the process which regulates the programmed cell death, which is largely involved in tumor diseases.

As previously discussed about *disease* and *drug* types, we have to consider a large set of available connections for them. Thus, we expected that a central node in these cases would be given by a quite generic entry. About *disease* type, in fact, we found as most central node the 3-methylglutaconic aciduria, a congenital metabolism anomaly related to leukemia. Despite this disease could be considered quite rare, its importance in our network structure is certainly given by the large amount of metabolite connections (we remember

that HMDB provides a very large amount of *metabolite* nodes that overcome the number of all the other node types, except by SNPs). The large quantity of metabolites in our structure weights on the number of disease connections and it proves the high centrality of a metabolic disease despite a genetic one. Moreover, we have also to take into account that this disease is related to leukemia, so we would have also a large set of genes and SNPs associated to it.

Considering the *drug* type, we obtained a very common drug as expected, given by the *Ibuprofene*. *Ibuprofene* is a very common anti-inflammatory drug that is used for treating pain, fever and inflammation which are all very general symptoms associated to a wide range of possible diseases. Thus, it is reasonable to assume that its number of connections is greater than other (more target specific) drugs.

A more in-depth explanation must be provided for the *metabolite pathway* type, where there are no seemingly reasonable explanations that prefer the found lipid metabolism to other macro-metabolite pathways. To explain its centrality we, once more time, come back to the original databases and to HMDB in this case. From a thorough inspection of HMDB we noticed that the most of them were studied using NMR chemical shift procedure. NMR chemical shift is a very common spectroscopy procedure to analyze biological compounds, but its signal is hardly related to particular nucleus types (e.g ^{1}H, ^{13}C, ^{15}N, $\cdots$). We could broadly describe this technique saying that as much hydrogen-like or carbonic-like structures are into the biological sample and much the signal should be easy to analyze. The metabolite relation to lipid metabolism is thus easy to study than other metabolism kinds, due to the large quantity of resonant nuclei involved[15]. This proves the high centrality of lipid metabolism regard other metabolite pathways.

These results are only preliminary analyses of CHIMeRA network-of-networks structure, but they are already able to clarify some potential use of our work. The only things that remain to be discussed are the usability and release of this new database to the research community.

3.7 CHIMeRA as a Service

We have discussed about the information stored into CHIMeRA database, but we have ignored how we could manage these data. More than the realization of a useful database, we have to provide an easy-to-use interface to encourage the research community to manage our processed information. We have already discussed about how modern databases are shared along the Internet and how these large quantities of data could be handled using database languages (ref. 3.1). Now we have to find the best solution for our application.

We developed a first version of CHIMeRA database using SQL (*Structured Query Language*) language[16] and in particular the SQLite one. SQLite is probably the easier solution for database management and the creation of efficient queries is straightforward. It is a well performing solution for standard relational databases, but it does not provide any facility for network structures. Moreover, SQLite is not directly comparable to client/server SQL database engines such as MySQL, Oracle, PostgreSQL or SQL Server since it is designed only for local data storage and individual applications. It is extremely efficient and simple in its applications, but it does not cover the requirements posed by our CHIMeRA structure and our needs about sharing information.

A more efficient solution is provided by modern graph databases (GDB). GDBs are

[15] Fatty acids in lipid mixtures are widely studied using NMR chemical shift since their molecular structure involves multiple resonant nuclei such ^{13}C, ^{31}P and ^{1}H.

[16] SQL is a domain-specific language designed for managing data held in a relational database management system (RDBMS) and it is particularly efficient in handling structured data.

Figure 3.8: NoSQL Performance Benchmark 2018 (source here). Absolute & normalize results for ArangoDBD, MongoDB, Neo4j and OrientDB. Comparison of time-performances using different (common) NoSQL queries. The first row shows the computing time on different NoSQL queries using ArangoDB. We mark with green boxes the solutions which perform better than ArangoDB and with red boxes the worse.

databases which use graph structures to represent and store information: there are two needed information for the database given by nodes and edges. The key concept behind this kind of storage is the relationship between entries. They go under the NoSQL (*not SQL*, or better "*Not only SQL*") database category, which store data according to more sophisticated models than simple tabular relations (typical model of SQL databases). GDBs allow simple and efficient retrieval of complex hierarchical structures by definition, representing the most efficient solution for our CHIMeRA database which is born as a network-of-networks architecture. Several solutions have been proposed to address graph storages and there are a wide range of possible GDB languages publicly available on-line (e.g Neo4j, OrientDB, Sparksee, AllegroGraph, $\cdots$). Based on our experience about these topics and driven by the available documentation (ref Fig. 3.8), we have chosen to use ArangoDB in our application.

ArangoDB is an open-source and free software released on Github for multi-model database management with a unified query language AQL (*ArangoDB Query Language*). ArangoDB database system is NoSQL, but its queries are very closed to SQL ones and, thus, they are easy to write also by no-expert users. The code core is written in C++ and, thus, it is extremely efficient from a numerical point-of-view (ref. Fig. 3.8). Moreover, it provides also a user-friendly web interface for network visualization and query development. The possibility to have a web interface allows an easy way to share our database on Internet as a service, increasing the usability of our tool. Moreover, query outputs can be also downloaded and used by external tools. Thus, using ArangoDB as service manager we provide a *Software as a Service* (SaaS) interface of our CHIMeRA database (*CaaS*). This project is still in work in progress and this CaaS is not yet publicly available[17].

We reformatted CHIMeRA network following the ArangoDB requirements and we created the graph database structure of our data. Using this database we have been able to perform the first queries and discuss about the results. The University of Bologna is currently involved into the HARMONY European project for the analysis of hematological data provided by multiple pharmaceutical companies. The HARMONY project aims to describe, analyze and model multiple data collected by various partners, producing a per-

[17] As soon as possible we intend to create it jointly to an adequate computational environment ables to support multiple external queries.

sonalized medicine framework for the study of hematological diseases. This project is based on the harmonization of different databases in the same way as our CHIMeRA project aims to merge multiple public data sources. The main focus of the HARMONY project is on diseases related to different kinds of *leukemia*. *Leukemia* is the most common type of cancer in children and it causes hundred of thousands of death every year. It is an hematological disease and its exact causes are still unknown. The developed CHIMeRA project could be used to contribute to this kind of researches, giving a wider biological overview about these diseases. Thus, we decided to formulate our first query on the *leukemia* disease.

We customized our query to extract only the 2nd neighbors related to this node. The pseudo-code used for our query is shown in 3.6.

Listing 3.6: CHIMeRA 2nd neighbors query

```
FOR x IN node_type_vertex
  FILTER x.name LIKE "looking_for_entry"
    FOR v, e, p IN 1..3 ANY x GRAPH "CHIMeRA"
      RETURN p
```

The query takes the node-collection (ArangoDB nomenclature) related to the searched node type (node_type_vertex in the code) and it filters all the names which satisfy the LIKE condition. Starting from the found nodes, it returns the output graph preview made by the 1st and 2nd neighbors (range of values 1..3 in the code).

We applied this query-like looking for *leukemia* node and we processed the results using Gephi as network viewer. The obtained network is shown in Fig.3.9: the network involves 9 460 nodes and 26 646 links. As can be seen by the plot, just considering the 2nd neighbors the obtained subnetwork is quite large and it highlights the biological complexity of this disease.

Using the "generic" name of *leukemia* we found 291 different types of leukemia diseases into CHIMeRA, which denote different facets of this disease. Despite these multiplicities of results, we noticed that they clustered in only 82 connected components, highlighting multiple similitudes between them. In particular, we found a giant components of 9 108 nodes and only other 6 components with more than 10 nodes. The giant component includes 165 different facets of *leukemia* disease, while the other connected components describe the remaining ones. The powerful of CHIMeRA network born exactly from the analysis of these cases, where we can infer missing information starting from the knowledge about analogous researches given by the full set of information related to the giant component found. In the giant component we can appreciate a description of the *leukemia* disease given by all the other node types: we have 587 diseases, 4 drugs, 2409 genes, 40 metabolites, 154 metabolite pathways, 5195 possible phenotypes and 719 SNPs related to them. The diseases associated to *leukemia* can help to highlight possible analogies (comorbidities) between this "difficult" disease and "easier" ones (cause and related disease connections) or simply provide a bridge to other node types (e.g drugs or genes) which are not directly related to the *leukemia* using databases individually. We would stress that, despite the *phenotype* node-type which includes the more general biological information, all the other amount of node-types represent only a small percentage of the available information (disease 0.9%, drug 0.01%, gene 12.8%, metabolite 0.03%, pathway 11.5%, SNP 0.6%, phenotype 39.3%). It is important to monitor also this kind of percentages because they could bring to possible biases in our description. A such biomedical overview could not be found using single-database approach and, to the best of the author's knowledge, only the CHIMeRA database is capable to map them.

The subnetwork extracted has more than half nodes as pendants (5 270/9 108 or 57%), i.e with degree score equal to 1. We have already discussed about this feature of CHIMeRA and, also in this case, we can use this behavior to connect other (possible) kinds of in-

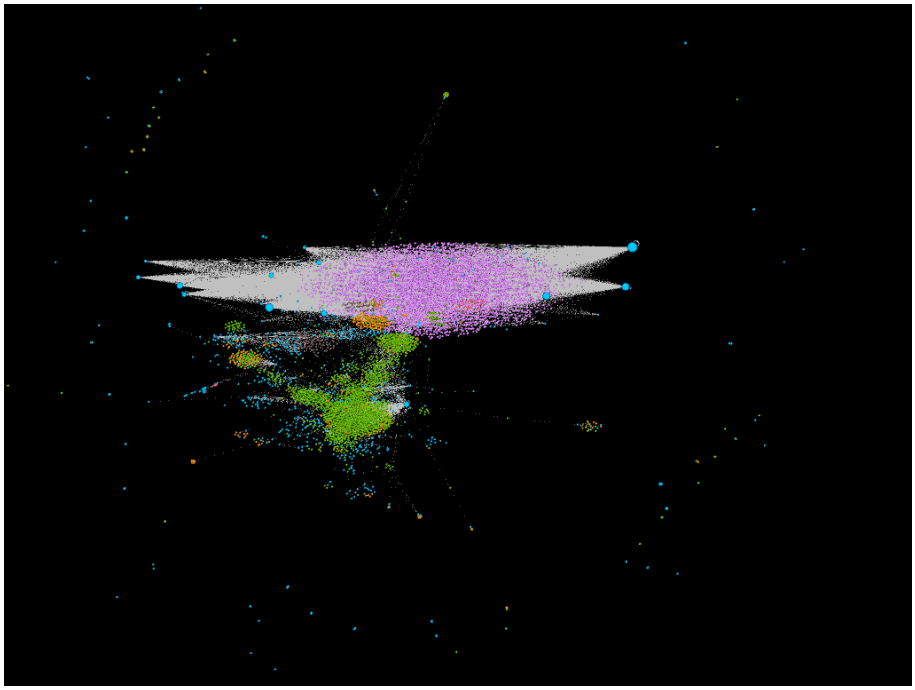


Figure 3.9: Output of *leukemia* query obtained by CHIMeRA graph database using 3.6. The subnetwork is made by the 2nd neighbors starting from all the nodes which include "leukemia" in their names. The subnetwork includes 291 different types of leukemias clustered into 82 connected components. The giant components is made by 9 108 nodes. CHIMeRA query is able to give a biomedical overview of the *leukemia* diseases mapping 838 diseases, 2 463 genes, 5195 SNPs, 154 metabolite pathways, 40 metabolites and 5 drugs associated to them.

formation to improve our disease description. We are still working on the analysis of the extracted information and, especially, about their biomedical interpretation related to the *leukemia* disease. Moreover, we have to see how we can combine our data to the HARMONY project samples. Thus, we end this chapter remarking the potential applications of a such network-of-networks structure and its capability of give us a more global overview of biomedical compounds in scientific researches.

Conclusions

We have concluded our discussion about the applications of Big Data Analytic algorithms to Biomedical data. In this work we have touched several and different topics related to this theme.

In the first chapter we have focused on the difficulties about information extraction, analyzing high-throughput datasets. The so-called *omics* datasets are becoming a very interesting research field in biology and medicine since, using modern data acquisition techniques, they are capable to give a wide range of useful data for the analysis of multiple diseases. A crucial role on this topic is given by tumors and, using *omic* data, we can design novel methods to identify the agents responsible for these diseases. Biomedical Big Data pose new challenges to the scientific research, since we have to convert them into useful information or, in other words, we have to be able to identify their informative cores. To this purpose we have designed the new DNetPRO algorithm as a novel feature selection method.

We have tried to show all the pros and cons about the proposed algorithm. Only knowing its limits we could be able to provide a reasonable interpretation about its results and for this reason we firstly tested our method on synthetic data. The proposed DNetPRO pipeline was tailored on gene expression applications and we have shown its application on real data, comparing its efficiency against other state-of-art models in which it is able to outperform them in the major part of the analyzed cases. Some these results are already published in international papers or they are in press. We would remark that DNetPRO could be used also as standalone feature selection algorithm, but, for sake of brevity, we have discussed its application on non-biological data only in the Appendices of this work.

In the second chapter we have moved to numerical applications in the deep learning research field. We have paid particular attention to the description and optimization of some state-of-art deep learning models. In this chapter we have also introduced three new custom libraries about this topic, which have been developed with different purposes: the NumPyNet is essentially an educational framework for the development of neural network models, while the Byron library is focused on the numerical performances; the rFBP library was designed for very particular applications and in this work we have just briefly shown one of them. Starting from the bases of neural network models, we have discussed about different kinds of functions (more or less straightforward) which are commonly involved in the construction of a deep learning neural network architecture. For each function, we have only summarily described its mathematical background, focusing instead on the critical points related to its algorithmic implementation.

We have used the two developed libraries (NumPyNet and Byron) to highlight possible ways to overcome these numerical issues. For sake of brevity, it has not been possible to go in deeper details about the numerical improvements performed, but all the developed codes have been shared and they are publicly available on the author's Github page. The modern scientific research, in fact, is not made only by papers and publications but it is acquiring even more importance the code development and, thus, its public availability. By sharing our code on the Internet, we want to encourage the research community to take in consideration also our promising results about these topics. We have touched

different state-of-art models and implementations of them along our discussion and in all the analyzed cases our results overcome them with not negligible results.

The results obtained using Super Resolution and Segmentation models are very promising for the analysis of biomedical images. Moreover, we have shown that deep learning models are capable of a very efficient generalization due to their vast amount of parameters and a well-programmed training section. In particular, our Super Resolution models were trained on general-purpose (natural) images, but they are, however, able to reconstruct biomedical NMR images better than standard methods. This, already discussed, result is due to the ability of the model in the identification of analogous textures and patterns between the training and validation images, without need of a tailored retrain. We have also seen how we can improve object detection efficiency using a pre - Super Resolution - processing: we could not show biomedical results on this topic due to lack of data and privacy reasons, but we have shown how a people-counting problem (Complex System task) is improved by this.

We would remark that our work was focused on the optimization of those codes only for CPU usages and, thus, we can not compare them to the wide world of GPU deep learning models. We would, however, stress that we have intentionally chosen to focus only on these computational environments, aiming to increase the usage of this kind of models also in research fields which do not need GPUs in their everyday works. There are, in fact, a lot of scientific applications which are tailored on CPUs architectures and which are pushed out to the deep learning researches, or which do not even try to use deep learning models afraid by their intensive computational demand. We developed the Byron library to overcome these issues and to highlight how a well-thought-out algorithmic implementation can overcome also the more computational expensive applications.

All the developed algorithms were intensively profiled against other state-of-art implementations and their pros and cons have been examined in order to find the best solution for a given problem. Code testing has been performed also on different operative systems, since how well an algorithm is made, it is useless if it works only on a well-defined machine. A continuous integration of our codes has been at the basis of all the proposed libraries, as much as a reliable and user-friendly code documentation.

We have concluded our discussion introducing the CHIMeRA project which, even though the analysis of its results is still in work in progress, gives us multiple points of discussion about Biomedical Big Data. There is an increasing interest about database harmonization in the last years and its need is given by the growing amount of publicly available data. The research community is still trying to keep up with the new demand of data analysis and an increasingly important role is played by the development of new computational strategies and techniques. Many European projects financed by the Horizon 2020 Research and Innovation program are focused on this topic and a particular attention is paid on the health-care research.

The CHIMeRA project could not be compared to such big research programs, but it is driven by the same kind of ideas. Its final purpose, in fact, is to use the wide range of available information and results, obtained by independent research studies, and combine them into a unique framework of analysis. Observational databases differ in both purposes and designs: they have been collected for different purposes and the logical organizations as much the medical terminologies can vary from source to source. A *Common Data model* (CDM) is designed to overcome these issues and to offer a unique solution for the information storage in the same way as our CHIMeRA project merges together information provided by multiple on-line databases. Unfortunately, the research community is developing a wide range of CDMs and as long as a single solution is not taken as standard, the problem can not be solved. Also in this case, we have not developed CHIMeRA as putative alternative to this purpose, but it is simply a temporary solution which allows us to perform a panoramic

overview of biomedical agents.

We have highlighted multiple possible usages of the developed CHIMeRA network-of-networks structure and we hope it can be useful as an integrative tool also for the biggest projects like the HARMONY one. In this work we have focused on the key steps which lead us to the ideas behind the CHIMeRA project and, moreover, we have described difficulties and their relative proposed solutions about the creation of a such network-of-networks database. Despite the work has been intense up to now, the more interesting part from a scientific point-of-view is just began. The CHIMeRA database is the only "code" discussed in this work which is not yet publicly available, due to its embryonic stage, but hopefully we can provide its first release as soon as possible.

Acknowledgment

The authors acknowledge EU IMI2 - HARMONY Healthcare Alliance for Resourceful Medicines Offensive against Neoplasms in HematologY n. 116026, EU COMPARE COllaborative Management Platform for detection and Analyses of (Re-) emerging and foodborne outbreaks in Europe n. 643476 and EU VEO - Versatile Emerging infectious disease Observatory n. 874735 for their support on biomedical analyses. A special thank goes to INFN Gruppo V AIM - Artificial Intelligence in Medicine, Progetto FILO-BLU Bando Lazio POR-FESR 2014-2020 LIFE2020 and EU ETN-ITN ImforFuture - Innovative training in methods for future data n. 721815 for what concern the development of machine learning and deep learning analyses show in this thesis.

Appendix A - Discriminant Analysis

The classification problems aim to associate a set of *pattern* to one or more *classes*. With a *pattern* we identify a multidimensional array of data labeled by a pre-determined tag. In this case we talk about *supervised learning*, i.e the full set of data is already annotated and we have prior knowledge about the association between data and classes.

In machine learning a key rule is played by Bayesian methods, i.e methods which use a Bayesian statistical approach to the analysis of data distributions. It can be proved that, if the underlying distributions are known, i.e a sufficient number of its moments are known with a sufficient precision, the Bayesian approach is the best possible method to face the classification problem (*Bayesian error rate*[36]).

Mathematical background

The exact knowledge of prior probabilities and conditional probabilities are generally hard to evaluate, thus a parametric approach is often needed. A parametric approach aims to create reasonable hypotheses about the data distribution and its fundamental parameters (e.g mean, variance, $\cdots$). In the following discussion, we are going to focus only on normal distributions for mathematical convenience, but the results could be easily generalized.

Given the multi-dimensional form of Gauss distribution:

$$G(\mathbf{x}|\mu,\Sigma) = \frac{1}{(2\pi)^{d/2}\cdot|\Sigma|^{1/2}}\cdot exp\left[-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right] \tag{3.2}$$

where $\mathbf{x}$ is a d-dimensional column vector, μ is the mean vector of the distribution, Σ is the covariance matrix ($d\times d$) and $|\Sigma|$ and Σ^{-1} the determinant and its inverse, respectively. We can notice the quadratic dependence of G by $\mathbf{x}$,

$$\Delta^2 = (\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu) \tag{3.3}$$

where the exponent (Δ^2) is called *Mahalanobis distance* of vector $\mathbf{x}$ from its mean. This distance can be reduced to the Euclidean one when the covariance matrix is the identity matrix ($\mathbf{I}$).

The covariance matrix is always symmetric and positive semi-definite by definition (useful information for the next algorithmic strategies) so it is invertible. If the covariance matrix has only diagonal terms the multidimensional distribution can be expressed as the simple product of d mono-dimensional normal distributions. In this case the main axes are parallel to the Cartesian axes.

Starting from a multi-variate Gaussian distribution[18], the Bayesian rule for classification problems can be rewritten as:

[18] In Machine Learning it will correspond to the conditional probability density.

$$g_i(\mathbf{x}) = P(w_i|\mathbf{x}) = \frac{p(\mathbf{x}|w_i)P(w_i)}{p(\mathbf{x})} = \frac{1}{(2\pi)^{d/2} \cdot |\Sigma_i|^{1/2}} \cdot exp\left[-\frac{1}{2}(\mathbf{x}-\mu_\mathbf{i})^T {\Sigma_i}^{-1}(\mathbf{x}-\mu_\mathbf{i})\right] \frac{P(w_i)}{p(\mathbf{x})} \tag{3.4}$$

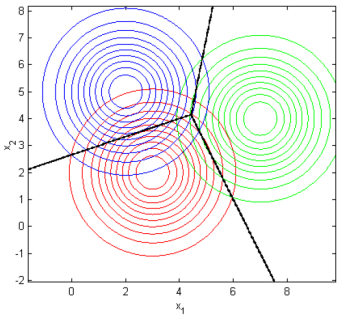
where, removing constant terms (π factors and the absolute probability density $p(\mathbf{x}) = \sum_{i=1}^{s} p(\mathbf{x}|w_i) \cdot P(w_i)$) and using the monotonicity of the function, we can extract the logarithmic relation:

$$g_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\mu_i)^T {\Sigma_i}^{-1}(\mathbf{x}-\mu_i) - \frac{1}{2}\log|\Sigma_i| + \log P(w_i) \tag{3.5}$$

which is called *Quadratic Discriminant function*.

The dependency by the covariance matrix allows 5 different cases:

- $\Sigma_i = \sigma^2 I$ **- DiagLinear Classifier**



This is the case in which features are completely independent, i.e they have equal variances for each class. This hypothesis allows us to simplify the discriminant function as:

$$g_i(\mathbf{x}) = -\frac{1}{2\sigma^2}(\mathbf{x^T x} - 2{\mu_i}^T\mathbf{x} + {\mu_i}^T\mu_i) + \log P(w_i) \tag{3.6}$$

and removing all the $\mathbf{x^T x}$ constant terms for each class

$$g_i(\mathbf{x}) = -\frac{1}{2\sigma^2}(-2{\mu_i}^T\mathbf{x} + {\mu_i}^T\mu_i) + \log P(w_i) = \mathbf{w_i}^\mathbf{T}\mathbf{x} + \mathbf{w_0} \tag{3.7}$$

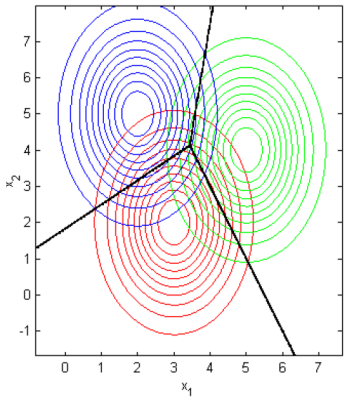
These simplifications create a linear discriminant function and the separation surfaces between classes are hyper-planes ($g_i(\mathbf{x}) = g_j(\mathbf{x})$).

With equal prior probability the function can be rewritten as

$$g_i(\mathbf{x}) = -\frac{1}{2\sigma^2}(\mathbf{x}-\mu_i)^T(\mathbf{x}-\mu_i) \tag{3.8}$$

which is called *nearest mean classifier* and the equal-probability surfaces are hyper-spheres.

- $\Sigma_i = \Sigma$ **(diagonal matrix) - Linear Classifier**

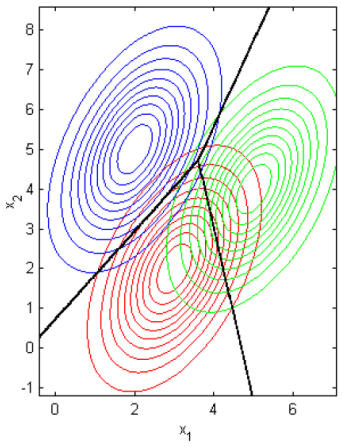


In this case the classes have same covariances but each feature has its own different variance. After the substitution of Σ in the equation, we obtain

$$g_i(\mathbf{x}) = -\frac{1}{2}\sum_{k=1}^{s}\frac{(\mathbf{x_k} - \mu_{i,k})^2}{{\sigma_k}^2} - \frac{1}{2}\log\prod_{k=1}^{s}{\sigma_k}^2 + \log P(w_i) \tag{3.9}$$

where we can remove constant ${\mathbf{x_k}}^2$ terms (equal for each class) and obtain another time a linear discriminant function and discriminant surfaces given by hyper-planes and equal-probability boundaries given by hyper-ellipsoids. We remark that the only difference from the previous case is the normalization factor of each axis that in this case is given by its variance.

- $\Sigma_i = \Sigma$ **(non-diagonal matrix) - Mahalanobis Classifier**



In this case we assume that each class has the same covariance matrix, but they are non-diagonal ones. The discriminant function becomes

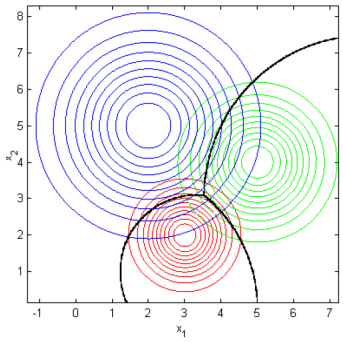
$$g_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x} - \mu_i)^T\Sigma^{-1}(\mathbf{x} - \mu_i) - \frac{1}{2}\log|\Sigma| + \log P(w_i) \tag{3.10}$$

where we can remove the $\log|\Sigma|$ term because it is constant for all the classes and we can assume equal prior probabilities. In this case we obtain

$$g_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x} - \mu_i)^T\Sigma^{-1}(\mathbf{x} - \mu_i) \tag{3.11}$$

where the quadratic term is the above told *Mahalanobis distance*, i.e a normalization of the distance according to the inverse of the covariance matrix. We can prove that expanding the scalar product and removing the constant $\mathbf{x^T\Sigma^{-1}x}$ term, we still obtain a linear discriminant function with the same properties of the previous case. In this case the hyper-ellipsoids have axes aligned to the eigenvectors of the Σ matrix.

- $\Sigma_i = {\sigma_i}^2 I$ **- DiagQuadratic Classifier**

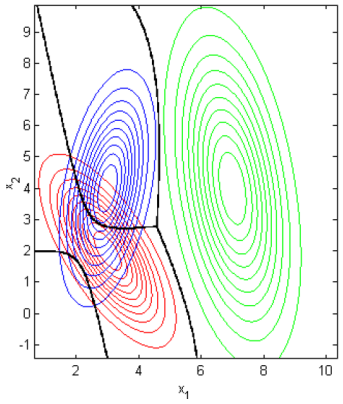


In this case we have a different covariance matrix for each class but they are all proportional to the identity matrix, i.e diagonal matrix. The discriminant function in this case becomes

$$g_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\mu_i)^T {\sigma_i}^{-2}(\mathbf{x}-\mu_i) - \frac{1}{2}s\log\left|{\sigma_i}^2\right| + \log P(w_i) \tag{3.12}$$

where this expression can be further reduced obtaining a quadratic discriminant function. In this case the equal-probability boundaries are hyper-spheres aligned to the feature axes.

- $\Sigma_i \neq \Sigma_j$ **(general case) - Quadratic Classifier**



Starting from the more general discriminant function we can relabel the variables and highlight its quadratic form as

$$g_i(\mathbf{x}) = \mathbf{x^T W_{2,i} x + w_{1,i}^T x + w_{0,i}} \quad \text{with} \quad \begin{cases} \mathbf{W_{2,i}} = -\frac{1}{2}{\Sigma_i}^{-1} \\ \mathbf{w_{1,i}} = {\Sigma_i}^{-1}\mu_i \\ \mathbf{w_{0,i}} = -\frac{1}{2}{\mu_i}^T{\Sigma_i}^{-1}\mu_i - \frac{1}{2}\log|\Sigma_i| + \log P(w_i) \end{cases} \tag{3.13}$$

In this case each class has its own covariance matrix Σ_i and the equal-probability boundaries are hyper-ellipsoids oriented to the eigenvectors of the covariance matrix of each class.

The Gaussian distribution hypothesis of data should be tested before using these classifiers. It can be evaluated using statistical tests as *Malkovich-Afifi* based on *Kolmogorov-Smirnov* index or using the empirical visualization of the data points.

Numerical Implementation

From a computational point-of-view we can exploit each mathematical information and assumption to simplify the computation and improve the numerical stability of our computation. We would remark that these considerations were taken into account in this

work only for the C++ algorithmic implementation, since these methods are already implemented in high-level programming languages as Python and Matlab[19].

In the previous section we highlighted the covariance matrix properties, i.e the covariance matrix is a positive semi-definite and symmetric matrix by definition and these properties allow the matrix inversion. The computation of the inverse-matrix is a well known complex computational step from a numerical point-of-view and in a general case can be classified as an $O(N^3)$ algorithm. Moreover, the usage of a Machine Learning classifier commonly matches the usage of a cross validation method, i.e multiple subdivision of the dataset into training and test sets. This involves the computation of multiple inverse matrices and it could represent the performance bottleneck in many real applications (the other computations are quite simple and their algorithmic complexity are certainly less than $O(N^3)$).

Using the mathematical information about covariance matrix we can find the best numerical solution for its inversion, that in this case is given by the Cholesky decomposition algorithm. The Cholesky decomposition or Cholesky factorization allows to rewrite a positive-definite matrix into the product of two triangular matrices (the first is the conjugate transposed of the second).

$$\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathbf{T}} = \mathbf{U}^{\mathbf{T}}\mathbf{U} \tag{3.14}$$

The algorithmic complexity is still the same, but the inverse estimation is simpler using a triangular matrix and the entire inversion can be performed in-place. It can also be proved that general matrix inversion algorithms suffer of numerical instability issues compared to the output of Cholesky decomposition. In this case the original matrix inversion can be computed by the multiplication of the two inverses as

$$\mathbf{A^{-1}} = (\mathbf{L^{-1}})^T(\mathbf{L^{-1}}) = (\mathbf{U^{-1}})(\mathbf{U^{-1}})^T \tag{3.15}$$

As second bonus, cross validation methods involve the data splitting in multiple non-independent chunks of the original data. The extreme case of this algorithm is given by the Leave-One-Out cross validation in which the data superposition between folds is $N-1$ (where N is the size of the data). The statistical influence of the swapped data is quite low and the covariance matrix would be quite similar across folds (the inverse matrix would be drastically affected from each slight modification of the original matrix instead). A second step of optimization can be performed computing the original full-covariance matrix of the whole set of data ($O(N^2)$) and modify it into the right k indexes at each cross-validation step ($O(N * k)$) that in the Leave-One-Out becomes a single editing case. This second optimization can also be performed in the Diag-Quadratic case substituting the covariance matrix with the simpler variance vector.

In 3.7 the implementation of Cholesky decomposition used to invert the covariance matrix is shown.

[19] For sake of completeness we have to highlight that the classification functions provided by Matlab, i.e classify, are already included into the base software packages, i.e no external Toolbox is needed, while for the Python case the most common package which implements these techniques is given by the scikit-learn library. Matlab allows to set the classifier type as input parameter of the function using a simple string which follows the same nomenclature previously proposed. Python has a different imports for each classifier type: in this case we found correspondence between our nomenclature and the Python one only in *quadratic* and *linear* cases, while the *Mahalanobis* classifier is not considered as putative classifier. The *diagquadratic* classifier is called GaussianNB (*Naive Bayes Classifier*) instead. The last important discrepancy between the two language implementations is found in variance evaluation (and corresponding covariance matrix): Matlab proposes the variance estimation only in relation to the mean so the normalization coefficient is given by the number of samples except by one ($N-1$), while Python computes the variance with a simple normalization by N.

Listing 3.7: Cholesky inverse matrix

```
#include <iostream>
#include <cmath>

void Cholesky (const int & n, float * mat, float * p)
{
  for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j)
    {
      const int idx = i * n + j;
      float sum = mat[i * n + j];

      for (int k = i - 1; k >= 0; --k)
        sum -= mat[i * n + k] * mat[j * n + k];

      if (i == j)
      {
        if ( sum <= 0.f )
        {
          std :: cerr << "Matrix is not positive definite" << std :: endl;
          std :: exit(1);
        }

        p[i] = 1.f / std :: sqrt(sum);
      }
      else
        mat[j * n + i] = sum * p[i];
    }
}

void CholeskyInv (const int & n, float * mat, float * mat_inv)
{
  float * p = new float[n];
  std :: copy_n(mat, n*n, mat_inv);

  Cholesky(n, mat_inv, p);
  for (int i = 0; i < n; ++i)
  {
    mat_inv[i * n + i] = p[i];

    for (int j = i + 1; j < n; ++j)
    {
      float sum = 0.f;

      for (int k = i; k < j; ++k)
        sum -= mat_inv[j * n + k] * mat_inv[k * n + i];

      mat_inv[j * n + i] = sum * p[j];
    }
  }
}
```

Both these two techniques have been used in the C++ implementation of the Quadratic Discriminant Analysis classifier and in the Diag-Quadratic Discriminant Analysis classifier used in the DNetPRO algorithm implementation (see 1.1).

Appendix B - Venice Road Network

Tourist flows in historical cities are continuously growing in a globalized world and adequate governance processes, politics and tools are necessary in order to reduce impacts on the urban livability and to guarantee the preservation of cultural heritage. The ICTs offer the possibility of collecting large amount of data that can point out and quantify some statistical and dynamic properties of human mobility emerging from the individual behavior and referring to a whole road network. In this work we analyze a new dataset that has been collected by the Italian mobile phone company TIM, which contains the GPS positions of a relevant sample of mobile devices when they actively connected to the cell phone network. Our aim is to propose innovative tools allowing to study properties of pedestrian mobility on the whole road network. Venice is a paradigmatic example for the impact of tourist flows on the resident life quality and on the preservation of cultural heritage. The GPS data provide anonymized geo-referenced information on the displacements of the devices. After a filtering procedure, we develop specific algorithms able to reconstruct the daily mobility paths on the whole Venice road network. The statistical analysis of the mobility paths suggests the existence of a travel time budget for the mobility and points out the role of the rest times in the empirical relation between the mobility time and the corresponding path length. We succeed to highlight two connected mobility subnetworks extracted from the whole road network, that are able to explain the majority of the observed mobility. Our approach shows the existence of characteristic mobility paths in Venice for the tourists and for the residents. Moreover the data analysis highlights the different mobility features of the considered case studies and it allows to detect the mobility paths associated to different points of interest. Finally we have disaggregated the Italian and foreigner categories to study their different mobility behaviors.

The datasets

The dataset used in this study has been provided by the Italian mobile phone company TIM and contains geo-referenced positions of tens of thousands anonymous devices (e.g. mobile phones, tablets, etc. ...), whenever they performed an activity (e.g. a phone call or an Internet access) during eight days from 23/2/2017 up to 02/03/2017 (Carnival of Venice dataset), and from 14/7/2017 up to 16/7/2017 (*Festa del Redentore* dataset). According to statistical data, 66% of the whole Italian population has a smart-phone and TIM is one the greatest mobile phone company in Italy whose users are $\sim$ 30% of the whole smart-phone population. The datasets refer to a geographical region that includes an area of the Venice province, so that it is possible to distinguish commuters from sedentary people and the different transportation means used to reach Venice. Each valid record gives information about the GPS localization of the device, the recording time, the signal quality and also the roaming status, which in turns allow to distinguish between Italian and foreigners. The devices are fully anonymized and not reversible identification numbers (ID) are automatically provided by the system for mobile phones and calls within the scope of the trial; the ID is kept for a period of 24 hours. During each activity a sequence of GPS

data is recorded with a 2 sec. sampling rate and the collection stops when the activity ends. As matter of fact during an activity most of people reduce their mobility except if they are on a transportation mean, so that the dataset contains a lot of small trajectories that have to be joined to reconstruct the daily mobility. After a filtering procedure these data provide information on the mobility of a sample containing 3000 – 4000 devices per day. Since the presences during the considered events were of the order of 105 individuals per day, as reported by the local newspapers, we estimate an overall penetration of our sample of 3 – 4%. The filtering procedure and the other statistical information about the sample penetration are discussed in the original paper [64].

Mobility paths reconstruction on the road network

The procedure of mobility path reconstruction considers separately the land mobility and the water mobility since the two mobility networks have different features, so that it is necessary to check carefully the transitions from one network to the other. To create a mobility path, we connect two successive points left by the same device using a best path algorithm on the road network with a check on the estimated travel speed to avoid unphysical situations and discarding the paths whose velocity is clearly not consistent with the typical pedestrian velocity (or ferryboat velocity). To end a land path and to start a water path, we require that at least two successive points of the same device are attributed to a ferryboat line by the localization algorithm. In the case of a single point on a ferryboat line, we force the localization of this point on the nearest road on the land.

The reconstruction of the mobility paths also allows to study how people perform their mobility on the road network. We consider the problem of determining the most used subnetwork of the Venice road network. The existence of mobility subnetworks could be the consequence of the peculiarity of Venice road network, where it is quite easy to get lost if you do not have a map. Therefore people with a limited knowledge of the road network move according to paths suggested by Internet sites or following the signs on the roads. To point out a mobility subnetwork we rank the roads of Venice according to a weight proportional to the number of mobility paths passing through each road. Thus We define a relevant subnetwork as a connected subnetwork that explains a considerable fraction of the observed mobility. In this case each road (identified by two nodes in the poly-line format) represents the link of our weighted graph and we can apply the DNetPRO technique shown in 1.3.3 to identify the network core with only closed paths[20].

Starting from the previously evaluated daily flows for each road, we order in a decreasing way the roads according to the observed flows. The DNetPRO algorithm scrolls down the list adding the road to a temporary list. At every step the "pruning process" starts on the selected roads cutting the isolated roads in order to get a connected subnetwork[21]. Therefore the number of nodes of the subnetwork increases in a discontinuous way, when the adding of a new road in the list allows to connect several previously selected roads. After several parametric scans, we found that the best result for our purposes is achieved by choosing about the 10% of the nodes in the whole Venice road network. In Fig 3.10 we show four consecutive selected subnetworks in the case of Carnival dataset to illustrate how the algorithm operates.

[20] Pendant nodes are unphysical solutions in our model since we are interested on the pedestrian mobility paths that bring people from one location to an other.

[21] Since we are interested on the largest connected component the *merging* parameter is off.

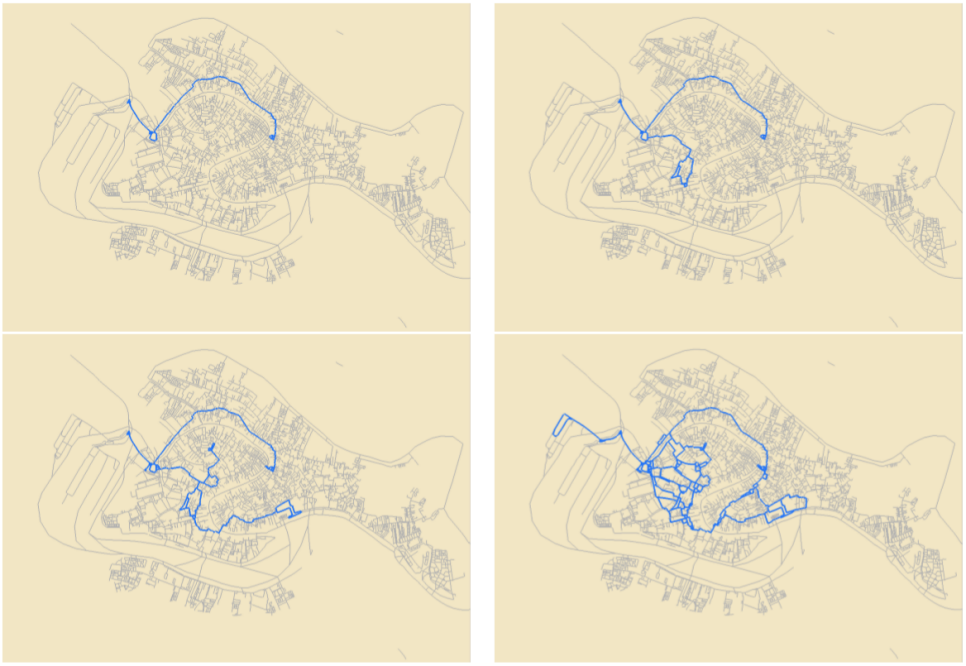


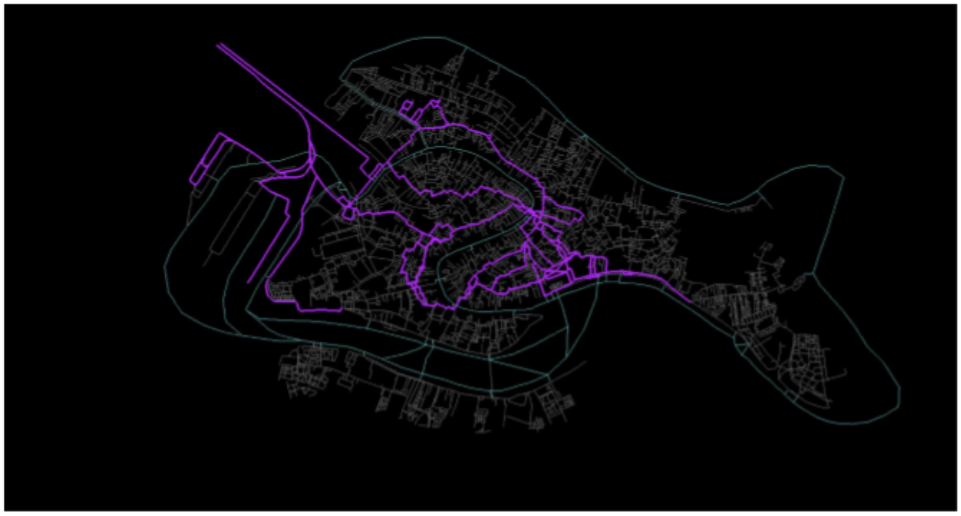
Figure 3.10: From top-left to right-bottom, we plot four mobility subnetworks with increasing number of roads, selected by the DNetPRO algorithm using the Carnival dataset.

Using the DNetPRO algorithm we are able to extract a subnetwork which explains the 64% of the observed mobility using 13% of the total road network length for the case of the Carnival dataset and 15% of the total length in the case of the *Festa del Redentore* dataset.

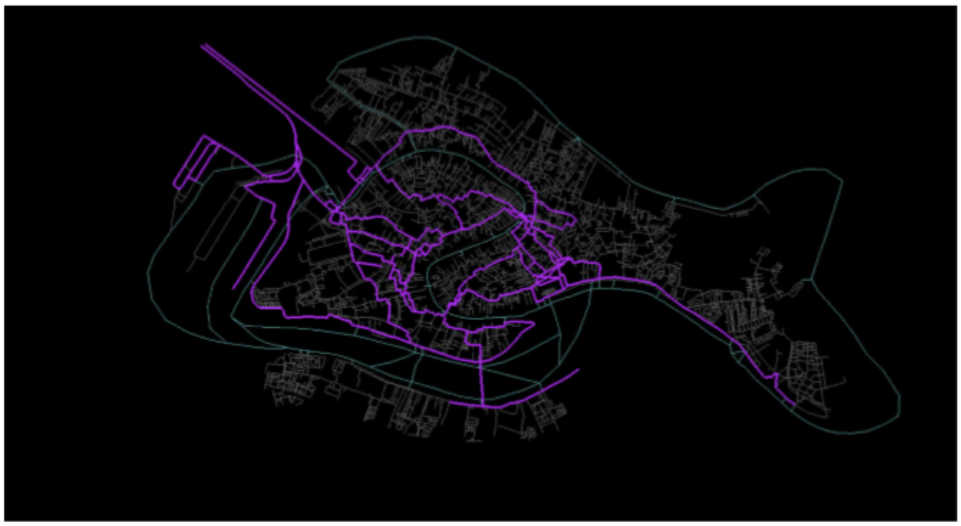
The selected road subnetworks are plotted in Fig 3.11 for both the datasets. As a matter of fact, many of the highlighted paths are also suggested by Internet sites. However, we remark some differences that can be related by the different nature of the considered events. During the Carnival of Venice the mobility seems to highlight three main directions connecting the railway station and the *Piazzale Roma* (top-left in the map), which are the main access points to the Venice historical centre, with the area around San Marco square, where many activities where planned during 26/02/2017. In the case of the *Festa del Redentore* the structure is more complex due to the appearance of several paths connecting the station and *Piazzale Roma* with the *Dorsoduro* district in front of the *Giudecca* island.

This geometrical structure could have a double explanation: on one hand the *Festa del Redentore* introduces an attractive area near the *Giudecca* island, where the fireworks take place in the evening; on the other hand the *Festa del Redentore* is a festivity very much felt by the local population, that knows the Venice road network and performs alternative paths.

On these subnetwork we also map the mobility of Italians and foreigns separately. The results of this application are deeply discussed in the paper.



(a)



(b)

Figure 3.11: Picture (a): selected subnetworks (highlighted in purple) from the road network of the Venice historical centre (in the background), that explain 64% of the recorded mobility in the datasets. The top picture refers to the Carnival mobility during 26/02/2017 and corresponds to 13% of the total length of the Venice road network. The picture (b) refers to the *Festa del Redentore* mobility during 15/07/2017 and corresponds to 15% of the total length of the Venice road network.

Appendix C - BlendNet

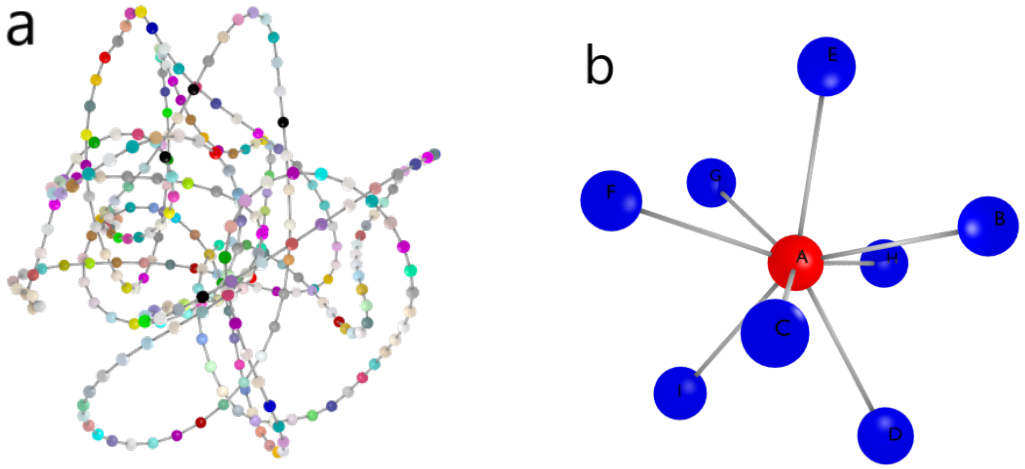


Figure 3.12: (**a**) Chain graph rendered by BlendNet software. Node colors are randomly generated by the tool. (**b**) Star graph rendered by BlendNet software. Node colors and labels are given as extra columns in node-list file.

Graph visualization is still an open problem in many applications. The problem is commonly related to large graph visualization in which problems arise from the rendering of a large number of nodes and a greater number of links between them (a graph with N nodes could have $(N \times N)$ possible links). An other open problem concern the multi-dimensional visualization of the graphs. Despite common graph tools compute the node coordinates in any space dimensions (and clearly the maximum number of possible dimensions for a visualization is only 3) the real visualization is often allowed only in 2D spaces. The counterpart of these problems concern a pretty visualization of the graphs, that it is often ignored by many tools which prefer focusing on simple renderings.

In this section we introduce a new custom graph viewer developed for pretty small-networks visualization in 2D and 3D, called BlendNet [16] (*Blender Network viewer*). BlendNet is an open-source project and it is released on Github under GPL license. All the small-graphs showed in this work are made using this tool and in particular the feature-signatures generated by the DNetPRO algorithm.

BlendNet is written in Python with the help of Blender APIs. Blender is now a standard for 3D rendering and it is commonly used in a wide range of graphical applications, starting from the simpler 3D dynamics to video-game applications. Blender is certainly more than a simple graphical viewer, but it provides an easy Python interface and a wide on-line documentation which make it a useful tool for graphical representation of 3D structures.

We are forced to use the Python version provided by Blender to use its APIs and any extra-package required by our application has to be installed with the appropriated pip. We use the networkx Python library for node coordinates computation and thus we have to update our Python-Blender with the appropriated packages. Moreover, since the code

can be difficult to manage for non-expert users, we have written an easy command-line interface to set the whole set of parameters required by the graph viewer that can be piloted via Makefile rules. The list of nodes and edges can be passed via command-line with the relative filenames, in the same format of the concurrent graph viewers (e.g *Gephi* software, the other graph viewer used in this work to generate the larger network structures of the CHIMeRA project).

The software project is a single script file and it includes a full list of possible examples and usages. Some of this examples are shown in Fig. 3.12. A full list of installation instructions is also provided for any operative system (Unix, MacOS and Windows). These instructions cover a full installation of Blender, Python and BlendNet package for administrator and no-root users (ref. Shut project [19]). With slight code editing we can obtain different node coordinates and shapes. Node colors, sizes and positions can also be given using the node-list file as independent columns.

Appendix D - Multi-Class Performances

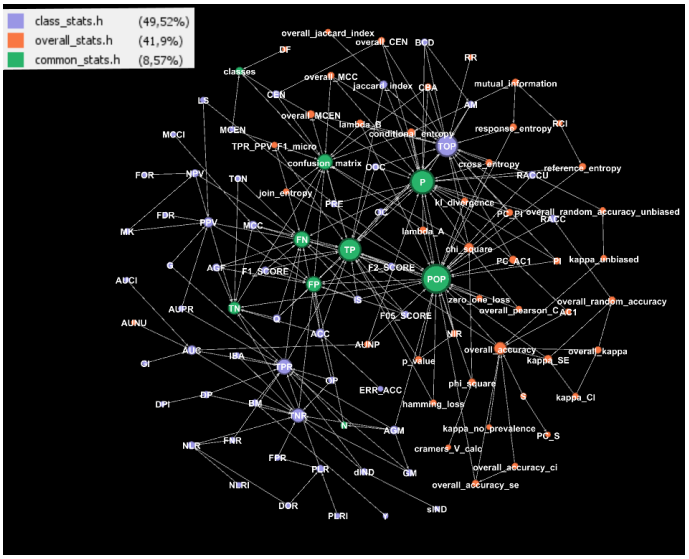


Figure 3.13: Multi classes score interaction graph. Each node identifies a different performance evaluator and the links are given by the interactions between mathematical formulations of each quantity. The graph has more than 100 nodes and more than 200 links. The node colors are given by the classes identified in the work of Sepand et al. [41].

The performances evaluation is a crucial task in any Machine Learning application. Given a set of patterns and its corresponding (true) labels, we can evaluate the efficiency of a given model with a comparison between labels and model outputs, i.e the predicted labels. There are a lot of different score functions that can be computed and each of them evaluates some aspects of the model efficiency. Any paper author choses the score that better highlights the advantages of its model and it is difficult to move around this large zoo of indicators. Moreover, (it is quite a constant in scientific research) when a paper is send to a peer-review, in many cases the reviewers suggest to check if other performance indicators are good enough for the showed results. This means that a lot of large simulations should be performed again and the appropriate variables recomputed to obtain the required scores.

At this point the main question is: are these scores totally independent one from each other? The brief answer is simply no. In a very interesting work of Sepand et al. [41] the authors show how we can compute a wide range of these scores starting from the evaluation of the simple confusion matrix[22], providing a full mathematical documentation and references about their numerical evaluations.

[22] The confusion matrix is a square matrix of shapes (N, N), with N the total number of classes in the

Despite the Python code provided by Sepand et al. explains these links between the mathematical quantities, they stop their analyses on the score evaluations, without any interest on the optimization of these computations. Starting from their work we analyzed the inter-connections between these mathematical formulas and we extracted the dependencies between the involved variables. In particular, a score function can be interpreted as a node and its connections could be given by the variables needed to evaluate it. This type of graphs are commonly called *factor graphs*. In a mathematical formulation of *factor graphs* there are different kinds of nodes (variables and factors, or equations). The focus of our analysis was not on the mathematical formalism of these kinds of graphs, but we aimed to a visualization of function interactions and an analysis of the numerical improvements derived from it.

In the work of Sepand et al. the authors identify three function classes: common statistics, class statistics and overall statistics. In Fig. 3.13 the interaction graph of these three classes is shown. The figure shows deep interactions between the three function classes and it highlights the dependencies of the different quantities involved. We can also use this kind of visualization to formulate computational considerations about the order in which these quantities could be evaluated. Since the graph is a direct graph by definition, we can start from the root node (the node without links which bring to it) and cross the network up to the leaf nodes (nodes without links which go out from them) like in a tree-graph (or more precisely a DAG, *Direct Acyclic Graph*). At each step of the percolation, the incoming nodes identify totally independent quantities. This independence means that the node-quantities can be potentially computed in parallel. To clarify this consideration we can reorganize the graph visualization minimizing the link lengths and obtaining a stratified graph in which each level identifies a potential parallel section. A graph with these properties was obtained using the dot visualization and it is shown in Fig. 3.14. As can be seen in the figure we can identify 7 levels in the graph and thus 7 potential parallel regions for the computation of the full set of functions.

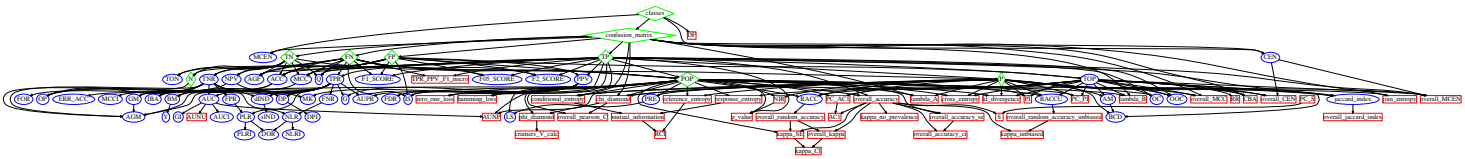


Figure 3.14: Re-organization of the graph in Fig. 3.13. The rendering was obtained using the dot visualization, i.e the minimization of the link lengths. The direct graph identifies the tree of dependencies and each level of the tree represents a set of independent functions that can be potentially computed in parallel. This graph is used as parallel scheme for the Scorer library.

These considerations allow us to create an optimized version of the code of Sepand et al., the Scorer library [23]. The Scorer library is the C++ porting of the PyCM library of Sepand et al. with a Cython wrap for the Python compatibility. Following the above told graph, the computation of score quantities are performed in parallel according to the 7 levels found. The parallelization strategy chosen uses the section keywords of OpenMP library to perform no-wait tasks that are computed by each thread of the parallel region.

current problem, whose entries are the number of right and false classifications. In particular, each entry of the matrix represents the predicted instances in a given class. If the class is the right one we call it as true positive item. As counterpart we have a false positive item.

The extracted graph includes more than 100 different quantities so writing the full set of parallel sections becomes an hard (and boring) work in C++. Moreover, update the graph with new quantities brings to a consequential update of the full code and also of the parallelization strategy. Each function was written as an anonymous-struct, i.e a functor, with an appropriate operator overloading. Each functor has a name given by a pre-determined regex (get_{function}) and the list of arguments follows the same nomenclature[23]. With these expedients we created a fully automated Python script which parses the list of functors, it computes the dependency graph and the parallelization levels and it gives back a compilable C++ script with the desired characteristics. In this way we can guarantee an easy way to update the library and moreover we overcome the boring writing of a long code. The automatic creation script is provided in the Scorer library and it should be used at each pull request or version update.

For a pretty/useful visualization of the computed quantities we rendered the interaction graph in an HTML framework. In this way we can insert with a CSS table the computed values in each node that can be discovered passing the mouse over the figure. An example of this rendering is given in the on-line version of the library [23].

In conclusion the developed Scorer library is a very powerful tool for Machine Learning performances evaluation which can be used either in C++ either in Python codes through its Cython wrap. The code is automatically generated at each update and automatically tested using continuous integration for any platform using Travis CI and Appveyor CI[24]. The code can be compiled using CMakefile or Makefile and a setup.py is provided for the Python version. So when you write a new paper on Machine Learning and you do not know what could be the most appropriate indicator to show in your research or you are afraid that a referee could ask you to compute an other one there is only one solution: compute them all using Scorer.

[23] If the functor receives in input the variable A and B we have to ensure that two functors named get_A and get_B will be provided. The only exception is given by the root functor.

[24] We perform tests for Unix and Windows environments. We check more than 15 combinations of environments and compilers.

Appendix E - Neural Network as a Service

One of the final goals of Machine Learning is certainly the process automation. We develop everyday complex models to perform tasks that should be automatically executed by a computer without human supervision. Neural Networks are classical mathematical tools used for these purposes and we have widely discussed about them in Chapter 2 of this work. Beyond Neural Network structures and purposes for which they are made, there is still an uncovered topic to discuss: the automation of these kinds of algorithms into a computer device. In this section we are going to discuss an implementation of these algorithms as a service in a computer server. In particular we will talk about the implementation of the *FiloBlu* service which is part of a project developed in collaboration with the Sapienza University (Rome) and the INFN Data Center CNAF of Bologna. This work is still in progress and its purpose goes beyond the current topic, so we will focus only on the implementation of the service, without any reference on the Machine Learning algorithm used. This is a further proof that the developed techniques are totally independent by the final application purpose.

A service is a software that is executed in background in a machine. In Unix machines it is often call daemon, while in Windows machine is called *Windows service*. A service starts only with administrator privileges and it goes on without any user presence. An other important requirement is the ability to restart, when some troubles occur in the machine functionality and/or at the boot of the machine.

A Machine Learning service could be used for applications in which we have to manage an asynchronous stream of data for long time intervals. An example could be the case in which the data provider is identified by an APP or a video-camera. These data should be stored inside a central database, that can be located in a different device or in the same computer in which the service runs. Since the service runs in background, the only communication channel with the user is given by log files. A log file is a simple readable file in which are saved the base information about the current status of the service. Thus, it is crucial to set appropriate check-points into the service script and chose the minimum quantity of information that the service should write to make user-understandable its status.

FiloBlu Service

In the *FiloBlu* project we have a stream of data provided by an external APP that are stored in a central database server. The Machine Learning service has to read the information stored into a database, it processes them and finally it writes the results into the same database. All these operations have to be performed with high frequency since the output results have to be shown in a real-time application. This frequency would be the clock-time of the process function, i.e at each time interval (as small as we like) the process task will be called and we will have the desired results in output. At the same time we have to take

care about the time required by our Machine Learning algorithm: not all the algorithms can process data in real time and the frequency of process function has to be less than the time required by the algorithm or we can loose some information.

We obtain the best efficiency from a service splitting as much as possible the required functionality in small-and-easy tasks. Small tasks can be evaluated as independent functions with an associated frequency that in this case can be reduced as much as possible. The *FiloBlu* functionalities can be reviewed as a sequence of 3 fundamental steps and other 2 optional ones: 1) read the data from the database, 2) process the data with the Machine Learning algorithm and 3) write the obtained results into the database; 4) update the Machine Learning model and 5) clear old log files are optional steps. To further improve the service efficiency we give each (independent) step to a different thread. The whole set of tasks are piloted by a master thread given by the service itself. In this way the service is computational efficient and moreover it does not weight on the computer performances. We have to take in mind that the computer which hosts the service has to be affected by the daemon process as less as possible either in memory either on computational efficiency. The last step is the synchronization of the previous tasks with appropriate clock frequencies.

Let's start from the data reading function. Since our data are assumed to be stored into a database, this function has to perform a simple query and extract the latest data inserted. Obviously the efficiency of the step is based on the efficiency of the chosen query. The data extracted are saved in a common container shared between the list of threads and thus it belongs to the master. The choice of an appropriate container is a second point to carefully takes in mind. This container should be light an thread-safe to avoid thread concurrency. While the second request is implementation dependent, the first one can be faced on using a FIFO container[25]. In this way we can ensure that the application will save a fixed amount of data and it will not occupy large portion of memory (RAM).

The second task is identified by the Machine Learning function which processes the data. The algorithm takes the data from the FIFO container of the previous step (if there are) and it saves the results into a second FIFO container for the next step. The time frequency of the step is given by the time required by the Machine Learning algorithm.

The third step takes the data from the second FIFO container (if there are) and it performs a second query (a writing one in this case) to the database. Also in this case the frequency is given by the efficiency of the chosen query.

The last two steps can be executed without time requirements and they are useful only on a large time scale.

Each step performs its independent logging on a single shared file. If an error occurs the service logs an appropriate message and it saves the current log-file in a different location to prevent possible log-cleaning (optional step). Then the service restarts.

We implemented this type of service in pure Python and the code is publicly available on Github [18]. The developed service was customized according to the server requirements of the project[26]. We chose the Python language either for its simplicity in the code writing either for its thread native module, which ensures a total thread-safety of each variable. Using a set of function decorators we are able to run each function (callback) in a separated-detached thread as required by the previous instructions. The project includes a documentation about its usage (also for general applications) and it can be easily installed via setup.py. In the *FiloBlu* project we used a Neural Network algorithm written in Tensorflow as Machine Learning model. Tensorflow does not allow to run background processes

[25] FIFO container, i.e *First-In-First-Out*, is a special data structure in which the first element added will be processed as first and then automatically removed from it.

[26] The FiloBlu service is a Windows service and it can not run on Unix machines. Moreover, the database used in the project is a MySQL one so the queries and the libraries used are compatible only with this kind of database.

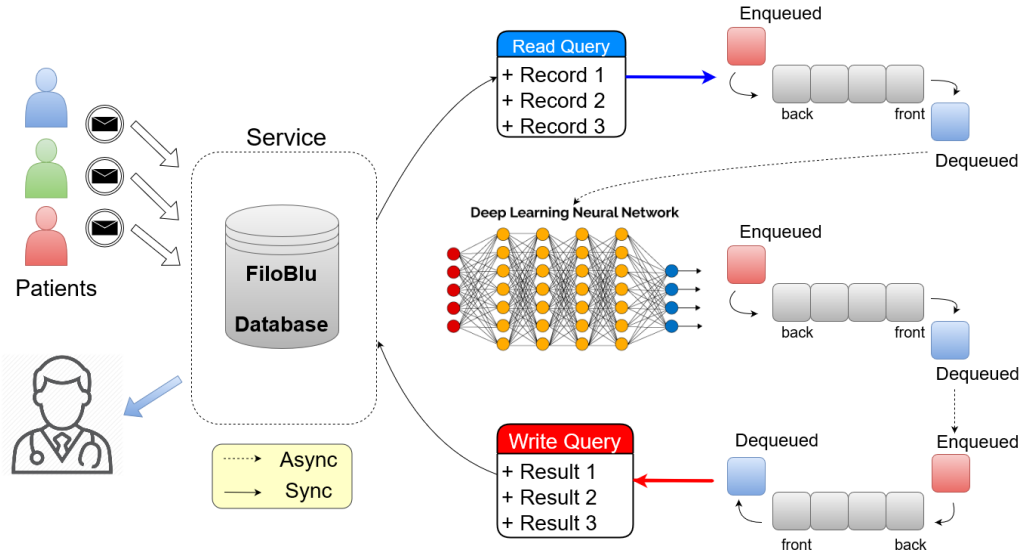


Figure 3.15: FiloBlu Service computation scheme.

directly, so the problem was overcame using a direct call to a Python script which performed the full list of steps into an infinite loop. In this way the service could be restarted also if the process-service was killed. The service can be driven using a simple Powershell script provided in the project.

Data Transmission

In the above configuration we focused on the pipeline which processes the stream of data, ignoring any problem about the communication between the external device and the machine which hosts the service. The *FiloBlu* project uses an external APP to send data to the main server, so we have two systems which have to communicate between them automatically via Internet connection. In general, we could manage sensitive data, that could be vulnerable using an Internet communication. To face this problem we developed a simple TCP/IP client-server package which also supports a RSA cryptography, the CryptoSocket package [24].

The communication security could be an important point in many research applications and a valid cryptography procedure is essential. The RSA cryptography is considered one of the most secure cryptography algorithm for data transmission and it is quite easy to implement. In the CryptoSocket package we implemented a simple wrap around the socket Python library to perform a serialization of our data which are (optionally) processed by our custom RSA algorithm. In this way different kinds of data could be sent by the client at the same time. The client script could be adapted with slight modifications for any user need and also complex Python structures could be transmitted between two machines (to the server). The cryptography module was written in pure C++ for computational efficiency and a Cython wrap was provided for pure-Python applications. CryptoSocket has only demonstrative purpose and so it works only for a 1-by-1 data transmission (1 server and 1 client).

Since this second implementation could be used also for other applications it was treated as a separated project and it has its own open-source code. The CryptoSocket package can be installed via CMake in any platform and operative system and a full list of installation instructions is provided in the project repository. The continuous integration of the project is guaranteed by testing the package installation across multiple C++ compilers and platforms via Travis CI and Appveyor CI.

Appendix F - Bioinformatics Pipeline Profiling

In this work many times we have talked about the performances evaluation of a scripts in terms of time performances and other system statistics. The importance in the understanding the state of our infrastructure is essential not only for ensuring the reliability and stability of a software but also for a more efficiency use of the available resources. In particular about what concern the memory, CPUs and diskIO management is useful to know the required amount of each step of our software to perform the better parallelization strategy. Metrics represent the raw measurements of resource usage that are used by a software or a collection of them. These might be low-level usage summaries provided by the operating system, or they can be higher-level types of data tied to the specific functionality or work of a component. These kind of data could be collected and aggregated by a monitoring system like *Telegraf*[27]. In general, the difference between metrics and monitoring mirrors the difference between data and information. Monitoring takes metrics data, aggregates it, and presents it in various ways that allow humans to extract insights from the collection of individual pieces.

In this section we focused on the importance of software monitoring. In particular we will talk about a work conducted in collaboration with INFN-CNAF of Bologna about the monitoring and the performance evaluation of a bioinformatics pipeline across various computational environments [25].

In this work a previously published bioinformatics pipeline was reimplemented across various computational platforms, and the performances of its steps evaluated. The tested environments were: I) dedicated bioinformatics-specific server II) low-power single node III) HPC single node IV) virtual machine. The pipeline was tested on a use case of the analysis of a single patient to assess single-use performances, using the same configuration of the pipeline to be able to perform meaningful comparison and search the optimal environment/hybrid system configuration for biomedical analysis. Performances were evaluated in terms of execution wall time, memory usage and energy consumption per patient.

GATK-LODn pipeline

The pipeline used in this work, GATK-LODn, has been developed in 2016 by Do Valle et al. [31], and codifies a new approach aimed to Single Nucleotype Polimorphism (SNP) identification in tumors from Whole Exome Sequencing data (WES). WES is a type of "next generation sequencing" data [94, 7, 81], focused on the part of the genome that actually codifies proteins (the exome). Albeit known that non-transcriptional parts of the genome can affect the dynamic of gene expression, the majority of cancers inducing mutations are known to be on the exome, thus WES data allow to focus the computational effort on the most interesting part of the genome. Being the exome in human approximately 1% of the

[27] An automatic installation guide for Telegraf is provided in the Shut [19] project for any OS and also for no-root users.

	Coverage	No. of Reads	Read Length	BAM file size	NGS size
Whole genome	37.7x	975,000,000	115	82 GB	104 GB
Whole genome	38.4x	3,200,000,000	36	138 GB	193 GB
Exome	40x	110,000,000	75	5.7 GB	7.1 GB

Table 3.4: Typical dataset size for a single patient of different types of next generation sequencing. BAM file size refers to the size of the binary file containing the reads from the machine.

total genome, this approach helps significantly in reducing the number of false positives detected by the pipeline. The different sizes of next generation sequencing dataset are shown in Tab 3.4.

The GATK-LODn pipeline is designed to combine results of two different SNP-calling softwares, GATK [62] and MuTect [15]. These two softwares employ different statistical approaches for the SNP calling: GATK examines the healthy tissue and the cancerous tissue independently, and identifies the suspect SNPs by comparing them; Mutect compares healthy and cancerous tissues at the same time and has a more strict threshold of selection. In identifying more SNPs, GATK has a higher true positive calling than Mutect, but also an higher number of false positives. On the other end Mutect has few false positives, but often does not recognize known SNPs. The two programs also call different set of SNPs, even when the set size is similar. The pipeline therefore uses a combination of the two sets of chosen SNPs to select a single one, averaging the strictness of Mutect with the recognition of known variants of GATK.

The pipeline work-flow includes a series of common steps in bioinformatics analysis and in the common bioinformatics pipelines. It includes also a sufficient representative sample of tools for the performances statistical analysis. In this way the results extracted from the single steps analysis could be easily generalized to other standard bioinformatics pipelines.

With the increasing demand of resources from ever-growing datasets, it is not favorable to focus on single server execution, and is better to distribute the computation over cluster of less powerful nodes. The computational pipeline also has to manage a high number of subjects, and several steps of the analyses are not trivial to be done in a highly parallel way. Thus, the importance of system statistics management as the efficiency usage of available resources are crucial to reach a compromise between computational execution time and energy cost. For these reasons our main focus is on the performance evaluation of a single subject without using all the available resources, as these could be more efficiently allocated to concurrently execute several subjects at the same time. Due to the nature of the employed algorithms, not all steps can exploit the available cores in a highly efficient way: some scales sub-linearly with the number of cores, some have resource access bottleneck. Other tools are simply not implemented with parallelism in mind, often because they are the result of the effort of small teams that prefer to focus their attention on the scientific development side rather than the computational one.

Moreover in order to obtain an optimal execution of bioinformatics pipelines, each analysis step might need very different resources. This means that any suboptimal component of a server could act as a bottleneck, requiring bleeding edge technology if all the steps are to be performed on a single machine. Hybrid systems could be a possible solution to these issues, but designing them requires detailed information about how to partition the different steps of the pipeline.

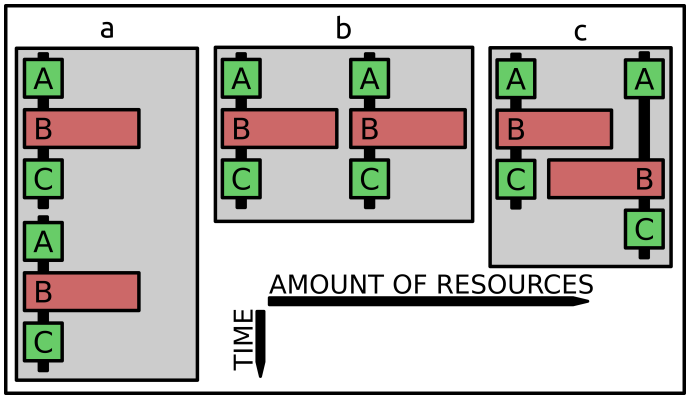


Figure 3.16: Examples of concurrency work-flow of two processes. The first case (a) represents a simple (naive) sequential work-flow; the second (b) highlights a brute force parallelization; the third (c) is the case of a perfect match between the available resources and the requested resources. Often brute force parallelization of pipelines done as in the image b ends up overlapping the most computationally intensive steps. Measuring the minimum viable requirements for the execution allow to better allocate resources as seen in the image c.

Computational Environments

There are two main optimization strategies: the first is to improve the efficiency of a single run on a single patient and the second is to employ massive parallelization on various samples. In both cases we have to know the necessary resources of the pipeline (and in a fine grain the resources of each step) and the optimal concurrency strategy to be applied to our work-flow (see Fig. 3.16). In the analyses we want to highlight limits and efficiencies of the most common computational environments used in big data analytics, without any optimization strategy of the codes or systems.

We also focused on a single patient analysis, the base case study to design a possible parallelization strategy. This is especially relevant for the multi-sample parallelization, that is the most promising of the two optimization strategies, as it does not rely on specific implementations of the softwares employed in the pipeline.

The pipeline was implemented on 5 computational environments: 1 server grade machine (Xeon E52640), 1 HPC node (Xeon E52683), 2 low power machines (Xeon D and Pentium J) and one virtual machine built on an AMD Opteron hypervisor. The characteristics of each node are presented in Tab. 3.5.

The server - grade node is a typical node used for bioinformatics computation, and as such features hundreds of GB of memory with multiple cores per motherboard: for these reasons we chose it as reference machine and the following results are expressed in relation to it.

The two low - power machines are designed to have a good cost - to - performance ratio, especially for the running cost[28]. These machines have been proven to be a viable solution for high performance computations [13]. Their low starting and running cost mean that a cluster of these machines would be more accessible for research groups looking forward to increase their computational power.

The last node is a virtual machine, designed to be operated in a cloud environment.

The monitoring tool used is *Telegraf*, which is an agent written in Go for collecting, processing, aggregating, and writing metrics. Each section of the pipeline sends messages

[28] Running cost is evaluated as the energy consumption that the node requires per subject, assuming that the consumption scales linearly with the number of cores used in the individual step.

CLASS	**server grade machines**		**low power machines**		**virtual machine**
CPU	Intel Xeon	Intel Xeon	Intel Pentium	Intel Xeon	AMD Opteron
version	E5-2683v3	E5-2640v2	J4205	D-1540	6386 SE
Microarchitecture	Haswell	Ivy Bridge EP	Apollo Lake	Broadwell	Piledriver
Launch Date	Q3'14	Q3'13	Q4'16	Q1'15	Q3'12
Lithography	22 nm	22 nm	14 nm	14 nm	32 nm
Cores/threads	14/28	8/16	4/4	8/16	16
Base/Max Freq	2.00/3.00	2.00/2.50	1.50/2.60	2.00/2.60	2.80/3.50
L2 Cache	35 MB	20 MB	2 MB	12 MB	16 MB
TDP	120 W	95 W	10 W	45 W	115 W
Total CPUs	2	2	1	1	1
total cores/threads	28/56	16/32	4/4	8/16	16
Total Memory	256 GB	252 GB	8 GB	32 GB	60 GB
System power	240 + 60 W	190 + 60 W	10 + 2 W	45 + 10 W	115 + 10 W
Electrical costs	650 €/year	550 €/year	26 €/year	120 €/year	273€ /year
System price	4000-6000 €	3000-5000 €	100-130 €	900-1200 €	2000-3000€

Table 3.5: Characteristics of the tested computational environments. Electrical costs are estimated as 0.25 €/kWh; CPU frequencies are reported in GHz; TDP: Thermal Design Power, an estimation indicator of maximum amount of heat generated by a computer chip when a "real application" runs.

to the *Telegraf* daemon independently.

Regardless of the number of cores of each machine we restrict the number of cores used to only two to compare the statistics: this restriction certainly penalize the environment with multiple cores but with a view of maximizing the parallelizations and minimize the energy cost it is the playground to compare all the available environments. Another restriction is applied to the chosen architectures: since available low - power machines provides only x86 - architectures also the other environments are forced to work in x86 to allow the statistics comparison.

Pipeline steps

The pipeline steps that have been examined are a subset of all the possible steps: we only focus on those whose computational requirements are higher and thus require the most computational power. These steps are:

1. **mapping:** takes all the reads of the subjects and maps them on the reference genome;
2. **sort:** sorts the sequences based on the alignment, to improve the reconstruction steps;
3. **markduplicates:** checks for read duplicates (that could be imperfections in the experimental procedures and would skew the results);
4. **buildbamindex:** indexes the dataset for faster sorting;
5. **indexrealigner:** realigns the created data index to the reference genome;
6. **BQSR:** base quality score recalibration of the reads, to improve SNPs detection;
7. **haplotypecaller:** determines the SNPs of the subject;

8. **hardfilter:** removes the least significant SNPs.

The following statistics were evaluated:

1. **memory per function:** estimate percentage of the total memory available to the node used for each individual step of the pipeline;
2. **energy consumption:** estimated as the time taken by the step, multiplied by the number of cores used in the step and the power consumption per core (TDP divided by the available cores). As mentioned before this normalization unavoidably penalize the multi-core machines but give us a term of comparison between the different environment;
3. **elapsed time:** wall time of each step.

The pipeline was tested on the patient data from the 1000 genome project with access code NA12878, sample SRR1611178. It is referred as a Gold Standard reference dataset [93]. It is generated with an Illumina HiSeq2000 platform, SeqCap EZ Human Exome Lib v3.0 library and have a 80x coverage. As Gold Standard reference it is commonly used as benchmark of new algorithm and for our purpose can be used as valid prototype of genome.

Results

Memory occupation is one of the major drawbacks of the bioinformatics pipelines, and one of the greater limits to the possibility of parallel computation of multiple subjects at the same time. As it can be seen in Fig. 3.17, the memory occupation is comprised between 10% and 30% on all the nodes. This is due to the default behavior of the GATK libraries to reserve a fixed percentage of the total memory of the node. The authors could not find any solution to prevent this behavior from happening. As it can be noticed, in the node with the greatest amount of total memory (both Xeon E5 and the virtual machine) the requested memory is approximately stable, as is always sufficient for the required task. The memory allocation is less stable in the nodes with a limited memory (Xeon D and Pentium J), as GATK might requires more memory than what initially allocated to perform the calculation. The exception to this behavior is the *mapping* step, that uses a fixed amount of memory independently from the available one (between 5 and 7 GB). This is due to the necessity of loading the whole human reference genome (version hg19GRCh37) to align each individual read to it. All the other steps do not require the human reference genome but can work on the individual reads, allowing greater flexibility in memory allocation.

As can be seen in Fig. 3.18 and Fig. 3.19, this increase of memory consumption does not correspond to a proportional improvement of the time elapsed in the computation.

The elapsed time for each step and for the whole pipeline can be seen in Fig. 3.18. It can be seen that there is a non consistent trend in the behavior of the different environments. Aside from the most extreme low power machine, the pentium J, the elapsed times are on average higher for the low power and slightly higher for the cloud node, but the time for the individual rule can vary. In the sorting step, Pentium J is 20 times slower than the reference. This is probably due to the limited cache and memory size of the pentium J, that are both important factors determining the execution time of a sorting algorithm and are both at least four to six times smaller than the other machines. The HPC machine, the Xeon E52683, is consistently faster than the reference node.

The energy consumption per step can be seen in Fig. 3.19. The low power machines are consistently less than half the baseline consumption. Even considering the peak of

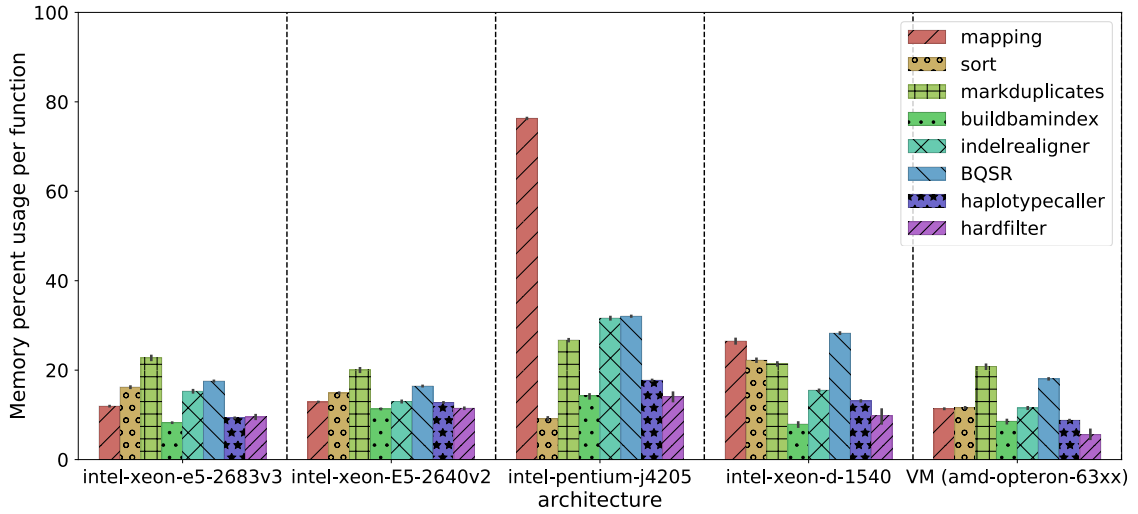


Figure 3.17: Memory used for each step of the pipeline. Due to the GATK memory allocation strategy, all steps use a baseline amount of memory proportional to the available memory. Smaller nodes, like the low power ones, require more memory as the baseline allocated memory is not sufficient to perform the calculation.

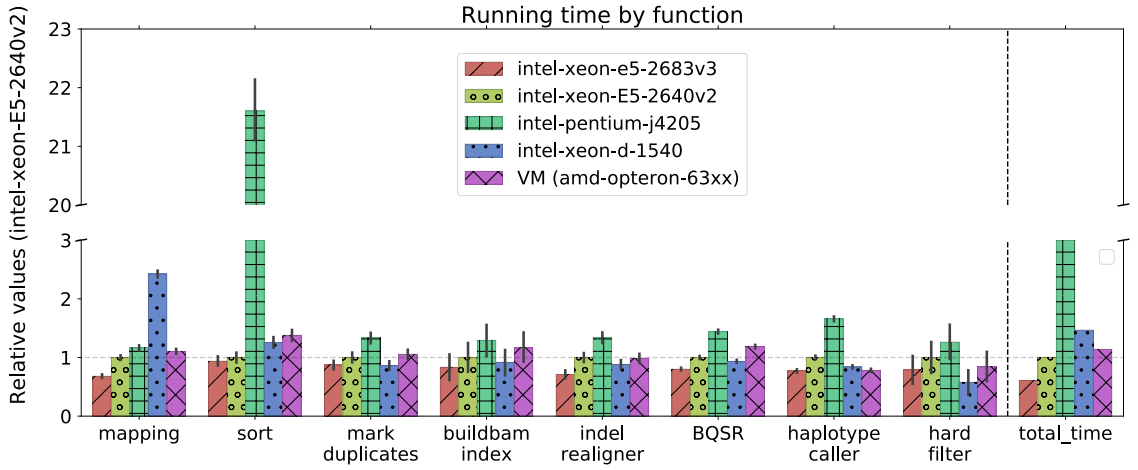


Figure 3.18: Time elapsed per step of the pipeline, and total elapsed time. In the sorting step, Pentium J is 20 times slower than the reference, probably due to the limited cache size.

consumption due to the long time required to perform the sorting, the most efficient low power machine, the pentium J, consumes 40% of the reference, and the Xeon D consumes 60% of the reference. The HPC machine, the Xeon E52683, have consumption close to the low power nodes, balancing out the higher energy consumption with a faster execution speed. The virtual machine has the highest consumption despite the fact that the execution time of the whole pipeline is comparable to the reference due to the high TDP compared to its execution time.

Conclusions

Bioinformatics pipelines are one of the most important uses of biomedical big data and, at the same time, one of the hardest to optimize, both for their extreme requisites and the constant change of the specification, both in input-output data format and program API.

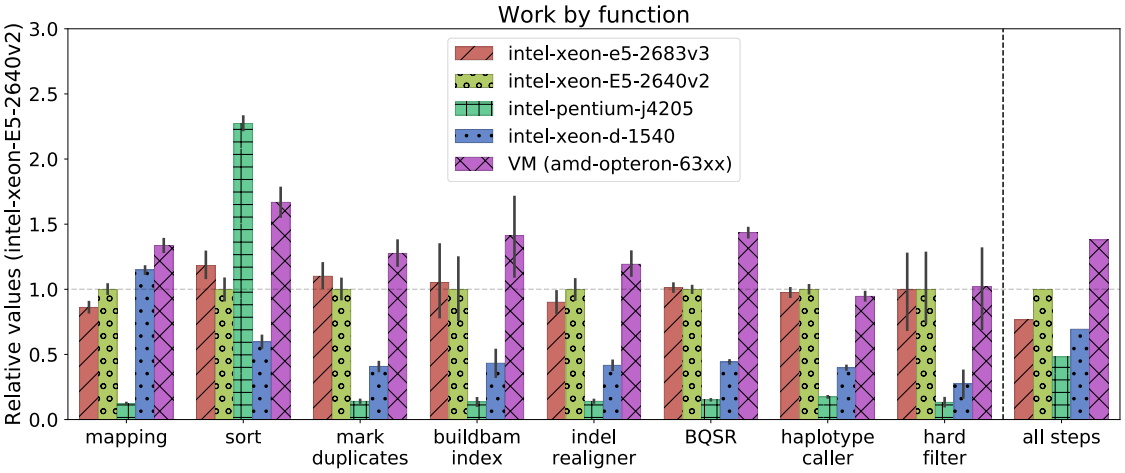


Figure 3.19: Energy consumption per pipeline step and on the whole pipeline. Energy consumption is estimated as the time taken by the step, multiplied by the number of cores used in the step and the power consumption per core (TDP divided by the available cores).

This makes the task of pipeline optimization a daunting one, especially for the final target of the results; physicians and biologists could lack the technical expertise (and time) required to optimize each new version of the various softwares of the pipelines. Moreover, in a verified pipeline updating the software included without a long and detailed cross-validation with the previous one is often considered a bad practice: this means that often these pipelines are running with under-performing versions of each software.

Clinical use of these pipelines is growing, in particular with the rise of the concept of *personalized medicine*, where the therapy plan is designed on the specific genotype and phenotype of the individual patient rather than on the characteristic of the overall population. This would increase the precision of the therapy and thus increase its efficacy, while cutting considerably the trial and error process required to identify promising target of therapy. This requires the pipelines to be evaluated in real time, for multiple subjects at the same time (and potentially with multiple samples per subject). To perform this task no single node is powerful enough, and thus it is necessary to use clusters. This brings the need to evaluate which is the most cost and time efficient node that can be employed.

In the cost assessment there are several factors that need to be considered aside of the initial setup cost, namely cost for running the server and opportunity cost for obsolescence. Scaled on medium sized facilities, such the one that could be required for a hospital, this cost could quickly overcome the setup cost. This cost does also include not only the direct power consumption of the nodes, but also the required power for air conditioning to maintain them in the working temperature range. Opportunity costs are more complex, but do represent the loss of possibility of using the most advanced technologies due to the cost of the individual node of the cluster. Higher end nodes require a significant investment, and thus can not be replaced often.

With this perspective in mind, we surmise that energy efficient nodes present an interesting opportunity for the implementation of these pipelines. As shown in this work, these nodes have a low cost per subject, paired with a low setup cost. This makes them an interesting alternative to traditional nodes as a workhorse node for a cluster, as a greater number of cores can be bought and maintained for the same cost.

Given the high variability of the performances in the various steps, in particular with the sorting and mapping steps, it might be more efficient to employ a hybrid environment, where few high power nodes are used for specific tasks, while the bulk of the computation is done by the energy efficient nodes. This is true even for those steps that can be massively

parallelized, such as the mapping, as they benefit mainly from a high number of processors rather than few powerful ones. In this work we focused only on CPUs computation, but another possibility could be an hybrid-parallelization approach in which the use of a single GPU accelerator can improve the parallelization of the slower steps. Each pipeline work-flow requires its own analyses and tuning to reach the best performances and the right parallelization strategy based on the use which it is intended but a low energy node approach is emerging as a good alternative to the more expensive and common solutions.

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] A. Battle, S. Mostafavi, X. Zhu, J. B. Potash, M. M. Weissman, C. McCormick, C. D. Haudenschild, K. B. Beckman, J. Shi, R. Mei, A. E. Urban, S. B. Montgomery, D. F. Levinson, and D. Koller. Characterizing the genetic basis of transcriptome diversity through rna-sequencing of 922 individuals. *Genome Research*, 2014.

[6] J. S. Beckmann and D. A. Lew. Reconciling evidence-based medicine and precision medicine in the era of big data: challenges and opportunities. In *Genome Medicine*, 2016.

[7] S. Behjati and P. S. Tarpey. What is next generation sequencing? *Archives of disease in childhood - Education & practice edition*, 98(6):236–238, 2013.

[8] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[9] R. Bellman and K. M. R. Collection. *Adaptive Control Processes: A Guided Tour*. Princeton Legacy Library. Princeton University Press, 1961.

[10] V. Boccardi, L. Paolacci, D. Remondini, E. Giampieri, G. Poli, N. Curti, R. Cecchetti, A. Villa, C. Ruggiero, S. Brancorsini, and P. Mecocci. Cognitive decline and alzheimer's disease in the old age: identified a sex specific "cytokinome signature". Oct 2019.

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

[12] C. Cenik, E. S. Cenik, G. W. Byeon, F. Grubert, S. I. Candille, D. Spacek, B. Alsallakh, H. Tilgner, C. L. Araya, H. Tang, E. Ricci, and M. P. Snyder. Integrative analysis of rna, translation, and protein levels reveals distinct regulatory variation across humans. *Genome Research*, 2015.

[13] D. Cesini, E. Corni, A. Falabella, A. Ferraro, L. Morganti, E. Calore, S. F. Schifano, M. Michelotto, R. Alfieri, R. De Pietri, T. Boccali, A. Biagioni, F. Lo Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, E. Pastorelli, and P. Vicini. Power-efficient computing: Experiences from the cosa project. *Scientific Programming*, 2017, 2017.

[14] I. S. Chan and G. S. Ginsburg. Personalized medicine: Progress and promise. *Annual Review of Genomics and Human Genetics*, 12(1):217–244, 2011. PMID: 21721939.

[15] K. Cibulskis, M. S. Lawrence, S. L. Carter, A. Sivachenko, D. Jaffe, C. Sougnez, S. Gabriel, M. Meyerson, E. S. Lander, and G. Getz. Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nature Biotechnology*, 31(3):213–219, 2013.

[16] N. Curti. BlendNet: Blender network viewer. https://github.com/Nico-Curti/BlendNet, 2019.

[17] N. Curti. DNetPRO pipeline: Implementation of the dnetpro pipeline for tcga datasets. https://github.com/Nico-Curti/DNetPRO, 2019.

[18] N. Curti. FiloBlu: Machine learning as service. https://github.com/Nico-Curti/FiloBluService, 2019.

[19] N. Curti. Shut: Shell utilities for no-root users. https://github.com/Nico-Curti/Shut, 2019.

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

[21] 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.

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

[23] N. Curti and D. Dall'Olio. Scorer: Machine learning scorer library. https://github.com/Nico-Curti/Scorer, 2019.

[24] N. Curti and A. Fabbri. Cryptosocket - tcp/ip client server with rsa cryptography. https://github.com/Nico-Curti/CryptoSocket, 2019.

[25] N. Curti, E. Giampieri, A. Ferraro, C. Vistoli, E. Ronchieri, D. Cesini, B. Martelli, C. Duma Doina, and G. Castellani. Cross-environment comparison of a bioinformatics pipeline: Perspectives for hybrid computations. *Springer, Cham, Euro-Par 2018: Parallel Processing Workshops*, 11339, 2019.

[26] N. Curti, E. Giampieri, G. Levi, G. Castellani, and D. Remondini. Dnetpro: A network approach for low-dimensional signatures from high-throughput data. *bioRxiv*, 2019.

[27] N. Curti, E. Giampieri, C. Mizzi, A. Fabbri, A. Bazzani, G. Castellani, and D. Remondini. A network approach for dimensionality reduction from high-throughput data. volume Conference paper, 2019.

[28] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

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

[30] M. Desai and M. Mehta. Techniques for sentiment analysis of twitter data: A comprehensive survey. pages 149–154, 04 2016.

[31] Í. F. do Valle, E. Giampieri, G. Simonetti, A. Padella, M. Manfrini, A. Ferrari, C. Papayannidis, I. Zironi, M. Garonzi, S. Bernardi, M. Delledonne, G. Martinelli, D. Remondini, and G. Castellani. Optimized pipeline of MuTect and GATK tools to improve the detection of somatic single nucleotide polymorphisms in whole-exome sequencing data. *BMC Bioinformatics*, 17(S12):341, 2016.

[32] 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.

[33] L. Eckhard. A universal selection method in linear regression models. *Open Journal of Statistics*, 2, 2012.

[34] G.-S. et al. DisGeNET: a comprehensive platform integrating information on human disease-associated genes and variants. *Nucleic Acids Research*, 45(D1):D833–D839, 10 2016.

[35] 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".

[36] K. Fukunaga. *Introduction to Statistical Pattern Recognition (2Nd Ed.)*. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[37] 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.

[38] C. Greene, J. Tan, M. Ung, J. Moore, and C. Cheng. Big data bioinformatics. *Journal of cellular physiology*, 229(12), 2014.

[39] C. J. e. a. Grondin. The Comparative Toxicogenomics Database: update 2019. *Nucleic Acids Research*, 47(D1):D948–D954, 09 2018.

[40] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Mach. Learn.*, 2002.

[41] S. Haghighi, M. Jasemi, S. Hessabi, and A. Zolanvari. PyCM: Multiclass confusion matrix library in python. *Journal of Open Source Software*, 3(25):729, may 2018.

[42] M. D. Harris, A. E. Anderson, C. R. Henak, B. J. Ellis, C. L. Peters, and J. A. Weiss. Finite element prediction of cartilage contact stresses in normal human hips. *Journal of Orthopaedic Research*, 30(7):1133–1139, 2012.

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

[44] C. A. e. a. Hidalgo. A dynamic network approach for the study of human phenotypes. *PLOS Computational Biology*, 5(4):1–11, 04 2009.

[45] R. R. Hocking. A biometrics invited paper. the analysis and selection of variables in linear regression. *Biometrics*, 32(1):1–49, 1976.

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

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

[48] J. J. Hughey and A. J. Butte. Robust meta-analysis of gene expression using the elastic net. *Nucleic Acids Research*, 2015.

[49] 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.

[50] 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.

[51] T. M. Johnson. Perspective on precision medicine in oncology. *Pharmacotherapy: The Journal of Human Pharmacology and Drug Therapy*, 37(9):988–989, 2017.

[52] J. Koster and S. Rahmann. Snakemake - a scalable bioinformatics workflow engine. *Bioinformatics*, 28:2520–2522, 08 2012.

[53] D. Kumari and R. Kumar. Impact of biological big data in bioinformatics. *International Journal of Computer Applications*, 101(11):22–24, 2014.

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

[55] M. Lee, K. Lee, N. Yu, I. Jang, I. Choi, P. Kim, Y. E. Jang, B. Kim, S. Kim, B. Lee, J. Kang, and S. Lee. Chimerdb 3.0: an enhanced database for fusion genes from cancer transcriptome and literature data mining. *Nucleic Acids Research*, 45, Jan 2017.

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

[57] J. Loscalzo, I. Kohane, and A.-L. Barabasi. Human disease classification in the postgenomic era: a complex systems approach to human pathobiology. *Molecular systems biology*, 3, Jul 2007.

[58] A. e. a. Maciejewski. DrugBank 5.0: a major update to the DrugBank database for 2018. *Nucleic Acids Research*, 46(D1):D1074–D1082, 11 2017.

[59] M. Malvisi, N. Curti, D. Remondini, G. Gandini, F. Palazzo, G. Pagnacco, J. L. Williams, and G. Minozzi. Combinatorial discriminant analysis applied to rnaseq data reveals a set of 10 transcripts as signatures of infection of cattle with mycobacterium avium subsp. paratuberculosis. *Animals (MDPI)*, submitted, 2019.

[60] G. Martìnez-Mekler, R. A. Martìnez, M. B. del Rìo, R. Mansilla, P. Miramontes, and G. Cocho. Universality of rank-ordering distributions in the arts and sciences. *PLoS One*, 11 2009.

[61] V. Marx. The big challenges of big data. *Nature Reviews*, 498(255), 2013.

[62] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo. The genome analysis toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, 2010.

[63] M. McKinney. Human genome project information. *Reference Reviews*, 26(3):38–39, 2012.

[64] C. Mizzi, A. Fabbri, S. Rambaldi, F. Bertini, N. Curti, S. Sinigardi, R. Luzi, G. Venturi, M. Davide, G. Muratore, A. Vannelli, and A. Bazzani. Unraveling pedestrian mobility on a road network using icts data during great tourist events. *EPJ Data Science*, 7(1):44, Oct 2018.

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

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

[67] H. Pang, S. L. George, K. Hui, and T. Tong. Gene selection using iterative feature elimination random forests for survival outcomes. *IEEE/ACM Transactions on Computational Biology and Bioinformatics / IEEE*, 2012.

[68] 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.

[69] I. Posa, S. Carvalho, J. Tavares, and A. R. Grosso. A pan-cancer analysis of myc-pvt1 reveals cnv-unmediated deregulation and poor prognosis in renal carcinoma. *Oncotarget*, 7(30):47033–47041, 2016.

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

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

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

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

[74] J. A. Reuter, D. V. Spacek, and M. P. Snyder. High-throughput sequencing technologies. *Molecular Cell*, 2015.

[75] L. Richardson. Beautiful soup documentation. *April*, 2007.

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

[77] B. Ru, C. N. Wong, Y. Tong, J. Y. Zhong, S. S. W. Zhong, W. C. Wu, K. C. Chu, C. Y. Wong, C. Y. Lau, I. Chen, N. W. Chan, and J. Zhang. TISIDB: an integrated repository portal for tumor–immune system interactions. *Bioinformatics*, 03 2019. btz210.

[78] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. *arXiv e-prints*, page arXiv:1801.04381, Jan 2018.

[79] K. Scotlandi, D. Remondini, G. Castellani, M. C. Manara, F. Nardi, L. Cantiani, M. Francesconi, M. Mercuri, A. M. Caccuri, M. Serra, S. Knuutila, and P. Picci. Overcoming resistance to conventional drugs in ewing sarcoma and identification of molecular predictors of outcome. *Journal of Clinical Oncology*, 27(13):2209–2216, 2009. PMID: 19307502.

[80] K. Sharma, F. Qian, H. Jiang, N. Ruchansky, M. Zhang, and Y. Liu. Combating fake news: A survey on identification and mitigation techniques, 2019.

[81] J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature Biotechnology*, 26(10):1135–1145, 2008.

[82] 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.

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

[84] J. Siek, L.-Q. Lee, and A. Lumsdaine. Boost graph library. http://www.boost.org/libs/graph/, June 2000.

[85] A. Sood, M. Hooda, S. Dhir, and M. Bhatia. An initiative to identify depression using sentiment analysis: A machine learning approach. *Indian Journal of Science and Technology*, 11(4), 2018.

[86] C. Terragna, D. Remondini, M. Martello, E. Zamagni, L. Pantani, F. Patriarca, A. Pezzi, G. Levi, M. Offidani, I. Proserpio, G. De Sabbata, P. Tacchetti, C. Cangialosi, F. Ciambelli, C. V. Viganò, F. A. Dico, B. Santacroce, E. Borsi, A. Brioli, G. Marzocchi, G. Castellani, G. Martinelli, A. Palumbo, and M. Cavo. The genetic and genomic background of multiple myeloma patients achieving complete response after induction therapy with bortezomib, thalidomide and dexamethasone. *Oncotarget*, 2016.

[87] D. S. e. a. Wishart. HMDB 4.0: the human metabolome database for 2018. *Nucleic Acids Research*, 46(D1):D608–D617, 11 2017.

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

[89] Y. Yuan, E. M. Van Allen, L. Omberg, N. Wagle, A. Amin-Mansour, A. Sokolov, L. A. Byers, Y. Xu, K. R. Hess, L. Diao, L. Han, X. Huang, M. S. Lawrence, J. N. Weinstein, J. M. Stuart, G. B. Mills, L. A. Garraway, A. A. Margolin, G. Getz, and H. Liang. Assessing the clinical utility of cancer genomic and proteomic data across tumor types. *Nature Biotechnology*, 32(7):644–652, 2014.

[90] X. Zhou and et al. Human symptoms–disease network. *Nature Communications*, 5.

[91] X. Zhou and R. Zafarani. Fake news: A survey of research, detection methods, and opportunities, 2018.

[92] M. Zitnik, R. Sosič, S. Maheshwari, and L. Jure. BioSNAP Datasets: Stanford biomedical network dataset collection. http://snap.stanford.edu/biodata, Aug 2018.

[93] J. M. Zook, B. Chapman, J. Wang, D. Mittelman, O. Hofmann, W. Hide, and M. Salit. Integrating human sequence data sets provides a resource of benchmark snp and indel genotype calls. *Nature Biotechnology*, 32, 2014.

[94] M. Zwolak and M. Di Ventra. Colloquium: Physical approaches to DNA sequencing and detection. *Reviews of Modern Physics*, 80(1):141–165, 2008.