POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Gestionale (Engineering and Management)

Tesi di Laurea Magistrale

Resolution of a scheduling problem with the aid of machine learning techniques

Relatore
Prof. Federico Della Croce di Dojola

Candidato
Riccardo Schiano di Cola

A.A. 2017/2018

Alla mia famiglia,
a chi ne fa già parte e a chi ne farà.
Perché senza di loro,
non sarei arrivato fin qui.

Contents

1. Introduction

Over the last fifty years, a lot of researches have been focused on the possibility to solve specific problems with the aid of machines. From the first step on the moon, up to nowadays self-driving cars, things have changed.

In the sixties, Gordon Moore, one of the founders of the microprocessors company Intel, wrote a famous paper (MOORE, 1965) with a law inside that is still working today. It says that the number of components per integrated circuit would have doubled every year. However, ten years after, in 1975, he revised the forecast to doubling roughly every two years. In this appearance, the Moore's Law has continued unabated up to the present, for 50 years. During this time, the overall progress in the technology reached a factor of around two billion. This means that , in general terms, computers are now two billion times as powerful as in the 50's. It seems very hard to comprehend this law, but however, it could be useful to make it a comparison with other industries. Applying Moore's Law to airline industries would increase the speed of actual airplanes to nearly the speed of the light, decreasing their cost to less than one dollar.

Going back to the first step on the moon, the Apollo Guidance Computer was equipped with a computing power of around two Nintendo consoles. Specifically, the Nintendo Entertainment System, which is a console entering the European market in 1985. This means that, in 20 years, the increase in power of computing units, made it possible to use some previous excellent product on a home product used for entertaining. In the same year, the Cray Research released a Supercomputer whose power has been uncontested for five years long. The same computing power has been matched by the Apple iPhone 4 in 2010.

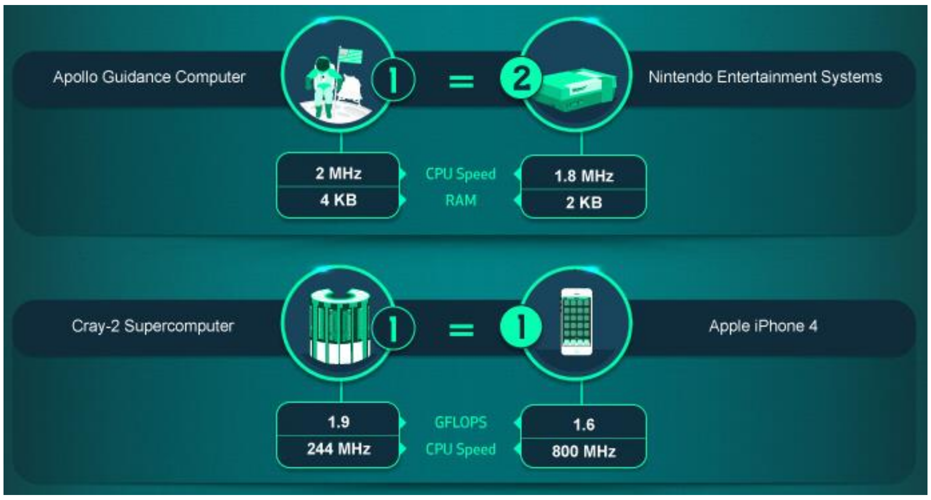


Figure 1.1: Comparison between processors during the technology growth.

With this growing in power of processors, computation have become more and more easy to accomplish. This growth affected also related scientific fields. Not only the field of computer engineering and associated, but also other branches requiring heavy computations in the scientific field. A good measure of the speed of a processor can be FLOPS, which stands for Floating Operations Per Second. A floating operation is an elementary operation, a sum or a product as well as the difference and the division on floating numbers, i.e. decimal numbers. Nowadays, a classic smartphone as the Samsung Galaxy S6 can make around 35 billion FLOPS while the PlayStation 4 moves around the 1'800 Billion FLOPS. Understanding the power of simple end products delivered to the almost totality of the population, could help understanding the power reached by the specific supercomputers nowadays.

However, why do we even need this computational power? One reason is that, some of the biggest challenges faced by human are extremely complicated. As an example, climate modelling and life science are one of those. This move forward of the technology raises another question. Is it possible to conceive a computer as powerful as human brain? To explain this, the following graph highlights some milestone during the evolution of this process. Even if human brain seem far behind, computers will go on increasing their computational power making complex tasks always easier and easier.

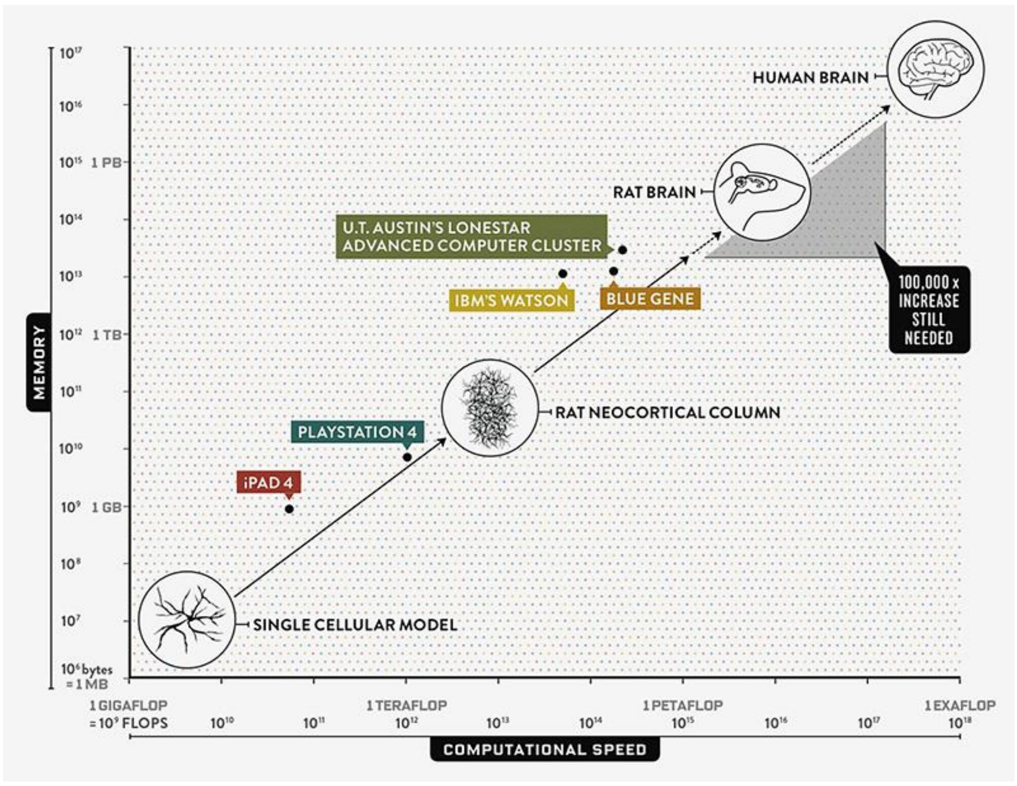


Figure 1.2: Graph showing the trend of the power of several well-known technological devices compared to some examples from natural world.

Shoulder to shoulder to the increasing computational power of computers goes the increasing in achievements made by scientist in fields where computers and machine have been used as calculators. One of those is the Operation Research field. It is a sub-field of applied mathematics using techniques from mathematical modelling, statistical analysis, and mathematical optimization to reach optimal or near-optimal solution for complex decision-making problems.

In this field, usually computers are programmed to perform algorithms and solve problems. Sometimes, for difficult problem, some algorithms may even present some problem-specific features. Usually, an algorithm is a set of specific instruction that the computer has to follow one after another to reach the solution of the problem. The most well-known problems addressed to this discipline are: critical path problem, facility location, scheduling, routing, personnel staffing, and determining optimal prices.

Furthermore, the growth of computers enhanced the creation of solver and modelers by software houses. These software include algorithms and helpful tools aiding the Operation Research scientists in progressing and finding new solution to problems. Nowadays, model of around hundreds of thousands of variable and constraints can be solved in matter of seconds and/or minutes. Moreover, the large volume of data required can be stored and manipulated in a very efficient way. This fact strongly helped and enhanced the progress made in this scientific field.

Another field experiencing a constant breakthrough due to the growing in power of computers is the Machine Learning field. It studies algorithms and models that make computer systems progressively improve their performance on a specific task. Usually, machine learning model are created to perform a task without being explicitly programmed to perform it. This means that, instead of using an algorithm to perform a task, machine learning models learn to perform the task looking at task already performed. This creates models able to make prediction on several situation. Usually, this strategy is used where it is difficult to describe a task with set of steps to perform. Email detection, image classification and speech recognition are the most well-known example in this field.

As it can be seen, the two fields present two approach which seem quite distant one from another. The former looks at solution coming out from a set of specific instruction, i.e. algorithms, coded with programming languages via computers. The latter focus on learning to accomplish a task without programming the computer to perform that specific tasks. Operation Research has this standard approach because it is easy to codify an algorithm using sets of simple instructions. However, when it comes to Machine Learning, this discipline provides solution to problems that are quite difficult to codify. So, it applies a non-standard approach in performing these tasks.

The focus of this work is to try to gather the two approaches. The next chapter will briefly introduce to the Operation Research field and specifically to the Scheduling sub-field. An overall introduction of this sub-field will be provided with a focus on the problem to be examined. The problem is a Two-machine Flow Shop Problem minimizing the sum of completion times. In this work will be considered the approach proposed in Della Croce et al. (2014), that is a matheuristic procedure able to outperform the state of the art.

Then, a chapter regarding the Machine Learning field follows the previous one. All the bases are provided with a specific focus on Deep Learning, a sub-field of Machine Learning, treating Artificial Neural Networks. All elements of these special networks are provided together with the path that is usually followed in building these models. Specifically, an entire section of this chapter is dedicated to Recurrent Layer and Long Short-Term Memory cells due to its importance in the final model.

After that, a chapter explaining how the problem has been faced starting from the creation of the dataset up to the training of the network, is presented. This chapter follows the path it has been followed during the research work. Two section are dedicated to the usage of Python and the Keras library. These tools helped a lot the prototyping giving the opportunity of focusing on the central core of the work without being concerned by the coding issue.

Following this chapter, there is another one focused on the creation of a predictor and the test of the results of the network on completely new and previously unseen data. This is an important step in Machine Learning field since it ensures that the network has the ability to perform not only on data that has already been seen but also on new data. This helps generalization and avoid the memorization by the model. After that, a section explaining the results obtained and the analysis performed follow. Then a rapid section explaining how the new matheuristic should work is provided. At the end, concluding remarks will complete the work.

2. Scheduling Problems

Scheduling focuses its attention on the optimal allocation of resources to activities over time. Nowadays, in a very highly competitive environment, it plays a critical role both for manufacturing and production systems as well as in services industries. Regarding the area in which scheduling is applied, resources and tasks can be very different one each other. In airline industry, resources could take the form of runways in airports and tasks might be landing and take-offs. For a construction company, resources and tasks could be crews assigned to specific construction phases of the project. Instead, an internet-related industry company can see resources as processing units in a computing environment and tasks as executions of software on these machines. In literature, the most common environment where scheduling problems take place is the manufacturing one. Here, resources are machines performing operations on jobs, the tasks.

Due to its practical importance, since the early 1950's, a considerable amount of research effort has been made on the subject. The number and variety of the models that can be created are virtually unlimited and most of the research has focused on deterministic machine scheduling as well as stochastic scheduling. In this environment, a *machine* is a resource that can perform only one operation at time, and the activity performed is referred as *jobs*. Moreover, a job is processed by only one machine at time.

In real world applications, some parameters may not be known in advance. Due to this feature, scheduling problems are divided in two main categories. In the deterministic field, it is assumed that all the information assigned to a specific problem are completely known, with certainty, in advance. Instead, the stochastic scheduling deals with the uncertainty of the parameters. Throughout this work, only deterministic cases are presented.

2.1. Framework and Notation

During the research time, a notation has evolved to capture the structure of the problems considered in literature and to completely describe it.

Usually, the number of jobs is denoted by n, and the number of machines by m. When subscripts i, j are used, typically i relates to the machines and j to the jobs.

Processing Time (p_{ij})**:** this represents the processing time of job j on machine i. The subscript i could be omitted if the processing time of job j does not depend on the machine or there is only a single machine.

Release Date (r_j)**:** This is the time the job is arrived at the system, i.e., the earliest time the job can be processed on a machine.

Due date (d_j)**:** Due date represents the expected completion date of the job j, that is the time at which the job should be shipped to the customer. If the date must be respected, it is also called *deadline* and it is referred as $\overline{d_j}$.

Weight (w_j)**:** the weight of a job is a kind of priority factor, evidencing the importance of the single job relatively to the others in the system.

Most of the time, a scheduling problem is defined by a notation like $\alpha|\beta|\gamma$. The field containing α describes the environment of the machines, while the β entry provides details about the characteristics of processing rules. Finally, the γ field describes the objective to be minimized.

In the α fields, these are these possible choices:

Single Machine** (1)**: this is the easiest case environment where there is only one machine processing jobs.

Identical machines in parallel** (Pm)**: in this scenario there are *m* machine in parallel. Job *j* requires a single operation that might be performed on all the machines or on a subset of those.

Machine in parallel with different speeds** (Qm)**: there are *m* machines in parallel where with different speeds v_i. The time, p_{ij}, the job *j* spends on the machine *i* is the ratio p_j / v_i.

Unrelated machines in parallel** (Rm)**: this environment can be seen as a generalization of the previous one. There are *m* machines in parallel where machine *i* can process job *j* at speed v_{ij}. Again, the time spent by a jobs *j* under the machine *i* is the ratio p_j / v_{ij}.

Flow Shop** (Fm)**: there are *m* machines in series. Each job *j* has to undergo an operation on each one of the machines. All the jobs will follow the same route.

Flexible Flow Shop** (FFc)**: a flexible flow shop is an extension of the previous problem where there are *c* stages in series where, at each stage, there are some machines in parallel. Each job has to be processed by just one machine at each stage. Again, all the jobs will follow the same route.

Job shop** (Jm)**: in this scenario there are *m* machines and *n* jobs. Each job could be processed by all the machines or by a subset of them. However, every job has its own route to follow.

***Flexible job shop** (FJc)*: this environment is a generalization of the previous one. There are *c* work centers, each one with some identical machines in parallel. Each job has its predetermined route and can be processed by each machine in each work center.

Open shop** (Om)**: there are *m* machines in the work center. Each job must be processed on each machine but there is no predetermined route to follow and, moreover, some processing time on some machine could be zero.

Regarding the β field, these are the possibilities:

Release dates**(r_j)**: if the fields is filled with this piece of data, it means that job *j* cannot start its processing before time r_j.

Preemptions** (prmp)**: with this processing feature, the scheduler is allowed to interrupt the work of a machine on a job and to start another one even if the previous job did not complete its processing.

Precedence Constraints** (prec)**: this entry can appear in the single machine or in a parallel machine scenario. In this case, the requirements are that one or more jobs must be completed before other jobs may be started processing.

Sequence dependent setup times** (s_{jk})**: the entry s_{jk} implies that between the processing of job *j* and job *k*, there must be an idle time where the machine must be re-organized to process another job. If the setup time is also dependent on the machine, the entry will include the subscript *i*, s_{ijk}.

Job families** (fmls)**: all the *n* jobs belong to *F* different families. Inside the same families, job may be processed one after another without setup time requirements.

***Batch Processing** (batch(b)):* there could be some machines able to process an entire batch of cardinality *b* simultaneously. In this case, the processing time of the entire batch is the longest processing time of the job inside the batch.

***Breakdowns** (brkdwn):* in this environment, it could be possible to have some unavailable machine at some time. In the deterministic scheduling, all the unavailability times are supposed to be known.

***Permutation** (prmu):* this entry in the β field requires that the permutation in which the jobs go through the flow shop is kept constant inside the system. This constraint forces to work under the FIFO (*First In First Out)* principle.

***Blocking** (block):* in a flow shop, there may be buffers in between two consecutive machines storing jobs waiting for the next processing step. If these buffers are considered to be limited, there may happen that one of them is full and the upstream machine is not able to release the job, causing the machine to be blocked.

***No-wait** (nwt):* the *no-wait* constraint is another flow shop requirement. In this scenario, jobs are not allowed to wait between two consecutive processing, implying that the starting time of a job must allow the job to be able to go through the flow shop without having to wait for any machine.

***Recirculation** (rcrc):* recirculation may happen in a job shop or in a flexible job shop where there is at least one job that has to undergo more than one operation with a machine or with a work center.

The field γ is related to the objective function to be minimized by the model. Usually, this objective is a function of the completion time of the jobs. Denoting with C_j the completion time of the job j, i.e., the time the job j exits from the systems, it is possible to define some functions regarding this piece of data.

***Lateness**:* the lateness of the job j can be defined as follows.

$$L_j = C_j - d_j$$

Lateness is greater than zero when the job is completed late and negative when it is completed before the due date.

Tardiness: the tardiness of the job j is

$$T_j = \max(C_j - d_j, 0) = \max(L_j, 0)$$

Since the tardiness of a job is always the max between lateness and 0, the tardiness cannot be negative.

Unit penalty: the unit penalty is computed as

$$U_j = \begin{cases} 1 \; if \; C_j > d_j \\ 0 \; otherwise \end{cases}$$

In the figure below, it is shown the behaviour of these function with respect to the completion time.

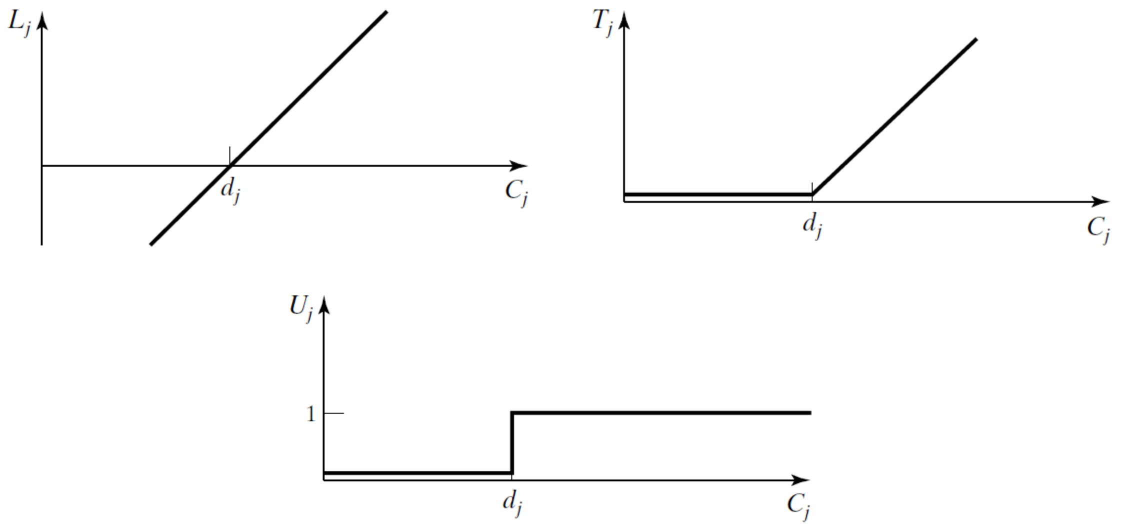


Figure 2.1: the behaviour of functions L_j, T_j and U_j.

Below, it is indicated a list of the most common objective function to be minimized in scheduling problems.

Makespan** (C_{max})**: This is the completion time of the last job.

$$C_{max} = max(C_1, C_2, \dots, C_n)$$

***Maximum lateness** (L_{max}):* it computes the biggest discrepancy between completion time and due date.

$$L_{max} = max(L_1, L_2, \dots., L_n)$$

Total weighted completion time: this metrics indicates the sum of the costs incurred by completing each job at time C_j, parameterized by the weights.

$$\sum_{j=1}^{n} w_j C_j$$

Total weighted tardiness: the same as before but taking into account the tardiness instead of completion time

$$\sum_{j=1}^{n} w_j T_j$$

Discounted total weighted completion time: this is a generalization of the previous metrics where the cost is discounted at a rate *r*.

$$\sum_{j=1}^{n} w_j(1 - e^{-r*C_j})$$

Weighted number of tardy jobs: this metrics it is used not only in the academic field but also in practice, since it is very easy to compute. It shows the sum of the number of late jobs parameterized by w_j.

$$\sum_{j=1}^{n} w_j U_j$$

2.2. Flow Shop

Often, in industrial production field, manufacturing and assembly phases are supposed to have series of operations to be executed one after another, creating a predetermined route for each batch that has to be processed. In this scenario, the machines are considered to be set up in series and the scheduler has to manage the flow of the jobs on which the machines have to work.

Sometimes, between two consecutive machine there could be an intermediate storage, like a simple buffer, where to stock the work in process. In case the products are physically small and easy to stock, this storage might be considered unlimited. Alternatively, if the products to be processed are physically large like televisions or printers, intermediate storage capacity may be an issue, causing the systems to block. Blocking is the situation where the intermediate storage between two machines is full and the upstream machine cannot release any job after its processing. This means the upstream machine has to wait to begin the processing of job if, at its completion time, the downstream buffer is already full.

When considering the flow shop with unlimited intermediate storage, the most common and general problem that comes out is the $Fm||C_{max}$. For this problem, the first question is whether it is enough to determine a permutation of the jobs that will remain constant as long as the jobs travel through the system. It is easy to picture a situation in which two jobs waiting for their processing under the same machine swap their position. In this case, the system will not operate under the First In First Out principle. Sometimes, allowing permutation change of jobs during the steps of the flow shop may lead to makespan minimization. It is obvious that, finding an optimal schedule when sequence changes are not allowed is much easier than finding it when changes are permitted.

However, it can be shown that there always exists an optimal schedule without job sequence changes between the first two machines and between the last two machines.

Theorem 1 (BAKER & TRIETSCH, 2009)***:*** when considering any regular performance measure in the flow shop scenario, it is enough to only look for schedules where the same job sequence occurs on the first two machines.

Proof: consider a schedule in which the sequences on the first two machines are different. Somewhere in the middle of this schedule there is a pair of jobs *i* and *j*, with the operation *(1, i)* preceding the adjacent operation *(1, j)* while the operation *(2, j)* preceding *(2, i)*. For this pair, it can be imposed on the first machine the same order of the jobs on the second machine (*j* before *i*), without affecting the performance measure. If we interchange operations *(1, i)* and *(1, j)*, then three situations could occur

- no operation is delayed, except the operation *(1, i)*,
- operation *(2, i)* is not delayed
- earlier processing of *(2, j)*, and other operation as well may result

Therefore, the operations swap should not increase the completion time of any operation on the second machine or on any following machine. Due to this, no job will have its completion time increased by the interchange, as well as no increase in any regular measure of performance would occur. The same argument applies to any schedule where job sequences differ on the first two machines, so the property must hold in general.

Theorem 2(BAKER & TRIETSCH, 2009)***:*** when considering the makespan of the flow shop model, it is enough to only look for schedules where the same job sequence occurs on the last two machines.

Proof: consider a schedule in which the sequences on the last two machines are different. Somewhere in the middle of this schedule, there should be a pair of jobs, *i* and *j*, with the operation *(m, j)* preceding the adjacent operation *(m, i)*, while the operation *(m-1, i)* preceding *(m-1, j)*. As a result of swapping operations *(m, i)* and *(m, j)*,

- no operation is delayed, except for *(m, i)*
- operation *(m, j)* ends before than *(m, i)* in the first schedule considered
- earlier processing of operations *(m, i)* and *(m, j)* may occur

Therefore, the interchange should not lead to an increase in the makespan of the schedule. Again, this argument applies to any schedule where job sequences differ on the last two machines. Therefore, the property must hold in general.

As a consequence of the previous two theorems, we should consider only permutation schedules in the following scenarios:

- Optimizing a regular measure of performance when *m=2*
- Optimizing makespan when *m=2* or *m=3*

2.3. Linear Programming

Most of the scheduling problems have been modelled and solved in literature by means of mathematical programming. Most of them belong to the category of linear programming and, specifically, to the field of combinatorial optimization. Before stepping into the problems, it is worthwhile to recap what a linear programming problem is and how they have been classified with respect to computational complexity.

A linear programming problem may be defined as the problem of *maximizing or minimizing a linear function subject to linear constraints*. These constrains could have the form of equalities or inequalities. There may be many variables and many constraints in the problem such that sometimes these problems are not so easy to solve. Moreover, some variables can be constrained to be nonnegative while others have to be treated as unconstrained. Furthermore, constraints may present at the same time inequalities and equalities.

Given a set of *n* variables subject to a set of *m* constraints. Here there is an example.

$$max \quad c_1x_1 + c_2x_2 + \ldots + c_nx_n$$

subject to:

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \geq b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$$

….

$$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m$$
$$x_1 \geq 0, x_2 \leq 0, \ldots, x_n \geq 0$$

Since everything is linear, it can be easily transformed to a linear product between vectors and matrices. Moreover, it is possible to add some variables to the problem and to modify a bit the objective function to reproduce it in a *standard form* that would enhance the notation as well as the generalization to all the LP models.

In the case of inequality constraints, slack variables can be added and surplus variables can be subtracted in order to create a set of equality constraints.

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \geq b_1$$
$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n - y_1 = b_1$$
$$y_1 \geq 0$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n + y_2 = b_2$$
$$y_2 \geq 0$$

In case of negative variables, it is possible to set:

$$x_2 = -y_2$$
$$y_2 \geq 0$$

Considering free variables, that are variables unrestricted in sign, they can be substituted with the difference between two positive variables.

$$x_s <\geq 0$$
$$x_s = u_i - v_i$$
$$u_i \geq 0, v_i \geq 0$$

Regarding the minimization or the maximization of the objective function, it is easy to observe that:

$$Max\big(f(x)\big) = -\,Min\big(f(x)\big)$$

With these changes, each LP model can be rearranged in the so-called *standard form.*

$$max\ \ c^T x$$

Subject to

$$Ax = b$$
$$x \geq 0$$

where:

- c is a n-dimensional vector
- x is a n-dimensional vector
- A is a m*n dimensional matrix
- B is a m-dimensional vector
- All the variables must be nonnegative.

Sometimes, in combinatorial optimization it is difficult to model problems with only the possibility to use continues variables. This is the case in which it is needed to introduce integers variables, i.e., variables that can take only integer values.

In this case, decision variables can be split in three categories:

1. Positive real variables $x_s \geq 0$
2. Positive integers variables $x_s \geq 0, integer$
3. Dummy variables $x_s \in \{0,1\}, integer$

2.4. Operational Complexity

After the introduction of linear programming models, it is possible to subdivide them from the point of view of the complexity related to the algorithms solving them. A problem can be seen as a general question to be solved, characterized by specific parameters and values.

An optimization problem *Π* requests to find the "optimal" solution that minimizes or maximizes the objective function *f(),* choosing it from a set of feasible solution.

$$\Pi : \min(f(x) \colon x \in X) \quad or \quad \Pi : \max(f(x) \colon x \in X)$$

An *instance* of a problem is a set of pieces of data specifying completely the values and the parameters of the problem. Furthermore, any optimization problem has its own decision version.

$$\Pi : min(f(x) : x \in X) \; ===> \; \Pi[D]: \; \exists \, x \in X : \; f(x) \leq K$$

The standard format of a decision version of an optimization problem is made by:

1. A specific instance of the problem assigning values to the problem parameters
2. A question related to the instances where the answer is just yes or no

Temporal complexity T_A of an algorithm A is usually expressed as a function of the dimension of the of the instance I. In order to have a complexity measure not dependent by the technology and by the speed of recent machines, it is convenient to express the computational effort as a number of elementary $N_A(I)$ used by A to solve I.

Temporal complexity of A can be defined as measure of the "worst case" between all the possible instances.

$$T_A(n) = sup\{N_A(I) : |I| = n\}$$

To evaluate the efficiency of A it is considered the asymptotic behaviour of $T_A(n)$ for $n \to \infty$. Using a notation similar to the one used in calculus for limits, it is said that:

$$T_A = O\big(g(n)\big) \quad if \quad \exists \, n', c : \; T_A(n) \leq c * g(n) \quad \forall n \geq n'$$

This means that T_A has a growth rate lower than g for $n \to \infty$.

An algorithm characterized by a computational complexity function *O(g(n))* where *g(n)* is a polynomial function of n, it is said to be a polynomial time algorithm. These problems are considered to be "easy" with respect to problems where the function *g(n)* is not polynomial.

It is now possible to introduce the classes of operational complexity (TADEI, DELLA CROCE, GROSSO, 2005).

- Class P: set of problems solvable in the worst case by means of a polynomial time algorithm
- Class NP: class of decision problems solvable in the worst case by means of a non-deterministic polynomial time algorithm

A remarkable difference between deterministic polynomial algorithm and non-deterministic polynomial ones is in the symmetry between answer "yes" and "no". In case of problem solvable with deterministic algorithms, if, for a specific instance, it is possible to determine that the solution Y is a truth-assignment, this means that the complementary problem is also solvable with a deterministic algorithm. This property does not hold in case of non-deterministic algorithms.

Thus, while the belonging of a problem to class P implies the belonging of its complementary version, the analogy for the class NP does not work. Moreover, every problem belonging to class P is also solvable with a non-deterministic polynomial algorithm. This means that between classes P and NP holds the relation $P \subseteq NP$. Even if it has not been proved, it is common sense that the relation is very likely to be $P \subset NP$. If not, all the problems belonging to NP could be resolved by means of polynomial algorithms. Instead, despite the amount of effort in research, there are polynomial algorithms for a lot of problems.

Thus, if classes P and NP do not coincide, it has sense to study the set of problem solved by non-deterministic polynomial algorithms but not solvable with polynomial ones. For this purpose, it can be helpful to introduce the concept of reducibility

Definition (TADEI, DELLA CROCE, GROSSO, 2005)***:*** A decisional problem Π' reduces to Π $(\Pi' \propto \Pi)$ if for any instance of Π' it is possible to construct in polynomial time an instance of Π such that from the solution of Π we can derive always in polynomial time the solution of Π.

From this definition it is possible to infer:

- Π' is a special case of Π
- Π is at least as hard to solve as Π'
- If $\Pi' \in$ P, then $\Pi' \in$ P

Now it is presented one of the most famous problem belonging to class NP. It is the so-called *Satisfiability problem* or *SAT*.

SAT: given a set of logical variables $U = \{x_1, x_2, \dots, x_n\}$ and a set of clauses $\{C_1, C_2, \dots, C_m\}$, where

$$C_j = (c_{1j} \vee c_{2j} \vee \dots \vee c_{kj}), \qquad c_{ij} = \{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\},$$

Does any assignment of values $x_1, x_2, \dots, x_n$ exist that makes true the following sentence?

$$C_1 \wedge C_2 \wedge \dots \wedge C_m$$

After that, it is now possible to introduce the class of NP-*complete* problems and the Cook's theorem.

Theorem 3 (TADEI, DELLA CROCE, GROSSO, 2005)***:*** Every problem belonging to NP class reduces to SAT

$\Pi \in NP \Rightarrow \Pi \propto$ SAT

This theorem says that *SAT* is the most difficult problem of the class NP. Moreover, it allows to introduce a completely new class of problems, the class NP-complete

Definition (TADEI, DELLA CROCE, GROSSO, 2005)***:*** the NP-complete class contains all the problems that satisfy these properties:

- $\Pi \in NP$
- SAT $\propto \Pi$

A new characteristic of this class is that all the problems belonging to it have the same level of difficulty. Every NP-complete problem reduces to another NP-complete problems. This means that if there existed a polynomial time algorithm for a NP-complete problem, this would solve in polynomial time all the NP-complete problems and the relation P=NP would hold.

Lastly, if for a problem Π is not possible to verify the membership to the class NP, but there exists a Π' belonging to NP-complete such that $\Pi' \propto \Pi$, then the problem would be at least as difficult as a NP-complete problem, but there is no guarantee of its belonging to NP problems. This creates another class of problems called NP-hard.

2.5. F2||C_{max} and Johnson rule's

Now, before introducing the well-known Johnson rule, the equations needed to compute the completion times of all jobs on all machines in a flow shop are presented. Given a schedule, the completion time of job *j* on machine *i* could be expressed as follows (PINEDO, 2008).

$$C_{i,1} = \sum_{k=1}^{i} p_{k,1} \qquad i = 1,\ldots\ldots,m$$

$$C_{1,j} = \sum_{k=1}^{j} p_{1,k} \qquad j = 1,\ldots\ldots,n$$

$$C_{i,j} = max(C_{i-1,j}, C_{i,j-1}) + p_{i,j} \qquad i = 2,\ldots\ldots,m \quad j = 2,\ldots\ldots,n$$

The first equation states that the completion time on machine *i* of job 1 is just the sum of all the processing time of job 1 from machine 1 up to machine *i*. This is quite obvious, since the first job of the sequence has no other job to wait for, then it can go through the system without any interruption.

The second equation expresses that the completion time on machine *1* of job *j* is the sum of the processing times on machine *1* of all the jobs from 1 up to *j*. Since the machine *1* has no waiting time and there is unlimited intermediate storage between machine *1* and *2*, the jobs can be processed one after another without any idle time on the first machine.

The last equation extends the computation of completion time on all the machine and for all the jobs. Clearly, the completion time on machine *i* of job *j* is the sum between the processing time of the job on that machine and the maximum between the completion time of the same sob on the previous machine and the completion time of the previous job on the same machine. It seems pretty intuitive that, before a job can start its processing has to wait for, whether the completion of the operation on the previous machine or the completion of the prior job on the same machine.

One of the first problem examined in literature was the *F2*||C_{max} problem. It is a flow shop scenario with two machines in series with unlimited storage in between. There are *n* jobs to be processed and its job has its processing time on both machines, p_{ij}. The purpose is to minimize the completion time of the last job of the system, i.e. the makespan.

The following rule is commonly referred as Johnson's rule and it is proved to be optimal for this specific problem.

It starts with the partition the whole set of jobs into two sets *S1* and *S2*. The first set should contain all the job with $p_{1j}<p_{2j}$, while in the second set should be put all the job with $p_{1j}>p_{2j}$. Jobs with $p_{1j}=p_{2j}$ can be stored in either set. The first set must be ordered in increasing order of processing times on machine *1* and must be processed first, this is known as shortest processing time rule (SPT). The second set have to follow the decreasing order of processing time on machine *2* and must be processed after, this rule is known as Longest processing time (LPT). This is usually referred as *SPT(1)-LPT(2).*

Theorem 4 (PINEDO, 2008)**:** any SPT(1)-LPT(2) schedule is optimal for F2||$C_{\max}$.

Proof: The proof is by contradiction. Suppose there is another optimal schedule. In such a schedule has to include a pair of adjacent jobs, a hypothetical job j followed by job k, that satisfies one of the following three conditions:

1. job j belongs to *S2* and job k to *S1*;
2. jobs j and k belong both to *S1* but $p_{1j} > p_{1k}$
3. jobs j and k belong both to *S2* but $p_{2j} < p_{2k}$

It is sufficient to prove that under any of these three conditions the makespan is reduced after a swap of jobs j and k. Assume that in the original schedule there is a job l preceding job j and a job m following job k. Let C_{ij} denote the completion time of job j on machine i in the original schedule and C'_{ij} denote the completion time of job j on machine i after the interchange. Swapping jobs j and k does not affect the starting time of job m on the first machine, as its starting time on this machine is equal to $C_{1l} + p_{1j} + p_{1k}$. However, it is interesting to compute when the second machine becomes available for processing job m. In the original schedule, this is the completion time of job k on machine 2, i.e., C_{2k}, and after the pairwise interchange this is the completion time of job j on machine 2, i.e., C_{2j}. in this case, it is enough to show that $C_{2j} \leq C_{2k}$ under any one of the three conditions described above.

The completion time of job k on the second machine in the original schedule is:

$$C_{2k} = max\big(max\big(C_{2l}, C_{1l} + p_{1j}\big) + p_{2j}, C_{1l} + p_{1j} + p_{1k}\big) + p_{2k}$$
$$= max\big(C_{2l} + p_{2k} + p_{2j}, C_{1l} + p_{1j} + p_{2j} + p_{2k}, C_{1l} + p_{1j} + p_{1k} + p_{2k}\big)$$

While, the completion time of job j on the second machine after the swap is:

$$C'_{2j} = max\big(C_{2l} + p_{2k} + p_{2j}, C_{1l} + p_{1k} + p_{2k} + p_{2j}, C_{1l} + p_{1k} + p_{1j} + p_{2j}\big)$$

Under the first condition, $p_{1j} > p_{2j}$ and $p_{1k} < p_{2k}$. Whit this in mind, the first terms within the max expressions of C_{2k} and C'_{2j} are identical. However, the second term in the last expression is smaller than the third term in the first expression while the third term in the last expression is smaller than the second term in the first expression.

So, under the first condition $C'_{2j} \leq C_{2k}$.

In the second condition $p_{1j}<p_{2j}$, $p_{1k}<p_{2k}$ and $p_{1j}>p_{1k}$. Thus, the second and the third term in the last expression are smaller than the second them in the first expression.

So, in the second condition $C'_{2j} \leq C_{2k}$ again.

The third condition can be proved in a similar manner as the second one.

Here comes the result, every *SPT(1)-LPT(2)* schedules are the only schedules that are optimal for the $F2||C_{max}$.

2.6. $F2||\sum C2$

This environment is the same as the previous one but, this time, the objective function to be minimized is the sum of the completion time on the second machine of each job. Just changing the objective function, the problem becomes NP-complete as proved by Garey et al. (1976). Due to *Theorem 1*, dealing with $F2||\sum C2$ is the same as dealing with $F2|prmu|\sum C2$. Thus, at least an optimal solution is known to be a permutation schedule, i.e. the sequence of jobs is the same on both machines.

The following is the MILP model associated to the problem. It is constructed by means of positional variables, that is, the model uses as variables the position of each job in the sequence. During the research times on this specific problem, it has been shown that this model works better with respect to the model with disjunctive constraints and variables.

$$min \sum_{j=1}^{n} C_{j2}$$

subject to

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j = 1, \ldots\ldots, n$$

$$\sum_{j=1}^{m} x_{ij} = 1 \qquad \forall i = 1, \ldots\ldots, n$$

$$C_{11} = \sum_{i=1}^{n} p_{1i} x_{i1}$$

$$C_{21} = C_{11} + \sum_{i=1}^{n} p_{2i} x_{i1}$$

$$C_{1j} = C_{1,j-1} + \sum_{i=1}^{n} p_{1i} x_{ij} \qquad \forall j = 2, \ldots\ldots, n$$

$$C_{2j} \geq C_{1,j} + \sum_{i=1}^{n} p_{2i} x_{ij} \qquad \forall j = 2, \ldots\ldots, n$$

$$C_{2j} \geq C_{2,j-1} + \sum_{i=1}^{n} p_{2i} x_{ij} \qquad \forall j = 2, \ldots\ldots, n$$

$$x_{ij} \in \{0,1\}, integer$$

Let C_{ki} be the completion time of job i on machine k, with $k=1,2$ and x_{ij} be binary variables with $i,j=\{1,....,n\}$. The variable x_{ij} is 1 if the i-th job is in position j of the sequence, 0 otherwise.

The first constraint states that, in each position j must be processed only one job while the second obliges the processing of each job. These two constraints ensure that each job belongs to just one position of the sequence and they are processed only once.

The third and the fourth constraints ensure the computation of the completion time of the first job on the first and on the second machine.

The fifth constraint makes sure that on the first machine all the jobs are pushed through the systems without waiting. Indeed, the completion time on machine 1 of each job from the second up to the last is just the sum of the previous job of the sequence and its processing time.

The second to last constraint guarantee that the completion time of each job on machine 2 is greater than the sum between the completion time of the same job on machine 1 and the processing time of the job on machine two.

The last constraints, instead, enforce the completion of the previous job on machine 2 before the job can start its processing on machine two.

These last two constraints together ensure that before each job start to be processed on machine two, both the processing on machine 1 of the same job and the processing of the previous job on machine 2 are completed.

As previously mentioned, this problem is NP-hard. This means that it is not known any polynomial time algorithm able to solve the problem. Moreover, there is no non-enumerative algorithm able to completely solve the problem up to the optimal solution. As such, for big instances of the problem, it becomes very time consuming to find an optimal solution.

In this case, heuristics methods are preferred. A heuristic algorithm is a rapid way to solve a specific problem looking for good solution, not ensuring optimality, but able to guarantee an acceptable computational effort. Since they are a problem dependent technique, they try to take full advantage of the particularities of the problem. However, since they are often too greedy, they usually get trapped in a local optimum. Moreover, heuristics do not provide theoretical bounds on the quality of the solution found.

As long as the research effort has been increased on this type of problems, new techniques and algorithms have been invented with the purpose of trying to solve these problems. One of those are meta-heuristics and math-heuristics.

Meta-heuristics are problem-independent techniques. As such, they do not take advantage of any specificity of the problem and, therefore, can be used as black boxes. In general, they are not greedy. In fact, they may even accept a temporary deterioration of the solution which allows them to explore more exhaustively the solution space and, thus, to get a better solution (that sometimes could coincide with the global optimum). Although a meta-heuristic is a problem-independent technique, it is nonetheless necessary to do some fine-tuning of its intrinsic parameters in order to adapt the technique to the problem at hand.

Matheuristics are optimization algorithms made by the hybridization of meta-heuristics and mathematical programming. They deal with the exploitation of mathematical programming techniques in a heuristic framework. This grants the robustness and constrained-computational time to mathematical programming approaches.

In 2014, Della Croce et al. devised a solution to the $F2||\sum C2$ that was able to outperform the current state of the art. Starting from a solution obtained by a Recovering Beam Search (DELLA CROCE ET AL., 2004) procedure, which is a constructive polynomial time heuristic that includes a recovering phase within the classic Beam Search approach, the two-machine total completion time flow shop problem has been solved improving the starting sequence under a re-optimization step. In this step, a commercial software was used to re-optimize only a specific window of the whole job sequence.

Let define *S'* as the starting sequence coming out from the RBS procedure and satisfying the model requirements. It is possible to define a neighbourhood *N(S,r,h)* by choosing a position *r* in the sequence and a size parameter h. With *S'(r,h)* it is referred to the sequence of jobs from position *r* up to position *r+h-1*. Authors called this sub-sequence the "job-window". The choice of the best solution inside the neighbourhood is simply accomplished by adding to the previous model this constraint:

$$x_{ij} = x'_{ij} \ \forall i \notin S(r,h), j \notin \{r, \dots, r + h - 1\}.$$

The constraint ensures that all job but those in the window are fixed in their position of the current solution, while the windows gets re-optimized. The resulting model, i.e. the window re-optimization model, is solved with an off-the-shelf MILP solver. Usually, commercial solvers are not suitable to solve big instances of difficult problems like this. However, for small instances, these solvers can easily find the optimal solution of the problem. Indeed, in the case of the re-optimization, the size of the problems has been lowered from n to 12. After the window re-optimization, a new *r* is picked randomly, and a new window re-optimization occurs. The procedure stops after all the *r* have been explored or the time limit has been reached.

The model is constructed in a way that even if the window re-optimization is done with only 12 jobs, the objective function to be minimized is still including all the job of the sequence. This means that for low values of the parameter *r*, the model tries to reduce the makespan of the window in order to gain a big improvement when computing the sum of all the completion time of the sequent jobs. Instead, for values of *r* near to *n-h+1*, the focus of the model is to minimize the sum of the completion time of the job of the sub-sequence.

The problems explored in the article had *n=100, 300* and *500.* The parameter *h*, i.e. the window size, was set to *12*. The processing times of the jobs were extracted randomly from a uniform distribution [1,100], as well as the parameter *r*. The following table shows the time parameters used, dependent on the problem size. The first column refers to the time limit of the whole algorithm, while the second columns specifies the time limit associated to the single window re-optimization.

Problem Size	Time Limit	Windows Time Limit
n=100	60s	10s
n=300	600s	60s
n=500	3600s	100s

As showed in the article, the procedure outperformed the current state of the art, showing improved efficiency and effectiveness with the respect to other two algorithms, one running an Iterated Local Search, while the other using a "population" of Simulated Annealing.

The choice of *h* equal to *12* was dictate by a mixture of experience and tests run by the authors of the articles. Moreover, the current choice of the position parameter *r* was done completely random, keeping trace of the previous ones already explored. Clearly, this randomness led the algorithms to do many re-optimizations without improving the current solution at all. Moreover, these attempts are time consuming and they could hurt the algorithm when there is a time limit to be respected.

The purpose of this work is to find a way to predict, before starting a window re-optimization, if optimizing the sub-sequence in that specific position *r* and with that specific size parameter *h* leads an improvement in the current solution or not. This will enable the algorithm to save time that could be used to more consecutive re-optimizations. To do so, Machine Learning techniques have been used. Before stepping into the discussion on how this task has been accomplished, it is worthwhile to recap what Machine Learning is and how it works. The next chapter has exactly this purpose.

3. Machine Learning

The human brain is the most powerful tool we have in our body. As the most important part and the most incredible organ, it manages every sound, smell, sight, taste and touch we perceive. It makes possible storing memories, feeling, emotions and experiences and let us dream. Without it, humans would be organism needed the only rudimental biological devices to sense things, such as food or danger.

Thinking about how far the technology has come, we are surrounded by many "smart" devices without a brain. However, they do contain a variety of sensors able to detect things from the environment as movement, lights and even chemicals. Most of the creatures without brain are provided by simple mechanisms able to detect everything necessary to survive.

In this parallelism between technology and nature, humans have always dreamed the possibility to create and build a machine able to think. From the invention of the first calculator by Alan Turing, up the present, research focus has been directed over the possibility to let machines be able to match human capability. However, for some specific tasks, computers have now reached and outperformed humans.

Since their creation, computers have been very powerful in completing tasks represented by a set of instructions. Thus, researchers have focused their effort in being able to codify tasks, making them "understandable" by a machine. In this way, in the recent past, computers have been explicitly programmed to execute specific activities where they were able to outperform humans.

One of the most famous case of a computer reaching human skills is Deep Blue. It was a chess-playing computer created by IBM that in 1996 challenged the world champion Garry Kasparov, playing at the same level of the Russian. Even if, around this event, a lot of complaints have been raised, it is still a very important step in the history of artificial intelligence.

However, it has been proved that computers are very strong when facing activities that can be easily codified in a set of instruction. In facts, some task that are very easy for a human, even for a child, becomes much more difficult for a machine and, on the other hand, some activities, easily done by a computer in microseconds, can last hours when done by human.

The infant brain weights around five hundred grams and it is able to easily solve problems that computer still find difficult to cope with. After some months, infants can recognize parents' faces, discern objects in the environment and even identify different voices. In around one year, they have developed intuition for basic physics, are able to track objects and can associate sound and meanings. In early childhood, they already manage a complex understanding on natural speak languages and grammar rules as well as a constant improvement of their vocabularies.

Most of the previous activities are very difficult and almost impossible when needed to be codified in a set of tasks. During the past decades, researchers and scientists have focused their attention in developing techniques allowing computers to learn from experience.

The next section will focus its attention on pointing out the differences in the fields of Artificial intelligence, Machine Learning and Deep Learning. Then it follows a focus on the specific subfield of Deep Learning. After that, the mechanism of learning inside a neural network is explained. Following, a focus on the supervised learning and on the most used architectures is provided.

3.1. Artificial Intelligence, Machine learning and Deep Learning

In the recent past, hyperinflated expectation have been raised around Artificial Intelligence. Humans have always dreamed about having machines and computers able to think and act like us; robots cleaning our homes, self-driving cars and videogames are just one of the most common ideas around the field of AI. Moreover, together with AI, two other abused words have been used: Machine Learning and Deep Learning. Before jump into the discussion, it may be fruitful to explain how they relate each other. After this, a rapid definition it is given.

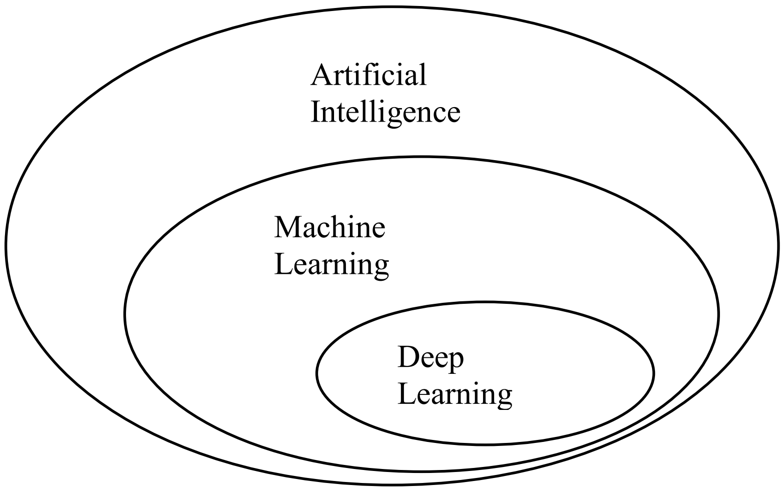


Figure 3.1: the connection between the fields of Artificial Intelligence, Machine Learning and Deep Learning.

Artificial Intelligence: *the effort to automate intellectual tasks normally performed by humans*. This field was born in 50's, when few computer scientists started asking if were possible to conceive a machine able to think. Due to the definition, it is the broadest area englobing both Machine Learning and Deep Learning.

Machine Learning: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E". In this definition by Tom Mitchell, there is the crucial point of all the discussion. Instead of focusing on the question, "can machine think?", Alan Turing replaced the question with "can machines do what we (as thinking entities) can do?". Changing the point of view of the problem makes possible to focus on let the computer learn patterns and rules by looking at the data. Moreover, this changes the classical paradigm of programming. In classical programming, a set of rules, i.e. the program, and data were given, in order to get the answers. In Machine Learning, data and answers are given both as input, to learn the rules behind. These rules can, then, be applied to completely new data to produce original answers.

A machine learning system is said to be trained instead of being explicitly programmed by humans. Features and examples are presented to the model which is able to extract statistical structures that could allow the system to get out with rules for automating the tasks.

It has started to flourish in 1990s and it has rapidly become the most prevalent, common and favourable subfield of AI. It usually deals with large amounts of data which other statistical techniques can struggle with. Thus, it exhibits little mathematical theory and it is engineering oriented. Moreover, most of the times, things are proved empirically instead of theoretically.

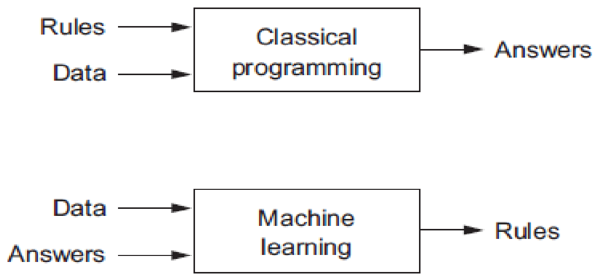


Figure 3.2: the different framework in classical programming and machine learning.

Machine learning is a vast field with a complex taxonomy around the subfields. Usually, three main categories subdivide this field and its algorithms.

- ***Supervised Learning:*** it consists of learning to map input to output. In this case, outputs are known targets, sometimes called labels or annotations. Most of the machine learning applications in the spotlight in the recent past, such as speech recognition, image classification and even language translation, belong to this subfield.
- ***Unsupervised Learning:*** this subfield consists in leaning and finding transformation of the input without any help coming from targets and labels. Usually, this subfield is often used in data analytics and *clustering* is one of its most well-known case.
- ***Reinforcement Learning:*** it consists of an *agent* receiving information from the environment and it has to learn how to act in order to maximize a sort of reward. Algorithms regarding self-driving cars and robotics belong to this category.

Deep learning: It is a specific field of Machine Learning. The word "deep" does not rifer to a deeper understanding of data. Instead, it takes its name by the structures of the models used. In Deep Learning, Neural Networks are used as a framework for learning. These networks are created by subsequent layers of neurons stacked on top of each other. The number of layers is also referred as *depth* of the Neural Network, from this, the word *deep* comes from. Even if the term Neural Network can remind to some biological features, there is no specific connection to that field. Also, these models do not have te purpose to describe the brain, neither there is evidence that human brain works in this way or implements something like the learning mechanism lying in the artificial neural networks.

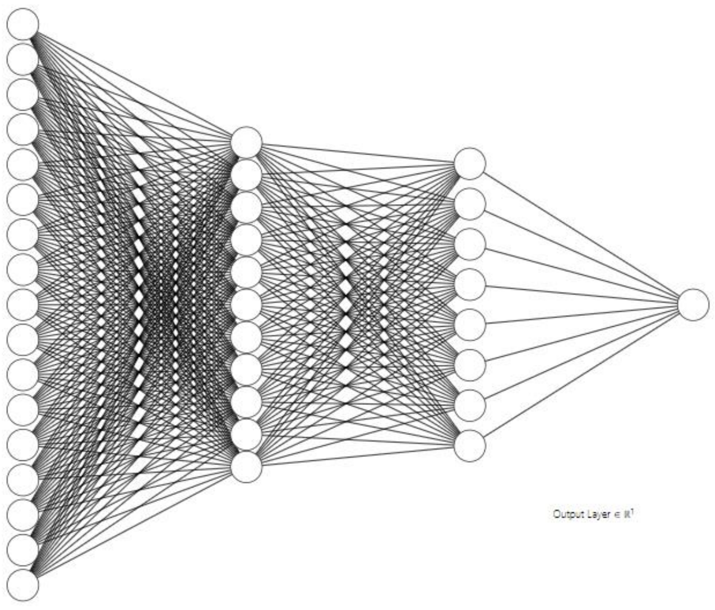


Figure 3.3: A simple example of an artificial neural network.

3.2. Deep Learning

Even if Deep Learning is an old subfield of Machine Learning, it only become famous around 2000's. From its rise up the present, it has achieved so far near-human level in some tasks that brought him fame. Remarkable results have been achieved even in speech recognition and computer vision, problems that seems pretty simple to human but have always been elusive for machines. Up to now, this are the best result Deep Learning scientist got with these techniques:

- Near-human level image classification
- Near human level speech recognition
- Near-human level handwrite transcription
- High level of machine translation
- High level of text-to-speech conversion
- Digital assistant like Google Now and Amazon Alexa
- Near-human level autonomous driving
- High level of ads targeting on the web
- High level of search results
- Ability to answer human natural language-question
- Superhuman Go playing

Despite these great achievements, scientist and researchers still do not know how much power have these techniques and how far they can push them to the limits.

As an example, the key idea behind computer vision, Convolutional Neural Networks, goes back to 1989. Moreover, the LSTM (Long-Short Term Memory) algorithm, for timeseries and sequences, were already been found and understood in 1989. However, they only become crucial to innovation in AI in 2010s. Why did they take too much time to be used and to be beneficial to the researchers? There are three force shaping the progress and the evolution of Deep learning: Hardware, Data and Algorithms.

Regarding hardware, in 20 years, off-the-shelf CPUs have become faster by a factor of approximately 5000. As a result, it is now possible to run some Neural Networks on each laptop we find in store. Moreover, with the increased effort made by company like NVIDIA and AMD in producing better GPUs to power realistic videogames, now typical Deep Learning models are starting to be trained on the graphics units, since they have become very powerful.

When entering the field of Data, it is clear that, nowadays, the ability to get information from every situation is at its best spot. These pieces of data are the fuel of every Neural Network. Over the past 20 years, the progress in data storage, as predicted by Moore's law, together with the incredible spread of the internet, have created an enormous quantity of data that have powered Neural Networks from the very beginning.

When it comes to Algorithms, until the late 2000's, Deep Learning was not so powerful due to the lack of a reliable way to train Neural Networks. The biggest problem was the inability of the gradient propagation over many layers of Neural Networks. Around 2010, the extension to new and better activation functions, weight-initialization and optimization schemes made it viable.

3.3. How Neural Networks work

Before stepping into the deep discussion of layers and architectures in Neural Networks, it is worthwhile to show an example on how Neural Networks work and what is the mechanism behind the learning.

It is given in image as an input and the Neural Network has to learn how to classify this image and to recognize which number is inside the image. This is a classification problem and it belongs to the category of supervised learning. It is called supervised because the neural network is provided also with the answer, i.e. the target, that it has to learn. Also, the word classification come from the action of classifying the images as belonging to a specific category. Here there is a figure to let it clear.

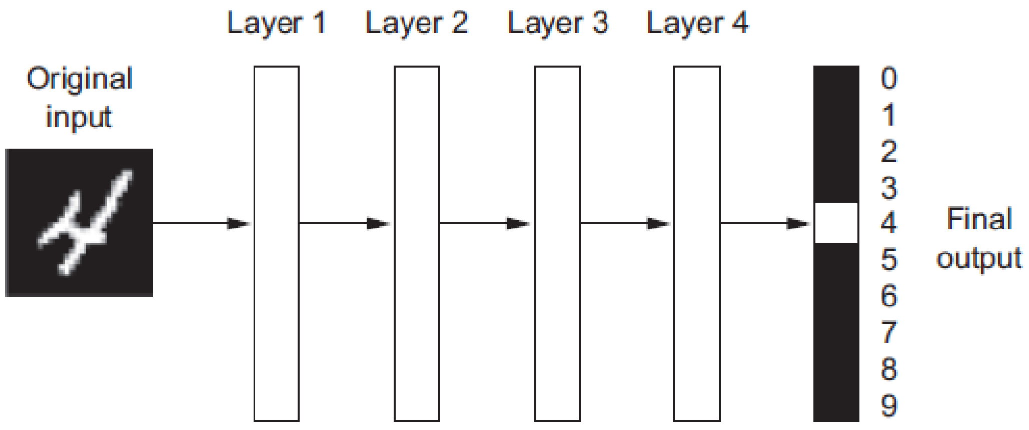


Figure 3.4: An example of neural network classifying human digits.

This a clean representation of the Neural Network. The input is the image with the number four. The image has to be processed inside an architecture of four layers. Note that the there is no relation between the number inside the image and the depth of the Neural Network. The structure would be the same even if the number inside the image were nine. Inside the Neural Network, the image is broken up into other different representations and some important parts are highlighted by each layer. In this multi-stage representation, the image is purified to be understood by the machine and to be classified. The next image points out how each layer re-represents the image iteratively. The output in this case, can be sees as a ten-element column vector with all zero and one just in the row associated with the number four, which is actually the fifth one due to the starting from zero.

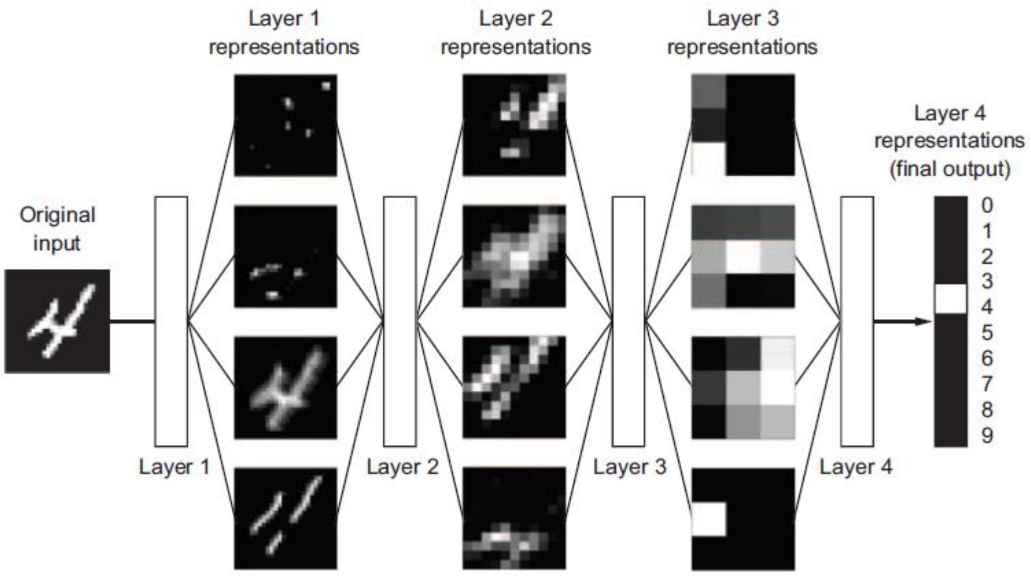


Figure 3.5: The specific representation of how a layer processes the input in the case of image classification.

These images help the visual representation of the learning mechanism. However, to better understand how it works from the analytically point of view, the next image will be self-explanatory.

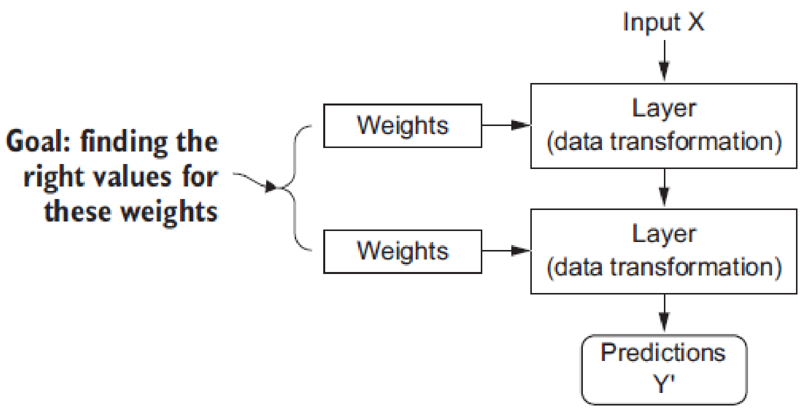

Input X
Layer (data transformation)
Layer (data transformation)
Weights
Weights
Goal: finding the right values for these weights
Predictions Y'

Figure 3.6: The neural networks in a nutshell.

The network takes an input in the first layer and associate it with weights. The first layer transforms the data computing one or more layers-specific functions and pass the output to the next layer that do the same. In the image there are only two layers, so the output of the second layer is already the output of the neural network, i.e. the prediction.

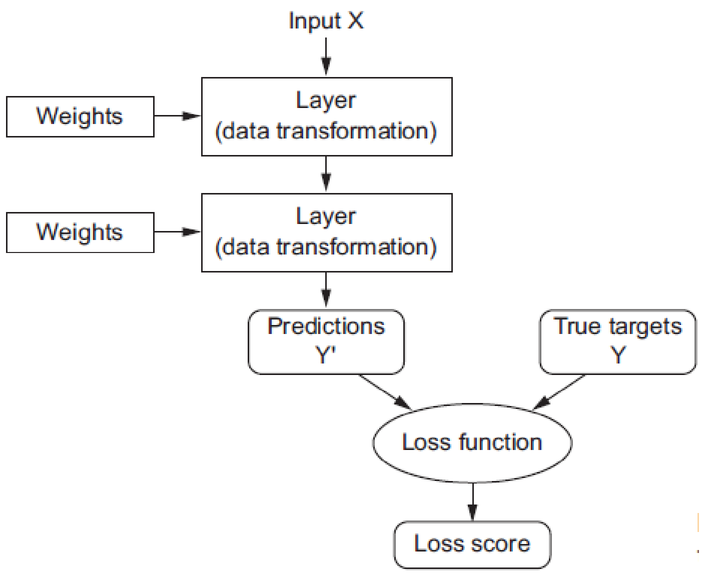


Figure 3.7: The learning framework inside a neural network.

After that, the prediction coming out from the Neural Network are compared with the so-called ground truth, i.e. the true target that we want to predict. A loss function is computed to be able to quantify how much "distance" there is between the prediction and the real target. Associated to the loss function there is also a loss score, that helps to understand the ability of the network to iteratively learn to predict.

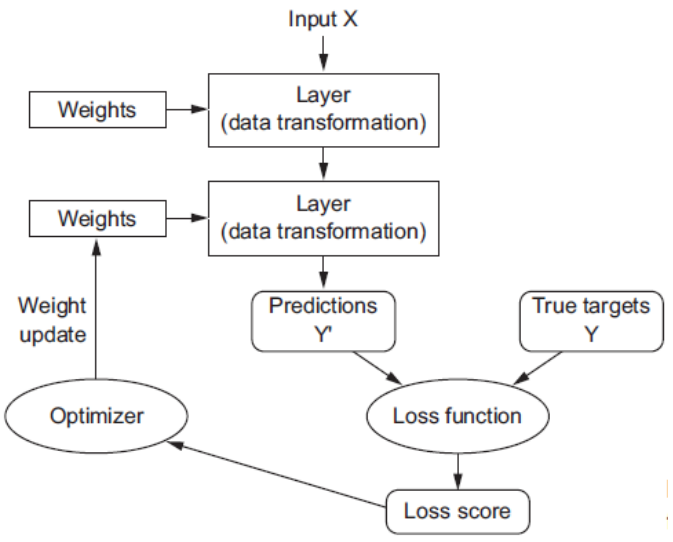


Figure 3.8: The learning process inside a neural network.

After the loss score is computed, an optimizer steps into to minimize this loss score. Minimizing the loss score, using weights as variables, permits the Network to find weights that can close the gap between prediction and the real targets. Iteratively applying this procedure, the loss score decreases and the accuracy for predictions increase. The procedure stops when an acceptable low level of loss score, or, accordingly, a reliable high level of accuracy is reached.

3.4. Inside Artificial Neural Networks

The previous example has been shown in order to clarify the learning mechanism of a Neural Network. However, it was quite general and there were no explanations regarding what happens inside a layer, how to compute loss function and so on. It is worthwhile to mention, step by step, how a neural network is constructed and modelled, together with all the features that will be used in next sections. Moreover, it will be analysed the case of a supervised learning, pointing out the differences between regression and classification problem.

3.4.1. Supervised Learning

The first part of approaching a neural network is to decide what to learn and how let the network to learn. Usually, most of the problem are treated via supervised learning. As already said, in supervised learning, the neural network is provided not only with the input but also with the output, i.e. the target, it has to learn. This is typically the case of regression and classification problem.

In regression problem, the neural network is asked to predict a numerical value given some input. In order to solve this task, it has to output a function *f*:

$$f: \mathbb{R}^n \to \mathbb{R}$$

Instead, in classification problem, the network has to specify which of k category some input belongs to. In order to execute this task, the program has to output a function g:

$$g: \mathbb{R}^n \to \{1,2,\ldots,k\}$$

3.4.2. Datasets

The key part of the learning mechanism are the pieces of data the neural network has to learn from. Following the T. Mitchell quote, the performance measure has to improve with the Experience. As such, without experience, there are no way to learn. Usually, the word dataset is used to indicate the data given to the model. Correctly managing datasets is crucial when dealing with the model ability of performing well not only during the training but also with new data. Usually, an entire dataset is split into three parts:

- ***Train Dataset:*** It is the dataset used directly for training. On this dataset, weights are updated in order to achieve the prediction over the targets. Usually, the learning process go through this dataset analysing fixed-sized batch of data. Every whole training session on the training dataset is called epoch.
- ***Validation Dataset:*** on this dataset, the learning process does not occur. This means that weights will never be updated using this dataset. However, this dataset is still useful because it helps to validate the model on pieces of data unseen during the learning process.
- ***Test Dataset:*** After the learning process has been completed, the test dataset comes into play. Its objective is to test the model on completely new and unseen data. Even if the validation dataset is not used to the weights update, it is still used every epoch to compute measures of accuracy and to select the final model. Instead, test dataset, is directly used to the prediction purpose and no learning mechanism occurs on it.

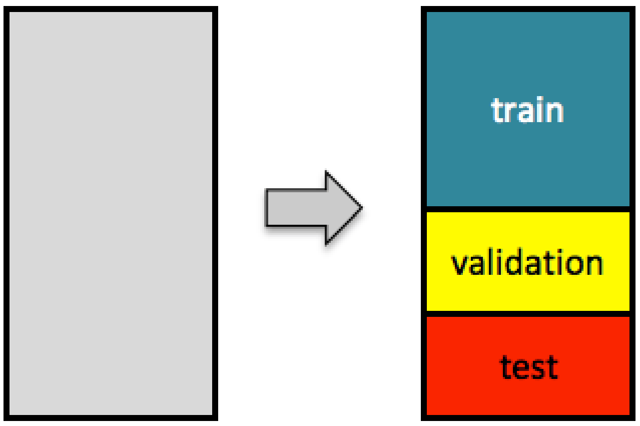


Figure 3.9: Dataset splitting.

3.4.3. Data pre-processing and feature-engineering

Before the model development, there an important issue coming into the question. How to prepare the input data to be elaborated by the neural network. Feature engineering is the process of applying transformation to the input data such that the algorithm can work better on them.

Data pre-processing, instead, makes the raw data more manageable by the model. Usually, there are some domain specific data pre-processing as for text and image data. However, there is a shared path of data for each kind of machine learning model. The very first step is the *vectorization*. Every neural network library is coded in such a way that it takes as input vector, matrixes and tensors. Then, input data must be normalized. Usually, it is not safe to feed neural network with data that takes relatively large values, or instead, data that comes from different heterogenous distribution. To make the learning part easier, usually the input data should have small values and be homogenous. The best way to accomplish this task is to normalize them subtracting the mean and dividing by the standard deviation. However, if some features cannot be properly described by the normal distribution, another way is to pre-process them by dividing by the maximum value.

Feature-engineering, instead, requires understanding the problem in depth. It is a way to represent the problem in a simpler way. This will help the learning mechanism. An example, as always, is better than thousands of words. Imagine that the purpose of a model is to learn to read the clock. Passing raw images of different clocks can be a way to approach the problem. However, a better input data could be the cartesian coordinates of the clock hands. This will reduce the cardinality of the input data and makes it easier to understand for a neural network. Yet, this is not the best way possible. Passing to the network the angles of the clock hands will be even better for the model.

Data pre-processing and features engineering are mandatory parts of building a neural network and cannot be avoided. The former implies just some lines of code and a good mathematical library, while the latter requires a good knowledge of the problem to be faced. Both of them requires a clearly of schooling and experience, as the entire deep learning field.

3.4.4. Activation functions

Activation functions are the most important part of neurons. They basically decide whether a neuron must be activated or not. Moreover, the output of neurons and layers, strongly depends on the kind of activation function. During the growth of deep learning model, some activation functions have shown better results compared to others. The following ones are the most used by machine learning scientist.

Linear: the output of the neurons is just the scalar product of weights and input. Actually, no function is applied at all. This is explicitly used in the last layer of regression problem, since the output will not be bounded.

Sigmoid: this is a smooth and continuously differentiable function. It outputs values between 0 and 1 and has a S-shape. Since it is not linear, the gradient is very high in the middle, around 0. This means that, specifically in the range of -3 and +3, a change in x would bring a large change in the value of y. Thus, this function tries to push the values to the extremes. This is a desirable property when it comes to binary classification.

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

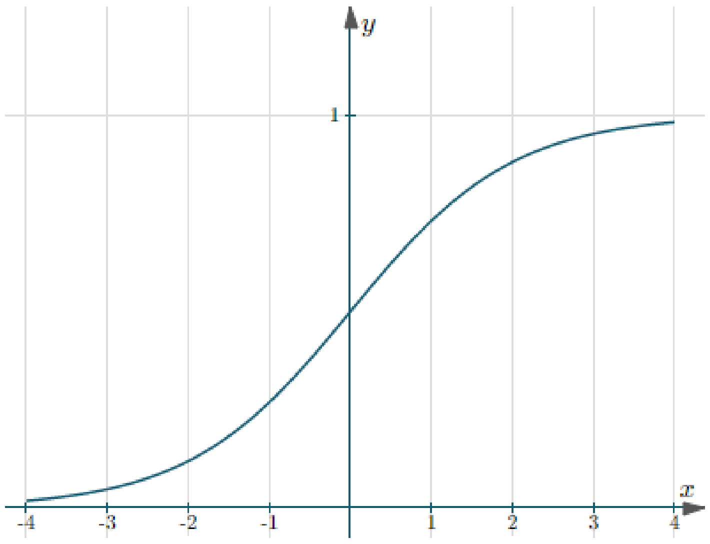


Figure 3.10: The graph of the sigmoid.

Tanh: this function is very similar to the sigmoid function. It outputs values between -1 and 1 and it is zero-centered. Actually, it is a scaled sigmoid and they are related by the following formula.

$$tanh(z) = \frac{2}{1 + e^{-2z}} = 2\sigma(2z) - 1$$

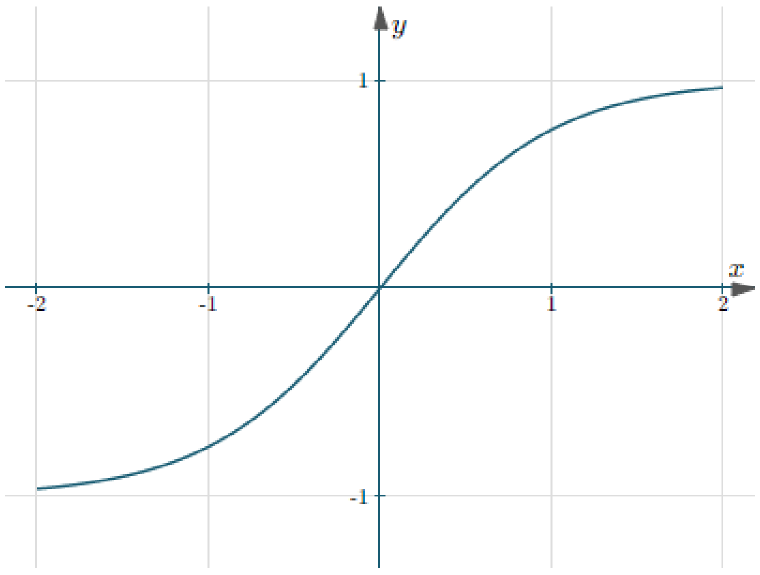


Figure 3.11: The graph of the hyperbolic tangent.

Rectified Linear Unit (ReLU): A different kind of linearity is used with this function. This is a hockey-stick-shaped function, able to output only positive values. Since it can output also zero values, sometimes, when the network has too many neurons in a layer, this function can help to let some outputs to be zero, such that the layer becomes sparse.

$$R(z) = max(0, z)$$

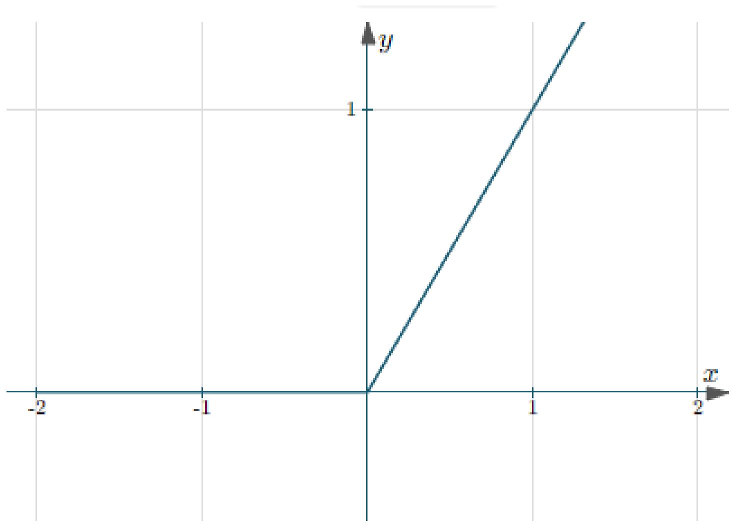


Figure 3.12: The graph of the Rectified Linear Unit function.

Leaky ReLU: sometimes it comes the need to avoid having sparse layers. In these cases, the best function to be used is a slightly modified ReLU. In the following formula, ε is intended to be a positive small number.

$$LR(x) = max(\varepsilon * x, x)$$

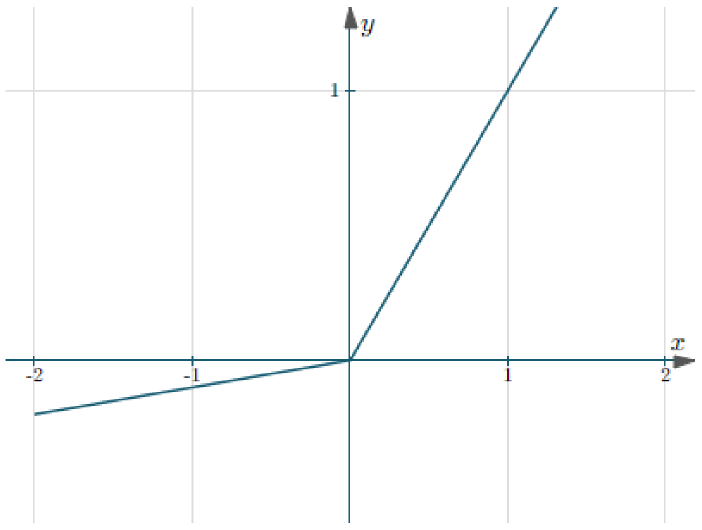


Figure 3.12: The graph of the Leaky ReLU.

Softmax: oftentimes, the purpose of a neural network is to classify input over multiple categories. Thus, single output from neurons is not satisfying. In this case, the network has to output a vector describing the probability distribution over a set of mutually exclusive targets. Moreover, it is required that the output of all neurons in the layer is affected by outputs of every other neurons in the same layer. This function requires that the sum of all the output in an entire layer is one. In this way, the output of a single neuron can be seen as a probability distribution telling how likely the belonging to that specific category is.

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

3.4.5. Layers

A layer is the data-processing unit of a neural network. It takes as input pieces of data in the shape of tensors and output already processed tensors. There are many types of layers, and each one is specifically used for a precise task. Usually, simple two-dimensional elements as vector and matrices are processed by *fully connected layer*, also called d*ensely connected*. Instead, tensors are typically processed with Recurrent Layers such as Long-Short Term Memory (LSTM) and Gated Recurrent Unit (GRU). Moreover, Image data, stored in four-dimensional input tensors, are processed by a specific two-dimensional layer, called *convolutional*. Since in the next chapter will be used only the Recurrent and Dense layers, the following discussion will focus only on these two.

Dense Layer: This is the easiest layer that can be used in a neural network. It takes as input a two-dimensional tensor and output another two-dimensional tensor. The next figure helps to figure out what a dense layer is and what it does.

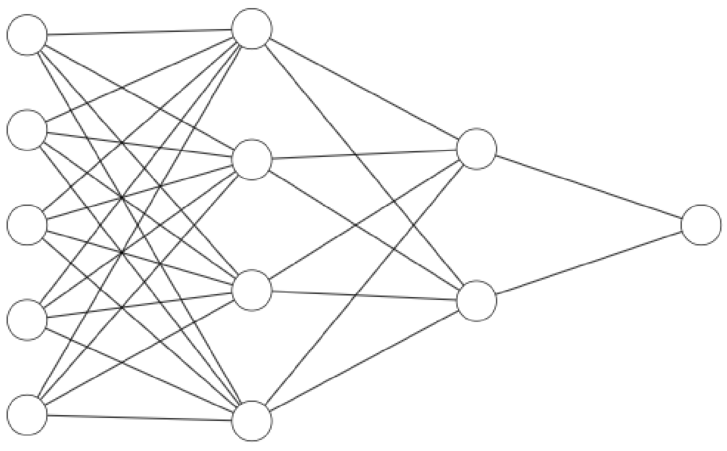


Figure 3.13: A neural network composed by only Dense layers.

A layer is an entire column of neurons. The first layer is called input layer and it is not a proper layer. Whereas, it is just a common way to draw neural network, this let the viewer understand how many inputs it has. The first layer of the network is the first Hidden Layer and has five neurons inside. Every neuron computes the following function:

$$f(w * x + b)$$

where:

- f is the activation function of the neuron
- w are the weights of the layer
- x is the input of the layer
- b is the bias term

However, this structure is too easy to be used to solve some specific problems where inputs belong to the three or four-dimensional tensor space. Moreover, every neuron does the same thing, without any link between neurons in a layer. However, sometimes input relies on strong connections between them and neural network must catch these links. Recurrent Neural Network can address this issue.

So far, the focus has been primarily on sets of data points that were assumed to be independent and identically distributed (i.i.d.). For many applications, however, this assumption will be a poor one. Here, it is considered a particularly important class of such data sets, namely those that describe sequential data. These often arise through measurement of time series, for example the rainfall measurements on successive days at a particular location, or the daily values of a currency exchange rate, or the acoustic features at successive time frames used for speech recognition. In this case, the following type of layers are proved to perform better than the previous one.

Recurrent Layer: these are networks with loop inside, allowing information to persist in the flow. Usually, a recurrent layer looks like something strange, but it can be seen as a set of multiple copies of the same network. The next image can clarify this concept.

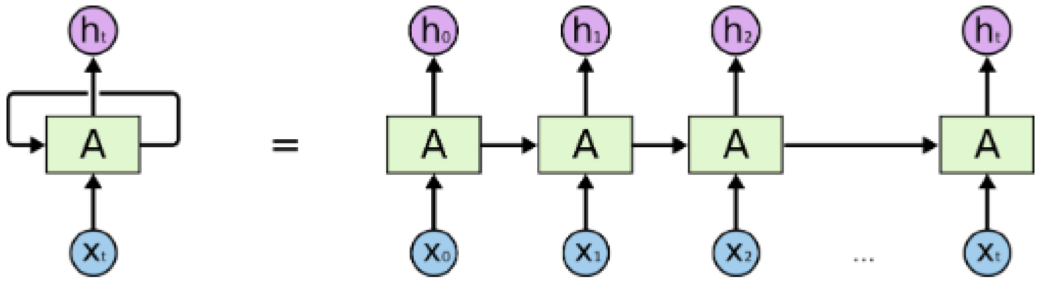


Figure 3.14: An unrolled recurrent neural network.

In this figure, there are both representation of a neural network. In the first one it is in a compact notation, while in the second one it is unrolled in multiple copies. From this figure, it is easy to understand why this network is so useful when facing sequences and timeseries. In the recent past, it has been applied to a huge variety of problems like language modelling, speech recognition translation and even image captioning. Essential to this great success is Hochreiter and Schmidhuber (1997), where Long-Short Term Memory cells have been introduced. Every RNNs have the shape of a chain with repeating cells of the network. Simple RNNs present just a single tanh module like the one in the figure below.

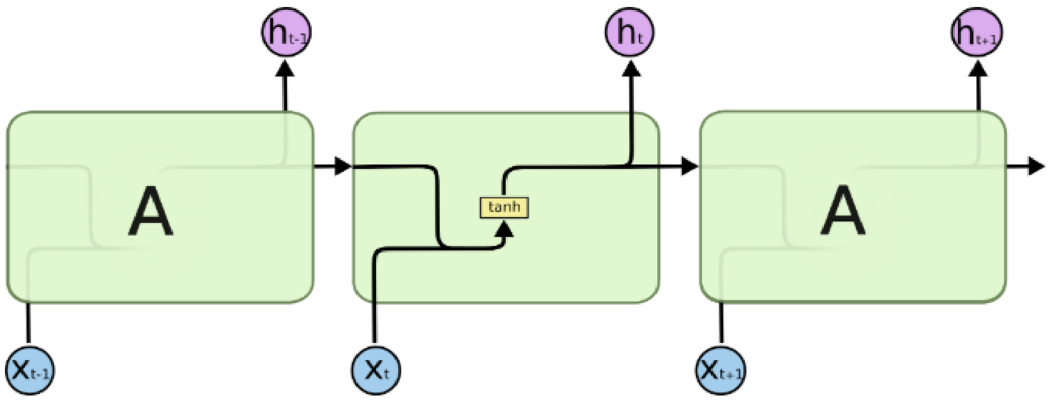


Figure 3.15: The repeating module in a standard RNN contains a single layer.

One of the appeals of recurrent neural networks (RNNs) is that they could connect previous information, stored in previous cells, to the present task. This actually works when the distance between crucial cells is not so big. An example can easily clarify the point.

Sharks live in the **sea**

I speak a perfect **Italian**

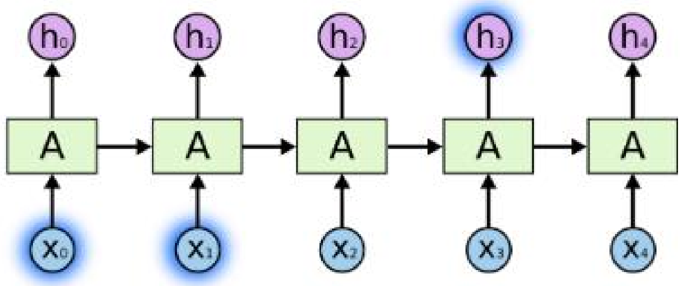


Figure 3.16: Short-term dependencies in a standard RNN.

Imagine a neural network trying to guess the last word in a sentence. In the first sentence, it is pretty intuitive that the word is going to be sea and there is no need for any further linkage with previous words outside the ones in the sentence. However, when it comes to the second example, the only thing we can get from the sentence is that it needs a name of a language. However, we do not know how to guess which language it is if we do not extract other pieces of information from the context.

I grew up in Italy, … … . I speak a perfect **Italian**

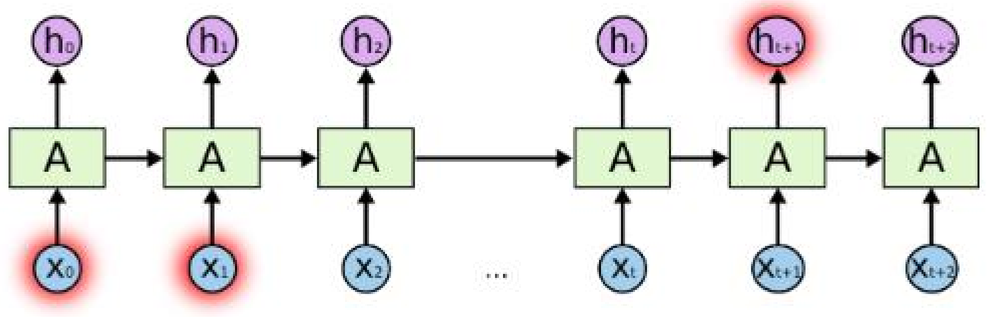


Figure 3.17: Long-term dependencies in a standard RNN.

Unfortunately, as the gap grows, RNNs becomes unable to learn this long-term dependency. In theory, these networks should not present these issues. However, as explored by some scientist, in practical application the problem exists.

Long-Short Term Memory: LSTMs have been explicitly programmed and designed to avoid the long-term dependency problem of simple RNNs. Their default behaviour is to remember information for long periods of time. Each cell has four interacting layers inside, like in the figure below.

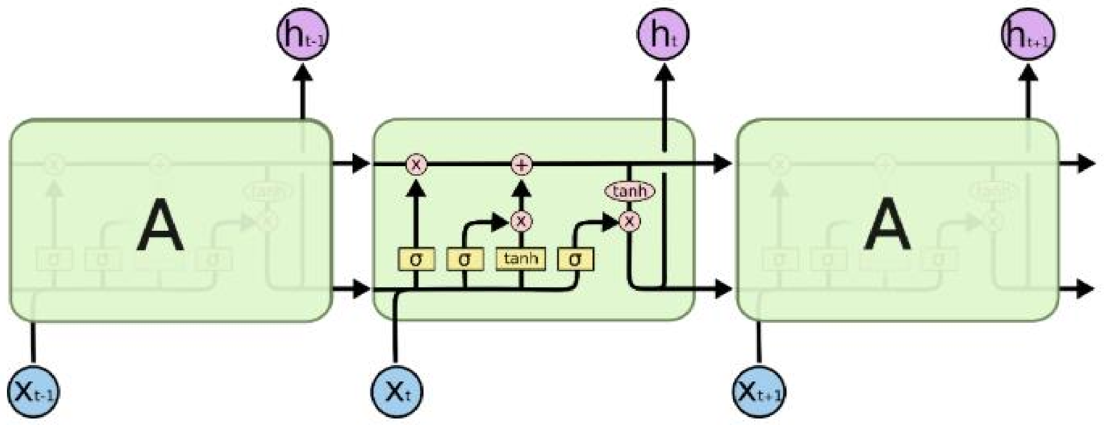


Figure 3.18: The repeating module in a LSTM contains four interacting layers.

The key idea behind LSTMs is the horizontal line on the top of the module. Here is where information flows inside the cell. Each cell has the ability to structure the information, adding or removing pieces, thanks to entity called gates. They are composed by a sigmoid function and a pointwise operation. Since the sigmoid output values between zero and one, it tells how much of information should let be go further. Actually, in LSTMs there are three gates, one with the purpose of protecting the state and another controlling it.

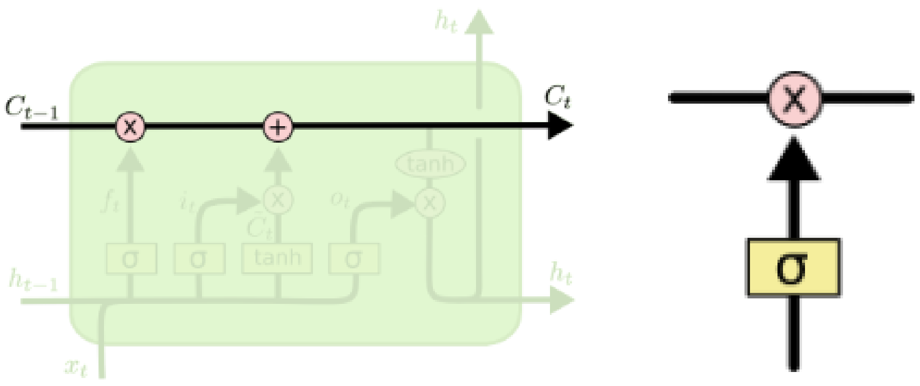


Figure 3.18: The Information flow inside a LSTM cell and the sigmoid operation on it.

Following, there is a step-by-step analysis of what happens inside a LSTM cell, with all the equations controlling the information flow. The first step is to decide what information has to be removed. This is accomplished via a “forget gate layer”. This layer looks at h_{t-1} and x_t and compute the sigmoid, outputting a number between 0 and 1 for each number in the cell state C_{t-1}. The number one means “store everything” while the number 0 stands for “completely forget”.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

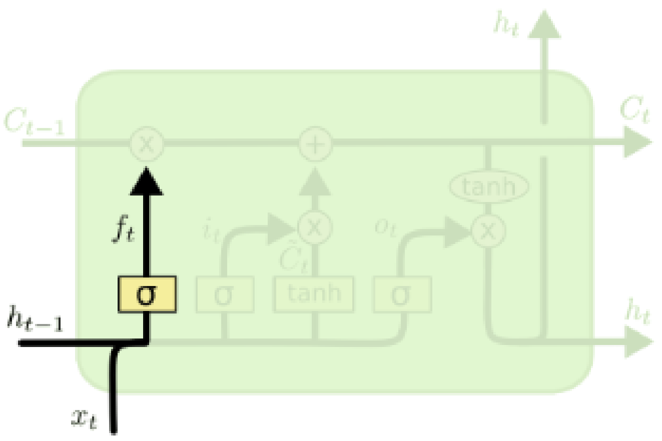


Figure 3.19: The forget gate layer in a LSTM cell.

The next step consists in deciding what new information must be stored in the cell state. This has two section. First, another sigmoid layer decides which values it will be updated. This layer is called “input gate layer”. Then, a tanh layer creates a vector of candidates that could be added to the state. The figure below will show the mechanism and the equations.

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$
$$C'_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$$

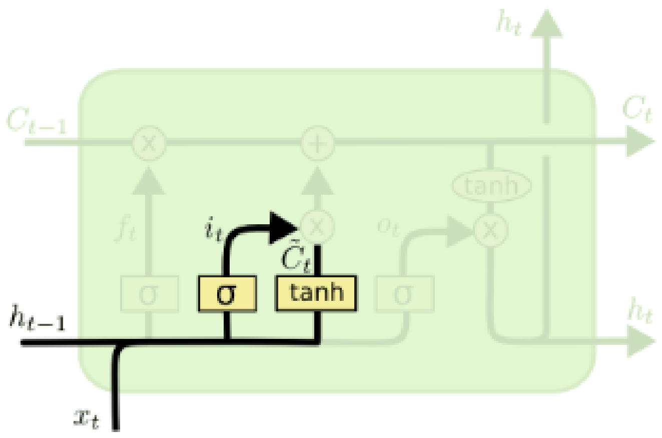


Figure 3.20: The input gate layer in a LSTM cell.

After that, there is the update of the previous cell state Ct-1 with the new one Ct. Inside the cell, there is the multiplication of the old state by ft and we add the multiplication of the previous two terms already computed.

$$C_t = f_t * C_{t-1} + C'_t * i_t$$

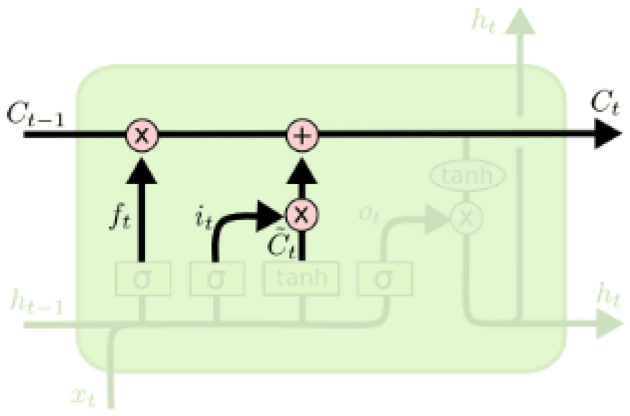


Figure 3.21: The update gate layer in a LSTM cell.

Lastly, the cell has to decide what to output based on the cell state. As a first operation, there is a computation of the sigmoid which choose what parts of the module state it is going to be output. Then, the cell state is computed through a tanh function and it is multiplied with the output of the previous sigmoid operation. In this way, the cell will output only the part it has chosen to.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

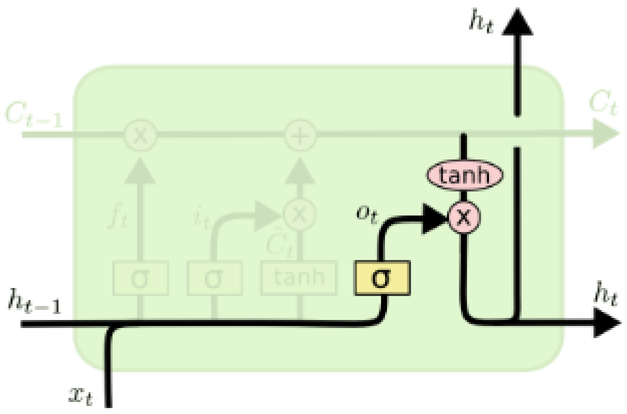


Figure 3.22: The final output computation in a LSTM cell.

Bidirectional RNN: sometimes there is need to learn representation not only from previous time steps but also from future one. This helps eliminating ambiguity and understanding better the entire contest. The cells could present the same characteristic as above and they can be LSTM, GRU or simple RNN. However, they present connection not only pointing forward but also oriented backward. From that, the name bidirectional come.

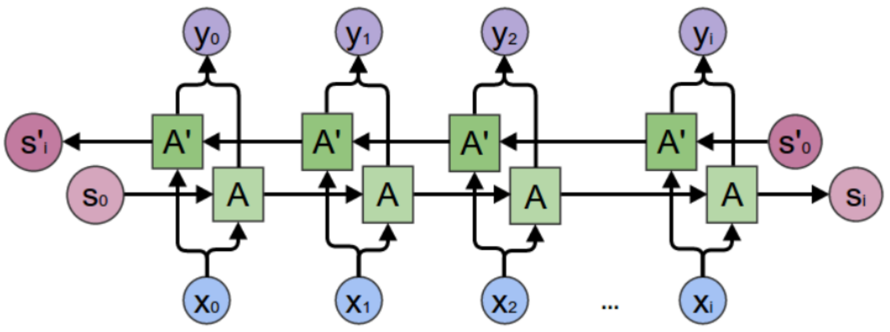


Figure 3.23: Classical bidirectional RNN.

3.4.6. Loss Functions

Once the network architecture has been defined and every layer is ready to execute its task, it is time to define the loss function, also called objective function, to be minimized during the training. It defines the measure of success achieved by the task. Furthermore, an optimizer must be chosen. It determines how the network will be updated based on the loss function. Every optimizer uses a variant of the stochastic gradient descent (SGD). A neural network can have multiple output and also multiple loss function. However, since the gradient descent process is based on a scalar loss value, it is mandatory that all the loss values are combined via averaging or other techniques into a single scalar quantity. Moreover, choosing the right objective function to be minimized is extremely important when it comes to achieve better performances. The network will be brutally trying to minimize the loss function. This means that, if the loss function is not well correlated with the success of the task, the network may come out doing unwanted things. Fortunately, when facing classical problem, the most used loss function are the following ones:

Problem Type	Loss Function
Binary classification	Binary crossentropy
Multiclass, single label classification	Categorical crossentropy
Multiclass, multi label classification	Binary crossentropy
Regression to arbitrary values	Mean Squared Error (MSE)
Regression to values between 0 and 1	MSE or Binary crossentropy

Binary Crossentropy: As can be understood from the formula below, the function gets bigger when the ground truth and predicted value are at the opposite values.

$$-\frac{1}{N}\sum_{i=1}^{N} y_i * log(y_i') + (1 - y_i)log(1 - y_i')$$

Where:

- y_i is the ground truth
- y'_i is the predicted value
- N is the number of elements in the dataset

Categorical crossentropy: the formula is pretty similar to the previous despite the fact that it takes into account more categories for the classification problem.

$$-\sum_{i=1}^{N}\sum_{j=1}^{m} y_{i,j} * log\left(y_{i,j}'\right)$$

Where:

- $y_{i,j}$ is the ground truth
- $y'_{i,j}$ is the predicted value
- m is the number of classes
- N is the number of elements in the dataset

Mean Squared Error (MSE): This is the classical formula of the mean squared error applied to the ground truths and the predictions

$$\frac{1}{N}\sum_{i=1}^{N}(y_i' - y_i)^2$$

Where:

- y_i is the ground truth
- y'_i is the predicted value
- N is the number of elements in the dataset

Mean Absolute Error (MAE): Sometimes, instead of MSE, the mean absolute error is used. The error will not be squared, but instead, all the absolute distance of predictions and ground truths are summed up.

$$\frac{1}{N}\sum_{i=1}^{N}|y_i' - y_i|$$

Where:

- y_i is the ground truth
- y'_i is the predicted value
- N is the number of elements in the dataset

3.4.7. Optimizers

Now, the neural network is ready, everything has been chosen and the training is almost starting. However, there is a missing part in the mechanism: the optimizers to be used. Currently, the most popular algorithms are mini-batch gradient descent, mini-batch gradient descent with momentum, Adam, AdaDelta, RMSprop and RMSprop with momentum. While it would be interesting to know how these algorithm works and when to use the correct algorithm, there is very little consensus among experts. The most important point, around optimizers, is that, for most deep learning scientist, the focus to push the cutting edge of neural networks is not on building more advanced optimization algorithms. Instead, the vast majority of finding in this field over the past decades have been obtained by looking at the architecture that are easy to train. With this in mind, it is more useful to focus also this work on the neural network architecture instead of focusing on optimizers formulas.

3.4.8. Overfitting and Underfitting

Usually, the first time a neural network is trained, it will not accomplish its work properly. Most of the times, this is a long task requiring time and experience. The most encountered problems when training artificial neural network are underfitting and overfitting. The first is the inability of the network to reach an acceptable level of accuracy on both training dataset and validation dataset. However, overfitting is the most encountered problem by machine learning scientists. It occurs when the model fails to generalize on new unseen data. This means that the accuracy of the prediction on the training dataset are very high while it performs poorly on the validation dataset. Usually, when training a machine learning model, *optimization* refers to the process of iteratively adjusting the weights to get the best performance measure, whereas *generalization* refers on how well the trained neural network performs on data never seen before.

Clearly, the goal is to achieve the best generalization possible, but the focus is only on the training part. So, the best machine learning scientist can do is to prevent overfitting in order to let the model generalize well on completely new data. This process is called *regularization*. This is a hard task and sometimes could require much more time than the training itself. The following images can help understanding the overfitting issue.

Image the purpose of the neural network is to learn the curve that can divide blue spots from red ones. As it can be seen, the black curve makes some mistakes but is still a good trade between optimization and generalization. However, the green curve is not generalizing, but it is actually *memorizing* the specific distribution of this specific sample. This means that when coming to apply the curve new unseen data, it will not be able to reach a good level of accuracy because of the biased learning.

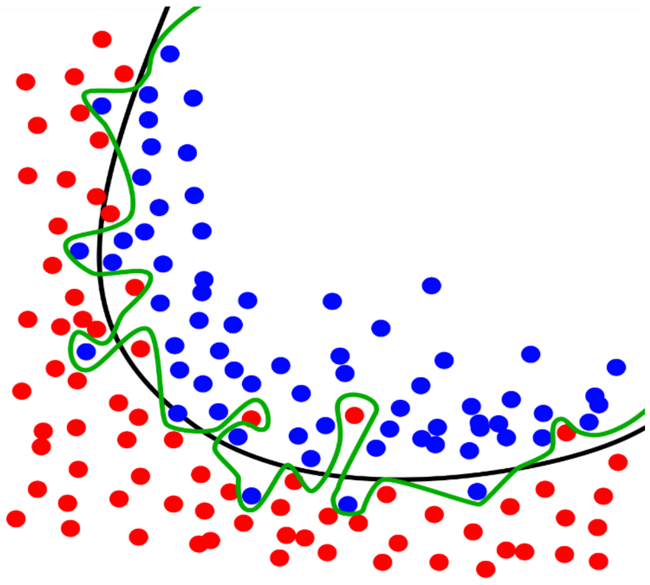


Figure 3.24: The difference between generalization and overfitting.

3.4.9. Regularization

When dealing with artificial neural networks, scientists and practitioners agree on a widely known principle, the so-called ***Occam's razor***. It states that among competing hypothesis explaining different observations equally well, one should choose the "simplest" one. With simplest one, it is referred the hypothesis that makes fewer assumption. This principle applies also in when dealing with neural networks. Given a dataset and an architecture, multiple sets of weights, i.e. multiple models, could explain the pattern inside the data. With this principle in mind, it should be chosen the easiest one. Simpler models are less likely to overfit then complex ones due to its lack of ***capacity***. A model's capacity is the ability to fit a wide variety of functions. Usually, the model capacity is monitored by the number of learning parameters. Intuitively, models with low capacity may struggle to learn data distribution, while, models with more parameters have more memorization capacity. Most of the time spent by deep learning practitioners regards this part of the work, i.e. finding the right balance between "too much capacity" and "not enough capacity".

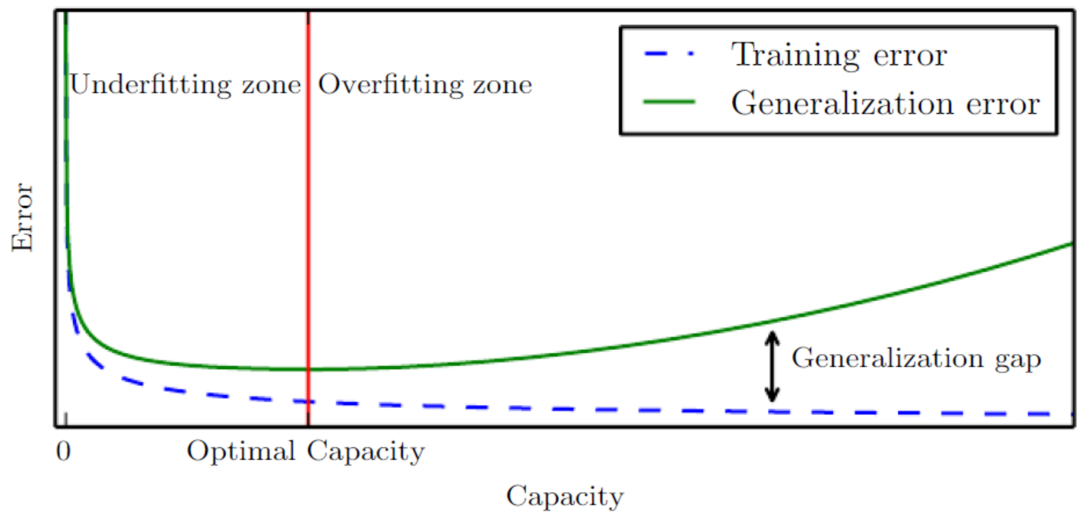


Figure 3.25: The trend of the error measure in a neural network depending on the model capacity.

In the figure above, it is shown how the error in a deep learning model behaves when modifying capacity in a model. It sounds clear that adding too much capacity to the model leads to overfitting, while removing capacity, does not help the learning task and may result in underfitting.

There are several techniques coming into help of deep learning experts when facing overfitting. One of the most used method is the *weight regularization*. It modifies the objective function of the network to be minimized by adding a cost associated with having large weights. Moreover, constraining the weights to be small makes their distribution more regular. There are two form of weight regularization:

- ***L1 regularization:*** The cost added to the loss function is proportional to the absolute value of weights coefficient, i.e. the so-called L1 norm of the weights

$$f(x) + \lambda \sum |w_i|$$

- ***L2 regularization:*** the cost function added to the loss function is proportional to the square of weight coefficient, i.e. the so called L2 norm of the weights. Sometimes, this type of regularization is also called weight decay in deep learning. However, despite the different word used, both refer to the same practice.

$$f(x) + \lambda \sum w_i^2$$

The parameter λ is chosen by the model creator. It is common practice to used negative powers of ten as 0.01 or 0.001. Exceeding some practical limit values will let the model underfit because of too much regularization.

Another used technique to prevent overfit is the ***Dropout.*** Dropout is applied to an entire layer and consists of randomly dropping out some output features. It is like randomly removing some links between layers. This procedure was developed by Geoff Hinton. He was inspired by a fraud-prevent bank mechanism. In his own words, "I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot". He was able to figure out that to successfully fraud a bank, it would require cooperation between employees. Thus, removing a subset and moving them around would prevent any tentative of cooperation. He thought that applying the same system to neural network would prevent “conspiracies between neurons and would prevent overfitting.

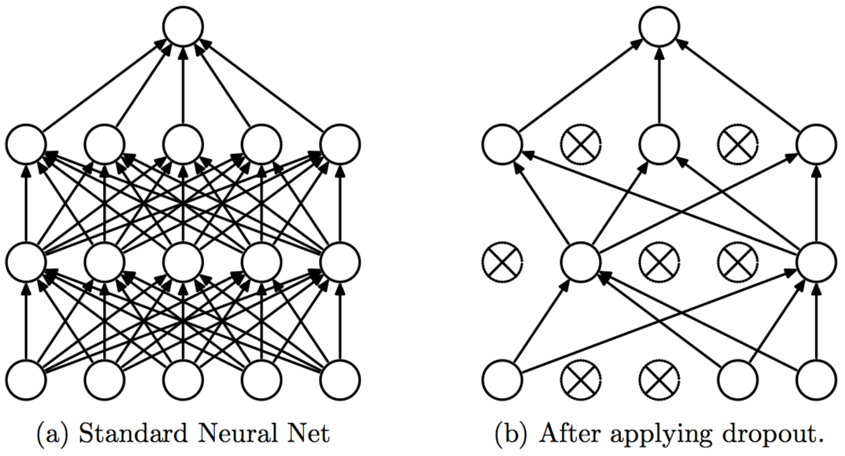


Figure 3.26: Example of a neural network with and without using dropout.

3.5. Hyperparameters Tuning

So far, all the parameters encountered have been described as static and seems they do not vary too much. However, weights in the learning process are continuously modified in order to reach an acceptable level in the measure of success chosen. After the completion of all the step described above, the last part comes into play. It is the tuning of the ***Hyperparameters***. Usually, in deep learning jargon, with the word parameters are indicated the weights inside the neural network. However, some other parameters are present in the construction of a neural network and, most of the times, they are even more important. Number of neurons in the layers, batch size, number of epochs, learning rate are just ones of the hyperparameters that have to be tuned. Clearly, the first configuration comes from experience, gut feeling and some precedent works on the same field. However, despite not reproducing an already working configuration, all new configuration will need a tune period to reach the level of accuracy it is needed. This procedure is a sequence of trial and error, aiming to reach the perfect structure that work on both training and validation dataset.

4. A completely new approach

So far it has been introduced the environment this work lies inside. In chapter one, a brief introduction to the scheduling problems and a satisfactory explanation of the two-machine flow shop problem has been presented. Moreover, two different problems, with different objective function have been dealt with. The approach to these two problems, even if they come from the same family, is completely different and leads to different result. The Johnson rule is a simple rule to apply and lead to optimal sequence, while the matheuristic in Della Croce et al. (2014) is a satisfying approach outperforming the current state of the art. However, since the matheuristic presented does not ensure the optimal solution, there should be room for enhancement.

The focus of this entire work is to find a way to improve the matheuristic using machine learning techniques presented in chapter two. In facts, the focal point is to replace the randomness of the parameters with a more consistent choice based on the learning mechanism of an artificial neural network. Since this approach has no previous attempts, there no certainty on its success or its failure. However, most of machine learning techniques are starting to be applied in many varied fields of study and, in the future, this could be one of this.

The next sections focus on the procedures applied during the research work and follow the same path. Firstly, an entire section is dedicated to the problem formulation for the learning part. Then, there is a section on how the datasets have been built. After that, the model is constructed, followed by the tuning of the hyperparameters and the final training. All the results are presented in the following chapter in order to reserve a complete part of this work to them.

4.1. Problem formulation

Before stepping into the creation of the network architecture as well as the shaping of the dataset to learn from, the purpose of the neural network has to be clear from the very beginning. The objective of the work is to create a predictor that could step into the matheuristic and replace the randomness of the r parameter, i.e. the one which establish the position, as well as the parameter h, deciding the size of the window. Actually, the h parameter is not random, but it was chosen using a mixture of trials and experience by the authors of the paper. However, since the focus of this work is on the improvement of the whole procedure, it will take into account both parameters. A future matheuristic should incorporate a predictor that could help choosing these parameters before the window re-optimization. This will avoid re-optimizations not leading to a real decreasing of the objective function.

The problem can be formulated in two ways, each one has pros and cons. One way to model it is as a regression problem. In this way, given a sequence coming out from the first RBS step or from consecutive re-optimizations, the predictor has to pick which are the parameters r and h leading to best improvement possible with that specific sequence.

$$f(S) \rightarrow \mathbb{R}$$

Where f is the prediction function, S is the input sequence and the output is a couple of scalars. Clearly, the pro in this procedure, is that, as long as it re-optimizes, and it changes the starting sequence, it can still be applied and focus on the best improvement possible. However, when the predictor has no best option, or it makes a mistake, the sequence does not change and re-using the predictor on the same sequence will lead to the same parameters and the matheuristic has to stop.

Instead, defining it as a binary classification problem, will mean that the predictor will output whether or not, a given sequence, together with windows size and position, will lead to an improvement.

$$f(S) \rightarrow \{0,1\}$$

Where, f is the prediction function, S is the input sequence and the output is binary variables. In this way, the improvement will not be the best possible as in the previous case. But, if the predictor output a "zero improvement" for a given sequence, it is still possible changing the parameters r and h in order to find a window to be re-optimized. Even if the two procedures seem pretty distant one from another, they are both very similar from the coding point of view. At this point, there is no insight on which one is better. So, since both decisions could lead to a good result, it has been decided to go on with the next step: the creation of the datasets.

4.2. Dataset creation

One of the crucial issues in the deep learning field is to provide the neural network with a solid dataset to learn from. As already explained in the previous chapter, feature engineering and data pre-processing are ones of the most important part in the model preparation. However, while data pre-processing focus more on the preparation of the data as input. Feature engineering deal with the issue on how to arrange and construct a dataset with useful information. To accomplish both tasks, the focus should be on the preparation of a coherent dataset, easy to manage and handling only useful pieces of data.

The focus of the learning is how the window size and the position affect the re-optimization. One point to consider is that as long as, the sequence is re-optimized, the system moves closer to a local minimum. This is an important feature of the problem to take into account. Moreover, when the sequence is reoptimized in the early position of the sequence, since the focus is on the whole system, sometimes minimizing the makespan of the subsequence let all the completion times of next jobs to shift, leading to a better solution than the one where the solver would have focused on minimizing the sum of completion time on the second machine of the subsequence. However, this characteristic is inside the problem itself, it is difficult to translate to a piece of data and it is something that the improvements in the consecutive re-optimization bring with themselves.

Furthermore, since the focus is on both parameters, the window size has to vary in a range, as well as the parameter r. To create a dataset with this purpose, there should be tried all the different couples of *(r, h)*. One way to accomplish this task is to take as input the solution coming from the RBS as in Della Croce et al. (2014) and to consecutively re-optimize the sequence with different position and windows size. However, the first issue is related to which sequence has to be re-optimized. In the paper, since the purpose is to reach the best minimum, the sequence that is always re-optimized is the best until that point. But, if in the creation of the dataset the same procedure was applied, the dataset would be unbiased. Because the re-optimizations done at the early stage are clearly more efficient since the sequence is far away to the local minimum. So, the sequence to be iteratively re-optimized has to be always the same for an entire combination of values *(r, h)*.

After that, other issues come into play. In the first stage of the re-optimization, it seems pretty intuitive that most of the couples *(r, h)* will lead to an improvement. However, after an entire re-optimization, should the procedure stop or not? If not, which sequence should be taken as the one to be re-optimized for a new entire combination of values? Since the future matheuristic should include more steps of re-optimization, also the dataset should provide to the neural network more re-optimization on the same instance. To solve these problems, the dataset should be constructed doing more re-optimization with a best-improvement strategy. This means that the sequence to be re-optimized in the next stage is always the one with the minimum objective function, i.e. the one with the greatest improvement among all the ones belonging to the same stage. In facts, starting from the RBS sequence, the dataset has been built re-optimizing an entire combination of parameters *(r, h)* for six stages. At the end of each stage, the sequence with the best improvement has been picked as a starting sequence for the next stage. Moreover, during the re-optimization stages, if an entire couple of values *(r, h)* does not lead to any improvement at all, the procedure stops to avoid wastes of time.

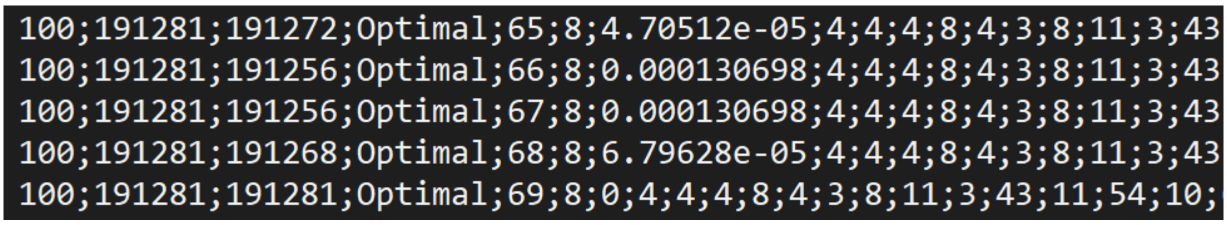
Lastly, it should be defined the range of the parameter *h*. This choice brings with itself the decision around the time limit to be applied. Since this parameter establish how many variables the sub-problem will have, increasing this value will let the problem grow in size, and thus in difficulty. Applying the same fixed time limit also for values of *h* greater than twelve, could result in not giving enough time to the solver to work on bigger sub-problems. Thus, the time limit has been increased to deal with this issue. Regarding the values for the size parameter, it has been decided to let it move from a minimum value of eight, up to the value of twenty.

When all the characteristics of the datasets have been decided, the problem has been coded using C++ language together with the same solver used in the matheuristic. Since the code of the matheuristic have been provided by the authors of the paper, a new code has been made using the same framework of the original one. The variables have been declared in the same way and all the essential parts have been left the same in order to keep the problem as equal as possible. Here a step-by-step procedure is presented to explain the way the algorithm of the dataset creation works.

```
• x = heuristic solution coming from RBS
• x' = starting solution
• x' = x
• for 6 times :
      o Re-optimize x' for all couple (r,h)
      o x* is the first best solution
      o x* = x'
      o if no improvement for all (r,h)
            ▪ break
```

The first problem size to be evaluated has been the one with one hundred jobs. The time limit for the window re-optimization has been set to twenty seconds for all the value of *h*. Thus, a dataset of twenty-five instances of this problem have been built. The first twenty instances have been used for the training part, as training and validation dataset. The remaining five instances have been utilized as the test dataset, at the end of the procedure. When creating the datasets, a time issue has raised. To completely create the first twenty instances of the problem, it has taken five entire days of computing. During this phase has been recognized that, constructing also the datasets for the three hundred jobs problem as well as the five hundred jobs problem it would have taken around months of computing. Due to the lack of machines and time, as well as not knowing if the technique would have led to a success, it has been decided that it would be better to focus only on the one hundred jobs problems as a starting evaluating phase.

The dataset constructed in this phase comes out in a .csv file. It brings in each row the number of jobs, the objective function value for the starting sequence, the objective function value for the re-optimize sequence, a string with the problem status from the solver, the position parameter *r*, the size parameter *h*, the improvement and a sequence of processing time on machine one, processing time on machine two, completion time on machine one and completion time on machine two for all the one hundred jobs. Here there is an example of some rows of the dataset.



```
100;191281;191272;Optimal;65;8;4.70512e-05;4;4;4;8;4;3;8;11;3;43
100;191281;191256;Optimal;66;8;0.000130698;4;4;4;8;4;3;8;11;3;43
100;191281;191256;Optimal;67;8;0.000130698;4;4;4;8;4;3;8;11;3;43
100;191281;191268;Optimal;68;8;6.79628e-05;4;4;4;8;4;3;8;11;3;43
100;191281;191281;Optimal;69;8;0;4;4;4;8;4;3;8;11;3;43;11;54;10;
```

Figure 4.1: An excerpt of the dataset.

The first row shows that with a re-optimization done in position 65 with a window size of 8, the solution decreased from 191281 to 191272 with an improvement $4.70512 * 10^{-5}$. Then, there is the sequence of all the processing times on the two machines as well as the completion times of the all jobs. In the second row, another re-optimization on the same starting sequence as been applied in a different position with the same windows size. The improvement is computed as follows:

$$\frac{f(S) - f(S')}{f(S)}$$

Where S is the starting sequence, S' is the sequence after of the re-optimization, and *f()* is the objective function computing the sum of the completion time on machine two of all the jobs. . As it can be seen, moving along very few positions, the re-optimization can lead to different improvements as well as there is no improvement at all in the las row.

Before stepping into the modelling of the neural network and the coding phase of this, it is worthwhile to mention the reasons behind the choice of using Python as a coding language, as well as the usage of Keras as a supporting library for the specific model building.

4.3. Why Python?

In the recent past, with the evolution and the diffusion of large amount of data as well as the software and automated services explosion, programming seems to be one of the most requested skill. Even in fields where there were no trace of programming and data manipulation in the past, nowadays there are large requests of jobs regarding the field of data analysis. With the amount of data growing in size, doing it by hand would need thousands of coordinated employees. Moreover, low level of data manipulation can be accomplished via the usage of off-the-shelf spreadsheets. However, when it comes to add some new and more interesting features to this analysis, programming languages seems to be more effective and less time consuming. As an example, one of the most well-known spreadsheets can "only" deal with 1,048,576 rows and 16,384 columns. Instead, when using programming languages, this amount is virtually unlimited. As already said, with the amount of data getting larger and larger, limits imposed by a spreadsheet could become an issue.

Furthermore, the growing in power of CPUs and computers, followed by their diffusion to the vast majority of the population, gives the possibility to deal with large amount of data even at home or with low-end products. The computer used to write this work has a CPU that can easily reach 3.00 GHz. This means that it can do around three billion of elementary operations in one second. This strongly helped the growing and the diffusion of high-level programming languages. These languages have a strong abstraction from the details of the computer and, in contrast to low-level programming languages, they usually are easier to use. Most of the times, they may use natural language elements and may entirely hide significant areas of computing systems, making the process of developing a program simpler and more understandable with respect to lower-level languages.

One of the most well-known web-site featuring questions and answer around computer programming, makes a graph on the number of questions regarding the different programming languages. In this way, it can measure the trend of usage of these languages. At the end of 2017, a data scientist on the web-site blog, made a research on these trends, showing the incredible growth of Python in high-income countries chosen by the World Bank. The graph shows the percentage of overall questions made in each month from 2012 to 2017 together with a growth prediction up to 2020, showing that Python will strongly overtake other programming languages. When it comes to non-high-income countries, the trend is a bit different regarding the absolute values of percentage of question. However, the graph still shows a great growth of Python with respect to the other languages.

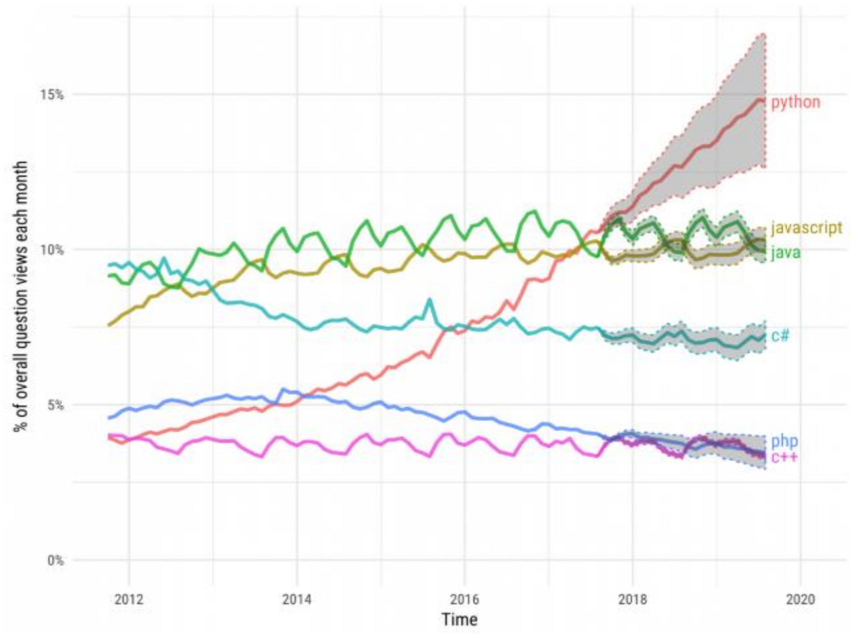


Figure 4.2: High-income countries growth of Python.

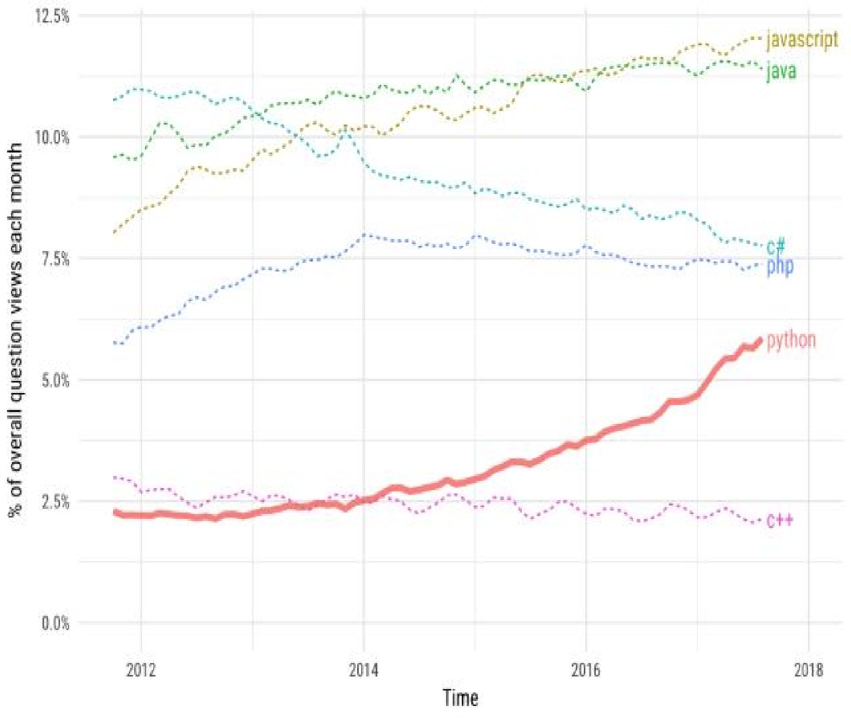


Figure 4.3: Non-high-income countries growth of Python.

At the time of writing, the trend for the most important and used programming languages still verifies that python is growing more than the others. This graph takes into account the overall percentage of question, answers and views around the entire world. As it can be seen, python usage presents a steadily increase, while other shows decreasing or stable trend. There is only one language showing along with Python the same increasing trend and it is R. Both are high-level languages sharing the same properties and same easy-to-use characteristic.

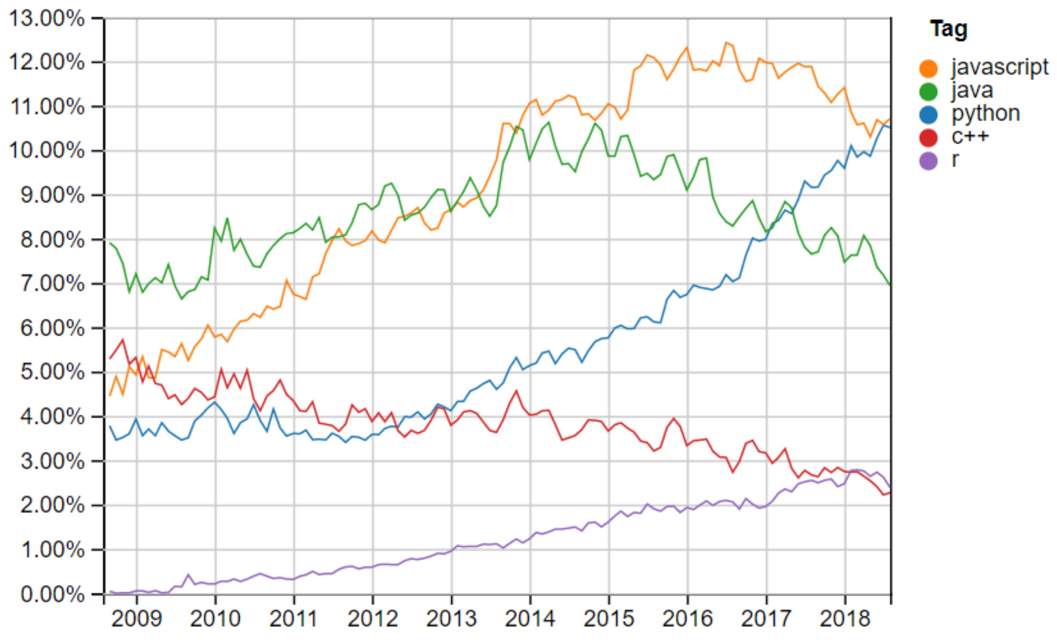


Figure 4.4: Overall usage of Python.

One of the main reasons behind the success of Python is the large number of libraries and frameworks facilitating coding and saving development time. This language is well-known for its concise, readable code, and is almost unrivalled when it comes to ease of use and simplicity, particularly for new developers. The explosion of Machine Learning and Deep Learning as well, let Python growing even more. One driving factor is that both Machine Learning and Deep Learning rely on very complex algorithms and multi-stage workflows. The less a developer has to worry about the problems of coding, using libraries and built-in functions, the more she can focus on finding solution to the problems and achieve the goal of the project. Moreover, due to the simple Python syntax, developers can spend more time focusing to test model and algorithms rather than programming. In addition, most of Machine Learning and Deep Learning projects usually change hands between development teams. Python code offers a great readability, becoming invaluable when it comes to collaborative programming.

4.4. Why Keras?

Keras is a deep-learning framework for Python providing an easy way to model and train almost any kind of neural network systems and deep-learning model. It was initially developed for researchers with the goal of fast experimentation. It brings with itself very important features allowing to run the code on GPUs or CPUs, making easier prototyping and testing and supporting convolutional and recurrent networks as well as any combination of them. Moreover, it supports arbitrary network architectures, being suitable for building any deep learning model. Furthermore, it is distributed under the permissive MIT license, meaning that can be used also for commercial products. It is compatible with all Python versions. Due to these qualities, Keras has been able to reach over 200’000 users, from academic researchers to start-ups and large companies, from graduate students to hobbyists. Currently it is used at Google, Netflix, Uber, Square and tons of fast-growing start-ups. Moreover, it is a popular framework on Kaggle, the machine learning competition website, where almost every recent competition has been won using Keras model. The following figure shows the Google web search interest for different deep-learning frameworks over the last 4 years. As it is shown, only TensorFlow shows more interest and they both follows an increasing trend.

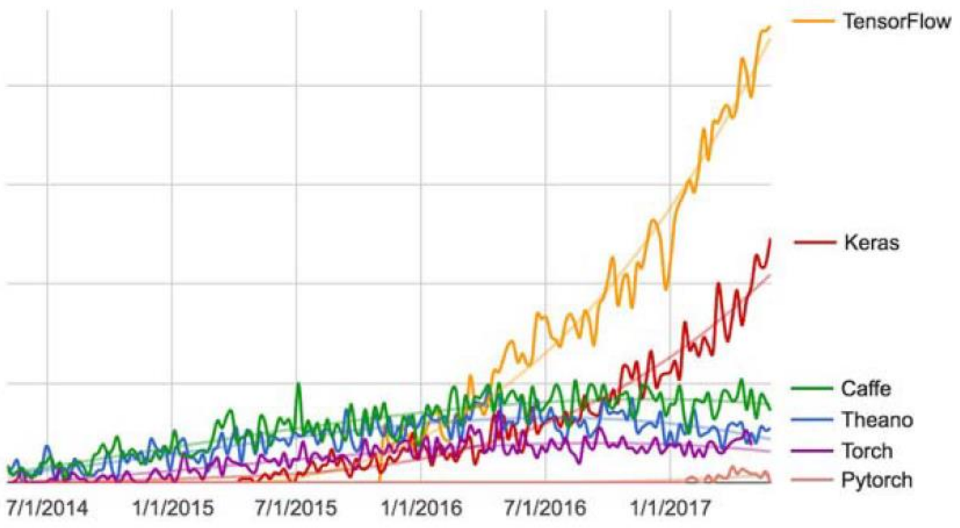


Figure 4.5: Overall usage of Python libraries.

TensorFlow is one of the primary platforms for deep learning. Specifically, it is an open source software library for high performance numerical computation. It has been developed by researchers and engineers from Google and it provides strong support for machine learning and deep learning. Due to the fact that Keras does not handle low-level operations, it uses a backend engine. There are several and TensorFlow is one of these, specifically, it is the one used in this work.

4.5. Feature engineering, data pre-processing and model formulation

Once the coding environment has been set, it is possible to go into the analysis of the feature engineering phase. This phase goes hand in hand with the choice of the model formulation as, based on the input the neural network has, it is possible to swap between classification or regression problem. Before stepping into the final shape of the entire work, it is worthwhile to mention the path leading to it, with all the previous approaches and the errors made since they helped to reach the last result.

The first way to approach the problem has been, given a sequence, to try predicting the triple *(r,h,i)* where with *i* indicates the value of improvement obtained. This is a typical regression problem, where the output is not a simple scalar but a vector. This approach seemed very powerful, but it had a logical bug inside and could not be used due to the way dataset has been built. As already said, the sequence in the dataset does not change after an entire stage of consecutive re-optimizations for all the couples *(r,h)*. With this approach, the neural network has been asked for predicting different values from the same input, i.e. the same sequence of processing times. This has led the neural network to predict a sort of mean value and it was not what was expected to do.

Due to the logical error made in the first approach, the second attempt was constructed in a slightly different way. It has been taken the best improvement *i*, i.e. the one with the highest value, for each stage. Then the input of the neural network has been the sequence as before and the neural network has been asked for predicting couples of value *(r,h)* associated with the sequence. However, since the dataset has twenty instances inside, each one with at most six stages of re-optimizations, in this way, the input sequence for the training phase were at most one hundred and twenty. In facts, this number was a bit lower but still around one hundred and ten. As a result, the input examples for the neural network were too few to learn such a complex problem.

In the third attempt, a change has been made. A classification problem has been set up and the focus has been on whether or not a given sequence, re-optimized in a specific position and a specific window size, would have led to an improvement. Thus, the input now has been the sequence, together with the parameters r and h. The output, instead, was a binary variable indicating with one the presence of an improvement and with the zero the absence. In this approach, an issue came out and it has been very difficult to deal with. Even if the procedure seemed very strong and consistent, without any logical bug, the neural network went on failing to predict the correct values. In facts, the values of processing time of the jobs come from a uniform distribution between one and one hundred. This means that the neural network has been provided with random values coming from a theoretical distribution. Thus, it failed to recognize the pattern to be learnt. Even adding two more pieces of information in the input as the starting position and the window size, they were not enough. One reason could be that, a perfect uniform distribution seems too little informative when it comes to predict and learn pattern form it. However, even this approach did not give the expected results, it helped a lot for the final achievement.

The last approach was only a re-arrangement of the previous one, where it has been added to the sequence more pieces of data with the purpose of perturbate more the sequence of uniformed distributed values. From the dataset, it has been extracted the values of processing times of each job according to the sequence order, then the values of position and window size, i.e. *(r,h)*. However, this time, it has been added to the input the values of the completion time on both machines of the first job before the re-optimized window and the value of completion time of the last job of the entire sequence. The reason behind is that, since the neural network has to learn if a given sequence re-optimized in a specific position and with a specific window size leads to an improvement, providing it with the completion times of the job just preceding the window and with the completion times of the last job, gives to the neural network the information on the situation outside the window. This helps the neural network understanding of how the window re-optimization is affected by the window as well as the entire sequence.

Since the key idea is to provide to the neural network a sequence, it has been chosen a LSTM as the first layer. Most of the works, where LSTM has been used, present this technique so, even if the field of application changed completely, it seemed a consistent choice to be applied. As explained in the previous chapter, a recurrent layer is made up by different cells processing inputs. In this case, the sequence is composed by the processing times of each job on both machines, the starting position, the window size, the completion times of the job before the window on both machine and the completion times on both machine of the last job in the sequence. Imagine a hypothetical x_0 is the first input of the sequence, it stores inside the processing times on both machines of the first job of the sequence and it is followed up by all the job in the sequence. However, when it comes to the input storing the starting position, it has inside only one feature, i.e. the single parameter r. The same applies for all the next inputs. In this situation, two choices were available. The first one was to creates three couples of input with the remaining six. As an example, completion times of the job at the end of the sequences could have been put together in a single input. However, to let the neural network see completely different inputs and to avoid aggregating features, it has been chosen the alternative way, and to duplicate the single features. This helped the network to differentiate one input from another. Thus, a single input of an LSTM layer is a matrix of shape *(2,106)* where two is the number of features and one hundred and six is the sum of the jobs number and the additional pieces of data provided to the network. However, when it comes to gives to the network an entire dataset of inputs, it is required the creation of a tensor.

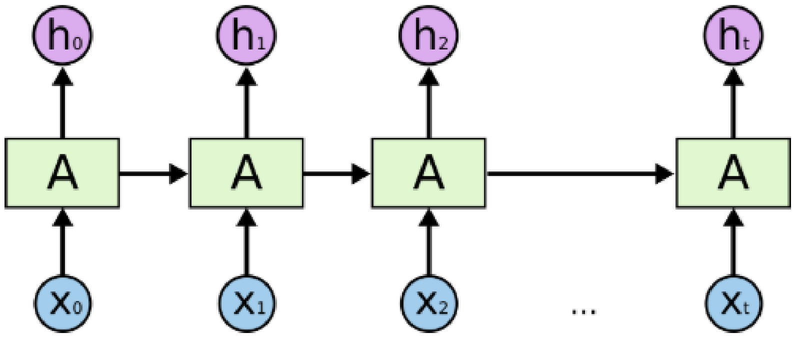
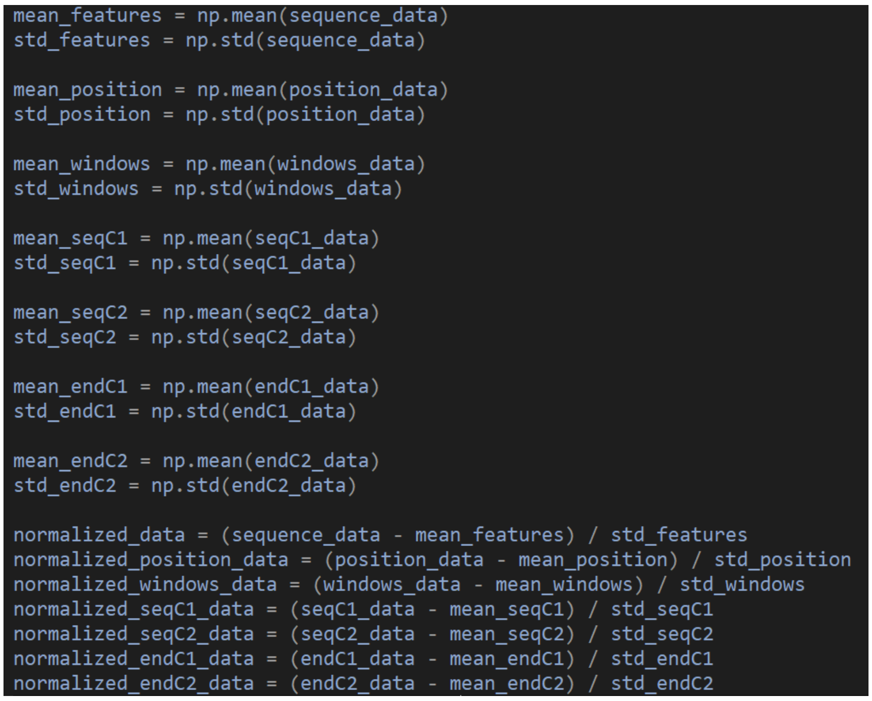


Figure 4.6: The structure of a Recurrent Neural Network.

To accomplish this task, data have been pre-processed in this way. Firstly, have been created the matrices and the tensors containing the data. A tensor called *sequence_data* stores all the processing times on both machine for each job in the sequence and for all the rows in the dataset. Then, the matrices *position_data* and *windows_data* stores all the values for the parameters *r* and *h*. Moreover, the matrices *seq_C1* and *seq_C2* stores the information regarding the completion times on machine one and two respectively of the job before the sequence. Lastly, the matrices *end_C1* and *end_C2* has inside the values of completion times on both machine of the last job in the sequence. The tensor and the matrices have been normalized accordingly to data inside. This task has been accomplished very easily with the functions *mean* and *std* inside the NumPy library. Following, there the formulas and the code associated to it.

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N}$$

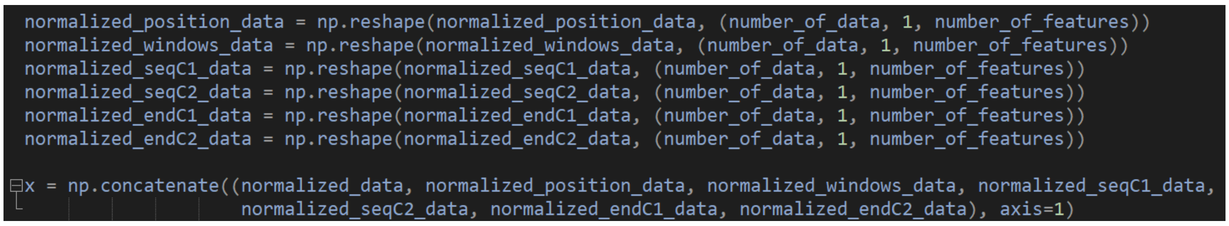
$$\sigma = \sqrt{\frac{\sum_{i=1}^{N} (x_i - \mu)^2}{N}}$$



```
mean_features = np.mean(sequence_data)
std_features = np.std(sequence_data)

mean_position = np.mean(position_data)
std_position = np.std(position_data)

mean_windows = np.mean(windows_data)
std_windows = np.std(windows_data)

mean_seqC1 = np.mean(seqC1_data)
std_seqC1 = np.std(seqC1_data)

mean_seqC2 = np.mean(seqC2_data)
std_seqC2 = np.std(seqC2_data)

mean_endC1 = np.mean(endC1_data)
std_endC1 = np.std(endC1_data)

mean_endC2 = np.mean(endC2_data)
std_endC2 = np.std(endC2_data)

normalized_data = (sequence_data - mean_features) / std_features
normalized_position_data = (position_data - mean_position) / std_position
normalized_windows_data = (windows_data - mean_windows) / std_windows
normalized_seqC1_data = (seqC1_data - mean_seqC1) / std_seqC1
normalized_seqC2_data = (seqC2_data - mean_seqC2) / std_seqC2
normalized_endC1_data = (endC1_data - mean_endC1) / std_endC1
normalized_endC2_data = (endC2_data - mean_endC2) / std_endC2
```

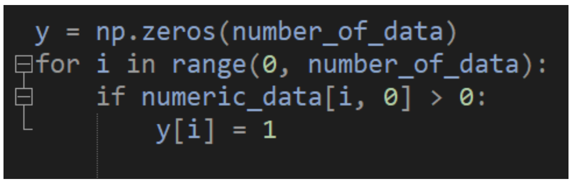
Figure 4.7: The code for the normalization of input data.

Then all the input has been reshaped and concatenated to reach the exact shape to fill the first layer. Regarding the target to be learn by the model, it has been easily coded creating a vector *y* full of zeros as labels and substituting each improvement greater than zero with a one.



```
normalized_position_data = np.reshape(normalized_position_data, (number_of_data, 1, number_of_features))
normalized_windows_data = np.reshape(normalized_windows_data, (number_of_data, 1, number_of_features))
normalized_seqC1_data = np.reshape(normalized_seqC1_data, (number_of_data, 1, number_of_features))
normalized_seqC2_data = np.reshape(normalized_seqC2_data, (number_of_data, 1, number_of_features))
normalized_endC1_data = np.reshape(normalized_endC1_data, (number_of_data, 1, number_of_features))
normalized_endC2_data = np.reshape(normalized_endC2_data, (number_of_data, 1, number_of_features))

x = np.concatenate((normalized_data, normalized_position_data, normalized_windows_data, normalized_seqC1_data,
                    normalized_seqC2_data, normalized_endC1_data, normalized_endC2_data), axis=1)
```

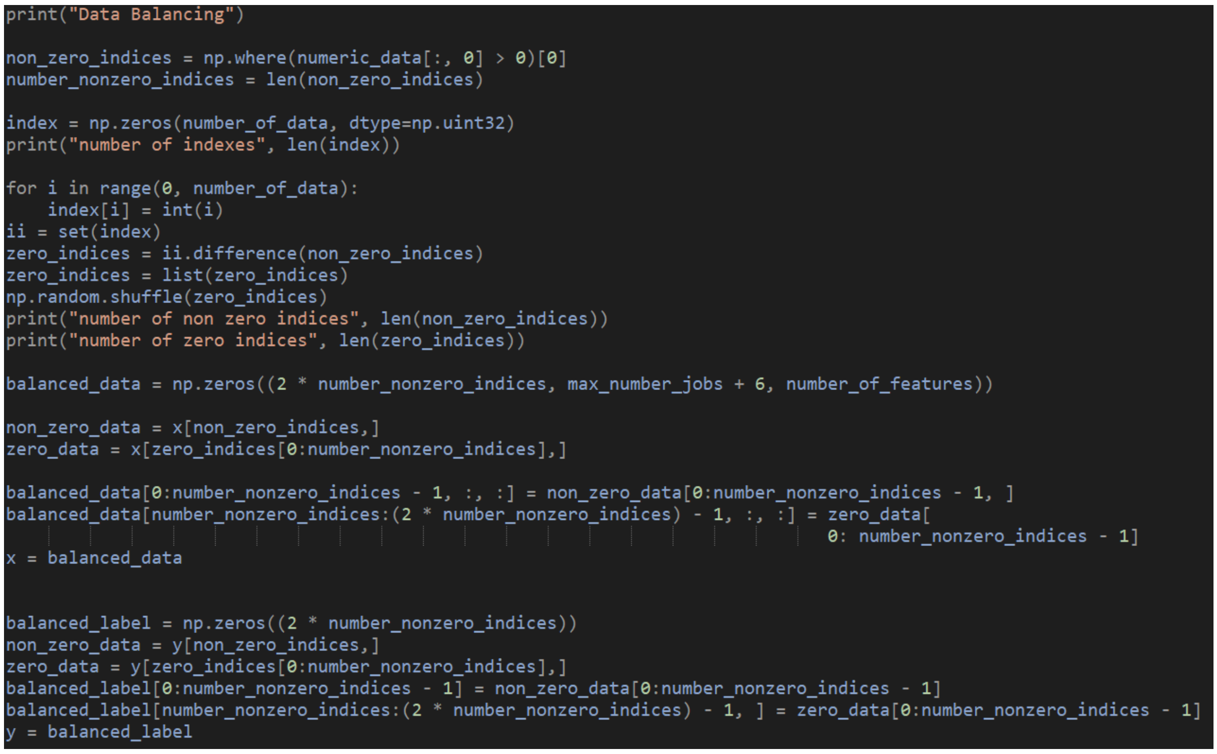
Figure 4.8: The code used for shaping the input data.



```
y = np.zeros(number_of_data)
for i in range(0, number_of_data):
    if numeric_data[i, 0] > 0:
        y[i] = 1
```

Figure 4.9: The code for the creation of labels.

Since this is a classification problem, the neural network has to learn to classify input in some specific categories. However, showing to the neural network some categories more often than other, will bias the learning letting it pend more to the categories seen more often. Due to this, the input must be balanced in order to have the same number of elements in both categories. In this specific case, since the situations with no improvement appeared more often, the input has been balanced removing some of them.



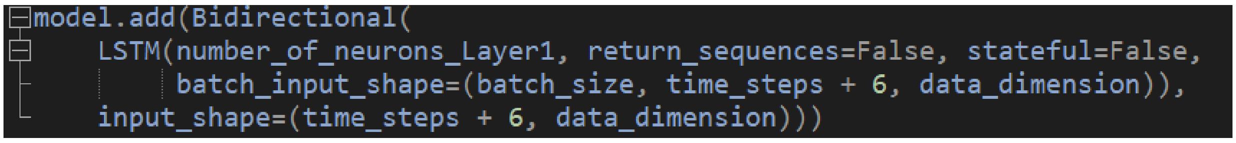
```
print("Data Balancing")

non_zero_indices = np.where(numeric_data[:, 0] > 0)[0]
number_nonzero_indices = len(non_zero_indices)

index = np.zeros(number_of_data, dtype=np.uint32)
print("number of indexes", len(index))

for i in range(0, number_of_data):
    index[i] = int(i)
ii = set(index)
zero_indices = ii.difference(non_zero_indices)
zero_indices = list(zero_indices)
np.random.shuffle(zero_indices)
print("number of non zero indices", len(non_zero_indices))
print("number of zero indices", len(zero_indices))

balanced_data = np.zeros((2 * number_nonzero_indices, max_number_jobs + 6, number_of_features))

non_zero_data = x[non_zero_indices,]
zero_data = x[zero_indices[0:number_nonzero_indices],]

balanced_data[0:number_nonzero_indices - 1, :, :] = non_zero_data[0:number_nonzero_indices - 1, ]
balanced_data[number_nonzero_indices:(2 * number_nonzero_indices) - 1, :, :] = zero_data[
                                                                                0: number_nonzero_indices - 1]
x = balanced_data


balanced_label = np.zeros((2 * number_nonzero_indices))
non_zero_data = y[non_zero_indices,]
zero_data = y[zero_indices[0:number_nonzero_indices],]
balanced_label[0:number_nonzero_indices - 1] = non_zero_data[0:number_nonzero_indices - 1]
balanced_label[number_nonzero_indices:(2 * number_nonzero_indices) - 1, ] = zero_data[0:number_nonzero_indices - 1]
y = balanced_label
```

Figure 4.10: The code for the input data balancing.

The following step has been the model creation. This can be done very easily in Python, when using Keras. In facts, the model required no more than thirty lines of code. The first line includes the creation of an object model of the kind *"Sequential"*. A sequential model is a linear stack of layers. It is possible to add layers to the model just calling the function "*add"*.

The following lines add to the model a bidirectional layer. Specifically, a bidirectional layer is a layer not only with forward connection but also presenting backward propagation. Since the network has to learn how the sequence of processing times and the other inputs affect the window re-optimization, the position of a job in the sequence is affected not only by the previous jobs, but also of what comes next. In facts, the position where to re-optimize is in the middle of the entire sequence. Clearly, this element is affected by the sequence of processing times before, but also by the windows size which is next to it and by the completion times at the end of the sequence. Thus, using a bidirectional layer helped the network to learn from the entire values of the sequence and not only on what occurs before.

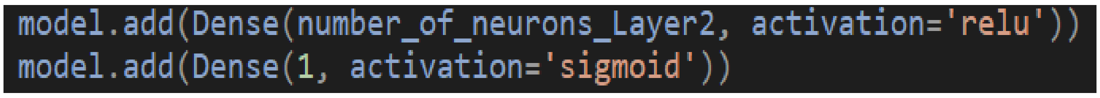
The bidirectional layer has been chosen to be of LSTM type due to the reason explained in the second chapter. Each LSTM cell has inside a specific number of neurons specified by the variable "*number_of_neurons_Layer1"*. The choice of declaring a variable and to not inserting directly the number helped the readability of the code as well as the possibility to tune this hyperparameter easily. Then the argument *"return_sequence"* of the LSTM layer has been set to *"False"*. This argument decides if returning all the sequence of values or just the last one. When stacking more LSTM layers, sometimes each cell output becomes the cell input of the next LSTM layer, so it could be useful to return the entire sequence. However, when the next layer is a Dense one, it does not take as input a sequence, so returning it cannot be done. The next argument is *"Stateful"* and again has been set to *"False"*. This argument takes the last state for each sample at index *i* in a batch and uses it as initial state for the sample of index *i* in the following batch. Actually, this function helps the flows of information throughout the entire dataset. This is useful when analysing timeseries, because the sequence of values is time dependent and this dependence goes through different batches in the dataset. However, since this dataset is composed by completely different instances, it has no sense to let the information flows from a problem to another. After that, since this is the first layer, it has been set the dimension of the batch size as well as its shape. This task has to be done only in the first layer because every next layer can infer its input shape by looking at the previous layer.



```
model.add(Bidirectional(
    LSTM(number_of_neurons_Layer1, return_sequences=False, stateful=False,
         batch_input_shape=(batch_size, time_steps + 6, data_dimension)),
    input_shape=(time_steps + 6, data_dimension)))
```

Figure 4.11: The line of code describing the first LSTM layer.

Another layer has been stacked after the first one. It is a Dense layer, a very simple one with a specific number of neuron and a ReLU as activation function. The last layer is, again, a Dense one, with one neuron inside since the output is a scalar and a sigmoid as activation function. The sigmoid function helps the neuron to binary classify the output due to its shape.

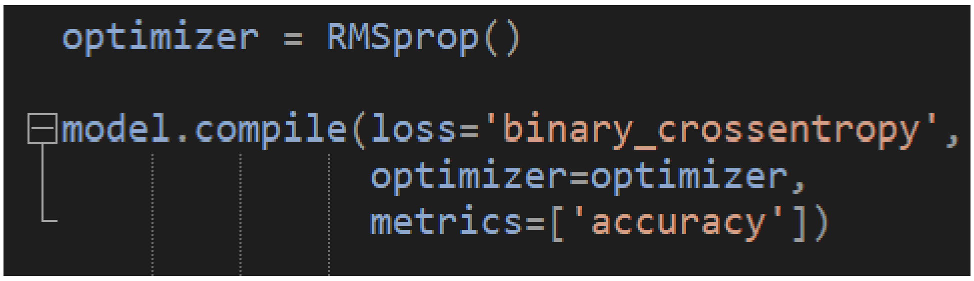


```
model.add(Dense(number_of_neurons_Layer2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Figure 4.12: The code for the following two Dense layers.

As optimizers has been chosen the *"RMSprop"*, it seemed a solid choice and no change in its own parameters have been made. Every parameter as the learning rate and others have been left as default. This choice has been made to not overcharge the phase of the hyperparameters tuning with too many parameters to be monitored.

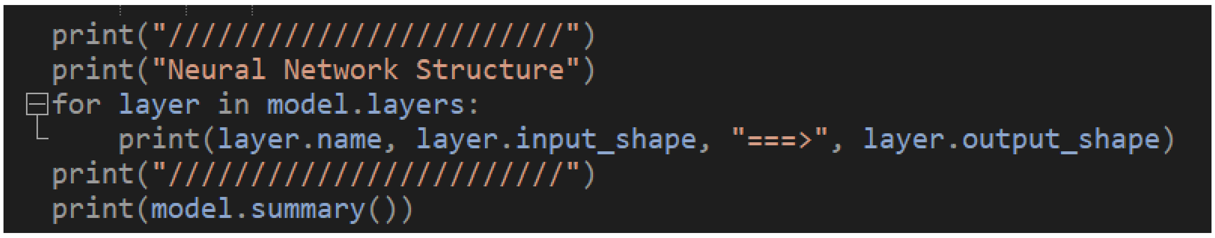
Before training the model, it has to be compiled via the *"compile"* function. It takes as argument the loss function, the optimizers, and the metrics to be measured. Since it is faced a binary classification problem, the loss function used has been the binary crossentropy and as a metrics the accuracy has been chosen.



```
optimizer = RMSprop()

model.compile(loss='binary_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

Figure 4.13: How the optimizer has been set up and the function used to compile the model.

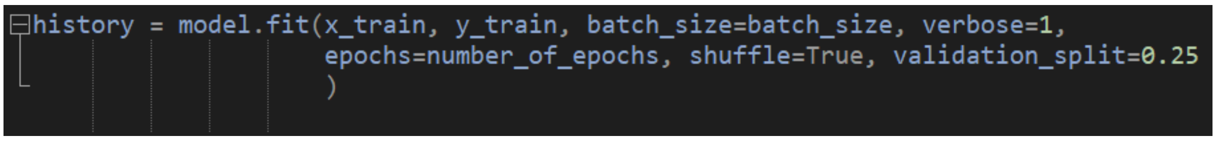
After these lines, it has been printed on the screen the structure of the neural network. This helped the process of understanding how the information flows from a layer to another and how the input changes its shape during the process. In the hyperparameter tuning phase, this helps the balancing of the number of neurons inside the layers and to avoid compressing information flows. It is easy sometimes, when building the model, to unbalance the inflow of information and its outflow. Sometimes, the inflow of information is too big related to number of neurons inside the layer. This cause the layer to compress the information in too few parameters, loosing important pieces that could help the learning. In the following line, there is the printing of the model summary. Again, this shows the situation regarding how many parameters are created inside the whole network. Since deep learning works more with trial and error than with a concise and precise theory, sometimes it is easy to lose track of how many parameters the network is managing. An unbalanced number of weights is one of the most likely reason of a not working model no. Overfilling and/or underfilling the neural network with too many layers and/or too many neurons as well as too few, cause overfitting and/or underfitting.



```
print("/////////////////////////")
print("Neural Network Structure")
for layer in model.layers:
    print(layer.name, layer.input_shape, "===>", layer.output_shape)
print("/////////////////////////")
print(model.summary())
```

Figure 4.14: How to visualize the Neural Network structure.

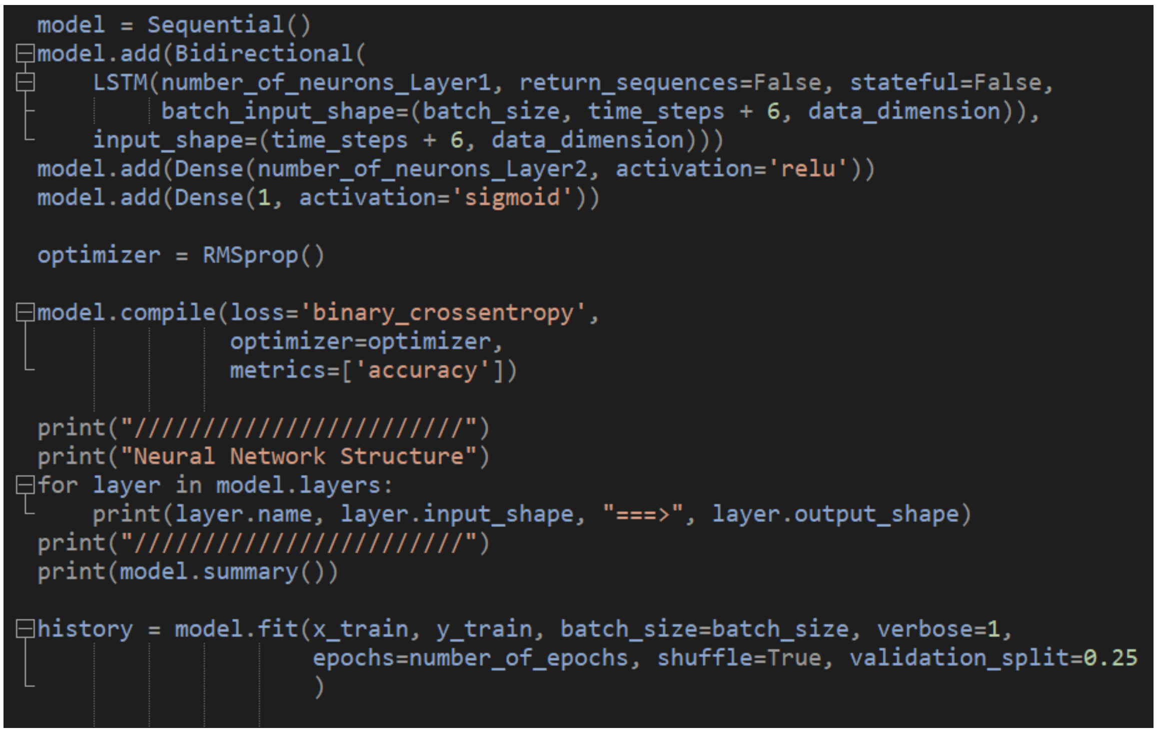
Lastly, the function setting the training has been called. It is the *"fit"* function and takes several arguments. The first two are the input and the target; then the parameter including the size of the batch together with the number of epochs have to be set. After that, the argument *"shuffle"* has been set to *"True"*. This means that before any epoch the training dataset is shuffled, and the training does not take examples in the sequence order. The last one is the argument *"validation_split"*. It indicates the percentage of the training dataset used for validation. In this case, it has been used a value of 0.25. This means that, a quarter of the entire dataset is dedicated to the validation. In this field, usually, validation splits move from a minimum of 0.15 up to a maximum of 0.35. A choice in the middle seemed the most consistent.



```
history = model.fit(x_train, y_train, batch_size=batch_size, verbose=1,
                    epochs=number_of_epochs, shuffle=True, validation_split=0.25
                    )
```

Figure 4.15: The training function.

Here there is the entire section of the Python code where the neural network has been set up and trained. As it can be seen, it required very few lines of code and let the prototyping phase as well as the hyperparameters tuning being more under control.



```
model = Sequential()
model.add(Bidirectional(
    LSTM(number_of_neurons_Layer1, return_sequences=False, stateful=False,
         batch_input_shape=(batch_size, time_steps + 6, data_dimension)),
    input_shape=(time_steps + 6, data_dimension)))
model.add(Dense(number_of_neurons_Layer2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

optimizer = RMSprop()

model.compile(loss='binary_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])

print("///////////////////////")
print("Neural Network Structure")
for layer in model.layers:
    print(layer.name, layer.input_shape, "===>", layer.output_shape)
print("///////////////////////")
print(model.summary())

history = model.fit(x_train, y_train, batch_size=batch_size, verbose=1,
                    epochs=number_of_epochs, shuffle=True, validation_split=0.25
                    )
```

Figure 4.16: How the entire model looks like.

4.6. Hyperparameters Tuning

This the last phase of the entire work but however, it is the one requiring more time. During this phase, a general workflow of machine learning has been followed. After assembling the dataset and defining the problem, the data have been prepared and a model has been built. Usually, the first model has enough statistical power but at its first launch will never perform perfectly. In this case two main assumption comes into play:

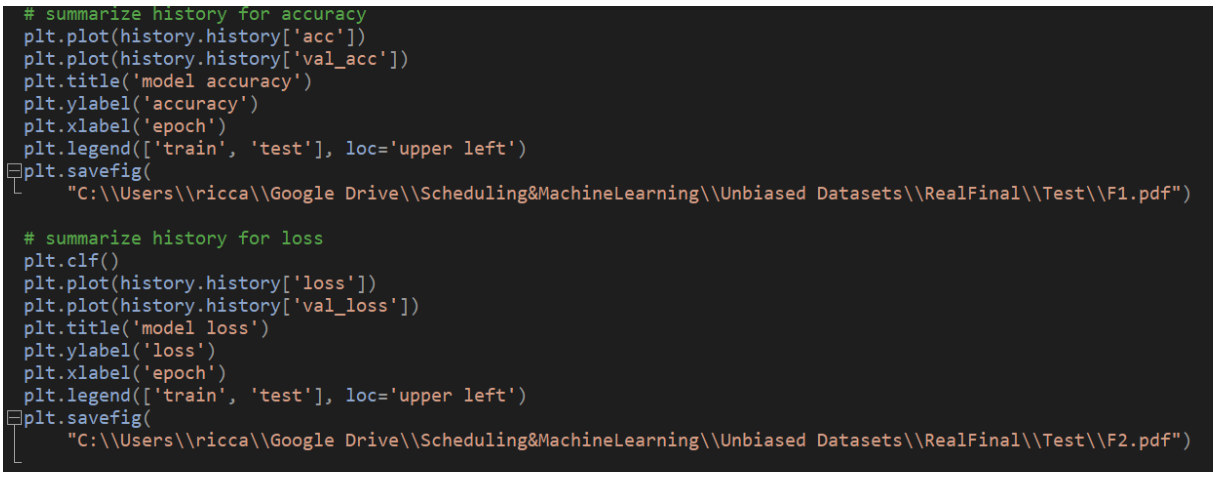
- The outputs can be predicted given the input
- The available data is sufficiently informative to learn the relation between inputs and outputs.

Remembering that the universal tension in machine learning is between optimization and generalization, the perfect model will stand in the border between overfitting and underfitting. To understand where the border between the overcapacity and undercapacity is, it is usually a good idea to cross it. This means that, usually, the first step is to develop a model that overfits. This task is pretty easy to accomplish and requires:

- Adding layers
- Adding neurons to layers
- Train for a big number of epochs

To better reach this goal, it can be useful to monitor the measure of success and the value of the loss function for the training dataset as well as for the validation dataset. An overfitting model will perform incredibly well on the training dataset but will perform poorly when it comes to the validation dataset. After each training, to see how it performed, it is sometimes useful to plot the results. This has been easily done via the Matplotlib library using the values inside the model. Internet is full of explanation on how to set up an environment for plotting the results after the training. Finding some insights on the code to be used has been very easy and it required, again, very few lines of code.

Firstly, it has been plotted the value of the accuracy for the training dataset and for the validation dataset. On the y axis has been put the accuracy value and on the x axis the number of epochs. This configuration helped to see how the accuracy changed over the training phase and how much was the difference between the training dataset and the validation dataset. Then, it has been plotted the value of the loss function for both datasets, the training and the validation ones. Again, on the x axis it has ben put the number of epochs to figure out how the loss decreased over time. The plots have been always saved to keep trace of the progresses achieved during the hyperparameter tuning phase.



```
# summarize history for accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig(
    "C:\\Users\\ricca\\Google Drive\\Scheduling&MachineLearning\\Unbiased Datasets\\RealFinal\\Test\\F1.pdf")

# summarize history for loss
plt.clf()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig(
    "C:\\Users\\ricca\\Google Drive\\Scheduling&MachineLearning\\Unbiased Datasets\\RealFinal\\Test\\F2.pdf")
```

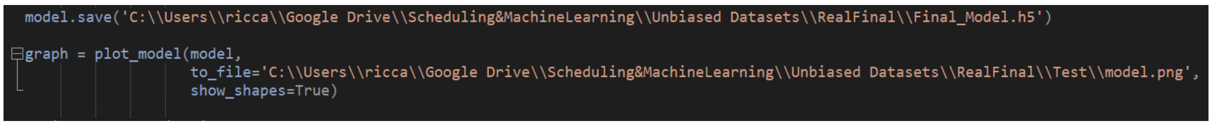
Figure 4.17: How the results have been constructed and visualized.

Even if the model shown in the previous section was already the final one, it was not the starting point. The very first model was completely different. It presented three LSTM layers and only one Dense layer at the end. Moreover, since the first step was to develop an overfitting model, it had more neurons. After the development of the overfitting model, it came into play the most difficult part. The model has been repeatedly modified, trained and evaluated again and again until its best point. The focus of this part is on removing overfitting, regularizing and tuning the hyperparameters until the model does not overfit nor underfit. Usually, there are some classical steps that comes into help in this phase:

- Adding dropout
- Trying different architectures, i.e. adding and/or removing layers
- Adding L1 or L2 regularization
- Try different hyperparameters (number of neurons, batch size, learning rate…)

Furthermore, in this phase there is no theory explaining how the change of hyperparameters affects the model performance. A trial and error approach is always preferred and a sort of local search is used. Most of the times, it is preferred to start decreasing the number of neurons until a satisfactory point. Then it can be adjusted the learning rate as well as the batch size. After that, dropout, L1 and L2 regularization as well as different architectures come into play. There are plenty of hyperparameters that can be changed and some of them could seem hidden. Inside the LSTM can be changed the activation function and the recurrent activation function. It can be inserted the dropout also in the LSTM layer and not only when moving from a layer to another. After that, the number of epochs can be increased or decreased accordingly to the behaviour of the model. However, in Machine Learning field a basic principle comes into help and it is the Occam razor's principle. So, a simple model should always be preferred. With this in mind, the choice has been to let some hyperparameters as default and just play around very few ones. One of these has been the learning rate, together with the activation functions and the dropout inside the LSTM layers. This helped a lot the procedure and speeded up the research of an adequate model for this specific problem.

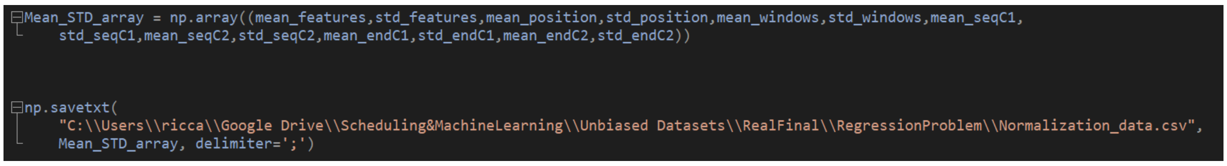
Once a satisfactory model came out, it has been trained the last time. This time, it has been added to the code a line where the model has been saved. Once creating a model that works well, there is no sense to re-train the model every time it is needed to predict something. To avoid this, Keras library has a function called *"save"*. It saves the model and all the parameters after the training so that it can be recalled in other codes where it has to be used. Moreover, using another function, the model has been plotted in its final shape.



```
model.save('C:\\Users\\ricca\\Google Drive\\Scheduling&MachineLearning\\Unbiased Datasets\\RealFinal\\Final_Model.h5')

graph = plot_model(model,
                   to_file='C:\\Users\\ricca\\Google Drive\\Scheduling&MachineLearning\\Unbiased Datasets\\RealFinal\\Test\\model.png',
                   show_shapes=True)
```

Figure 4.18: Saving and plotting the model.

Furthermore, another two lines of code have been added before training the model for the very last time. When dealing with data pre-processing, all the pieces of data have been normalized. However, when recalling the model to build the predictor, data will be normalized again. This task cannot be accomplished using mean and standard deviation computed on the test dataset. In facts, the normalization parameters that have to be used must be the same used for training. To do so, a vector of values have been saved in order to re-call it for further uses. Thanks to NumPy library, it took just two lines of code and they were the following.

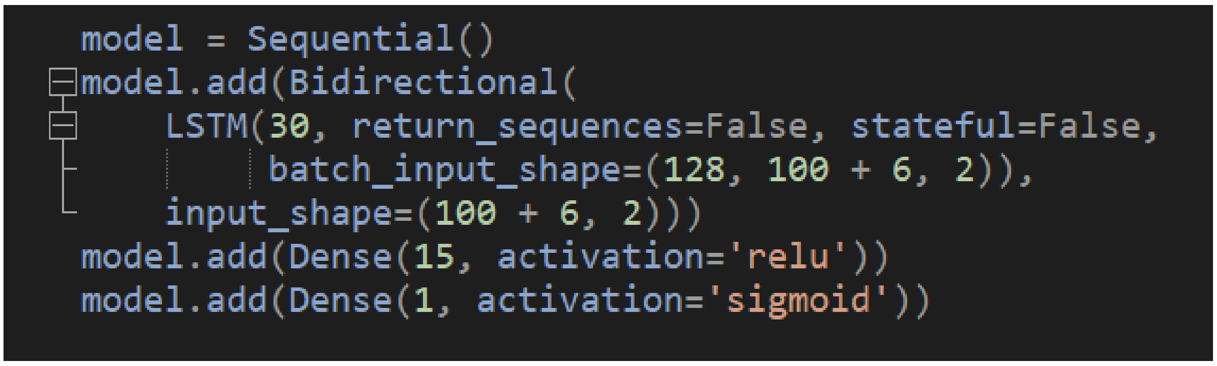


```
Mean_STD_array = np.array((mean_features,std_features,mean_position,std_position,mean_windows,std_windows,mean_seqC1,
    std_seqC1,mean_seqC2,std_seqC2,mean_endC1,std_endC1,mean_endC2,std_endC2))


np.savetxt(
    "C:\\Users\\ricca\\Google Drive\\Scheduling&MachineLearning\\Unbiased Datasets\\RealFinal\\RegressionProblem\\Normalization_data.csv",
    Mean_STD_array, delimiter=';')
```

Figure 4.19: Saving the mean and the std to future normalizations.

4.7. The last model

Fortunately, the hyperparameter tuning reached a very successful model performing greatly on both training and validation datasets. This is not granted when dealing with completely new models that have to learn new tasks. The model presents a single LSTM layer with 30 neurons inside, followed up by two Dense layers. The first one has 15 neurons, while the second one, since it is the last one, presents just 1 neuron to shape the output as a scalar. The batch size has been put on 128. This number comes out from other previous works in the machine learning field. Usually, it is chosen a number coming from the powers of two, balanced with the amount of data in the entire dataset. As often in this field, there is no specific theory around this choice, but it works well for most of the projects. Thus, the input batch of the first layer is a tensor of shape *(128,106,2)*, where 106 is the sum of the number of jobs and the other six pieces of data associated to them and 2 is the number of features. Moreover, this model has been trained for 400 epochs.



```
model = Sequential()
model.add(Bidirectional(
    LSTM(30, return_sequences=False, stateful=False,
         batch_input_shape=(128, 100 + 6, 2)),
    input_shape=(100 + 6, 2)))
model.add(Dense(15, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Figure 4.20: The last model.

The flows of information inside the network is dictated by the number of neurons and by the kind of layers. Every time, a batch of shape *(128,106,2)* enters the LSTM layers and starts its processing. Since the LSTM layers has 30 neurons, it is bidirectional, and it does not return the sequence, the output of the first layers is a batch of *(128,60)*. This batch enters the second layers and start its transformation. Since the second layer Dense with 15 neurons, the output is a batch of shape *(128,15)*. Lastly, the batch outputted by the second layer becomes the input of the third one. Here, there is another Dense layer with only one neuron inside. This layer applies the sigmoid function to the input and output a scalar between 0 and 1 to classify the improvement. The figure below helps explaining what happens in the network. However, it does not take into account the single batch or the entire dataset, but just the shapes dictated by the layers.

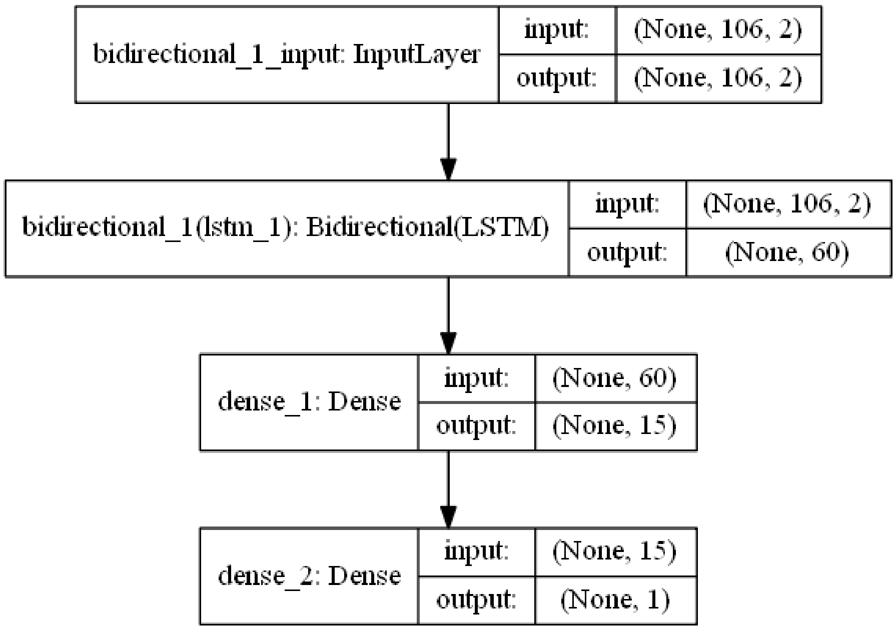
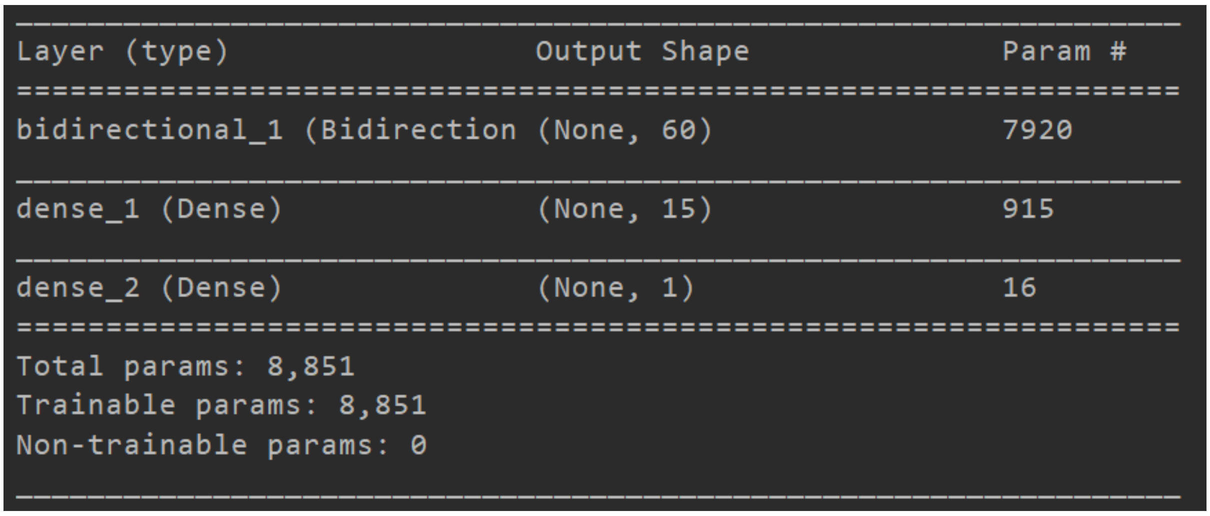


Figure 4.21: The last model structure.

In the previous section, it has been shown that the model summary can help understanding the number of weights associated to each layer. The figure below shows the model summary for the last working model. As it can be seen, most of the work is accomplished by the LSTM layer since it has the biggest number of weights inside. Since the input is a sequence, it has been given more power to a sequences-specific layer. This choice seemed to be the more consistent and at the end it paid back. Moreover, having another Dense layer in between the first and the last helps the compression of information avoiding losing too much of it. This situation has been strongly focused on in the hyperparameter tuning.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bidirectional_1 (Bidirection (None, 60)                7920
_________________________________________________________________
dense_1 (Dense)              (None, 15)                915
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 16
=================================================================
Total params: 8,851
Trainable params: 8,851
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.22: The model summary.

Since the model reached a great performance during the training, the accuracy was very high for both training and validation dataset. The model was able to reach a value of around 0.95 for the training dataset and 0.92 for the validation dataset. Even if there is a little discrepancy between both, this is a very reliable result to work with. There was no sense in going on training because the model reached a plateau and to have a greater accuracy, it would need too many epochs. The trade-off between the time consumption and the accuracy increase achievement was strongly in favour of the time consumption. Moreover, these values are very consistent, and a greater accuracy could have caused overfitting, letting the validation accuracy to decrease. The same situation applies for the loss function. Obviously, it goes hand in hand with the accuracy and also this result was consistent with the power of model that has been reached. The graphs below show the behaviour of the accuracy and loss function for training and validation datasets during the training.

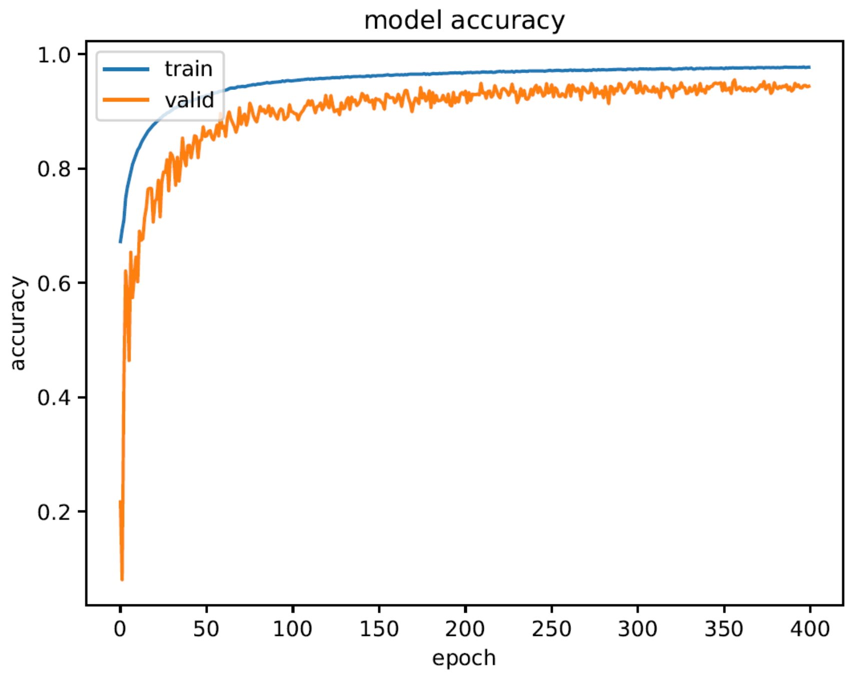


Figure 4.23: The model accuracy on train and validation datasets over the number of epochs.

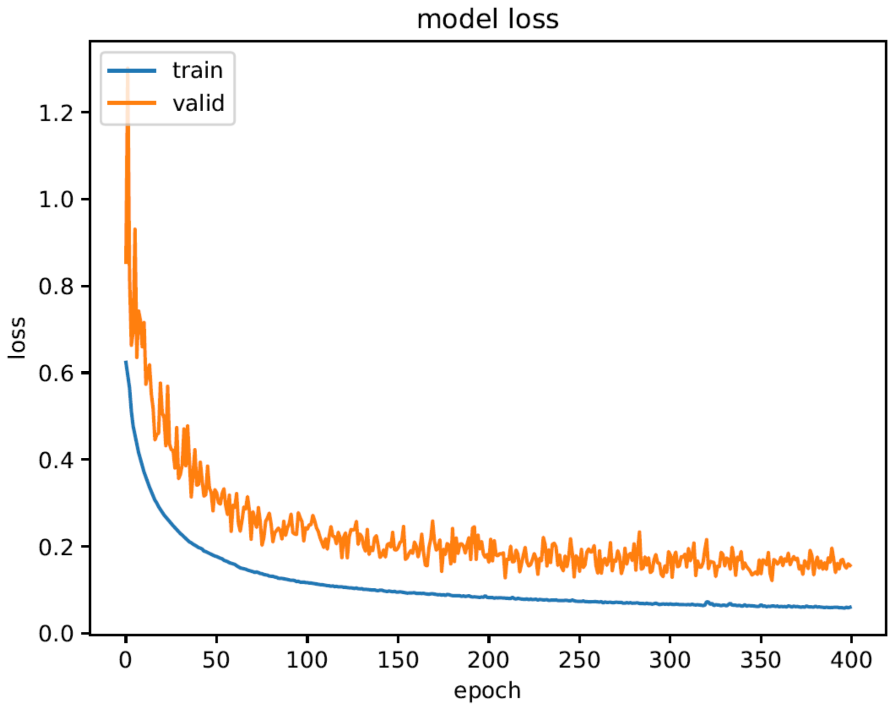


Figure 4.24: The model loss function for train and validation datasets over the number of epochs.

After the model has been trained and a satisfactory level of accuracy has been reached, the following step has been the creation of a predictor to ultimately run the procedure on a testing dataset. However, this is the subject of the next section. This would increase the understanding of how the testing phase works and to focus on this part specifically.

5. The predictor

The previous chapter examined the problem in its entirety. A description of the field of Operation Research with a focus on the scheduling problem is followed by an entire section on the Machine Learning field explaining how Neural Networks work. Then the work has been addressed with the results of the training part presented in the last chapter. This chapter instead, will focus on the creation of a predictor, that will step into the future matheuristic to help the decision of whether or not re-optimize the sequence. In doing this, Python and Keras will be used again, since it will be easy to re-call a previously trained model and to use Keras built-in functions. Then, an entire section is dedicated to the computational results obtained and further improvements that can help reaching better performances.

5.1. The testing phase

After the training, the next phase to be performed has been the testing one. In the testing phase, usually, a completely new dataset is created, normalized and used to test the goodness of the training phase. The idea behind is that, if a new dataset is created and this dataset comes from the same distribution of the previous one, the model should perform in the same way, despite a little value of tolerance. This means that, the same value of accuracy should be seen also for the last phase. To accomplish this part of the work, usually a predictor is coded. The predictor is none other than a function taking into consideration all the weights of the training phase. After the training, when a satisfactory level of accuracy and model loss is reached, the model has been saved. With it, all the weights have been stored. Due to that, it is possible to re-call the set of weights, with a specific function in Keras, and apply these weights on new input. Considering S as a sequence input, presenting all the features and all the characteristics of all the previous input sequences used in the training phase, the prediction function $f()$ outputs a value between 0 and 1 using the weights adjusted in the training phase.

$$f(S) \in [0,1]$$

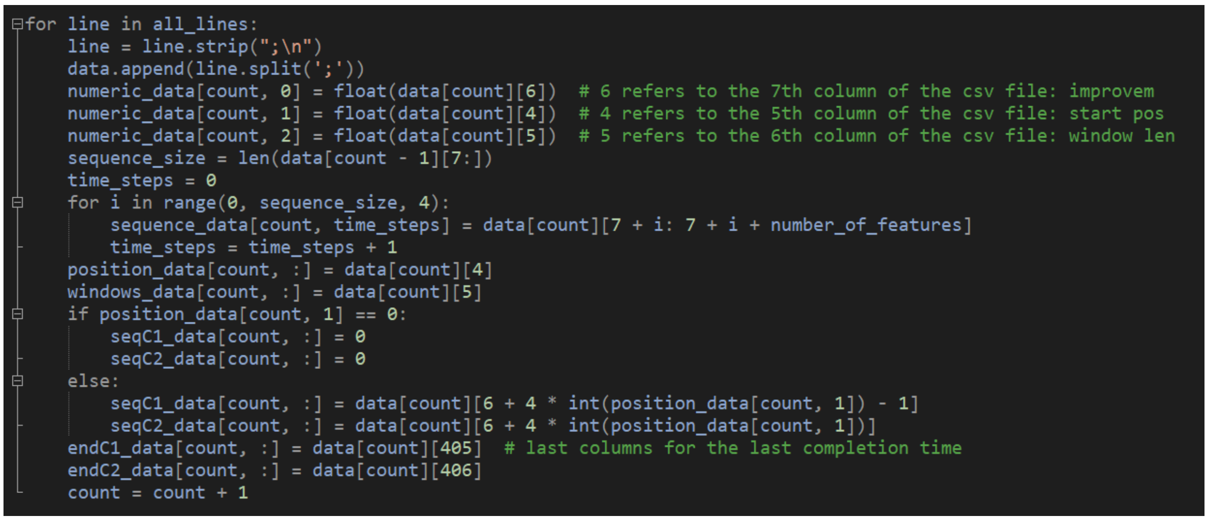
Since the output of the prediction function is a real number classifying a binary situation, it has sense to consider as 1 all the numbers greater or equal than 0.5 while considering as 0 all the number lower than the threshold.

After having confirmed the testing phase, the prediction function can be implemented in the matheuristic code. The idea behind is that, focusing only on the re-optimization where an improvement has been predicted, the whole algorithm would be able to save time that in the old matheuristic has been spent doing all the re-optimization. This will mean that the matheuristic would only re-optimize when it is almost sure that an improvement will be found.

5.2. The creation of a predictor

The procedure started with the creation of a new dataset. This dataset contained four instances inside, that were completely new for the neural network. The code created for the predictor worked only on this dataset. In order to let things clear, even if the training dataset and the test dataset come from the same C++ software, they have been built separately to not confuse things in this phase. From now, with the word dataset will be referred only the test dataset if not specified.

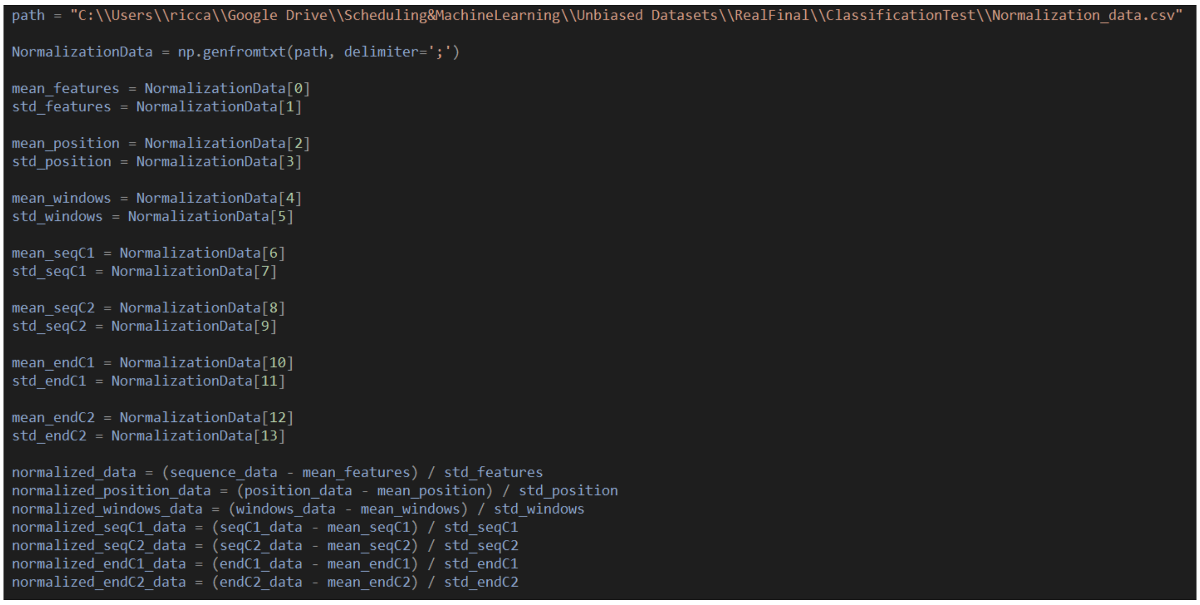
The first step has been to read data in the test dataset in the same way it has been made for the training part. So, the dataset has been imported and data has been stored in the same vector and matrices having the same size as in the previous chapter. Again, *"numeric_data"* stored the improvement, the starting position and the window size for all re-optimizations. *"sequence_data"* stored the processing times of each jobs in the ordered sequence. *"position_data"* and *"windows_data"* stored the parameters *r* and *h*. The remaining four matrices stored the completion times on both machine of the job before the sequence and of the last job. For clarity and coding issue, it has been decided to let the same name for the same variable in both training and test code.



```
for line in all_lines:
    line = line.strip(";\n")
    data.append(line.split(';'))
    numeric_data[count, 0] = float(data[count][6])  # 6 refers to the 7th column of the csv file: improvem
    numeric_data[count, 1] = float(data[count][4])  # 4 refers to the 5th column of the csv file: start pos
    numeric_data[count, 2] = float(data[count][5])  # 5 refers to the 6th column of the csv file: window len
    sequence_size = len(data[count - 1][7:])
    time_steps = 0
    for i in range(0, sequence_size, 4):
        sequence_data[count, time_steps] = data[count][7 + i: 7 + i + number_of_features]
        time_steps = time_steps + 1
    position_data[count, :] = data[count][4]
    windows_data[count, :] = data[count][5]
    if position_data[count, 1] == 0:
        seqC1_data[count, :] = 0
        seqC2_data[count, :] = 0
    else:
        seqC1_data[count, :] = data[count][6 + 4 * int(position_data[count, 1]) - 1]
        seqC2_data[count, :] = data[count][6 + 4 * int(position_data[count, 1])]
    endC1_data[count, :] = data[count][405]  # last columns for the last completion time
    endC2_data[count, :] = data[count][406]
    count = count + 1
```

Figure 5.1: How data have been stored as input in the testing phase.

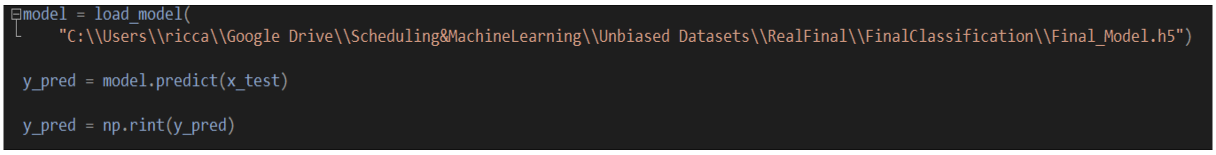
After that, data has been normalized using the means and the standard deviations computed in the previous phase. In this way, test data can prove not only the generality of the model but also the generality of the pre-processing part. Since the aim is to create a predictive algorithm, the test dataset helped testing the overall procedure instead of only the neural network model. Using the NumPy built-in function *"genfromtxt"* is has been constructed a vector with values stored in the .csv previously created in the training phase. Then, following the same order in which values have been stored in sequences, variables have been assigned with the values. Finally, the normalization on input data occurred.



```
path = "C:\\Users\\ricca\\Google Drive\\Scheduling&MachineLearning\\Unbiased Datasets\\RealFinal\\ClassificationTest\\Normalization_data.csv"

NormalizationData = np.genfromtxt(path, delimiter=';')

mean_features = NormalizationData[0]
std_features = NormalizationData[1]

mean_position = NormalizationData[2]
std_position = NormalizationData[3]

mean_windows = NormalizationData[4]
std_windows = NormalizationData[5]

mean_seqC1 = NormalizationData[6]
std_seqC1 = NormalizationData[7]

mean_seqC2 = NormalizationData[8]
std_seqC2 = NormalizationData[9]

mean_endC1 = NormalizationData[10]
std_endC1 = NormalizationData[11]

mean_endC2 = NormalizationData[12]
std_endC2 = NormalizationData[13]

normalized_data = (sequence_data - mean_features) / std_features
normalized_position_data = (position_data - mean_position) / std_position
normalized_windows_data = (windows_data - mean_windows) / std_windows
normalized_seqC1_data = (seqC1_data - mean_seqC1) / std_seqC1
normalized_seqC2_data = (seqC2_data - mean_seqC2) / std_seqC2
normalized_endC1_data = (endC1_data - mean_endC1) / std_endC1
normalized_endC2_data = (endC2_data - mean_endC2) / std_endC2
```

Figure 5.2: Input data normalization.

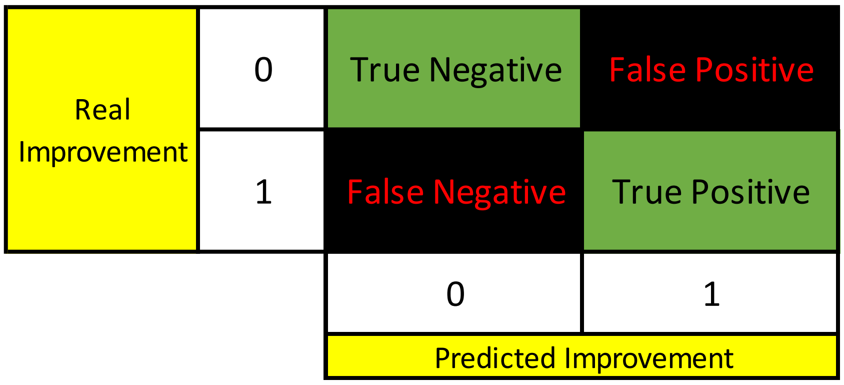
The next step has been the shaping of the test input data in the same way it has been done in the training phase. Exactly the same code has been used, since the aim is to feed the same neural network. After the input, also the vector containing the labels have been created in the same way. Finally, the model has been loaded and the function *"predict"* has been called. The function *"load_model"* can recall the architecture of the model and the weights of the model. With this, all the function associated to the model are callable again. Thus, after calling the model, the function predict has been used. It generates output prediction using the input. In this case, the input has been the data coming from the test dataset. Since the output of the prediction is a scalar from 0 to 1, it has sense to interpret the intermediate values as 0 if they are less than 0.5 and as 1 if they are greater or equal than 0.5. To do this, the built-in function *"rint"* of the NumPy library has been used.



```
model = load_model(
    "C:\\Users\\ricca\\Google Drive\\Scheduling&MachineLearning\\Unbiased Datasets\\RealFinal\\FinalClassification\\Final_Model.h5")

y_pred = model.predict(x_test)

y_pred = np.rint(y_pred)
```

Figure 5.3: The model has been recalled and the function "*predict"* has been used.

In order to set up a clean environment for the testing phase, the testing dataset has been split in single instances. This has helped the testing phase, measuring the accuracy for every instance and helping to check the procedure. For each instance, it has been computed the number of True positives, True Negatives, False Positives and False Negatives and they have been compared among instances. With True Negatives (TN) it is referred the situation where the prediction and the real value agree on the absence of an improvement. Instead, with True Positive (TP) is indicated the presence of a real improvement predicted by the predictor. However, when it comes to make mistakes, a False Positive (FP) occurs when the prediction of an improvement is computed when there is no improvement at all. The remaining situation, where no improvement is predicted when it actually exists, is called False Negative (FN). Usually, the terms True and False are used to describe the real condition, while Negative and Positive are used to refers to the prediction call.



Real Improvement	0	True Negative	False Positive
	1	False Negative	True Positive
		0	1
		Predicted Improvement	

Figure 5.4: The table shows of possible combination between prediction and real presence of improvements in the testing dataset.

In order to better visualize the results coming out from the testing phase, a table of confusion, sometimes called confusion matrix, has been used. As the previous figure, it shows the number of predictions associated to each of the four categories earlier described. Moreover, two confusion matrices have been used, one reporting the exact number of True Positives, True Negative, False Positives and False Negatives and another one reporting the normalized value, using the percentage. To perform the analysis of the results, also four ratios have been considered.

- **Precision**: it is the ratio between the number of True Positives over all the right predictions.

$$\frac{TP}{TP + TN}$$

  It is a ratio describing how many correct predictions of improvement, the model is able to make, over the entirety of correct predictions.

- **Sensitivity**: it is the ratio of the number of True Positive over the number of all the condition positive.

$$\frac{TP}{TP + FN}$$

  It is the percentage of number of improvements detected over all the real improvement in the testing dataset. It gives the probability to detect an improvement given that the improvement exists.

- **Specificity**: it is the ratio between the number of True Negatives over all the condition negative.

$$\frac{TN}{TN + FP}$$

It is the percentage of number of no-improvement detected over all the real situations where no improvement has occurred. It gives the probability to correctly detect the absence of improvement given that the improvement does not exists at all.

- **Accuracy**: it is the ratio between the number of True Positive and True Negative over all the element in the dataset.

$$\frac{TP + TN}{TP + TN + FP + FN}$$

  This ratio explains the percentage of correct prediction over the entire dataset.

Moreover, for each instance, a ratio called **Improvement Presence Percentage (IPP)** has been computed. It stores the percentage of improvement found in the dataset over all the entire re-optimizations. The formula behind is the following.

$$\frac{FN + TP}{TP + TN + FP + FN}$$

This ratio has helped understanding for how many re-optimizations, in percentage, a real improvement has been found.

5.3. Analysis of the results

After executing all the procedure required in the Machine Learning framework, it is time to move on to the results. As already said, the results were incredible on the training dataset and on the validation dataset. When it came to the testing phase results, they showed a very interesting behaviour deserving a deep analysis. Again, the entire testing dataset, composed by four instances, has been split in four little datasets, each one including only one instance. This choice has been made to correctly analyse, step-by-step, every single instance in-depth.

Starting from the first instances, it had 6786 examples inside, this number comes out from the sum of each couple of parameters r and h available, multiplied by the times the whole set of re-optimizations has occurred. The confusion matrix and the computation of some useful ratios presented some interesting results. The confusion matrix below shows the values for each of the four categories. As it can be seen, the number of True Negatives is 2848, while the number of True Positives is almost half of it, and it is 1406. Then, the number of False Positives is 1124 and the value associated to the False Negative is 1772. Stepping into the analysis, it is worthwhile to mention that Improvement Presence Percentage is around 0.47. This means that only the 47% of the re-optimizations has led to an improvement.

The first ratio to be analysed has been the Sensitivity. It indicates the probability to detect an improvement in the situation where it exists for sure. In this case, the model presents a value of around 0.45. This number is affected by the big value associate to the False Negatives. This is, actually, not an incredible result for the testing phase. A greater sensitivity value would affect positively a possible future matheuristic with the ability to detect improvement where it would occur.

Then, it has been analysed the Specificity of the instance. It is a crucial indicator, and the result is very good. This ratio shows the ability of the model to correctly detect no improvements where they would not occur. The value is around 70%, meaning that, in a future matheuristic the predictor would correctly detect no improvements around 7 times out of 10. This could result in a great amount of time saved by the algorithm.

After that, the next indicator has been the Precision. It is around 0.36, meaning that, on all the correct predictions, only the 36% are associated to the improvement. This indicator is affected both by the low number of real improvements over the whole number of re-optimizations in the dataset and by the low level of Sensitivity. This means that, when predicting right, only 36% of prediction are associated to the improvement. This ratio can be considered as the IPP computed only on the right prediction. Clearly, if the IPP considering the whole dataset is around 0.47, it is unlikely that this ratio will exceed that value. However, this value is around three-quarter of the IPP, so it confirmed the fact that, for this specific instance, the model is better in predicting no improvement than when it has to predict the presence of it.

The final ratio considered is the Accuracy of the whole model, computed as the right prediction for both classes, over the entirety of the predictions. The value is around 0.57, meaning that around 60% of predictions are correct.

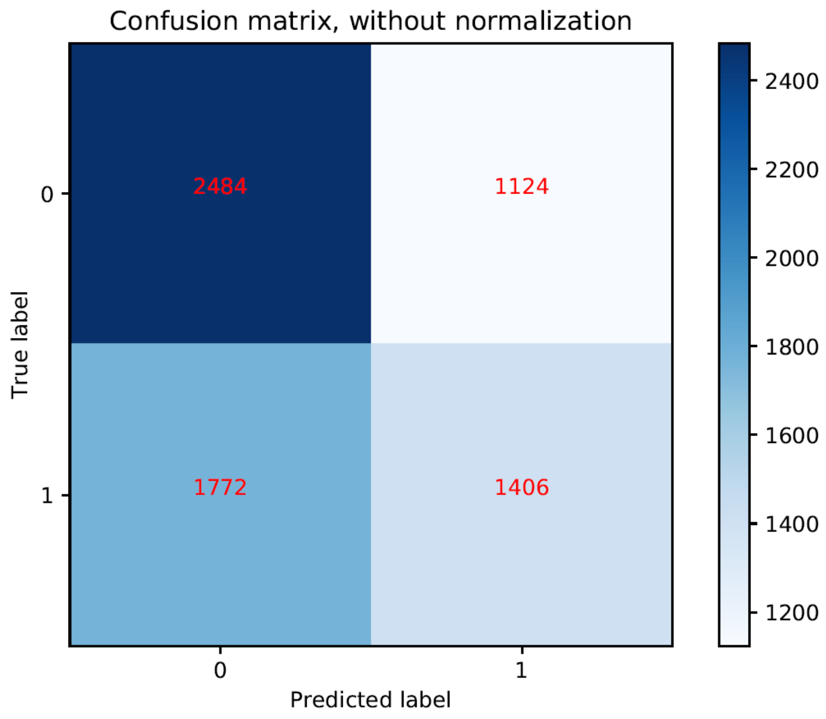


Figure 5.5: The confusion matrix related to the first instance.

IPP	0.468317123
Sensitivity	0.442416614
Specificity	0.688470067
Precision	0.361439589
Accuracy	0.573239022

Figure 5.6: The values of the ratios for the first instance.

The second instance has the same number of rows inside of the previous, meaning that the procedure did not present and early stop. In this case, the instance presents a great number of False Positives while a very low number of False Negatives. In facts, True Positives are 1907, True Negative 1562. When it comes to the mistakes, False Positives were 2919, while False Negative only 398. In this specific case, the Improvement Presence Percentage is 34%. Thus, this instance presents a very low number of improvements compared to the previous one.

In this case, despite the first instance, a great level of Sensitivity has been reached. This means that, in this case, when a real improvement has occurred in the dataset, the model has been able to detect it 83% of the times. Clearly this number is strongly affected by the low level of False Negatives, however, this is a very good result that can be considered as reliable .

The Specificity of this instance, instead, is not good as the first one. Since all the mistakes are localized in the False Positive category, this value is lower then the previous case. It is around 0.35. In a possible matheuristic, such a low value like this, would mean that the predictor will predict improvements where, actually, there is no room for it 65% of the time.

Following, Precision has been computed. Its value is around 54% and it is very interesting. Since it can be considered as an IPP applied to all the correct predictions, in this case the predictor was able to correctly predict more the situation presenting improvements. This led the predictor to have a high percentage value of improvements, with respect to the whole dataset. Clearly, this is also affected by the particular distribution of the mistakes in this instance.

At the end, every comes to the Accuracy. Even if the value is lower then the previous case, its value is still around 0.52%. Again, it is not satisfactory enough as the training phase, however this instance presents a very strange behaviour, with some remarkable points to be addressed for the construction of a predictor for the future matheuristic.

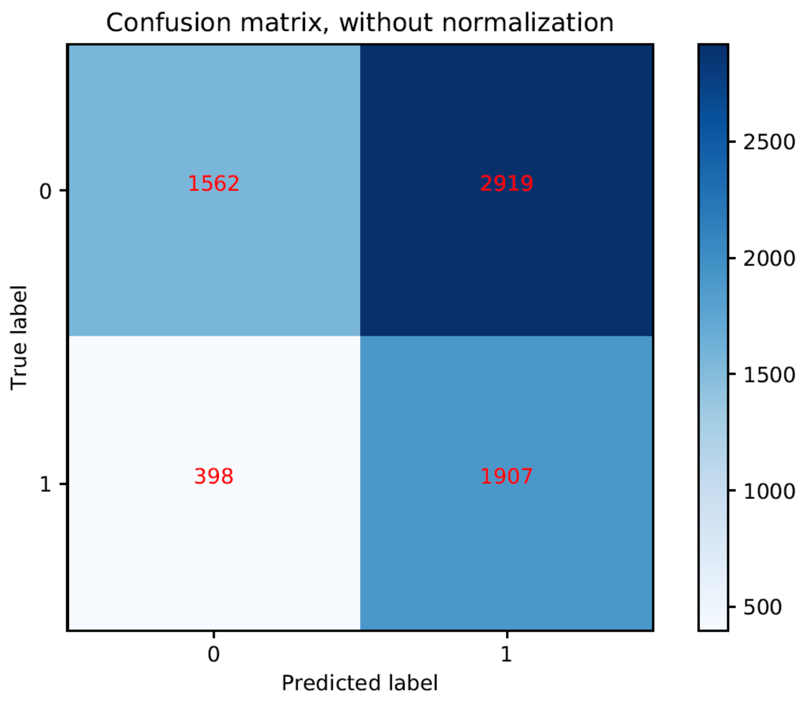


Figure 5.7: The confusion matrix related to the second instance.

IPP	0.339669909
Sensitivity	0.827331887
Specificity	0.348582906
Precision	0.549726146
Accuracy	0.511199528

Figure 5.8: The values of the ratios for the second instance.

After that, the third instance has been analysed. It stores inside the same number of examples of the previous two instances. This instance has been the most satisfactory one in terms of absolute value of correct prediction regarding the situation where improvements have been correctly detected. In facts, the number of True Positives is 2988 and the number of True Negatives is 1374. Instead, the number of False Negative is 1013 and the number of False Positive is 1411. This instance presented the highest value of IPP, it is around 59%.

The Sensitivity ratio has been very high also in this case, presenting a value of 0.75. This value has been affected by the highest value among all the instances of True Positives. The model, in this case, showed up a great ability to correctly detect the presence of improvement when it really occurred, not only in terms of percentage, but also in absolute terms.

Instead, the Specificity has showed a lower value compared to the first instance. However, in this case is even bigger then the previous instance. The value is around 50% meaning that the model has the ability to correctly predict no improvements half of the times.

When it came to compute Precision, again a good result has ben output. On all the correct predictions, the model has been able to correctly detect more times the situation where improvements occurred. Its value is 0.69. For the future matheuristic, this is a good result in term of predicting improvements where they should really occur, leading to successfully reaching an interesting number of consecutive re-optimizations.

With all this information, it has been computed also the Accuracy of the model. Despite the previous two, this model presents a higher value, being around 65%. This means that only the 35% of the prediction are wrong. This, together with the other ratios puts the model in a very good spot for its implementation in a future matheuristic.

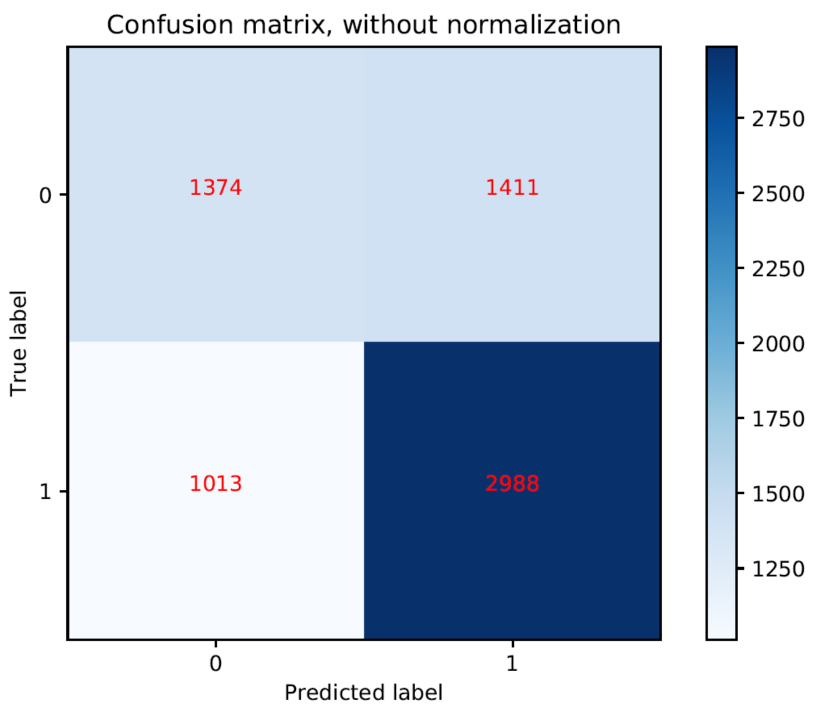


Figure 5.9: The confusion matrix related to the third instance.

IPP	0.589596228
Sensitivity	0.746813297
Specificity	0.493357271
Precision	0.685006878
Accuracy	0.642793988

Figure 5.10: The values of the ratios for the third instance.

The last instance, the fourth one, presented a very good behaviour in terms of correctly predicting situation where no improvements occur. This instance had the greater value of accuracy, mostly affected by this great ability. Again, the overall number of examples is always 6786. The number of True Negatives is 4338, while the number of True Positives is only 255. Instead, when it comes to False Positives, the value is 163 and False Negatives are 2030.

The IPP ratio for this instance is quite low as in the second instance and it is again around 34%. The low absolute number of correct improvements reached by the algorithm building the dataset, affected mostly the ability to the model of detecting no-improvements.

In facts, Sensitivity in this instance has been the lowest one, being around 0.11. This value is the less satisfactory one in the entire testing procedure. This means that, only 1 out of 10 times, the model is able to correctly predict an improvement. However, if the inability to correctly detect an improvement can be a weak spot for the model, it has been balanced by the following ratio.

Computing the Specificity, this case presented a great result able to balance the weakness came out with the previous ratio. Its value is the greater one, being around 97%. This means that, almost no errors have occurred when predicting the absence of improvements. In a future matheuristic, what could mostly affect the good success of the predictor would be its strength in avoid waste of time. This means that, a situation where a wrong prediction of no-improvement is preferred over a wrong prediction of improvements. More precisely, False Positive are worse than False Negative. This is because the algorithm will spend time when a False Positive will occur, while it will not when a False Negative comes out.

Due to the great number of correct predictions of absence of improvement, together with the fact that this instance presented very few examples where improvements occurred, the Precision is 5%. It is a low value, but it is mostly affected by the type of instance and by model strength. Its result has to be carefully interpreted to be sure to not consider it as a wrong situation.

This instance, among all, presented the greater value of accuracy. Its value is about 68% and for the distribution of results above discussed, is the most interesting and satisfactory one.

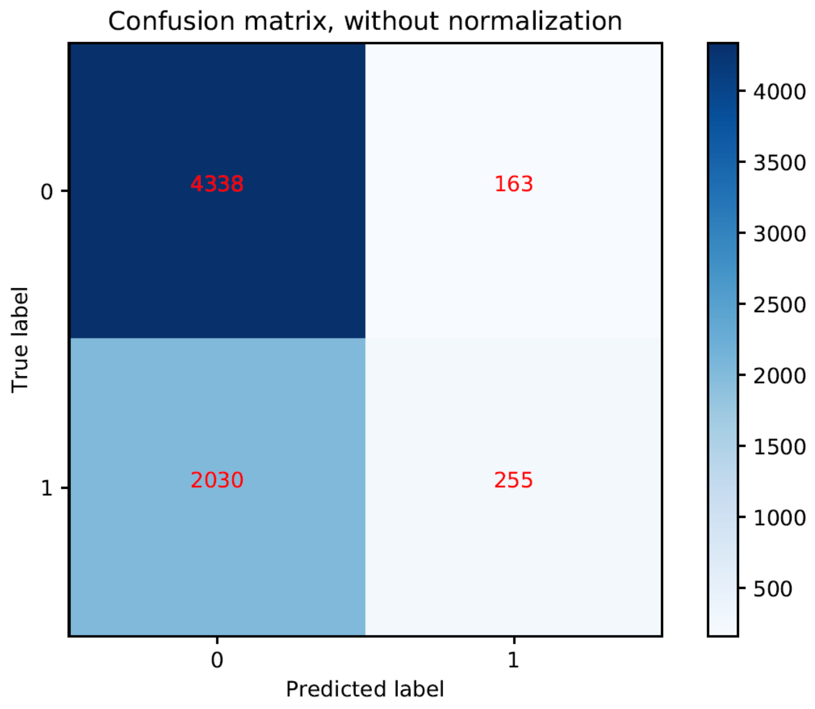


Figure 5.11: The confusion matrix related to the fourth instance.

IPP	0.336722664
Sensitivity	0.111597374
Specificity	0.963785825
Precision	0.055519268
Accuracy	0.67683466

Figure 5.12: The values of the ratios for the fourth instance.

So far, it has been discussed the analysis of each instance separately. Interesting ratios has been computed and deeply discussed. However, an overall analysis on the entire dataset can be fruitful to fully understand what the average behaviour of the model is. For this purpose, the same ratios have been used as well as the confusion matrix. However, a normalized confusion matrix has been also plotted to let things clear.

The testing dataset, in its entirety, stores inside the sum of all the examples of each of the four instances previously presented. This number is 27144 and it is divided among the four categories in this way. True Positives are 6556 and True Negatives are 9758. Instead, False Positives are 5617 and False Negatives are 5213. From the normalized confusion matrix can be seen that the percentage value associated to both True prediction is always higher than the value associated to the wrong prediction. This means that, the model makes more correct prediction than mistakes. Moreover, the entire dataset showed an IPP ratio of 43%.

The overall Sensitivity is about 0.56. This means that the model is able to predict a real improvement a bit more than half of the times. This result is good if compared to the fact that randomly picking a number gave it a probability of around 50%.

The Specificity computed over the entire dataset is 64%. This value confirms what has been seen during the analysis of single instance. The model performs well when it has to avoid useless re-optimization, predicting a correct situation of absence of improvement more than half of the time.

Precision value is around the value of IPP, showing a percentage of 40 points. This ratio means that among correct prediction, it maintains the same probability to detect an improvement.

Finally, the Accuracy is 60%. Even if this value was expected to be higher after the training phase, this does not mean that this result is not satisfactory. The model is able to predict correctly 6 out of 10 times, meaning that it can still overtake the randomness of flipping a coin.

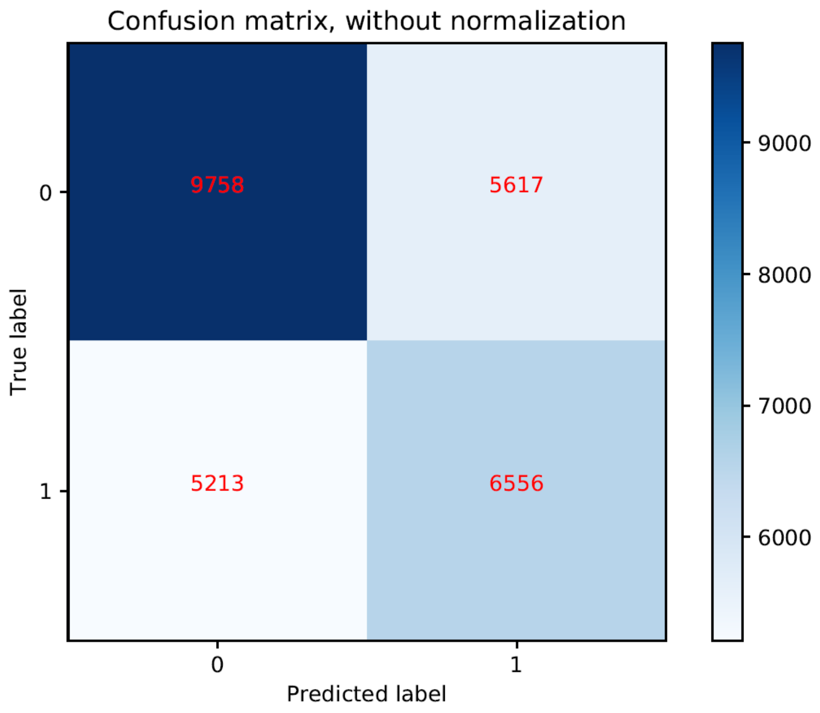


Figure 5.13: The confusion matrix related to the entire testing dataset.

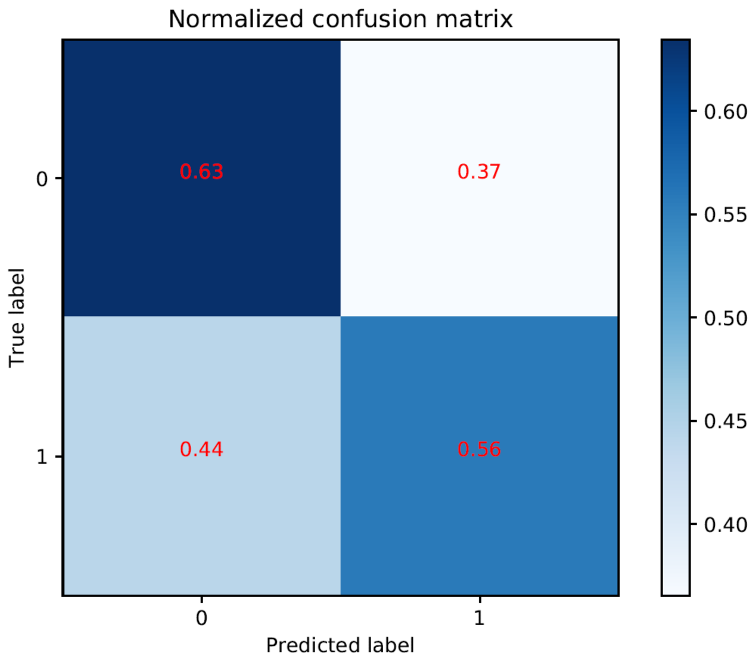


Figure 5.14: The normalized confusion matrix related to the entire testing dataset.

IPP	0.433576481
Sensitivity	0.557056674
Specificity	0.634666667
Precision	0.40186343
Accuracy	0.601016799

Figure 5.12: The values of the ratios for the entire testing dataset.

5.4. Interpreting the results

As it has previously explained, result of testing phase are quite distant of what was expected. Even if the model does not perform in the same way as during the training, some interesting features and characteristics have been highlighted in the previous section. In every instance, one or more ratios showed up an interesting behaviour, that could be useful for the future part of this work. However, before stepping into the further works that can follow this one, it could be fruitful to address some reason to the distance in the performance between the validation dataset and the test dataset. Since this work constitutes a completely new way to approach to Scheduling and more general to Combinatorial Optimization problems, after having worked for months on this kind problem, some interpretation can follow the presentation of the results.

One reason associated to the distance between the performance on validation dataset and the testing one, can be the following. Training and validation datasets presented the same instances inside. However, testing dataset present completely new instances. The model may have learnt how to solve those specific problems incredibly well during the training, spreading its knowledge also for the validation dataset even if it has not been trained exactly on this dataset. This let the model to not present strictly the classic situation of overfitting. However, when it came to predict on completely new problems, the model presented less accuracy in predicting a possible room for improvement in the sub-problem. This behaviour may be caused by several factors. One of those may be the difficulty to train a neural network to solve optimization problems like this. Since the objective of this work has been an attempt in merging two completely different approach, this may require more time and research effort to be exploited. However, the results still seem pretty interesting and they may be enhanced by further researches.

With this in mind, two scenarios can be explored. More problems can be started to be solved with Machine Learning techniques, inserting sort of predictors in matheuristic for solving difficult problems. The other one is a focus on specific scheduling problems, presenting similar features and characteristics as the one presented in this work. Following this way, the creation of new type of layers and neurons, dedicated exactly to perform these kinds of tasks would be a way to enhance the integration of the two fields.

Even if, at the time of writing, LSTM layers are very powerful in working with sequences and timeseries, a great way to improve the above results is to create a completely new type of layers that is able to work with job sequencing and scheduling. Usually combinatorial optimization, as in the scheduling case, may require consecutive interchanges of position before reaching an optimal solution or a satisfactory one. The model has been trained to understand the connection of the parameters inside every single sequence. However, the creation of a completely dedicated neuron may enhance the procedure. This neuron could be trained as well, but it should be able to figure out also the interchanges of jobs parameters inside consecutive sequences. After having worked on the job swaps, this neuron could be able to predict whether or not a sub-sequence will lead to an improvement, not only based on the sequence, but also using the information coming from the previous job position interchanges.

At time of writing, there were no time to inspect those factors. However, the obtained result can be also treated as a satisfactory one for the training part of the two-machine flow shop problem with three hundred and five hundred jobs. The working model reached in this work can be used as a starting point for the training part of both situations. It may be possible nearly no adjustment will require the new model to work.

5.5. The new matheuristic

Due to the lack of time, the predictor has not been tested inside the matheuristic to check if it is able to outperform the previous algorithm were parameters have been chosen always randomly.

As further works, it could be presented the implementation of the predictor in a testing matheuristic for the problem of one hundred job to see if it is able to outperform the previous matheuristic. One way to do this is by randomly choosing the position and the windows size parameter and asking to the predictor whether or not re-optimize. Both algorithms are now presented in the form of consecutive steps to show how a possible future matheuristic could incorporate the predictor. In Della Croce et al. (2014) a step-by-step procedure it is already presented, and it is the following.

```
❖ $x'$ = heuristic solution from the RBS stage
    o Repeat
        ▪ Set Improvement = False
        ▪ Repeat
            • Pick $r$ randomly
            • Compute $S'(r,h)$
            • Minimize
            • $x^*$ is the optimal solution
            • if $f(x') > f(x^*)$ then
                o $x' = x^*$
                o Set Improvement= True
            • End if
        ▪ Until improved or all $r$ values tried
    o Until not improved or time limit expired
```

Instead, using the predictor, the step-by-step procedure should be :

```
❖ $x'$ = heuristic solution from the RBS stage
    o Repeat
        ▪ Set Improvement = False
        ▪ Repeat
            • Pick $h \in \{8,...,20\}$ and $r \in \{1,...,n-h+1\}$ randomly
            • Compute $S'(r,h)$
            • Ask to predictor
            • if predicted improvement = 1
                o Minimize
                o $x^*$ is the optimal solution
                o if $f(x')>f(x^*)$ then
                    ▪ $x' = x^*$
                    ▪ Set Improvement = True
                o End if
            • End if
        ▪ Until improved or all couples of $(r,h)$ values tried
    o Until not improved or time limit expired
```

This procedure is the easiest one and takes both parameters regarding position and windows size completely random. Even if it has more steps inside, it could re-optimize more and focus only on promising re-optimizations.

Another way could be constantly increasing the parameter *h*. After the first stage, i.e. the RBS, since the solution is far away from a local optimum, it is more likely, even with a reduced window size, to reach an improvement. Moreover, in doing this, a time saving will occur, since reducing windows size will implicate a lower usage of time. However, after having re-optimize a satisfactory number of times, the windows size can be increased to perturbate the solution and to relocate it to new spot where a better solution could be reached. The algorithm should work as follows. The first solution comes from the RBS procedure. It starts with the parameter *h* set equal to 8. Then, the parameter *r* is chosen randomly, and the predictor is called. If the predictor outputs a possible improvement and the solver minimizes, then the parameter *h* is increased, and the procedure starts again. However, if the solver does not find an improvement or the predictor outputs that there is no improvement for that sequence, a counter will permit to use the same parameter *h* only for three times. This should help the procedure to rapidly move on values of windows size greater, that could ensure a better likelihood of finding an improvement. The step-by-step procedure is now presented and could behave as follows.

```
❖ $x'$ = heuristic solution from the RBS stage
    o Repeat
        ▪ Set Improvement = False
        ▪ Pick $h$=8
        ▪ Count=0
        ▪ Repeat
            • Pick $r \in \{1,...,n-h+1\}$ randomly
            • Compute $S'(r,h)$
            • Ask to predictor
            • Count=Count+1
            • if predicted improvement = 1
                o Minimize
                o $x^*$ is the optimal solution
                o if $f(x')>f(x^*)$ then
                    ▪ $x' = x^*$
                    ▪ Set Improvement = True
                    ▪ $h$=$h$+1
                o end if
            • else if count=3
                o $h$=$h$+1
            • end if
        ▪ Until improved or all couples of $(r,h)$ values tried
    o Until not improved or time limit expired
```

6. Concluding remarks

The purpose of the work has been the attempt of creating an algorithm that could merge the approach coming from two different fields. A static approach, where algorithms are coded step by step to perform a specific task comes from the Operation Research field. A more recent approach, were a computer can learn to perform a task without having been explicitly programmed to perform that specific task, comes from the Machine Learning field. Specifically, it has been used a neural network structure to cope with the possibility to create a new procedure for the resolution of a scheduling problem. The starting point has been a matheuristic coming from the article of Della Croce et al. (2014). There, the two-machine total completion time flow shop problem has been solved with a matheuristic able to re-optimize iteratively different small windows of fixed size of the whole sequence of jobs. In the article, the parameter associated to the window, i.e. the window size, has been set to a fixed value coming out from a mixture of trials and experience. The starting point of the sequence, instead, has been chosen randomly every time a re-optimization had to take place. In this work, the shaping and the training of a neural network, together with the creation of a predictor for the above exposed matheuristic has taken place.

The first step has been the analysis of the problem in its entirety, its similarities and its differences with the problem solved to the optimality by the Johnson rule, i.e. the minimization of the makespan in a two-machine flow shop problem. Then, the matheuristic presented in Della Croce et al. (2014) has been examined to fully understand how to possibly enhance the procedure. After that, this work has focused on the creation of the experience to be given to the neural network. A code, based on the original code of the paper, has been written to fully create the datasets needed. In this section, a time issue came out due to the lack of machines. An incredible amount of time would have been required by the software to run with only one machine, so it has been chosen to start a first evaluating phase for the problem with 100 jobs.

During this work, an interesting experience has been coding with Python and Keras. This work would have needed more time and effort if Python were not used. As already shown, the usage of this language is showing an increasing trend. Even if it is not recommended to be used in situation as the creation of the dataset, during the prototyping phase of the neural network, it has strongly helped and simplified the coding part. Modelling with Python, using Keras as the primary Deep Learning library let always the focus to be on the problem, and no worries have been addressed to code errors and bugs. This extremely easy coding environment is one of the most successful driving factors of the deep learning field, together with the incredible processing power that can be found even on home-end products.

After the dataset building part, the creation of the model has been the center of the work. Since the problem was completely new, there were no sources to be used. A glance to the machine Learning field has been given to fully come up with an idea on how treat the situation. Bidirectional Recurrent Neural Network, with LSTM cells have been used as the crucial core of the Neural Network. This kind of layer had shown consistency and reliability in previous works in Machine Learning field, so it has been chosen as the essence layer for the network developed inside this work. The problem has been treated before as a regression and then as classification. In the first approaches, a few little mistakes have been made due to the inexperience and the novelty of the procedure. However, after a long effort in the research of a working model, especially in the hyperparameter tuning phase, results came out and were very promising.

Then, the focus has been on testing the procedure just trained on a new set of experience. To do so, a new dataset has been created. A predictor has been coded and tested on this new dataset. Even if the model during the training did not present overfitting, it actually showed less efficiency and capacity when coming to predict on new instances. However, results are very interesting, and some point have been highlighted to fully understand the situation. Some instances were remarkable from the point of view of ratio values and behaviour. Most of the power has been centered in the ability of the model to predict the absence of the improvement. This is a reliable result that can be leverage on for the implementation of the predictor in the matheuristic.

Due to the lack of time, it has not been tested the performance of the new predictor against the performance of the matheuristic. However, two step-by-step procedures showing how it can be included to the algorithms to be applied further for a possible future matheuristic have been exploited. Further works may start from this point and move on applying this procedure to more Combinatorial Optimization problems. Even if, during this work, the neural network has been applied to a single problem, this procedure seems viable to be exploited even for other kind of problems. More difficult problems coming from the scheduling family and/or completely new problems in the field of Combinatorial Optimization seem to be a good starting point to extend this approach. However, a different direction may lead to the creation of special layers and neurons able to deal with this kind of problems.

Another interesting behaviour of the model can be explored to see if this is a characteristic that can be found over and over. Since the model did not present a strict situation of overfitting while it actually did a bit, can be examined. Applying the learning phase and the predictor to new Combinatorial Optimization situation can be checked if this behaviour still appears in different problems. This is an interesting point deserving further inspection and verification.

Furthermore, until now, neural network models have been used mainly for artificial intelligence solutions and used mostly by companies in private sector. An increasing in popularity of this kind of approach could attract more scientists, leading to incredibly good results in the future. However, at the time of writing, this seem far enough from the actual situation. From the writer point of view, with the hardware power obtained by technology, the power of deep learning models can still be explored and applied to the vast majority of situations. Instead, due to the lack of consciousness around these techniques, most practitioners see them as a black box, difficult to apply and to adapt. But, after a difficult phase of introduction in the field, things start to be clear and easy to understand. Then, everything is enhanced by an incredible network of people on Internet using deep learning. This community is permanently exchanging opinion on blog and specific websites. It is very likely to find the solution of a problem on the web. Moreover, with library as Keras and programming languages as Python, the approach to the field becomes even simpler. As a result, these techniques seem to have still uncovered power to be used for several purposes.

7. Bibliography

Baker, K. & Trietsch, D. (2009), Principles of Sequencing and Scheduling. Hoboken: Wiley

Bishop, C. (2006). Pattern Recognition and Machine Learning. New York: Springer Science+Business Media

Buduma, N. (2017). Fundamentals of Deep Learning. Sebastopol: O'Reilly Media.

Chollet, F. (2017). Deep Learning with Python. New York: Manning Publications Co.

Della Croce, F., Ghirardi, M., & Tadei, R. (2004). Recovering beam search: enhancing the beam search approach for combinatorial optimization problems. *Journal of Heuristics*, *10*, 89–104.

Della Croce, F., Grosso, A., & Salassa, F. (2014). A matheuristic approach for the two-machine total completion time flow shop problem. *Annals of Operations Research*, *213*, 67-78.

Goodfellow, I., Bengio, Y. & Courville, A. (2016). Deep Learning. Cambridge: The MIT Press.

Gulli, A. & Pal, S. (2017) Deep Learning with Keras. Birmingham: Packt Publishing.

Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory, *Neural Computation,9(8),*1735-1780.

Lawler, L., Lenstra, J., Rinnooy Kan, A. & Shmoys, D. (1993). Sequencing and Scheduling: Algorithms and Complexity in Elsevier Science Publishers(1993), Handbooks in OR & MS, Vol. 4 (445-522). Amsterdam: Elsevier.

Lutz, M. (2013). Learning Python. Sebastopol: O'Reilly Media.

M. R. Garey, D. S. Johnson & Ravi Sethi, (1976) The Complexity of Flowshop and Jobshop Scheduling. Mathematics of Operations Research 1(2):117-129.

Mitchell, T. (1997), Machine Learning. Boston: McGraw Hill Science/Engineering/Math.

Moore, G. E. (1965), Cramming more components onto integrated circuits, Electronics 38(8).

Murphy, K. (2012). Machine Learning, A Probabilistic Perspective. Cambridge: The MIT Press.

Patterson, J. & Gibson, A. (2017). Deep Learning, A Practitioner's Approach. Sebastopol: O'Reilly Media.

Pinedo, M. (2008). Scheduling. Theory, Algorithms and Systems, 3rd Edition. New York: Springer Science+Business Media

Raschka, S. & Mirjalili, V. (2017). Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn and TensorFlow, 2nd edition. Birmingham: Packt Publishing.

Shalev-Shwartz, S. & Ben-David, S. (2014). Understanding Machine Learning: From Theory To Algorithms. New York: Cambridge University Press.

Stroustrup, B. (2013). The C++ Programming Language, 4th Edition. Boston: Addison-Wesley

Tadei, R., Della Croce, F. & Grosso, A. (2005). Fondamenti di Ottimizzazione. Bologna: Esculapio.

8. Ringraziamenti

Il lavoro di tesi è praticamente ultimato, ed è ora il momento di passare ai ringraziamenti. Mi sembra doveroso, in prima istanza, ringraziare il Prof. Federico della Croce per l'opportunità datami. La possibilità di elaborare una tesi all'estero è stata un'occasione importante per la mia formazione e la mia crescita personale. Egli ha creduto in me sin dal primo momento, affidandomi un lavoro tanto interessante e stimolante quanto impegnativo. I suoi consigli, incredibilmente precisi e costanti, sono stati fondamentali per la buona riuscita di questo lavoro. In Francia, luogo dove parte di questo lavoro è stato elaborato, sono stato seguito e supervisionato dal Prof. Vincent T'Kindt e dal Prof. Romain Raveaux. Entrambi, con conoscenza, preparazione ed estrema gentilezza hanno fatto fronte alle mie mancanze di studente, seguendomi e aiutandomi costantemente anche durante la fase di elaborazione del lavoro svoltasi in Italia.

Finiti i ringraziamenti ufficiali, è ora di passare ai ringraziamenti meno formali, quelli di cui ho forse più ansia e dove non posso, e soprattutto non voglio, dimenticare nessuno. Non sono mai stato bravo con le parole, sin da bambino. Ho sempre preferito i numeri alle lettere e mi sono sempre comportato di conseguenza. Ad ogni modo, ho comunque apprezzato l'apporto e l'aiuto di quanti hanno speso anche una minima parte del loro tempo per dedicarlo a me durante questo percorso. Tre anni lontano da casa sono abbastanza per capire quanto la propria terra e la propria casa siano importanti nella vita di una persona. Parto proprio da questo, dalla casa, per iniziare i miei ringraziamenti.

Ringrazio la mia famiglia, per avermi supportato, e sopportato, con tutte le loro forze in questo periodo, anche e soprattutto, nei momenti di difficoltà. Per primo, ringrazio mio padre, per la sua estrema pazienza e la sua incredibile calma, perché in ogni situazione riesce sempre a trovare una soluzione rapida ed efficace, dimostrandosi spesso, più ingegnere di quanto lo sia io. A lui bastano poche parole in ogni occasione, e io che di parlare non ho quasi mai voglia, questo lo ho sempre apprezzato. A seguire, ringrazio mia madre, perché nei suoi piccoli e grandi gesti è stato facile notare la dedizione e l'amore che ha per la famiglia. Perché a suo modo esagera, esagera nel mettere cose nelle valigie, esagera nel riempire i pacchi del terrone, esagera per apprensione. Tutta questa sua esagerazione non è altro che amore in eccesso, del quale spesso mi sono lamentato anche se non avrei dovuto. Poi, ci sarebbe da ringraziare mia sorella, che in tre anni mi ha sempre aspettato con ansia e affetto. Con il suo fare sbarazzino, è sempre riuscita a sdrammatizzare i momenti più difficili, e anche quelli importanti. Perché forse ancora non sa "cosa deve mettersi" alla mia laurea, e sta già pensando a cosa indossare al mio matrimonio, di cui forse conosce persino la data. Senza di loro, questo traguardo non sarebbe mai stato raggiunto, lo considero infatti come un traguardo nostro e non mio, perché tutto ciò sarebbe stato impossibile, senza l'impegno, l'amore e la dedizione di tutti e quattro.

Finita la mia famiglia, è ora di passare ad un'altra famiglia. Una famiglia che mi ha accolto sin dal primo giorno e mi ha sempre trattato come se già ne facessi parte, da sempre. Mi riferisco ovviamente alla famiglia della mia fidanzata, nonché famiglia di uno dei miei migliori amici. Il dibattito qui è ampio, perché bisogna stabilire chi viene prima. Ma non vale la pena iniziarlo, perché sappiamo già chi vuole prevalere. Inizio con Eugenio e Giuliana, il loro aiuto e la loro presenza sono stati fondamentali nella mia crescita personale, soprattutto in questi tre anni lontano da casa. Non dimenticherò mai tutte le loro attenzioni, anche le più piccole, arrivate sempre nei momenti più importanti, quando io ne avevo più bisogno. Mai invadenti e al contempo sempre presenti, rappresentano per me una seconda famiglia. Poi c'è Luca, amico, cognato, fratello. Lui non lascia mai trapelare niente, un po' come me, ma in fondo mi vuole bene ed io ne voglio a lui. Non lo diciamo mai ed entrambi preferiamo sostituire le esternazioni di dolcezza con discussioni animate. Punta di diamante della famiglia, (Luca avrà qualcosa da ridire) c'è lei, la mia fidanzata. Simona meriterebbe una tesi intera, molto più lunga e dettagliata di questa, per tutto quello che rappresenta nella mia vita. Forte, decisa, intraprendente, ha sempre le parole e le soluzioni giuste. Non sono mai stato bravo né a dirlo, né a dimostrarlo, ma lei è capace di leggermi negli occhi e sapere già cosa penso e cosa provo. Sono un uomo migliore grazie a lei e tanti dei miei successi, passati e futuri, sono stati e saranno anche merito suo.

Ed ora, è tempo di passare agli amici, quelli veri, che conosci da piccolo e che non ti lasciano mai, quelli che sono sempre presenti, anche quando non ti vedono e non ti sentono. Ringrazio Totti, cugino e amico allo stesso tempo, sempre pronto ad aspettare che tornassi per dedicarmi del tempo, per chiacchierare e per scambiarci opinioni sui più disparati argomenti. Perché in un certo qual modo, la pensiamo uguale sulla maggior parte degli argomenti e questo ci rende più che amici. Ringrazio Luciano e Gianluca (Scaletta), anche loro fratelli più che amici, perché insieme non mi hanno mai fatto pesare la distanza, mi hanno fatto sentire sempre presente e vicino, soprattutto quando ero lontano. Non li ho mai ringraziati, ma sanno che ho apprezzato i loro gesti perché mi conoscono davvero. Ultimo, ma non meno importante, devo ringraziare Gennaro. Lui, è un fratello maggiore per me, mi segue, mi aiuta, mi guida nelle decisioni più difficili. I suoi consigli sono sempre perfetti e ha sempre del tempo da dedicare a me, nonostante sia super impegnato. Siamo fatti alla stessa maniera, per certi versi, entrambi sappiamo di esserci l'uno per l'altro e questo è la base della nostra amicizia.

In tante occasioni sono stato lontano e distante. Questa situazione è difficile sia per chi parte sia per chi rimane. Tanti sabati sera e domeniche, non ho potuto dedicare del tempo alle persone che volevo bene, soprattutto alla mia fidanzata. Ad ogni modo, c'è chi mi ha "rimpiazzato" e merita i miei ringraziamenti. A Mara e Paola insieme con Salvatore e Mario, perché hanno sopportato la mia Simo quando io non c'ero, e si sono sempre preoccupati di organizzare qualcosa tutti insieme per i miei ritorni. Qualcuno di loro, in questi anni, si è anche divertito a chiamarmi "torinese" e "polentone" e adesso sconta una "pena" in mezzo alla nebbia.

Ultimi, ma solo per ordine di età, ci sono due gioiellini che hanno cambiato il mio modo di essere in pochissimo tempo. Parlo dei piccoli Alessandro e Cecilia. La gioia che mi ha dato l'arrivo e la nascita di Ale è stata indescrivibile. Vedere i suoi primi passi e i suoi primi piccoli guai è qualcosa di veramente incredibile. Insieme, poi, ci sono le piccole emozioni che nell'ultimo periodo vivo ogni giorno con la piccola Cecilia. Tra un "bau" e un tiro di palla, ogni momento è un'occasione di gioco e di divertimento. Entrambi mi hanno ricordato di quanto ancora io sia bambino dentro, e di quanto mi piaccia averli "tra i piedi".

Tanti nomi mancano, tanti nomi di persone che mi vogliono bene e che, con una parola o con una piccola azione, hanno contribuito al raggiungimento di questo traguardo. Sarebbe difficile elencarli tutti, pertanto, questi ringraziamenti sono anche per loro.