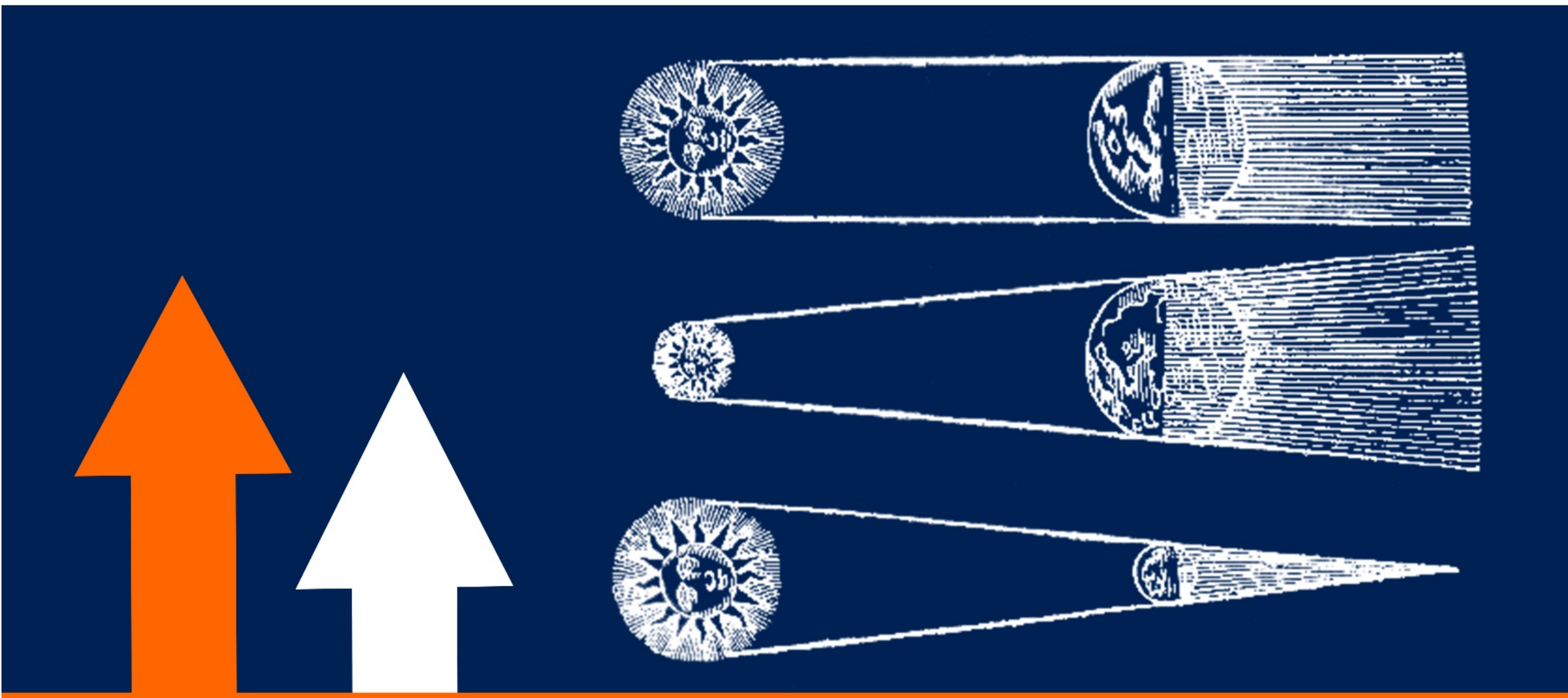
Itinerari per l'alta formazione

DEEP LEARNING FOR BEGINNERS

Greta Falavigna



CNR - Consiglio Nazionale delle Ricerche

IRCrES - Istituto di Ricerca sulla Crescita Economica Sostenibile

Direttore	Emanuela Reale
Direzione	CNR-IRCRES *Istituto di Ricerca sulla Crescita Economica Sostenibile* Via Real Collegio 30, 10024 Moncalieri (Torino), Italy Tel. +39 011 6824911 / Fax +39 011 6824966 segreteria@ircres.cnr.it www.ircres.cnr.it
Sede di Roma	Via dei Taurini 19, 00185 Roma, Italy Tel. +39 06 49937809 / Fax +39 06 49937808
Sede di Milano	Via Corti, 12, 20121 Milano, Italy Tel. +39 02 23699501 / Fax +39 02 23699530
Sede di Genova	Corso Ferdinando Maria Perrone 24, 16152 Genova, Italy Tel. +39 010 6598798

Redazione	Emanuela Reale Giancarlo Birello Antonella Emina Serena Fabrizio Anna Perin Isabella Maria Zoppi
✉	redazione@ircres.cnr.it
🌐	www.ircres.cnr.it/index.php/it/produzione-scientifica/pubblicazioni

Itinerari per l'alta formazione 4, gennaio 2022
ISBN: 978-88-98193-23-3
gennaio 2022 by CNR-IRCrES

Deep Learning for Beginners

GRETA FALAVIGNA

CNR-IRCrES – National Research Council, Research Institute on Sustainable Economic Growth, Via Real Collegio 30, 10024 Moncalieri TO, Italy

corresponding author: greta.falavigna@ircres.cnr.it

ABSTRACT
The aim of the present handbook is to drive the reader toward a deepen knowledge of *Deep Learning* (DL) methodologies and their functioning.
Even if DL has ancient ancestry, only in recent years they have been strongly revalued and consequently applied. However, it is noteworthy that in this field, the increasing computational power played a fundamental role: *Deep Learning* models are techniques of success, but, at the same time, they are also expensive from a computational point of view and not always clearly understandable.
The handbook starts considering the "big family" of *Artificial Intelligence* (AI), and continues clarifying what are *Machine Learning* (ML) and *Deep Learning* (DL). The reader will discover that DL is based on *Artificial Neural Networks* (ANN), then the fundamentals of ANNs will be presented and discussed, until getting to understand the working and the differences among different architectures.
Finally, at the end of the reading, the reader will have in hands all knowledge required for recognizing different *Deep Learning* architectures, their functioning, and the fields of their applications.

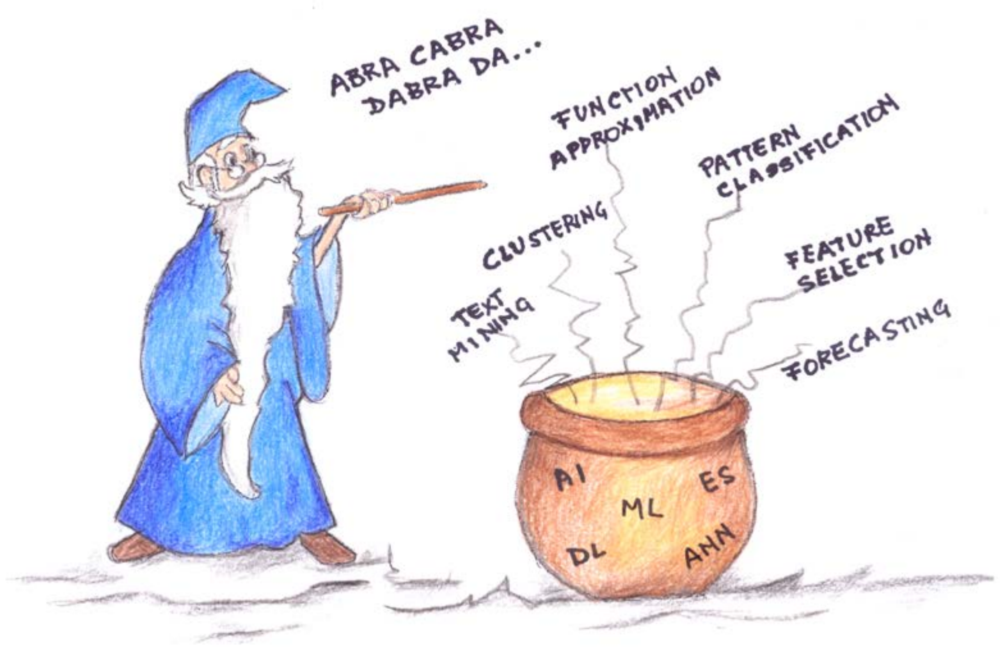
KEYWORDS: Artificial Intelligence; Machine Learning; Deep learning; Expert Systems; Artificial Neural Networks.

DOI: 10.23760/978-88-98193-2022-04

ISBN: 978-88-98193-25-7

HOW TO CITE THIS ARTICLE
Falavigna, G. (2022). *Deep Learning for Beginners*. Moncalieri: CNR-IRCrES (Itinerari per l'alta formazione 4). Disponibile da http://dx.doi.org/10.23760/978-88-98193-2022-04

TABLE OF CONTENTS



Acknowledgements: I thank my colleague Antonella Emina for her evaluable effort in reading and editing the present handbook. I also thank my mother for the drawing of Merlin the Wizard and his experiments.

1 Introduction

The present handbook represents an introduction to *Deep Learning* (DL) and *Artificial Neural Networks* (ANNs), suitable for beginners and for whom who need to understand better how these models work.

The main goal is to analyse the fundamentals of DL models, that are *Artificial Neural Networks*, and their functioning, debunking the reputation of “black-boxes”. *Deep Learning* techniques are complex, mathematically expensive, but the increase of the computational power of informatics allows to easily obtain results and to understand how they are achieved.

Starting from the definition of *Artificial Intelligence* and *Machine Learning* techniques, the handbook focuses on *Deep Learning* and on its fundamental units: the *Artificial Neural Networks*.

After a brief introduction on the history and a more technical section on the mathematics of ANNs algorithms, the handbook presents how to model a *Deep Learning* model and related problems.

Through simple numerical examples, the handbook will help the reader in understating not only the working, but also the big power of DL framework. With the aim to recognize which are the models proposed by scientists, we discuss the basic architectures of ANN, aware of not being exhaustive.

The conclusion section of the handbook argues briefly how to choose the parameters of ANNs, together with strengths and weaknesses.

Finally, two Appendixes are provided: the first one presents the more used distance measures, while the last one is a glossary of main significant terms used in the field of *Deep Learning*.

At the end of the reading, the reader will be able to understand what is a *Deep Learning* model and how it works, what is the difference between DL and other methodologies, and when applying a specific ANN topology.

2 AI, ML, DL, ANNs… Who are they? Definitions…

One of the main difficulties that people encounter studying about *Artificial Intelligence* is a certain confusion at the definition level. What is the difference among *Artificial Intelligence* (AI), *Machine Learning* (ML) and *Deep Learning* (DL)? In addition, *Artificial Neural Networks* (ANNs) are *Deep Learning*?

In general, everyone has an idea of what these notions are, but whether or not they are the same thing is not always clear.

The *Artificial Intelligence* is the “mother class” of different objects and its aim is to teach computers to learning how human behave. The final goal is building models able to simulate the cognitive capabilities of human brains.

Machine Learning, as suggested by Arthur Lee Samuel (1959), is a "field of study that gives computers the ability to learn without being explicitly programmed". The main goal is to build AI programs able to automatically define other programs for interpreting data and predicting results.

Figure 1 presents some examples of ML considering the three learning approaches: when there is a teacher controlling and correcting results (i.e., supervised learning), and when the teacher is absent (i.e., unsupervised learning) [1]. Finally, the reinforcement learning deals with sequential decision problems, in which the action to be taken depends on the current state of the system and determines its future one. We will approach with the first two algorithms later in the text (section 4.2).

So, what is the *Deep Learning*? This is a "class" of models improving the learning process through *Artificial Neural Networks* that, thanks to powerful algorithms, are able to simulate the functioning and the structures of human brain.

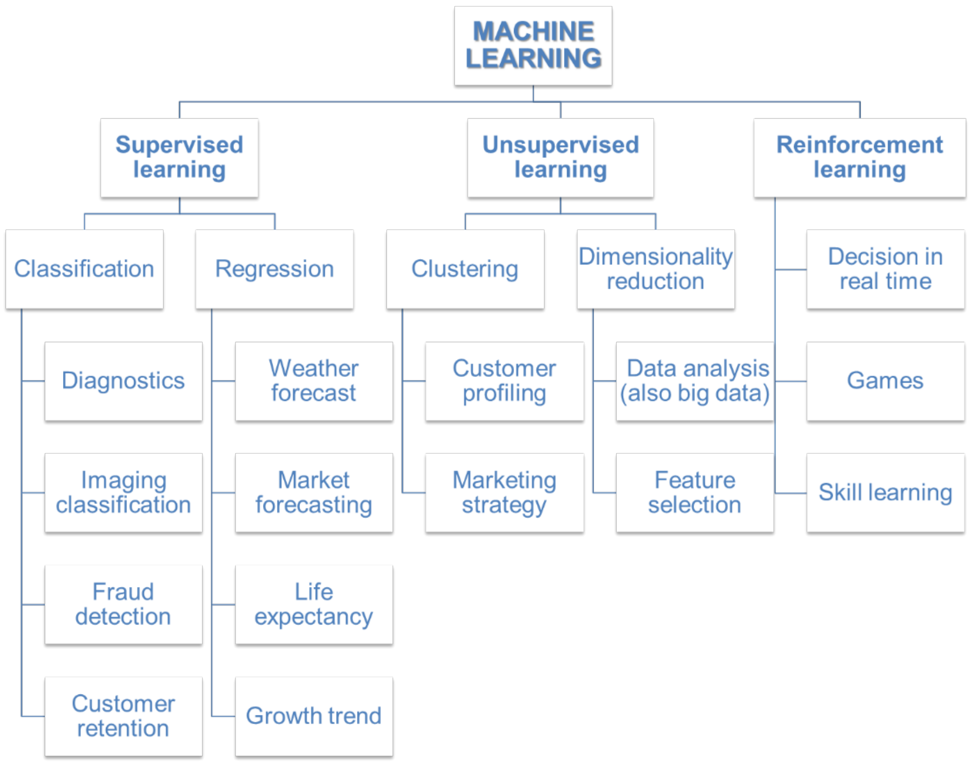


Figure 1. Examples of ML distinguished by learning algorithm.

[1] For an in-depth review of ML techniques see Ray (2019). For an interesting discussion on learners' performance see Cerulli (2021).

In sum, *Artificial Intelligence* learns how human brain behaves, *Machine Learning* exploits the capabilities of AI models for interpreting data, defining relations and applying them to similar patterns, and finally, *Deep Learning* and *Artificial Neural Networks*, thanks to powerful learning algorithms, allow to analyse big set of data without the human intervention, because they are able to modify themselves for achieving optimal results.

In *Machine Learning* (and *Deep Learning*), algorithms are defined starting from reality. A learning algorithm is called **Learner** and it allows to create other algorithms.
A **Learning Algorithm** represents a set of instructions of *Machine Learning* allowing a computer program to imitate the brain functioning.

Figure 2 represents the substantial difference between ML and DL: the first one needs of human activities in features detection, DL are completely autonomous because they adapt themselves learning from the reality. Indeed, *Deep Leaning* techniques are able to identify which features are really significant for the problem and which are less important.

A typical example of ML is when we are on a social network and some pages are automatically proposed. A software is able to understand which are our interests, looking at our previous searches, and automatically it shows us similar pages. This is an algorithm of *Machine Learning* that learns our tastes. When the learning process is more complex, as for example in the case of Autonomous Vehicles, cars are able to recognize the presence of a pedestrian, assess a danger and automatically activate the braking mechanism. In addition, they can recognize road signs and help the driver while driving. In these cases, DL algorithms are implemented.

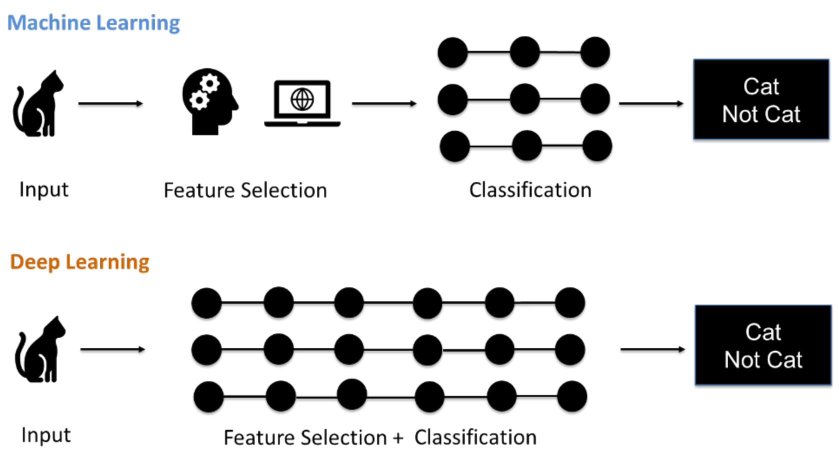


Figure 2. *Machine Learning* and *Deep Learning* processes by comparison.

However, as presented in table 1, the greater power of *Deep Learning* is counterbalanced by the need of large data sets and a greater computational power.

Before concluding this section, another term used in the field of AI is *Expert Systems* (ES). An ES is a computer program reproducing the human reasoning, the logic, the belief, the opinion, and the experience (Plant & Stone, 1991; Fu, 1995). Following the definition of the OECD[2], the ES is represented by two parts: the knowledge base and the inference engine and, in general, they are driven by IF-THEN rules. Main limitations of ES are the hypersensitivity to missing or noisy

[2] The OECD definition of Expert System is available at this site: https://stats.oecd.org/glossary/detail.asp?ID=3384 (last visit: January 09 2022).

data and the working in sequential manner. However, the choice between the adoption of an ES or an ANN depends from data availability and from theoretical knowledge of problem. Indeed, ANNs are favourite when the theory is unclear, but the availability of data is high; whereas, if there is not a big amount of data, ES are always preferred.

Machine Learning	**Deep Learning**
Pros	
Small data sets	Learns features and classifies automatically
Non-computationally intensive	High accuracy
Cons	
Feature selection with human intervention	Very large data sets
Lower accuracy then DL	Computationally intensive

Table 1. Pros and Cons of *Machine Learning* and *Deep Learning.*

2.1 A championship of five teams for *Machine Learning*!

Before explaining in details the meaning of *Deep Learning* and *Artificial Neural Networks*, it is worthy of note that in *Machine Learning* we can find different schools of thoughts.

If you think that *Machine Learning* and its techniques are topics only for mathematicians or computer scientists or engineers, you are wrong. Indeed, it is not only a computational question: every method of resolution is driven by a reasoning on how a scientist approaches a problem. As suggested by Domingos (2015), we can identify 5 main currents of thoughts, as if they were 5 different teams: Evolutionists, Bayesians, Analogists, Symbolists, and finally Connectionists. Each of these teams is distinguished from the others in three fundamental steps: Representation, Evaluation, and Optimization. The Representation is the formal language used for expresses his models; the Evaluation is the estimate of the goodness of the model; the Optimization is the algorithm identifying the model with highest score.

Table 2 shows main differences among the five approaches (Domingos, 2015). For the Symbolists, who are inspired by philosophy, psychology and logic, learning is the inverse of deduction. Connectionists are inspired by neuroscience and physics to perform a reverse engineering operation of the brain. Evolutionists are inspired by genetics and evolutionary biology to carry out numerical simulations of evolution. Bayesians base their theories on statistics, for them learning is a form of probabilistic inference. Finally, the Analogists are influenced by psychology and mathematical optimization, and believe that we learn from extrapolations based on similarity criteria.

Often, the players of one team teach to players of other teams with the aim to improve the competition and results.

Now, conscious of "doing *Machine Learning*" is not only "doing mathematics", we can start studying the definition of *Deep Learning*.

Teams	**Representation**	**Evaluation**	**Optimization**
Evolutionists	Classification systems	Graphical modeling	Genetic research
Bayesians	Bayesian or Markov networks	Graphical modeling	Statistical (Bayesian) inference
Analogists	Support vectors	Margins	Constrained optimization
Symbolists	Logic (i.e. decision trees)	Accuracy	Inverse deduction
Connectionists	Artificial Neural Network	Squared error	Gradient descent

Table 2. Five teams and their characteristics for problem solving.

2.2 The “sound of *Deep Learning*”: a univocal scientific definition

Even if in previous section we have understood what *Deep Learning* is about, we need to find a univocal scientific definition. Literature proposes several explanations and we report some of the most significant:

1. From the web: “Deep Learning is about learning the knowledge chunk in a form of multiple levels of representation and abstraction to make up higher level information from lower level information (e.g., sound, image, etc.)”.
2. Bengio (2009): “Deep learning is about learning feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features”.
3. LeCun et al. (2015): “Deep learning (also known as deep structured learning or hierarchical learning) is learning data representations, as opposed to task-specific algorithms. Learning can be supervised, semi-supervised or unsupervised”.
4. Deng & Yu (2014): “Deep learning is a class of machine learning algorithms that: (1) use a cascade of multiple layers of nonlinear processing units for feature extraction and transformation. Each successive layer uses the output from the previous layer as input, (2) learn multiple levels of representations that correspond to different levels of abstraction; the levels form a hierarchy of concepts”.
5. Zhang et al. (2018): “Deep learning is a process not only to learn the relation among two or more variables but also the knowledge that governs the relation as well as the knowledge that makes sense of the relation”.

We could propose many other definitions, but it is clear that DL is to gain knowledge for a hierarchy of features, where features change on the base of problem analysed. Definition [5] gives an additional power to DL models: they are not only able to select significant features, but they recognize relations among variables and, more important, they extract the knowledge from these relations.

The fundamentals of *Deep Learning* are, as suggested in definition [4], the *Artificial Neural Networks* (ANNs), that, more than all other models, are able to reproduce the same mechanisms of human reasoning.

Looking at previous definitions, we can reassume that *Deep learning* is a particular case of feature learning characterized by *Artificial Neural Networks* with two or more layers (often called multi-layers), capable of processing information in a non-linear way.

Now, the definition of *Deep Learning* is clearer, and then we can pass to deepen the *Artificial Neural Networks.*

Deep Learning (DL): is a particular case of feature learning characterized by *Artificial Neural Networks* with two or more layers (often called multilayers or hidden layers) capable of processing information in a non-linear way.
When an ANN is composed by only one or two hidden layers can be called **Shallow Neural Network**.

3 Artificial Neural Networks (ANNs): A brief introduction

The *Artificial Neural Networks* are a representation of human neural brain in mathematical terms and their main goal is to simulate its functioning process. They are composed by “artificial neurons” (also called “nodes”) that are the crude approximations of biological neurons found in human brains.

ANNs are studied because the formalization of a single neuron is simple and simple architectures can be very powerful.

A variety of problems are solved through ANNs because they outperform other computational methodologies. In details, ANNs are particularly effective in (Basheer & Hajmeer, 2000):

1. *Pattern classification*: that deals with the assignment of an unknown input pattern to one of preassigned classes on the base of characteristics of class. This is the major ability of ANN and it is the so-called Classification power. In this case, the ANN topology is supervised. In finance, ANNs have been widely applied in the assignments of rating judgements (Falavigna, 2008a; Falavigna 2008b; Falavigna, 2012). A comprehensive review of applications can be found in the paper of Abiodun et al. (2019).
2. *Clustering*: data are grouped on the base of similarity and dissimilarity among input patterns. This specific process, is often used as pre-processing phase with the aim to reduce the noise in the input data. The ANN architecture is, in this case, unsupervised. For a comprehensive survey of applications see Liao & Wen (2007).
3. *Function approximation*: input-output data are trained with the aim to approximate the hidden rules linking input to output. In general, Multi-layer ANNs are universal approximators (Hecht-Nielsen, 1990; Hornik et al., 1989). A short but effective review of ANNs topologies for function approximation can be found in Zainuddin & Pauline (2008).
4. *Forecasting*: the ANN is applied to time-series in order to predict subsequent behaviour considering a given scenario. Starting from a well-known dataset, the neural network is able to produce outputs expected from a given input. A special case of this is time series prediction with dynamic/recurrent networks (i.e., Long- and Short- Term Memory networks, LSTM). A

systematic review of application in prediction and forecasting is proposed by Neu et al. (2021).

5. *Optimization*: ANN is also able to maximize or minimize an objective function, given a set of constraints. Hopfield networks are commonly used with this aim. ANNs are often coupled with Genetic Algorithm with the aim to improve their performance. See for an interesting application in environmental field the work of Li et al. (2021).
6. *Association*: the ANN works as a pattern associator trained on perfect input data. In a second phase, the ANN is applied for classifying corrupted data. In this case, Hopfield networks are applied in the field of image recognition. A review on the application of Hopfield NN and Boltzman machine in imaging recognition is proposed by Marullo & Agliari (2021).
7. *Control*: recurrent ANNs are often used for aiding an adaptive control system. A review of applications in chemical control process can be found in Hussain (1999).
8. *Text mining*: deep learning models are used for the automatic processing of natural textual language available in large quantities on electronic files. DL allows to extract and structure the contents and themes for a quick analysis (of a non-literary type), the discovery of hidden information and/or to automate some decisions. Interesting commentaries on text mining in medical field are provided in Falavigna (2020) and Falavigna (2021).

A really interesting application of ANN is the brain modelling where the goal is to understand how real brain works. This can improve the knowledge about human intelligence and help in defining strategies or remedial actions for patients with damaged brains. Not only, the main idea is to exploit advantages of ANNs for building artificial systems with the aim to build effective devices, to improve machine power, to automatize processes, and to analyse relations among data.

Finally, we can reassume the main applications until today of ANNs in the following (non-exhaustive) two research streams[3]:

- Brain modelling:
  - Models of human development for helping children with developmental problems.
  - Simulations of adult performance with the aim at understanding of how the brain works.
  - Neuropsychological models for suggesting remedial actions for brain damaged patients.
  - Artificial Life simulations for clarifying brain evolution, structure, growth, etc.
- Real world applications:
  - Pattern recognition (for instance, speech recognition).
  - Data analysis (for instance, data mining).
  - Noise reduction (for instance, ECG noise reduction).
  - Control systems (for instance, autonomous adaptable machines).
  - Computer games (for instance, intelligent agents).
  - Financial modelling (for instance, predicting stocks or shares value).
  - Other time series prediction (for instance, weather forecasting).

[3] For a deeper analysis of applications of ANNs, see Abioudun et al. (2018).

It is then clear that ANNs have many advantages, otherwise the great success found in scientific applications would not be explained and in the following paragraphs, after presenting a short history of ANNs, we will explain their architecture, their working, their strengths, and their weaknesses.

3.1 A short history of *Artificial Neural Networks*

With the aim to describe the working of biological neurons, Warren McCulloch and Walter Pitts modelled the first Artificial Neural Network on electrical circuits. This happened in 1943, while a few years later, in 1949, Hebb contributed to the knowledge about the networks' functioning, demonstrating that the performance of the networks improved with each use. This was the first evidence that artificial neurons are able to learn as biological ones.

However, in those years, the computational power of computer was limited, but in the 1950s, in the IBM laboratories it was anyway possible to run a simple artificial neural network. Unfortunately, the experiment failed.

In the late 1950s, the so called "ADALINE" and "MADALINE" models were defined by Widrow and Hoff (1959, Stanford). The ADAptive LINear Elements (i.e., ADALINE) was used for recognizing dichotomous patterns of streaming bits of a phone line with the aim to predict the next bit. The Many ADALINE (i.e., MADALINE) was the first multi-layer *Artificial Neural Network*, built for eliminating the echoes on phone lines. This study would never have been born if in 1958 Rosenblatt had not created, from the observation of eye movement of a fly, the first neuron that was called Perceptron.

Researchers studied very seriously the mathematical formalization of learning capability of Perceptron, for example Widrow and Hoff defined different learning rules, but in 1969 Minsky and Papert published a book on weaknesses of Perceptron starting a dark period for the development of *Artificial Neural Networks*.

One of the main motivations of the failure of ANNs was surely the great expectations on these models that, on paper, were very powerful, but, in reality, they were really computationally intensive. In those same years, ethical discussions on the possibilities of "thinking machines" arose. This debate was really determinant in the failure of ANNs, just think that it is open until now.

Fortunately, the promising results obtained with simple neural networks and the growing computational capacity of computers played in favour to a reprise of ANNs, and between 1972 and 1975, Kohonen and Anderson proposed a prototype of the well-known Self-Organizing Map (SOM), representing the unsupervised networks more applied.

Starting from these years, the interest in ANNs was renewed and Hopfield defined a new idea of connections between neurons. Indeed, until now synapses travelled only from left to right (i.e., feed-forward); now Hopfield networks provide also for "travels" in the opposite direction (i.e., back-ward). At the same time, researchers studied more complex topologies of ANNs, where each layer solves different type of problem.

During the 1980s, studies focused on the mathematical formalization of learning algorithms and their optimization. The so-called "back-propagation" algorithm is the result of these researches, but the performances are not yet convincing in terms of learning rate.

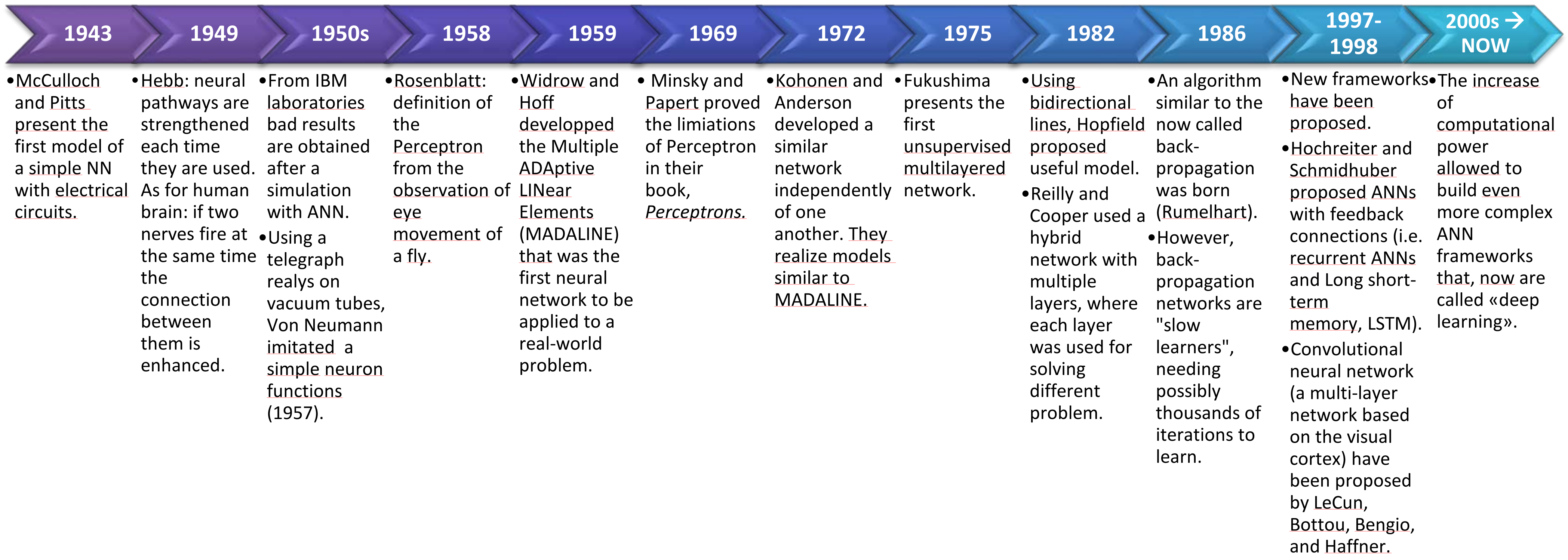
Contemporaneously to computational improvements in computer hardware and software, also complexity of ANNs increased. In 1990s years, recurrent networks, Long- and Short-Term Memory networks (LSTM), and Convolutional networks have been proposed whereas applications of complex ANNs were even more widespread.

In more recent years, new terminology has been adopted for ANN models with more layers with the aim to underline the ability of these models to “go on the deep” of data and extract knowledge from them. Hence, from now, multi-layer networks are also called *Deep Learning* models.

Sum up, as shown in figure 3, even if ANNs were born more than half century ago, they were not immediately appreciated, but now the *Deep Learning* methodologies are very powerful and its applications are considered in many fields.

From the history of ANNs and DL, literature converges in suggesting some main advantages for applying ANNs and for continuing to study in this field.

First of all, artificial neurons are very simple, moreover if compared to the big power of ANNs. Indeed, they are good learners and they are able to generalize behaviour minimizing errors. In general, unlike the standard models, they do not require any restrictions on data to analyse, and they are noise tolerant. Finally, thanks to the growing computational power, many types of networks are proposed and each of them is very flexible. The biggest weakness is that more complex is the framework, and more difficult is to have under control what happens in the deep layers. In the last paragraph of this handbook, strengths and weaknesses of ANNs will be deeply discussed.



1943	• McCulloch and Pitts present the first model of a simple NN with electrical circuits.
1949	• Hebb: neural pathways are strengthened each time they are used. As for human brain: if two nerves fire at the same time the connection between them is enhanced.
1950s	• From IBM laboratories bad results are obtained after a simulation with ANN. • Using a telegraph realys on vacuum tubes, Von Neumann imitated a simple neuron functions (1957).
1958	• Rosenblatt: definition of the Perceptron from the observation of eye movement of a fly.
1959	• Widrow and Hoff developped the Multiple ADAptive LINear Elements (MADALINE) that was the first neural network to be applied to a real-world problem.
1969	• Minsky and Papert proved the limiations of Perceptron in their book, *Perceptrons.*
1972	• Kohonen and Anderson developed a similar network independently of one another. They realize models similar to MADALINE.
1975	• Fukushima presents the first unsupervised multilayered network.
1982	• Using bidirectional lines, Hopfield proposed useful model. • Reilly and Cooper used a hybrid network with multiple layers, where each layer was used for solving different problem.
1986	• An algorithm similar to the now called back-propagation was born (Rumelhart). • However, back-propagation networks are "slow learners", needing possibly thousands of iterations to learn.
1997-1998	• New frameworks have been proposed. • Hochreiter and Schmidhuber proposed ANNs with feedback connections (i.e. recurrent ANNs and Long short-term memory, LSTM). • Convolutional neural network (a multi-layer network based on the visual cortex) have been proposed by LeCun, Bottou, Bengio, and Haffner.
2000s → NOW	• The increase of computational power allowed to build even more complex ANN frameworks that, now are called «deep learning».

Figure 3. Flow diagram of ANNs short-history.

3.2 From biology to mathematics

In previous section, we presented an *Artificial Neural Network* as the informatics representation of human neurons. Now, we start from the biology in order to understand as translating in mathematical formalization the working of a neuron.

Artificial Neural Networks (ANNs): are a representation of human neural brain in mathematical terms.

To doing this, it is necessary to present the biological neuron and its main characteristics. Figure 4 shows a picture of a biological neuron and its drawing with labels of its different parts. In details:

- A neuron is an electrically excitable cell communicating with other cells via specialized connections called *synapses*. It is the main component of nervous tissue in all animals. Plants and fungi do not have nerve cells. Within the soma, there is the *nucleus* that contains the genetic material in the form of chromosomes
- A typical neuron consists of a cell body (*soma*), *dendrites*, and a *single axon*. The axon and dendrites are filaments that extrude from the soma. Dendrites typically branch profusely and extend a few hundred micrometers from the soma.
- At the farthest tip of the axon's branches are *axon terminals*, where the neuron can transmit a signal across the synapse to another cell.

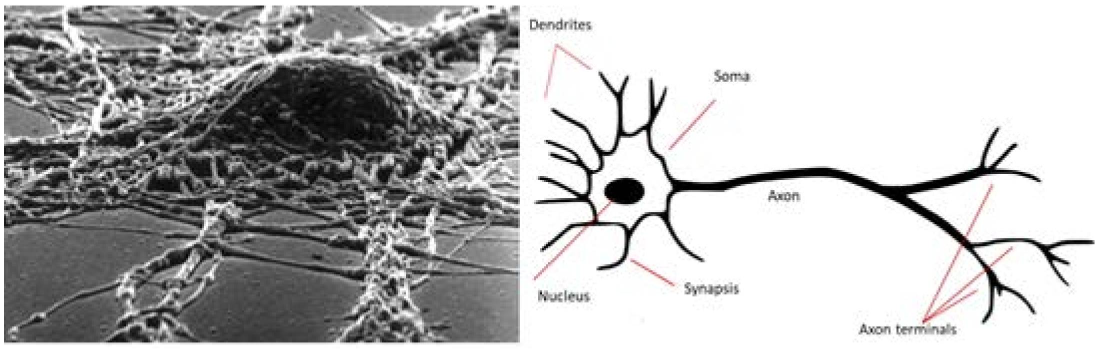


Figure 4. Biological Neuron (Adapted from Wikimedia Commons).

Even if it may seem very hard to traduce the neuron in mathematical terms, figure 5 shows all fundamental elements, that basically are only three:

1. A *set of synapses* or *connecting links*. Unlike a synapse in the brain, the synaptic weight of an artificial neuron may lie in a range that includes negative as well as positive values.
2. An *adder* for summing the input signals, weighted by the respective synapses of the neuron; the operations described here constitute a linear combiner.
3. An *activation function* for limiting the amplitude of the output of a neuron.

An *Artificial Neural Network* is composed by artificial neurons linked together by weights and organized in layers.

The layer represents a collection of neurons, but in simple frameworks, it can also consist of just one neuron (or node). Hidden layers are layers between inputs and output layer, where ANN's weights are calculated and stored. Note that the counting index of layers starts with the first hidden layer up to the output layer. However, in general, the set of inputs is also indicated as a layer (i.e., input layer), but this is not counted in the number of layers of the network.

The neuron is then the main element of an ANN and only understanding its working it is possible to investigate how results of *Deep Learning* models are obtained.

The first basic model was proposed by McCulloch and Pitts in the mid-1950s (the so-called "Threshold Logic Unit", TLU), and a more sophisticated version has been studied by Rosenblatt in 1957 that defined a single layer made by more TLUs (the so-called "Perceptron"). Nevertheless, in general, the Perceptron refers to a single TLU.

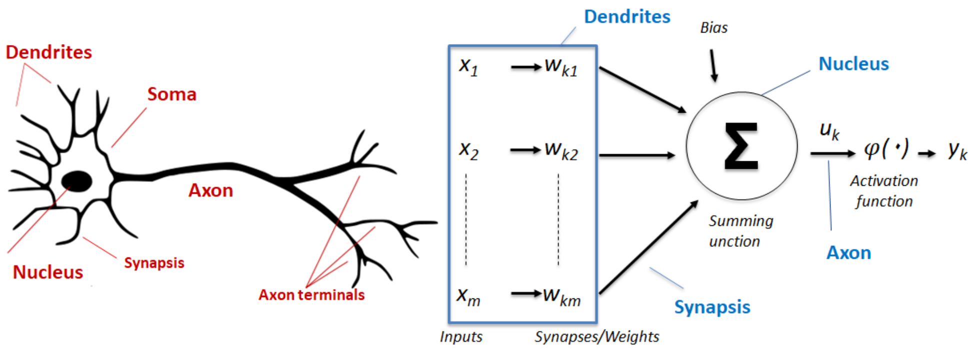


Figure 5. From biological to artificial neuron (Adapted from Wikimedia Commons).

Let us consider figure 6. We can define the *k*-th neuron, u_k, as:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j$$

and its output, y_k, as:

$$y_k = \varphi(u_k + b_k)$$

From this simple formulation, it is clear that the result (y_k) depends from weights. Indeed, in ANN models, synaptic weights are the real unknown. They determine the connections, that represent the collection of knowledge extracted by the network.

In addition, the neuronal model also includes an external bias, denoted with b_k. The bias can be positive or negative and it affects the activation of neuron (neuron fires). This parameter improves the flexibility and the fitting of the neural network.

In general, the bias is considered as an input (x_0) equal to +1, then the *k*-th neuron is represented as follows:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \qquad [1]$$

and its output

$$y_k = \varphi(u_k) \qquad [2]$$

The weight of the bias assumes a role very important because it allows the “activation state” of the neuron. If the neuron does not activate, its weights decrease affecting seriously the feature detection process.

Last but not least, result depends also from the activation or transfer function $\varphi(u_k)$. Literature suggests several functional forms that we will discuss later.

The **activation or transfer function** defines the output of a neuron given an input or a set of inputs.

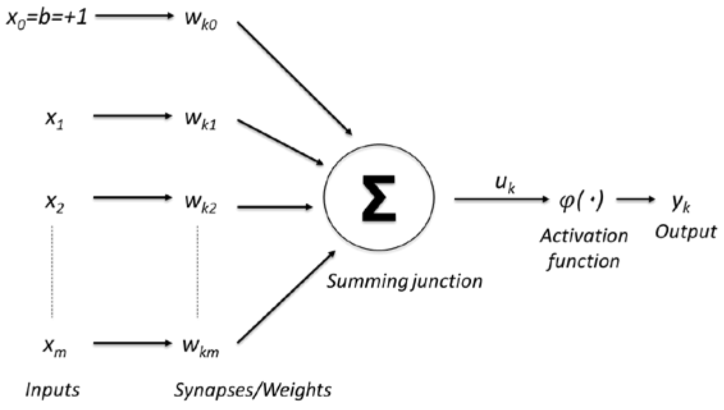


Figure 6. The Perceptron.

In order to understand the role of bias, let us consider figure 7. In figures (a) and (b) is drawn a simple neuron with 2 inputs and 1 output. The transfer function is the Rectified Linear Unit (ReLU): it returns 0 if the argument is negative, otherwise it returns the maximum value of the function.

Let us consider figure 7 (a) and set b = +1:

$$u_k = \sum_{j=1}^{m} w_{kj}x_j =$$

= 0*1 + 1*0.35 + 2*(-2) = -3.65

the output will be:

$$y = \varphi(u_k) = \max\{0; -3.65\} = 0$$

Results are 0 and -3.65, but the ReLU function returns 0 when results are negative, then *y* is equal to 0.

Looking at figure (b) and b = +1, previous equations change as follows:

$$u_k = \sum_{j=1}^{m} w_{kj}x_j =$$

= 4*1 + 1*0.35 + 2*(-2) = 0.35

and

$$y = \varphi(u_k) = \max\{0; +0.35\} = 0.35$$

In this case result is 0.35.

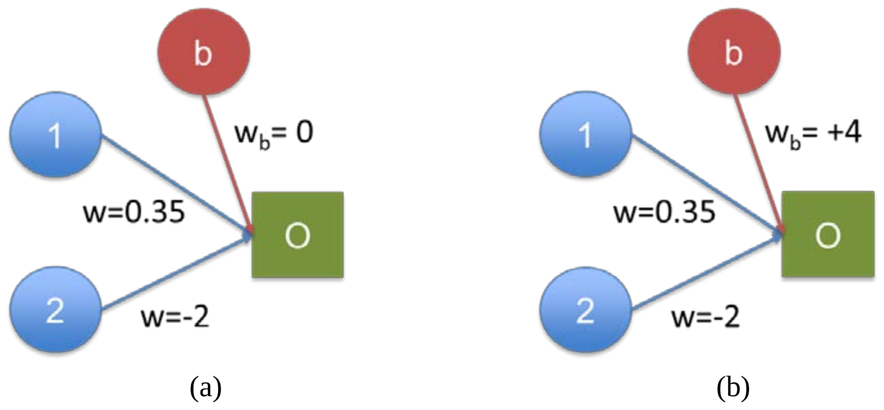


Figure 7. The role of bias weights.

Each neuron becomes active if the amount of signal it receives exceeds an activation threshold, then in the first case (a), the neuron is not activated (signal=0), on the contrary, in the second case (b), the signal is positive and the neuron is activated.

In this section we have understood as a biological neuron can be translated in mathematics, and, in addition, we have run a simple example showing what means when neuron is activated.

Now, we are ready for building our first Perceptron exploiting the logic gate operators and displaying that the non-linearity of network allows to solve easily simple problems. However, before continuing it is necessary to precisely define the data and their labels adopted from now on:

- *Input data*: are data entering into the model. They are organized in a matrix where, in general, rows are observations and columns variables (x_i).
- *Target data*: are the known results for each subject presented into the network.
- *Output data*: are the results of the network for each subject presented into the model.

The **Perceptron** is a single-layer neural network, so that the simplest architecture of *Artificial Neural Network*.

For instance, suppose we want to classify some cakes into two classes (i.e., saleable and not-saleable). We know a set of variables characterizing the quality of 10 cakes: height, diameter, weight, and pastry color (five classes of colors). The input layer is represented by a 10x4 matrix where, in general, rows are cakes and columns are their characteristics (Figure 8 (a)). The target is represented by a column vector of 10 elements equal to 0 if the corresponding cake is not-saleable and 1 otherwise (Figure 8 (b)). The output is a column vector with 10 responses of the ANN (one for each cake) to the question "considering these 4 variables, is this cake saleable?" (Figure 8 (c)). Clearly, we hope that the neuron is able to learn well from input and target in order to recognize if cake is good or not, because, in this manner, we can delegate to him the "quality control" before the sale.

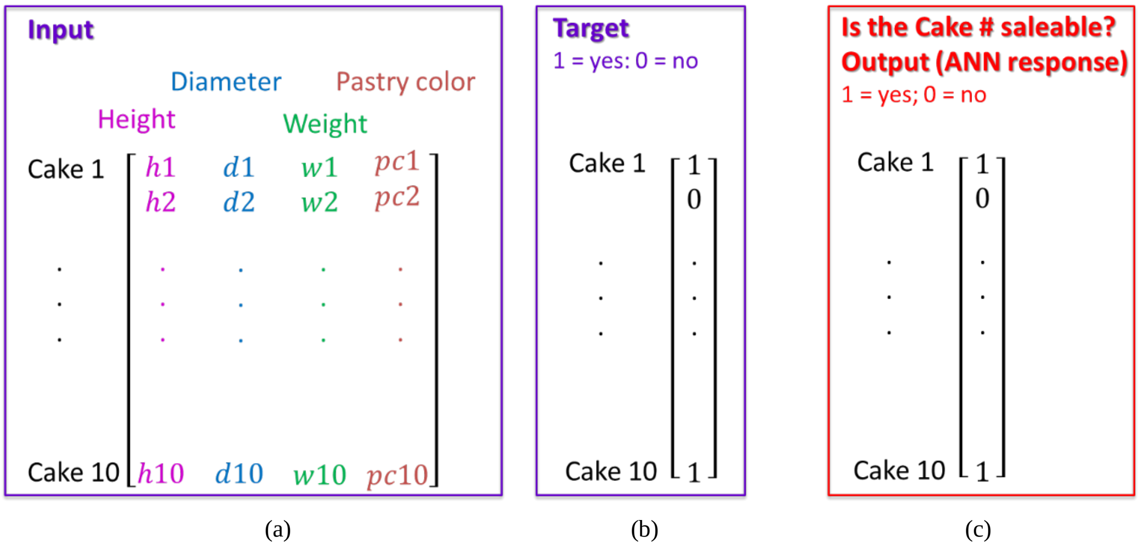


Figure 8. Dataset for ANNs.

3.3 Logic Gate Operators: linear and non-linear separability

In this section we approach with three logic gate operators with the aim to show the power of neuron/network in solving non-linear separable problems.

Let us consider True equal to 1 and False equal to 0, we study the following three logic gate operators:

Logic gate operator: is a model or a device implementing a Boolean function. Starting from binary inputs, the model produces a binary output.

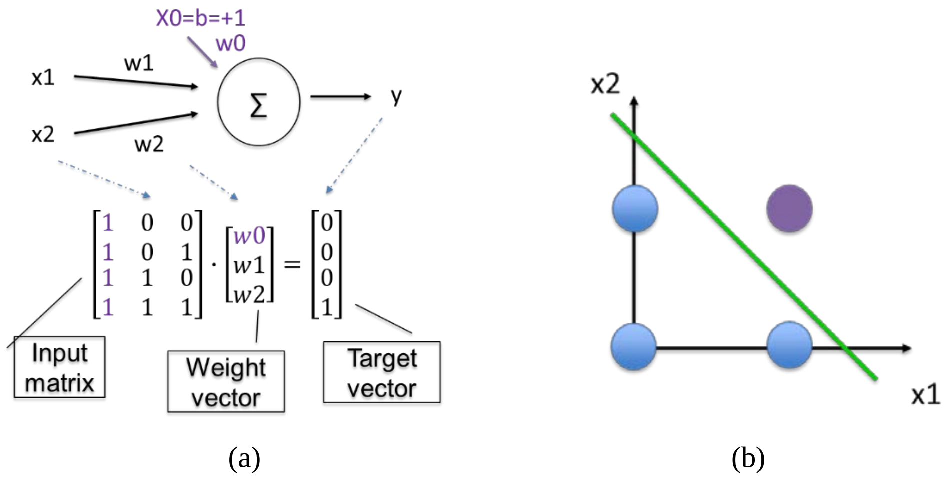
1. AND: True if both inputs are true, False otherwise.
2. OR: True if at least one operand is true, False if both are false.
3. XOR: True if both operands are different. False if both are true or false.

Let us start with the **AND operator**. Inputs (x1 and x2) and targets (y) are the following:

x1	**x2**	**y**
0	0	0
0	1	0
1	0	0
1	1	1

Figure 9 (a) represents inputs, bias, weights and targets. The input matrix has 4 rows (i.e., the number of elements for each variable), and 3 columns (i.e., the bias, x1, and x2). The bias is always +1, weights are collected in a vector of three rows, one for each presented variable. We know the target with the correct response of the neuron for each pair of elements (i.e., target vector). Figure 9 (b) shows the drawing of the problem: find the *boundary* allowing to classify correctly the items. The green straight line divides the plane into two subsections separating the two classes (in light-blue the False targets; in purple the True target).

The neuron starts assigning to w1 and w2 a value (i.e., +1), then it finds the w0. The assignment of initial values to weights can be set to a fixed value or random. Later in the text, we will see how from the initial values, the neuron/network is able to find the weights minimizing the errors in classification or prediction (i.e., back-propagation algorithm).



$$\begin{bmatrix}1&0&0\\1&0&1\\1&1&0\\1&1&1\end{bmatrix}\cdot\begin{bmatrix}w0\\w1\\w2\end{bmatrix}=\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

Figure 9. The AND problem.

Since the problem is to classify the input into two classes, we adopt as transfer function the *step* or *hard-limit* function:

$$hardlim(u_k) = \begin{cases} 1 \; if \; u_k \geq 0 \\ 0 \; if \; u_k < 0 \end{cases}$$

Starting from input and target matrixes, considering equations [1] and [2], and the transfer function, we can define the following system:

$$\begin{cases} w0 \cdot 1 + w1 \cdot 0 + w2 \cdot 0 \\ w0 \cdot 1 + w1 \cdot 0 + w2 \cdot 1 \\ w0 \cdot 1 + w1 \cdot 1 + w2 \cdot 0 \\ w0 \cdot 1 + w1 \cdot 1 + w2 \cdot 1 \end{cases} \qquad [3]$$

Let us consider the first equation of the system:

$$w0 \cdot 1 + w1 \cdot 0 + w2 \cdot 0 \qquad [4]$$

the corresponding target is 0, then, considering the hard-limit function, the [4] must be lower than 0.

The same consideration can be done for the second and the third equations.

Looking at the fourth expression:

$$w0 \cdot 1 + w1 \cdot 1 + w2 \cdot 1 \qquad [5]$$

in this case the corresponding target is 1, then, following the hard-limit function, the [5] must be higher than 0.

The system [3] is now as follows:

$$\begin{cases} w0 \cdot 1 + w1 \cdot 0 + w2 \cdot 0 < 0 \\ w0 \cdot 1 + w1 \cdot 0 + w2 \cdot 1 < 0 \\ w0 \cdot 1 + w1 \cdot 1 + w2 \cdot 0 < 0 \\ w0 \cdot 1 + w1 \cdot 1 + w2 \cdot 1 \geq 0 \end{cases}$$

If w1 and w2 are equal to +1, the system is reduced as follows:

$$\begin{cases} w0 < 0 \\ w0 + 1 < 0 \\ w0 + 1 < 0 \\ w0 + 2 \geq 0 \end{cases} \qquad \begin{cases} w0 < 0 \\ w0 < -1 \\ w0 < -1 \\ w0 \geq -2 \end{cases} \qquad [6]$$

and the optimal value for w0 ranges between -2 and -1, as shown in figure 10.

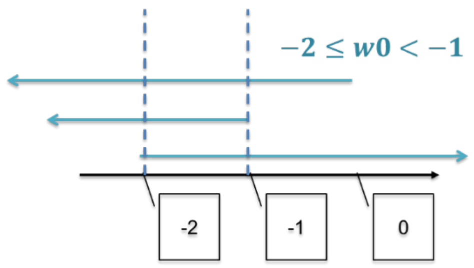


Figure 10. Solution for AND logic gate operator.

Setting w0 = -1.5, we can find the evidence that the value fit well substituting -1.5 in [6].

$$\begin{cases} -1.5 < 0 \\ -1.5 < -1 \\ -1.5 < -1 \\ -1.5 \geq -2 \end{cases}$$

This example shows that the defined Perceptron is able to find the optimal boundary separating the plane. Starting from inputs and targets, and assigning randomly values to weights (w1 and w2), we were able to find the weight w0 for the AND logic gate problem.

What happens for the **OR logic gate**?

Let us consider the following inputs (x1 and x2) and targets (y) representing the **OR operator**:

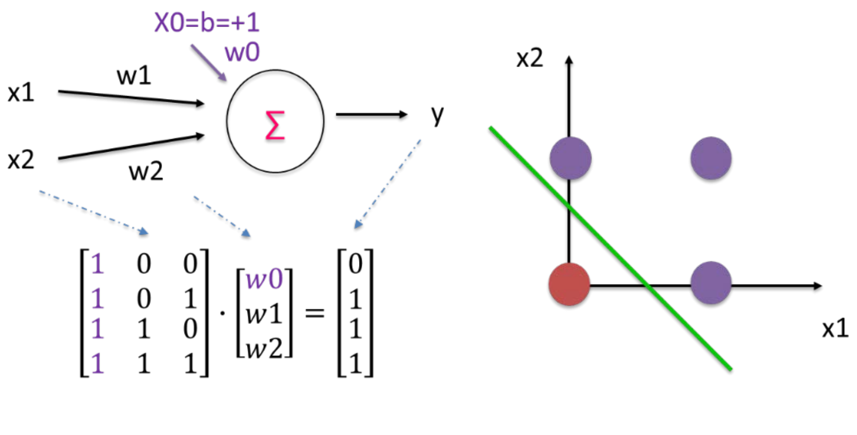
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

OR: True if at least one operand is true, False if both are false.

Figure 11 (a) presents the neuron for solving the OR operator problem. The first column of the input matrix shows all unitary values, because it represents the bias. The vector of weights considers weights for input 1 and input 2, whereas the target vector identifies the correct response according to the OR rule.

As in the case of AND logic gate, figure 11 (b) shows that the solution is to find values of weights that, given those targets, are able to draw the boundary separating the plane into two sections (in red the False target; in purple the True targets).

The procedure for finding the optimal result is the same seen before, as well as the hard-limit function as activation rule.



$$\begin{bmatrix}1 & 0 & 0\\1 & 0 & 1\\1 & 1 & 0\\1 & 1 & 1\end{bmatrix}\cdot\begin{bmatrix}w0\\w1\\w2\end{bmatrix}=\begin{bmatrix}0\\1\\1\\1\end{bmatrix}$$

(a) (b)

Figure 11. The OR problem.

Starting from input and target matrixes, considering equations [1] and [2], and the hard-limit transfer function, we can define the following system:

$$\begin{cases}w0\cdot 1+w1\cdot 0+w2\cdot 0\\w0\cdot 1+w1\cdot 0+w2\cdot 1\\w0\cdot 1+w1\cdot 1+w2\cdot 0\\w0\cdot 1+w1\cdot 1+w2\cdot 1\end{cases} \qquad [7]$$

Let us consider the first equation of the system:

$$w0\cdot 1+w1\cdot 0+w2\cdot 0 \qquad [8]$$

the corresponding target is 0, then, considering the hard-limit function, the [8] must be lower than 0.

Unlike what happened to the AND operator, the second, the third, and the fourth elements present a target equal to 1, then the corresponding equations have to be higher than 0.

The system [7] is now as follows:

$$\begin{cases}w0\cdot 1+w1\cdot 0+w2\cdot 0<0\\w0\cdot 1+w1\cdot 0+w2\cdot 1\geq 0\\w0\cdot 1+w1\cdot 1+w2\cdot 0\geq 0\\w0\cdot 1+w1\cdot 1+w2\cdot 1\geq 0\end{cases}$$

If w1 and w2 are equal to +1, the system is reduced as follows:

$$\begin{cases}w0<0\\w0+1\geq 0\\w0+1\geq 0\\w0+2\geq 0\end{cases} \qquad \begin{cases}w0<0\\w0\geq -1\\w0\geq -1\\w0\geq -2\end{cases} \qquad [9]$$

and the optimal value for w0 ranges between -1 and 0, as shown in figure 12.

Setting w0 = -0.5, we can find the evidence that the value fit well substituting -0.5 in [9].

$$\begin{cases}-0.5<0\\-0.5\geq -1\\-0.5\geq -1\\-0.5\geq -2\end{cases}$$

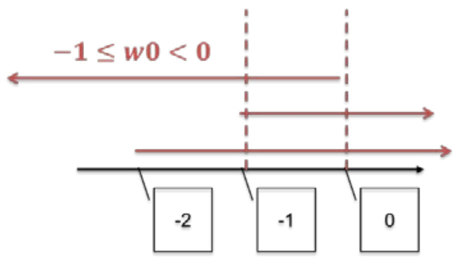


Figure 12. Solution for OR logic gate operator.

As in the case of AND operator, we find the weight for the bias (w0) able to identify a boundary for separating the plane and finding the correct solution for the OR problem.

Finally, figure 13 reassumes the neurons built for AND and OR operators with the solution in the Cartesian plane.

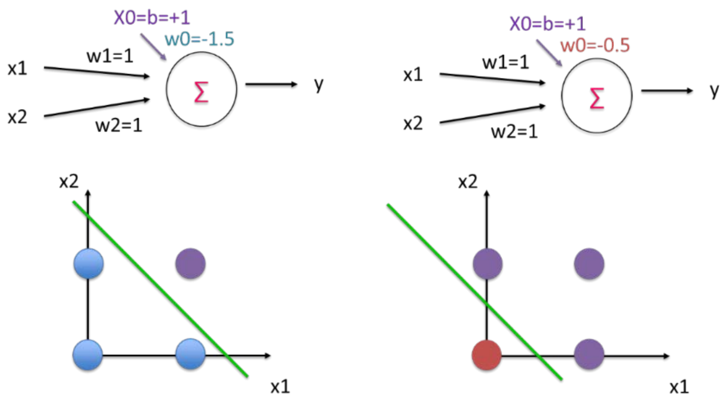


Figure 13. Perceptrons for AND and OR logic gate operators.

What about the **XOR logic gate operator**?

In the case of XOR, the inputs (x1 and x2) and the targets are the following:

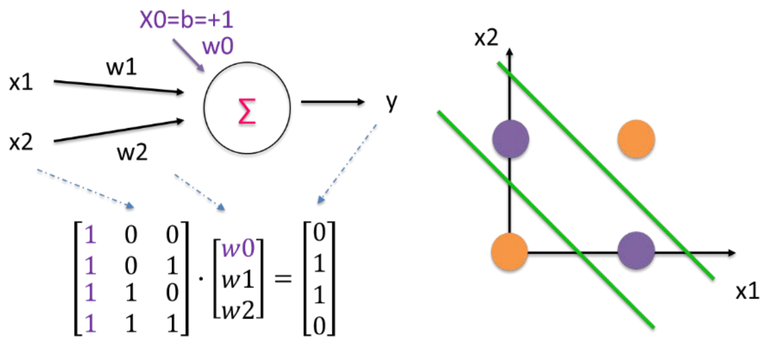
x1	**x2**	**y**
0	0	0
0	1	1
1	0	1
1	1	0

XOR: True if both operands are different. False if both are true or false.

Figure 14 (a) presents the neuron built for solving the problem. Input, weight and target matrixes have the same size od previous cases, and the first column of input matrix represents the bias.

Figure 14 (b) shows the XOR operator in the cartesian plane. In this specific case, we find that it is necessary to define two straight lines, two boundaries, for solving the problem (in orange the False targets; in purple the True targets).

However, let us start, as in previous cases, applying the hard-limit function to our neuron.



$$\begin{bmatrix}1&0&0\\1&0&1\\1&1&0\\1&1&1\end{bmatrix}\cdot\begin{bmatrix}w0\\w1\\w2\end{bmatrix}=\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$$

(a) (b)

Figure 14. The XOR problem.

Starting from input and target matrixes, considering equations [1] and [2], and the hard-limit transfer function, we can define the following system:

$$\begin{cases} w0\cdot 1+w1\cdot 0+w2\cdot 0\\ w0\cdot 1+w1\cdot 0+w2\cdot 1\\ w0\cdot 1+w1\cdot 1+w2\cdot 0\\ w0\cdot 1+w1\cdot 1+w2\cdot 1\end{cases} \qquad [10]$$

Let us consider the first equation of the system:

$$w0\cdot 1+w1\cdot 0+w2\cdot 0 \qquad [11]$$

the corresponding target is 0, then, considering the hard-limit function, the [11] must be lower than 0. The same result is found for the last equation of [10], whereas the second and the third equation have to be higher than 0 because the corresponding target is 1. The final system is now:

$$\begin{cases} w0\cdot 1+w1\cdot 0+w2\cdot 0<0\\ w0\cdot 1+w1\cdot 0+w2\cdot 1\geq 0\\ w0\cdot 1+w1\cdot 1+w2\cdot 0\geq 0\\ w0\cdot 1+w1\cdot 1+w2\cdot 1<0\end{cases}$$

If w1 and w2 are equal to +1, the system is reduced as follows:

$$\begin{cases} w0<0\\ w0+1\geq 0\\ w0+1\geq 0\\ w0+2<0\end{cases} \qquad \begin{cases} w0<0\\ w0\geq -1\\ w0\geq -1\\ w0<-2\end{cases}$$

It is clear that in this manner it is impossible to find the correct solution. We can see the evidence in figure 15, where is reported the solution of the system for w0.

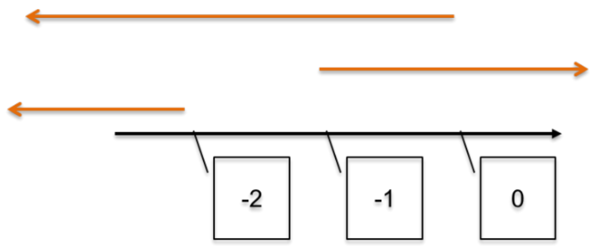


Figure 15. Solution for XOR problem with a single neuron.

Obviously the neuron used until now does not work, but the solution is very simple. Seeing figure 14 (b) the solution is clear, we have to use two neurons, one for each boundary.

For solving the XOR problem, we need another operator, the **NAND logic gate** (Not-AND). This operator is an inverted AND gate, as shown in figure 16 (in light-blues the False targets; in purple the True target).

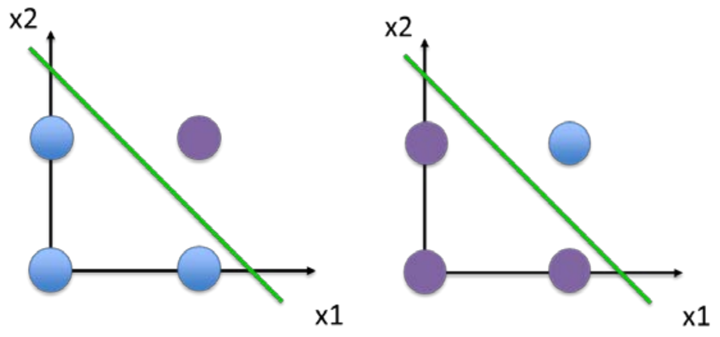


Figure 16. AND and NAND operators.

Let us consider the following inputs (x1 and x2) and target (y) for NAND operator:

x1	**x2**	**y**
0	0	1
0	1	1
1	0	1
1	1	0

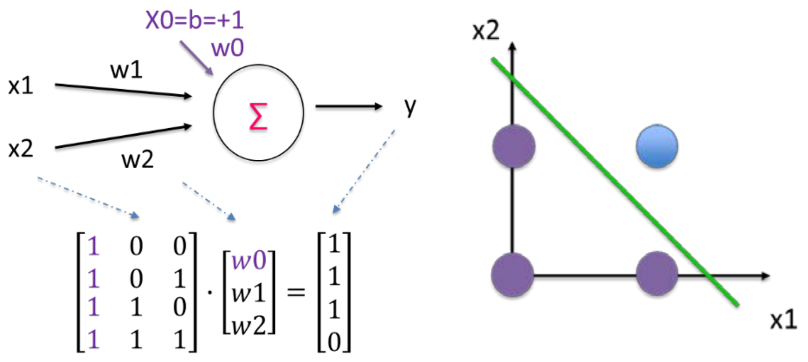
NAND: False if both operands are true. True otherwise.

Remember that inputs and target for the AND operator are exactly the opposite:

x1	**x2**	**y**
0	0	0
0	1	0
1	0	0
1	1	1

AND: True if both inputs are true, False otherwise.

Figure 17 (a) presents the neuron with inputs, target, weight matrixes where the first column of the input matrix is represented by the bias. Figure 17 (b) presents newly the visualization of the problem (in light-blues the False target; in purple the True targets).



$$\begin{bmatrix}1 & 0 & 0\\1 & 0 & 1\\1 & 1 & 0\\1 & 1 & 1\end{bmatrix}\cdot\begin{bmatrix}w0\\w1\\w2\end{bmatrix}=\begin{bmatrix}1\\1\\1\\0\end{bmatrix}$$

(a) (b)

Figure 17. The NAND problem.

Starting from input and target matrixes, considering equations [1] and [2], and the hard-limit transfer function, we can define the following system:

$$\begin{cases} w0 \cdot 1 + w1 \cdot 0 + w2 \cdot 0 \\ w0 \cdot 1 + w1 \cdot 0 + w2 \cdot 1 \\ w0 \cdot 1 + w1 \cdot 1 + w2 \cdot 0 \\ w0 \cdot 1 + w1 \cdot 1 + w2 \cdot 1 \end{cases} \qquad [12]$$

Let us consider the first equation of the system:

$$w0 \cdot 1 + w1 \cdot 0 + w2 \cdot 0 \qquad [13]$$

the corresponding target is 1, then, considering the hard-limit function, the [13] must be higher than 0, and the same reasoning can be done for the second and the third equation. Only last equation should be lower than 0. The system is then the following:

$$\begin{cases} w0 \cdot 1 + w1 \cdot 0 + w2 \cdot 0 \geq 0 \\ w0 \cdot 1 + w1 \cdot 0 + w2 \cdot 1 \geq 0 \\ w0 \cdot 1 + w1 \cdot 1 + w2 \cdot 0 \geq 0 \\ w0 \cdot 1 + w1 \cdot 1 + w2 \cdot 1 < 0 \end{cases}$$

If w1 and w2 are equal to -1 (NAND is the opposite of AND, then we can set the weights with the same value, but with the opposite sign), the system can be reduced as follows:

$$\begin{cases} w0 \geq 0 \\ w0 - 1 \geq 0 \\ w0 - 1 \geq 0 \\ w0 - 2 < 0 \end{cases} \qquad \begin{cases} w0 \geq 0 \\ w0 \geq 1 \\ w0 \geq 1 \\ w0 < 2 \end{cases} \qquad [14]$$

and the optimal value for w0 ranges between +1 and +2, as shown in figure 18.

Setting w0 = +1.5, we can find the evidence that the value fit well substituting +1.5 in [14].

$$\begin{cases} +1.5 \geq 0 \\ +1.5 \geq 1 \\ +1.5 \geq 1 \\ +1.5 < 2 \end{cases}$$

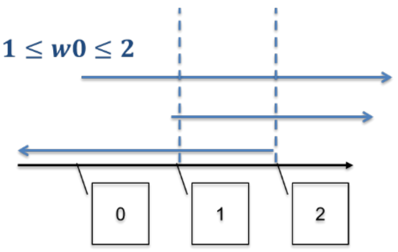


Figure 18. Solution for NAND problem.

Now, we are ready for solving the **XOR operator**. In details, we need to build an *Artificial Neural Network* with two layers and three neurons (two in the hidden layer and one in the output layer):

1. Neuron 1: NAND operator (hidden layer).
2. Neuron 2: OR logic gate (hidden layer).
3. Neuron 3: AND operator (output layer).

XOR: True if both operands are different. False if both are true or false.

Figure 19 shows the visualization and the structure of the network. The network presents a bias for each layer, always equal to +1, but with unknown weights. The inputs x1 and x2 are both linked to the two neurons (i.e., NAND and OR) and these are linked together through an AND function (in purple the True targets, the False targets are respectively in light-blue, in red, and in orange).

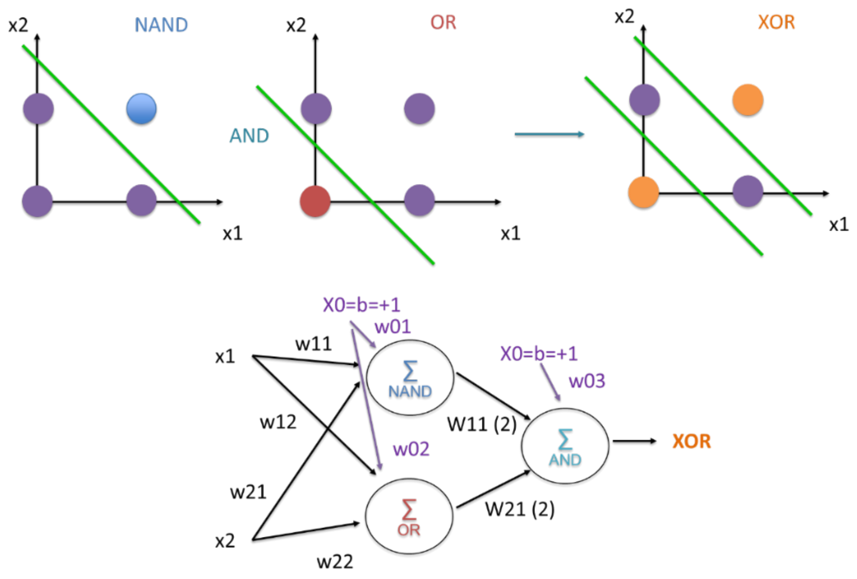


Figure 19. The XOR framework.

Even if figure 19 can seem complex, in reality, we already know the majority of parameters. We can start from the first layer, solving the NAND and the OR neurons (figure 20).

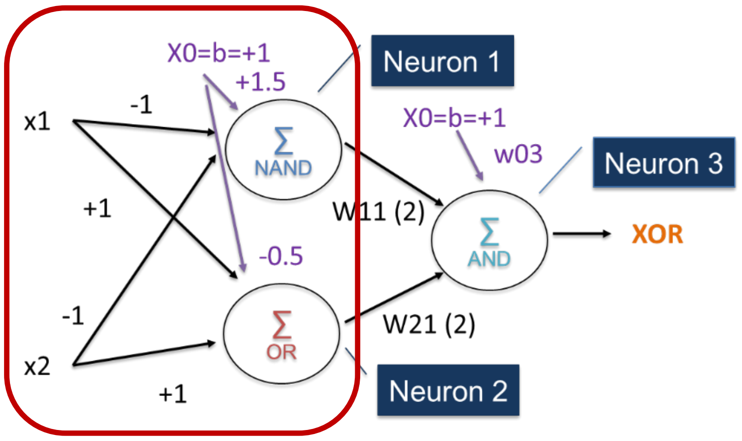


Figure 20. Parameters of the first layer.

Inputs and targets for XOR problem are:

x1	**x2**	**y**
0	0	0
0	1	1
1	0	1
1	1	0

and the activation function is always the hard-limit function.

Let us look at the NAND neuron. Weights for bias, x1, and x2 are respectively +1.5, -1, and -1, as found before. Substituting these parameters values in [12], we obtain the following system:

$$\begin{cases} 1.5\cdot 1 - 1\cdot 0 - 1\cdot 0 \\ 1.5\cdot 1 - 1\cdot 0 - 1\cdot 1 \\ 1.5\cdot 1 - 1\cdot 1 - 1\cdot 0 \\ 1.5\cdot 1 - 1\cdot 1 - 1\cdot 1 \end{cases}$$

Hard-limit function

$$hardlim(u_k) = \begin{cases} 1\ if\ u_k \geq 0 \\ 0\ if\ u_k < 0 \end{cases}$$

Solving and applying the hard-limit function, we find another vector:

$$\begin{cases} 1.5\cdot 1 - 1\cdot 0 - 1\cdot 0 = 1.5 \\ 1.5\cdot 1 - 1\cdot 0 - 1\cdot 1 = 0.5 \\ 1.5\cdot 1 - 1\cdot 1 - 1\cdot 0 = 0.5 \\ 1.5\cdot 1 - 1\cdot 1 - 1\cdot 1 = -0.5 \end{cases} \qquad \begin{cases} \geq 0 \dashrightarrow 1 \\ \geq 0 \dashrightarrow 1 \\ \geq 0 \dashrightarrow 1 \\ < 0 \dashrightarrow 0 \end{cases}$$

Looking at the first equation of the model, the result is +1.5, applying the hard-limit function the corresponding value is 1, because 1.5 is higher than 0. The same reasoning can be done also for the second and the third equation, whereas the result of the fourth is negative and then the corresponding value is 0.

Let us consider now the OR neuron. Weights for bias, x1, and x2 are respectively -0.5, +1, and +1, as found before. Substituting these parameters values in [7] we obtain the following system:

$$\begin{cases} -0.5\cdot 1 + 1\cdot 0 + 1\cdot 0 \\ -0.5\cdot 1 + 1\cdot 0 + 1\cdot 1 \\ -0.5\cdot 1 + 1\cdot 1 + 1\cdot 0 \\ -0.5\cdot 1 + 1\cdot 1 + 1\cdot 1 \end{cases}$$

Solving and applying the hard-limit function, we find another vector:

$$\begin{cases} -0.5\cdot 1 + 1\cdot 0 + 1\cdot 0 = -0.5 \\ -0.5\cdot 1 + 1\cdot 0 + 1\cdot 1 = 0.5 \\ -0.5\cdot 1 + 1\cdot 1 + 1\cdot 0 = 0.5 \\ -0.5\cdot 1 + 1\cdot 1 + 1\cdot 1 = 1.5 \end{cases} \qquad \begin{cases} < 0 \dashrightarrow 0 \\ \geq 0 \dashrightarrow 1 \\ \geq 0 \dashrightarrow 1 \\ \geq 0 \dashrightarrow 1 \end{cases}$$

Result from the first equation is lower than 0, then, applying the activation function, the assigned value is 0. In the other three cases, results are always higher than 0, and the result from transfer function is 1.

Now, we are ready for solving the output layer. At this point we have 2 input vectors, one from each neuron: x3 from the NAND operator and x4 from the OR one:

x3	**x4**
1	0
1	1
1	1
0	1

We know that the last neuron is an AND operator and we want to find the weight of the bias allowing to obtain results of XOR (figure 21). Applying the AND to x3 and x4, the *y* is:

$$\begin{bmatrix} \boldsymbol{x3} \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \text{AND} \quad \begin{bmatrix} \boldsymbol{x4} \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \longrightarrow \quad \begin{bmatrix} \boldsymbol{y} \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Notice that *y* is exactly the result of XOR.

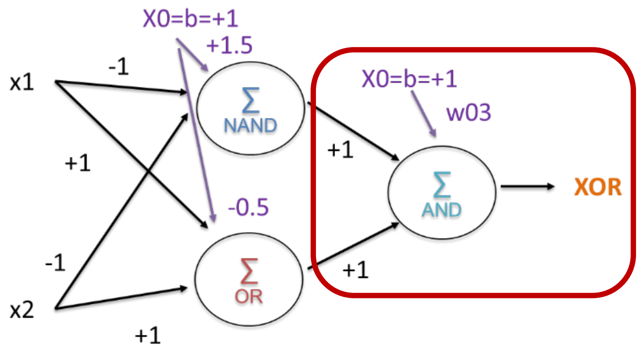


Figure 21. Parameters of the output layer.

However, we want to find the weight of bias (w03). Looking at [1] and [2], an input bias with unitary values, weights for x3 and x4 equal to +1, we can define the following system where w03 is the weight of bias and it is the unknown:

$$\begin{cases} w03 \cdot 1 + 1 \cdot 0 + 1 \cdot 0 \\ w03 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 \\ w03 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 \\ w03 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 \end{cases}$$

Looking at the first and the fourth equation and applying the hard-limit function, they can be set lower than 0. In the other cases (i.e., second and third equations), results have to be higher than 0. The system is the following:

$$\begin{cases} w03 \cdot 1 + 1 \cdot 0 + 1 \cdot 0 < 0 \\ w03 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 \geq 0 \\ w03 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 \geq 0 \\ w03 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 < 0 \end{cases}$$

The solution for w03 is then:

$$\begin{cases} w03 < 0 \\ w03 + 2 \geq 0 \\ w03 + 2 \geq 0 \\ w03 + 1 < 0 \end{cases} \qquad \begin{cases} w03 < 0 \\ w03 \geq -2 \\ w03 \geq -2 \\ w03 < -1 \end{cases}$$

and w03 ranges between -2 and -1 (for instance, w03=-1.5), as shown in figure 22.

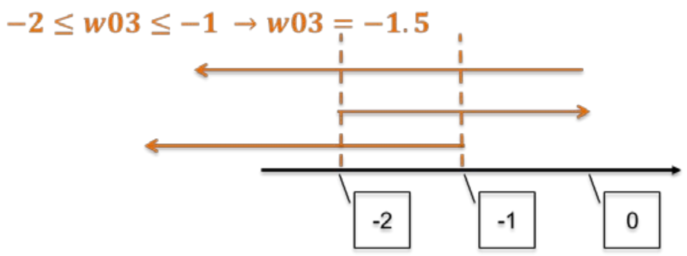


Figure 22. Solution of XOR problem with ANN.

Which is the lesson learnt?

In general, with two inputs, weights define a one-dimensional decision boundary that is a straight line in the bidimensional input space. With *n* inputs, weights define a decision boundary that is an *n–1* dimensional hyperplane in the *n* dimensional input space.

If input patterns can be classified using a single hyperplane, the problem is linearly separable (i.e., AND, OR or NAND). Otherwise, such as for XOR problem, the problem is not linearly separable and it is necessary to define more complex architectures and a single TLU is not enough.

Finally, presented exercises on logic gates allow us to understand how building a neuron and our first two-layer *Artificial Neural Network* (or *Shallow Neural Network*). Figure 23 shows the final result for XOR operator and the correct terminology in *Deep Learning* methodologies. The input layer contains the so-called "neurons" (even called "nodes") with the initial sample, the hidden layer is the "dark side" of network, that is the layer where neurons do magic and present signals to the target (or output) layer. There is not a rule for establishing the correct number of neurons in the hidden layer and, in addition, the "dark side" can be formed by more than one hidden layer (as a class of magicians!). When ANNs present only one or two hidden layers, they are also called "Shallow Neural Networks". If the architecture of network is more complex and the "dark side" is considerable (more hidden layers with many neurons), the ANN is called *Deep Learning*. Another characteristic of the ANN in figure 23 is that links among layers travel from left to right, and they do not come back, then we have built a *feed-forward shallow neural network*.

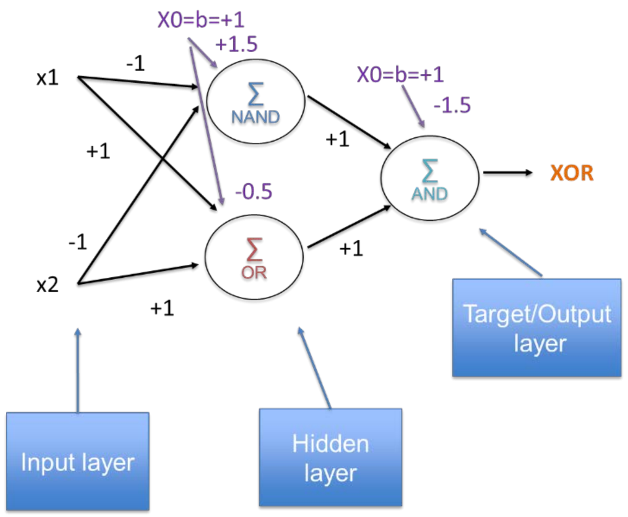


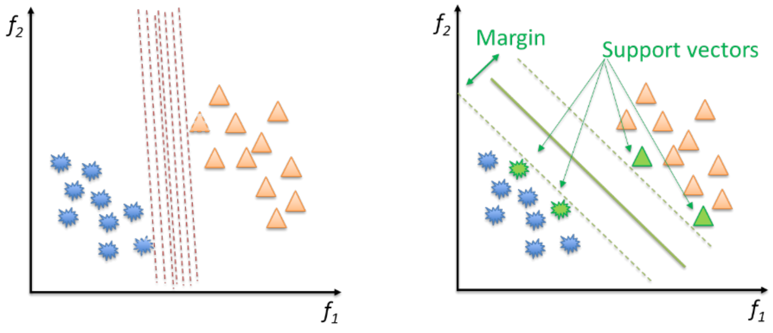
Figure 23. Two-layer feed-forward *Artificial Neural Network.*

3.4 Focus: *Artificial Neural Networks* versus *Support Vector Machines*

Support Vector Machines (SVMs) are a *Machine Learning* methodology for classification and non-parametric regression based on kernel functions. They have been proposed for the first time by Vapnik in 1995.

Support Vector Machines (SVM) are classifiers using hyperplanes able to separate two classes and when they apply a sigmoidal kernel function, they are equivalent to a Perceptron (1 input layer and 1 output layer). Elements of the initial sample are vectors with all characteristics (or features) describing them. The main goal of the algorithm is to find a hyperplane able to separate vectors, and all vectors near to the optimal hyperplane are called “support vectors”. Let us consider the figure 24, where there is a sample of data represented in two dimensions (i.e., the vectors contain two features f_1 and f_2. Each vector is an element of the database). From panel (a) of figure 24 we can easily understand that there is an infinite number of lines (dashed lines in red) that can separate the two classes represented. Panel (b) presents the results of the SVM algorithm searching the optimal boundaries maximizing the distance between the two groups (dashed lines in green). Boundaries are searched in order to be adjacent to vectors, that from now are called *support vectors* (in green in the panel (b)). The distance between the two boundaries is called *margin*, and in the area between the two boundaries there should be no points. Clearly, in reality it is possible (and highly probable) that data are in the area between boundaries. In this case, a technique called *softmap* is applied in order to consider the distance between the point in the area and the boundary near the support vector of the same class of the non-classified point (this is the case of non-completely separable classes).

From the mathematical point of view, the SVMs can be solved with a quadratic programming under constraints.



(a) (b)

Figure 24. Support Vector Machine algorithm.

However, not always it is possible separating the classes without difficulties, even with the *softmap* algorithm, then a solution is mapping data in a space (i.e., *feature space*) with more dimensions than the starting space. For instance, we can transform data from two dimensions in three, and observe if now classes are well separated. Even if this approach is computational expensive when the number of dimensions is high, the SVMs adopt a kernel function that allows to obtain the benefits of mapping in a higher-dimensional space without really applying it (i.e., the process is called *kernel trick*). There are many kernel functions (i.e., linear, polynomial, Gaussian, exponential, Laplace, sigmoidal, hyperbolic-tangent, quadratic, …), and choosing the optimal is not a simple work. The solution is trying empirically.

SVMs are very powerful *Machine Learning* techniques that are often used together with ANNs. However, they work well when classes are linearly or non-linearly separable, but when the dimensionality increases results are not satisfying, and their ability in classifying decreases dramatically.

For an exhaustive explaination of this *Machine Learning* technique, see Suthaharan S. (2016).

4 Modelling Artificial Neural Networks

4.1 Functions for ANNs

For solving the examples in previous sections, we have applied the three following functional operations without realizing it:

1. The *weight function*: it is the product of weights times input(s).
2. The *net input function*: it is the function used for considering the bias (in general, it is the product of a weight times the input, but other functions exist).
3. The *transfer* (or *activation*) *function*: it is the function producing the outputs starting from the net input (in the examples the ReLU or hard-limit/step function).

Equation [1]

$$u_k = \sum_{j=1}^{m} w_{kj} x_j$$

Equation [2]

$$y_k = \varphi(u_k)$$

First two functions are represented by the equation [1], and in general the sum of product is the most frequently adopted. However, for the record, many other functions exist and also the NN designer can define a new weight or net input function. For instance, in recent years, the convolution weight function is applied when the problem is the image detection. In this case, the function returns the mathematical convolution of weights matrix and inputs. Another example is the use of “distance functions”[4], very well-known in clustering models, as weight/net input functions. Later, we will present some network architectures requiring weight and net input function different from the sum of product.

On the contrary, the attention of the ANN user on the transfer function (i.e., equation [2]) has to be very high. Indeed, if we apply an activation function that does not fit, the ANN returns erroneous results.

4.1.1 Transfer or Activation functions

In this section, we briefly show the most frequently applied transfer functions in *Artificial Neural Networks*.

Let us consider equations [1] and [2] reassumed in the following expression:

[4] In Appendix A there is a brief description of main distance measures.

$$y = \varphi(u_k) = \varphi\left(\sum_{j=1}^{m} w_{kj} x_j\right)$$

Figure 25 shows two unit-step functions: the hard-limit (or step) function and the symmetric hard-limit one.

Hard-limit/step function:

$$hardlim(u_k) = \begin{cases} 1\ if\ u_k \geq 0 \\ 0\ if\ u_k < 0 \end{cases}$$

This function, that we have used intensively in the logic gates examples, describes the rule "all-or-none" of the basic model of McCulloch-Pitts.

Symmetric hard-limit function:

$$hardlims(u_k) = \begin{cases} 1\ if\ u_k \geq 0 \\ -1\ if\ u_k < 0 \end{cases}$$

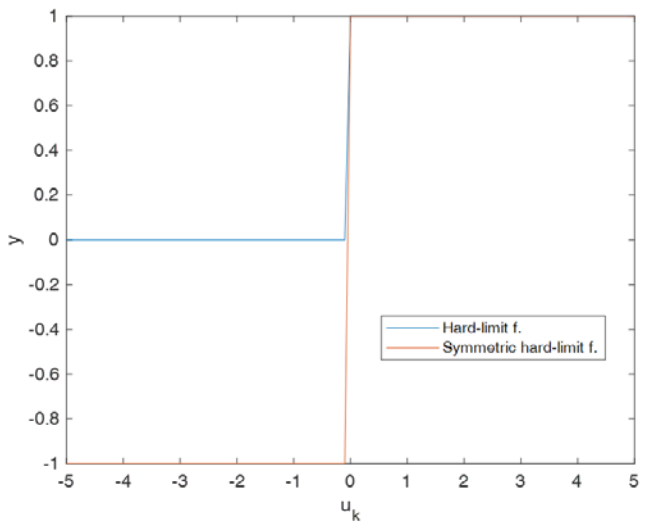


Figure 25. Hard-limit and Symmetric hard-limit functions.

In both cases, functions return only two values: the hard-limit function returns 0 or 1; whereas result of the symmetric hard-limit can be -1 or +1.

Figure 26 presents linear and positive linear functions. In general, the linear function is used in the output layer because the non-linearity, that is one of the main characteristics of *Deep Learning*, is represented in hidden layers.

Linear function:

$$purelin(u_k) = au_k + b$$

Positive linear function:

$$poslin(u_k) = \begin{cases} u_k\ if\ u_k \geq 0 \\ 0\ if\ u_k < 0 \end{cases}$$

The *ReLU function* applied in previous section is a particular case of positive linear function, where if result is positive, the function returns its maximum value.

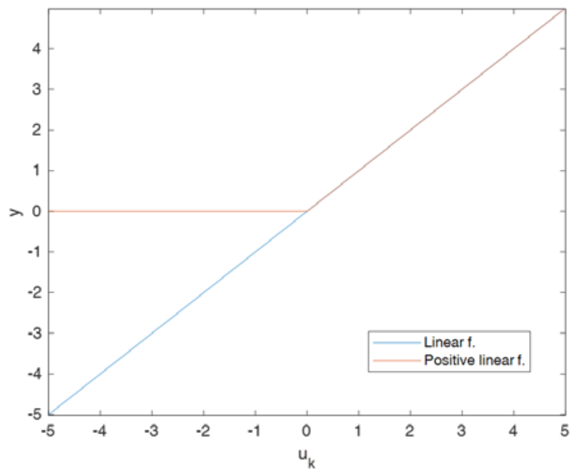


Figure 26. Linear and Positive linear functions.

Figure 27 shows the triangular basis and the radial-basis functions. The first one is applied rarely; on the contrary the radial basis function characterizes the homonymous ANNs presented in section 6.2.

Triangular-basis function:

$$tribas(u_k) = \begin{cases} 1\text{-}|u_k| \; if \; -1 \le u_k \le +1 \\ 0 \quad otherwise \end{cases}$$

Radial-basis function:

$$radbas(u_k) = e^{-u_k^2}$$

Normalized Radial-basis function:

$$radbasn(u_k) = \frac{e^{-u_k^2}}{\sum e^{-u_k^2}}$$

The function is equivalent to the radial-basis, except that output vectors are normalized by dividing by the sum of the pre-normalized values.

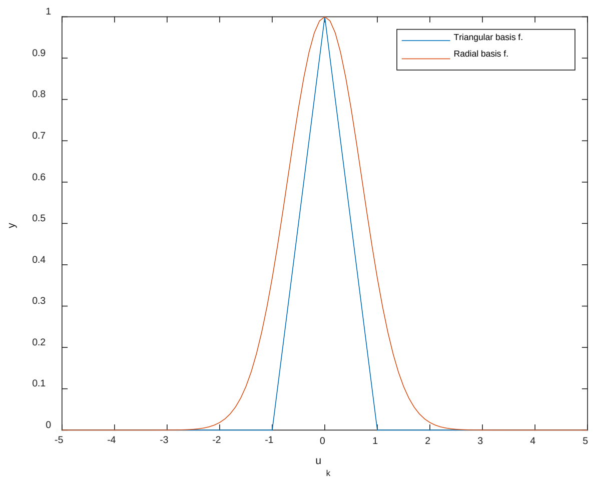


Figure 27. Triangular and Radial-basis functions.

Figure 28 presents two of the most frequently adopted functions: the logistic and the hyperbolic tangent sigmoid functions. These two functions are the continuous version of the hard-limit and symmetric hard-limit functions.

Logistic function:

$$sigmoid(u_k) = \frac{1}{1 + e^{-u_k}}$$

this function returns values between 0 and 1.

Hyperbolic tangent sigmoid function:

$$tanh(u_k) = \frac{e^{-u_k} - e^{-u_k}}{e^{-u_k} + e^{-u_k}}$$

this function returns values between -1 and +1.

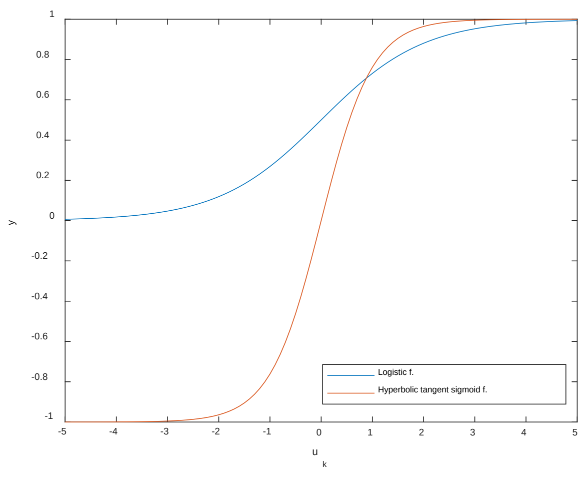


Figure 28. Logistic and Hyperbolic tangent sigmoid functions.

The following two functions are used when ANNs are applied for clustering. Indeed, the *competitive function* is successfully applied in Probabilistic Neural Networks (section 6.2.2), Self-Organizing Map (section 6.4.1) or Competitive Networks (section 6.4).

Softmax function:

$$softmax(u_k) = \frac{e^{u_k}}{\sum e^{u_k}}$$

This function is used also for transforming output between 0 and 1 and having as result a probabilistic value (Figure 29).

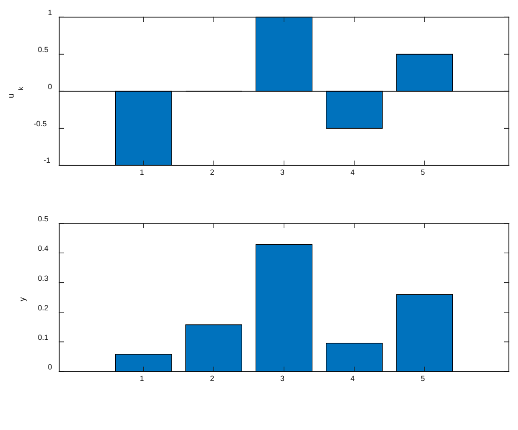


Figure 29. The softmax function.

The *Competitive function* assigns 1 to the maximum value presented.

For instance, let us see figure 30. If u_k= [-2; 0.9; 0.1; -0.2], then *y= compet(*u_k*)* = [0; 1; 0; 0]. The second input element is the highest one, then the corresponding result is 1.

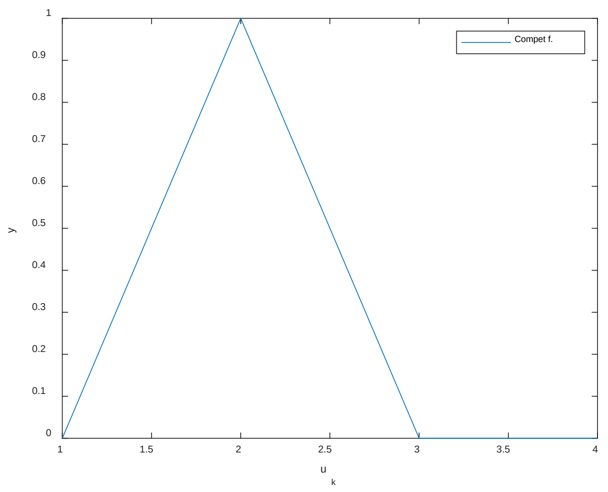


Figure 30. The competitive function.
(The first, the third, and the fourth element of the input vector u_k present a result equal to 0).

Which activation function to adopt is often a question mark, and the most efficacy method is "try, change, and try again". However, there are some considerations that we can take into account when we are programming the ANN. In details:

- Classification Problems #1 (for instance, 2 classes and 1 output)
    - For example, considering a two-layer network, we should use a *logistic/hard-limit* between the hidden and the target/output layer, and a *logistic* function between input and hidden layer. If the network presents more than one hidden layers, the optimal transfer function should be the logistic one. The logistic function can be substituted the *hyperbolic tangent sigmoid* function.
      Notice that the adoption of a logistic function in the output layer allows to obtain a probability result. For example, an output of 0.7 on a unit corresponding to a particular class would indicate a 70% chance that the input data represents a member of that class.
- Classification Problems #2 (for instance, multiple-classes and 1 output per class)

  - Between the hidden and the output layers, we should apply a *softmax* transfer function, whereas, in other layers a *logistic* or *a hyperbolic tangent sigmoid* function should be adopted.
    In general, if we have more than two classes, and we use one output unit for each class, we need to use the *softmax* activation function. However, it is also possible to adopt different network design, as suggested by Falavigna (2012).
- Regression/ Function Approximation Problems
  - In the last layer (from hidden to output/target layer), we should adopt a *linear* (or *positive linear*) transfer function, whereas in other layers a *logistic* or a *hyperbolic tangent sigmoid* function should be implemented.
    In this case, the output is not dichotomous or discrete, but continuous. In this case, the goal of the network is modelling the distribution of the output variables conditioned by the input variables.
- Clustering: in general, a *competitive* transfer function is applied.

Notice that previous considerations on the adoption of transfer functions are not rules, but only suggestions provided by applications presented by literature. Following the philosophy of ANNs, the model has to learn from reality, then also transfer functions can be different on the base of the specific analysed reality.

Finally, it is noteworthy that, when the expected result (i.e., the output) is dichotomous, it is possible to use a continuous transfer function and to apply an algorithm in order to find a threshold able to assign output values under the threshold to 0 and over the threshold to 1, as proposed by Falavigna (2012).

4.2 *Artificial Neural Networks* architectures

In previous sections, we have introduced the concept of neuron, the functions and its working, finally we have built our first two-layer feed-forward *Artificial Neural Network*.

However, even if our simple ANN works very well in solving logic gate problems, the research in this field has grown hand in hand with the increases of computational capabilities of computers.

In this paragraph, we present different architectures of networks considering the number of layers, the typology of connections, and finally the method of learning.

In the history of ANN, the first definition of a network is the Perceptron, represented by the connection of more Threshold Linear Unit (TLU) of McCulloch and Pitts.

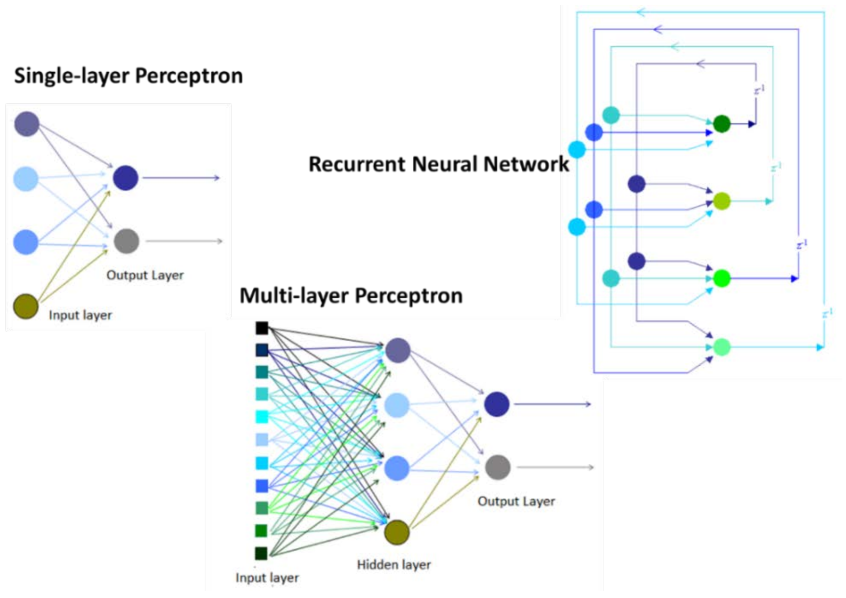
Increasing the complexity, and, at the same time, the flexibility, the ANNs presented more layers, more neurons, and not only feed-forward connections but also back-ward ones.

In this regard, please note that in ANNs the counting index of layers starts with the first hidden layer up to the output layer, and, in general, the set of inputs is also indicated as a layer (i.e., input layer), but this is not counted in the number of layers of the network. Until now, with logic gate operators, we have seen some single-layer networks (i.e., Perceptrons for AND, OR, NAND) and a *shallow ANN* (i.e., two-layer FFNN/Perceptron for XOR).

In details, we can identify three architectures (figure 31):

1. *Single-layer Feed-forward NNs* or (*Single-layer*) *Perceptron*: One input layer and one output layer of processing units. "Feed-forward" means that there are not feed-back connections: all synapses travel from left to right (e.g., a Single-layer Perceptron, figure 31 (a)).
2. *Multi-layer Feed-forward NNs* (less commonly called *Multi-layer Perceptron*): One input layer, one output layer, and one or more hidden layers of processing units. For instance, in figure 31 (b) is presented a two-layer feed-forward ANN (or *Shallow Neural Network*). Even in this case, "Feed-forward" means that synapses travel from left to right: feed-back connections are absent (note that if there are 2 hidden layers ANNs are called *Shallow Neural Networks* or *Deep Learning* if the complexity of architecture increases).
3. *Recurrent NNs*: Any network with at least one feed-back connection. It may, or may not, have hidden units. (e.g., a Simple Recurrent Network, figure 31 (c)).

Figure 32 presents the more frequently used neural networks subdivided on the base of connection typology. These frameworks will be presented later in the text. It is necessary to specify that models presented are not exhaustive, because the flexibility of NNs allows users to design network on the base of problem analysed.



(a) (b) (c)

Figure 31. Network architectures.

Another characteristic of the ANN working is the *training phase*. Thinking of logic gate examples, we found that a neuron or a simple ANN is able to learn from the target and find the best weight that minimizes the errors.

From this point of view, there are two different methods of training:

- *Supervised training* or *learning with a teacher*: the current output of the network is compared to the desired one (target). The weights are adjusted for minimizing the error between the current output and the target. This adjustment is carried out iteratively with the aim of eventually making the neural network obeys the teacher. A typical example of supervised learning is when students have written an exam, and having the exam marked by the teacher and shown which questions the students answered incorrectly. After being shown the correct answers, the students are expected to learn then understand how to answer those questions correctly.

- *Unsupervised training* or *learning without a teacher* or *self-organized learning*: there are only the inputs and targets are not presented to the network. The layer is competitive (i.e., competitive transfer function), and through the competition neurons are grouped. The layer adopts a “winner-takes-all strategy”, where the neuron with the greatest total input wins the competition and turns on; all the other neurons switch off. The most common unsupervised networks are the Self-Organizing (Feature) Maps (i.e., SOM or SOFM). They present an efficient learning algorithm able to represent non-linear complex functions. Example of unsupervised learning is the man that is learning how to ride a car by himself. He will start by entering the car and start the engine with the ignition key, then put down the clutch and put the car in first gear and press the accelerator for the car to move forward and then manage to control the steering. He continually practices the driving steps in a large open field and gradually masters the driving technique, and then over time starts to enter the road to drive skilfully.[5]

[5] The example is inspired by Beale et al. (2019).

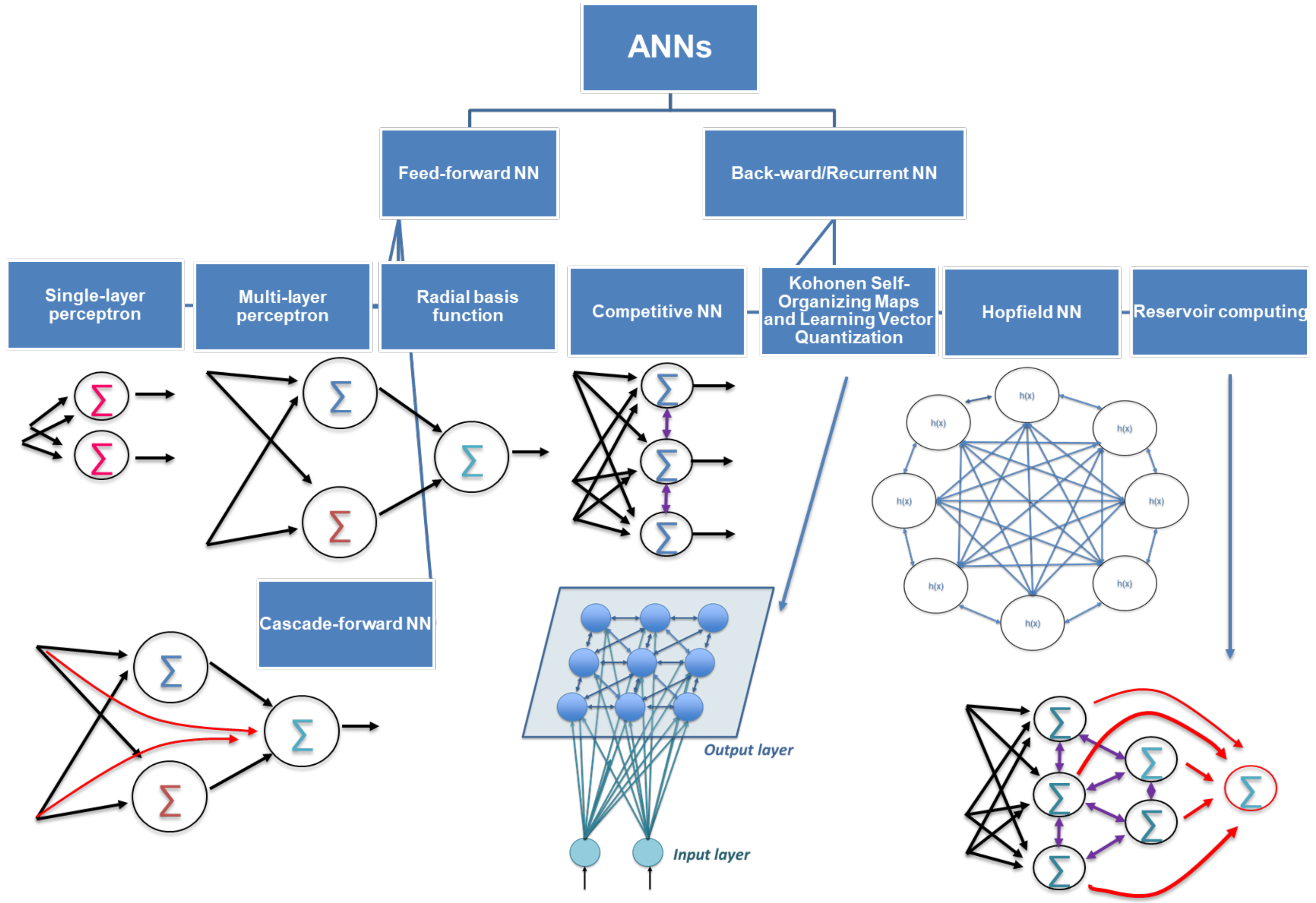


Figure 32. ANN architectures on the base of connections.

4.3 Training and Validation

Therefore, the time to understand how the ANN learns has come. First of all, it is necessary to point out the meaning of some terms.

The ANN is trained on a sample of data of which we know the input variables and also the targets. For example, we want to know the weather of tomorrow (sunny or wet). We build a sample of observations where input variables are pressure, temperature, wind, and so on… the target is the information for each observation of the weather (sunny or wet). In this first step (i.e., training), we divide the sample into two subsamples and we use the first one (i.e., the **training set**), that is the biggest (in general 2/3) for "training" our network. Here, the NN learns from the training set rules linking the inputs so that training patterns are correctly classified. Pay attention to weights because they are the collectors of these information.

Once the training sample is correctly classified (our forecasting is correct) or the error is very small, the remaining observations of the initial sample (1/3, the **validation set**) is used for validating and testing if ANN has learned well in the training phase. This validation phase is crucial because is a test for the network: if ANN does not fail in the validation phase it is ready to classify patterns it has never "seen" before (the so called "out-of-sample" observations). In this manner, the ANN applies the learnt lesson from the training set on remaining data (validation set), testing its ability to generalize.

Figure 33 shows these two phases: the first is a *learning* or *training phase*. The ANN learns to recognize rules among patterns and it stores them in the weights' matrixes. In the following phase (i.e., *generalization* or *validation phase*), the ANN tests its ability and if it has been able to learn well the lesson, we can generalize results on similar out-of-sample data.

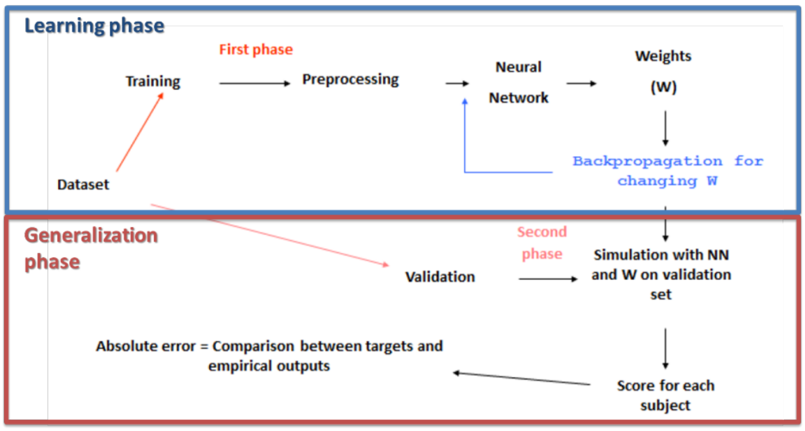


Figure 33. Training, Validation, Learning and Generalization.

The rule of subdividing the initial sample in the proportion of 2/3 for training and 1/3 for validation can be changed, but it is necessary that the training set is representative of the reality investigated. For learning well, ANN needs large amount of data, and, at the same time, they should be pre-processed for deleting possible outliers.

However, looking at complex models, it is possible to use an automated algorithm for feature selection. These algorithms work assigning even more small weights to non-representative features, until these last are eliminated from the model.

In section 7.1, good-practices of pre-processing activities are suggested.

It is then clear that the decisive phase is that of training, because is here that ANN learns and it has to learn well. We call **back-propagation algorithm** the automated process applied for updating the weights, and in next section, we will deepen this procedure.

4.4 The training process

Until now, we have only said that ANN learns from data and stores information into weights' matrixes; however, we have not investigated how ANN updates weights, then, now, we will see the idea under the back-propagation process.

At the beginning, weights are randomly initialized and adjusted step-by-step until the results (outputs) are satisfying (i.e., there is no difference between targets and outputs, or the difference is very little). This process assumes that the ANN has "to repeat the lesson" more times before arriving to obtain correct weights.

In addition, the reader must know that the rules presented here are not the only possible ones, but they are the most frequently used and the working mechanism is always the same.

4.4.1 The Perceptron learning rule

Let us consider a simple neuron with binary inputs and a binary target at time t. Weights are defined by $w_{ij}(t)$ and the basic idea is updating weights with a small amount $\Delta w_{ij}(t)$. In this manner, at time (t+1) we obtain:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t).$$

Suppose that the target output of unit j is $targ_j$ while the actual output is y_j. The activation function is the hard-limit, then:

$$y_j = hardlim(\sum x_i \cdot w_{ij}) = \begin{cases} 1 \; if \; u_k \geq 0 \\ 0 \; if \; u_k < 0 \end{cases}$$

Notice that the target is the correct output, while the actual output (y_j) is the response of the neuron.

Now, we are ready to "attend school" our neuron, and to observe its learning. A solution for learning well is to update weights with small changes, following the "if… then" rule:

- *If* y_j is equal to $targ_j$ *then* we can do nothing because ($targ_j - y_j$) is equal to 0 and w_{ij} are optimal weights
- *If* $y_j = 1$ and $targ_j = 0$, ($targ_j - y_j$) = ($0 - 1$) = -1 and $\sum x_i \cdot w_{ij}$ is too large, then we have to reduce it. Notice that x_i is a binary variable and we can act only on the weights:
  - when $x_i = 1$ we have to decrease w_{ij}: $w_{ij} - \eta(targ_j - y_j)\, x_i = w_{ij} - \eta(-1)x_i = w_{ij} + \eta x_i$;
  - when $x_i = 0$, $x_i \cdot w_{ij} = 0$, so $w_{ij} - \eta = w_{ij} - \eta x_i = w_{ij} - 0 = w_{ij}$;
    then $w_{ij} \rightarrow w_{ij} - \eta x_i$

Reasoning: why $\sum x_i \cdot w_{ij}$ is too large? The response of the neuron y_j is the result of the hard-limit function. If y_j is equal to 1, this means that $\sum x_i \cdot w_{ij}$ is higher than 0, so it is necessary to diminish it. In which manner? How much we have to diminish the value? The input variable x_i is done, then we can act only on the weights. We start defining a "rate" η in order to decrease iteratively the values of weights and to find those w_{ij} such that the response of neuron is equal to the corresponding target ($targ_j - y_j = 0$).

- *If* $y_j = 0$ and $targ_j = 1$, $targ_j - y_j = +1 - 0 = +1$ and $\sum x_i \cdot w_{ij}$ is too small, then we have to increase it. Even in this case, notice that x_i is a binary variable and we can act only on the weights:
  - when $x_i = 1$ we have to increase w_{ij}: $w_{ij} + \eta(targ_j - y_j)\, x_i = w_{ij} + \eta(+1)x_i = w_{ij} + \eta x_i$;
  - when $x_i = 0$, $x_i \cdot w_{ij} = 0$, so $w_{ij} + \eta = w_{ij} + \eta x_i = w_{ij} + 0 = w_{ij}$;

    then $w_{ij} \rightarrow w_{ij} + \eta x_i$

    Reasoning: why $\sum x_i \cdot w_{ij}$ is too small? The response of the neuron y_j is the result of the hard-limit function. If y_j is equal to 0, this means that $\sum x_i \cdot w_{ij}$ is lower than 0, so it is necessary to increase it. In which manner? How much we have to increase the value? The input variable x_i is done, then we can act only on the weights. We start defining a "rate" η in order to increase iteratively the values of weights and to find those w_{ij} such that the response of neuron is equal to the corresponding target ($targ_j - y_j = 0$).
- → The solution is

$$w^*_{ij} \rightarrow w_{ij} + \eta \cdot (targ_j - y_j) \cdot x_i;$$

$$\boxed{\Delta w_{ij} = \eta \cdot (targ_j - y_j) \cdot x_i \rightarrow \textit{Perceptron Learning Rule}}$$

The positive parameter η is called the *learning rate* or step size and it defines how smoothly we shift the decision boundaries. The *Perceptron Learning Rule* needs to be applied repeatedly and one pass through all weights for the training sample is called *one epoch of training*.

Clearly, the main goal of the presented algorithm is to minimize the difference between target and output, so the rule stops when Δw_{ij} is as closer to 0 as possible.

There are other learning rules, as for example the Hebbian l. r. that was the first one proposed for solving the Perceptron neuron.

The Learning rule is a method applied repeatedly to the ANN with the aim to improve the performance of network.

The *Hebbian learning rule* specifies how much the weight of the connection between two units should be increased or decreased in proportion to the product of their activation. On the base of the rule "Cells that fire together, wire together", Hebb (1946) defined that if two neurons had the same output, the weight of the connection would increase (increase of weight), otherwise the weight decreases. This rule has been implemented in the first Perceptron proposed by Rumelhart et al. (1986, 1987), but this learning rule requires the orthogonality of data, and this is a big limit to their extensive applications.

4.4.2 The Error Back-propagation and gradient descent algorithm

The process used by the neuron or network for updating weights is called back-propagation algorithm, and obviously, increasing the complexity of network, also the mathematical formalization increases. However, the idea is the same seen for the learning rule, but here, the inputs can be non-binary, the network is multi-layered, and the activation function is not necessary a hard-limit.

The goal of the back-propagation is always updating weights for minimizing the errors and for obtaining that the response of the network corresponds to the presented target.

Even in this case, the process is "adaptive" because weights are updated step-by-step.

Let us consider an ANN with two layers (i.e., one hidden layer and one output layer) where $z = [z_1, z_2, \cdots, z_s]$ is the output vector of a two-layer Feed-Forward Neural Network (i.e., FFNN), and $x = [x_1, x_2, \cdots, x_d]$ is the input vector. The target (i.e., the real output) is $t = [t_1, t_2, \cdots, t_s]$ and t_i can be equal to +1 or -1. The FFNN is presented in Figure 34.

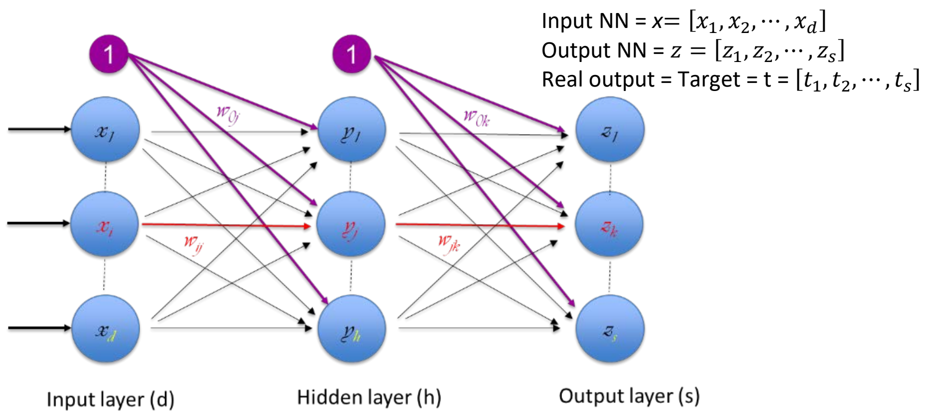

Input NN = $x = [x_1, x_2, \cdots, x_d]$
Output NN = $z = [z_1, z_2, \cdots, z_s]$
Real output = Target = t = $[t_1, t_2, \cdots, t_s]$
1
1
x_1
x_i
x_d
y_1
y_j
y_h
z_1
z_k
z_s
w_{0j}
w_{ij}
w_{0k}
w_{jk}
Input layer (d)
Hidden layer (h)
Output layer (s)

Figure 34. Two-layer Feed-Forward Neural Network (i.e., one hidden and one output layer).

The output z_k of the network can be found solving the following equation:

$$z_k = f\left(\sum_{j=1}^{h} w_{jk} y_j + w_{0k}\right) = f_2\left(\sum_{j=1}^{h} w_{jk} \cdot f_1\left(\sum_{i=1}^{d} w_{ij} x_i + w_{0j}\right) + w_{0k}\right)$$

The equation shows that the output depends strictly from all weights, and from activation functions. Notice that in this architecture there are two transfer functions: one from input to hidden layer (i.e., f_1), and one from this last and the output layer (i.e., f_2). In addition, these functions may differ from each other.

Starting from this consideration, we can define the final error of network as a function of inputs and weights. This function is a **loss function**, and it is defined as[6]:

$$J(\boldsymbol{w}, \boldsymbol{x}) \equiv \frac{1}{2} \sum_{c=1}^{s} (t_c - z_c)^2$$

The error $J(\boldsymbol{w})$ on the entire training set is the average of $J(\boldsymbol{w}, \boldsymbol{x})$ on all patterns $\boldsymbol{x}$ belonging to the training set.

The problem to solve is always the same: how to reduce $J(\boldsymbol{w})$?

We can reduce it modifying the weights (w) in the opposite direction to the «*gradient j*».

[6] This is not the only loss function possible. Indeed, the Sum Squared of Errors (SSE) is often used, or the Mean Squared Error (MSE) too. See section 4.5 for details.

The gradient indicates the direction of greatest growth of a function (of several variables), then moving in the opposite direction we can reduce (to the maximum) the error.

When the minimization of the error occurs through steps in opposite direction to the gradient, the back-propagation algorithm is also referred to as **gradient descent**.

When in ANN the back-propagation algorithm is taken into consideration, in general, it refers to gradient descent or to the "Stochastic Gradient Descent" (i.e., SGD) algorithm that updates weights subdividing randomly the input patterns of the training set. However, many other back-propagation algorithms exist, as, for example, the Levenberg-Marquardt algorithm, the Bayesian Regularization, and so on[7]. In these cases, the habit is calling directly the name of the optimization algorithm. In general, the difference among algorithms is based on the mathematics used for minimizing the errors of the ANN.

What does "gradient descent" mean?

Let us consider that we want to reach the top of a mountain as quickly as possible.

The most efficient solution is to remount the mountain following the *line of maximum slope*. This means that we have to measure iteratively the slope of the mountain in order to know which path is the quickest.

Technically, if we are at the valley bottom and we want to reach the top of the mountain, we can use the method of the *gradient ascent*; conversely, if we are on the top of the mountain and we want to reach the valley bottom (mathematically, the global minimum), we can adopt the algorithm of the *gradient descent*.

Even if the mathematics is more complex, the problem is always the same: finding a procedure for updating the weights and minimizing the error.

4.4.2.1 w_{jk} hidden-output update

We start calculating the partial derivative of function J with respect to w_{jk} that are weights between hidden and output layers:

$$\frac{\partial J}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}}\left(\frac{1}{2}\sum_{c=1}^{s}(t_c - z_c)^2\right) = (t_k - z_k)\frac{\partial(-z_k)}{\partial w_{ij}} = \cdots = -(t_k - z_k)f_2'(net)y_j$$

Notice that only z_c depends from w_{jk}.

Fixing $\delta_k = (t_k - z_k)f_2'(net)$, then the gradient is

$$\frac{\partial J}{\partial w_{jk}} = -\delta_k y_j$$

and we can reduce the weights in the opposite direction to the gradient:

$$w_{jk} = w_{jk} + \eta\delta_k y_j$$

where η is exactly the *learning rate*.

[7] See for recent advancements in learning rule Farizawani et al. (2020) and Shrestha & Mahmood (2019).

4.4.2.2 w_{ij} input-hidden update

We start calculating the partial derivative of function J with respect to w_{ij} that are weights between input and hidden layers:

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\frac{1}{2}\sum_{c=1}^{s}(t_c - z_c)^2\right) = \cdots = -\sum_{c=1}^{s}\delta_c w_{jc} f'(net_j)x_i = -x_i f_1'(net_j)\sum_{c=1}^{s} w_{jc}\delta_c$$

Notice that only net_j depends from w_{ij}.

Fixing $\delta_j = f_1'(net_j)\sum_{c=1}^{s} w_{jc}\delta_c$, then the gradient is:

$$\frac{\partial J}{\partial w_{ij}} = -\delta_j x_i$$

and we can reduce the weights in the opposite direction to the gradient:

$$w_{ij} = w_{ij} + \eta\delta_j x_i$$

where η is exactly the *learning rate.*

4.4.2.3 Incremental vs Batch training

Another point to take into consideration is when the weights are updated.

Literature suggests two possibilities:

1. *The incremental training* (or sequential or online): it starts setting up the network with zero initial weights, biases, and learning. The weights are updated as each input is presented.
2. The *batch training* (or offline): it provides that weights and biases are only updated after all the inputs and targets are presented. In this case, remember that one pass through all weights for the whole training set is called “epoch of training”.

In general, the incremental training is used for dynamic networks, whereas the batch training is more suitable for static ones.

4.4.3 When stopping the training process? Overfitting and Regularization

One of the most sensitive problem of the learning process is when ANN became very able to recognize the training sample, but ANN fails when new subjects/elements are presented. In this case, the ANN is not able to generalize, and this is a very big problem.

This issue is called **Overfitting** and it happens when in training data there is a noise, for the limited size of training sample, and for the complexity of classifiers (Ying, 2019).

One of the simplest methods for getting over the overfitting is the **Early Stopping Regularization**. As shown in figure 35, during the learning on training set, the error (E on the ordinate axis) tends to 0 because the network is learning better. We can observe that the curve of errors of validation set decreases until a minimum and after it starts increasing. The ANN stops the learning at the epoch of minimum error obtained for the validation set.

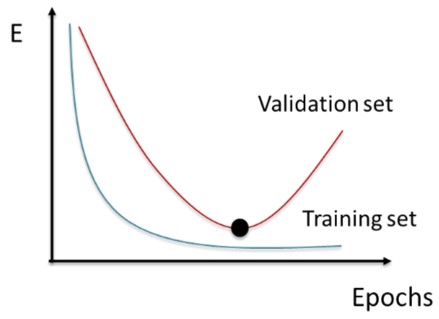


Figure 35. Early stopping regularization.

Which is the meaning of this rule? The increase of errors on validation set means that the ANN is not able to recognize well the unknown data (because they are not in the training set). For this reason, stopping the training phase when the error on validation is the minimum allows to obtain probably an error, but that is the minimum possible.

4.4.4 Back-propagation algorithm: global or local minima?

The back-propagation and the gradient descent show that errors weights are strictly linked. Thanks to this algorithm, weights are updated with the aim to decrease the error until 0, and thanks to the multi-layered architecture of the ANN, weights for each neuron are adjusted on the base of other neurons. However, increasing the complexity of network, also the probability to find a local and not global minimum exists. Following Domingos (2015), let us consider a simple network with only one weight, where the relation is expressed by the figure 36.

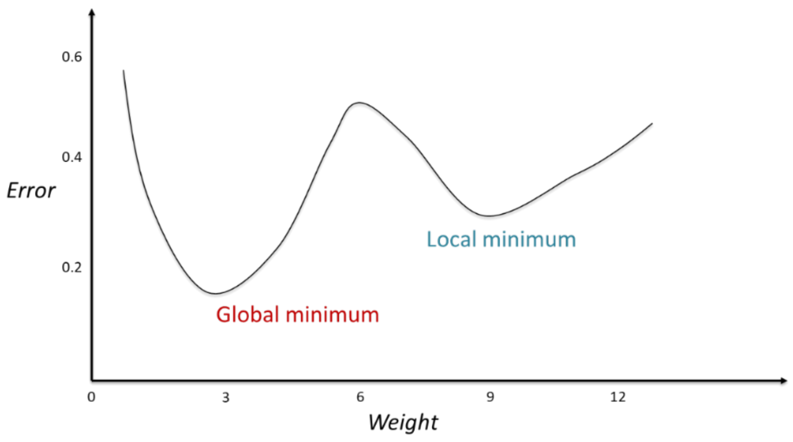


Figure 36. Weight and error, an example.

The curve shows its **global minimum** when weight is equal to 3 that represents the optimal weight. However, this is true only if weight has been initialized with a value lower than 3. What happens if weight is initialized to 7? In this case, the optimal weight is 9 that corresponds to a **local minimum**. This could be a significant problem, because back-propagation, moreover when the complexity of architecture increases, can suggest fake results minimizing the error on local and not global minimum. The weight remains as entrapped in the hollow of the curve, without distinguishing the type of minimum, and this can lead to learning errors. However, even if this is a weakness of the algorithm, today is shown that the local minimum is, all the same, a good result, and often it is preferable. Increasing the number of layers, the weight remains entrapped in a local minimum only if it happens in all layers, and this is an improbable possibility. As suggested by Domingos (2015), also local minima allow to obtain satisfying results.

4.5 The Performance of *Artificial Neural Networks*

The performance of ANN is prevalently calculated through the Sum Squared Error (SSE) that is measured as the sum of squared differences between targets and ANN outputs:

Sum Squared Error (SSE) =

$$= \sum_{i=1}^{N} (e_i)^2 = \sum_{i=1}^{N} (t_i - y_i)^2$$

However, the average of SSE can be used. In this case, we adopt the Mean Squared Error (MSE) that computes the mean squared error loss between target values and network predictions:

Mean Squared Error (MSE) =

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

A similar measure is the Root Mean Squared Error (RMSE) that is nothing else than the root of the MSE:

Root Mean Squared Error (RMSE) =

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

When the ANN is used for classification problem, it can be also convenient to evaluate the classification power through the Receiver Operating Characteristic curve (ROC) and the corresponding Area Under the Curve (AUC). However, this methodology can be applied when the output is binary (or if it is possible to transform outputs in dichotomous results).

The ROC is the graphical representation of the true positive rate (sensitivity) on the ordinate axis and false positive rate (1-specificity) on the abscissa one.

For understanding the sensitivity and specificity, let us consider the following confusion matrix in table 3. Columns represent the real output (the target), whereas, the rows represent the empirical outputs provided by the model.

The *sensitivity* is defined as:

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

and a high value means that an element positive in the reality will never be classified as negative by the model.

The *specificity* is defined as:

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

and a high value means that an element negative in the reality will never be classified as positive by the model.

	Reality	
Model	**P**	**N**
+	*True Positive (TP)* An element positive in the reality that is classified positive by the model	*False Positive (FP)* An element negative in the reality that is classified positive by the model (Type I error)
-	*False Negative (FN)* An element positive in the reality that is classified negative by the model (Type II error)	*True Negative (TN)* An element negative in the reality that is classified negative by the model

Table 3. Confusion Matrix.

The coronavirus disease 2019 (COVID-19), declared global pandemic on 11 March 2020 by the World Health Organization (i.e., WHO), provides us the opportunity to make an example of the meaning of these indicators. Consulting the Italian National Institute of Health website (i.e., Istituto Superiore della Sanità)[8], we read that tests carried out on the nasal antigenic swab provide the following results relating respectively to sensitivity and specificity: 70-86% and 95-97%. Results on sensitivity suggest that the probability that a patient really sick is classified as healthy ranges between 14% and 30% (1-sensitivity). On the contrary, results on specificity suggest that the probability that a patient really healthy is classified as sick ranges between 3% and 5%. So, if the result of an antigenic swabs is positive, we can be confident to have the COVID19 (i.e., the specificity is high); otherwise, if the result of the swab is negative, we cannot be completely sure that the result is correct, and that the negative result is in reality a false negative (i.e., the sensitivity does not present comfortable values: from 14% to 30% is a big range).

An example of the ROC curve is presented in figure 37 where are compared the performance of three hypothetical models. The AUC is the area under the ROC curve and the model fitting well is that with the highest AUC value. In this specific case, the Model 1 (blue curve) is the best one.

The AUC represents the probability of correct prediction based on probabilities estimated by the model. The rule of thumb indicates that:

- when AUC = 0.50 the model has no discriminatory capacity.
- when AUC varies between 0.70 and 0.80 the model has an acceptable discriminator power.
- when AUC ranges between 0.80 and 0.90 the model discriminated in an excellent way.
- when AUC> 0.90 the model has a "super-excellent" discrimination power. This result is very difficult to obtain in real applications.

[8] Data are extracted from https://www.iss.it/primo-piano/-/asset_publisher/3f4alMwzN1Z7/content/diagnosticare-covid-19-gli-strumenti-a-disposizione.-il-punto-dell-iss (last visit January 15 2022)

Finally, using these concepts, the *Accuracy* of the model can be computed as the ratio between the number of correct classifications and the sample size:

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

The ROC curve, as well as the AUC, and the sensitivity and specificity rates are used in many fields where the evaluation of classification or prediction is required. They are very simple indexes, but powerful in terms of meaning. In addition, similar consideration can be done for MSE or RMSE, that are always used, for example, in regression analysis.

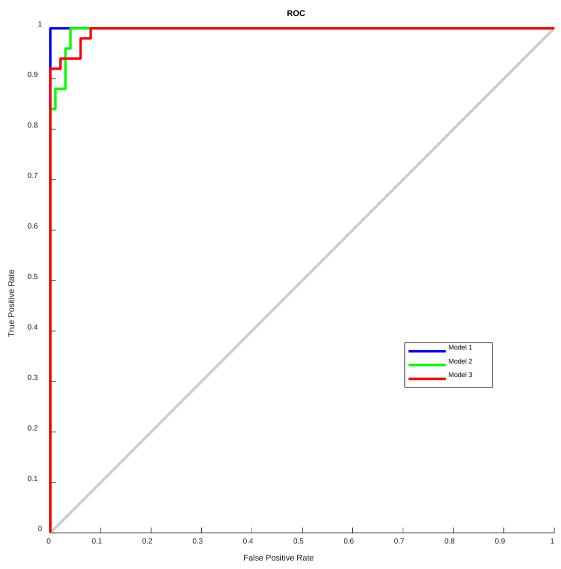


Figure 37. ROC curves.

Finally, starting from table 3, in pattern recognition and classification, two concepts linked to confusion matrix are often used:

- *Precision* that is the Positive Predicted Value (PPV) measured as:
$$PPV = \frac{True\ Positive}{True\ Positive + False\ Positive}$$
This index represents how many instances the model classifies correctly.
- *Recall* that is the sensitivity presented before. It represents the robustness of the model and it indicates how many instances are missing.

The harmonic mean of precision and recall give us the **F$_1$ score (**also known as **F-score** or Sørensen-Dice coefficient or Dice Similarity Coefficient, DSC), that measures the test accuracy as follows:

$$\boldsymbol{F_1} = 2 \cdot \frac{Precision\ \cdot Recall}{Precision + Recall} = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

It can assume values between 0 and 1. It assumes a value of 0 only if at least one of the two is equal to 0, while it assumes a value of 1 if both precision and recovery are valid. Higher values indicate higher precision and recall of the model.

A general formulation of F-score is the **F$_\beta$ score** where β represents how much more relevant recall is than precision. In details, it is calculated as follows:

$$\boldsymbol{F_\beta} = (1+\beta^2)\cdot\frac{Precision\ \cdot Recall}{\beta^2\cdot Precision + Recall} = \frac{(1+\beta^2)\cdot TP}{(1+\beta^2)\cdot TP + FP + \beta^2\cdot FN}$$

Values of β depend from the importance that the user assigns to precision and recall. For instance, if β=2 the user considers the importance of recall twice that of precision. Notice that if β=1, the F$_\beta$ score is equal to the F$_1$ score.

Sum up, statistics offers many methods for evaluating the performance of a model, and the choice depends from the aims and the hypotheses of the analysis. For instance, if we are physicians, and we are testing a model helping us to choose for patients' hospitalization, we are interested in a model able to maximize the sensitivity, so that a model able to correctly recognize if a patient needs to be hospitalized. However, a model maximizing the true positives can present low specificity, so that, it is possible that the model advises the physician to hospitalize a patient when it is not necessary. Obviously, from the financial point of view, this is not a good result for the manager of the hospital because a non-necessary hospitalization is a cost, but the physician is sure to hospitalize all those need cares.

This evidence means that the goodness of results measured through these indexes cannot ignore the hypotheses of the model. An interesting case study on the trade-off between the physician and the manager aims is presented in Casagranda et al. (2016) and Falavigna et al. (2019). Other interesting applications and discussion on ANN performance can be found in Costantino et al. (2017), Falavigna et al. (2019), and Ippoliti et al. (2021).

4.5.1 A "sticky note" on the accuracy: bias and variance

A brief note is necessary about the accuracy in *Machine Learning*, because there are two concepts covering big relevance that are *bias* and *variance*. These two measures are relevant moreover when on testing sample the performance is not satisfying. Let us consider four friends (Paul, John, George and Ringo) going to the pub for a beer, playing music, and playing darts. Paul have an optimal aim and all his darts find its mark. In figure 38, Paul is in the lower left corner because both bias and the variance are low: 6 darts, all in the center of the aim. John is very short-sighted and he has difficulty in focusing distant objects, then his results are really unsatisfying. In figure 38, John is in the top right corner, his performance presents high bias and high variance: only one dart is near to the target, and the other five are very far. If we would like to find a position on the board representing the mean of the 6 darts, we will find even the mean will be very distant from the target. We can then conclude that for John bias and variance are both very high.

George is presented in the lower right corner of figure 38. We know nothing about his sight defects, but playing darts certainly is not his favorite activity; indeed, his darts are all close to each other, but well away from the center of the target: his bias is high, while the variance is low.

Finally, Ringo is the most playful of his friends and has played darts without commitment. From figure 38, all darts are far from the center of the target, but trying to do a mean of his shots, the result is not far from the center. We can then conclude that for Ringo the bias is low and the variance is high.

This simple example (inspired by Domingos, 2015) shows that for evaluating the performance of a learner it is necessary to consider the bias (real mistake), and the variance (the distance between results and targets).

For evaluating bias and variance of a model, it is necessary to compare its results on random subsample of the training set. If the model continues to do the same errors, the learner presents bias problem; if errors are not regular, the model presents variance problem and it is necessary to add data.

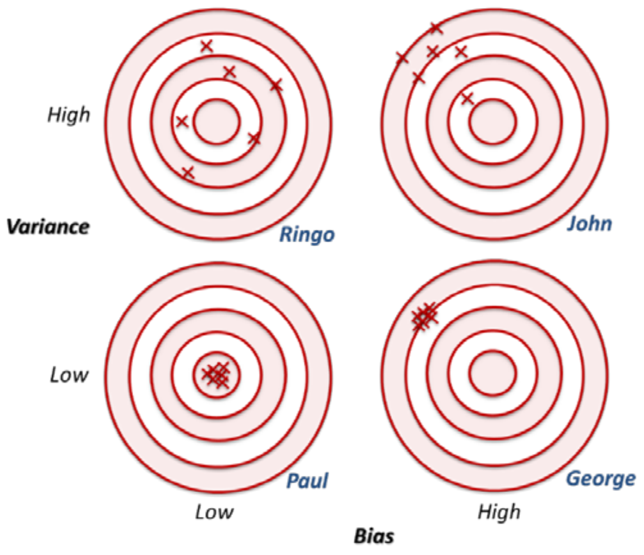


Figure 38. Bias and variance for target shooting of Paul, John, George, and Ringo.

Sum up, we realize that the "bias-variance dilemma" is a crucial issue in the evaluation of ML/DL model. In general:

- The *bias* happens when erroneous assumptions are made in the learning algorithm. If the bias is high, the algorithm suffers from *underfitting* problem: the algorithm is not able to identify fundamental relations between inputs and targets. In figure 38, the result of George represents the problem of underfitting;
- The *variance* depends from fluctuation in the training set. If the variance is high, the algorithm suffers from *overfitting*: the algorithm is confused by random noise in data of training sample. In figure 38, the result of Ringo represents overfitting problem.

Bias: difference between the prediction of the algorithm and the correct value to predict.
Variance: variability of model prediction for a given data point.
Underfitting: a model is not able to capture the underlying pattern of the data.
Overfitting the model captures the noise along with the underlying pattern in data. In this case, the problem is that data are very noisy.

Figure 39 presents what happens when the algorithm suffers from overfitting (panel (a)), underfitting (panel (b)), and optimal performance (panel (c)).

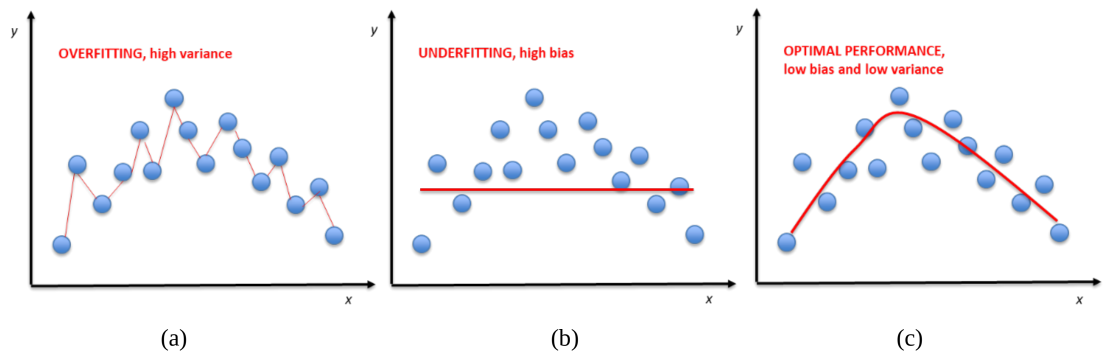


Figure 39. Overfitting, Underfitting and Optimal performance of a ML/DL algorithm (Adapted from Singh, 2018).

4.6 An exercise with a *Shallow Neural Network*: performance and regularization

Now, we are ready to understand the operational process of an *Artificial Neural Network* and the evaluation of results. In this section, we analyse a database in order to assign to each element a value, with the aim to evaluate the performance of the model and the regularization algorithm.

Let us start using the bodyfat_dataset provided by Matlab R2019b representing a sample of 252 people with the following 13 attributes: Age (years); Weight (lbs); Height (inches); Neck circumference (cm); Chest circumference (cm); Abdomen 2 circumference (cm); Hip circumference (cm); Thigh circumference (cm); Knee circumference (cm); Ankle circumference (cm); Biceps (extended) circumference (cm); Forearm circumference (cm); Wrist circumference (cm).

For each of 252 people, we know the Body Fat[9] and we want to define an ANN able to assign a Body Fat to people out-of-sample (i.e., people not comprised in the training sample) starting from these 13 attributes.

Then, we build a simple two-layer Feed-Forward Neural Network (i.e., FFNN) with 1 hidden layer (with 20 neurons) and 1 neuron in the output one (figure 40).

Linear function:

$$purelin(u_k) = au_k + b$$

Hyperbolic tangent sigmoid function:

$$tanh(u_k) = \frac{e^{-u_k} - e^{-u_k}}{e^{-u_k} + e^{-u_k}}$$

The sample has been subdivided into 2 subsamples: the 70% of observations represents the training set, the remaining 30% represents the validation set. The activation function between the input layer and the hidden one is a hyperbolic tangent sigmoid function, while from the hidden to the output layer has been set a linear function. The choice of a linear function is due to the fact that the Body Fat is a continuous number. The weight matrix between input and hidden layer is a 13x20 because we have 13 attributes and 20 neurons in the hidden layer, while from this last and the output layer weights are collected in a 20x1 vector.

[9] The Body Fat is the portion of the human body that consists of fat (Medical Dictionary for the Health Professions and Nursing, 2012, from https://medical-dictionary.thefreedictionary.com/body+fat , last visit January 16 2022).

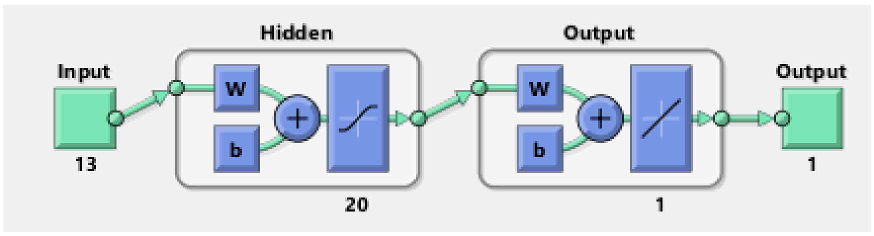


Figure 40. FFNN with 1 hidden layer with 20 neurons.

The loss function adopted is the Mean Squared Error (MSE).

Remember that one pass through all weights for the whole training set is called epoch of training.

When should the algorithm stop? The minimum value of MSE on validation set has been reached at epoch 3, as shown in figure 41. The back-propagation algorithm stops when the performance on validation set starts to increase, in this case this happened at epoch 3 (following the early-stopping criterion discussed in section 4.4.3) and the corresponding MSE (on the validation) is 16.67.

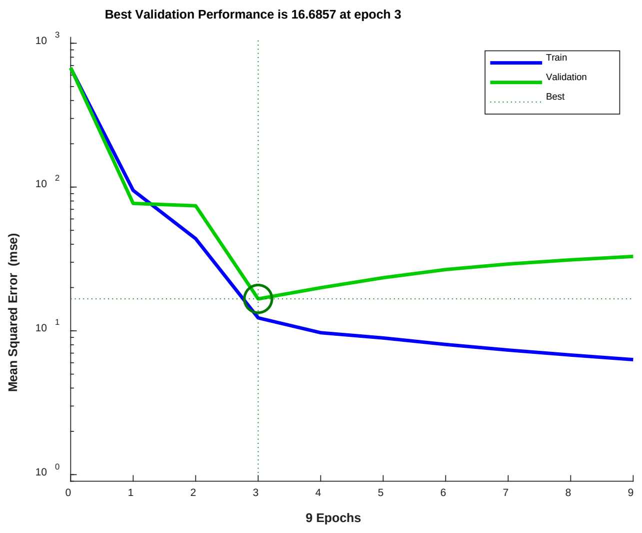


Figure 41. Early stopping for FFNN.

Another measure for evaluating the performance of ANN is to do a regression between the outputs of the NN and the targets. If the training is perfect, the network outputs and the targets are equal, but this perfect relationship happens very rarely in practice.

Figure 42 presents regressions for training and validation. The dashed line shows the perfect result (target-output=0), the solid line draws the best fit linear regression line between outputs and targets (in blue for the training and in green for the validation).

R value is the classic R-squared calculated in statistics and represents the goodness-of-fit of the regression. A value close to 1 indicates that there is an exact linear relationship between outputs and targets; a value close to 0 defines worse performance of ANN.

In general, obtained value of R squared on validation set is not an unsatisfying result (i.e., 0.85), but we may want to obtain a better result.

First of all, we can try to re-run the FFNN because each time the network is initialized, the parameters (for instance, initial values of weights) are different and results could change.

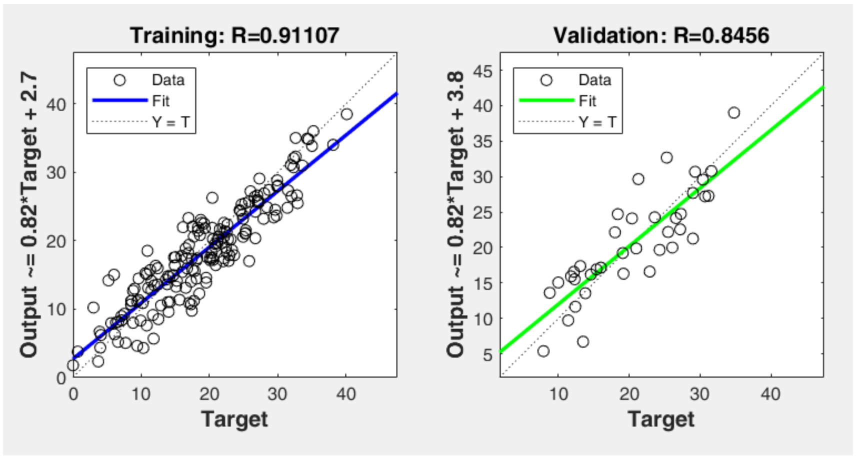


Figure 42. Linear regression for training and validation sets.

Figure 43 and figure 44 show respectively the performance on training and validation, and the regression between output and target for samples. The best value of MSE reached is 46.78 at epoch 3, higher than the previous result and also the R-squared of regression is lower in both cases (i.e., for training and validation sets) if compared with those obtained in previous model.

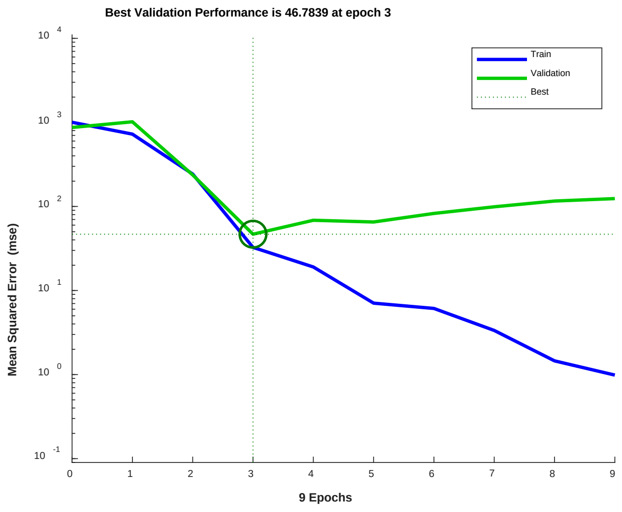


Figure 43. Early stopping for re-initialized FFNN.

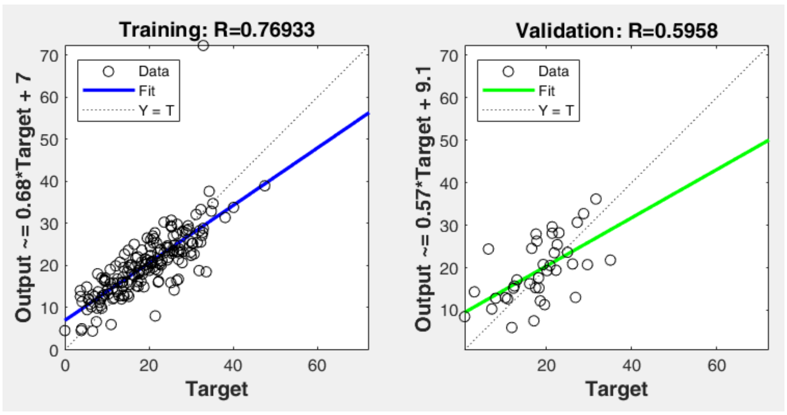


Figure 44. Linear regression for training and validation sets (re-initialized FFNN).

If we tried to re-initialized newly the network, we would find different results. Then, we will re-initialize the network until we are not satisfied.

Another strategy could be changing the transfer function between the hidden to output layer in a logistic function. Results are very unsatisfying, indeed the lowest MSE is equal to 98.9 (figure 45). However, this is not a surprising result because the target does not vary between 0 and +1, as instead the logistic function does.

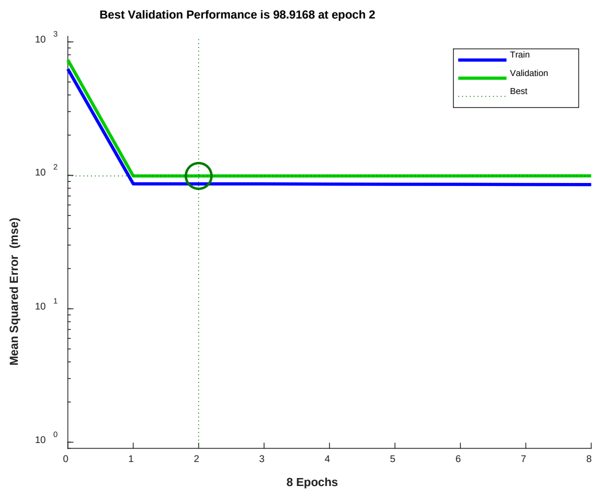


Figure 45. Early stopping for re-initialized FFNN with logistic function between hidden and output layer.

We can do other trials, adding layers, modifying the number of neurons, and it is possible that we will find better result, but we will not.

From the example, we have shown that many factors have to be considered when a ANN is run and that, when even one parameter changes, the ANN outputs also change.

There is not a rule for understanding when the best performance is reached, but with *Artificial Neural Networks*, and in general *Deep Learning* methods, it is necessary to find empirically the solution, trying and trying again.

In next paragraph, we will investigate the informative power of ANN’s parameters, and we will show why ANNs are used in a large variety of fields.

5 Galeotto was the algorithm and he who wrote it… Feature Selection with *Deep Learning*

In present section, we explain how we can exploit information in matrix weights through a simple example. From the one hand, we will show the power of ANN in prediction/classification, and from the other hand, we will introduce two simple ways to do feature extraction. [10]

5.1 "Magic Mirror on the wall, will Juliet tell me yes?"

Romeo and Juliet are friends and students: Romeo attends informatics and Juliet classical literacy. Romeo is secretly in love with Juliet and he would like to ask Juliet to go to the cinema one night, but he fears that her refusal.

Is there any way to predict Juliette's reaction?

Romeo is a very good student, but with girls he is very shy and then a bit clumsy. He decides to observe the habits and the behaviour of Juliet with the help of some friends and consulting the social networks.

For 7 non-consecutive days, each evening Romeo investigates about:

1. Did she have to study?
2. Were we in the weekend (i.e., Saturday or Sunday)?
3. Did she go out for a drink?
4. Did she go out for dinner?
5. Did she go to the cinema?
6. It was raining?

Romeo defines a table in which the rows correspond to the days and the columns to collected information (table 4). Value 0 means "No", whereas value 1 identifies "Yes": for instance, the day 1, it was raining, she had to study, it was not a weekend day, she went out for a drink, she did not go out to dinner, and finally she did not go to the cinema.

Day	Rain?	Study?	Weekend?	Drink?	Dinner?	Cinema?
1	1	1	0	1	0	0
2	0	0	0	0	0	0
3	0	0	0	1	0	1
4	1	1	1	1	0	0
5	0	0	1	0	1	0
6	0	1	0	1	0	0
7	1	1	1	1	1	1

Table 4. Observation of Juliet habits.

After a deeper study of the situation, Romeo is ready for asking Juliet to go together to the cinema. He checks that on Saturday it will not rain, Juliet should not go out either drinking or for dinner. However, she will have to study, and Romeo is doubtful.

[10] The example has been run under R version 4.0.3, package *neuralnet* version 1.44.2.

While he was about to send the message on WhatsApp to Juliet, he remembers the Perceptron just studied.

He then decides to build a neuron with a logistic activation function with the aim to predict if Juliet will say yes to his invitation. As loss function, Romeo adopts the Sum Squared Errors (SSE)=$\sum(target - output)^2$ and he decides to initialize randomly the weights.

Figure 46 presents the neuron built by Romeo using the data in table 4. The Perceptron has 5 input variables (i.e., Rain, Study, Weekend, Drink, Dinner), the bias, and 1 target (i.e., Cinema). After the training and the validation phases, the error of the network is very low (i.e., 0.036), then Romeo can be confident about the results.

Logistic function:

$$sigmoid(u_k) = \frac{1}{1 + e^{-u_k}}$$

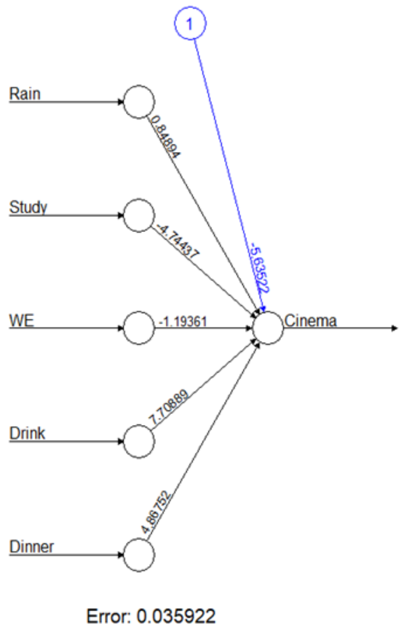


Figure 46. Perceptron for Juliet.

Now, as the Evil Queen of the Snow-White fairy tale, he interrogates the neuron as if it was the Magic Mirror: "Magic Mirror on the wall, will Juliet tell me yes?"

The input is the following vector:

Rain?	Study?	Weekend?	Drink?	Dinner?
0	1	1	0	0

and the response of the network (the output) is 0.00000864, a value really too low to be confident in a positive replay of Juliet.

Nevertheless, Romeo does not give up and he analyses the weights of neuron with the aim to define a winner strategy.

Five variable and 1 target-neuron, then the weights are collected in the following vector of five elements (table 5):

Variables	Weights
Rain	0.849
Study	-4.744
WE	-1.194
Drink	7.709
Dinner	4.868

Table 5. Weights for each variable (Perceptron with one target/output layer)

Looking at the signs of weights, indicating how each variable affects Juliet's decision, Romeo infers the following considerations:

- Juliet is certainly studious (high weight in absolute value, |4.744|), and if she has a lot to study, she will hardly come out (- sign).
- However, Juliet enjoys very much having fun and having a drink with friends, indeed the variable "Drink" presents the greater weight with positive sign (+7.709). The same consideration for the dinner, the sign of the weight is positive (+4.868).
- Juliet prefers going to the cinema not in the weekend (variable WE with negative sign, -1.194).
- The weather condition does not affect too much the decision of Juliet about the cinema (0.849).

Romeo is now convinced that he has to change his strategy, and now he wants to try inviting Juliet on Wednesday, when the weather will be sunny, she should not have to study and she should not go out drinking, however she will go out for dinner.

Now, the input vector is the following:

Rain?	Study?	Weekend?	Drink?	Dinner?
0	0	0	0	1

The neuron response (the output) is then 0.418. Not bad score for the strategy of Romeo, but he would be more confident, then Romeo changes and he will ask Juliet to go together to the cinema on Wednesday, when the weather will be sunny, she should not have to study, but she will go out both drinking and having dinner:

Rain?	Study?	Weekend?	Drink?	Dinner?
0	0	0	1	1

Now, the response of the "Magic Mirror" is 0.999, definitively recognizing the latter as the winning strategy.

However, Romeo is not completely convinced yet, and he wants to try interrogating an *Artificial Neural Network* (or so called "Shallow Neural Network") with 1 hidden layer with one neuron. The input matrix, transfer functions (i.e., logistic activation function is applied in both layers) and initialization weights remain the same than before and in figure 47 is presented the new model. Notice that even in this case, the network is trained and validated on input sample presented in table 4.

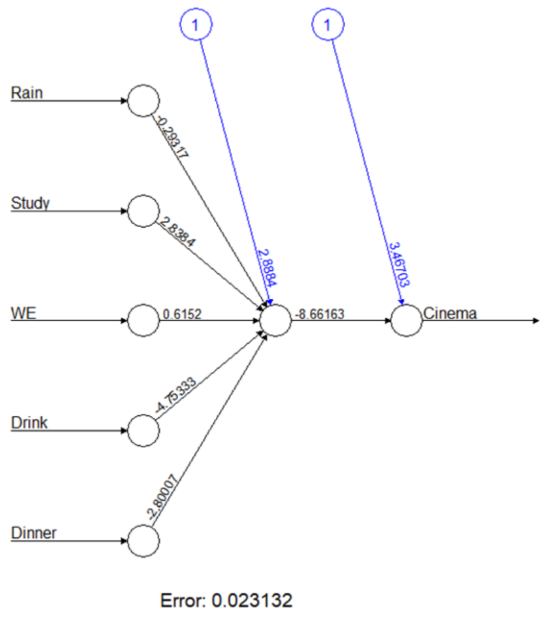


Figure 47. ANN with one hidden layer (1 neuron) – ANN1.

Now, the error decreases (0.023 instead of 0.036), then the model classifies better than the previous one.

Looking at the weights, we have a vector of weights 5x1 (5 variables in the input layer and 1 neuron in the hidden one) and a single weight between hidden and output layer. In order to find the weight for each variable, we need to do the matrix multiplication (table 6).

Weights.1L	Weights.2L
-0.293	
2.838	
0.615	
-4.753	
-2.800	
	-8.662

Variables	W.1L*W.2L
Rain	2.539
Study	-24.585
WE	-5.329
Drink	41.172
Dinner	24.253

Table 6. Weights for each variable (ANN with one hidden layer and one neuron, ANN1).

The size of weights is higher, but considerations are not changed. This means that probably the responses of Perceptron on previous strategies are confirmed. Table 7 shows results that effectively are in line with those obtained with one neuron.

Strategy	Rain?	Study?	Weekend?	Drink?	Dinner?	ANN1 output
1	0	1	1	0	0	0.006
2	0	0	0	0	1	0.258
3	0	0	0	1	1	0.968

Table 7. ANN responses to three Romeo's strategies and ANN with 1 hidden layer and 1 neuron (ANN1).

Romeo is definitively enjoying himself in defining *Artificial Neural Networks* and he decides to increase the number of neurons in the hidden layer. The new architecture of the network is presented in figure 48.

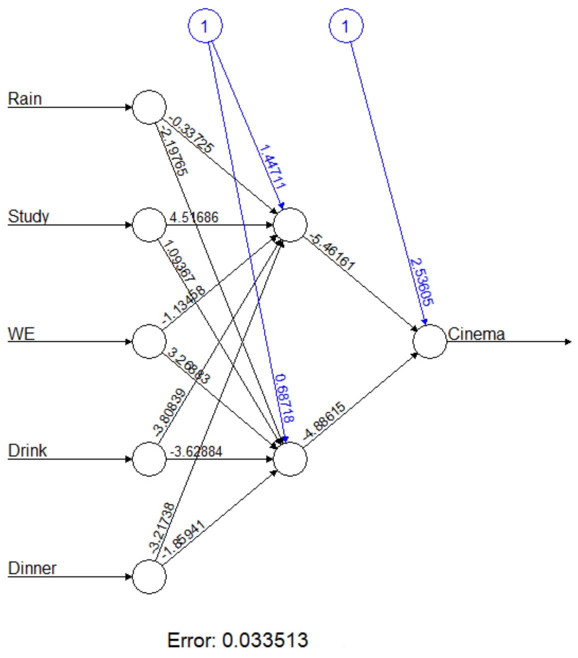


Figure 48. ANN with one hidden layer (2 neurons) – ANN2.

The error is increased, then the previous model (i.e., ANN1: 1 hidden layer with 1 neuron) outperforms this one (i.e., ANN2: 1 hidden layer with 2 neurons).

Looking at the weights, we can observe that each input is linked to each neuron in the hidden layer, then, we have a matrix of weights 5x2 (5 variables in the input layer and 2 neurons in the hidden one) and a vector of weights 2x1 between the hidden and output layer (i.e. 2 neurons in the hidden layer and 1 in the output one). In order to find the weight for each variable, we need to do the matrix multiplication (table 8).

Weights.1L		**Weights.2L**
Weights$_{1L.1}$	**Weights$_{1L.2}$**	
-0.337	-2.198	-5.462
4.517	1.094	-4.886
-1.135	3.269	
-3.808	-3.629	
-3.217	-1.859	

Variables	**W.1L*W.2L**
Rain	12.580
Study	-30.013
WE	-9.775
Drink	38.531
Dinner	26.657

Table 8. Weights for each variable (ANN with one hidden layer and two neurons, ANN2).

Even in this case, the size of weights is higher, but considerations do not change.

The responses of the interrogation of Romeo are then presented in following table (table 9):

Strategy	**Rain?**	**Study?**	**Weekend?**	**Drink?**	**Dinner?**	**ANN2 output**
1	0	1	1	0	0	0.000
2	0	0	0	0	1	0.642
3	0	0	0	1	1	0.922

Table 9. ANN responses to three Romeo’s strategies for the ANN with 1 hidden layer and 2 neurons (ANN2).

Romeo is now satisfied and on the strengths of previous considerations he sends a WhatsApp message to Juliet following the third strategy.

Unfortunately, we do not know if Juliet really accepted the Romeo invitation, but we can still make some considerations on weights.

Indeed, until now operations made on weights allow us to understand which input affects the more the response of the ANN and the "direction" of this influence. However, we are not able to compare the weights of input-variables with each other.

In the following section, we present the Garson index methodology, that allows to evaluate weight of each input variable in percentage terms.

5.2 The power of weights: the Garson index

In this section, we present a simple method for transforming ANN weights into percentage values. Literature suggests other methodologies, but Garson indexes are simple to calculate and powerful (Garson, 1991; Olden & Jackson, 2002). In literature, Garson indexes have been widely applied; see, for instance, Falavigna (2012), Casagranda et al. (2016), Ippoliti et al. (2021), Falavigna (2008c).

Let *i* input variables (where *i* = 1, ..., I), *j* hidden nodes (where *j* = 1, ..., J) and *k* output neurons (where *k* = 1, ..., K), matrixes are signed with capital letters and their elements with minuscule ones; Garson index is a vector made by:

$$G_{ik} = \frac{W^*_{ik}}{Z_k} \cdot 100$$

where:

$$W^*_{ik} = \frac{|W_{ij}|}{S_j} \cdot |W_{jk}|$$

$$S_j = \sum_{i=1}^{I} |w_{ij}|$$

$$Z_k = \sum_{i=1}^{I} w^*_{ik}$$

We show the calculus for the more complex model presented (i.e., the two-layer ANN, ANN2), for the other two architectures (i.e., the Perceptron and the ANN1) the procedure is the same:

Looking at table 8, let us start in calculating the absolute value of each element of the first weight matrix (W.1L) and the S_j values that are the sum of each column:

	|Weights.1L|	
	$\|$Weights$_{1L.1}\|$	**$\|$Weights$_{1L.2}\|$**
	0.337	2.198
	4.517	1.094
	1.135	3.269
	3.808	3.629
	3.217	1.859
S_j	13.014	12.049

Now, we calculate the first factor (i.e., $\frac{|w_{ij}|}{S_j}$) of matrix W_{ik}^* as follows:

	Weights.1L	
	|Weights$_{1L.1}$|	**|Weights$_{1L.2}$|**
	0.337/13.014	2.198/12.049
	4.517/13.014	1.094/12.049
	1.135/13.014	3.269/12.049
	3.808/13.014	3.629/12.049
	3.217/13.014	1.859/12.049
S$_j$	13.014	12.049

|Weights$_{1L.1}$|/*S$_1$*	**|Weights$_{1L.2}$|/*S$_2$***
0.026	0.182
0.347	0.091
0.087	0.271
0.293	0.301
0.247	0.154

We proceed multiplying the obtained matrix with the vector of weights between hidden and output layers (i.e., $|W_{jk}|$), after having calculating the absolute value of each element. The result is the following vector:

$\|W_{ij}\|/S_j$	
|Weights$_{1L.1}$|/*S$_1$*	**|Weights$_{1L.2}$|/*S$_2$***
0.026	0.182
0.347	0.091
0.087	0.271
0.293	0.301
0.247	0.154

$\|W_{jk}\|$ =|Weights.2L|
5.462
4.886

	W_{ik}^*
	1.033
	2.339
	1.802
	3.070
	2.104
Z_k	10.348

In addition, we have made the sum of column (i.e., Z_k). Finally, we are ready for calculating Garson indexes dividing each value with the result in the last row (i.e., $\frac{W_{ik}^*}{Z_k} \cdot 100$).

Garson Index	**Garson Index (%)**
0.100	10.0%
0.226	22.6%
0.174	17.4%
0.297	29.7%
0.203	20.3%

Now, we can calculate the Garson indexes to the three models defined by Romeo, results are presented in columns (2), (3), and (4) of the following table (table 10):

(1)	**(2)**	**(3)**	**(4)**	**(5)**
Variables	**Perceptron**	**ANN1**	**ANN2**	**Sign**
Rain	4.38%	2.59%	10.0%	+
Study	24.50%	25.12%	22.6%	-
WE	6.17%	5.44%	17.4%	-
Drink	39.81%	42.07%	29.7%	+
Dinner	25.14%	24.78%	20.3%	+

Table 10. Garson Indexes.

Results on Garson indexes confirm that the biggest role in defining the ANN network is played by the variable Drink, followed by Dinner and Study. However, the Garson index alone does not reveal if variables affect results in a positive or negative manner.

Concluding, the Garson indexes and the analysis on weights obtained in previous section (column (5) in blue in table 10) have to be studied together if we want investigating the determinants of the target.

6 Basic architectures of *Deep Learning*

In this section we present the main topologies of *Artificial Neural Networks*, that are the fundamentals of *Deep Learning* models.[11]

Thanks to the increasing computational power, the number of architectures proposed by researchers increased, together with their complexity. As said before, *Deep Learning* and, more in general, *Machine Learning* are more flexible than standard statistical and econometric techniques, then, the possibility to build hybrid methodologies is very high. In this manner, it is possible to exploit the advantages of all techniques and, at the same time, to cover their weaknesses.

The following section represents a short review of main ANN basic frameworks and it is not comprehensive of all existent models, but it recalls those well-known and applied in the majority of applications.

Before starting with the review, it is necessary to explain better two word-pairs regarding neural networks:

1. *Static* vs *Dynamic*: the difference is based on inputs. If time-series are used as inputs, the neural network is called “Dynamic”; otherwise it is “Static”. When the NN is dynamic, it is said “*with memory*”.
2. *Feed-forward* vs *Back-ward/Recurrent*: the difference is based on connections. If neural network presents connections travelling from left to right without going back, the ANN is called “Feed-forward”, otherwise it is “Back-ward/Recurrent”.

This means that we can find Static Feed-forward or Static Recurrent NN, and Dynamic Feed-forward or Dynamic Recurrent NN.

6.1 Cascade Forward Networks

This architecture is very similar to supervised feed-forward networks. The difference is that in this topology are included weight connections from input layers to each layer, and from each layer to successive ones. In figure 49 red and green arrows show the difference from the simple feed-forward neural network. Indeed, neurons of the input layer are connected not only with neurons of the first hidden layer but also with the second hidden layer and with the output one (red arrows).

[11] All examples presented in this section have been run with Matlab R2019b, Deep Learning Toolbox.

At the same time, also neurons of the first hidden layer are connected not only with the following layer, but also directly with the output one (green arrows).

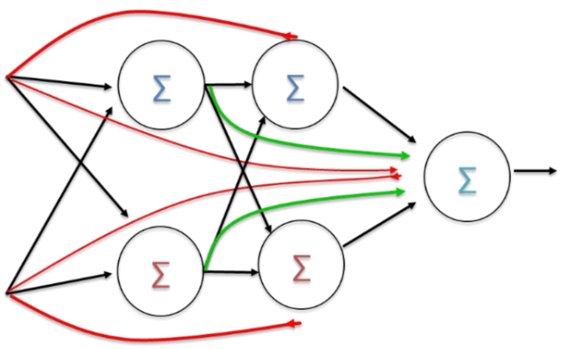


Figure 49. Cascade Forward Network topology.

In general, the Cascade Forward neural networks are used for any kind of mapping.

6.2 Radial Basis Function (RBF)

The Radial Basis Function is a supervised feed-forward neural network with only one hidden layer (i.e., two-layer NN: one hidden layer and one output layer), but it presents predefined activation functions.

Between the input layer and the hidden one, the ANN presents a gaussian transfer function (i.e., radial basis function); from the hidden to the output layer, the network presents a linear transfer function.

Let us consider a neuron k, it is equal to:

$$y_k(x) = \sum_{1=1}^{M} w_{kj}\phi_j(x) + w_{k0}$$

Radial basis function:

$$\phi(x) = radbas(x) = e^{-x^2}$$

Linear function:

$$purelin(x) = ax + b$$

However, the difference between RBF and a two-layer ANN is the argument of the function ϕ. Indeed, the x in this case is a measure of distance[12] between the input vector and vectors formed from the rows of the input weight matrix. Figure 50 shows the RBF topology:

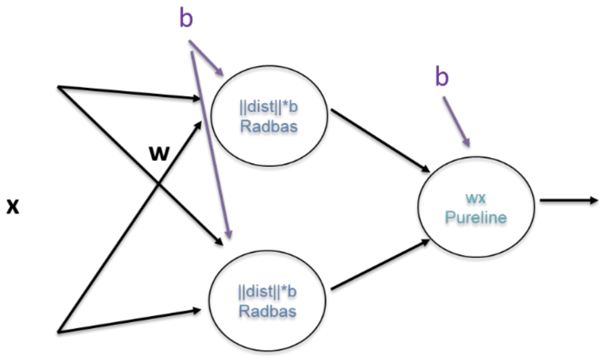


Figure 50. Radial Basis Function topology (RBF).

[12] In Appendix A, a short review of main distance measure is provided.

The ||dist|| can be measured with different distances, but the more frequently used is the Euclidean one. However, all distance measures, well known in clustering analysis, can be here used.

But… what is the difference in practice?

The answer is represented in figure 51 where pictures (a) and (b) show respectively how feed-forward neural network and radial basis function work. The former classifies observations through hyperplane intersections, the latter exploits distance measure for grouping observations in clusters.

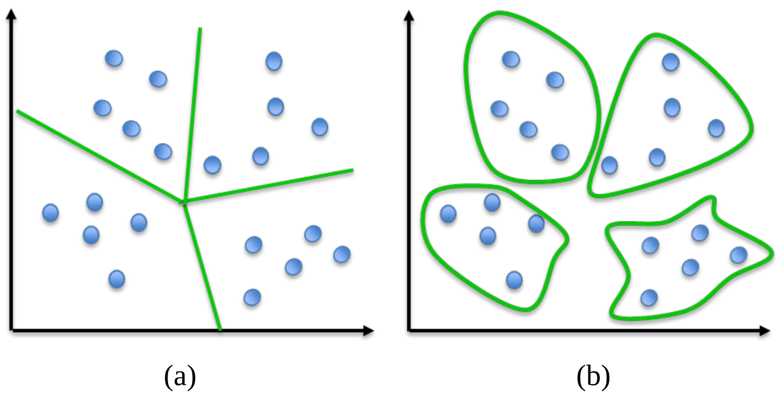


Figure 51. FFNN and RBF in comparison.

The RBF neural networks are used moreover for interpolations or in forecasting in time series; indeed, they are successfully applied in weather forecasting.

From the RBF family, two topologies of NN deserve attention: the Generalized Regression Neural Network (GRNN) for forecasting and Probabilistic Neural Network (PNN) for classification.

6.2.1 Generalized Regression Neural Network (GRNN)

The GRNNs are a specification of RBF where the transfer function between the hidden layer and the output one is slightly different from the linear one. This type of ANNs is used in forecasting and in general in function approximation.

Let us consider *n* as the dot product of a row of weight vector (between hidden and output layer) and the input vector from hidden layer (*xh*), all normalized by the sum of the elements of *xh*.

Linear function:

$$purelin(n) = an + b$$

Figure 52 shows the topology of GRNN where the argument of the linear transfer function between hidden and output layer is the *n* measure.

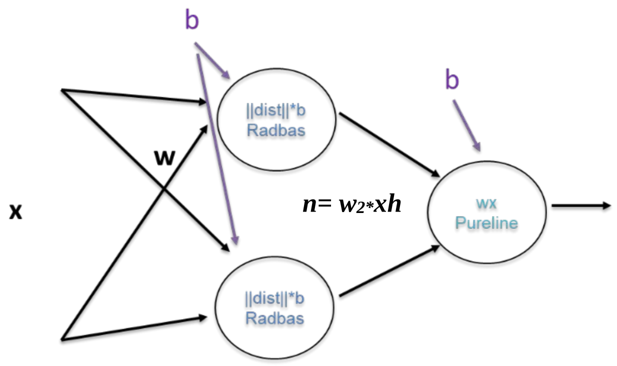


Figure 52. Generalized Regression Neural Network topology (GRNN).

6.2.2 Probabilistic Neural Network (PNN)

The PNN, such as the GRNN, differs from the Radial Basis Functions by the second transfer function. In this case, the input from the hidden layer became the argument of a competitive function, that assigns 1 to the maximum value presented. When a new input is presented, the first layer computes distances from the input vector to the training input vectors and produces a vector whose elements indicate how close the input is to a training input. In this layer, a measure of similarity between the new input and the training inputs is measured.

The second layer sums these contributions for each class of inputs to produce as its net output a vector of probabilities. Now, a competitive transfer function in the output layer picks the maximum of these probabilities, and produces a 1 for that class and a 0 for the other classes. Before the computation of competitive function, the probabilities can be considered as results of a membership functions of fuzzy logic methodology, where probabilities represent the probability of the input to belong to each output class. In this case, the competitive function allows to choose the class of destination (i.e., membership). Figure 53 shows the architecture of PNN that is equal to the RBF topology with the exception of the second transfer function.

In general, PNNs are used in classification problems and the following exercise shows a simple example of their working.

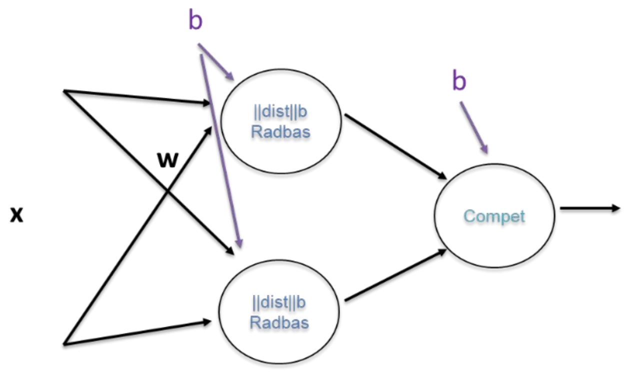


Figure 53. Probabilistic Neural Network topology (PNN).

Let us consider three inputs, each of two elements (x_1 = [1, 2], x_2 = [2, 2], and x_3 = [1, 1]), and one target (y = [1, 2, 3]). Supposing that a new input (out-of-sample) is presented to the PNN (x_4 = [1.7, 0.9]), figure 54 shows training inputs (x_1, x_2, x_3) in blue and the (out-of-sample) new input (x_4) in red.

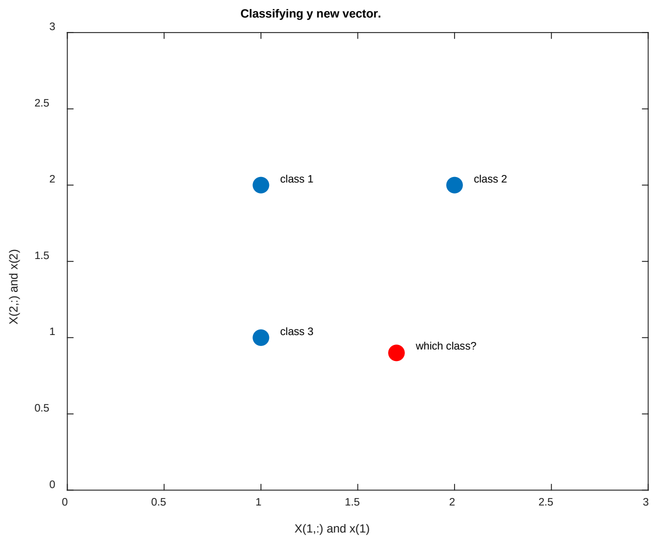


Figure 54. Classification problem on the Cartesian coordinate system.

The PNN calculates the probability of each input to belong to a class. Notice that the ANN knows that x_1 belongs to class 1 (target = 1), x_2 belongs to class 2 (target = 2), and finally x_3 belongs to class 3 (target = 3). When x_4 is presented as a new input, the PNN does not know the class for this element, so the NN calculates the probabilities that x_4 belongs to class 1, class 2, and class 3. Through the competitive transfer function, the output neuron transforms the obtained probabilities, assigning 1 to the class with the highest probability value. In this simple example, the results are 0 for both classes 1 and 2, and 1 for the third class. It is then clear that PNN assigns class 3 to the new input, as shown in figure 55.

Thanks to this exercise, we have shown why PNNs are applied with success in classification problem: their high ability to assign patterns to their correct class.

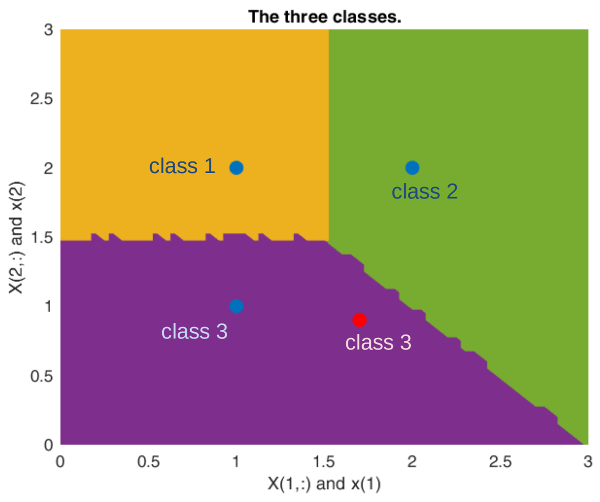


Figure 55. Assignment of x_4 to class 3 through PNN.

6.3 Back-ward/Recurrent Network

Are called Recurrent Networks all architectures where connections between layers are not only feed-forward, but also back-ward (in red in figure 56).

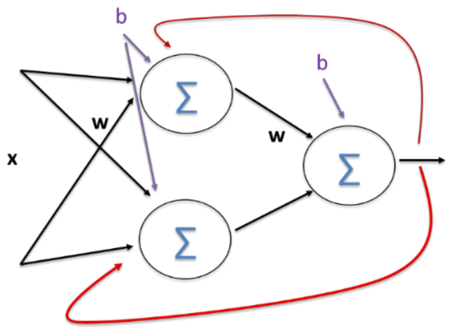


Figure 56. Recurrent Network topology.

These typologies of ANN can learn sequential or time-varying patterns. Even if they can require more computational power, the weights allow to consider two effects:

- *Direct effect:* a change in the weight causes an immediate change in the output. In figure 56, these feed-forward connections are black arrows (in purple for the bias).
- *Indirect effect*: some of the inputs to the layer are also functions of the weights. In this architecture, connections can come back and they are called back-ward, presented with red arrows in figure 56.

The back-propagation algorithm used for updating weights is the same for both typologies of connections. However, when the effect is indirect, the algorithm is called "dynamic back-propagation".

They are applied in many fields: prediction in financial markets, channel equalization in communication systems, phase detection in power systems, sorting, fault detection, speech recognition, and so on… Nevertheless, when back-ward connections are provided, the complexity of NN increases, such as the computational requirements. In addition, it is not always simple to understand what happen in hidden layers, moreover if there are more than one layer with more neurons.

With that in mind, back-ward connections are often used and they can be applied with satisfying results.

6.3.1 Nonlinear Autoregressive Network (NARX)

One of the most interesting architecture of Recurrent networks are the NARX models. They are Nonlinear supervised AutoRegressive network with eXogenous inputs, then they are recurrent and dynamic. Indeed, they present feed-forward and back-ward connections (from now "feedback connections") enclosing several layers of the network.

The peculiarity of this architecture is that it is not only recurrent but it is also dynamic. This last adjective means that ANN learns from previous values of inputs and/or outputs.

This model is based on the linear ARX[13] model used in time-series, and the output can be defined as follows (Beale et al., 2019):

$$y(t) = f\,(y(t-1),\, y(t-2),\, \ldots,\, y(t-n_y),\, u(t-1),\, u(t-2),\, \ldots,\, u(t-n_u))$$

where the next value of $y(t)$ is regressed on previous values of the output and previous values of an independent (exogenous) input signal $u(t)$.

A simplified architecture of a NARX NN with one exogenous input (u_t), one output and three feed-forward hidden layers is presented in figure 57.

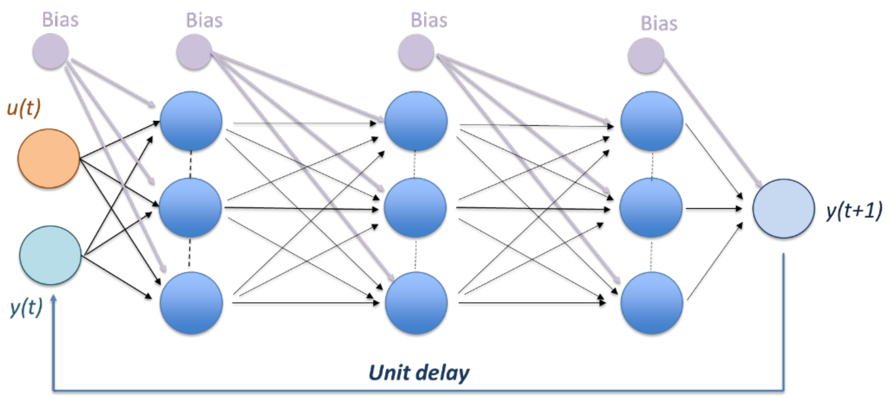


Figure 57. Simplified representation of a NARX model with one exogenous input, one output and three feed-forward hidden layers.

Notice that this architecture sums together non-linearity (this depends from activation functions used), memory (time-series are used as inputs), and feed-back connections (feed-forward connections in the hidden model and back-ward from output to input for computing the unit delay).

6.3.2 Long Short-Term Memory Neural Networks (LSTM)

The LSTM neural networks deserve a special attention. These topologies are composed by one or two LSTM layers that are recurrent networks for time-series analysis. The architecture of a LSTM layer is not simple because considers together time-series and features.

Let us consider a time-series X with C_S features (*called channels*), figure 58 shows the LSTM layers, where h_t represents the output (also called *hidden state*) and c_t identifies the cell state at time t.

[13] The ARX model can be linear or non-linear and they refer to AutoRegressive with eXogenous variables.

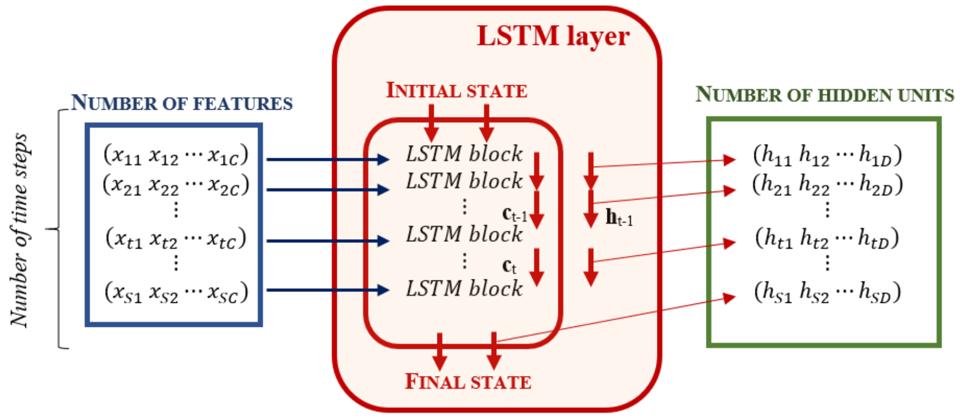

LSTM layer
Number of features
Initial state
Number of hidden units
Number of time steps
$(x_{11}\, x_{12} \cdots x_{1C})$
$(x_{21}\, x_{22} \cdots x_{2C})$
$(x_{t1}\, x_{t2} \cdots x_{tC})$
$(x_{S1}\, x_{S2} \cdots x_{SC})$
LSTM block
LSTM block
LSTM block
LSTM block
c_{t-1}
c_t
h_{t-1}
$(h_{11}\, h_{12} \cdots h_{1D})$
$(h_{21}\, h_{22} \cdots h_{2D})$
$(h_{t1}\, h_{t2} \cdots h_{tD})$
$(h_{S1}\, h_{S2} \cdots h_{SD})$
Final state

Figure 58. LSTM layer flow.

At the first-time step, final output and updated cell state are calculated considering the initial state of the network in the first LSTM block.

At time t, c_{t-1} and h_{t-1} together with the next step of the sequence are used for calculating the output and c_t (i.e., the updated cell state).

The output or hidden state and the cell state compose the state of the LSTM layer. At time t the output state contains the output at step t from the corresponding LSTM block. Information learned in previous time step is collected in cell state. The information is learned and updated step-by-step through gates, as shown in figure 59 (Beale et al., 2019).

In figure 59, f_1 is the *forget gate* that controls the reset state of cell; f_2 represents the *cell candidate* that adds information to the cell state; f_3 represents the *input gate* that controls the state update of cell; finally, f_4 is the *output gate* that controls the cell level added to the hidden state. All gates are not linear: logistic or sigmoidal functions are applied.

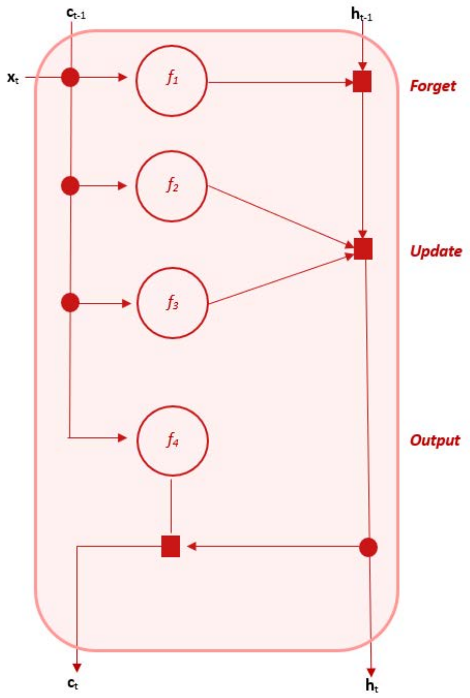


Figure 59. Gates in LSTM layer. How they forget, update, and produce cell and hidden states.

These networks are often used in *Deep Learning* framework because they are effective in classification and regression.

In general, for classification LSTM NN are used before a fully connected layer (as in convolutional networks) and a *softmax* function; whereas for regression, the output is obtained directly after the fully connected layer.

An application field is the text mining, where NNs are able to classify different sentences into different classes.

6.4 Competitive Networks

Competitive networks are neural networks made by only one layer of competitive neurons. They are, together with the Self-Organizing Map (SOM), the only neural networks with unsupervised learning process, that is to say that in the training set, the pre-assigned values (i.e., the targets) are not introduced in the model (figure 60).

The competitive learning rule works following three main steps (Haykin, 1999):

1. The only layer is made by neurons equal among them, but weights are randomly initialized, so they can give different responses to the same input patterns.
2. The neuron whose weight vector is closest to the input vector is updated to be even closer. In this manner, when a new similar input pattern will be presented to the network, this neuron will win the competition among the other neurons, and from now it will be called "winner-take-all neuron".
3. As more and more inputs are presented, each neuron in the layer closest to a group of input vectors soon adjusts its weight vector toward those input vectors.

Through this process, the competitive network categorizes the input vectors presented, and its main application is the cluster analysis. Indeed, the Self Organizing Maps (SOM) are based on a competitive layer and, for his reason, they are the ANNs more frequently applied in clustering problem.

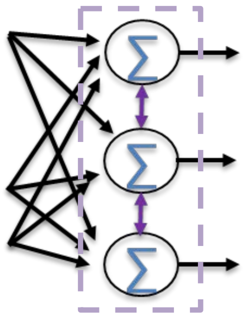


Figure 60. Competitive Network topology.

6.4.1 Self-Organizing Map (SOM)

Self-Organizing Maps are unsupervised and competitive neural networks proposed for the first time by Tuevo Kohonen in 1982. They are inspired by the brain's cortex where neighbouring neurons are activated by similar stimuli. In details, this ANN topology considers the reciprocal influence of neurons: neurons close to active neurons strengthen the links; neurons who are far are weakened.

The SOM network is composed by one input layer and one output one that is represented by a grid which competitive neurons are placed on, as presented in figure 61. As in previous architecture, each input is associated via connection weights to all neurons of the output layer.

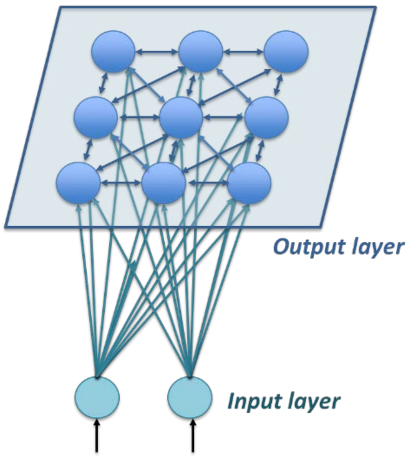


Figure 61. Self-Organizing Map (SOM) topology.

As for competitive NNs, the competitive layer considers the distance between weights and inputs, while the competitive transfer function returns 0 for each neuron except for the winner (i.e., "winning neuron") that keeps 1. In this manner, it is associated to the most positive element of net input.

SOMs allow not only to recognize the distribution of inputs they are trained on (as the competitive layer), but also their topology.

The Self-Organized Feature Map (SOFM) are similar to SOMs, but the algorithm starts with a pre-assigned grid of inputs topology.

When many inputs are presented, the network returns good results in clustering inputs, as in the example of man learning to drive: learning by trying and trying again…

Let us consider to group 1,000 subjects on the base of two variables (input 1 and input 2)[14]. We start defining a topology 8x6, then in output layer we find 48 neurons (figure 62).

[14] The sample used is the "simplecluster_dataset", provided in Matlab R2019b.

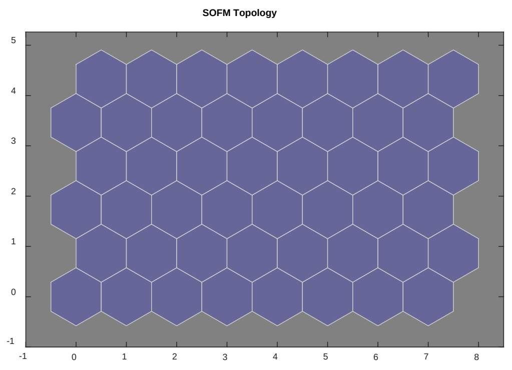
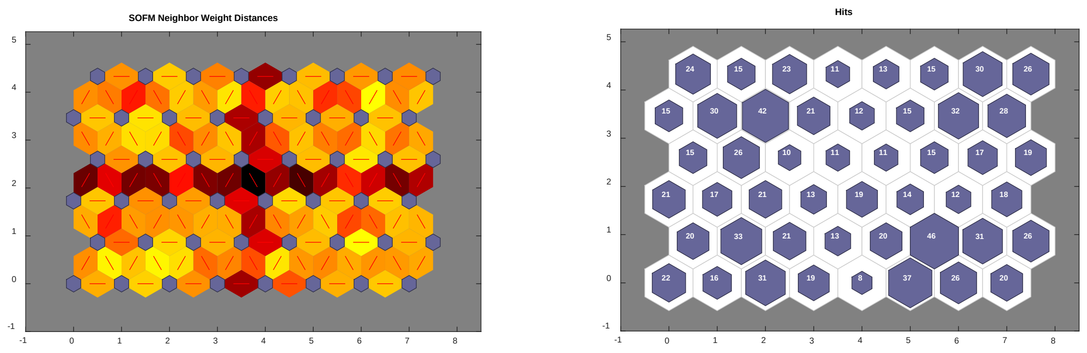


Figure 62. Self-Organizing Feature Map (SOFM) output layer.

The training phase is composed by three steps:

1. The inputs are presented one at time.
2. After the winning neuron is identified, weights of winning neuron and its neighbours are approached to input vector.
3. Weights of winning neuron are updated proportionally and weights of neighbours change on the base of half of the leaning rule.

Figure 63 (a) represents the 48 neurons (blue hexagons), regions among neurons containing the weights (red line). Colours of regions represent the distance size (i.e., darker the colour, larger the distance).



(a) (b)

Figure 63. Neighbours' weights among 48 neurons in output layer.

From the representation (a), the ANN grouped inputs into 4 clusters, and in figure 63 (b), the number of elements grouped for each neuron is provided.

Considering the high capability of SOM and SOFM in clustering, they are often used in pre-processing analysis for reducing the size of input layer.

6.5 Learning Vector Quantization (LVQ)

The Learning Vector Quantization network is a supervised ANN with a first competitive layer and an output linear layer. The competitive layer learns to classify input vectors considering a competitive function, and the negative distance between weights and inputs. In the first layer, the Kohonen learning rule is applied, as used by SO(F)M.

The output layer is linear and it transforms the competitive layer's classes into target classifications, and for this reason the LVQ networks are almost always used with SO(F)Ms (figure 64).

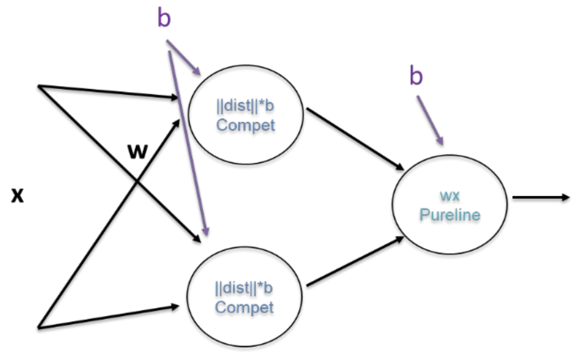


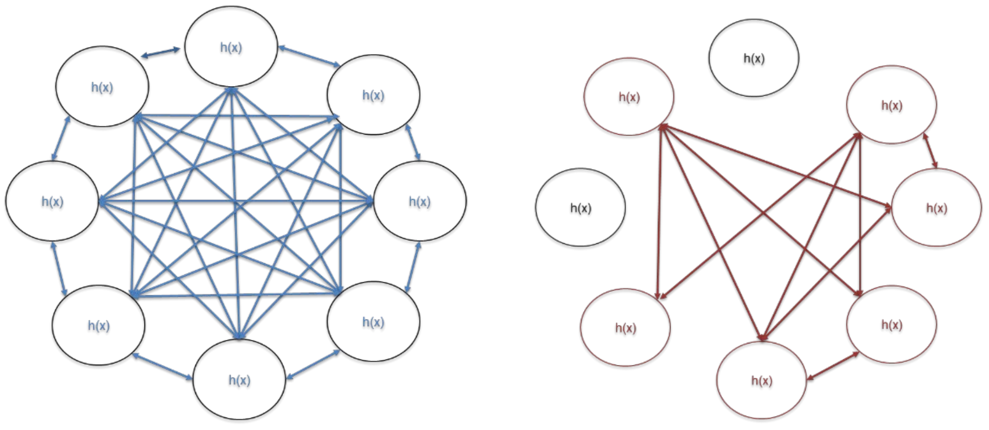
Figure 64. Learning Vector Quantization (LVQ) topology.

6.6 Hopfield Neural Network

In 1982 Hopfield proposed a recurrent unsupervised fully connected neural network with a symmetric hard-limit function:

$$h(x) = \begin{cases} 1 \; if \; x \geq 0 \\ -1 \; if \; x < 0 \end{cases}$$

As shown in figure 65 (a), the Hopfield is composed by only one layer where each neuron is simultaneously input and output.



(a) (b)

Figure 65. Hopfield network topology (a) and learning process (b).

Considering the activation function, neurons are activated or not and weights are updated using a specific "energy" function. Figure 65 (b) shows the network during or after the learning, where only some neurons are activated (in red), whereas two nodes have been switched off (in dark grey).

This topology is very powerful in creation of associative memories, resistant to the alteration of operating conditions, and the solution of combinatorial optimization problems. A strength of Hopfield networks is moreover the ability to recognize an image even when it is not exact. Let us consider an erased written (figure 66 (a)), a Hopfield network well trained is able to recognize and complete the written (figure 66 (b)).

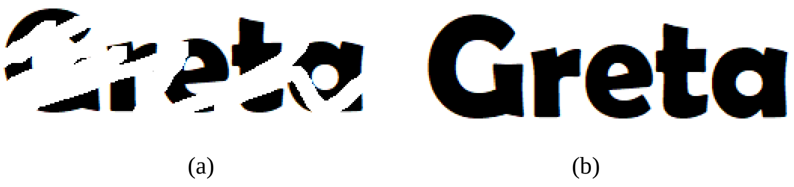


Figure 66. Hopfield recognizing output.

A generalization of Hopfield neural networks is the **Boltzmann Machine**¸ proposed by Ackley, Hinton and Sjnowski in 1985. They substituted the deterministic neurons of Hopfield networks with probabilistic ones. In this manner, new networks are characterized by a probability distribution of states where neurons with higher energy present a lower probability of those with lower energy. The probability to find the network in a specific case is represented by the well-known in thermodynamics Boltzmann distribution. In general, they are used successfully in image recognition.

6.7 Reservoir Computing

With Reservoir Computing we refer to a set of methodologies for training recurrent and dynamic supervised neural networks (for instance: Echo State Networks, Liquid State Machines, Back-propagation algorithm decorrelation, and so on…).

Figure 67 shows a simple topology of a Reservoir Computing ANN. We can observe that there are two different types of connections: feed-forward (in red) and recurrent (in purple). Then, the essential idea is training the network considering separately the two phases:

1. The *reservoir*, that is the recurring part of the network.
2. The *readout*, that is the non-recurring part (feed-forward) of the network.

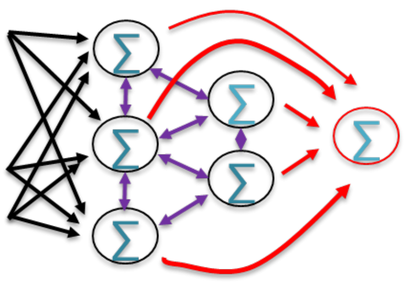


Figure 67. Reservoir Computing NN.

When the first phase has been fixed generating randomly the internal connections, the readout is trained through a linear regression.

This topology is used successfully in time-series analysis, and the architecture can be very complex.

6.8 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNN) are complex architectures based on a convolution layer. In mathematics, the convolution is an operation that produces a function (*z*) expressing how the shape of a function (*f*) is modified by another one (*g*). The term "convolution" refers both to the function result and to the process for obtaining it.

CNNs are composed by many convolutional hidden layers, then they are a *Deep Learning* model. More frequently used layers refer mainly to the following three types:

1. A *Convolution process* inputs with some convolutional filters, each of which activates certain features from the analysed inputs.
2. A *Rectified Linear Unit* activation function (i.e., ReLU), that allows to activate only significant features to submit to the next layer.
3. A *Pooling phase* simplifies the output by performing a non-linear downsampling, reducing the number of parameters that the network must learn.

After learning in the "convolutional phase", the selected features are passed to a fully connected layer for the classification.

ReLU function: returns 0 if the argument is negative, otherwise it returns the maximum value of the function.

Softmax function:

$$softmax(x) = \frac{e^x}{\sum e^x}$$

These complex architectures are used with success in image detection, indeed they are inspired to the human visual cortex.

As shown in figure 68, the first layer (i.e., convolutional layer) applies a convolution on inputs and then produces an output. In this layer, a ReLU and a pooling phase[15] act in order to reduce the number of parameters.

The obtained outputs are the net-inputs of a second fully connected layer, that works as a supervised recurrent network where the activation function is represented by a Support Vector Machine[16] or by a *softmax* function.

[15] This is a characteristic layer of convolutional network. In this phase, a kernel function is applied for dimensionality reduction. Two kernel functions are the most commonly used: the max pooling and the average pooling. In this layer, all information collected in the features is compressed.

[16] For a simple explanation of SVM, see section 3.4.

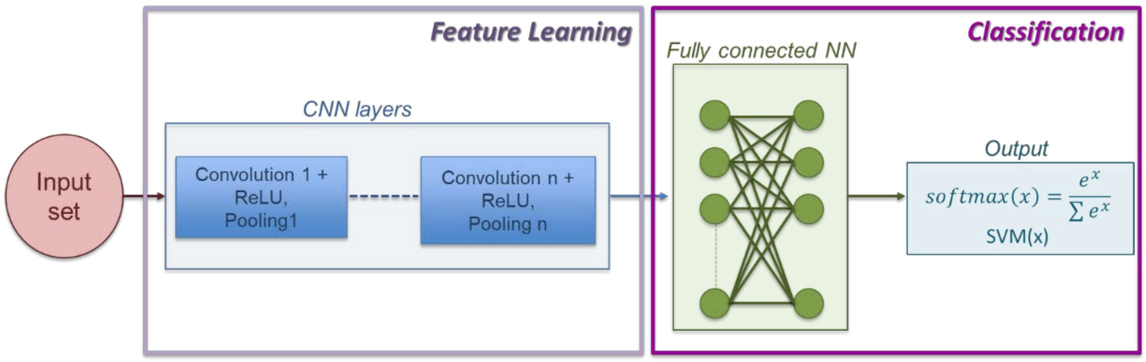

Feature Learning
CNN layers
Input set
Convolution 1 + ReLU, Pooling1
Convolution n + ReLU, Pooling n
Classification
Fully connected NN
Output
$softmax(x) = \frac{e^x}{\sum e^x}$
SVM(x)

Figure 68. Convolutional Neural Network (CNN) topology.

7 Final considerations

In this paragraph we sum up conclusions about *Deep Learning* methodologies and *Artificial Neural Networks*. In a first sub-section, we list and discuss the phases of methodologies' definition, while in the final sub-section a debate on strengths and weaknesses of these models is proposed.

7.1 Steps for ANN model definition: the choice of correct parameters

The first step in defining an ANN model is identifying correctly the problem and its formulation. In particular, in this phase it is necessary to clarify the relation of cause-effect in order to exclude reverse causality problem (step 0). Once identified the problem, we can pass to design the model and then to choose the network architecture and all parameters (step 1). After the training and the validation of the ANN (step 2), we can test the model (step 3). Now, if we are satisfied, we can continue including the identified model in an algorithm or in another software, and, at the same time, we proceed updating all parameters (step 4); otherwise, we have to restart with the step 1 (or step 0 if data are not convincing), as shown in figure 69 (Basheer & Hajmeer, 2000).

First two phases (step 0 and step 1) are the most critical because the goodness of data is essential in order to have good results, and, at the same time, results depend also from model parameters.

In **step 0**, before dividing data in training and validation set, they have to be analyzed and preprocessed. Databases too big can confuse the ANN, but an initial sample too small can compromise the generalization ability of network (see section 4.5.1 for a discussion on overfitting and underfitting topics). There is not a rule for defining the size of the dataset, the only solution is evaluating the accuracy after the validation/testing step. Looking at, for instance, the case of classification of a dataset in three classes. The starting sample must represent well all three classes and elements of each class have to be controlled for removing outliers[17].

[17] In statistics, the outliers are values significantly different from other observations of the database.

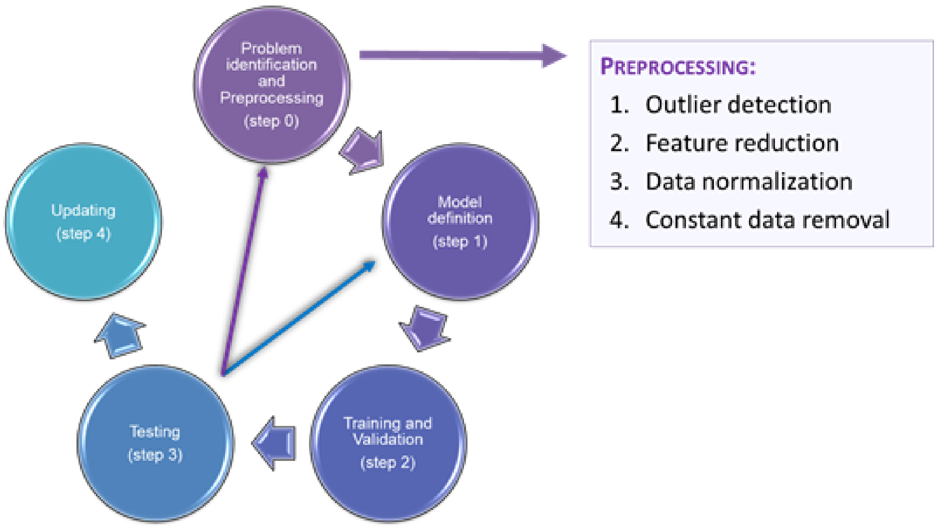


Figure 69. ANN developing steps.

In addition, a very relevant issue is to reduce the number of features to only those really determinant. This is a fundamental point because the ability of ANN to classify or generalize depends from the quality of information stored in the initial sample. In this sense, the input dataset can be treated with the Principal Component Analysis (PCA) or with the Support Vector Machines (SVM). The PCA is a statistical technique commonly used for identifying the component(s) more crucial of a dataset. Considering, for instance, a town where the majority of shops is concentrated in a single street and its crossroads. Obiovously, some shops exist also in other areas, but if we want to have shopping, we have to go in the main street (figure 70). Nevertheless, we have never been to this town, and therefore we do not know where the street is, we only know the GPS coordinates of the shops.

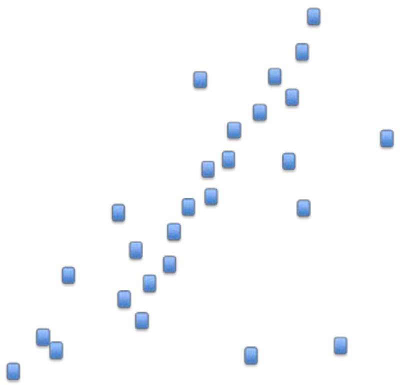


Figure 70. Map of shops in the town.

How searching the shopping street?

We can start placing the origin of a coordinate system on the mean of shops' coordinates (step 1, figure 71 (a)), and after we can rotating the axes minimizing the distance of shops from the x-axis (step 2, figure 71 (b)). Now, the new direction is the direction along which the dispersion of the data is the greatest (see the dashed lines in figure 71 (b)). This is called *first principal* *component*. We can continue rotating for searching the second principal component, and we can carry on until the *y* became very small for each shop.

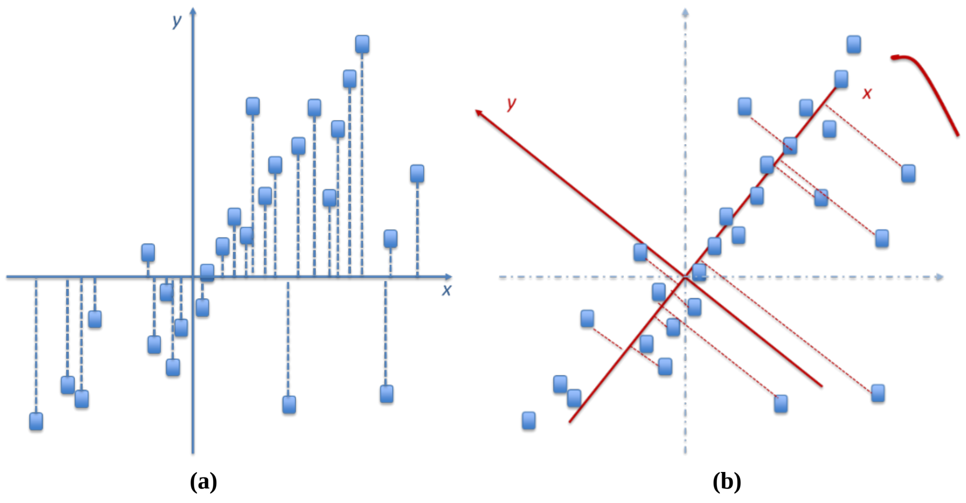


Figure 71. Searching the correct component.

This example, inspired to Domingos (2015), is very simple but it explains better as principal component analysis works.

Support Vector Machines (SVM) work defining vectors on boundaires among different groups of data, as explained in section 3.4.

Another preprocessing step, always strongly recommended, is the normalization of data between 0 and 1 or -1 and +1. A standard formula applied for normalizing the variable x_i into a predefined range $[\xi_{1,}, \xi_2]$ is the following:

$$z_i = \xi_1 + (\xi_2 - \xi_1)\left(\frac{x_i - x_i^{min}}{x_i^{max} - x_i^{min}}\right)$$

Where z_i is the normalized value of x_i and x_i^{min} and x_i^{min} are its minimum and maximum values.

Last but not least, it is a good practice removing inputs/targets that are constant.

Once having pre-processed data, we can start with next step (i.e., **step 1**) that is the most interesting. Indeed, it is here that we have to set the ANN architecture and its parameters. In particular, we face with the following decisions:

1. *Initialization of ANN weights*. As discussed in previous sections, weights are initialized and, as shown in section 4.4.4, they can affect the probability to find a local or global minimum or, more generally, they can affect ANN results and performance. A general rule is that weights are initialized uniformly in a small range with zero-mean random numbers (Rumelhart et al. 1986).
2. *Learning rate (η) of back-propagation algorithm*. A high learning rate accelerates the training and weight vector (or matrix) changes significantly from one epoch to another. This allows that the ANN difficultly will find the convergence and the error will continue to oscillate. On the contrary, a small learning rate will find the global minimum, but in a very slow way. A

methodology to adopt is to apply an adaptive learning rate, varying along the training. In this manner the optimal weight vector (or matrix) will be easily find.

3. *Activation or Transfer function*. The function transforms the weighted sum of signals in a neuron. The majority of applications adopt the non-linear and differentiable sigmoid function, because its properties are essential requirements for back-propagation algorithm. However, there is not a specific rule, besides the type of problem to solve (i.e., for instance, classification or forecasting).
4. *Early stopping criterion*. In section 4.4.3, we have discussed about the possibility to stop the network when performance of ANN on validation diverges from results on training. The criterion used for measuring performance can be defined by the ANN designer, but in general it is used the Sum-of-Squared-Errors (SSE):
$$SSE = \sum_{p=1}^{N}\sum_{i=1}^{M}(t_{pi} - o_{pi})^2$$
where N is the number of the training sample, M is the number of output nodes, t are the targets and o are the outputs of the ANN.
It is noteworthy that the choice of the metric adopted depends also from the problem to solve.
5. *Number of epochs*. In this case, the criterion is empirical. Indeed, when the error starts to increase, it is necessary to stop the back-propagation algorithm and set the number of epochs at this time.

Remember that one pass through all weights for the whole training set is called **epoch of training**.

6. *Incremental or Batch training*. This choice has been discussed in section 4.4.2.3. In general, batch training is used in recurrent and/or dynamic networks, but this mode is more likely to be entrapped into a local minimum. Even in this case, the advice is to choose the mode empirically.
7. *The size of hidden layer*. Literature suggests that only one hidden layer is sufficient for approximating any type of function (Hornik et al., 1989). However, if the function shows discontinuity, it is recommended to increase the number of hidden layers (i.e., two at least). The number of nodes in hidden layers is strictly related to the complexity of patterns presented to the ANN. However, increasing the number of nodes means also increasing the time necessary for computation. As suggested by Basheer & Hajmeer (2000), figure 72 represents what happens when too few or too many hidden nodes are defined into the hidden layer(s). Considering a simple problem where *y* is a function of *x*; in figure 72 are represented training and validation points (light-blue dots for training and stars in water green for validation). The dash-dot line in pink represents the generalization of a network with too few nodes in the hidden layer(s), whereas the dashed line in purple shows the performance when too many hidden nodes are defined. The blue curve is the result of network when the optimal number of hidden nodes is set up.

Unfortunately, there is not a mathematical rule for establishing an a-priori number of hidden nodes, but we can only proceed with trials and errors. For our knowledge, many authors try to suggest some formula, as for instance the following two simple rules:

For instance, Masters (1994) indicates that the number of hidden nodes has to be equal to: $(number\ of\ inputs * number\ of\ outputs)^{1/2}$. Hecht-Nielsen (1990), following the

Kolgomorov theorem, suggests a number of hidden nodes lower or equal to the number of inputs +1.

However, researchers converge on the idea that the optimal solution can be found only empirically, that is to say trying and re-trying.

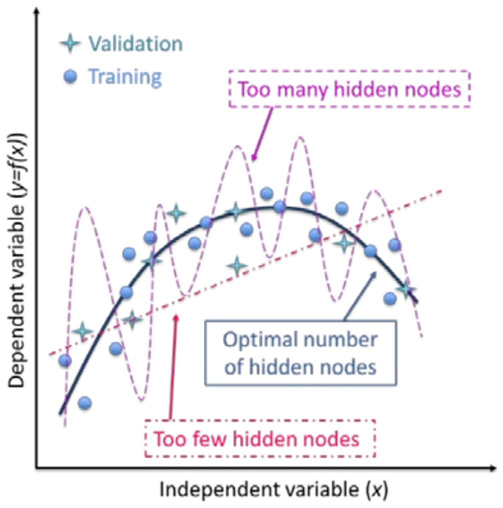


Figure 72. Size of hidden layer(s) and network performance.

7.2 Advantages and Disadvantages

Now, we have therefore arrived at the end of the handbook and it is time to take stock about *Deep Learning*, in particular about *Artificial Neural Networks*.

Strengths of ANNs can be reassumed in the following list:

- Information is stored not in the initial database, but in all nodes of all layers of ANN. For this reason, if pieces of information are lost, the ANN continues working. *ANNs are not sensitive to information losses*.
- After the training, the ANN is able to assign an output also if incomplete information is presented as input and without losing in performance. Then, *ANN is able to work also with incomplete information*.
- If a part of network does not work, the ANN is able the same to generate output without errors. *The ANN is fault tolerant*.
- ANN define properties of patterns in each layer. Then, *ANNs perform very well feature extraction*.
- ANNs are useful in designing *non-linear relations*.
- ANNs require *neither specific data distribution nor model hypotheses*.
- ANN, for all reasons explained in the handbook, are *very flexible tools*.
- Thanks to layers, *ANNs study data at a bigger extraction level* in comparison with *Machine Learning* techniques.
- Finally, considering *Machine Learning* and *Deep Learning*, Aggarwal (2018) suggests that the accuracy of results from *Deep Learning* models increases with the increasing of the

amount of data, as shown in figure 73. On the contrary, growing the size of input data, the accuracy of *Machine Learning* techniques decreases.

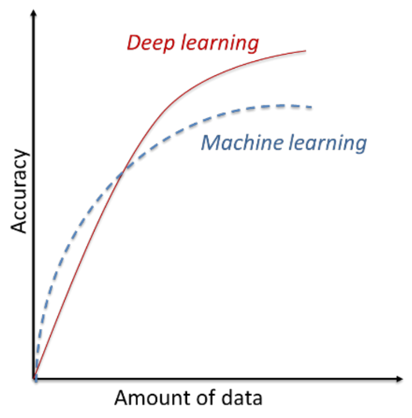


Figure 73. Accuracy and data: *Deep Learning* vs *Machine Learning*.

Nevertheless, also *Artificial Neural Networks* and then *Deep Learning* present weaknesses. In particular, we can list them as follows:

- ANNs and even more *Deep Learning* require high computational power. They are *dependent from hardware equipment*.
- What happens in hidden layers is not always clear because in complex architectures it is difficult to identify all mathematical results in hidden nodes. For this reason, ANNs are often considered *black-boxes*.
- ANNs can suffer of *overfitting* and/or *underfitting*, that we discussed in section 4.4.3;
- The size of initial data is determinant for ANN successful results. The training sample has to be representative of reality, so that ANN is able to learn correctly all characteristics of data. *ANN results depend from the size (and quality) of training data*.

8 Appendix A: Distance measures

Euclidean distance

Let us consider two variables X and Y of length m:

$$X = x_1, x_2, \ldots, x_m \text{ and } Y = y_1, y_2, \ldots, y_m$$

We can define their distance as follows (Danielsson, 1980):

$$D_E(X,Y) = \sqrt{\sum_{i=1}^{m}(x_i - y_i)^2}$$

Minkowski distance

Let us consider two variables *X* and *Y* of length *m*:

$$X = x_1,\ x_2,\ \dots,\ x_m \text{ and } Y = y_1, y_2, \dots, y_m$$

We can define their distance as follows (Ichino & Yaguchi, 1994):

$$D_M(X,Y) = \sqrt[r]{\sum_{i=1}^{m} (x_i - y_i)^r}$$

where *r* is a parameter that can assume the following values:

- *r*=1: Manhattan distance;
- *r*=2: Euclidean distance;
- $r \to +\infty$: when the distance between vectors is maximum.

Lagrange-Tchebychev distance

Let us consider two variables *X* and *Y* of length *m*:

$$X = x_1,\ x_2,\ \dots,\ x_m \text{ and } Y = y_1, y_2, \dots, y_m$$

We can define their distance as follows (Rajola, 2003):

$$D_{LT}(X,Y) = \max_{1 \le i \le +\infty} |X_i, Y_i|$$

Mahalanobis distance

Let us consider two variables *X* and *Y* of length *m*:

$$X = x_1,\ x_2,\ \dots,\ x_m \text{ and } Y = y_1, y_2, \dots, y_m$$

We can define their distance as follows (Penny, 1996):

$$D_{Mah}(X,Y) = \sqrt{(X-Y) \cdot \sigma(X,Y)^{-1} \cdot (X-Y)^T}$$

where $\sigma(X,Y)$ is the covariance matrix and *T* is the size of vector when *X* and *Y* are multivariate variables.

Correlation

Let us consider two variables *X* and *Y* of length *m*:

$$X = x_1,\ x_2,\ \dots,\ x_m \text{ and } Y = y_1, y_2, \dots, y_m$$

The linear correlation coefficient *r* is equal to the covariance of standardized variables X* and Y* (Orsi, 1995).

Its formulation is the following:

$$r = \frac{\sigma_{X,Y}}{\sqrt{\sigma_x \cdot \sigma_y}}$$

The coefficient *r* ranges between -1 and +1. Values equal to ±1 represent perfect linear dependence. If *r* = -1, the points are perfectly placed on a line with negative slope and then variables are inversely and proportionally correlated; otherwise, if *r* = +1, points are placed on a line with positive slope and then if a variable grows, also the other variable will increase. Finally, if *r* = 0, the variables are not correlated.

<u>Jaccard distance</u> (only for dichotomous variables)

Let us consider two dichotomous variables *X* and *Y* of length *m*:

$$X = x_1,\ x_2,\ \ldots,\ x_m \text{ and } Y = y_1, y_2, \ldots, y_m$$

The similarity among them can be measured defining the following quantities:

- M01 = number of attributes where *X* has 0 and *Y* has 1;
- M10 = number of attributes where *X* has 1 and *Y* has 0;
- M00 = number of attributes where *X* has 0 and *Y* has 0;
- M11 = number of attributes where *X* has 1 and *Y* has 1.

The Jaccard index (Jaccard, 1901) can be calculated as follows:

$$J = \frac{M11}{M01 + M10 + M11}$$

As exercise, different distances have been calculated on a database of 10 elements and 3 variables. The number of distances calculated is (m*(m − 1))/2, then 10*9/2, sorted in increasing way and depicted in the figure 74[18].

The Jaccard distance has been excluded because it can be calculated with dichotomous variables, while, in this case, we have only continuous items.

Euclidean, Minkowsky and Lagrange-Tchebychev present similar results, where Mahalanobis and Correlation show very different results. This means that the choice of the distance is crucial in the analysis. In addition, other metrics exist, as, for instance, the *cityblock*, the *cosine*, the *spearman* and the *hamming*. A deepen explanation on metrics can be found in Berkhin (2006), even though the most commonly applied are those presented in this appendix. However, it is always possible for the user to define specific metrics.

[18] Distances have been calculated with Matlab R2019b.

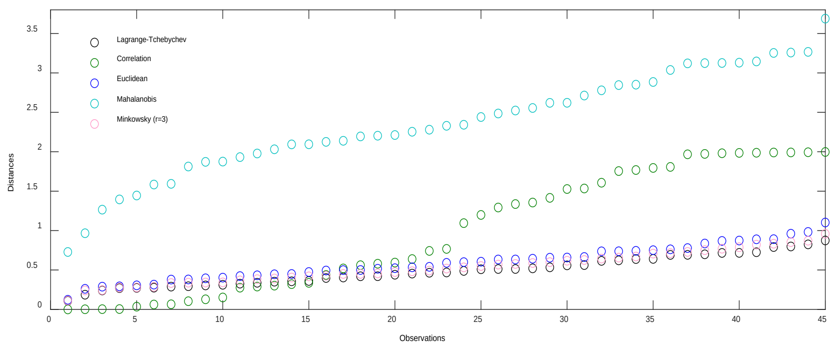


Figure 74. Different distances based on different metrics.

9 Appendix B: Basic Glossary

- ACCURACY: the accuracy of a model is measured as the ratio between the correct classification and the size of dataset.
- ACTIVATION FUNCTION: transforms the weighted sum of inputs in a neuron.
- ARTIFICIAL INTELLIGENCE (AI): represents the theory and development of informatics systems aiming at teaching computers to learning how human behave. The final goal is building models able to simulate the cognitive capabilities of human brains.
- ARTIFICIAL NEURAL NETWORK (ANN): is a representation of human neural brain in mathematical terms and its main goal is to simulate its functioning process. ANNs are composed by neurons and one or more layers.
- AUC: the Area Under the Curve represents the probability of correct prediction based on probabilities estimated by the model. The rule of thumb indicates that:
  - when AUC = 0.50 the model has no discriminatory capacity.
  - when AUC varies between 0.70 and 0.80 the model has an acceptable discriminator power.
  - when AUC ranges between 0.80 and 0.90 the model discriminated in an excellent way.
  - when AUC> 0.90 the model has a “super-excellent” discrimination power. This result is very difficult to obtain in real applications.
- BACK-PROPAGATION ALGORITHM: defines the process used by the neuron or network for updating weights.
- BACK-WARD CONNECTION: is a connection that does not travel from left to right, but, on the contrary, it goes from right to left.
- BIAS: is the difference between the prediction of the algorithm and the correct value to predict.
- DEEP LEARNING (DL): is a particular case of feature learning characterized by *Artificial Neural Networks* with two or more layers (often called multi-layers), capable of processing information in a non-linear way.
- DYNAMIC NEURAL NETWORK: is a network where input dataset is represented by time-series. These ANNs are also said “with memory”.

- EARLY STOPPING REGULARIZATION: this rule stops the training of the ANN, when the performance on validation set starts to increase. This algorithm allows to avoid the overfitting problem.
- EPOCH OF TRAINING: indicates the number of passes of the entire training dataset completed by the ML algorithm.
- EXPERT SYSTEM: is computer program reproducing the human reasoning, the logic, the belief, the opinion, and the experience (Plant & Stone, 1991; Fu, 1995).
- FEED-FORWARD CONNECTION: is a connection travelling from left to right.
- F-SCORE: is also known as Sørensen-Dice coefficient or Dice Similarity Coefficient, DSC. It represents the accuracy of a model considering the precision and the recall measures. It ranges between 0 and 1, where 1 indicates the perfect accuracy of the model tested.
- GARSON INDEX: defines the percentage of importance of features using ANN weights.
- GENERALIZATION PHASE: the ANN tests its ability and if it has been able to learn well the lesson, it is able to generalize results on similar data.
- GENERALIZATION: is the ability of an ANN to learn rules from a dataset and to apply relations learnt to another unknown set of data.
- GRADIENT: indicates the direction of greatest growth of a function (of several variables).
- GRADIENT DESCENT ALGORITHM: is the name of back-propagation algorithm when the minimization of the error occurs through steps in opposite direction to the gradient.
- HIDDEN LAYER: in ANNs, is the layer between input and output layers. In multi-layer neural networks, more than one hidden layer exists.
- INPUT: in ANNs, input sample represents the starting data on which the network is trained/validated/tested.
- LAYER: in ANNs, it represents a collection of neurons, but in simple frameworks, it can also consist of just one node. Hidden layers are layers between inputs and output layer, where ANN’s weights are calculated and stored. Note that the counting index of layers starts with the first hidden layer up to the output layer. Looking at the number of layers, we can distinguish between single-layer ANN architecture and multi-layer ANN framework. In ANNs, we can find, input layer, hidden layer(s), and output layer.
- LEARNER: is another way to call a learning algorithm.
- LEARNING ALGORITHM: represents a set of instructions of *Machine Learning* allowing a computer program to imitate the brain functioning.
- LEARNING PHASE: in this phase, the ANN learns to recognize rules among patterns and it stores them in the weights’ matrixes.
- LEARNING RATE*:* defines how smoothly the user shifts the decision boundaries of the Perceptron learning rule.
- LEARNING RULE: is a method applied repeatedly to the ANN with the aim to improve the performance of network. There are many learning rules depending from the network’s typology.
- LOGIC GATE OPERATOR: is a model or a device implementing a Boolean function. Starting from binary inputs, the model produces a binary output. Considering “true” = 1 and “false” = 0, the logic gate operators are (table 11):
  - *AND*: The output is true when both inputs are true. Otherwise, the output is false.
  - *OR*: The output is true if at least one operand is true. False if both are false.

- *XOR*: The output is true if both operands are different. False if both are true or false.
- *NAND*: The output is false if both operands are true. True otherwise.
- *NOR*: The output is true if both inputs are false. Otherwise, the output is false.
- *XNOR*: The output is true if the inputs are the same, and false if the inputs are different.
- *NOT* (special case): This operator works with only one input. If the input is 1, then the output is 0. If the input is 0, then the output is 1.

Input		**Output**						
A	**B**	**AND**	**OR**	**XOR**	**NAND**	**NOR**	**XNOR**	**NOT (only input A)**
0	0	0	0	0	1	1	1	1
0	1	0	1	1	1	0	0	1
1	0	0	1	1	1	0	0	0
1	1	1	1	0	0	0	1	0

Table 11. Logic Gate operator outputs.

- MACHINE LEARNING (ML): is a “field of study that gives computers the ability to learn with-out being explicitly programmed” (Samuel, 1959). The main goal is to build AI programs able to automatically define other programs for interpreting data and predicting results.
- MEAN SQUARED ERROR (MSE): is used for evaluating the performance of an algorithm. It is calculated as the mean squared error loss between target values and model predictions.
- MULTY-LAYER FEED-FORWARD NETWORK (Multi-layer FFNN): is an ANN with one input layer, one output layer, and one or more hidden layers of processing units. It does not present feed-back connections (e.g., a Multi-layer Perceptron or Shallow Networks if there are 2 hidden layers or Deep Learning increasing the complexity of architecture).
- NET INPUT FUNCTION: it is the function used for considering the bias (in general, it is the product of a weight times the input, but other functions exist).
- NEURON: is the crude approximations of biological neurons found in human brains and it is the base computational unit of ANN. It is sometime called NODE.
- NODE: is another way to call a neuron.
- OUTLIER: is a value significantly different from other observations of the database.
- OUTPUT: in ANNs, the output represents the response of the neural network for each element of the input sample.
- OVERFITTING: it happens when the model captures the noise along with the underlying pattern in data. In this case, the problem is that data are very noisy.
- PERCEPTRON: is an algorithm for supervised learning of binary classifiers. It is a single-layer neural network, so that the simplest architecture of *Artificial Neural Network*.
- PRE-PROCESSING PHASE: represents all techniques adopted for analysing a dataset and providing good quality data to the model.
- PRECISION: is the Positive Predicted Value (PPV) and it is calculated as the true positives divided by the sum of true positives and false positives. This index represents how many instances the model classifies correctly.

- PRINCIPAL COMPONENT ANALYSIS: is a statistical technique commonly used for identifying the features more crucial of a dataset. It is often used in a pre-processing phase for decreasing the size of the initial sample and the noise in the data.
- RECALL: is the sensitivity. It represents the robustness of the model and it indicates how many instances are missing.
- RECURRENT NETWORK: any network with at least one feed-back connection. It may, or may not, have hidden units. (e.g., a Simple Recurrent Network.).
- REDOUT PHASE: in reservoir computing framework it is the feed-forward (i.e., the non-recurring) part of the network.
- RESERVOIR PHASE: in reservoir computing framework it is the recurring part of the network.
- ROC: the Receiver Operating Characteristic is the graphical representation of the true positive rate (sensitivity) on the ordinate axis and false positive rate (1-specificity) on the abscissa one.
- ROOT MEAN SQUARED ERROR (RMSE): is used for evaluating the performance of an algorithm and it is the root of the MSE.
- SENSITIVITY: is measured as the true positives divided by the sum of true positives plus the false negatives. High value means that an element positive in the reality will never be classified as negative by the model.
- SHALLOW NEURAL NETWORK: is a neural network with one or, at maximum, two hidden layers.
- SINGLE-LAYER FEED-FORWARD NETWORK (Single-layer FFNN): is an ANN composed by one input layer and one output layer of processing units. No feed-back connections (e.g., a Single-Layer Perceptron).
- SPECIFICITY: is calculated as true negatives divided by the sum of true negatives plus false positives. High value means that an element negative in the reality will never be classified as positive by the model.
- STATIC NEURAL NETWORK: is a network where input dataset are not time-series. These ANNs are said “memoryless”.
- STOCHASTIC GRADIENT DESCENT ALGORITHM: works as the gradient descent algorithm but weights’ updates are made subdividing randomly the input patterns of the training set.
- SUM SQUARED ERROR (SSE): is a measure of the performance of an algorithm and it is calculated as the sum of the squared differences between targets and model outputs.
- SUPPORT VECTOR MACHINE (SVM): are a *Machine Learning* methodology for classification and non-parametric regression based on kernel functions.
- SYNAPSIS: in ANNs, it represents the connection between neurons of the neural network.
- TARGET: in ANNs, the target is the true state of the elements presented to neural network in the training/validation phase.
- TESTING SET: in ANNs, it is possible that algorithm considers not only a validation set but, in addition, another sample for measuring the generalization capability of neural network. In this case, the testing set is represented by 1/6 of the initial sample.
- TRAINING SET: in ANNs, is the starting sample of data on which neural network is trained. In general, it is represented by the 2/3 of the whole input sample.

- TRANSFER FUNCTION: is another way to call the transfer function.
- UNDERFITTING: a model is not able to capture the underlying pattern of the data.
- VALIDATION SET: in ANNs, is the portion of initial sample that is used for validating the performance of neural network. In general, it is represented by 1/3 of the initial sample. If the algorithm considers also a testing set, the validation set is represented by the 50% of 1/3 (i.e., 1/6).
- VARIANCE: variability of model prediction for a given data point.
- WEIGHT FUNCTION: in ANNs, it is the product of weights times input(s).
- WEIGHT: in ANN, it represents the mathematical measure of synapsis between the layers of the neural network.

10 Index of Figures

11 References

Abiodun, O.I., Jantan, A., Omolara, A.E., Dada, K.V., Mohamed, N.A., & Arshad, H. (2018). State-of-the-art in artificial neural network applications: A survey. *Heliyon*, *4*(11), e00938.

Abiodun, O.I., Jantan, A., Omolara, A.E., Dada, K.V., Umar, A.M., Linus, O.U., Arshad, H., Kazaure, A.A., Gana, U., & Kiru, M.U. (2019). Comprehensive review of artificial neural network applications to pattern recognition. *IEEE Access*, 7, pp. 158820-158846.

Aggarwal, C.C. (2018). *Neural networks and deep learning*. Cham: Springer Nature. Available at https://doi.org/10.1007/978-3-319-94463-0

Basheer, I.A., & Hajmeer, M. (2000). Artificial neural networks: fundamentals, computing, design, and application. *Journal of microbiological methods*, *43*(1), pp. 3-31.

Beale, M.H., Hagan, M.T., & Demuth H.B., (2019), Deep Learning Toolbox^{TM}. A user's guide (R2019b). Natick, MA. MathWorks.

Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, *2*(1), pp. 1-127.

Berkhin, P. (2006). A survey of clustering data mining techniques. In *Grouping multidimensional data* (pp. 25-71). Berlin, Heidelberg: Springer.

Casagranda I., Costantino G., Falavigna G., Furlan R., & Ippoliti R. (2016) Artificial Neural Networks and risk stratification models in Emergency Departments: The policy maker's perspective. *Health Policy*, 120, pp. 111–119.

Cerulli, G. (2021). Improving econometric prediction by machine learning. *Applied Economics Letters*, *28*(16), pp. 1419-1425.

Costantino G., Falavigna G., Solbiati M., Casagranda I., Sun B.C., Grossman S.A., Quinn J.V., Reed M.J, Ungar A., Montano N., Furlan R., & Ippoliti R. (2017). Neural networks as a tool to predict syncope risk in the Emergency Department. *Europace*, 19, pp. 1891-1895.

Danielsson, P.E. (1980). Euclidean distance mapping. *Computer Graphics and image processing*, *14*(3), pp. 227-248.

Deng, L., & Yu, D. (2014). Deep learning: methods and applications. *Foundations and trends in signal processing*, 7(3-4), pp. 197-387.

Domingos, P. (2015). *The master algorithm: How the quest for the ultimate learning machine will remake our world*. New York: Basic Books.

Falavigna G. (2008a). New contents and perspectives in the risk analysis of enterprises. *International Journal of Business Performance Management*, 10(2/3), pp.136-173.

Falavigna G. (2008b). A rating model simulation for risk analysis. *International Journal of Business Performance Management*, 10(2/3), pp. 269-299.

Falavigna G. (2008c). An analysis of key-variables of default risk with complex systems. *International Journal of Business Performance Management*, 10(2/3), pp. 202-230.

Falavigna G. (2012). Financial ratings with scarce information: A neural network approach. *Expert Systems with Applications*, 39(2), pp. 1784-1792.

Falavigna G., Costantino G., Furlan R., Ippoliti R., Queen J., & Ungar A. (2019). Artificial Neural Networks and risk stratification in Emergency Departments. *Internal and Emergency Medicine*, 14(2), pp. 291-299.

Falavigna G. (2020). Prediction of general medical admission length of stay with natural language processing and deep learning: a pilot study. *Internal and Emergency Medicine*, 15(6), pp. 917-918.

Falavigna, G. (2021). Deep learning algorithms with mixed data for prediction of Length of Stay. *Internal and Emergency Medicine*, 16, pp. 1427-1428.

Farizawani, A.G., Puteh, M., Marina, Y., & Rivaie, A. (2020, April). A review of artificial neural network learning rule based on multiple variant of conjugate gradient approaches. *Journal of Physics*. Conference Series *1529*(2). DOI: 10.1088/1742-6596/1168/2/022022

Fu, L., (1995). *Neural Networks in Computer Intelligence*. New York: McGraw-Hill.

Fukushima, K. (1975). Cognitron: a self-organizing multi-layered neural network. *Biological Cybernetics*, 20, pp. pp. 121-136.

Garson, G.D. (1991). Interpreting neural-network connection weights. *AI Expert*, *6*(4), pp.46-51.

Haykin, S. (1999). *Neural Networks. A comprehensive foundation*, Second Edition, Singapore: Pearson Education.

Hebb, D.O. (1949). *The organisation of behaviour: a neuropsychological theory*. New York: Science Editions.

Hebb, D.O. (1949). *The organization of behavior*. New York: Wiley.

Hecht-Nielsen, R. (1990). *Neurocomputing*, Addison. Wesely Publishing Company.

Hornik, K. Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators, *Neural Networks*, *2*(359366), pp. 3168-3176.

Hussain, M.A. (1999). Review of the applications of neural networks in chemical process control-simulation and online implementation. *Artificial intelligence in engineering*, *13*(1), pp. 55-68.

Ichino, M., & Yaguchi, H. (1994). Generalized Minkowski metrics for mixed feature-type data analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, *24*(4), pp. 698-708.

Ippoliti R., Falavigna G., Zanelli C., Bellini R., & Numico G. (2021). Neural networks and hospital length of stay: an application to support healthcare management with national benchmarks and thresholds. *Cost Effectiveness and Resource Allocation*, 19(67), pp. 1-20.

Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37, pp. 547-579.

Kohonen, T. (1973). A new model for randomly organized associative memory. *International Journal of Neuroscience*, *5*(1), pp. 27-29.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), pp. 436-444.

Li, Y., Jia, M., Han, X., & Bai, X. S. (2021). Towards a comprehensive optimization of engine efficiency and emissions by coupling artificial neural network (ANN) with genetic algorithm (GA). *Energy*, 225(C). DOI: 10.1016/j.energy.2021.120331.

Liao, S.H., & Wen, C.H. (2007). Artificial neural networks classification and clustering of methodologies and applications – literature analysis from 1995 to 2005. *Expert Systems with applications*, *32*(1), pp. 1-11.

Marullo, C., & Agliari, E. (2021). Boltzmann Machines as Generalized Hopfield Networks: A Review of Recent Results and Outlooks. *Entropy*, *23*(1), p. 34.

Masters, T., (1994). *Practical Neural Network Recipes in C++*. Boston, MA: Academic Press.

McCulloch, W.S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), pp. 115-133.

Minsky, M., & Papert, A.S. (1969). *Perceptrons, An Introduction to Computational Geometry*, Cambridge: MIT Press, Mass.

Neu, D.A., Lahann, J., & Fettke, P. (2021). A systematic literature review on state-of-the-art deep learning methods for process prediction. *Artificial Intelligence Review*, pp. 1-27.

Olden, J.D., Jackson, D.A. (2002). Illuminating the "black box": a randomization approach for understanding variable contributions in artificial neural networks. *Ecological Modeling*, *154*(1-2), pp.135-150.

Orsi, R. (1995). *Probabilità e inferenza statistica*. Bologna: Il Mulino.

Penny, K.I. (1996). Appropriate critical values when testing for a single multivariate outlier by using the Mahalanobis distance. *Journal of the Royal Statistical Society*: Series C (Applied Statistics), *45*(1), pp. 73-81.

Plant, R.E., & Stone, N.D., (1991). *Knowledge-based Systems in Agriculture*. New York: McGraw-Hill.

Rajola, F. (2003). *Customer relationship management: Organizational and technological perspectives*. Berlin, Heidelberg: Springer.

Ray, S. (2019, February). A quick review of machine learning algorithms. In *2019 International conference on machine learning, big data, cloud and parallel computing* (COMITCon), pp. 35-39. Faridabad, India: IEEE. DOI: 10.1109/COMITCon.2019.8862451.

Rumelhart, D.E., Hinton, G.H., & Williams, R.J. (1986). Learning internal representations by back-propagating errors. *Nature*, 323(99), pp. 533-536.

Rumelhart, D.E., Hinton, G.E., & Williams, R.J. (1987). Learning internal representations by error propagation. In J. L. McClelland & D. E. Rumelhart (eds), *Parallel Distributed Processing* (pp. 318-362), vol. 1. Cambridge, MA: MIT Press.

Samuel, A.L. (1959). Some studies in machine learning using the game of checkers. IBM *Journal of research and development*, *3*(3), pp. 210-229.

Shrestha, A., & Mahmood, A. (2019). Review of deep learning algorithms and architectures. *IEEE Access*, 7, pp. 53040-53065.

Singh, S. (2018, May 21). Understanding the Bias-Variance Tradeoff. Available on https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229 (last visit January 09 2022).

Suthaharan S. (2016). *Support Vector Machine. In: Machine Learning Models and Algorithms for Big Data Classification*. Integrated Series in Information Systems, vol 36. Boston, MA: Springer.

Vapnik, V. (1995). *The Nature of Statistical Learning Theory*. New York: Springer.

Ying, X. (2019, February). An overview of overfitting and its solutions. *Journal of Physics*. Conference Series, *1168*(2). DOI:10.1088/1742-6596/1168/2/022022

Zainuddin, Z., & Pauline, O. (2008). Function approximation using artificial neural networks. *WSEAS Transactions on Mathematics*, *7*(6), pp. 333-338.

Zhang, W.J., Yang, G., Lin, Y., Ji, C., & Gupta, M.M. (2018, June). On definition of deep learning. In *2018 World automation congress* (WAC). pp. 1-5. IEEE.

Itinerari per l’alta formazione

Abstract

The aim of the present handbook is to drive the reader toward a deepen knowledge of *Deep Learning* (DL) methodologies and their functioning.
Even if DL has ancient ancestry, only in recent years they have been strongly revalued and consequently applied. However, it is noteworthy that in this field, the increasing computational power played a fundamental role: *Deep Learning* models are techniques of success, but, at the same time, they are also expensive from a computational point of view and not always clearly understandable.
The handbook starts considering the "big family" of *Artificial Intelligence* (AI), and continues clarifying what are *Machine Learning* (ML) and *Deep Learning* (DL). The reader will discover that DL is based on *Artificial Neural Networks* (ANN), then the fundamentals of ANNs will be presented and discussed, until getting to understand the working and the differences among different architectures.
Finally, at the end of the reading, the reader will have in hands all knowledge required for recognizing different *Deep Learning* architectures, their functioning, and the fields of their applications.

ISBN : 978-88-98193-25-7