POLITECNICO DI TORINO

Department of Mechanical Engineering

Master's Degree Thesis
(2019-2020)

Design optimization of Hybrid Electric Vehicles based on Deep Learning algorithms

Thesis advisors

Daniela Anna Misul

Claudio Maino

Alessandro Falai

Alessia Musa

Candidate

Alessandro Di Mauro

Abstract

Recent years have seen the flourishing of the so-called *green wave* and environment-related topics have been discussed in many ways for various reasons. The use of *Hybrid Electric Vehicles* (HEVs) is a valid way to achieve tank-to-wheel (TTW) CO_2 emissions reduction.

The choice of the design parameters, such as engine displacement or power of the electric machine, remains of fundamental importance. To this end, various algorithms have been deployed to effectively calculate the TTW CO_2 emissions of a specific HEV layout. One of this is Dynamic Programming (DP). However, it cannot always be used as it requires high computational power and time.

The main goal of this study is to develop an algorithm that can be used in the context of optimized design of HEVs. The tool to be developed should be far lighter than other deterministic algorithms such as DP and ensure comparable results at the same time.

The technology of choice is *Deep Learning Neural Networks (DNNs)*. It is a branch of machine learning so a part of the vaster field of *Artificial Intelligence*. This particular kind of algorithms mimic the behaviour of a human brain: various connections between different *layers* of *neurons* enable the flow of information and the possibility for the net to adjust itself and learn.

A pipeline of two *DNNs* is implemented to assess whether the vehicle will successfully complete the driving cycle of choice (feasibility), and in that case predict the TTW CO_2 emissions. The dataset available is composed by a set of different design parameters for HEVs. The pipeline is trained in *Supervised Learning*.

Promising results emerge from the study as the AI algorithm is able to produce feasibility predictions with an accuracy higher than 95%, and TTW CO_2 estimates with less than 1% error. The implementation is based on Keras and Tensorflow libraries.

Contents

List of figures

Introduction

Neural network theory overview

Data management

Neural networks based model

Results and discussion

Introduction

1.1. Motivations

The last few years have seen increasing interests in environment-related topics in both public opinion and national governments. As it is well known, more stringent regulations are declared almost every year to preserve the existing equilibrium and hopefully reverse dangerous trend [1]; it is our responsibility to comply with them. Among the aforementioned topics, vehicles pollutants emissions, and relative legislations, are surely a key point of the discussion. Direct consequence of these thematic are for sure important changes in private companies' guidelines, both for economic growth and regulations compliance. These changes are in turns the leading factor in the technological thrive we are seeing with respect to "*green technologies*".

As already stated, vehicle emissions are one of the most discussed point of the whole *green wave.* It is no surprise that the automotive industry is also one of the most affected ones. Conventional Internal Combustion Engine (*ICE*) vehicles are characterized by some well-known problems with respect to the abovementioned thematic. In particular, they emit pollutants (*e.g.: NO_x, CO, CO_2 and unburned hydrocarbons*) in order to produce the power required by the driver. All these chemicals are of course harmful to the environment. Moreover, it should be reminded that petroleum and its derivatives are surely not renewable.

Obviously, it is impossible to just abandon once and for all the automotive industry and all the advantages it assures to our society. All that being said, it is necessary to research and develop new and more refined technologies to comply with the environment needs, now more challenging than ever. The areas of research are extremely wide and include mechanical systems, control systems, pollutants reduction systems and many more fields such as hybrid vehicles, Hybrid Electric Vehicles (*HEV*) in particular. All those areas of development require precise and fast design analyses, fast prototyping, and extreme computational accuracy. In this scenario, the aid of automatic tools during the design phase is essential and the related advantages are very clear.

1.2. Technological solutions

The need to comply to more and more stringent regulations in terms of vehicles emissions really stretched the areas of research in which modern companies are working. One of the most prominent and promising field of development in this sense is surely electric traction [2].

When talking of electric traction, a simple distinction to well define the scenario is possible: *Battery Electric Vehicles (BEV)* and *Hybrid Electric Vehicles (HEV)*. As the name suggests, the first category achieves traction by a battery pack and an electric machine; the specifications are based on the particular type of *BEV* and the expected mission it is supposed to endure. These vehicles are mostly appreciated because of their capability of achieving zero emissions *Tank-To-Wheel (TTW)*; they surely represent a viable option to drastically reduce local pollution in big urban aggregates. Their limitations are however as evident as their strengths: the large battery packs introduce

problems in terms of space and weight management; their reduced autonomy range is not always enough, especially for extra-urban missions; this problems is also amplified by the hours-long charging time; finally, batteries disposal is today a not completely solved issue that needs to be addressed quickly. *HEVs,* on the contrary, use both *ICE* traction and electric traction. The general system is obviously more complicated, and this is one of the main drawbacks of these technologies as it means higher costs. However, this approach solves the issues related to the autonomy range thanks to the contemporary use of the electric motors and the *ICE*. Moreover, it anyway ensures lower emissions, especially in urban conditions where the electric traction really shines, and of course lower consumption [3].

HEVs in particular can be classified on the basis of their specific architecture. Two main categories arise, namely *Series* and *Parallel HEVs.* Series-HEV represents today a niche field of application; this is because of the nature of the architecture: the *ICE* is only connected to the electric motor via an inverter, and its only function is to produce electric power to charge the battery or to power the electric motor. This means that the *ICE* is not directly connected to the wheels, so that the electric motor must comply with the entire power request. Moreover, the *ICE* cannot operate in charging and powering mode at the same time: this requires the knowledge of the mission in advance with fair precision, otherwise the vehicle will be forced to use the *ICE* only to propel himself (usually the *ICE* is small) or to stop and use it to charge the battery-pack. On the other hand, parallel-HEVs can obtain driving power from the *ICE* and the electric motor at the same time. Both "*power-lines*" converge into a *coupling device* that allow to sum the torques coming from the two power sources. This approach to *HEVs* is currently the most commonly used by car manufacturers as it ensures a wider range of application. Notice that further classifications of parallel_HEV are also possible on the basis of the position of the electric motor (e.g.: P2, P3 and P4) It should be mentioned that "*complex architectures*" also exist, namely *series-parallel-HEVs,* that combines the advantages of the two categories.

1.3. Control logics overview

With the term "*Control Logic*" we are referring to the strategy upon which the vehicle decides what to perform at a particular moment in time. It is of key importance when talking of HEVs power management: a good control strategy is essential to achieve the best performance possible and fully exploit the capabilities of this technologies. In general, a *control logic* takes as input some parameters related to the state of the vehicle (e.g.: speed, road gradient, power requested by the driver) and can give as output choices regarding, for example, the gear to be selected, the power split between *ICE* and electric motor, the necessity to enter in battery charging mode and so on. These parameters are usually chosen by the control logic to achieve better fuel consumption or better pollutants emissions.

Different *control logics* have been developed to comply with different needs: time and computational limitations, the need to find a true optimal solution or the possibility to implement the logic on an actual vehicle. Three families of control logics can be identified: heuristic strategy, static optimization methods, and global optimization methods [3].

As it is known, a *heuristic* approach is based on the concept of "*good-enough*" decisions, based on the occurrence of certain events. *Fuzzy logics, Neural Networks,* and *Rule Based* strategies belong to this category. In general, they perform a specific action based on predefined control variables.

Obviously, they are not capable of generating true optimal solutions, however they are extremely light from a computational point of view and can be easily implemented *on-board*.

Static optimization refers to the concept of instantaneous optimization of the *equivalent fuel consumption*. The state variables of the vehicle are transposed into an *equivalent amount of fuel consumption* via a predefined formula and the controlled parameters are changed to minimize it. "*Equivalent Consumption Minimization Strategy*" (*ECMS*) is part of the category. Since this strategy minimize the *equivalent fuel consumption formula* for each time instant, so not considering the whole of the mission, it cannot guarantee a true optimal solution. There is however the possibility to implement them *on-board.*

Global optimization methods are the only ones that can ensure a true optimal solution to a specific problem. They consist in algorithms able, given the entire mission, to find a control strategy that guarantee the minimum of a specific function (the function may be consumption-related only or based on both consumption and pollutants emissions). As already stated, they need the entire mission in order to compute a solution, so they are not implementable *on-board*; moreover, they are very heavy from a computational standpoint. *Dynamic Programming* and *Genetic Algorithms* are examples of global optimization methods. In particular, *Dynamic Programming* will be dealt with more in details in the following chapter since it represents a starting point of this project. The *Genetic Algorithms* approach consists in a strategy that, starting from an initial population (various combinations of controlled parameters), are able to find a single individual that represents the optimal solution. They simulate the natural "selection process" through the use of a scoring system that evaluates each individuals of the population, assessing their probability to generate other similar individuals in successive generations: higher the score, higher the probability.

1.4. Technical choices

The aim of the present project is to provide an efficient and effective tool to be used during the design phase of an HEV. The tool will give valuable information regarding the best design parameters, with respect to fuel consumption and CO_2 emissions, regarding a specific mission. In particular, this project represents the extension and refinement of an existing tool that has been proved to be able to predict the emission of an HEV starting from its design parameters. The aforementioned tool uses *Deep Learning* algorithms to achieve this goal, so an algorithm of the same nature is used to ensure compatibility between the two projects.

Deep Learning (DL) is a specific branch of *Machine Learning* which sits in the wider field of *Artificial Intelligence [4].* A *DL* algorithm simulates the biological brain, its connection and information flows. It can be visualized by a series of interconnected "*Layers*", each one consisting in a series of "*Nodes*". Each node can be thought as an element in which an information enters, is modified, and finally leaves, heading towards the next element. The term "*Learning*" comes from the fact that these algorithms are able to adjust themselves through a process of *training* called "*Back Propagation*". They are extremely well suited for highly nonlinear problems. It is possible to define two big families of algorithms: *Supervised* and *Unsupervised Learning Algorithms.* In the first category, the algorithm receives both the data and the solutions, and it should train itself on those solutions. In the *Unsupervised* algorithms instead, no solution is given to the software that is free to elaborate the data as it thinks its best. *Deep Learning* algorithms are a particular type of *Supervised Learning* algorithms.

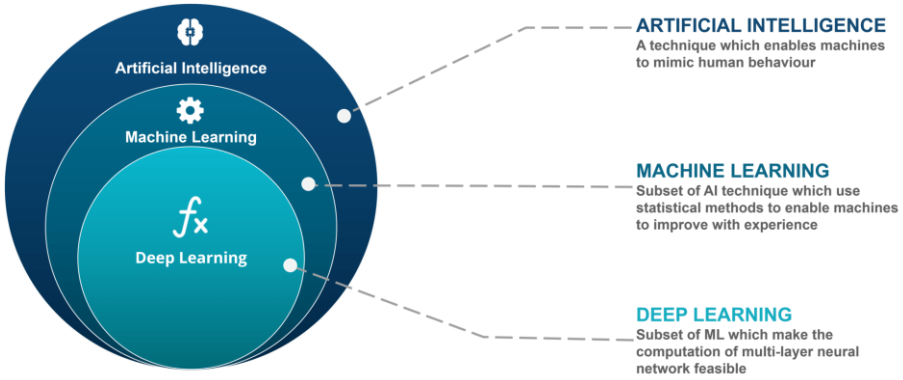


Figure 1. Structure of A.I. theories.

The main difference between *Deep Learning* and "standard" *Machine Learning* lies on the fact that the first one is characterized by a more complex structure. This type of architectures is proved to be able to better *learn* from big amount of data. It is no surprise that modern companies try to exploit, as best as they can, the concept of "*Big Data*".

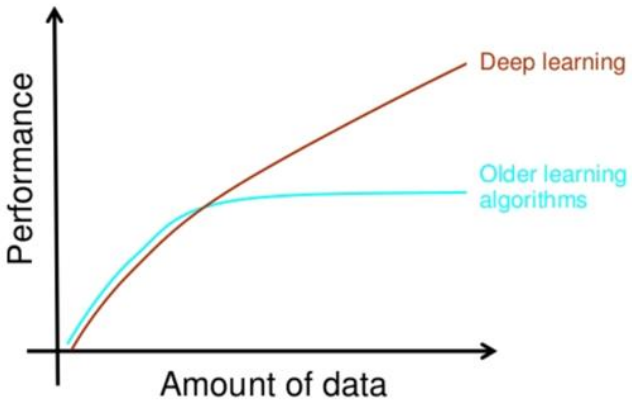


Figure 2. Deep Learning performance behaviour].

The learning process for the present project is based on the results of an optimized algorithm of the *Dynamic Programming (DP)* type The reasons that have led the choice upon a *Deep Learning* algorithm revolve around one concept: even though it is extremely precise and reliable, *DP* is highly time consuming and require very good hardware capabilities to perform sufficiently well. On the other end, if well trained, an algorithm based on *Deep Learning* can produce god enough results in an extremely shorter time window. Moreover, as it will be demonstrated and already mentioned in *Figure 2*, Deep Learning algorithms can positively exploit an increase in the training dataset. This will actually be an important point of the present dissertation.

Should be mentioned that, based on their core intention, a further classification in possible for this kind of algorithms. As a matter of fact, *Supervised Learning* algorithms have usually two main goals: *classification* or *regression*, we define them *predictive*. *Unsupervised Learning* methods are instead used for *clustering* or *anomalies detection* and are usually described as *descriptive.*

Figure 3. Supervised and Unsupervised Learning common use.

1.5. Project contribution and dissertation outline

As already stated, the present project lies in a wider research that has the aim to produce an efficient tool able to assist the operator in realistic application during the design phase of an HEV. The pre-existing tool, form which this project starts, effectively predicts tailpipe emissions of an HEV starting from its design parameters, regarding a specific driving cycle. *Deep Learning* is the technology of choice for the aforementioned reasons and the starting datasets are originated from a *Dynamic Programming* tool.

From a very general point of view, the *DP* algorithm achieves two goals at the same time: it computes the emissions for the optimal sets of design parameters and, in doing so, it distinguish between configurations that are able to complete the predefined cycle (referred to as *feasible examples/configurations*) and configurations that are not (referred to as *unfeasible examples/configurations*). The pre-existing tool rely its training phase on *feasible examples* only since its aim is a *predictive regression*. From this situation arises the need for a *classification* algorithm that is able to predict in advance which examples are *feasible* and which are not. It is clear at this point that such an implementation is a sort of filter to be applied one step before the regression process. Obviously, since the two algorithms should communicate smoothly, the working environment is the exact same: the code is written in Python, using Anaconda Spyder as working environment and Tensorflow as backend. Keras is instead the main library used to develop the net.

In the present project, not only is presented the *classification* part of the code, it is also analysed the *pipeline* composed by the two algorithms developed and the relative compound results. Notice that different architecture of HEV will be dealt with but, due to reasons that will later be clear, only one architecture, namely P4-HEV, will be further investigated. The dissertation will start from an overview on the *Neural Network* theory and its core features; it will then proceed to explain the structure of the datasets and their management; finally the logic of the actual algorithm will be explained and the results showed and discussed.

Future developments should include the possibility to train the nets here presented on a dataset that also contains information relative to various driving cycle, this will ensure a wider range of application and a more powerful tool. Moreover, different type of architectures (only *fully connected nets* are here analysed) should be also tested in order to assess the most reliable and accurate one.

Neural network theory overview

2.1. History and modern scenario

Even though *Artificial Intelligent* seems to be a very modern concept, with respect to public opinion, the idea to design a machine that can mimic the human behaviour is older. The first concept comparable to the aforementioned idea is probably the *Touring Machine* and the relative studies of its inventor Alan Touring. Anyway, the first real effective effort in designing an actual *Neural Net* is without a doubt "*Perceptron*", presented by Frank Rosenblatt in 1958. It was characterized by only two layers, input and output, but it relies on the concept of *Error Back Propagation*, that is the solid foundation for all the modern application of this fascinating technology. However, it is only in 1986 when a "*Multi-layer Perceptron*" was presented, featuring and intermediate layer, that this field really started to thrive.

Unfortunately, even though it was showing promising results, it failed the test of the field application. This was not because of lack of potential or flaws in the theory behind it, but it was due to technological limitations related to the hardware available at that time. The new millennium solved the problem presenting to the public more and more advanced microprocessors.

From this point on, new achievements and results never stopped arriving. In 1998 the first *Convolution Neural Network* makes its appearance. This particular type of net is the foundation of all the image recognition software based on A.I. nowadays. Actually in 2012 AlexNet performs with excellent results this task, winning ImageNet [5]. Notice that form approximately 2015, the average performance of the best performing softwares is better than the human performance in that specific competition. Image recognition is not the only field in which machines have outperformed humans, just consider the game of chess for example.

The fields of application can be however very different from what the imagination propose: from faces recognition to generation of fictitious visage, from intelligent digital assistants to public interests guessing and monitoring. A completely new field of research started from the "simple" ideas of Touring and many companies related to those arguments are today thriving. Some of the most resonant names are of course Google, Facebook, IBM, and Amazon. However, many others smaller companies or even start-ups are creating this innovative substrate over which next generations will be born [4].

The private sector is surely not the only one interested in this revolution of thinking. Academic researches regarding *Artificial Intelligence* applications are flooding the scientific journals and completely new academic courses are forming.

It really seems a field where the only limitations are *"self-imposed"*. For sure soon after the technological progress follows the ethical and public debate.

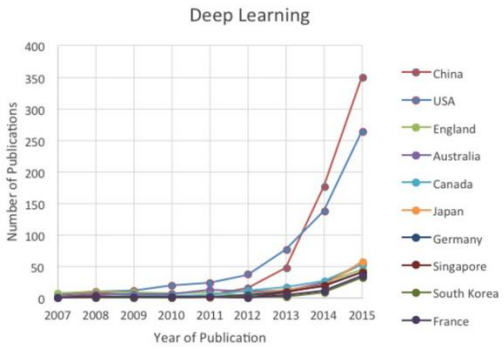


Figure 4. Deep learning related publications per year.

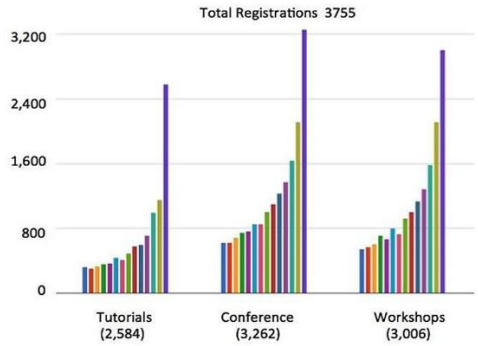


Figure 5. Meetings and material per nations.

Notice that, regarding the private sector, not only new companies are being born having their core business in *Artificial Intelligence,* but also companies of different extraction are implementing *AI* in their business model. They understand the potential of such a technology and aim to exploit it.

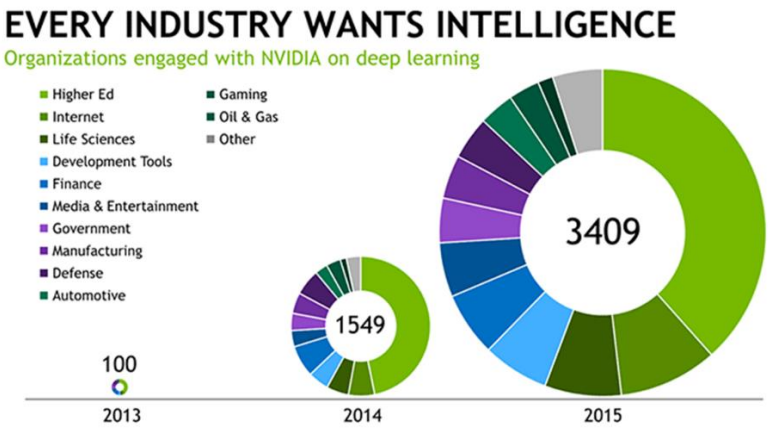


Figure 6. Every industry wants intelligence.

2.2. Neural Network main features

Deep Neural Network structures can be very different from each other on the basis of their aim and the resources/data at disposal [6]. However, they always show some common features. For the purpose of this project, it is sufficient to describe predictive models basic working flow. To provide a sufficient overview, it is convenient to follow the aforementioned working flow.

As already stated, predictive models need a training dataset that is equipped with the exact solutions for the *examples* proposed. Such solutions will be referred to as *labels*. Starting point of the whole training-predictive process is actually data management and manipulation. This aspect will be discussed separately in the following chapter.

Once the datasets are ready, the training phase can start. It always consists of two distinct process, a *forward-pass*, which will produce an output, and a *backward-pass*, that is responsible for the "adjusting" part of the training phase. During the *forward-pass*, the information coming from the dataset, namely the entries of the dataset, enters the *layer* of *neurons*. These are two key parameters of every type of neural network. Generally speaking, a *neuron* can be thought as an element in which the input value is transformed by a predefined function, the *activation function*, and then heads towards the next *layer*. Different *activation functions* have been developed to try and solve various issues; they will be analysed in detail. When this process reaches the last layer, an output is produced. The *output layer* varies in structure based on the purpose of the aim: *regression* neural networks will have only one *neuron* since they produce a single value as output; *classification* nets usually possess a number of output *neurons* equal to the number of classes they have to distinguish. Obviously also the type of data being outputted is different.

At this point the results, right or wrong, are available, we refer to them as *predictions*. Using *labels* as mean of comparison it is possible to compute a value which gives indication on the *predictions'* accuracy. Such value is computed with the use of what is known as *Loss* Function. Notice that the term "accuracy" is now indicating a general performance index; more precise terminology will be introduced, referring to specific performance indices. The *Loss Function* is the foundation of the learning algorithm since on its minimization is based the learning process. Since the *Loss Function* is based on the *predictions* of the net, it is clear that different types of output need different *Loss Functions.* Moreover, *regularizers* are used to "force" the *loss function* to behave in certain ways, avoiding *overfitting* for example.

Key feature of the learning algorithm is also what is known as *learning rate*. In general, it regulates how "large" the changes are during the adjusting phase of the learning process. This is probably the parameter that has the greatest impact on net's performance.

The features abovementioned are usually referred to as *hyperparameters*. This term is used to distinguish them from the parameters that the net automatically adjusts during the learning process, namely the *weights*. The *hyperparameters* also need to be adjusted, however they are not automatically set by the learning algorithm itself. Different kinds of *hyperparameters tuning procedures* exist (e.g.: *grid search, random search, Bayesian Optimization)*. *Hyperparameter tuning* is probably the most important part of a *Deep Learning* application since it can greatly influence predictive performance.

2.2.1. Activation functions

A single *neuron* with *Single Input Single Output* (*SISO*) structure can be represented as in Figure 7.

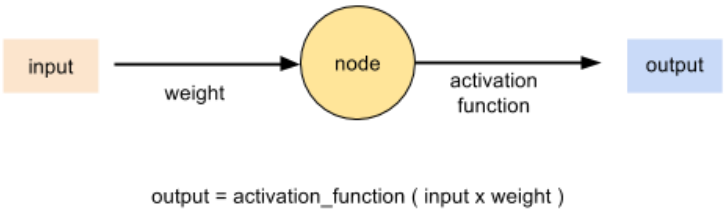
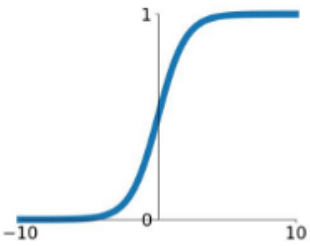


Figure 7. Representation of a single neuron. SISO system.

Notice that usually the structure of a node is of *Multi Input Multi Output (MIMO)* type, creating the so called "fully connected" layout that will be deepened later.

As already mentioned, different kinds of *Activation Functions* have been studied and tested [7]. The first applications saw the use of *Sigmoid* function and *Hyperbolic tangent* function. These functions have a limited "image", mathematically speaking (Figure 8, Figure 9). This can be a benefit or a limitation: if the function is thought for the output layer of a classification net, their results can be easily interpreted as one class or the other (two border of the function's image). However, if used for the inner layers of a deep neural network they can cause troubles: for very high (or very low) input values, the output is very similar even for very different values. Modern theory saw the increasing use of "ramp" function. They can effectively boost the learning process by always ensuring different output values for different inputs; most commonly used ramp functions are *Exponential Linear Unit (ELU), Rectified Linear Unit (ReLU)* and *Leaky*-ReLU (Figure 10, Figure 11, Figure 12).

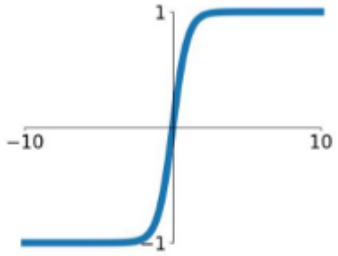
Sigmoid Function



$$\sigma(x) = \frac{1}{1 + e^{x}}$$

Figure 8. Sigmoid activation function.

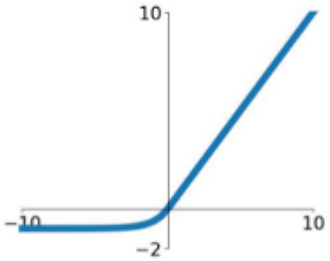
Hyperbolic Tangent Function



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

Figure 9. Hyperbolic Tangent activation function.

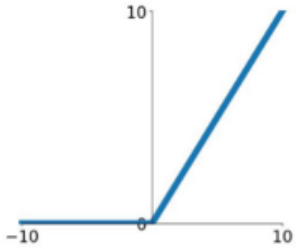
ELU Function



$$\sigma(x) = \begin{cases} \alpha(e^{x} - 1) \; for \; x < 0 \\ x \qquad for \; x > 0 \end{cases}$$

Figure 10. ELU activation function.

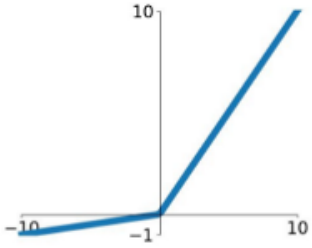
ReLU Function



$$\sigma(x) = \begin{cases} 0 \quad for \; x < 0 \\ x \quad for \; x > 0 \end{cases}$$

Figure 11. ReLU activation function.

Leaky-ReLU Function



$$\sigma(x) = \begin{cases} 0.1x \quad for \; x < 0 \\ x \qquad for \; x > 0 \end{cases}$$

Figure 12. Leaky-ReLU activation function.

2.2.2. Fully connected architecture

For the present project an architecture of the type "*Fully connected*" is selected. As already introduced, it is a particular type of *MIMO* system in which each node (*neuron*) is connected with all the nodes of the previous and the following *layer*. This approach, as visualized in Figure 13, gives an intuition on the name of choice for this kind of algorithms: *Neural Networks.*

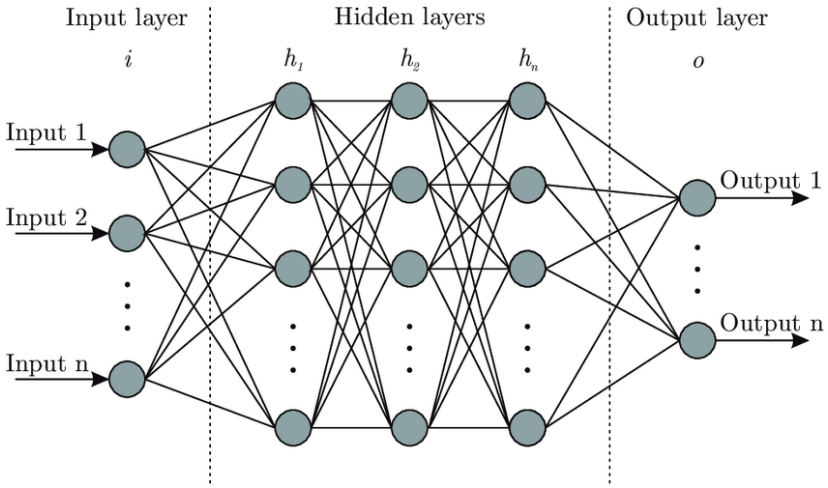


Figure 13. Neural network visualized.

Notice how each *neuron* relates to the previous and the following ones. From the figure are also noticeable the *Input Layer* and the *Output Layer*. Their structure, namely the number of nodes in these layers, is dictated respectively by the number of *"features"* by which each example of the dataset is composed and the number of outputs we ask the net to produce. Referring to Section 1.5, this project deals with two fully connected deep neural networks: a *Classification Neural Network (cDNN)* and a *Regression Neural Network (rDNN)*. Even though the dataset is basically the same, the outputs are different in type and purpose: the *cDNN* has two output neurons (since it choose between two classes), the *rDNN* has instead a single neuron output layer (since it predict a single value). The nature of the *Input Layer* will be deepened later when explaining the dataset.

When referring to "connections" in neural networks we are referring to sum operations: each node takes as input the sum of the contribution of all the neurons in the previous layer, each multiplied by a *weight*, sums it up with a *bias* term and the compute the output through some sort of *activation function*. Those *weights* and *biases* are the parameters on which the learning algorithm operates, increasing or decreasing them, to learn from its errors.

Increasing the net dimensions, number of *hidden layers* and *neurons* for each layer, the learning potential of the net increases. This concept, even though it looks fairly straightforward, not always translates into real performance improvements, it could actually be a source of issues as we will see later. Notice that, for the purpose of this project, an automated search for the correct number of layers and nodes has been developed.

2.3. Training algorithm

It has already been introduced that the learning phase is in two steps: a forward and a backward pass. These two steps take the name of *Forward Propagation* and *Back Propagation.* In this chapter will be explained the mathematical model that composed this type of procedure. Notice that during the *Forward Propagation* no parameters adjusting is performed. This step serves the only goal to produce an output, it is the predictive side of the algorithm. Since the *Back Propagation* needs a prediction to start, the forward pass should come first. It is clear that some sort of initialization for the *weights* is needed. Different types of initialization have been tested in the literature and some of them will be introduced in this project.

Without entering in too many details, notice that the dataset available for the training procedure is only a fraction of the whole dataset. Some of the examples of the complete database are preserved to perform the evaluation of the neural networks. Two evaluation steps are of key importance: the *validation* and the *testing*. They will be explained in detail in the following chapters. Anyway, each fraction of the dataset is composed by two main part: the one containing the *example features (X)* and the one containing the *labels (Y).*

2.3.1. Forward propagation

The process by which a neuron's output, relatively to a specific layer "*i*", is generated can be effectively described by the following mathematical expression:

$$\begin{cases} x_k = \sum_{j=1}^{m} \left(w_{k,j} * z_j\right) + b_k \\ z_k = \sigma(x_k) \end{cases}$$

Where:

- x_k: input of the k^{th} node of layer *i*
- m: number of neurons of layer *i-1*
- $w_{k,j}$: weights related to the connection between node *k* (of layer *i)* and node *j* (of layer *i-1*)
- z_j: output of the j^{th} node of layer *i-1*
- b_k: bias value for node *k*
- $\sigma()$: activation function
- z_k: output of node *k* of layer *i*

For a whole layer "*i*" we obtain a system that is the following:

$$\begin{cases} z_{1,i} = \sigma(x_{1,i}) \\ z_{2,i} = \sigma(x_{2,i}) \\ \vdots \\ z_{n,i} = \sigma(x_{n,i}) \end{cases}$$

$$\begin{cases} x_{1,i} = b_{1,i} + z_{1,i-1} * w_{1,1} + z_{2,i-1} * w_{1,2} + \cdots + z_{m,i-1} * w_{1,m} \\ x_{2,i} = b_{2,i} + z_{2,i-1} * w_{2,1} + z_{2,i-1} * w_{2,2} + \cdots + z_{m,i-1} * w_{2,m} \\ \vdots \\ x_{n,i} = b_{n,i} + z_{n,i-1} * w_{n,1} + z_{n,i-1} * w_{n,2} + \cdots + z_{m,i-1} * w_{n,m} \end{cases}$$

So now is clear that:

$$\begin{cases} z_{1,i} = \sigma(b_{1,i} + z_{1,i-1} * w_{1,1} + z_{2,i-1} * w_{1,2} + \cdots + z_{m,i-1} * w_{1,m}) \\ z_{2,i} = \sigma(b_{2,i} + z_{2,i-1} * w_{2,1} + z_{2,i-1} * w_{2,2} + \cdots + z_{m,i-1} * w_{2,m}) \\ \vdots \\ z_{n,i} = \sigma(b_{n,i} + z_{n,i-1} * w_{n,1} + z_{n,i-1} * w_{n,2} + \cdots + z_{m,i-1} * w_{n,m}) \end{cases}$$

It is best practice to write the system in matrix form, the system is then:

$$\begin{bmatrix} x_{1,i} \\ x_{2,i} \\ \vdots \\ x_{n,i} \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,m} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,m} \end{bmatrix} * \begin{bmatrix} z_{1,i-1} \\ z_{2,i-1} \\ \vdots \\ z_{m,i-1} \end{bmatrix} + \begin{bmatrix} b_{1,i} \\ b_{2,i} \\ \vdots \\ b_{n,i} \end{bmatrix}$$

It reduces to:

$$\bar{x}_i = \boldsymbol{w} * \bar{z}_{i-1} + \bar{b}$$

And finally:

$$\bar{z}_i = \sigma(\boldsymbol{w} * \bar{z}_{i-1} + \bar{b})$$

Notice that $\bar{z}_i$ and consequently b are vectors with dimension "nx1", with *n* being the number of neuron of the i^{th} layer; $\bar{z}_{i-1}$ is a vector "mx1", with *m* being the number of neurons of the layer *i-1*; finally $\boldsymbol{w}$ is a matrix with dimension "nxm".

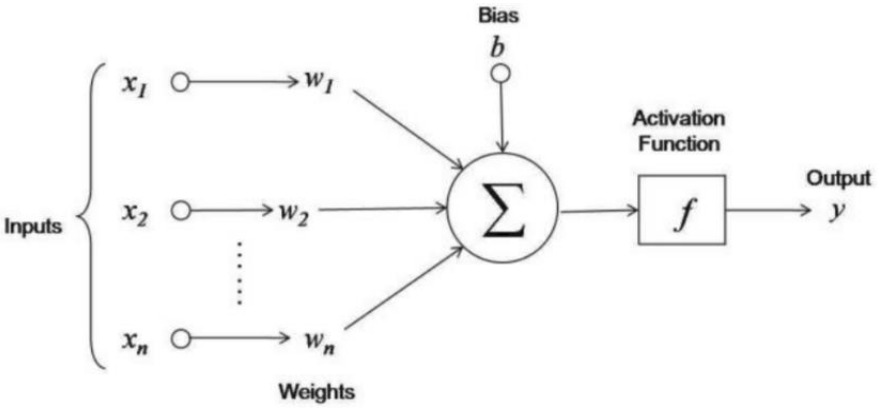


Figure 14. Forward propagation.

2.3.2. Backward propagation

This is the beating heart of the learning process. As a reminder, notice that here the algorithm is using the training set only. Should be clear by now that each example in the dataset produces an output which dimensions are dictated by the specific application. The *Loss Function (L)* however calculates a "loss-value" for each entries; all these values need to be combined in a single quantity, assessing the performance of the net on the whole set (or a portion of it). To combine all the losses another function is used: the *Cost Function (J)*.

If we indicate the predictions as "$\hat{y}$" and the labels as "y" we can summarize the concept above as follows:

$$Loss = L(\hat{y}, y)$$

$$Cost = \frac{1}{l}\sum_{i=1}^{l} L(\hat{y}_i, y_i)$$

The quantity "*l*" refers to the number of examples analysed in a single pass. Different approaches are present in the theory. The most general one and the first one to be implemented, and probably the best choice for very small dataset, is the procedure to consider the entire dataset. A more modern approach is to consider a portion of the dataset for each pass or even only one example. More detailed information are given below.

Since *Cost* and *Loss Functions* depend on *predictions,* so *weights* and *biases*, and *labels*, and since labels are constant throughout the learning process, we can conclude that both functions finally depend on weights and biases.

$$Cost = J(w, b)$$

The key point of the *back propagation* is to minimize the function "*J*".

All the algorithms here analysed are based on the concept of "*Gradient Descent*" [8]. As the name suggests, the procedure relies on the computation of the *Gradient* of the *Cost Function*; notice that function *J* has usually a high number of variables. The *Gradient* gives information regarding the "direction" of the steepness of the function. Using an iterative approach, the algorithm searches for the minimum of the cost function, moving in the direction indicated by the gradient.

The general formulation of the algorithm can be written as:

$$w_{n+1} = w_n - \alpha \nabla_w J(w_n)$$

The formulation is usually more complicated than this, based on the choice of the chosen *Optimizer*. Parameter "α" is of great importance, it represents the so-called *Learning Rate*. It can be thought as the "distance" covered by each iterative step. The choice of the *Learning Rate* is without a doubt one of the most important step in the optimization phase; however, since it is very difficult

to assess a single value for the whole process, modern optimizer modify it in order to adapt it to the specific situation.

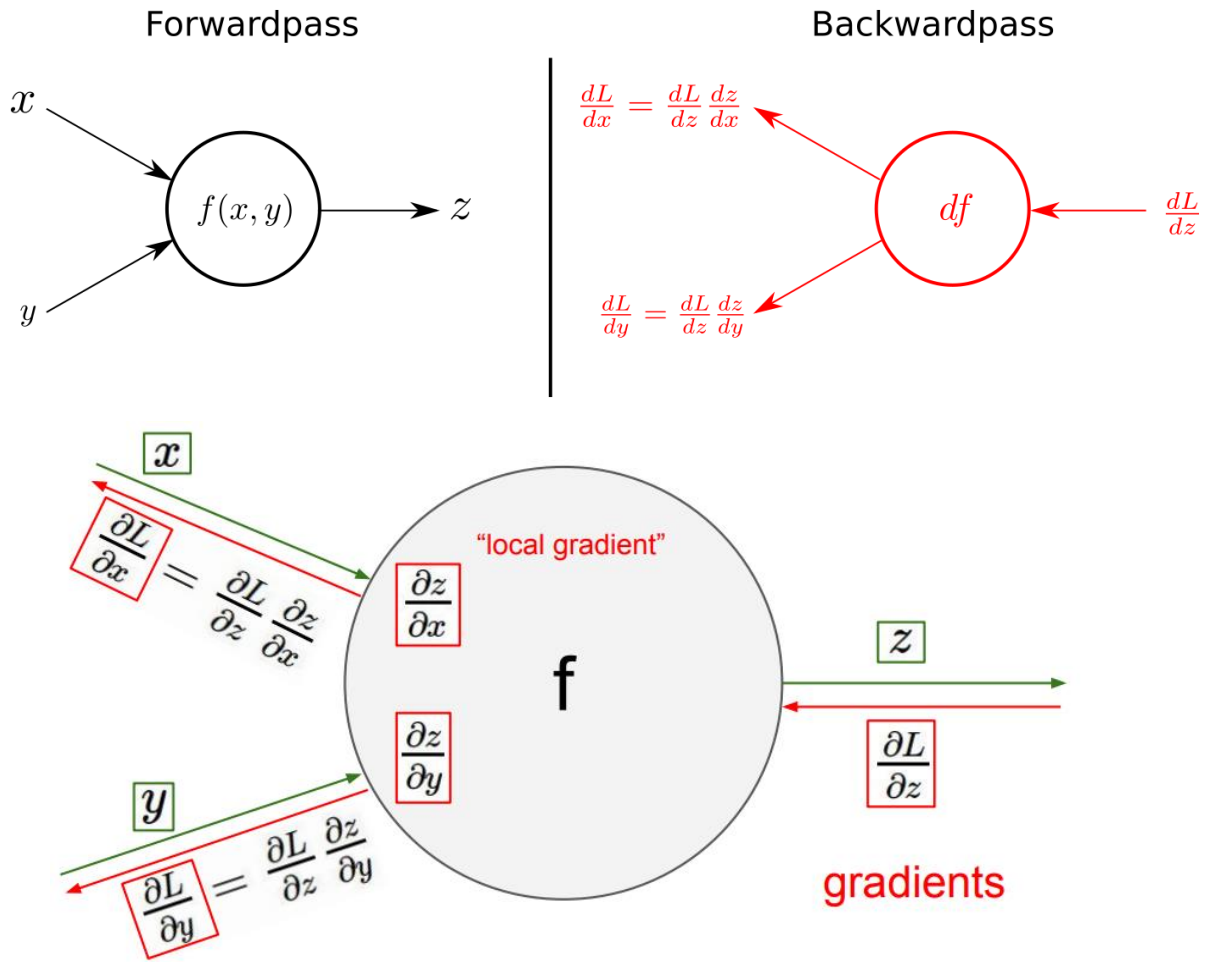


Figure 15. Forward and Backward Propagation visualized.

Notice how, during backward propagation, it is available the gradient of the *loss function with respect to the output:* $\frac{\partial L}{\partial z}$. However, what is needed is the derivative of the *loss function* with respect to the parameters to be adjusted, the weights and biases. This is achieved by product of derivatives, as depicted in Figure 15. The *loss function* is differentiated several times during this process.

Once the partial derivatives with respect to each weight are available, the adjusting process takes place with a formulation that in general is:

$$\begin{cases} w_{i+1} = w_i - \alpha * \dfrac{\partial \mathrm{L}(\mathrm{w},\mathrm{b})}{\partial \mathrm{w}} \\ b_{i+1} = b_i - \alpha * \dfrac{\partial \mathrm{L}(\mathrm{w},\mathrm{b})}{\partial \mathrm{b}} \end{cases}$$

The partial derivatives with respect to each weight will represent the quantity $\frac{\partial L}{\partial z}$ for the subsequent step of the *backward propagation*. The iterative procedure goes on as explained, adjusting the weights to "descent" along the *cost function* in the direction of maximum steepness.

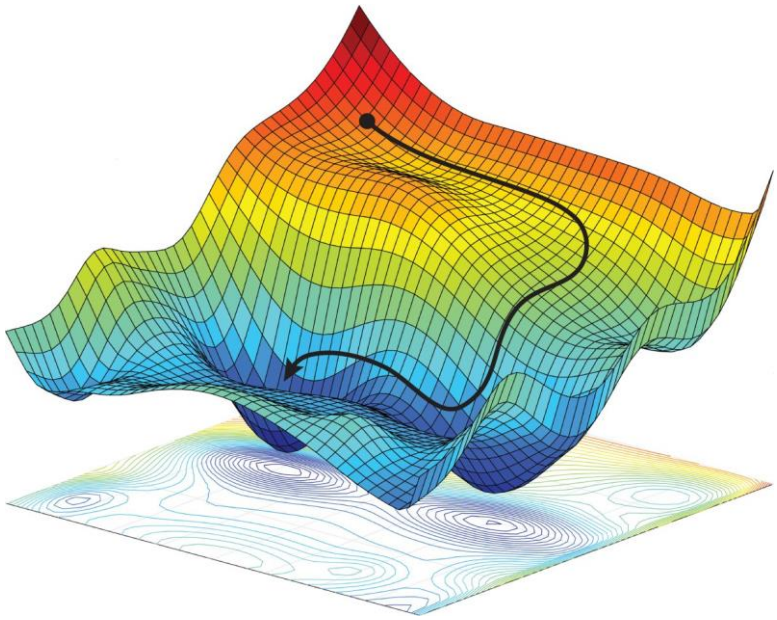


Figure 16. Gradient Descent visualized.

As already stated, the number of examples processed in one pass is not standard and it actually depends on the type of optimizer chosen. For completeness, notice that each time an algorithm goes through the whole training dataset, this is called an epoch.

2.3.3. Validation

It is common use to evaluate the net after each epoch to monitor the performance on a set of examples that the net is not trained on [10]. This practice enables the operator to see if the learning procedure is going smoothly. The procedure is called validation and the dataset used takes the name of *Validation Set*. As the training set, also this one is composed by two parts: *examples* and *labels*. Notice however that the *validation set*, or better, its predictions are never back-propagated. The sole role of the validation process is to monitor the performance throughout the learning procedure, not to adjust the weights and biases.

As we will see later, on the basis of validation results, different architectures are compared, and the best-performing is selected. Since we are choosing a particular architecture on the basis of its results, validation performance cannot be taken as a reference for future field applications.

2.4. Loss functions

It has already been introduced that different goals for a neural network means different *Loss Functions* [11]. This project manages a pipeline of two *deep neural networks*, the first one operates a binary classification action and the second one performs a regression. It is then no surprise that the two DNNs have separate *loss function*.

The *rDNN*, already developed and available at the beginning of this project, uses the well-known *Root Mean Square Error (RMSE)* function. It is described by the following expression:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

The *RMSE* gives valuable information regarding the fitness of predictions with respect to the *true labels*. Moreover, it has one important features: since its value depends on the square of the error, it is sensible to outliers. In the context of this project it was important to avoid strong outliers: a slightly wider point cloud is to be preferred to a narrower one that shows a strong outlier.

The *cDNN*, which development is a contribution of the present project, implements instead what is known as *Binary Cross Entropy* function. It is a particular case of the more general *Cross Entropy*. The formulation is as follows:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i)$$

Looking at the possible values of y_i and $\hat{y}_i$ it is clear the behaviour of the function:

- y_i: *true labels* of the n examples. There are only two possible values: 0 or 1.
- $\hat{y}_i$: *predictions*. The possible values vary from 0 to 1 continuously.

In the summation term, there are two part, only one at a time will be "activated": the first one if the *label* is 1, the second one vice versa. The logarithmic term, ideally, should be 0: this would means a perfect prediction. Obviously this will not be the case; it is however reasonable to think that in most cases $\hat{y}_i$ values will be similar to the *labels* so the logarithmic function is used to produce a value near to zero if the prediction is similar to the *labels.*

Notice finally that *Cross Entropy* computed with this formulation gives always negative values. This is dealt with in different ways, the most common and practical ones are two: ignore the issue and develop a code to maximize the cost function instead of minimize it; or simply add a minus to the formulation.

2.5. Hyperparameters

The *hyperparameters* are the main governing features of the behaviour and the structure of the net itself. The term is used to distinguish them from the *weights* and *biases*, usually called parameters of the net. The predictive performance of the net are strictly related to the choice of the *hyperparameters*, so the process by which they are selected is of fundamental importance. Notice that even a slight change in one of their values could drastically change the results of the prediction process or the learning capabilities of the net.

However, it should be reminded that, even though a single *hyperparameter* can influence the performance, acceptable results are achievable only by analysing the overall behaviour of the particular combination of *hyperparameters*. This concept means that, in the context of field applications, one-dimensional analyses should be only the starting point for the optimization of the net. It is far more important to test different combinations of *hyperparameters* than make one of them vary, keeping the other constant.

When referring to these features it can be distinguished between "*structural*" and "*optimizer-related*" hyperparameters. Type of architecture (e.g.: *fully connected, convolutional o, recursive*), number of *internal layers,* number of *neurons* per layer are examples of *structural hyperparameters.* They influence the very structure of the net, and in turn the shape of the cost function. The second category is instead related to the behaviour of the *optimizer* of choice. It includes *learning rate, epochs, regularizer, batch size, optimizers, and initialization*. Notice that any particular characteristic of the net the designers wish to let vary and analyse can be considered a *hyperparameter.*

With reference to Figure 16, notice how the cost function is a two-dimensional function is a three-dimensional space. This is an extreme simplification. Should be clear by now that the cost function depends on all the weights and biases present in the net:

$$Cost = J(w, b)$$

To have a two-dimensional cost function it would mean that only two parameters are present, and the resultant net will be an extremely simple one since the weights stand also for the number of connections.

Not only the function is not that simple, there are also more than one cost function to be analysed since it depends on the hyperparameters. The resulting problem of minimization is therefore extremely vast.

2.5.1. Number of layers and neurons

As already stated, these two parameters are the driving factors for the structural shape of the net. Theoretically speaking a single layer neural network can approximate any continuous function. This statement holds true because of the *Universality Theorem* [12] which explanation is beyond the scope of this project. More recent studies however proved that an increasing number of layer (increasing *depth* of the net) should guarantee a better learning potential. This is true especially when dealing with big data.

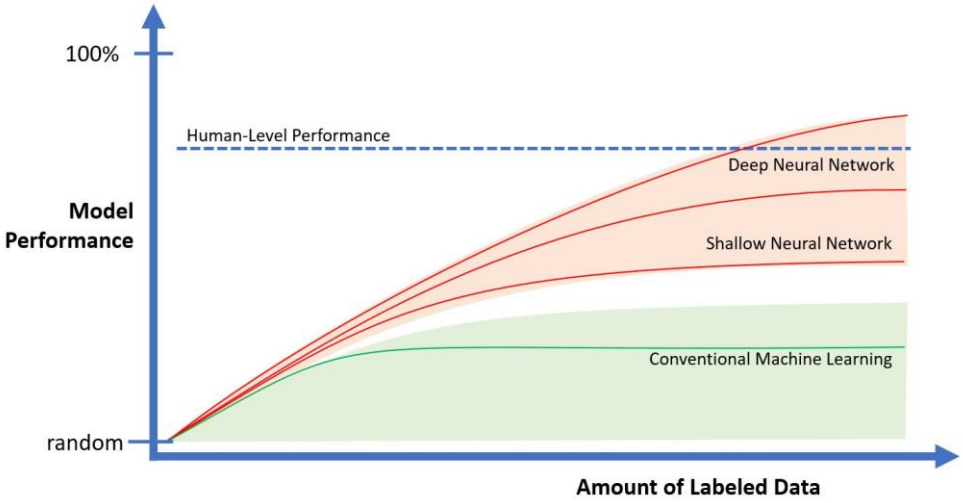


Figure 17. Shallow and deep neural networks performance curve.

2.5.2. Learning rate

Learning rate is probably the hyperparameter by definition. It is the one parameter around which gravitate most of the concept of neural networks as they are here presented. As above mentioned, it represents the amplitude of the descending step along the cost function following the gradient. In general, a higher learning rate produces a faster convergence due to the increased step size. However, a big step is not always desirable, it can in fact cause stability issues and sensitivity to outliers; it can even cause diverging behaviour: it is however rare, since a diverging learning curve is very evident and the possible causes are only a few with a too high learning rate being the most probable. A smaller learning rate is of course more stable and less sensitive to outliers but, due to the decreased size of the step it needs more iterations to converge to the minimum of the cost functions. In addition, a too small learning rate could get the algorithm stuck in a *local minimum*, with the gradient not being able to point in the correct direction.

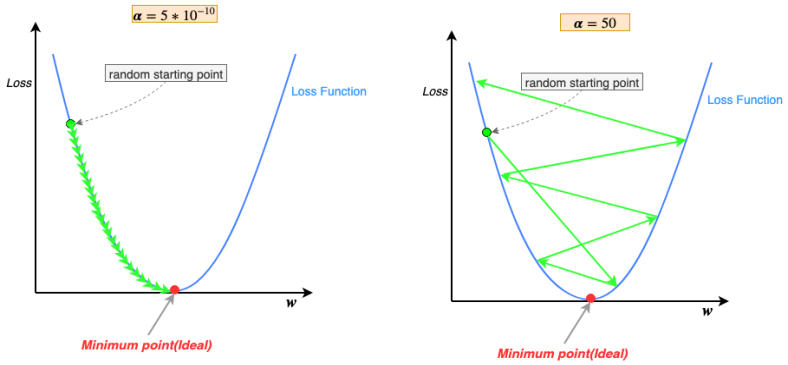


Figure 18. Low and High Learning Rate.

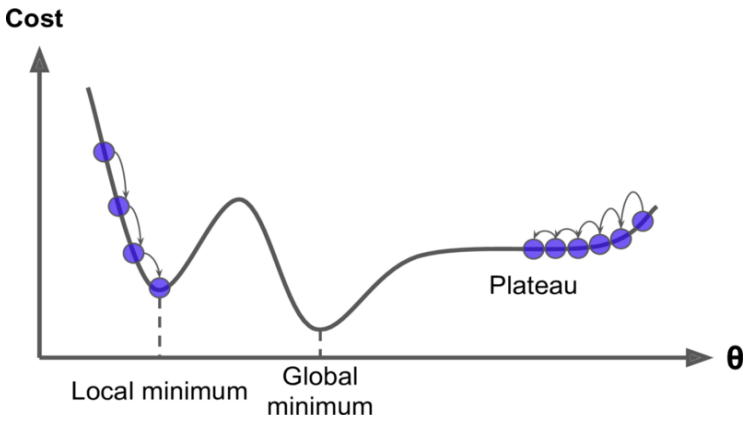


Figure 19. Local minimum issue.

An interesting point of view is achieved by looking also at the *Learning Curves* of a specific algorithm. This is actually the main instrument by which developers monitor the behaviour of the net. They give valuable information regarding not only the performance but also the integrity of the learning algorithm.

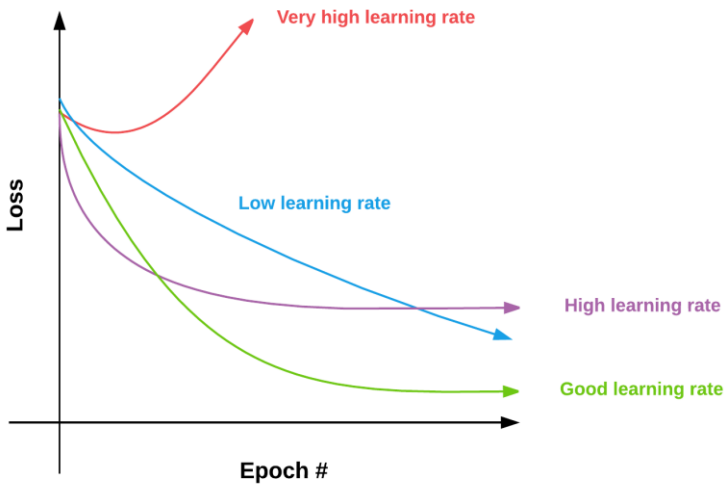


Figure 20. Learning curves for various learning rate amplitudes.

2.5.3. Batch size

With the term *Batch Size* we refer to the number of *examples* processed in a single pass, both in forward and backward propagation. This concept, crucial during training, is also valid during strictly predictive phases, however it does not influence the results of the prediction in this case.

Referring to *Section 2.3.2*, the cost function is actually calculated on a number of examples that is the *batch size*. The result, and the ensuing changes to the weights, are done on the basis of those example only. For clarity, let us resume the logical steps:

1. *Forward Propagation* with examples' features
2. Computation of the *Cost Function* using the *true labels*
3. *Back propagation* and weights changes

On the basis of the actual numerosity of the batch, we distinguish:

- *Full Batch Gradient Descent:* it uses the whole training set for every single pass. This means that each iteration corresponds to an *epoch* of the learning process. This approach is the only one guaranteed to find the minimum point (assuming no local minima are present) because every step of the descending process take into accounts all the examples, so the gradient always points in the direction of maximum steepness. However, one strong limitation is the time needed to process all the examples in one pass. This issue is magnified when the training set is very vast. The resulting learning curve is usually very smooth.

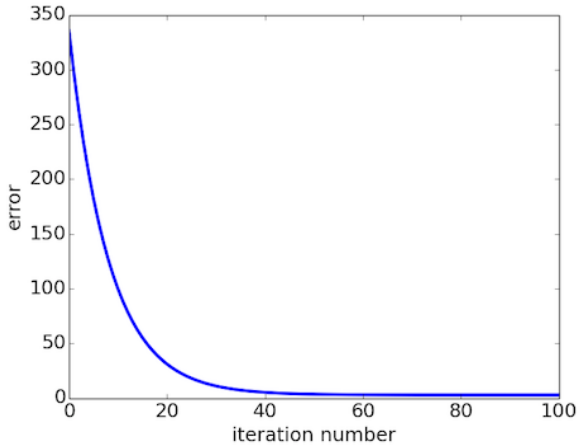


Figure 21. Full batch gradient descent learning curve.

- *Mini Batch Gradient Descent:* this approach is based on the concept for which a good enough update of the weights can be achieved using only a limited amount of training examples. It is clear that, since the cost function is computed on a portion of the training set, the gradient in this case does not point in the direction of maximum steepness. It can happen, and it does happen, that the cost function increase after a pass, but the general trend is of course decreasing. In this case one iteration does not coincide with one epoch. This approach is usually more rapid, even though it cannot guarantee the optimal solution to be find. However, due to the possible wrong steps that could happen, it is best practice to reduce

the learning rate as the process advances, so as to avoid too large step in the incorrect direction. This is referred to as *Learning Rate Decay* and it can represents another *hyperparameter*. As the *batch size* tends to the whole training set size, the curve becomes smoother and smoother as the process tends to the full batch gradient descent.

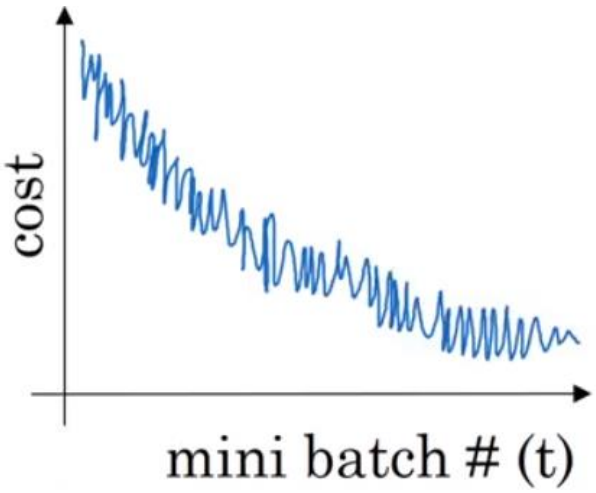


Figure 22. Mini batch gradient descent learning curve.

- *Stochastic gradient descent:* without loss of precision it can be considered as a *mini batch gradient descent* with only 1 example for each batch. The updates are very frequent, although very imprecise. This leads to an extremely erratic exploration of the *cost function* that usually translates in higher computational time and worse results in term of learning performance.

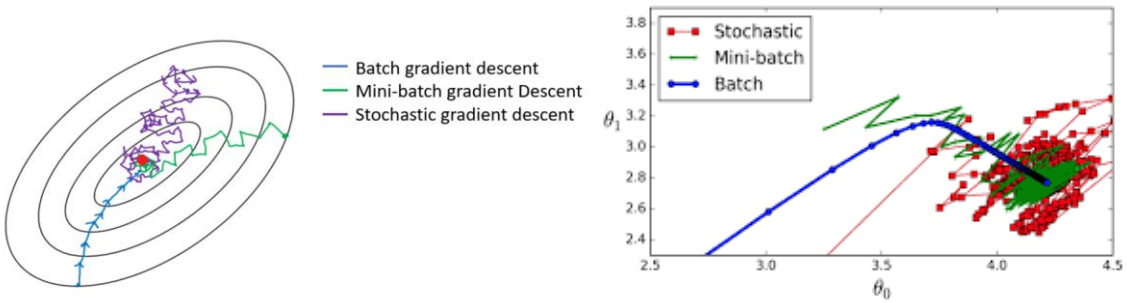


Figure 23. Full batch, mini batch and stochastic gradient descent.

2.5.4. *Epochs*

Number of *epochs* defines how many times the learning algorithm goes through the whole training set. In general it should be high enough to give the net the opportunity to learn properly, so to avoid *underfitting*, but no so high to cause *overfitting*. These are two possible issues that can happen during the learning phase. It is important to notice that the number of *epochs* is not the only parameter that intervene in this behaviour. Net complexity, so *depth* and number of *neurons*, also plays an important role: the higher the learning potential of the net, more likely is the *overfitting* to occur.

The analysis of the learning curve is again of fundamental importance to spot and fix *fitting* problems. The comparison between training loss and validation loss, as we will see, gives valuable information.

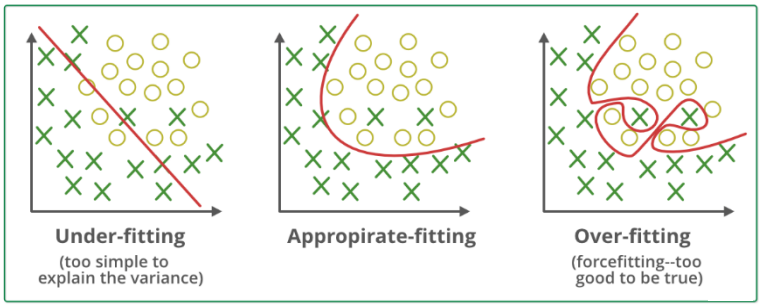


Figure 24. Under fitting and overfitting for classification tasks.

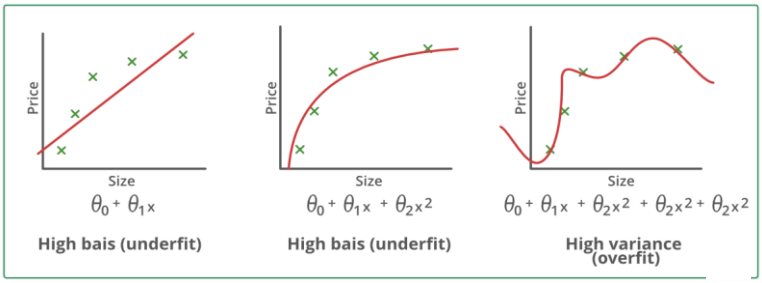


Figure 25. Under fitting and overfitting for regression tasks.

Let us now analyse the two situations:

- ***Overfitting:*** in general a net is said to be overfitting when it performs incredibly well on the training set, even perfect fit is achievable, but fail to generalize to never seen before examples. Referring to Figure 24 and 25, a classification net for example could come up with extremely complex boundaries to include all the training example, but those borders does not apply decently to the whole problem. Similarly, a one dimensional overfitting function could increase the degree of its polynomial shape to pass through each training point, however this does not mean that the trend is well approximated. Using validation learning curve we are able to spot this situation: the training curve always decreases as the validation curve initially decreases (algorithm learning) and then increases (net overfitting).

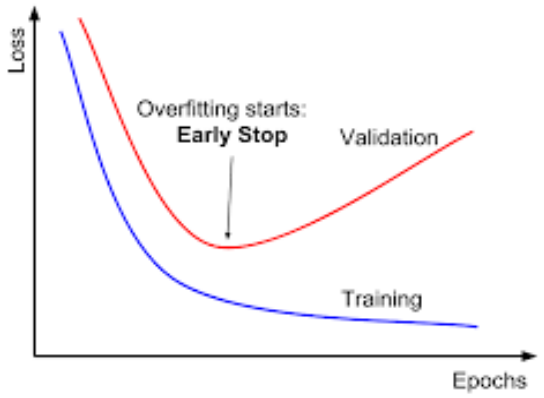


Figure 26. Overfitting learning curve.

- ***Underfitting*****:** underfitting can be described as the opposite problem of overfitting. The net has not got sufficient time to learn. As a consequence, the error (so the loss function) is high in both training and validation. Referring to Figure 24 and 25 once again, it is evident the issue related with underfitting. This particular situation can also be caused by a non-sufficiently complex architecture: should be clear by now that *depth* and *neurons* dictates the learning potential, if they are not enough for the application *underfitting* can occur. Underfitting is more difficult to spot from the learning curve since it leads to almost identical behaviour of training and validation. However, if the performance are not sufficient we can talk of *underfitting* issues.

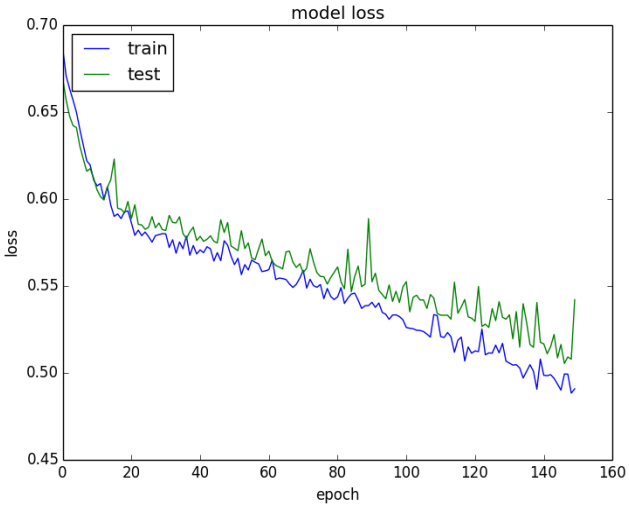


Figure 27. Possible undefitting curve..

2.6. Adam optimizer

The pre-existing project, starting ground for the present project, analysed different *optimizers* to define the one of choice. However, the results in that case showed that it was possible to achieve more or less the same performances in terms of prediction accuracy. Notice that *AdaGrad* was chosen because it would find acceptable configurations more often than the others.

In the context of this study another approach has been adopted. Since the author believes that sufficient literature is present to compare different *optimizers*, it has been chosen *Adam* on the basis of the publication relative to this particular algorithm [13].

Adam was presented by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in their 2015 ICLR paper. The name "*Adam*" it is not an acronym and it is derived from "*Adaptive Moment Estimation*".

Adam differs from classical *Mini Batch Gradient Descent* since it does not maintain a single *learning rate* for each weights' update. Quoting directly from the paper:

<<*The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients*>>.

The authors proceed to explain how *Adam* combines the advantages of two other very commonly used optimizers, namely Adaptive Gradient Algorithm (*AdaGrad)* and *Root Mean Square Propagation (RMSProp)*:

- *Adagrad:* it maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems)
- *RMSProp:* it maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy)

In fact, *Adam* makes use of the average of the second moments of the gradients, additionally o the first one. With these two averages *Adam* updates the parameter-specific learning rates. The just mentioned average is a moving average.

Besides from the mathematical standpoint it is of great interest to look at the compared results given in the abovementioned study. Referring to Figure 28 and 29, it is clear that *Adam*, and quoting:

<< […] compares favourably to other stochastic optimization methods>>

Adam was applied to the logistic regression algorithm on the MNIST digit recognition and IMDB sentiment analysis datasets, a Multilayer Perceptron algorithm on the MNIST dataset and Convolutional Neural Networks on the CIFAR-10 image recognition dataset.

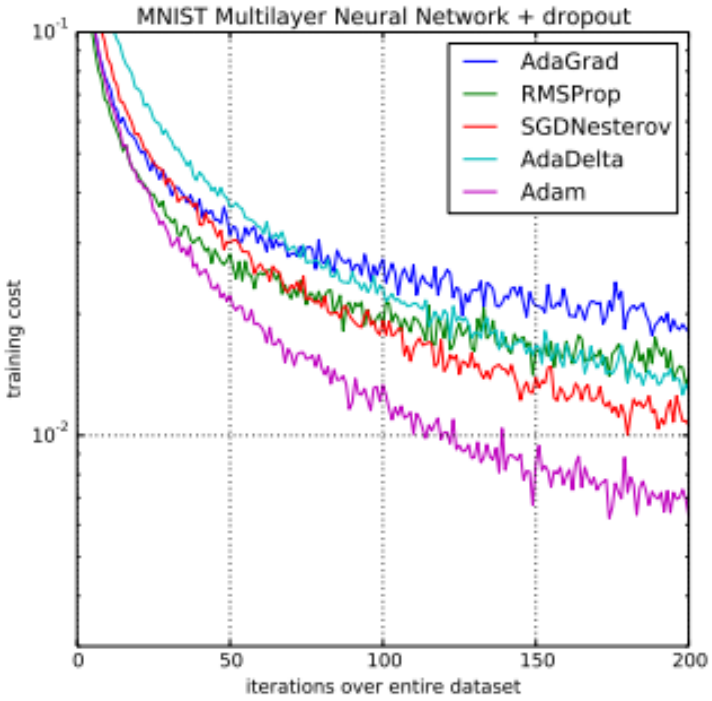


Figure 28. Adam and other algorithms on MNIST database.

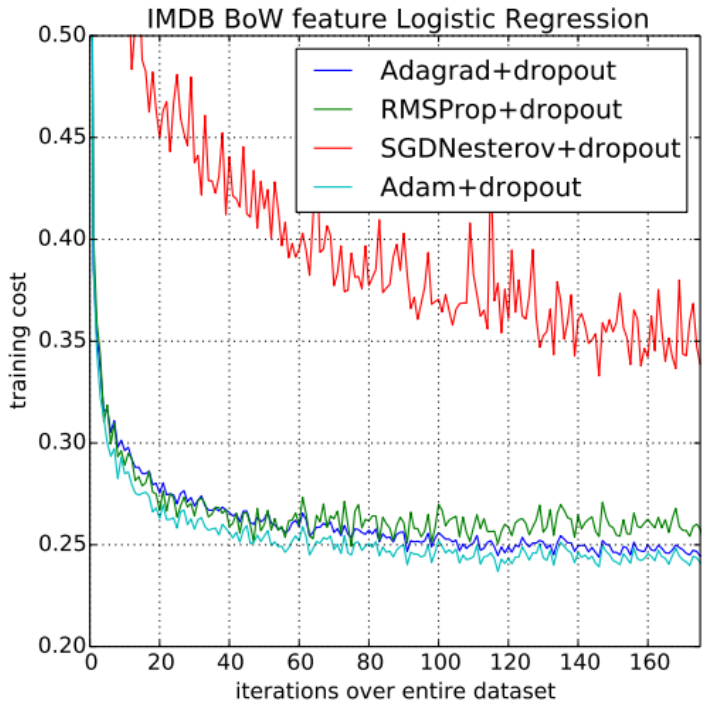


Figure 29. Adam and other algorithms on IMDB database.

For the abovementioned reasons *Adam* is the optimizer of choice for both the *classification* and *regression deep neural networks* of the pipeline.

2.7. Initialization

When the net is activated the first time, during the first forward propagation, the *weights* of all connections are of course not adjusted yet. A starting point is therefore necessary to compute predictions for the first time and build the learning process on them. Notice that the optimization process is dependent on this phase. As it is already been explained, the loss function depends on the weights so choosing a combination of them means to choose the starting point for the *descending* process of the learning phase. A good initialization can therefore greatly contribute to a fast learning process.

Different approaches have been tested and some general considerations are possible. In particular, we should always avoid initializing to zero (or any other constant values) all the weights. This, even though it seems plain and simple, can cause problems. Since the update of the weights is based on the computation of the gradient that in turns is related to the weights-based loss function; if all the weights are equal so will be the updates and the weights connected to the same neuron remain the same. This is known as *symmetry problem* and should always be avoided.

Modern approaches starts instead from a *random* initialization. This means that the weights are set to a random value at the start of the process. The procedure breaks effectively the symmetry. However, some issues are still present. In particular, if some weight is set to a very small or a very large value. To avoid this problem, the values are often normalized/standardized between -1 and 1, for example. This kind of initialization can draw values from mainly two types of distribution: *uniform* or *normal*.

Some of the best initialization are the so called "*Xavier*" and "*He*" initialization, respectively for *tanh* and *ReLU* activation functions. They ae defined as "*size informed*" initialization. They are in fact multiplied by a factor that takes into account the size of the previous layer of neurons:

$$\sqrt{\frac{1}{size_{l-1}}}$$

This formulation can assume different forms and values, but the general concept remains the same.

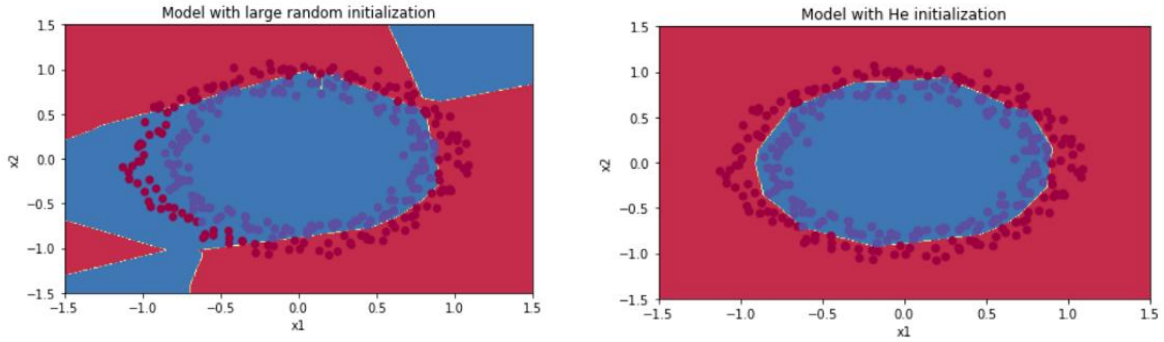


Figure 30. Random initialization compared with He initialization. Classification task.

Two common issues are the so called *vanishing and exploding gradient*. Even though they are not solely related to weights initialization, this is proved to be one of the main causes.

With *Vanishing Gradient* is described the phenomenon by which the gradient calculated during *back propagation*, and the relative updates, is too small; this cause the weights to be insufficiently adjusted and in turns a not acceptable learning process.

Exploding Gradient is the exact opposite: the gradient becomes bigger and bigger, causing the algorithm to diverge completely.

2.8. Hyperparameters' tuning

Should be clear by now that the *hyperparameters*, or better their combination, are the main influencing factors in terms of net performance. Should also be reminded that acting on a single parameter at a time is extremely counterproductive since the principle of superposition of effects does not apply to these kind of problems. Moreover, for different training sets or different training set's sizes, the combination that grants the best performance is not assured to remain the same. These considerations lead to the need to implement an automatic searching technique that is able to assess which combination is more likely to perform the best. Remember that since we are training the algorithm on a set of examples that at best is very similar to future applications but never identical, we cannot have the certainty that the selected combination will be the best possible one.

It turns out that the choice of the correct combination of parameters is also far more time consuming than the actual training of the net once the combination has been chosen. This is however obvious: the *hyperparameters' tuning* procedure is essentially a technique that involves several training and testing steps.

Several searching technique have been proposed, from the basic concept of just manually try different combinations and analyse the results to more complex algorithm that are able to explore the so called "*hyperparameters' space*" more systematically. The most common searching techniques are for sure *grid search* and *random search*. They are of straightforward application and fairly simple to implement. Another promising technique is *Bayesian optimization*. This procedure scan the *hyperparameters' space* applying a logic that should ensure better results.

The *space* in which these algorithm search needs to be defined beforehand. It is usually very wide, especially if there are no insights regarding the problem to be tackled. All the parameters that we impose to vary can be considered as the degree of freedom of this optimization problem.

2.8.1. Grid search

Given a finite number of possible combinations, this is the only searching method that is assured to find the absolute best among them. The functioning simple: the algorithm simply tries out every single possible combination in the grid, saves their results and chooses the best performing combination.

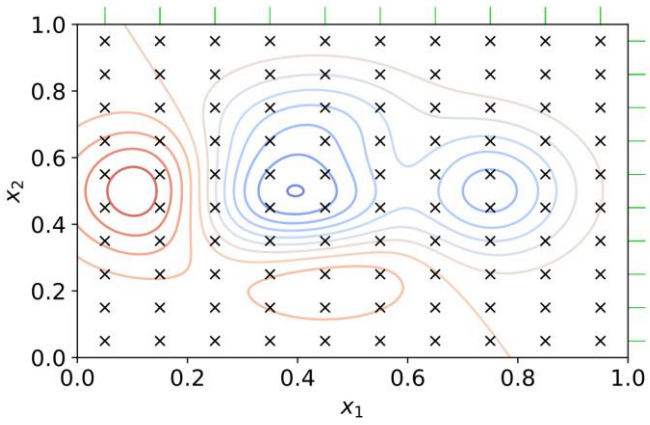


Figure 31. Grid search for a two dimensional problem.

This technique shows two main drawbacks. The first one is related to computational time: as it turns out, trying every single combination is extremely time consuming, especially for very large hyperparameters' spaces. It is not uncommon to deal with spaces with six or seven dimensions, and very complex tasks require even more dimensions. The second, and probably more important drawback, is the impossibility to use continuous distribution to create the space to be analysed: this method requires the implementation of a grid. It can be forced to vary its steps in different ways (e.g. logarithmically), but it should remain a grid. This limits the possibility to effectively explore the space, unless a very fine grid is implemented prolonging the search even more.

2.8.2. Random search

By far the most used technique nowadays. It can be implemented with extreme ease and grants very good results. It draws casually from the space a certain number of combinations and tests them out. Once the results re ready, it chooses the best. In fact, given a certain amount of tries, it is likely to perform better than grid search or better, given a certain performance goal it is likely to find a combination that allows those performance faster than grid search. Even though it is counterintuitive, the concept becomes obvious when visualized.

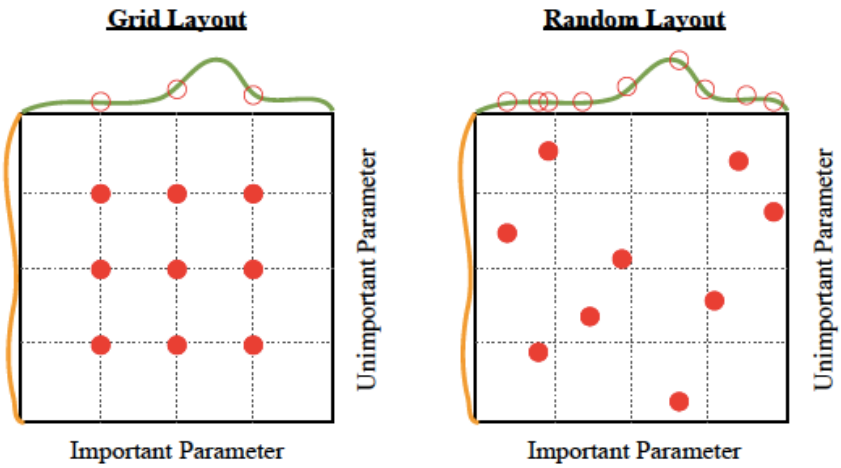


Figure 32. Grid search and random search visualized.

Referring to Figure 32, it can be noticed how in a hypothetical 2-dimensional problem that is composed of parameters of different importance, random search allows the algorithm to explore more the space. In particular, notice how for grid layout the algorithm is only seeing 3 different values for the important parameter, since the majority of them is superposed if projected along the corresponding axis. Moreover, random search allows to use continuous distributions so unlocks the possibility to spot areas of the space the grid was blind to. These considerations, combined to the fact that a random search algorithm is also easy to parallelize on multiple machines, leads developers to usually prefer random search rather than grid search.

Also in the case of random search, the axes of the hyperspace in which the algorithm searches, namely the various *hyperparameters*, can be forced to vary in a particular way. This is of great importance to explore significantly different areas of the space: having a great number of tries drawn from a space whose boundaries are too close, is not so interesting. It would be better, for example, to have the same number of tries in different order of magnitude rather than almost equally spaced samples.

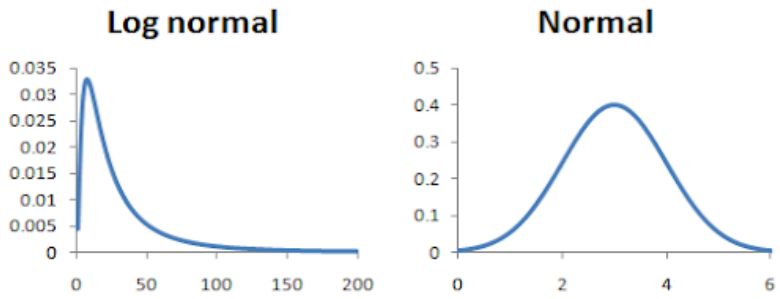


Figure 33. Log normal and normal distribution.

2.8.3. Bayesian optimization

Bayesian optimization is a particular kind of black-box optimization algorithm based on the *Bayes Theorem*. Without diving into the statistical and mathematical details, which are beyond the scope of this study, we can state that:

<<Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself>> [14]

This applies to machine learning, for example, when it comes to predict the areas of the *hyperspace* that is more likely to show good results in minimizing a certain objective function [15]. Notice that in this case the objective function is not simply the loss function, it is rather the algorithm of hyperparameters' tuning itself.

Although showing very good results, the implementation is not so simple. Moreover, it is very difficult to parallelize. For these reasons it not the most common choice of modern application.

Notice that it has been tested in the present study but due to complications the results were not coherent, so they have been discarded.

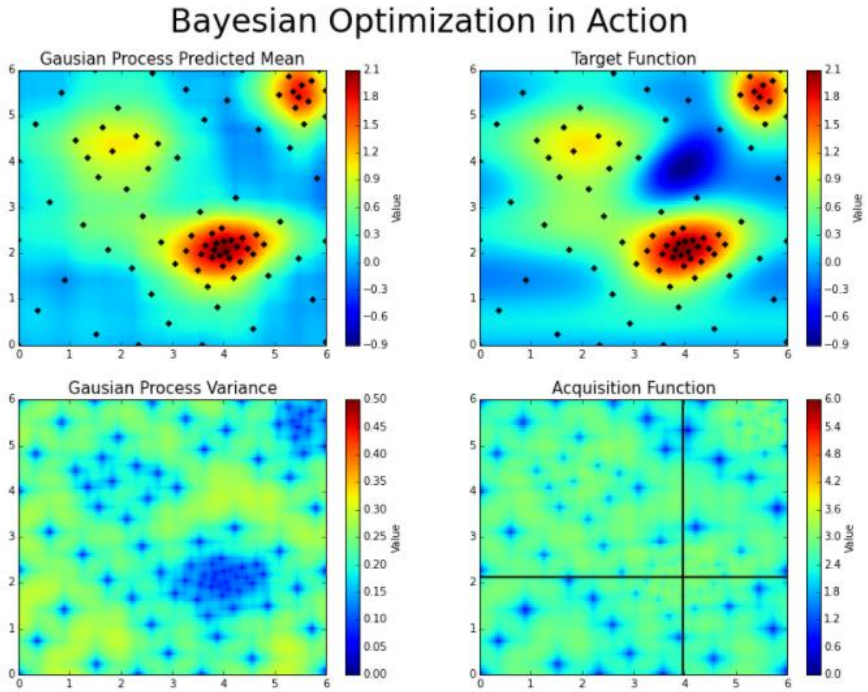


Figure 34. Bayesian optimization process.

The figure shows how the *acquisition function* (the one that dictates the area of the space to investigate) draws samples from the most valuable zones of the *target function* (red zones in the upper-right quadrant).

2.9. K-folds cross validation

The importance of testing the net on data it never saw should be clear by now. There are several ways to assess the overall possible performance of a neural network during the *hyperparameters' tuning* phase.

Something to keep in mind is the heuristic nature of this kind of predictive algorithm. Moreover, since it needs to learn from a set of examples, even though it is possible to make them sufficiently representative of the problem, there will always be the possibility that the training set is somehow biased. At minimum, it can be stated that it influences the learning process. Also, when *validating* the results on a set of examples, the *validation set* could also influence the performance of the net. The stability and coherence of the measure are therefore at risk if the dataset is not sufficiently numerous. When there is the risk of biased measurements of the net performance a helpful instrument is the so called *k-fold cross-validation*, which functioning will here be explained.

In general, when creating the dataset to be used (*training set, validation set* and *test set*), the test set is usually selected first, in a casual manner and after properly shuffle the whole training set. The choice of the ratio, referred here as *t/t ratio*, by which the test set is selected is important in order not to have a too sparse test set or a too small training set.

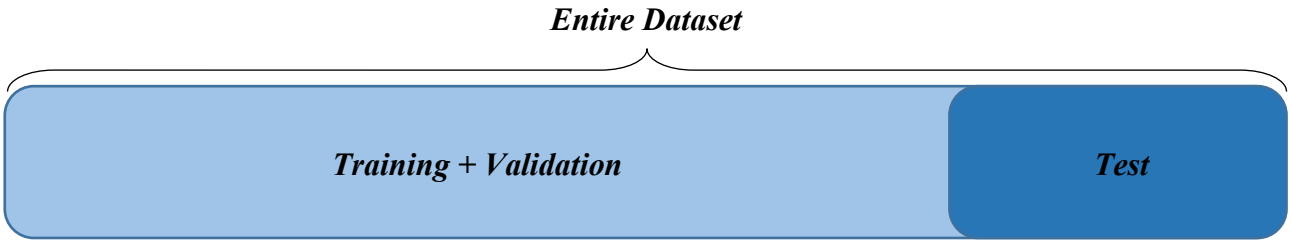


Figure 35. Train, validation and test sets.

K-folds cross validation involves the choice of a certain number of *folds* by which the "*training+validation*" set is divided. Then, the training and validation process is repeated "*K*" number of time, varying the validation set each time from the subset selected. This procedure allows to have "*K*" differently composed training sets and "*K*" different validation sets, without ever seeing the test set yet.

In particular, with reference to Figure 36, the operational steps are the following (notice that a *4-fold cross validation* is used):

1. Training using subsets 1, 2 and 3. Validating using subset 4 – 1st metric available.
2. Training using subsets 1, 2 and 4. Validating using subset 3 – 2nd metric available.
3. Training using subsets 1, 3 and 4. Validating using subset 2 – 3rd metric available.
4. Training using subsets 2, 3 and 4. Validating using subset 1 – 4th metric available.

Once the four metrics have been computed and the average value is available, considerations on it are possible and, if necessary, the net is tested on the test set.

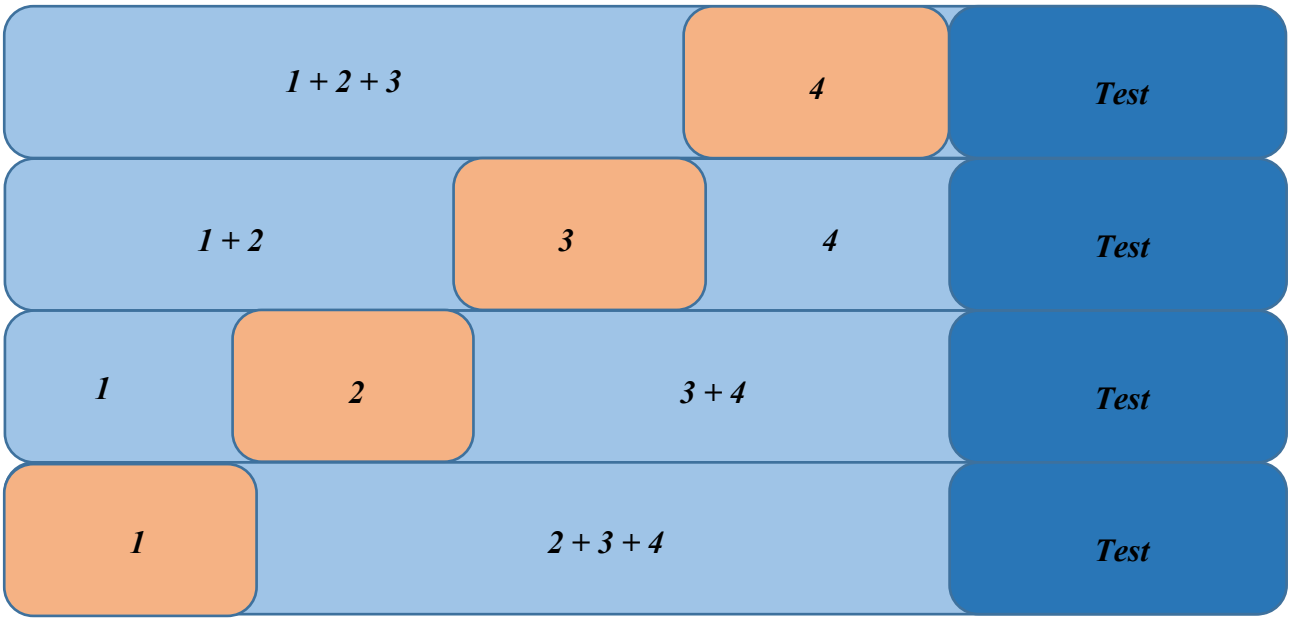


Figure 36. 4-folds cross validation steps.

Data management

3.1. Data acquisition

Main goal of the present project is to produce a tool that should be able to reproduce a *Dynamic Programming (DP) algorithm* prediction accuracy, with extremely lower computational time. This specific kind of numeric simulation represents probably one of the best performing algorithm in terms of prediction accuracy; it is however a purely deterministic approach, meaning that the entire mission should be known a-priori and the simulation time will surely be high. It is then no surprise that the data that compose the datasets are direct results of said *DP algorithm*; in this way the nets are trained to reproduce exactly those results.

In this study different architectures will be dealt with, namely P2, P3 and P4 HEV architecture. Three different datasets are therefore at disposal (plus a P2 dataset that was used only few times to run preliminary simulations; this dataset in however older than the others and its data are considered less reliable), one for each architecture.

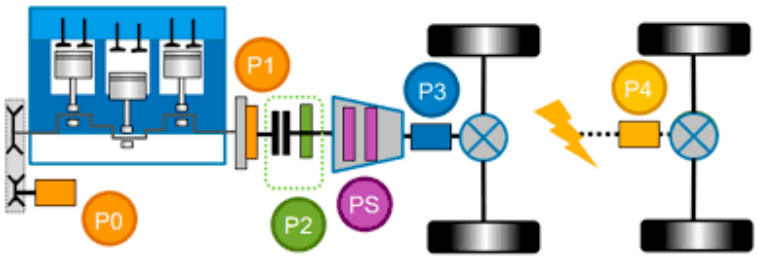


Figure 37. Possible simple HEV architectures visualized.

The *DP*'s simulations are based on the *World Harmonized Vehicle Cycle (WHVC)*. It is a chassis dynamometer test developed based on the same set of data used for the development of the *World Harmonized Transient Cycle (WHTC)*. For completeness, the velocity profile of the *WHVC* is given below.

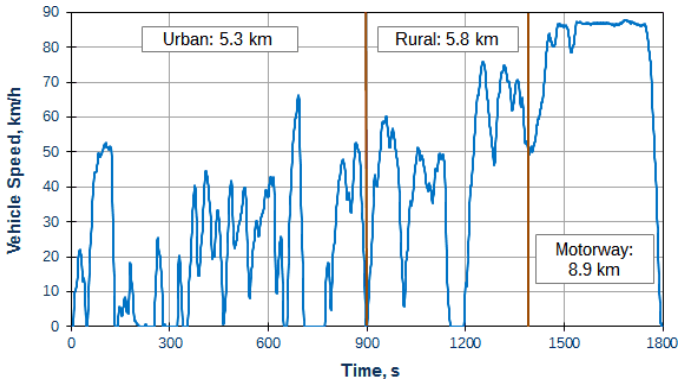


Figure 38. WHVC velocity profile.

3.1.1. Dynamic programming

Since on its functioning is based the data acquisition, it is considered useful to describe in more details how *Dynamic Programming* operates.

The working flow is based on a decision making process relying on a backward-forward logic. A control grid is implemented in order to describe all possible states of the system and the path connecting them; the system is composed by a finite number of control variables that define the possible inner point of the grid; a state variable is also monitored.

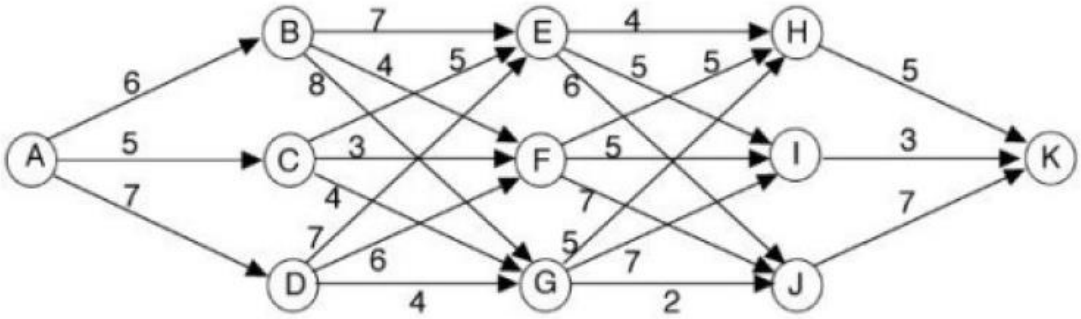


Figure 39. Dynamic programming grid and connections.

Referring to Figure 39, the goal is to reach point "*K*", starting from point "*A*" and using the shorter path possible. The values present on the arrows represent the length of the path connecting two states of the grid. The logic is to compute, starting from the final point, the shorter path connecting the previous nodes to the one considered, each node at a time. The shorter path is addressed as *cost-to-go* path/distance. So for example, the *cost-to-go* distances for nodes H, I and J are respectively 5, 3 and 7 (only one possible path in this layer). Going on to the previous layer, the *cost-to-go* distances for nodes E, F and G are:

E. 5 + 3 = 8. Passing through node I.
F. 5 + 3 = 8. Passing through node I.
G. 7 + 2 = 9. Passing through node J.

The procedure continues back to the starting point, searching for the *cost-to-go* distances for all the nodes. Finally, a value for point A will be available and it will be the *cost-to-go* of the all path connecting points A and K.

The same logic can be applied to control strategy optimization of HEVs. Each step will then be a different time instant of the mission and each node will be a powertrain state. In this application, Figure 40, the only state variable is the *State Of Charge (SOC)* of the battery, and only three states for it are possible: Low, Medium or High.

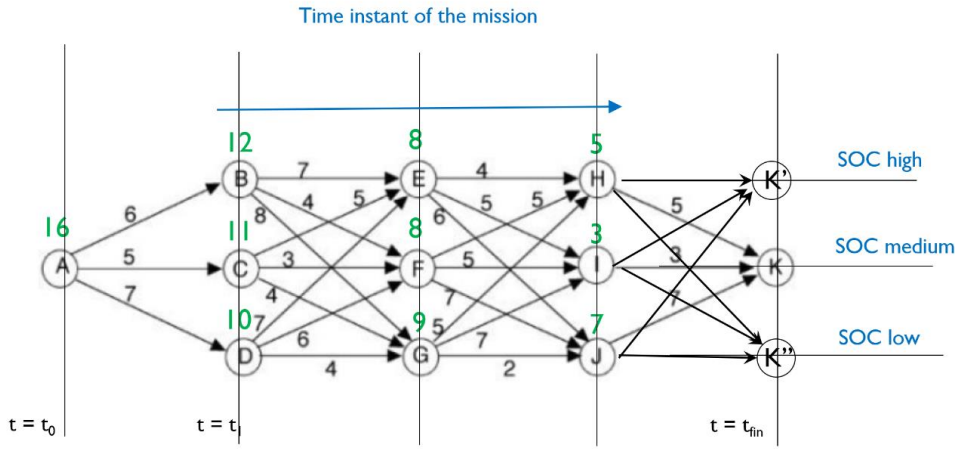


Figure 40. Dynamic programming for an HEV control strategy.

Each node is actually the combination of the two control variables: *Front Power Flow* and *Gear Number*. *Regenerative Breaking* is applied during braking action. Control variables are also discretized to reduce the computational effort required by the optimization algorithm. In particular:

- *Front Power Flow*: battery charge (-1, -0.5), pure thermal (0), power split (0.5) and pure electric (1)
- *Gear Number*: 1st, 2nd, 3rd, 4th, 5th and 6th

The dynamic model of the vehicle (for a P2 architecture) is described by the following system of equations that links the velocity and power of the vehicles component to the control variables:

$$\begin{cases} \omega_{tcd} = \tau_{gb}(GN) * \tau_{fdf} * \dfrac{V_{veh}}{R_w} \\ \omega_{ice} = \tau_{tcd,ice} * \omega_{tcd} \\ \omega_{emf} = \tau_{tcd,emf} * \omega_{tcd} \\ P_{emf,mech} = FPF * P_{tcd,mech} \\ P_{ice,mech} = (1 - FPF) * P_{tcd,mech} \end{cases}$$

Similar equations ae available also for P3 and P4 architectures.

As aforementioned, the aim of the optimization algorithm is to minimize a certain function. In this case the cost function has the following structure:

$$J = \alpha\left(\frac{FC}{FC_{ref}}\right) + (1 - \alpha)\left(\frac{NO_x}{NO_{x,ref}}\right)$$

Where *FC* is the cumulative fuel consumption over the mission and NO_x are the cumulative NO_x emissions over the mission. "α" is instead a weighting factor that "shifts the attention" of the algorithm on fuel consumption or emissions: different strategy are therefore available, from FC-

oriented to NO_x oriented. At each step the less expensive action is performed to keep function *J* as low as possible.

Moreover, at the start of the simulation, boundary conditions are imposed, namely *SOC-start* and *SOC-final*. This means that usually the state of charge at the end should be higher or equal to the SOC at the beginning of the simulation. In this way, complete depleting strategy for *J* minimization are avoided.

Possible results, coming from a study that used the application here described, could be the following.

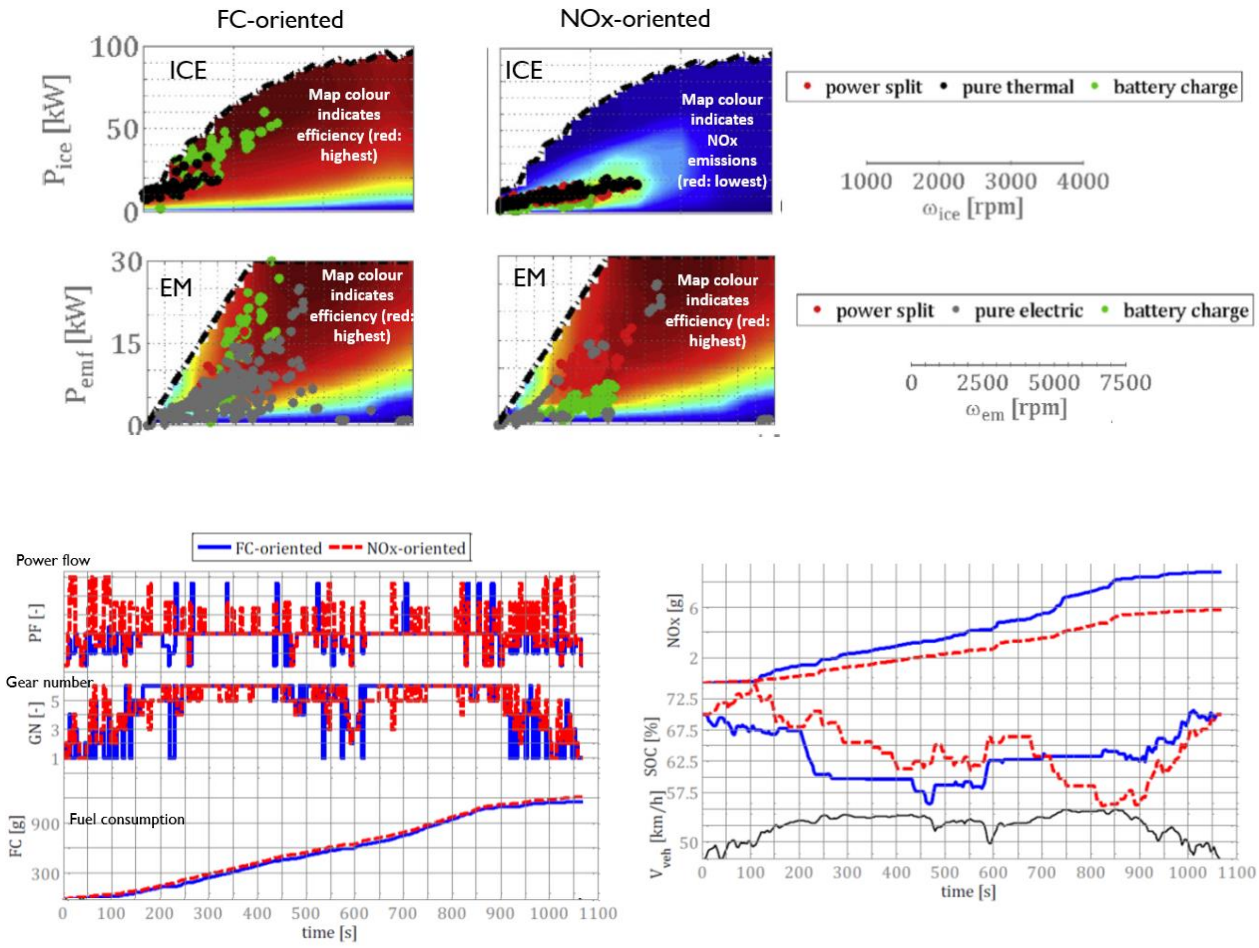


Figure 41. Dynamic Programming simulation possible results.

The results above were obtain on an *Artemis Motorway Driving Cycle (AMDC)* simulation.

3.2. Dataset composition

The three available datasets have the same general structure. They are composed of 1500 examples. Here lays the major difference with the pre-existing project: in that context the dataset was of 5000 example, and the vast majority of them was *feasible* (the layouts they represented was able to complete the driving cycle); now the 1500 examples are for a third *unfeasible*. This condition means that the *Regression Deep Neural Network* will now have much less example to train on. At the same time, also the *Classification* one will train on fewer examples. We will see that the dimension of the training set represents a key point for the performance.

Each example in the datasets is composed of 8 features. From the tables is evident that the datasets P2 and P3 presents the same feature, dataset P4 instead does not. This is because the first two share the implementation of a *speed-coupling* device that connects the *electrical machine* to the rest of the system. P4 architecture instead include the *electrical machine* on a separate axle, so we should consider the transmission ratio of that axle.

In general, the features can be described as follows:

- *EngDispl:* engine displacement, measured in *litres*; it is the displacement of the ICE.
- *PE ratio:* it is the ratio between the *Electrical Machine* power and the maximum energy of the battery pack.
$$PE = \frac{P_{em}}{E_{bat}}$$
- *EM1Power/EMsPower:* power of the electrical motor, measured in *kiloWatts*.
- *EM1SpRatio:* transmission ratio at the speed-coupling device.
- *FDpSpRatio:* transmission ratio of the primary axle.
- *FDsSpRatio:* transmission ratio of the secondary axle.
- *CrateDis:* maximum C_{rate} during discharge of the battery. Defined as:
$$C_{Dis} = \frac{1h}{t_{I=max}}$$
where $t_{I=max}$ is the discharging time at max intensity current.
- *CrateCh:* maximum C_{rate} during charge of the battery. Defined as:
$$C_{Ch} = \frac{1h}{t_{I=max}}$$
where $t_{I=max}$ is the charging time at max intensity current.
- *CO2ttw:* CO_2 tank-to-wheel measured in *grams*; values as predicted by the DP algorithm.

A sample of the three datasets follows.

Table 1
Extract of the P2 dataset.

# Ex	EngDispl [l]	PE ratio [-]	EM1Power [kW]	EM1SpRatio [-]	FDpSpRatio [-]	CrateDis [-]	CrateChar [-]	CO2ttw [g]
1	3,0573	7,797852	118,9307	4,147461	3,09473	11,8418	7,0020	331,5416
2	2,9045	9,96582	74,8877	4,377930	3,23145	11,0215	7,35352	10000
…	…	…	…	…	…	…	…	…
1500	2,8156	28,32031	59,082	4,449219	4,65430	7,9336	10,5117	10000

Table 2
Extract of the P3 dataset.

# Ex	EngDispl [l]	PE ratio [-]	EM1Power [kW]	EM1SpRatio [-]	FDpSpRatio [-]	CrateDis [-]	CrateChar [-]	CO2ttw [g]
1	4,7029	5,200195	123,8721	4,893555	3,49707	6,802734	10,82227	341,2689
2	2,7428	5,610352	122,3682	4,334961	3,782227	6,216797	8,876953	336,2393
…	…	…	…	…	…	…	…	…
1500	3,7719	28,55469	61,2305	3,519531	3,962891	10,95703	8,378906	10000

Table 3
Extract of the P4 dataset.

# Ex	EngDispl [l]	PE ratio [-]	EMsPower [kW]	FDpSpRatio [-]	FDsSpRatio [-]	CrateDis [-]	CrateChar [-]	CO2ttw [g]
1	2,6158	5,073242	106,0303	4,142578	14,79883	8,759766	9,287109	377,1168
2	3,9209	5,063477	113,208	4,576172	10,47461	7,998047	8,033203	389,4292
…	…	…	…	…	…	…	…	…
1500	3,9406	29,25781	52,0508	4,689453	10,44141	6,339844	11,54297	10000

The *CO2ttw* feature represents the *true label* for the training process. As mentioned more than once, the present project implements a *pipeline* of a *cDNN* and an *rDNN*. Since the two nets have different goals, the labels should be different too.

It can be noticed from the tables that in the *CO2ttw* column some 10000 values are present. Those values are fictitious ones: it is the way in which the *DP software* tells the operator that a certain example could not complete the cycle.

3.3. Data manipulations for labels generations

From the considerations just mentioned it is clear that some manipulations are necessary. In particular, to obtain an effective *labels feature* column the following operations where performed prior to any type of simulation:

- A copy of the entire dataset is saved as it is.
- The values different from 10000 are substituted with "1", meaning a *feasible* example.
- The values equal to 10000 are substituted with "0", meaning an *unfeasible* example.

These steps allows to create a dataset that can be used for the *cDNN* only: the information regarding the actual emissions is not present anymore.

To train the *rDNN*, after the *classification step* of the *pipeline*, all the *feasible* example will point to the starting dataset in which the CO_2 values are present. In this way we are able to train the *rDNN* effectively.

3.4. Data split

From Section 2.9 should be clear the logic behind the generations of the training, validation and test set. To summarize, the whole dataset is shuffled and divided into *X-set* (the *features* of the examples) and *Y-set* (the *labels*); then a *t/t ratio* is chosen and the *test* set is randomly selected. Supposing the procedure explained in Section 3.3 has already been performed, it is now possible to proceed with the *Hyperparameters' tuning* procedure as already described. Notice that the procedure include various *k-fold cross validation*. In this application *K* is equal to 8, so eight subsets are created for the *validation* process.

After a *hyperparameters' combination* has been selected, the *cDNN* is tested on the test set. At this point the procedure goes on to the *regression* step. The *training set* for the *rDNN* is directly derived from the *classification* one: the examples that was *feasible* in the *cDNN training set* make up for the *rDNN one*. For the *validation set* the procedure is the same since they actually derive from the same bigger dataset. The *test set* for the regression step is more complex. Since the net should be tested considering also the performance of the classification step, the *true positive* examples of the classification test set are isolated; a test is performed on them. The net is also separately tested on the *false positives* (*unfeasible* examples marked as *feasible* by the *cDNN*).

3.5. Data normalization

Normalize the data used is surely standard practice in *Deep Leaning* application. It is indeed proven that the net can greatly benefit from the normalization of the *training set*. This practice is especially needed when the distribution of the data that will be used is not clearly known.

Moreover, unless we know for sure that all the features we are managing range in the same order of magnitude, it is mandatory to normalize the input data between some values or in the same order of magnitude. With reference to Table 1, 2 and 3, it can be observed that the feature corresponding to the power of the electric motor reaches values of a hundred and more. This is for sure a valid reason to normalize the data.

Notice that the *normalization technique* is applied only to the *X-set*, namely the *examples' features*, and not to the *Y-set*. Moreover, since the net will be learning from a normalized dataset, also the *validation* and *test sets* should be normalized, otherwise the results will not be consistent.

Different technique have been used in different application to achieve an effective normalization of the data. In the context of this project it is implemented the *standardization technique*. The function is the following:

$$\tilde{x} = \frac{x - \bar{x}}{std(x)}$$

Where:

- $\tilde{x}$: new entry of the dataset after normalization.
- $\bar{x}$: mean of all the example for the specific feature considered (column of the dataset).
- x: entry of the dataset before normalization.
- $std(x)$: standard deviation of the specific feature considered (column of the dataset).

This type of *normalization* is preferred to the one used in the pre-existing project because it is believed to better deal with outlier values. In particular, it was used the *min-max scaling technique*, which implements the following formula:

$$\tilde{x} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Where:

- $\tilde{x}$: new entry of the dataset after normalization.
- x_{min}: minimum value of the specific feature.
- x_{max}: maximum value of the specific feature.

This procedure, although effectively limiting the range of all the features between 0 and 1, suffers from the fact that a single outlier could cause the dataset not be evenly distributed. Imagine for example the case in which all *EM-Power* features are between 0kW and 50kW except for one value that is 100kW. The *min-max scaling* would restrict the vast majority of the dataset between 0 and 0.5. This cannot happen with a *standardization* because every example is considered to make up for the scaling factors, namely the *average* and the *standard deviation*.

Should be mentioned that however the *standardization* cannot accomplish the squeeze of the dataset between two a-priori known values, and those values will not be the same for all the features. However, the key point is to force the dataset to be in the same order of magnitude.

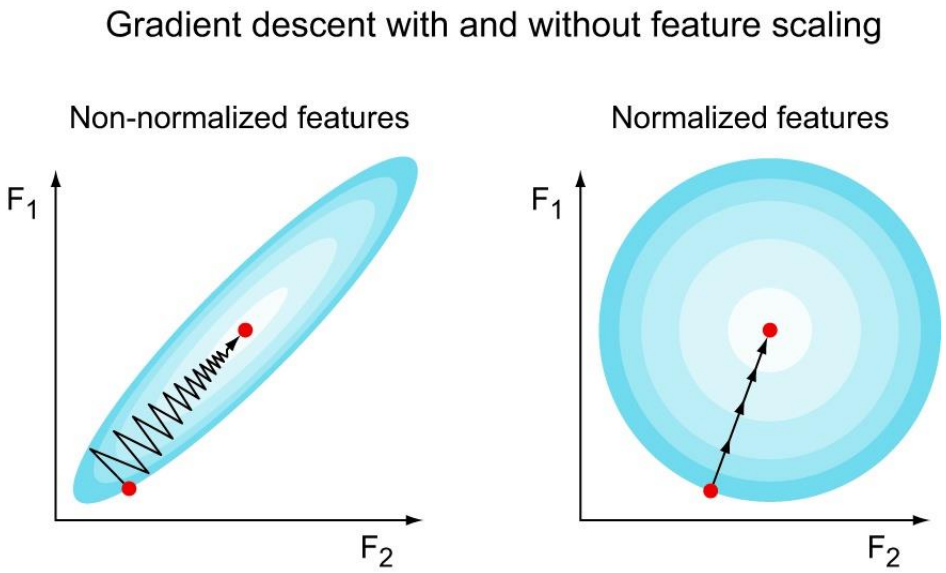


Figure 42. Effect of normalization on gradient descent.

From Figure 42 it is evident that, in case of non-normalized features, the gradient descent algorithm could get into complications. This phenomenon in related to the fact that he cost function could assume an "uneven" shape, like the one depicted in the figure. This cause the algorithm to bounces back and forth along the cost function. Normalizing the data effectively reduce this possibility and the convergence is more fast and the overall process more stable.

Neural networks based model

4.1. Working environment and libraries

The development of the present project has been carried out using *Python 3* programming language. The reasons of this choice are evident: *Python* is based on a very simple syntax offering at the same time numerous tools and libraries for different applications. Moreover, they are usually well performing but very user-friendly.

In this context, some of the main libraries used are listed below:

- *Tensorflow*: created by the Google Brain team, it is an open source library for numerical computation and large-scale machine learning. Tensorflow offers a vast choice of machine learning and deep learning models and algorithms. It uses Python to provide a convenient front-end API for building applications with the framework, while executing those applications in high-performance C++. It is used in this project as back-end library.

- *Keras*: Keras is an open-source neural network library written in Python. It is capable of running on top of Tensorflow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. It was developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System), and its primary author and maintainer is François Chollet, a Google engineer.

- *Scikit-learn*: it is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting and k-means, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

- *NumPy*: it is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

- *Pandas*: it is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

- *Matplotlib*: it is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits.

4.2. Metrics

Before entering into model's details, it should be explained how the performance of the nets ae being evaluated. Once again, keep in mind that the model is based on two different deep neural networks: one for *classification* and one for *regression*. It is therefore obvious that at least two different metrics should be used.

In general a "metric" is a function that, given *predictions* and *true labels*, is able to summarize the performance of the net on a specific simulation. For clarity, are considered "metrics" only single-value quantity, so for example a *confusion matrix* (often used in the context of this project) is not one, even though being a crucial tool to analyse classification performance.

4.2.1. cDNN metric

The results of a *classification task* are generally structured in *predicted labels* and *true labels*. In the context of this process, a *binary classification task* is present: the *cDNN* should predict only two classes, *feasible (1)* or *unfeasible (0)*. All the considerations present in this paragraph are therefore related to *binary classification*; however, should be stated that the concepts here present are all extensible to *multi-class classification*.

In a *binary classification* problem only four different outcomes are possible, referring to the above mentioned *predicted* and *true labels*. They are:

- *True Positives (TP)*: *feasible* examples correctly marked as *feasible*.
- *False Positives (FP)*: *unfeasible* examples incorrectly marked as *feasible*.
- *True Negatives (TN)*: *unfeasible* examples correctly marked as *unfeasible.*
- *False Negative (FN)*: *feasible* examples incorrectly marked as *unfeasible*.

The prediction for all the examples present in a dataset can be summarized in a particular table called *confusion matrix*. Using this tool, a complete overview of the performance is given; however, is up to the operator to judge the results. This means that an algorithm cannot take decisions based on this tool, or better, some kind of data should be harvest from it.

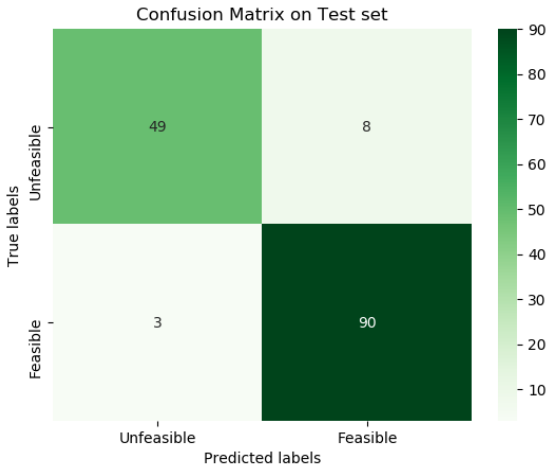


Figure 43. Example of confusion matrix.

Different single-values metrics are computable starting from the above mentioned table. Three of the most used ones are surely *Accuracy (Acc)*, *F1 score (F1)* and *Matthews Correlation Coefficient (MCC)*.

They are computed using the following relations:

- *Accuracy:*

$$Acc = \frac{TP + TN}{(TP + FN) + (TN + FP)} = \frac{TP + TN}{P + N}$$

This quantity represents the fraction of *correct predictions* that net was able to produce. This is surely the most intuitive way of describing the performance of the net during *classification phase*. It ranges from 0 to 1.

- *F1 score*:

$$F1 = 2 * \frac{precision * sinsitivity}{precision + sensitivity} = 2 * \frac{PPV * TPR}{PPV + TPR}$$

Where:

  - *Precision, or Positive Predicted Value (PPV)*:

$$PPV = \frac{TP}{TP + FP}$$

  - *Sensitivity, or True Positive Rate (TPR)*:

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

F1 score is a compound measure that takes into account both *precision* (fraction of the predicted positives that are indeed positives) and *sensitivity* (fraction of true positives correctly predicted as such). It values the most the *positive class* in its computation. Even though it is not as intuitive as *Accuracy*, it is a more stable evaluation of predictive performance: in case of strongly uneven dataset, with only few *positive* examples, is able to correctly judge is the net is performing well. It ranges from 0 to 1.

- *Matthews Correlation Coefficient*:

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

MCC takes into account all four classes of binary classification. It is indeed believed to be the best "one-number" performance indicator [17], since it is considered the most informative one. Notice that, differently from the *F1 score* it correctly judges the predictive performance regardless of the choice for the positive class. In fact, in the previous example, if the class are inverted, the *F1 score* could now be misleading. The *MCC* instead will continue to give consistent measurements.

For the abovementioned reasons the *MCC* is chosen to be the driving factor in the *hyperparameters' selection process* and during performance assessment for the *classification phase*. It ranges from -1 to 1.

4.2.2. rDNN metric

The *regression task* needs a metric that is able to measure how much the model's predictions deviate from the true values of the dataset. The chosen tool to accomplish this result is *R squared*, also known as R^2 *or Coefficient of Determination*.

It can be computed using the following formula:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

Where:

- $SS_{tot} = \sum_i (y_i - \bar{y})^2$
- $SS_{res} = \sum_i (y_i - f_i)^2$
- $\bar{y}$: mean value of the true labels
- y_i: true value
- f_i: predicted value

It is of common use in statistics and in general for linear regression performance assessment. It should be noted that its possible values range from *-inf* to 1.

The *R squared* coefficient is a way of comparing the "simple average" approximation to the model predictions. In particular:

- $R^2=1$: perfect fit by the model on the true labels.
- $R^2=0$: the model is performing like the "average approximation" would perform.
- $R^2<-inf$: the model prediction are in contrast with the true labels trend.

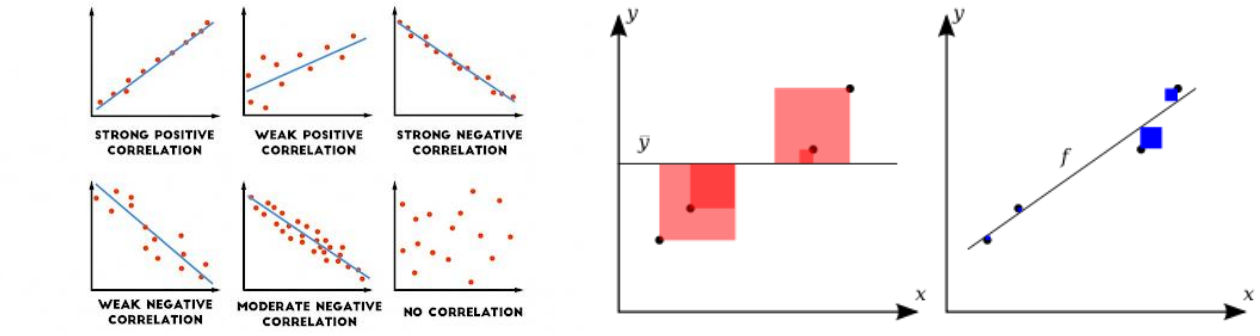


Figure 44. Coefficient of determination visualized.

4.3. Model general logic

As stated various time, the model is actually a pipeline of two *deep neural networks*. Let us now recap the overall logic behind it.

The model is in two stage:

1. Assess feasibility: a classification-DNN (cDNN) is used to predict whether a specific layout is capable of completing the cycle or not (*feasible* or *unfeasible*)
2. Predict CO_2 emissions: a regression-DNN (rDNN) is used to predict the CO_2 emitted by the *feasible* layout.

The formulation of the model can therefore be written as:

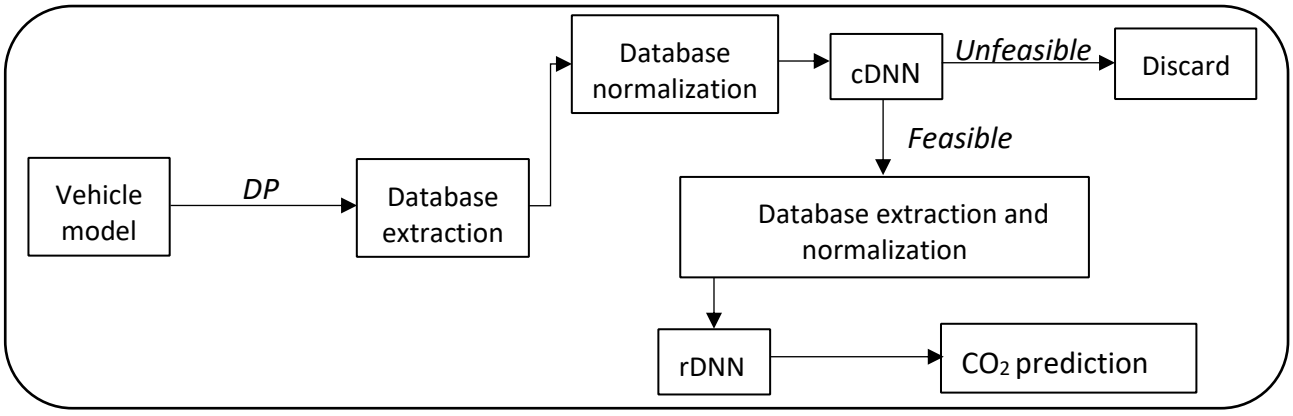
$$\begin{cases} Feas = f(DesPars, \theta_c) \\ CO2 = f(DesPars(Feas), \theta_r) \end{cases}$$

Where *Feas* are the *feasible* layouts as spotted by the cDNN; *DesPars* are the design parameters, namely the entry of the dataset before normalization; *DesPars(Feas)* are the design parameters of the layout classified as *feasible*; finally θ_c and θ_r are the features of the cDNN and the rDNN like hyperparameters, weights and activation functions. The two functions that appear in the formulation will be learnt by the NNs and the weights will be automatically adjusted.

Before each of the two abovementioned stage, a pre-processing of the data is needed to eliminate any dimension differences.

Moreover, an automatic hyperparameter selection, based on random search, is performed to avoid the need to manually search for optimal features configuration.

The whole logic behind the model can therefore be summarized as follows.



4.4. Multi-stage Deep Neural Network model

As already stated, the aim of this project is to design a tool able to extract patterns that lead to a CO_2 prediction. The DP algorithm intrinsically works in two ways: it is capable to assess which layout will complete the cycle and for those who can, asses its CO_2 emissions. Even though its performances are excellent, they come at a high time cost. To match these capabilities and overcome time related problems, two DNNs are implemented in the model.

A schematic of the DNNs pipeline is presented below.

Both DNNs are composed of an input layer, several hidden layers, and an output layer. To avoid gradient vanishing problems and to enhance convergence efficiency ReLU activation function is used in the hidden layers [18]. Different studies have also demonstrated that a batch-normalization layer can speed-up the convergence especially for DNNs [19], therefore it is applied in this project in both nets. Finally, a dropout-layer is used in the rDNN to reduce overfitting. Adam (adaptive moment estimation) algorithm is implemented in both nets since it ensures a fast convergence and stable update of the weights and biases, together with an effective learning rate automatic control.

The cDNN will produce a binary output indicating whether the example is *feasible* or not. If it is, it will reach the rDNN and a value of CO_2 will be predicted.

The following figure visually represents the pipeline, highlighting its core features.

Notice how modules like "*batch-norm layer*" or "*dropout layer*" are included sequentially after the actual *hidden layer* of the net. This visualization in used because it reflects the actual implementation in *Keras* environment. In fact, *Keras* provides for a sequential framework that allows the operator to stack layer on top of the other using simple sequential lines of code. This approach, although being restrictive for more advanced developers, allows beginners to effectively construct very complex architecture with few lines of code. This is also one of the reasons why *Keras* environment was chosen in the first place.

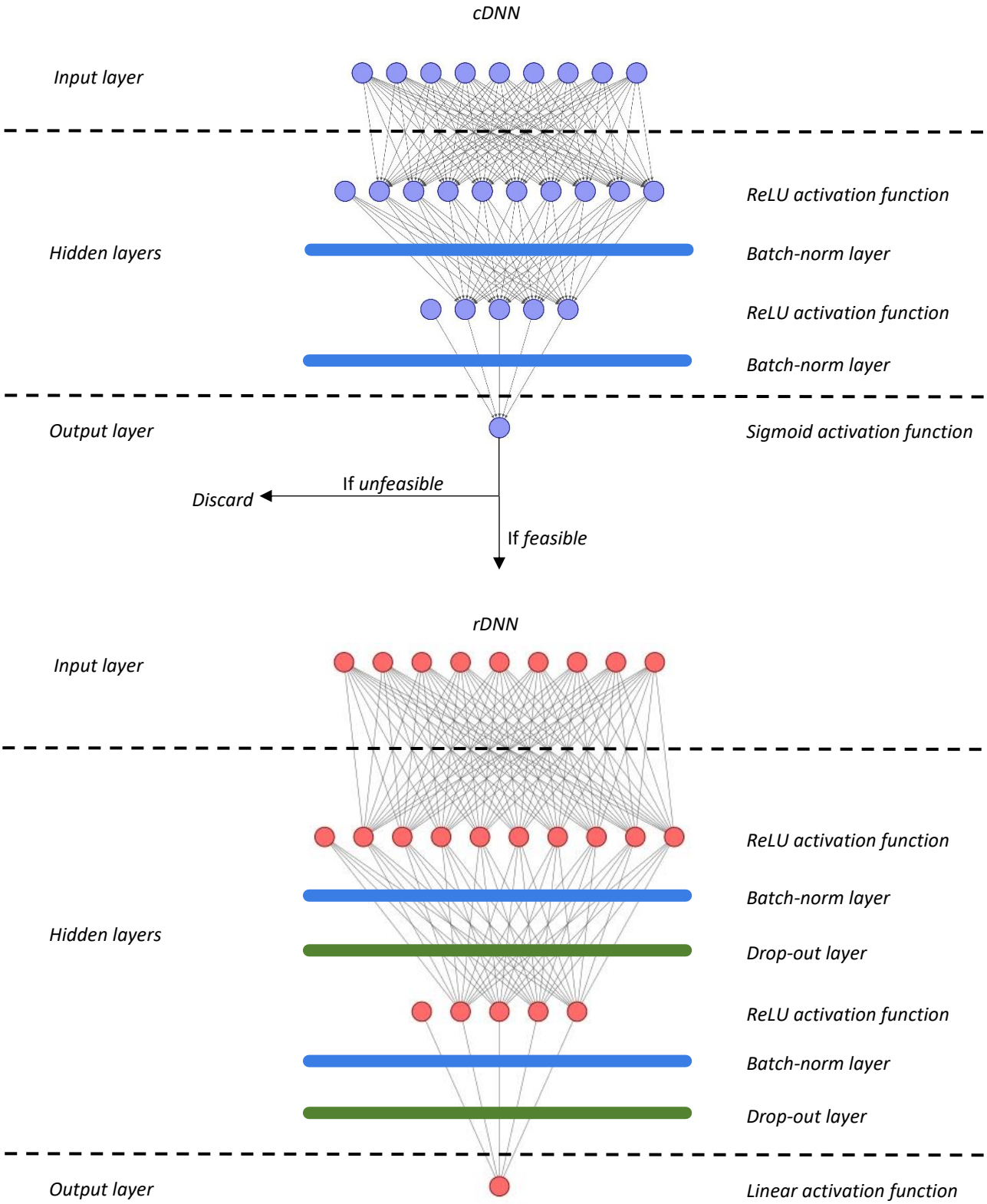


Figure 45. cDNN and rDNN functioning logic.

4.5. Learning curves

After the definition of the needed datasets, the model architecture and some basic parameters (e.g.: first learning rate, regularizer coefficient, number of layers and neurons), first trials are possible in order to implement correctly what has been addressed to as *Learning Curves*.

Those are functions describes the *learning process*. They are obtained by extracting the value of the *Loss Functions* (remember there are two separate loss functions to analyse) after each epoch of the *Training Process* both on the *Training Set* and on the *Validation Set*. With this procedure we are able to monitor if the learning process is consistent or if it is showing strange behaviour. An example of the learning curves obtained from actual simulations are presented below.

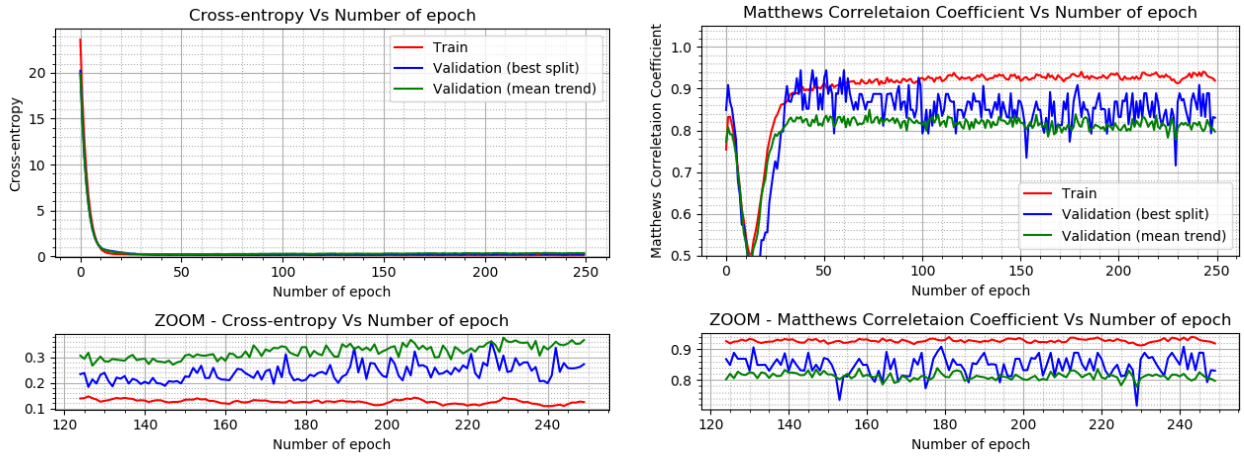


Figure 46. Loss function (left) and MCC (right) at different epochs – classification step.

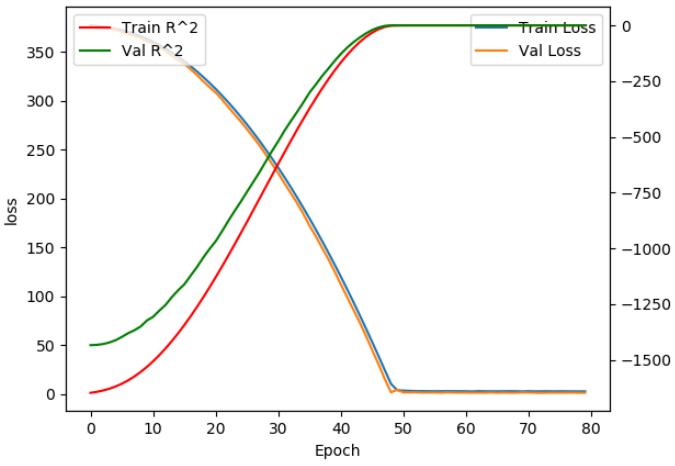


Figure 47. Loss function and R squared at different epochs - regression.

It has already been mentioned how, from the interpretation of these curves, it is possible to spot incorrect learning process. In this particular example, referring to Figure 46 and looking at the loss function, it is possible to spot a slight *overfitting*. The zoomed section of the graph shows indeed that the validation and training curves are actually diverging slightly. This phenomenon is more evident looking at the *MCC* behaviour, in particular for the best split curve (blue).

Other possible misbehaviour could be the plateauing of the validation curve at too high values, meaning an *underfitting* or at least an incorrect value for α, namely the *learning rate*.

Those curves helped greatly in preventing selection of evidently wrong epoch number. This we know can cause different problems, including the two abovementioned.

4.6. Optimization methods

The theory offers various tools to intervene on the behaviour of the nets, and in turns on their performance. Some of them were studied in the context of this project and implemented if considered a valid option to enhance the overall results. The vast majority of them has been implemented in order not to incur in *overfitting*, even if it does not show clearly.

4.6.1. Dropout layer

Dropouts are a very interesting approach to the overfitting problem. The core functioning is surprisingly simple: during each pass of the earning process (meaning each forward-backward cycle), a fraction of the neurons of each hidden layer are shut down. This means that they do not take part neither in the *prediction* or the *weights update* process. The selection of the neurons that will be shut down is completely random. The probability of a single neuron shutting down is passed to the software that automatically carries on the random selection. Should be highlighted that during the prediction phase, when the net is actually used or during validation/testing, all the nodes are at net's disposal. This could generate some inconsistencies between *training* and *validation* performance (e.g.: validation performance higher than training). However, this phenomenon shows up usually during the first few epochs of the training and then disappears.

It has been proved [20] that this way of proceeding effectively helps the net to avoid *overfitting*. This could be explained by the fact that, since the net can no longer rely on each single neuron (all of them could potentially be deactivated), it is forced to learn the general pattern of the trend rather than creating precise internal path that lead to a perfect training fit.

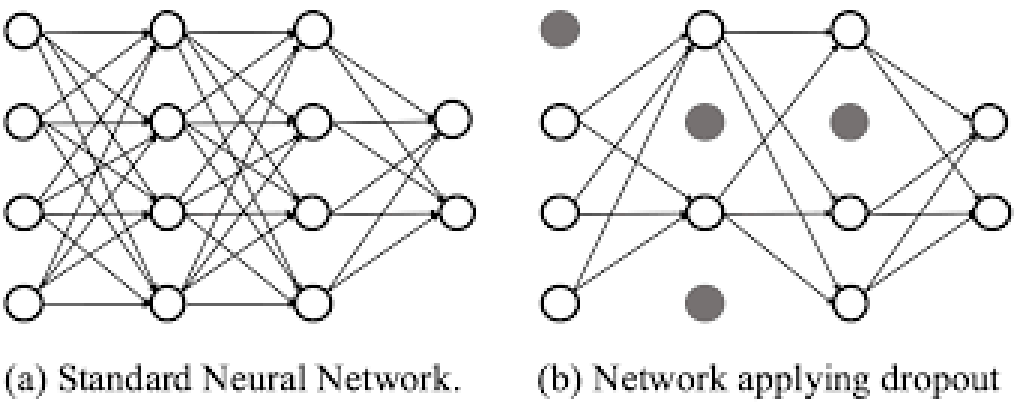


Figure 48. Dropout strategy visualized.

Referring to Figure 45, notice that *Dropout Layers* are only present in the *rDNN*. Even though it is considered an effective method to overcome *overfitting*, another method was already implemented in the *Classification Deep Neural Network*. Some preliminary trials verified that it was enough to avoid overfitting. Moreover, it is added sequentially using *Keras* environment.

4.6.2. Early stopping

As the name suggests, *early stopping* stops the training process in advance with respect to when it was design to. This method is usually adopted to avoid *overfitting* when it is happening; so, it does not prevent it, rather it stops it. Even though being simple and usually effective, it is not used in the present project. The reasons for this choice will be clearer later, however in general the authors where interested in analysing the whole training process and, if the procedure was stopped, this would not be possible.

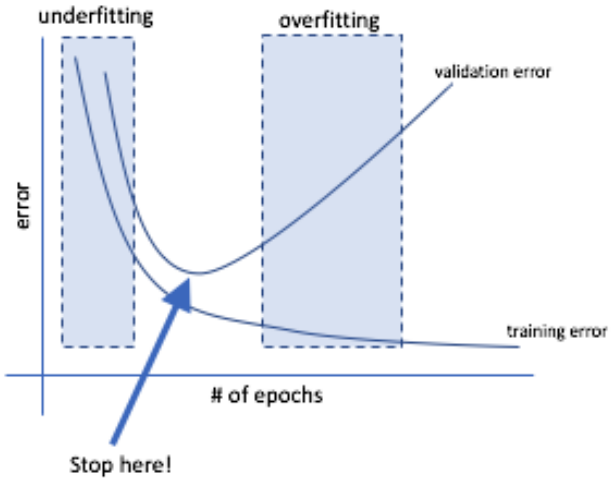


Figure 49. Early stopping on loss function.

Referring to Figure 49 the benefits of such a technique are evident. It should be stated that the *loss function* is not the only quantity that can be monitored in order to apply early stopping. One of the selected metrics could be monitored, namely R^2, the *MCC* or *Accuracy*. However, it should be kept in mind that the two approaches not always coincide, this is true especially for classification tasks. In fact, a lower loss function (*Cross Entropy*) does not always imply higher *Accuracy*. This is because classification tasks are based on a *Decision Thresholds*. Once again, this theme will be deepened later.

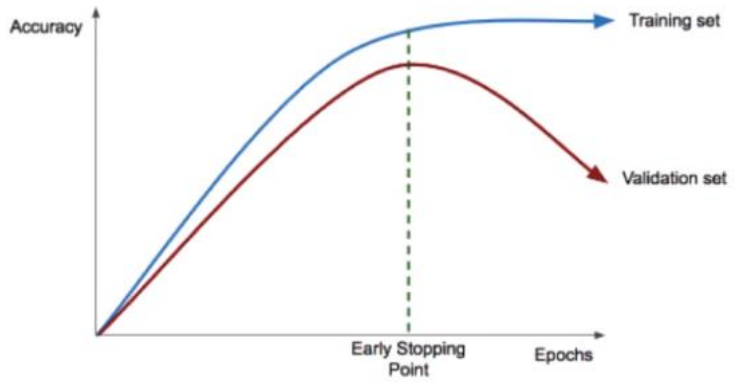


Figure 50. Early stopping montoring Accuracy.

Moreover, referring to Figure 46 and 47, notice how the *MCC* is more noisy than R^2. *Early stopping* is actually very sensible to noisy signal, especially if *patience* coefficient is not used. *Patience* is a parameter that controls how many epochs the algorithm waits before stopping the *training procedure* in hopes of future improvements. This is another reason for which *early stopping* is not here implemented.

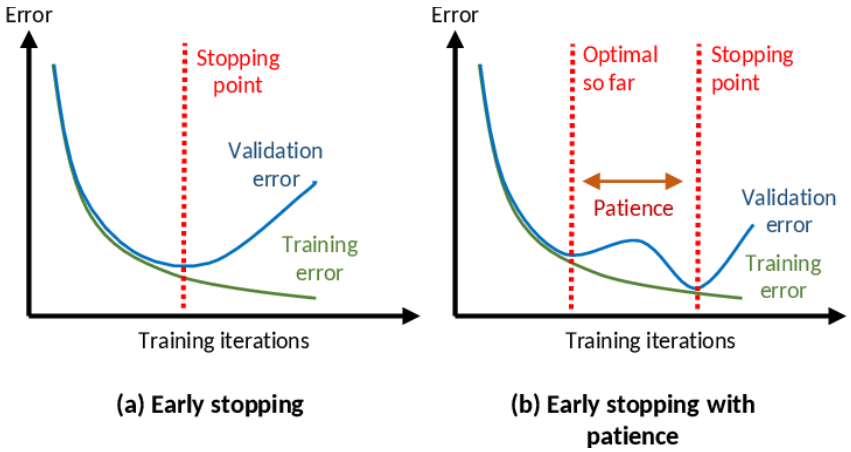


Figure 51. Early stopping with patience.

4.6.3. Regularizer

Regularization is another common technique used to tackle the *overfitting issue*. The logic is simple: since the weights' update depends directly on the *Cost Function*, we can add a penalty term to the *Cost Function* that takes into account the size of each term. So, being the goal of the algorithm to minimize the *Cost Function*, a side effect of the whole process will also be to keep the weights' size. In practice:

$$J_{tot} = J_{loss} + J_{reg}$$

We have already described how low values for weights help to have better prediction and in particular more stable model. This effect can be thought using the concept for which the net becomes more sparing in assigning weights values and, instead of relying on particular nodes excessively increasing their size, relies on the whole set of neurons.

The most common used *regularizer terms* are for sure *L2 (or Ridge) regularizer* and *L1 (or Lasso) regularizer*. The difference is in the penalty term used.

L2 regularizer term is the following:

$$J_{reg} = \frac{1}{2} * \lambda * \sum_i |w^2|$$

The term used in the *Ridge regularizer* as we can see depends on the sum of the squared weights. So, thinking at back propagation, the derivative term with respect to al the weights will have a direct dependence on the size of the weights that will tend to decrease it.

L1 regularizer term is instead:

$$J_{reg} = \frac{1}{2} * \lambda * \sum_i |w|$$

It is evident that the only difference is the lack of a square in the summation term. This will lead to a *constant* derivative term. If the weights are reduced more and more toward zero, with *Lasso regularizer* we can achieve the "*zeroing*" of the weight. This is good for model compression since it reduces the computational effort of the algorithm (you do not need to calculate path with zero weights). However, this can also lead to a loss of predictive power for the same reason. *L1* is usually preferred for very sparse signals.

The coefficient λ indicates the "aggressiveness" of the penalty applied to the *Cost Function* and in turns to *weights' updates*. It is straightforward to understand why: the larger λ, the greater the penalty and the reduction of the weights.

In this application is preferred the *L2 regularization* after few preliminary trials. It will be applied to every *hidden layer*.

4.7. Batch normalization

Batch normalization is an incredibly powerful tool to implement in any neural network application, and especially in *Deep Neural Network* applications.

As explained in [19], during the training of a neural network the distribution of the weights can change drastically, this phenomenon is defined by authors as *Internal Covariate Shifts* and is considered a cause of possible slowdowns during the learning process.

The functioning of this technique is fairly straightforward: after the computation of the outputs of each neuron for a specific layer, the mean and standard deviation of all the features across the *batch* are computed; they are then used to normalize the all the outputs. Moreover, some scaling factors are introduced in order to be able to intervene on the resulting distribution. The general procedure can be summarized with the following expressions:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

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

$$\bar{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

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

Where:

- m: number of examples in the minibatch.
- x_i: output of the neuron.
- μ_B: mean value of the batch's single feature outputs.
- σ_B^2: variance of the minibatch.
- $\bar{x}_i$: output after normalization.
- y_i: final output after shift and scaling.
- γ, β: learnable parameters.

This technique has been proven to generates faster convergences and in general higher performance.

it is however affected by some criticality:

- *Variable batch size*: If batch size is of 1, then variance would be 0 which does not allow batch norm to work. Furthermore, if we have small mini-batch size then it becomes too noisy and training might affect. There would also be a problem in distributed training. As,

if you are computing in different machines then you must take same batch size because otherwise γ and β will be different for different systems.

- *Recurrent Neural Networks (RNNs)*: In an RNN, the recurrent activations of each time-step will have a different story to tell (i.e. statistics). This means that we must fit a separate batch norm layer for each time-step. This makes the model more complicated and space consuming because it forces us to store the statistics for each time-step during training.

Batch normalization is actually not the only *norm technique* that has been developed. Some of the others are *Weights Normalization, Layer Normalization, Instance Normalization* and *Group Normalization*.

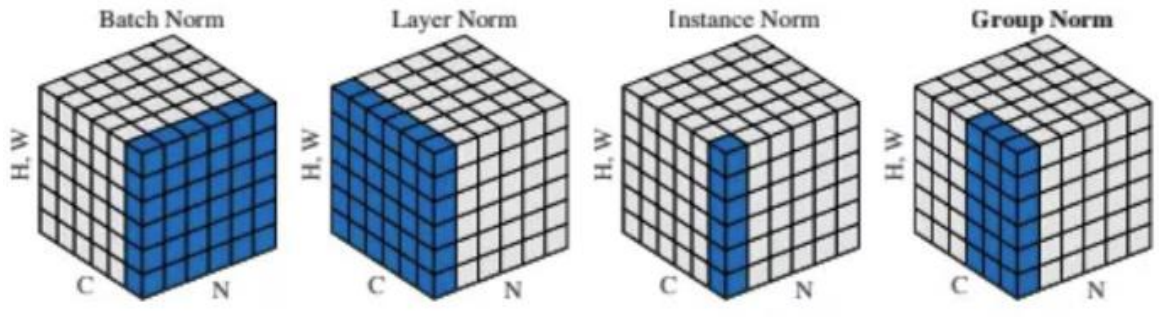


Figure 52. Visual comparison of different normalization techniques. N is the batch dimension; C is the channel/feature dimension.

These techniques differ from each other on the basis of the "dimension" across which the normalization is performed:

- *Batch-norm*: it normalizes single features across the batch.
- *Layer-norm*: it normalizes single examples across all features.
- *Instance-norm*: it normalizes single instances.
- *Group-norm*: in-between technique.

Moreover, a *switchable normalization technique* has been proposed which is able to switch from a method to another depending on which technique performs the best.

4.7.1. Keras batch-norm issue in cDNN

Referring to Figure 45, it is evident that a *batch-norm layer* is implemented after each hidden layer. This produced huge benefits to the predictive performance of the net from the very early stages of the project. However, some criticalities have emerged too.

In particular, a very strange behaviour of the *cDNN* was spotted from the *learning curves*. This is a practical proof of the extreme importance of such an analysis. The trend abovementioned was evident only in the *validation* learning curves; the *learning algorithm* was therefore not believed to be the cause.

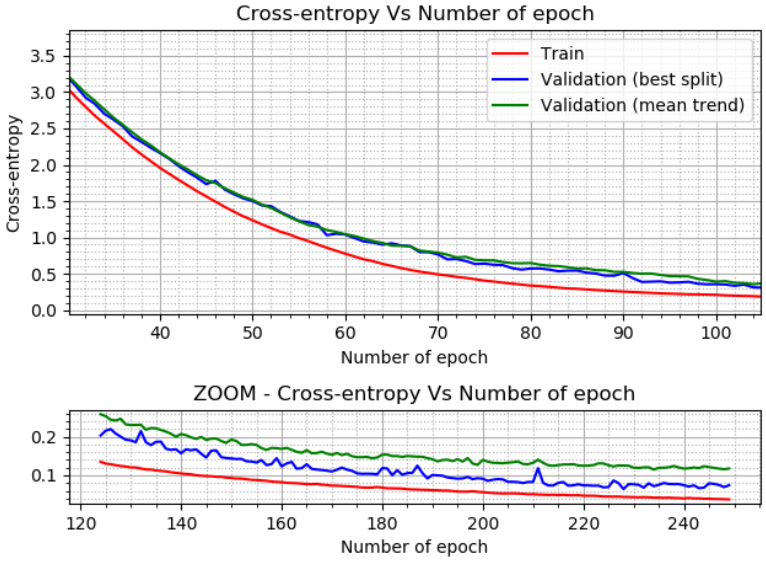


Figure 53. Cross-entropy (upper part is also zoomed-in) to varying of the epochs.

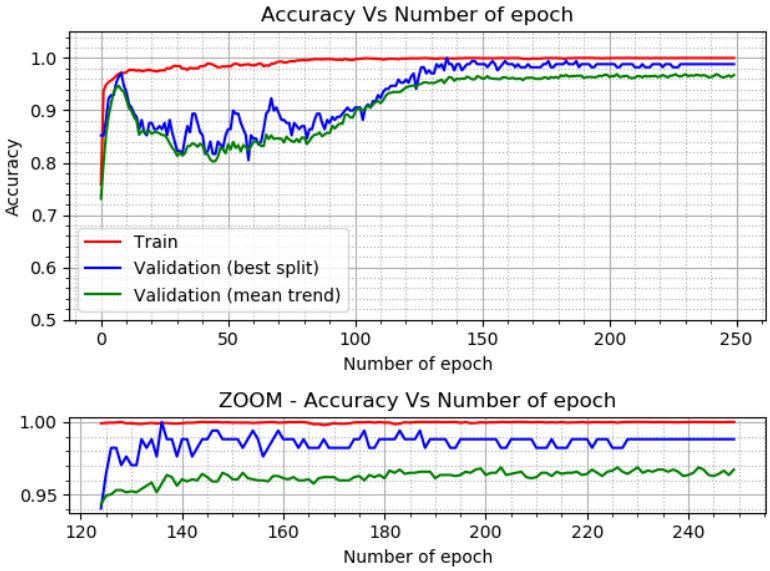


Figure 54. Accuracy to varying of the epochs.

Figure 53 and especially Figure 54 show the abovementioned behaviour. As it is evident, there is a critical decrease of the *Accuracy* starting from about epoch 20 to epoch 50; then the *Accuracy* goes up again to stabilize around epoch 120. Notice how the performance at the final epoch are however acceptable and seem to not be influenced by this behaviour. Anyway, it was considered important to understand the causes of this trend.

The *Loss Function* curve have not showed any particularly strange behaviour, or better, the author was not able to spot one; this is due to both the scale used and the formulation of the *Loss Function* for the classification task, namely the *Cross-Entropy*. Notice that, even though the *Matthews Correlation Coefficient* was considered the main metric in evaluating the performance, *Accuracy* is here selected to present the results because of its straightforward meaning and interpretation.

For clarity, the author proposes again the formulation for the monitored quantity:

- *Cross-Entropy*: $J(\theta) = \frac{1}{n}\sum_{i=1}^{n} y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i)$
- *Accuracy*: $Acc = \frac{TP+TN}{P+N}$

Should be clear by now that *Cross-Entropy* is a "value-related" quantity; by this expression, the author means that the value it assumes depends on the difference of the *true labels* and the *predictions*. Now the *predictions* are a continuous distribution from 0 to 1; the *Cross-Entropy* is sensible to all the possible values, regardless from the predicted class. At the same time, *Accuracy* is instead a "class-related" quantity; this means that it is only sensible to change in class, driven by a *decision threshold* that usually, and also in the present application, is set to 0.5.

Let us analyse a practical example: suppose monitoring a *prediction* for a specific example which *true label* is *feasible (1)*. The prediction to varying of the epochs goes at first from 0.3 to 0.49: being all the predicted values under 0.5, the predicted class will be 0, so the Accuracy will also be 0 since no correct prediction was made however, the *Cross Entropy* is decreasing; the learning goes on and the *prediction* goes from 0.49 to 0.51: now that the *threshold* is crossed the *Accuracy* jumps to 1 but the *Cross-Entropy* does not change much since the values are only 0.02 apart; in the last segment of the training the *prediction* finally converges to 0.96 from 0.51: the *Accuracy* stays the same but the *Cross-Entropy* is drastically decreased. The example just presented explains the possible discrepancies between the two metrics here used. The same concept applies to *MCC* and it indeed showed the strange trend.

To correctly analyse the problem, it has been implemented a module that is able to monitor the progression of a prediction throughout the learning process. The example is chosen randomly, and the procedure was iterated several times observing the same behaviour for numerous examples.

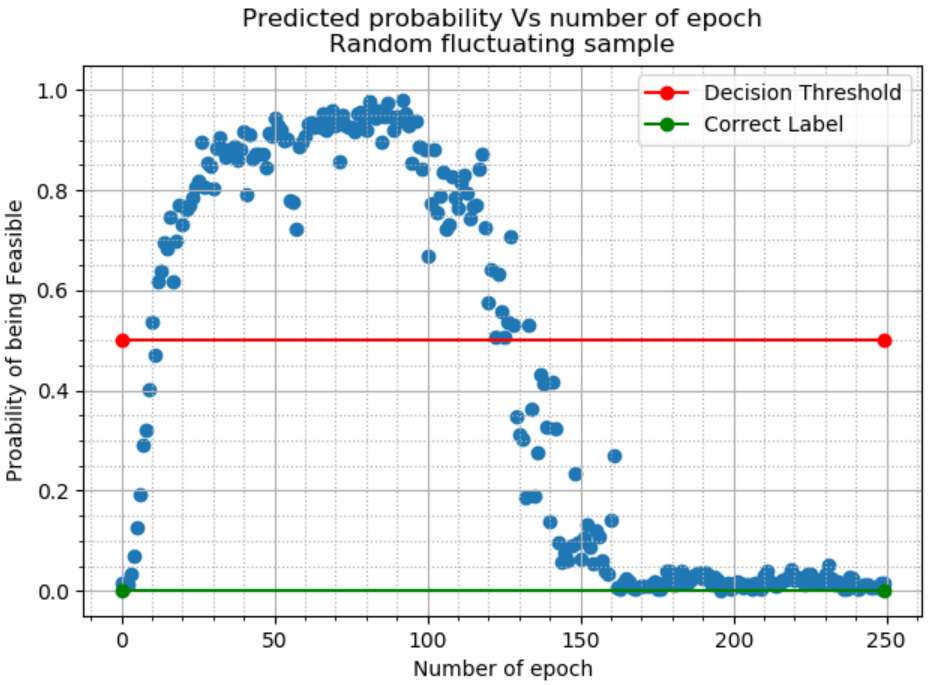


Figure 55. Randomly selected example fluctuating prediction.

Figure 55 describes exactly what is happening for several examples. They pass from being predicted correctly to being misclassified, then they come back to correct classification.

Several possible causes were explored and in particular:

- *Overfitting*: the classical diverging behaviour of the *Loss Function* is however not present and after 100 epochs the trend seems to disappear. This is not the standard indication of overfitting and classical remedy seemed indeed not to work. This possibility was then discarded.
- *Stochastic nature of the algorithm*: as we already know, Adam optimizer is a particular kind of mini-batch gradient descent algorithm. So, the algorithm has a stochastic core functioning, based on the random selection of the mini-batch to process at each pass. This was the most promising reason however, after researches in the literature, no similar cases were found. For this reason, this possibility was also discarded.
- *Batch-norm layer*: after methodical tries on different sections of the program, the author observed that deactivating the batch-norm layer the behaviour seemed to disappear.

At this point further researches proved that indeed the *Batch Normalization Layer* was the cause of the trend. In particular, the issue depends on how the normalization is actually performed in *Keras environment.*

4.7.2. Keras computation logic

At its core, *Keras environment* include two different computational stages of phases. In particular, as the website of *Keras* itself explains, it uses "*training mode*" during training phases and "*inference mode*" for all the phases that somehow involves predictions without backpropagation. This means that during *Validation*, *Keras* works in *inference mode*.

The main difference between the two modes, for what concerns the issue here addressed, lays in the computational procedure of the means and variances needed for the normalization. More in details:

- *Training mode*: mean and variance are calculated for the single batch. So, for each batch the following operations are performed:
  - Computation of mean and variance for each neurons output across the whole batch (so if the layer has 3 nodes and the batch has 10 examples, it will produce 3 means and 3 averages computed across 10 values each)
  - Standardization of all the output using the mean and variance just computed.

- *Inference mode*: mean and variance are mobile averages computed as the training advances. So, the procedure is the same, however the mean and variance used for the normalization is computed as follows:

$$m_{new} = \beta * m_{old} + (1 - \beta) * m_{new\ batch}$$

Where:

- $m_{new\ batch}$: is the statistics coming from the latest *training* batch.
- m_{old}: is the statistics coming from the penultimate *training* batch.
- m_{new}: is the statistics used for current batch in *validation*.
- β: is the momentum term.

Momentum term β is the key concept regarding this issue. It dictates how fast the moving average adapts to changes in the considered statistics. Observing the formula, it can be noticed that a new value will give a contribution of "$(1 - \beta)$". Consider that the standard value chosen by *Keras* for β is 0.99. This means that a new value will contribute for only 1% of the average. The problem with this approach is then that the average statistics are very slow to adapt to the changes. This in turns cause the *Batch-Norm Layer* to use completely inconsistent statistics.

Indeed, after the first *back propagation* the weights will be adjusted causing the neurons' output to change accordingly. The newly adjusted weights however generate outputs that are completely out of range with respect to the one used before: their distribution is shifting. This finally leads to certain neurons' paths being extremely fallacious, hence the first branch of the trend in Figure 55.

After some epochs, the moving averages finally catch-up with the actual outputs of the single neurons. *Validation predictions* then slowly converge to more reasonable ones. This explains the descending branch of Figure 55.

To overcome the problem a β value of 0.85 has been introduced.

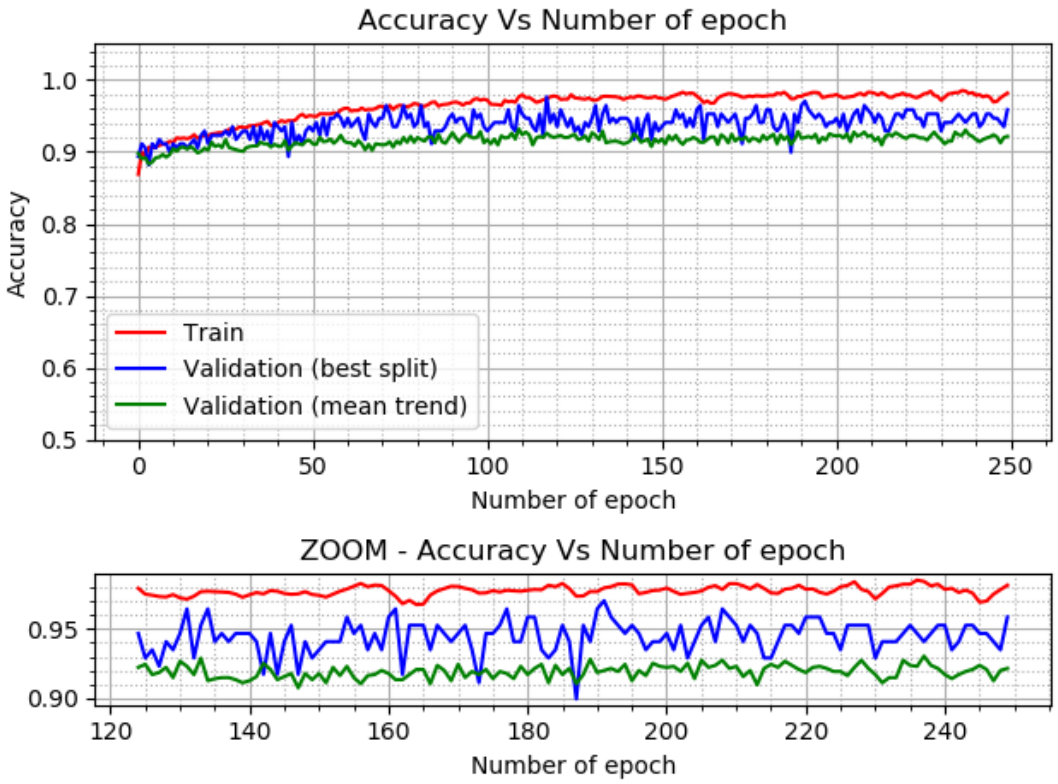


Figure 56. Accuracy trend to varying of the epochs after decreasing momentum term to 0.85.

As it is evident from the graph in Figure 56, the trend that was present before is now completely gone.

4.8. Hyperparameters' space and random search logic

At this point, the general architecture of the two nets is available. Moreover, all the tools that the *pipeline* is going to use are implemented correctly. Last step is therefore to define the *Hyperparameters' space (hyperspace)* and the logic behind the selection process.

It has already been discussed how a well-selected *hyperparameters* can undoubtably increase performance. The general idea behind an automated algorithm for their selection is to make the pipeline able to adapt to different scenarios and datasets. Moreover, it is not granted that the same net will be selected every time: each time the starting dataset is shuffled in a different manner, so the whole process becomes extremely stochastic.

In the following section will be described the *hyperspaces* of the two nets and reasons behind their choices.

4.8.1. Hyperspace definition

Should be kept in mind that even if some preliminary analyses are needed to spot some general trends, focusing too much on one-dimensional analyses is a mistake. The performance of a *Deep Neural Network* strongly depends on the combination of its *hyperparameters* rather than on their specific value.

For this reason, the strategy adopted in the context of this project is different from the one adopted in the pre-existing one. Rather than executing a big set of one-dimensional analyses, a quite large *hyperspace* is defined, and long simulations are deployed.

For the *cDNN* the *hyperspace* is 5-dimensional, and it is the following:

Table 4

cDNN hyperspace	
Learning rate	0.00001 – 0.1
Hidden Layers	1 – 15
Neurons first hidden layer	20 – 300
L2 regulrizer	0.0001 – 0.09
Batch size	16 – 516

The *hyperspace* for the *rDNN* is kept the same as in the pre-existing project; it is 6-dimensional.

Table 5

rDNN hyperspace	
Learning rate	0.005 – 0.5
Hidden Layers	1 – 6
Neurons first hidden layer	30 – 80
Batch size	8 – 64
Weights initialization	*Xavier, Random, truncated normal*
Dropout	0 – 0.7

Notice that the activation functions are ReLUs for the hidden layers of both nets; *cDNN* has a sigmoid activation function as output and *rDNN* has again a ReLU. Adam is the optimizer of choice of both nets.

It can be noticed from the two tables that is present an *hyperparameter* called "Neurons first hidden layer"; with this expression the author is referring to a design choice operated in the pre-existing project. Since it was too expensive to search for a specific number of neurons for each layer, it was decided that each hidden layer should contain half the neurons of the previous one. The relation is therefore the following:

$$N_i = \frac{N_{i-1}}{2}$$

Where N_i is the number of neurons for the current layer; N_{i-1} is of course the number of neurons of the previous one.

For completeness, the *dropout* coefficient is indicating the probability to deactivate the neuron. Moreover a L2 regularizer of 0.01 is chosen for the *rDNN*.

4.8.2. Hyperparameters' selection logics

The algorithm behind the hyperparameters' choice is based on random-search. The reasons for this choice have already been explained; anyway they can be summarized by stating: for a given amount of time, considered not enough to explore the entire hyperspace, random search is likely to perform better or at least as good as grid search. Bayesian optimization was proven to be effective, however due to difficulties in implementing it, it was discarded.

Also in this case there are two separate algorithms, one for the *cDNN* and one for the *rDNN*. Since the algorithm developed in the context of the pre-existing project was thought to be valid, it is leaved as it is, only slightly modified to overcome an issue that was spotted in the code. The abovementioned issue could have led to the discard of some configurations that was actually better than the ones retained.

Just to resume, a *Random Search* is based on the concept that, after a *hyperspace* is defined, a number of random configurations are selected and validated; the *validation procedure* that is chosen is a *k-fold cross validation* able to ensure reliable estimate of a configuration performance in a possible future application.

The algorithm dictating the choice of the *cDNN*'s hyperparameters is the following:

1. The whole *hyperspace* is divided into smaller "*hypercubes*" (they are just *sub-spaces*). To accomplish this result is sufficient to set a value for the number of sectors each axis should be split into; the algorithm automatically considers the resulting *hypercubes* in the following steps. For this application, a standard value of 3 is chosen for each axis.
2. A set of combinations is selected from the whole *hyperspace*. In the first stages of the study this set was quite large and included 60 tries to cope with the vastity of the *hyperspace*. The goal of this preliminary search is to select promising *sub-sectors* in which perform a finer search. To accomplish this, a *threshold MCC value* is set as a control. Each *sub-sector* that shows at least one combination performing better or equal to that *threshold* is selected for next step.
3. A new set of combinations is choses for each promising *sub-sector*. The best performing one is selected as the one representing the whole *sub-sector*.
4. After point 3 is complete, all the combinations selected for each *sub-sector* are compared and the best performing one is selected to be tested.
5. Final training is performed.
6. The resulting *cDNN* is tested on the *Test Set* and the results are analysed.

Notice lastly that each evaluation is the result of a *k-fold cross validation* with *K=8*. Moreover, it should be kept in mind that the best configuration is considered such on the basis of its average *MCC* score during *cross validation*; the procedure, as explained in Section 2.9, includes the random selection of portions of the *train/validation set* to be assembled in a *training set* and a *validation set*. This means that there will be a "*best performing split*". The *final training* at point 5 is performed choosing as *training set* the just mentioned split. Finally, if less than three *sub-spaces* performed well enough, the three best performing are selected and further analysed.

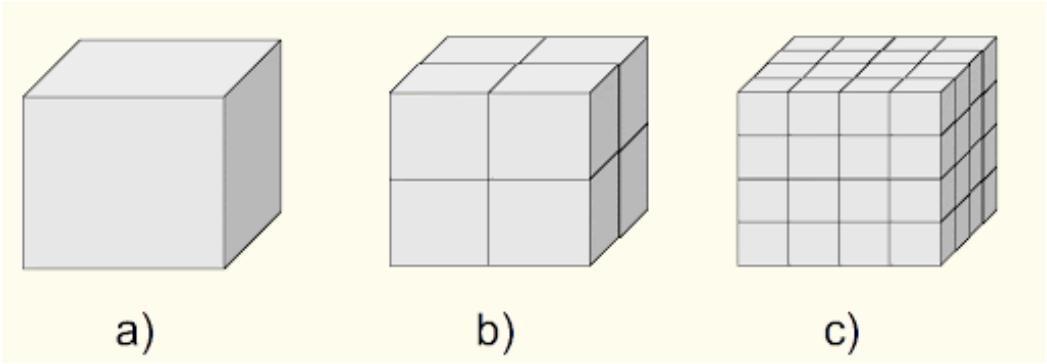
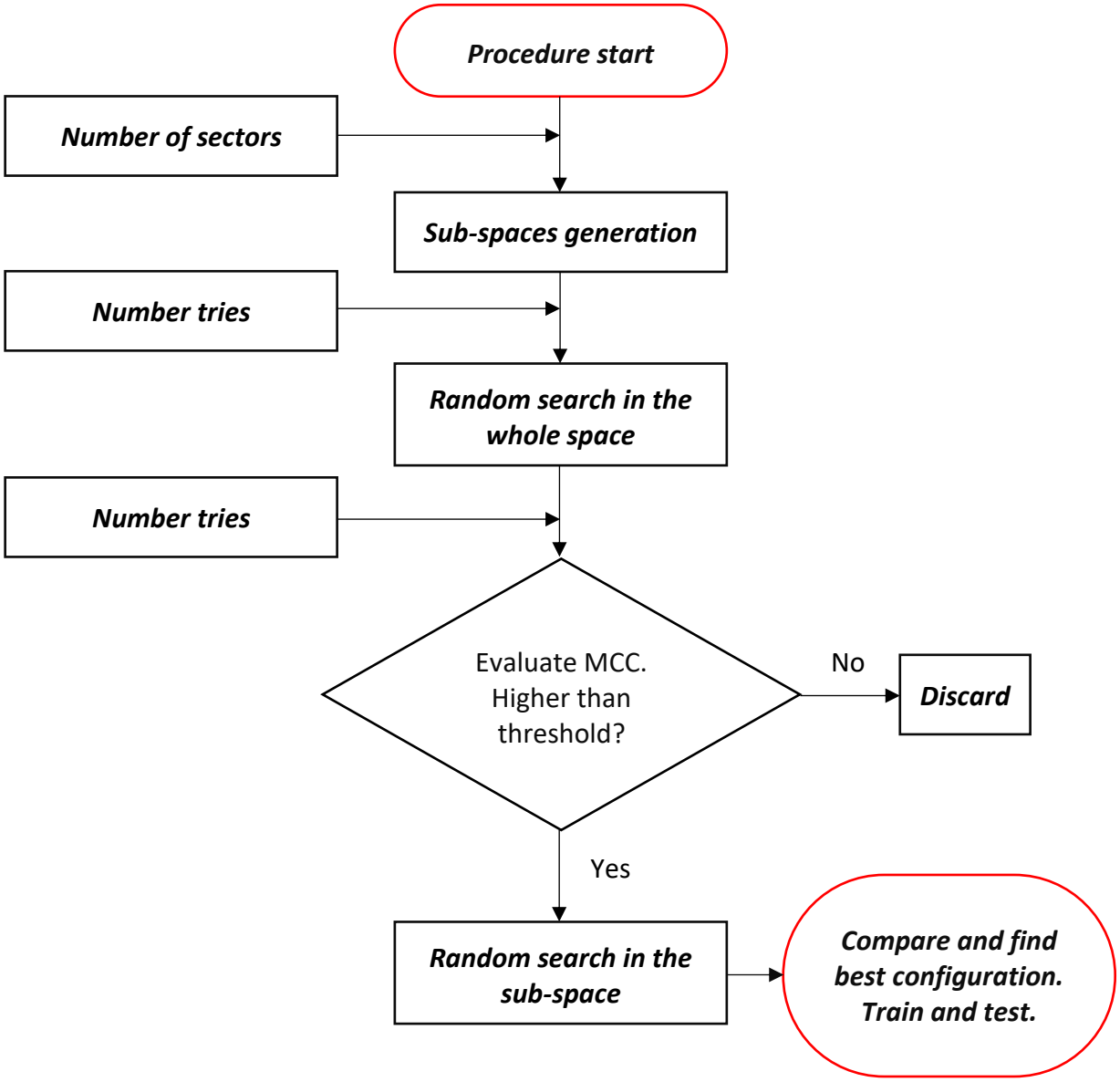


Figure 57. Sub-sectors generation. (a) whole hyperspace; (b) 2 sectors per axis; (c) 4 sectors per axis.



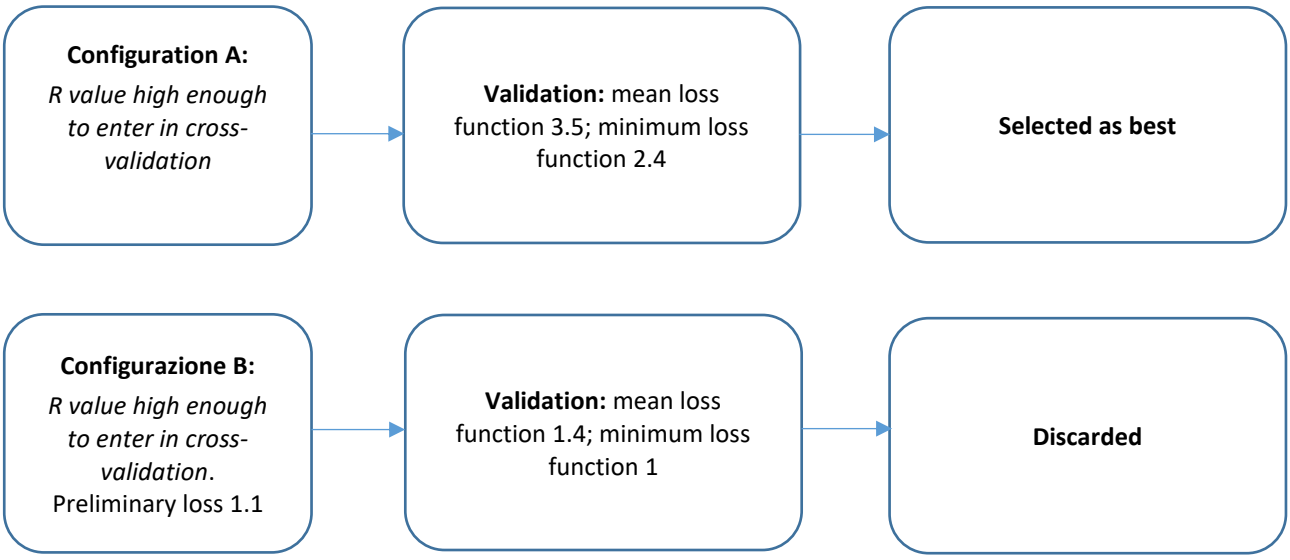
Main feature of this algorithm is that is always able to produce at least one configuration, regardless of the specific *MCC score*. This choice was made to avoid the possibility of *never-ending loops* showing up in case of wrong threshold being set.

The algorithm behind the *rDNN hyperparameters' selection logic* is the following:

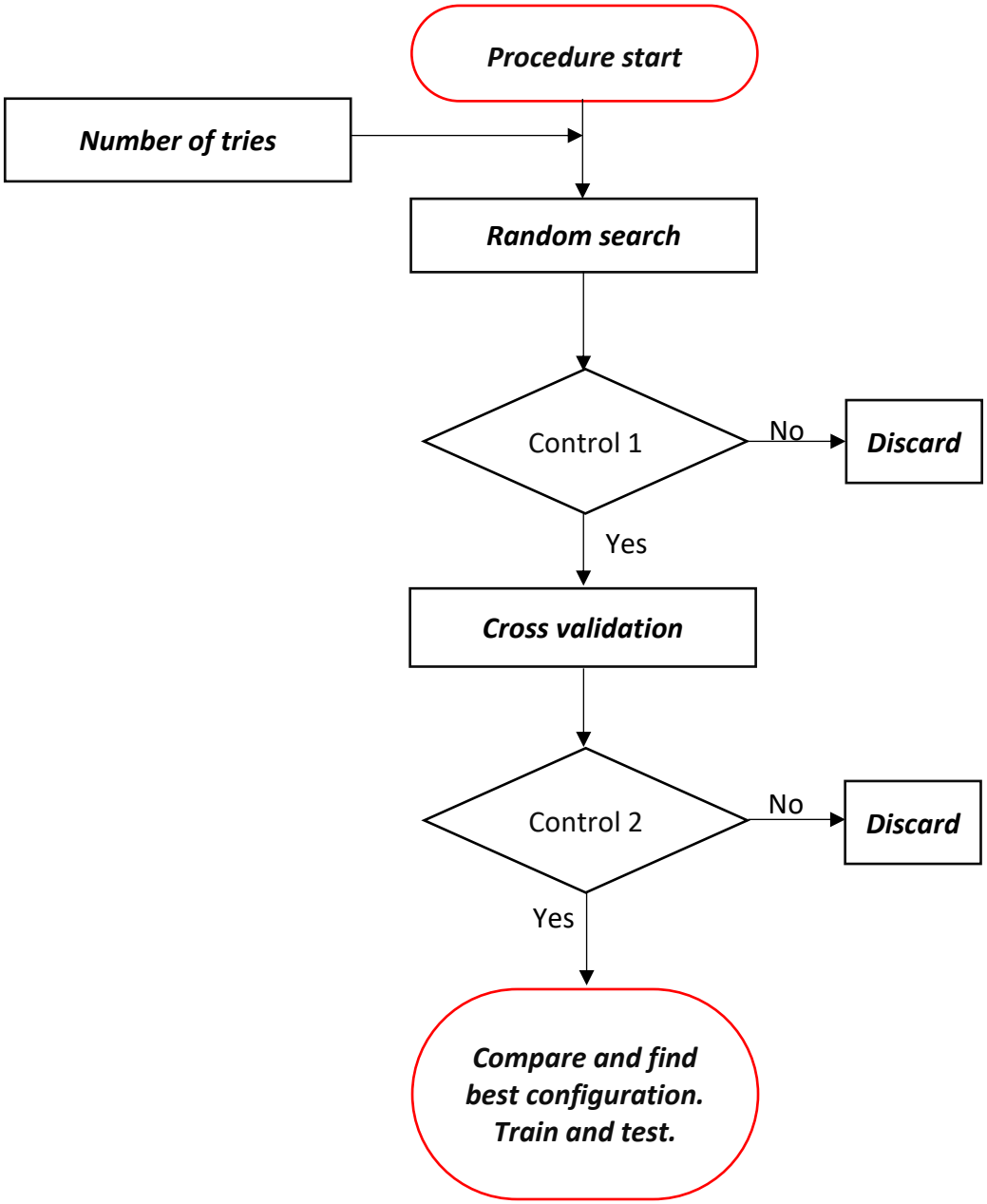
1. A random search is started.
2. If the configuration shows a high enough *R score,* then a *5-fold cross validation* is performed.
3. Now the procedure splits in two branches:
    a. If no configuration has been selected yet, the mean value of the loss function of the 5 folds should be at maximum 10% higher than the one showed in the preliminary analysis. This is to ensure a stable-performing configuration.
    b. If a configuration has already been selected, sufficient condition is to show a mean loss function across the 5 folds that is lower than the already selected one.
4. When all the pre-defined tries have been extinguished, the procedure saves he best performing configuration and trains it on the best split of the cross-validation procedure.
5. The resulting net is tested, and the results analysed

Notice that if at the end of the random search no configuration is selected, even though experience shows that this is a very remote case, a new random search is started. It is evident that the procedure can actually get stuck in a loop. However, this have never happened in any of the simulation performed in the context of this project.

This procedure is the one implemented by the author of this project based on the pre-existing one. As already stated, the previous one showed a criticality: the control that now is step 3.a was too restrictive. It also provided for a comparison between mean loss function value and preliminary value, but it would discard the configuration if the mean were higher than the preliminary one. This could have led to the discard of configurations that showed better performance of the already selected one. For his reasons, point 3 was split in two cases as already explained.



The procedure can therefore be schematized as follows.



Where "Control 1" is point 2 and "Control 2" is point 3 (that distinguish two cases).

Results and discussion

5.1. Preliminary analyses for the cDNN

Before exposing the final simulations and results some preliminary analyses need to be accounted for. It is reminded that the main goal of the project is to produce a tool that is able to be used in substitution of (or in combination with) *Dynamic Programming*. The chosen technology for the achievement of this goal is *Deep Learning* and in particular a *Pipeline* of two *Deep neural Network.* Given the abovementioned considerations, the final results of the project are considered the ones regarding the whole pipeline, even though the vast majority of the whole study has been to produce the code relative to the *Classification Deep Neural Network (cDNN)*.

The starting point of the whole project, besides the study of the theory and the pre-existing project, has been to select a dataset from the three at disposal. After this section, different analyses will be presented. They include:

- Preliminary sensitivity analysis and detailed sensitivity analysis.
- False negatives and false positives trend.
- False negatives distribution.
- Multiclass classification net for architecture recognition.

5.1.1. Database selection

Three starting databases were available, corresponding to P2, P3 an P4 architecture. However, the analysis of all three datasets would have meant too much time and resources. For this reason, a preliminary study has been conducted across the three databases to assess which one was the worst performing.

The simulations included:

- 60 tries for the first stage random search.
- 60 tries for each sub-sector random search.

The *hyperspace* was the following:

Table 6

Database selection hyperspace

Learning rate	0.00001 – 0.1
Hidden Layers	1 – 15
Neurons first hidden layer	20 – 300
L2 regulrizer	0.0001 – 0.09
Batch size	16 – 516

Notice that it coincides with the one presented in Section 4.8.1.

The graph that follows resumes the three abovementioned simulations.

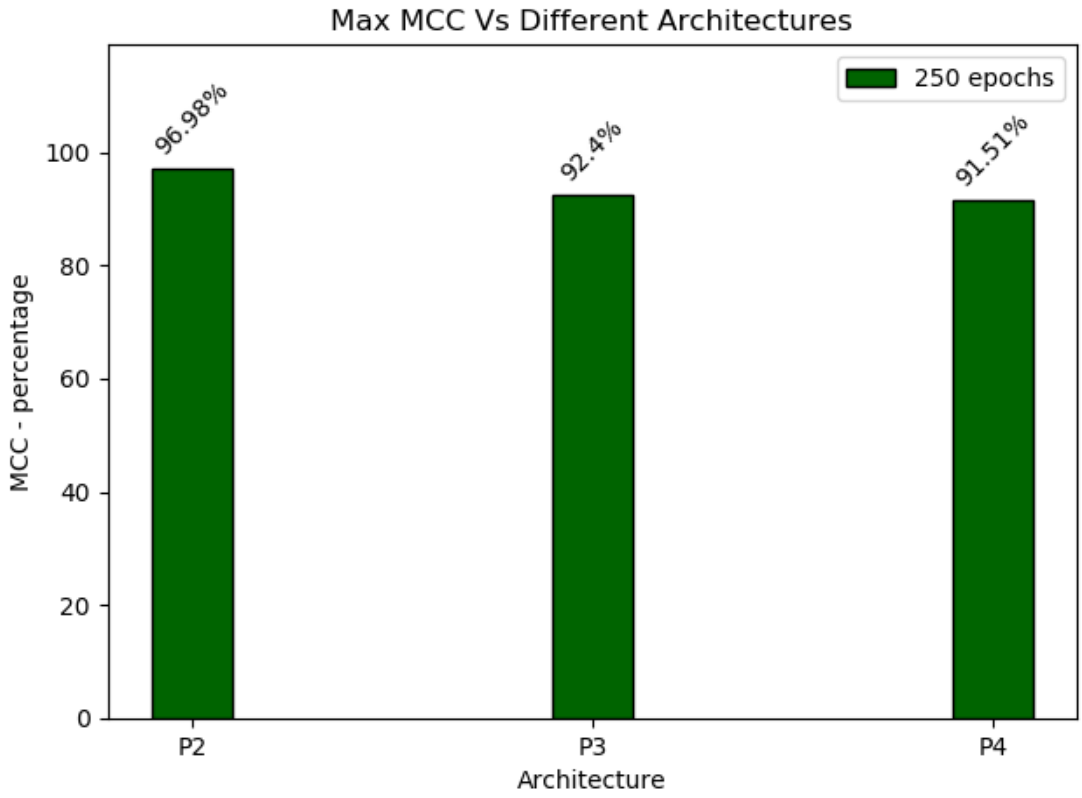


Figure 58. MCC for the three architectures. Preliminary analyses.

The results were quite interesting: for no evident reasons, and in many other "manual" simulations, P4 database was performing at lower performances in terms of *classification*. Many observations regarding the distribution of the design parameters were made, however no decisive conclusions have been made.

One possible explanation was given looking at two design parameters, that are actually the ones that distinguish from P4 and P2/P3, at least in for the algorithm. As we know, P2 and P3 architecture use a speed coupling device to connect the *Electrical Motor* to the transmission system in a point that depends on the specific architecture. Instead, P4 directly connects the *Electrical Motor* to the secondary axle. So P2 and P3 architectures includes a parameter called "EM1SpRatio" (the speed-coupling device transmission ratio), and P4 includes a parameter called "FDsSpRatio" (transmission ratio at the secondary axle differential).

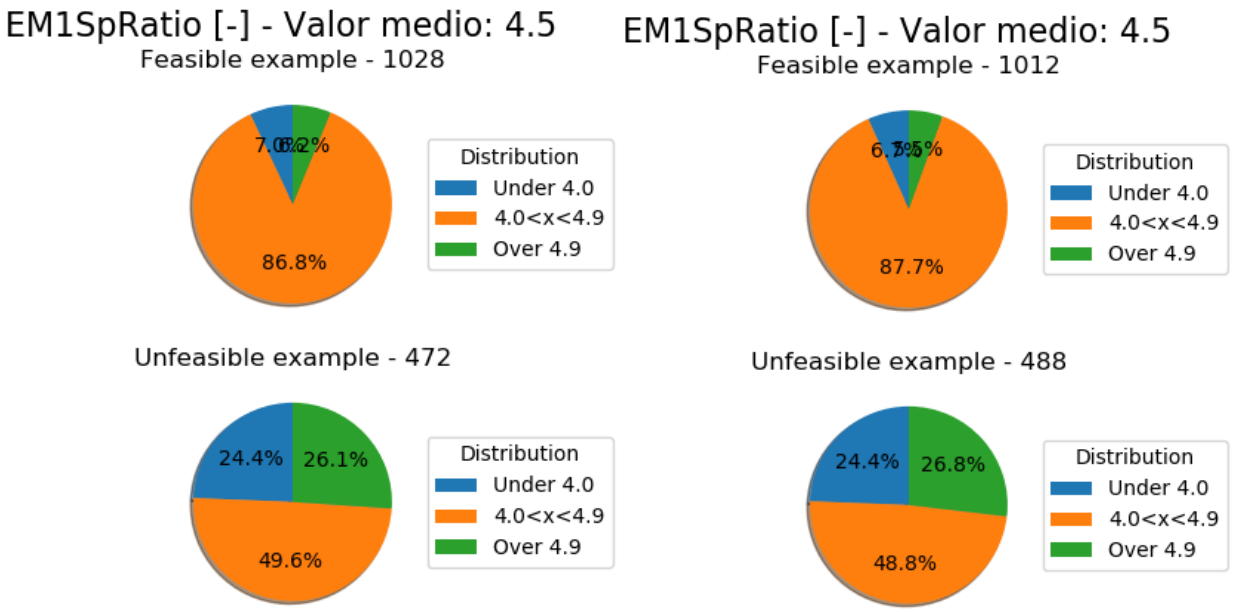


Figure 59. EM1SpRatio for P2 an P3 architectures.

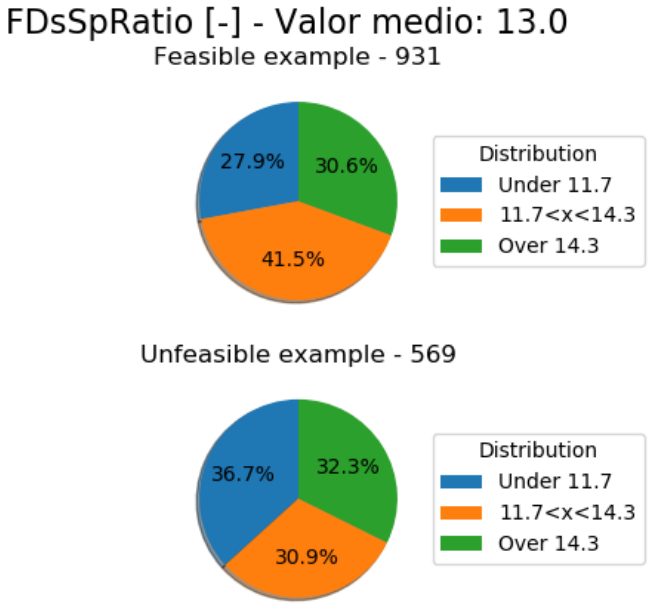


Figure 60. FDsSpRatio for P4 architecture.

The graphs above clearly show how the "EM1SpRatio" is more "characterizing". With this expression the author means that, using the abovementioned feature, the *cDNN* could acquire a trend more easily since the vast majority of the *feasible* example have a value for that feature that is under the mean value of the entire distribution. This phenomenon in not present in "FdsSpRatio".

For the reasons here described, P4 dataset is chosen as the one to perform the simulations on.

5.1.2. Sensitivity analysis

One of the key points of this project is to monitor the *classification performance* at varying of the *training set size*. The assumption behind this procedure is that the performance should benefit from an increased *training set*.

Since it is not possible to increase the examples included in the database, the performance of the net is monitored at varying "*train+validation/test split*" (*t/t split*). Should be clear by now that reducing the portion of *examples* used for *testing*, more examples will be available for *training*. It should be reminded that it is crucial to use some sort of *cross-validation*: the *k-folds* procedure in particular avoids misleading results caused by strong dependence on a specific training dataset; moreover, a series of simulations will ensure consistent results.

The simulations showed in Section 5.1.1 served another important goal. Since 60 tries per step would have meant too much time to carry on the needed simulations, and since too few tries would have led to poor *hyperparameters selection*, the *hyperparameters' space* is reduced around the optimal point selected by the algorithm in the abovementioned simulation. In particular, it highlighted the following hyperparameters combination.

Table 7

Database selection simulation results	
Learning rate	0.0021
Hidden Layers	2
Neurons first hidden layer	181
L2 regulrizer	0.03
Batch size	31

So, the hyperspace chosen to perform the other simulations is the following.

Table 8

Sensitivity analysis hyperspace	
Learning rate	0.0002 – 0.02
Hidden Layers	1 – 4
Neurons first hidden layer	130 – 230
L2 regulrizer	0.003 – 0.3
Batch size	16 - 128

At first, 8 simulations are performed to further analyse the problem. They are divided as follows:

- 2 simulations for 60/40 *t/t split*: one with 100 epochs training and the other with 250 epochs training.
- 2 simulations for 70/30 *t/t split*: one with 100 epochs training and the other with 250 epochs training.
- 2 simulations for 80/20 *t/t split*: one with 100 epochs training and the other with 250 epochs training.
- 2 simulations for 90/10 *t/t split*: one with 100 epochs training and the other with 250 epochs training.

The results are summarized by the following graph.

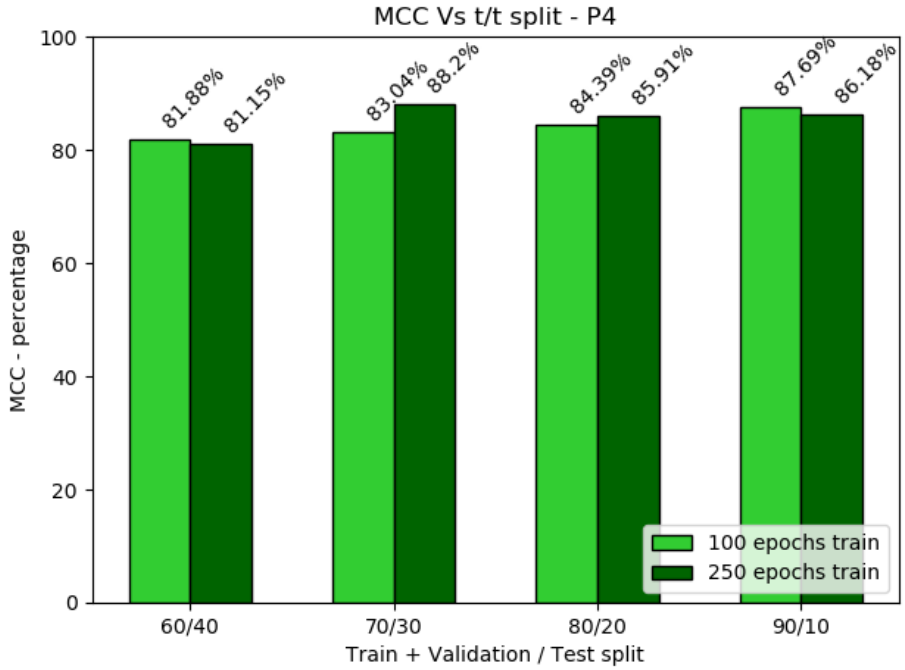


Figure 61. Matthews Correlation Coefficient at varying t/t split. P4 architecture.

Figure 61 clearly shows an interesting trend which seems to confirm the initial theory: the performance increase with increasing *training set*.

However, some values are considered outliers: one for all, the *MCC* of the 70/30 *t/t split* with 250 epochs train. It shows a peak of 88.2%, clearly out of the trend. For this reason, it is considered necessary to further deepen the situation with more simulations.

Notice also that in half of the simulations, 100 epochs train performed better than 250 epochs train and vice versa. This leads to the choice of 250 epochs for future simulations to have a complete picture of the training behaviour. Moreover, no clear overfitting-related issue are spotted.

In the context of these first 8 simulations, another study is carried out: the number of *False Negatives* and *False Positives* is monitored to spot possible trend. The results re the following.

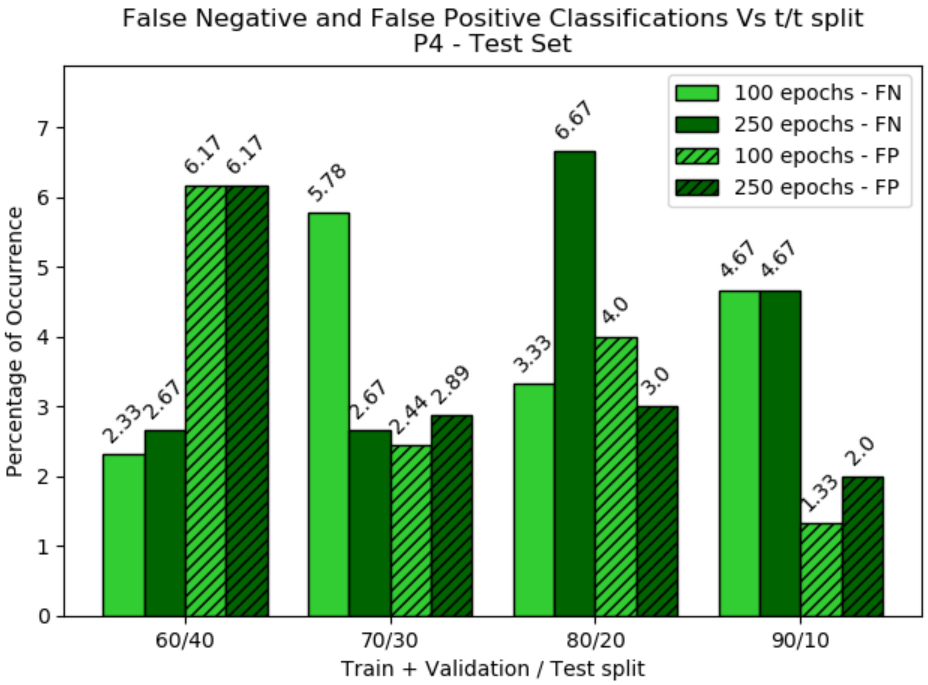


Figure 62. False negatives and false positives at varying t/t split. P4 architecture.

Notice that the values on each bar are normalized: they can be thought as a percentage of occurrence. Unfortunately, no particular trends are evident.

At this point, 24 simulations are performed, divided in the following manner:

- 6 simulations for 60/40 *t/t split*: 250 epochs training.
- 6 simulations for 70/30 *t/t split*: 250 epochs training.
- 6 simulations for 80/20 *t/t split*: 250 epochs training.
- 6 simulations for 90/10 *t/t split*: 250 epochs training.

The results of these simulations are averaged, and the standard deviation is computed and highlighted directly on the histogram.

Table 9

Matthews Correlation Coefficient - P4

	I	II	III	IV	V	VI	mean	std
90/10	85,79%	78,79%	80,36%	84,50%	88,79%	85,80%	84,01%	3,16%
80/20	82,92%	82,22%	87,22%	85,09%	84,39%	75,01%	82,81%	3,55%
70/30	82,08%	86,34%	81,59%	76,82%	81,63%	81,62%	81,68%	2,55%
60/40	81,88%	82,13%	76,46%	82,21%	79,69%	76,45%	79,80%	2,33%

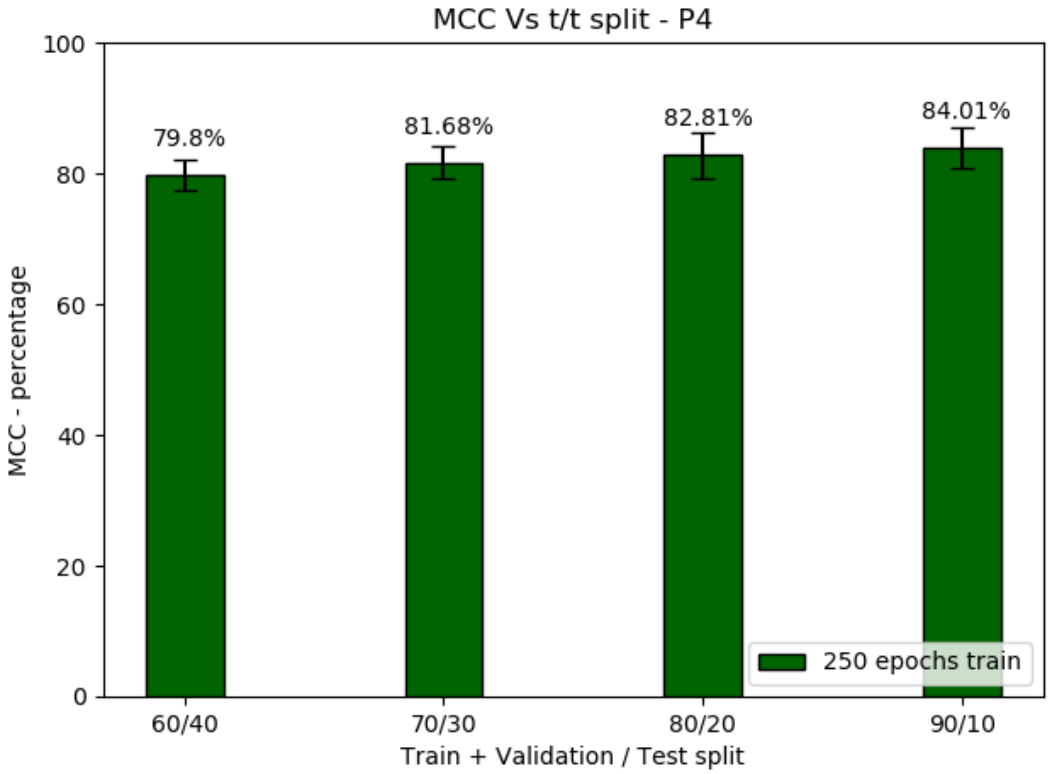


Figure 63. Matthews Correlation Coefficient at varying t/t split. P4 architecture.

The results of these simulations are surely more reliable than the previous ones since they are the outcome of 6 independent simulations for each *t/t split*.

The trend observed before is now clearer and undeniable: as the *training set* increase in size, the performance on the test set increase as well.

Now that various simulations with the same set-up are available, it can be observed something interesting. In the very first simulation, the one related to the *database selection*, the performance were higher; here however, they are lower and in all 24 simulations, never reached 90% *MCC*. This is believed to be a side effect of the reduced number of tries. Even though the *hyperspace* is reduced, a finer search is always going to be beneficial in the long run.

The number of *False Negatives* and *False Positives* is here analysed as well. In the tables, all the values are normalized on a 100-base, meaning that they represent percentage of occurrence once again.

Table 10

False Negatives - Normalized

	I	II	III	IV	V	VI	mean	std
90/10	3	7	5	4	3	2	4,11	1,65
80/20	3	3	2	3	2	4	2,89	0,66
70/30	4	1	4	5	4	2	3,52	1,32
60/40	4	1	6	3	3	4	3,47	1,40

Table 10

False Positives - Normalized

	I	II	III	IV	V	VI	mean	std
90/10	4	3	4	3	2	5	3,56	0,77
80/20	5	5	4	4	5	8	5,17	1,14
70/30	4	5	4	6	4	7	5,11	0,90
60/40	5	7	5	6	6	7	6,03	0,87

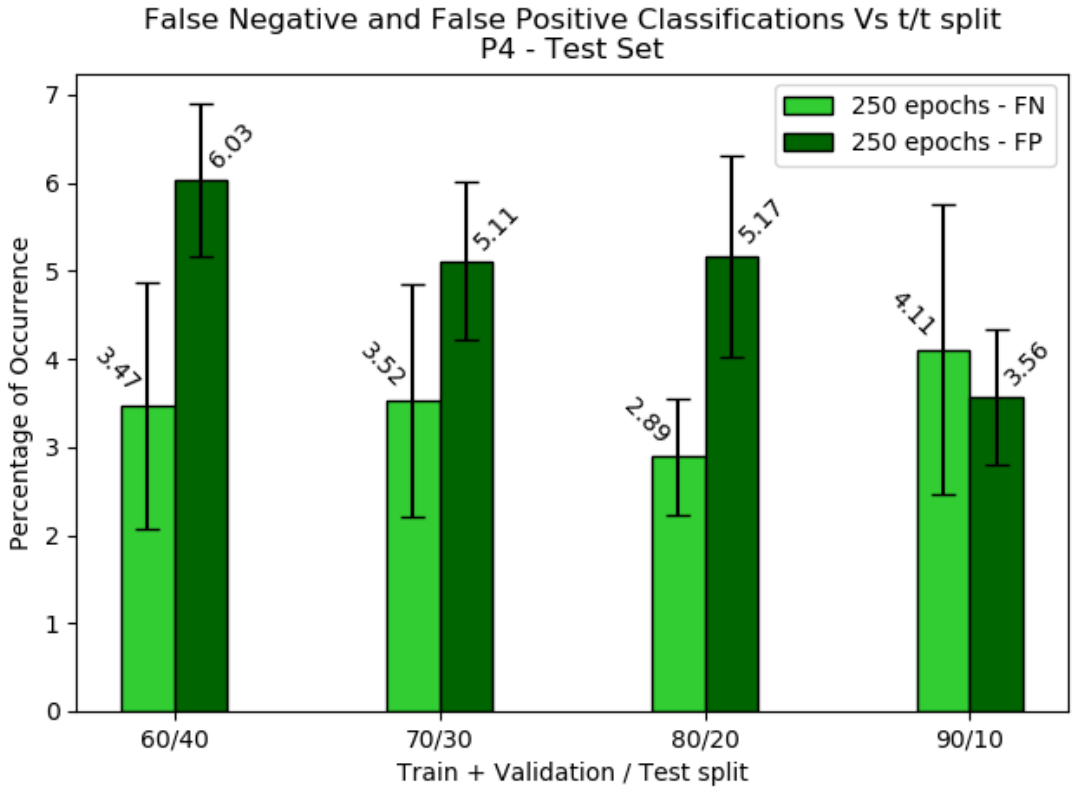


Figure 64. False negatives (light green) and false positives (dark green) at varying t/t split.

Unfortunately, also in this case no particular trend is noticeable. It is only possible to observe the general increase in performance that in this particular study reflects in a decrease of the overall incorrect classification.

5.1.3. False negatives distribution

A *false negative* is an example that is *feasible*, so able to complete the cycle, but incorrectly marked as *unfeasible* by the *cDNN*. This misclassification exposes the algorithm to a problem of "*lost data*". By this expression, the author means that the algorithm loses some information by assuming some *layout* will not complete the *driving cycle*, so in turns not performing regression on them.

To try and understand if there is a pattern in the *false negatives'* misclassification issue, they have been monitored over the 24 simulations abovementioned. All the *false negative* examples are here highlighted in red over the blue distribution of the complete P4 dataset.

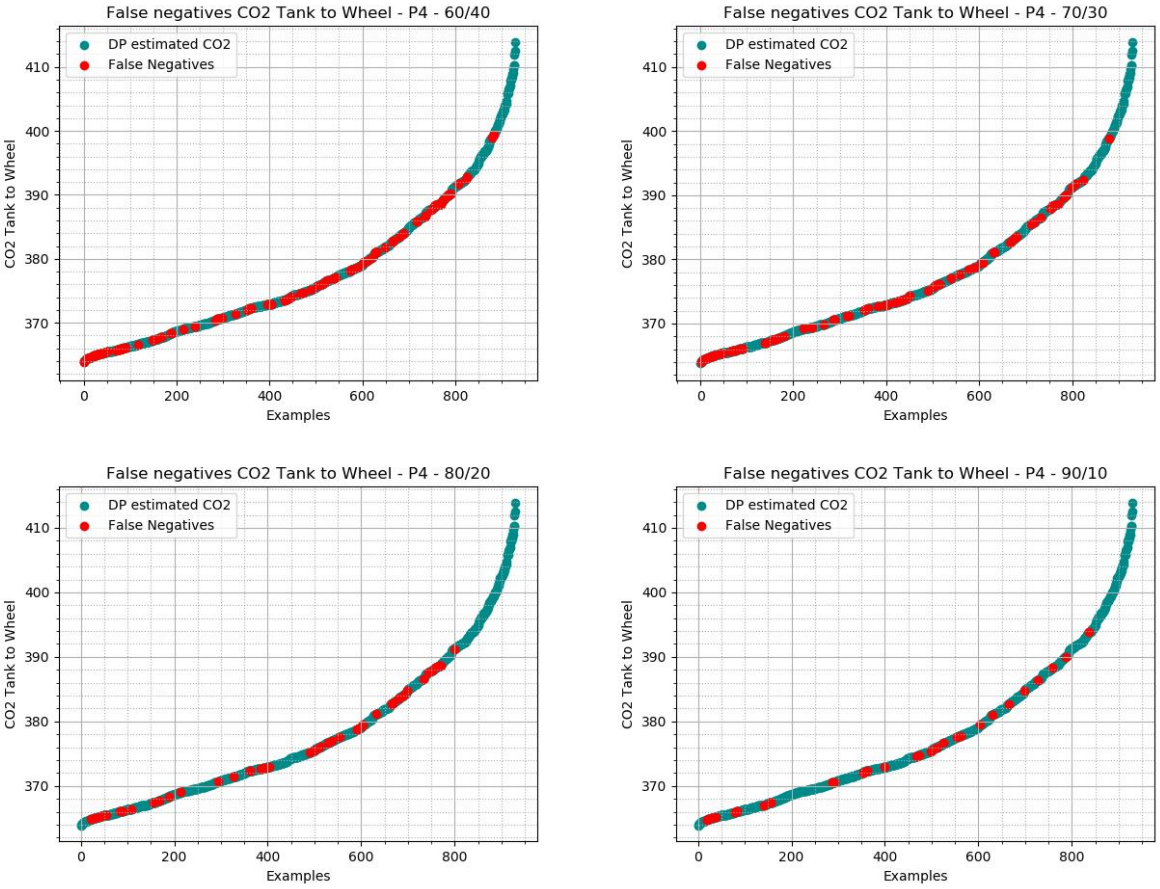


Figure 65. False negatives distribution for varying t/t split.

Obviously, the larger the *test set* the more *false negatives* will be found. Asides from the decrease in performance due to the decreasing *training set* size, this is a pure visual effect due to the fact that the net is tested on more *examples*. Anyway, the goal is to observe if any trend occurs. Unfortunately, no particular trend is evident. The only possible observation is that only the higher part of the distribution seems to not be affected by the possibility to be misclassified as *unfeasible*. The "head" of the distribution instead, seems to be affected, and this could mean that possible good layouts will be excluded.

5.1.4. Architecture recognition

In this section is presented a side study of the present project. It is addressed to as "side study" because the main goal was not to predict CO_2 value or make *feasibility* classification. The main goal of this study was rather to predict the nature of the architecture a layout, given its design parameters.

To achieve this objective, the net was modified to make it able to perform multiclass classification. The dataset is also modified: the three datasets are combined into one larger dataset where the column related to the CO_2 emission is no more present; instead, a *label feature* including the type of architecture is included (P2, P3 or P4).

Referring to Section 3.2, it is evident that the three datasets have not comparable features. As already mentioned, P4 has a different feature for the speed ratio regarding the electrical motor coupling. This issue is dealt with by creating both features in the dataset: if an example does not include that feature, it is set to zero.

This kind of approach to the problem however introduces a very strong dependency of the *label* to a single feature. By this the author means that for the net it will be extremely simple to spot P4 layouts, since they are the only ones characterized by a non-zero "FDsSpRatio" feature. However, the net will probably not be able to distinguish properly between P2 and P3 architectures since their datasets are very much comparable. The results clearly confirm this assumption.

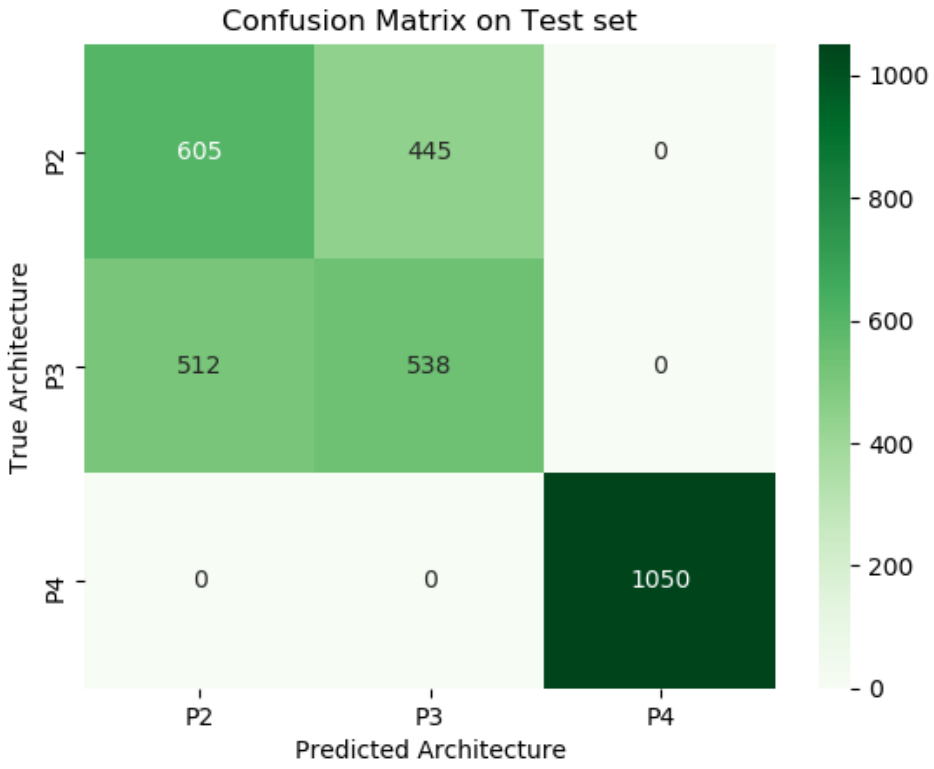


Figure 66. Multiclass classification net. Architecture recognition.

Perfect predictions are achieved for P4 layouts; instead a completely random classification is evident for the other two architectures.

An attempt to "force" the net to recognize the architectures is performed by introducing another feature to the dataset indicating the position of the electrical motor. In this way we are de-facto telling the net which architecture is the correct one. The goal was only to verify that the net was able to correctly pick-up the information for future *multi-architecture regression tasks*.

Figure 67. Multiclass classification net. Architecture recognition. Manifest EM position.

As it was expected, the net never fails to classify all the tests examples.

One last step of this study is to try a *feasibility* classification on a hybrid dataset P2/P3. Since the two datasets are very much comparable, it is plausible to think that the net is able to perform reasonably well on this task. Indeed, the net was able to perform the *feasibility* classification with excellent results. It achieved 96.2% *MCC*.

In conclusion, it can be stated that, in a future *multiclass feasibility classification task*, where a dataset is created as explained in this section, the net will surely be able to perform well even without the auxiliary feature: it can pick-up the information regarding the P4 architecture from the "FDsSpRatio" feature and then perform classification normally.

5.2. Pipeline simulations

After validating the capabilities of the *cDNN* it is possible to examine the behaviour of the whole pipeline on a complete "*classification + regression*" task. It is reminded that the size of the dataset used for this application is reduced from the one used in the pre-existing project. This in theory can cause a decrease in performance due to a decrease in *training set size*.

It should be stated that the pre-existing code is believed to be affected by an issue related to the normalization method. The previous code applies normalization, namely *min-max scaling*, to the *test set* independently from the *training set*. It is indeed correct to not consider the *test set* when selecting minimum and maximum value of the distribution not to incur in *data leakages*. However, the *minimum* and *maximum values* used to normalize the *test set* should be the same as the one used for *training set*. Otherwise the distribution of the dataset's examples could be inconsistent.

The abovementioned issue was causing the net to perform strangely and to produce not coherent test performance. After addressing the issue, the behaviour normalized, and the simulations could begin.

The test procedure included 6 simulations, performed on the P4 dataset, and carried out across the whole *pipeline*.

The *hyperspaces* for the two nets are the following.

Table 11

cDNN hyperspace	
Learning rate	0.0002 – 0.02
Hidden Layers	1 – 4
Neurons first hidden layer	130 – 230
L2 regulrizer	0.003 – 0.3
Batch size	16 - 128

Table 12

rDNN hyperspace	
Learning rate	0.005 – 0.5
Hidden Layers	1 – 6
Neurons first hidden layer	30 – 80
Batch size	8 – 64
Weights initialization	*Xavier, Random, truncated normal*
Dropout	0 – 0.7

It is also highlighted that the *t/t split* is 90/10. So, 90% of the dataset's examples will be used for training and validation while the remaining 10% is used for testing. Moreover, since the *rDNN* datasets derive directly from the *cDNN* ones, they will be approximately $^{2}/_{3}$ of the corresponding *cDNN* ones as they should contain only *feasible* examples.

The final results are now presented.

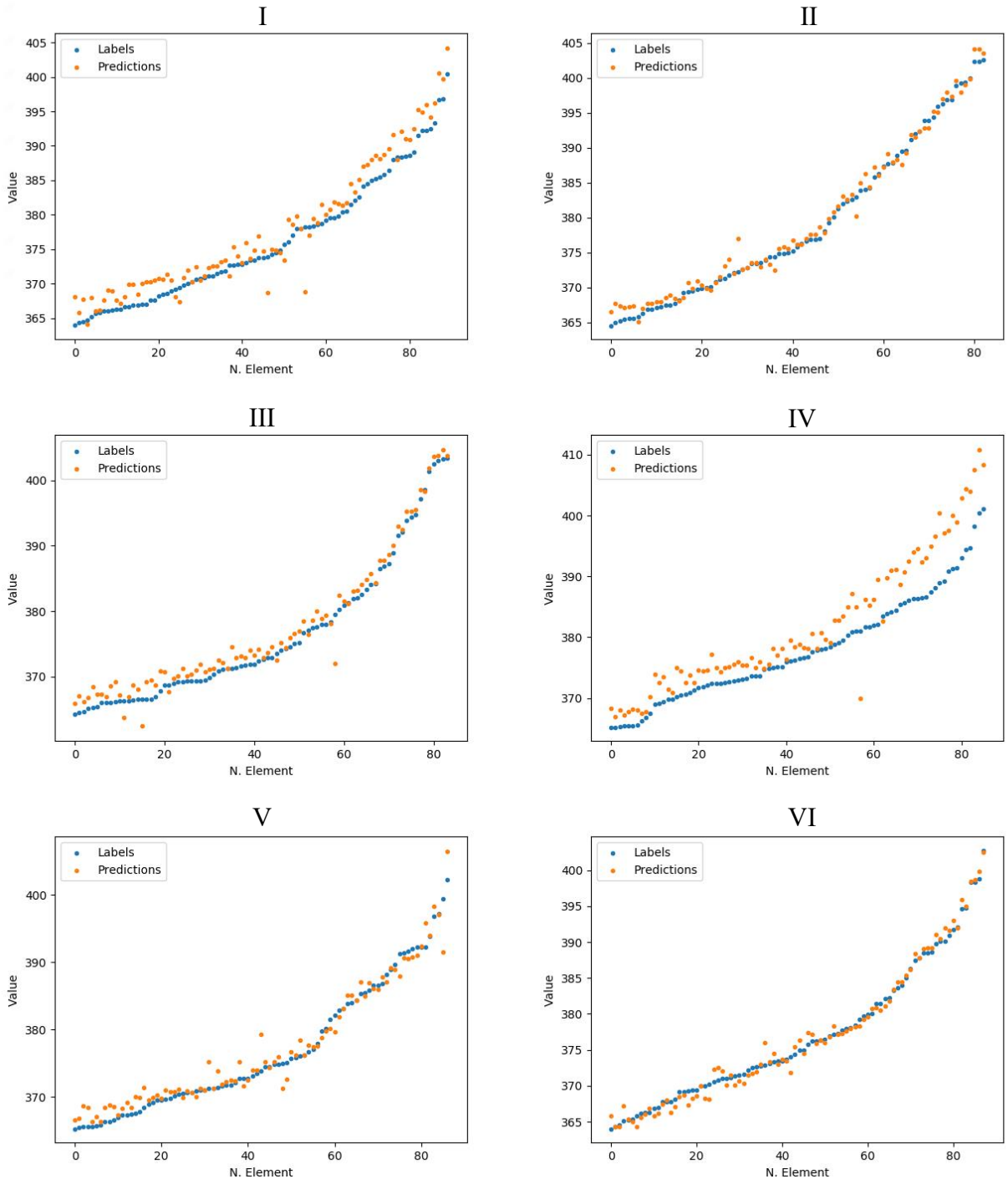


Figure 68. Pipeline simulations. True labels and predictions.

The numbers are resumed in the following table.

Table 13

Pipeline performance - Regression step

	Error							
	Mean		***Max***		***Min***		***R_value***	***Loss***
	Absolute	***Relative***	***Absolute***	***Relative***	***Absolute***	***Relative***		
I	2,11	0,01	9,39	0,02	0,03	0	92,17%	2,5
II	0,92	0	4,67	0,01	0,01	0	98,83%	1,25
III	1,49	0	7,43	0,02	0,03	0	97,18%	1,77
IV	4,12	0,01	11,48	0,03	0,15	0	67,20%	4,61
V	1,28	0	7,88	0,02	0	0	96,29%	1,93
VI	0,78	0	3,13	0,01	0,01	0	98,86%	1,06
mean	1,78	0,00	7,33	0,02	0,04	0,00	91,76%	2,19
std	1,13	0,00	2,78	0,01	0,05	0,00	11,21%	1,18

Highlighted are the best (green) and worst (red) simulations.

The performance is more than acceptable with the worst simulation showing a mean relative error of only 1%, which translates in only 4.12g of absolute error. It is worth mentioning that *R score* punish greatly the performance of the 4th simulation because it considers the difference between the *true labels* and the *predictions* rather than their absolute values. However, this can be avoided simply increasing the threshold of acceptance during the validation phase. This value was indeed set to a relatively low one (50%) to be able to analyse suboptimal results as the one highlighted in red.

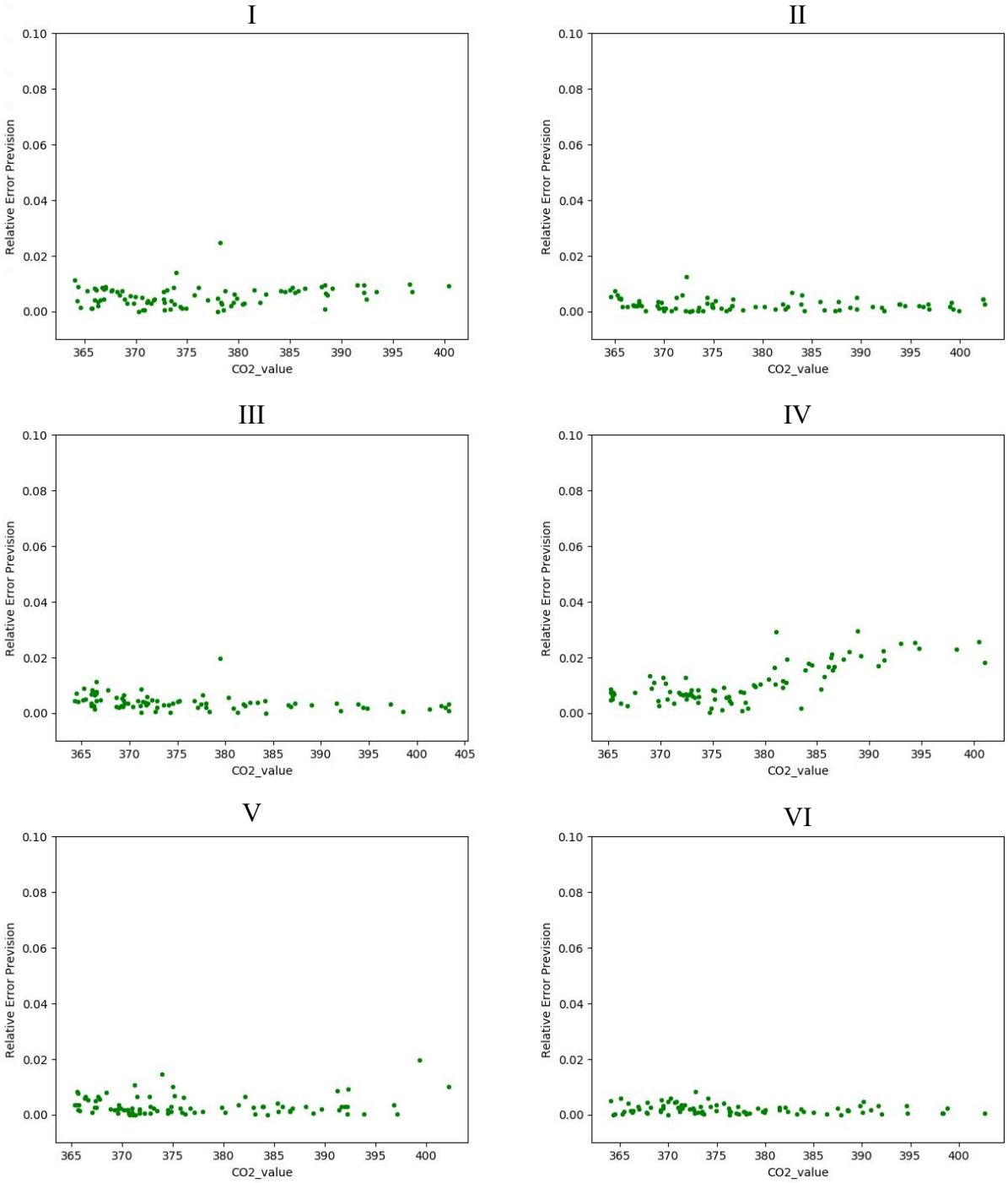


Figure 69. Pipeline simulations. Relative error.

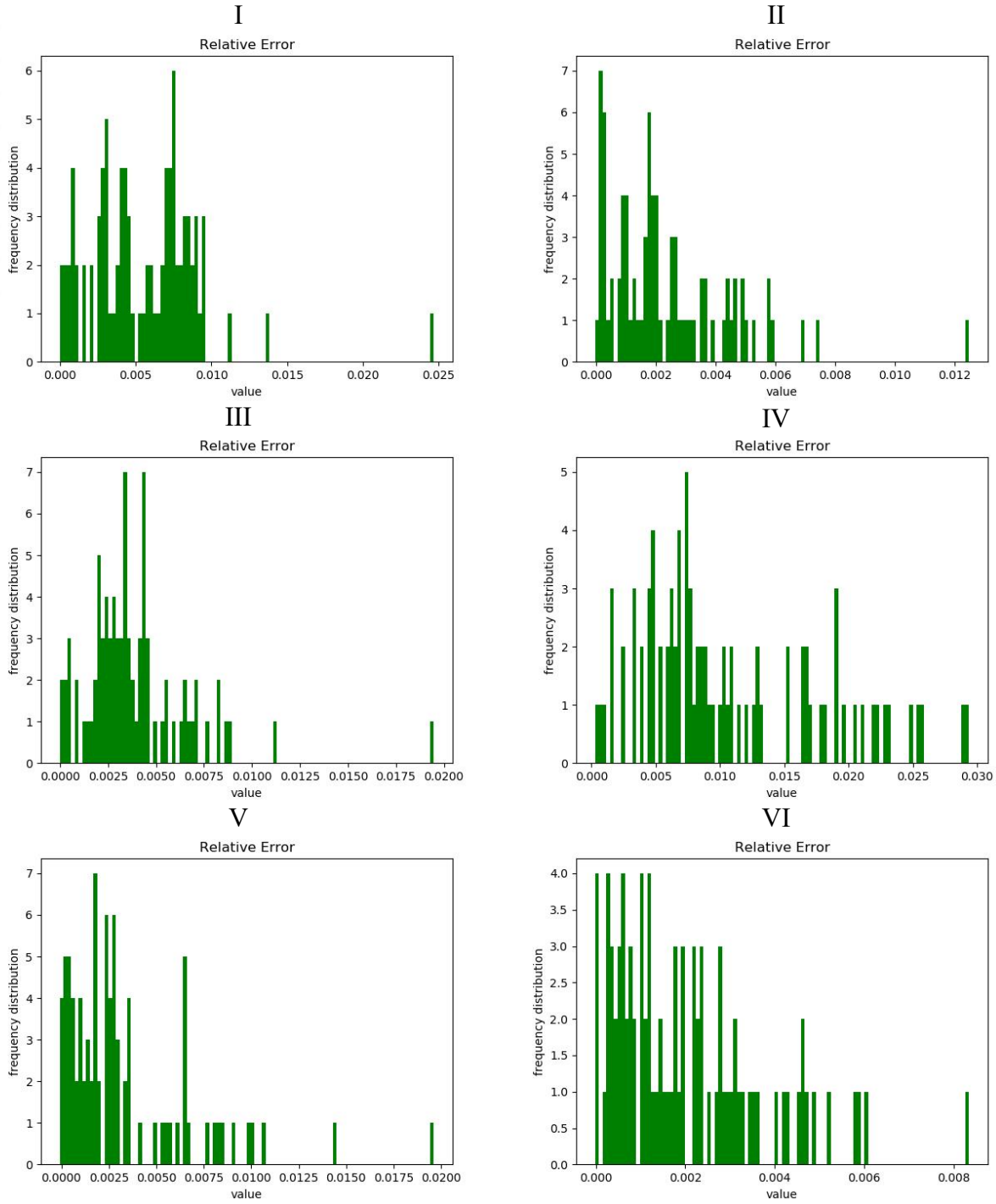


Figure 70. Pipeline simulations. Relative error frequency distribution.

The relative errors and their frequency distribution are perfectly consistent with the results obtained in the pre-existing project. They are indeed very convincing, and the approach here presented is believed to be a success.

5.3. False positives analysis

A *False Positive* is an example that is *unfeasible* but is incorrectly marked as *feasible* by the *cDNN*, and a CO_2 value will be predicted by the *rDNN*. The main issue with this type of misclassification is that a *layout* that will not be able to complete the driving cycle could potentially be marked as the best one, and lead to completely fallacious designs.

One last step is to consider the *false positive examples* separately and perform regression on them to monitor their *predictions* in terms of CO_2.

Since not all the false positives are equally dangerous, rather they become riskier as they approach the "head" of the CO_2 distribution, it has been chosen to observe only the ones that reach top 10% of the distribution. The selection procedure is as follows:

1. We note the *maximum* and *minimum* predicted value of CO_2 for the *true positives.*
2. We compute the difference between the two and divide the difference by 10.
3. We add the just computed value to the *minimum*.
4. We observe only the *false positives* for which the *prediction* is under this threshold.

The results of this method are resumed in the following table.

Table 14

False positives analysis

	I	***II***	***III***	***IV***	***V***	***VI***
min	364,07	365,14	362,41	366,93	366,33	364,30
top 10%	368,07	369,04	366,63	371,31	370,33	368,11
False Positives	363,15	-	-	367,26	368,08	-
	368,06	-	-	367,37	368,20	-
	367,19	-	-	-	368,36	-
	-	-	-	-	363,03	-
	-	-	-	-	369,30	-
	-	-	-	-	367,97	-

In the table are highlighted the *threshold value* and the *minimum value* for all the simulations. Each value reported in the table is under the *threshold* and the two in red are even under the *minimum*.

It is evident that the possibility to mark an *unfeasible layout* as the best one is present and should be addressed.

Conclusions

To resume, the present project used a total of 3 datasets related to three different *Hybrid Electric Vehicles Architecture*, namely P2, P3 and P4. These datasets are the result of multiple simulations of a *Dynamic Programming Algorithm* run on the WHVC driving cycle. They are used to train a *Pipeline* of two *Deep Neural Networks*; the first one performs *Classification* and has the goal to produce *Feasibility Predictions*; the second one performs *Regression* with the aim to predict *Tank-to-Wheel* CO_2 emissions of a series of layouts included in the datasets.

The results of the simulations are more than convincing both for *feasibility prediction* and *regression*. There is no doubt that the net correctly picks up the trend hidden in the *Design Parameters* and effectively interprets them to produce valuable information. Obviously, extensive simulations should be carried out for future industrial application, especially for what concern the issue of the *false negatives* and *false positives*.

Could be of interest to test different types of *Deep Neural Networks* architectures to try and increase the performances even further. Another interesting development could be to implement *Bayesian Optimization* during the *hyperparameters' selection* process. The *Architecture Recognition* task should be deepened and implemented effectively. Lastly, the dataset should be extended to include cycle related features, and various examples coming from different driving cycle should be combined to form a unique dataset on which train the *pipeline*.

In conclusion, *Artificial Intelligence*, and in particular *Deep Learning*, has been a reliable tool to approach the tasks related to this project.

There is no doubt that this technology will transform the way we think and interact with the world.

Bibliography

[1] Josh Miller, Li Du, Drew Kodjak (2018), *Impacts of world-class vehicle efficiency and emissions regulations in select G20 countries.*

[2] Fuad Un-Noor, Sanjeevikumar Padmanaban, Lucian Mihet-Popa Mohammad Nurunnabi Mollah and Eklas Hossain (2017), *A Comprehensive Study of Key Electric Vehicle (EV) Components, Technologies, Challenges, Impacts, and Future Direction of Development.*

[3] Venditti, Mattia (2015), *Innovative Models and Algorithms for the Optimization of Layout and Control Strategy of Complex Diesel HEVs, Politecnico di Torino.*

[4] Cirrincione, Giansalvo, *Deep Learning, EXIN*.

[5] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks.*

[6] Juergen Schmidhuber (2014), *Deep Learning in Neural Networks: An Overview.*

[7] Bin Ding, Huimin Qian, Jun Zhou (2018), *Activation functions and their characteristics in deep neural networks*

[8] Ruder, Sebastian (2016), *An overview of gradient descent optimization algorithms.*

[9] Boldrini, Nicoletta (2019), *Reti neurali: cosa sono e a cosa servono.*

[10] Brian J. Taylor, Marjorie Darrah, Christina D. Moats (2003), *Verification and validation of neural networks: A sampling of research in progress*.

[11] Feiping Nie, Hu Zhanxuan, Xuelong Li (2018), *An investigation for loss functions widely used in machine learning.*

[12] Yulong Lu, Jianfeng Lu, *A Universal Approximation Theorem of Deep Neural Networks for Expressing Distributions.*

[13] Jimmy Lei Ba, Diederik P. Kingma (2015), *Adam: a method for stochastic optimization.*

[14] *Page 156 of Machine Learning, McGraw Hill.*

[15] Jasper Snoek, Hugo Larochelle, Ryan P. Adams, *Practical Bayesian Optimization of Machine Learning Algorithms.*

[16] Baodi Zhang, Fuyuan Yang, Lan Teng, Minggao Ouyang, Kunfang Guo, Weifeng Li, Jiuyu Du (2019), *Comparative Analysis of Technical Route and Market Development for Light-Duty PHEV in China and the US.*

[17] Davide Chicco, Giuseppe Jurman (2020), *The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation.*

[18] Vinod Nair, Geoffrey E. Hinton (2019), *Rectified Linear Units Improve Restricted Boltzmann Machines.*

[19] Sergey Ioffe, Christian Szegedy (2015), *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.*

[20] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov (2014), *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.*

[21] Finesso Roberto (2019), *Optimization of the control strategy of HEVs, Politecnico di Torino.*

[22] Roberto Finesso, Ezio Spessa, Mattia Venditti (2016), *Robust equivalent consumption-based controllers for a dual-mode diesel parallel HEV.*