Contents

Chapter 1

Data-Driven Modelling

Data-Driven Modeling (DDM) has revolutionized the approach to system design and analysis, placing intensive data usage at the core of decision-making and modeling processes.

This chapter and chapter-2 assumes basic knowledge of traditional modeling through countinuous time (Laplace) and discrete time (Z-transform), and some concept of machanics like the motion equation.

1.1 The Emergence of Data-Driven

We will discuss the importance of DDM in system analysis, emphasizing how the ability to extract models directly from data can enhance prediction accuracy and model adaptability to changes in the operational context.
This approach becomes increasingly mandatory when dealing with complex systems whose dynamics are often unknown, making reference to classical transfer functions (TF) unfeasible.

$$TF = G(s) = \frac{Y(s)}{U(s)}$$

Fluid dynamics, atmospheric phenomena, and general real systems are often referred to in these cases.
The approach used for complex nonlinear systems with known **physics** was often to study their dynamics and write motion equations (or Lagrangians).

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{q}_i}\right) - \left(\frac{\partial T}{\partial q_i}\right) + \left(\frac{\partial V}{\partial q_i}\right) = \frac{dL}{dq_i} \tag{1.1}$$

This is a nonlinear equation that needs linearization. However, it demands significant computational effort and becomes challenging to control due to the high number of degrees of freedom

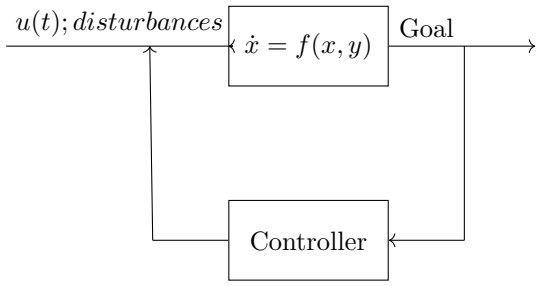


Figure 1.1: Caption

In this historical period, where the collection of an indefinite number of data is possible, data-driven modeling is gaining traction.

1.2 Modelling with Data: System Identification

System Identification is a fundamental pillar in DDM, enabling the extraction of models directly from empirical data.
System Identification is the process of identifying and modeling dynamic systems based on observed data, allowing the model to generalize the future dynamics. We will explore how this approach is essential for gaining a detailed understanding of the system's behavior without relying on pre-existing (physical) theoretical models.

Note: System identification differs from *model reduction* methods, where the latter often starts with a known model. However, due to numerous degrees of freedom, calculating various states incurs high computational costs. Thus, a shift is made from a model

$$\dot{x} = f(x, u)$$

to a model

$$\dot{a} = f(a)$$

with $a << $ degrees of freedom, enabling a faster and more stable controller.

System identification methods can be categorized into:

- **Linear**: Aiming to reduce to a model like

  $$\dot{x} = \boldsymbol{A}x + \boldsymbol{B}u$$

  For a linear system, there are already theories to construct the optimal controller.

- **Nonlinear**: It is more challenging to determine a model and harder to control.

Generalizing:

Given a series of input and output measurements $(\bar{u}, \bar{y})$, a family of models $\boldsymbol{M}(\beta, u)$ is chosen. Its output for an input $\bar{u}$ is $\hat{y} = \boldsymbol{M}(\beta, \bar{u})$. With a metric $D(\hat{y}, \bar{y})$ indicating the distance between the actual output value $\bar{y}$ and the estimated $\hat{y}$, the goal is to find values of $\boldsymbol{\beta}$ such that $\hat{y} = \boldsymbol{M}(\beta, \bar{u})$ guarantees a minimal $D(\hat{y}, \bar{y})$

Theorem1.2

All these methods are based on **Regressions**. For this specific case, both linear and nonlinear methods were analyzed. In the following sections, Linear Regression and Multi-layer Perceptrons will be discussed. Regardless of the chosen methodology, estimating parameters $\boldsymbol{\beta}$ is the most complex phase, requiring optimal data management (**Model Selection**) and various error minimization theories. All this will be discussed more specifically in the following chapters.

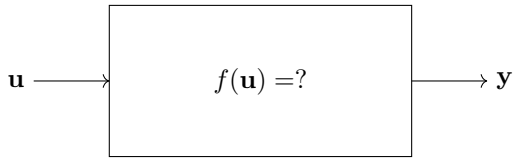


Figure 1.2: Black Boxe system

A problem can be classified as **Black box** or **Gray box**, depending on how much information we have about the system itself. If the relationship between inputs and outputs is known, the system is classified as Gray box. In this particular case, we will consider and study a **black box** system because the relationship is unknown, and we must rely on the measurements $(\bar{u}, \bar{y})$. This means that if we can obtain some information about the system, it can transition from being classified as gray to black, or vice versa.

1.3 Data Processing and Model Selection

Before delving into system analysis techniques through regression methods, it's essential to identify and define the data you'll be working with.
Having clean data makes the study and parameter determination more efficient in terms of both time and accuracy. Applying error minimization algorithms alone is not enough; a clean base is required.

Model selection and **preprocessing** are fundamental phases in data analysis and model training, directly impacting the quality and performance of models. This phase involves preparing and cleaning the data so that it can be effectively used in machine learning models.
Operations performed in the **preprocessing** phase include eliminating insignificant or missing data, managing outliers, or normalizing the dataset.

Listing 1.1: Function that removes NaN values

```
def clean(y, X):
    S = np.sum(X, axis=1)
    bad = np.isnan(S)
    X = np.delete(X, bad, axis=0)
    y = np.delete(y, bad)
    bad = np.isnan(y)
    X = np.delete(X, bad, axis=0)
    y = np.delete(y, bad)
    return (y, X)
```

Model selection techniques are classified into probabilistic and resampling, simulating model performance through probabilistic and statistical parameters (Chapter: Performance Metrics).

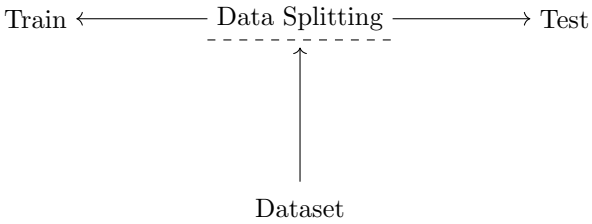


Figure 1.3: Data Splitting: Train and Test

The dataset is partitioned into **train** and **test** sections. A data selection technique used in this case is **data splitting**, involving the random partitioning of the dataset into *train* and *test*. The model is determined on the *test*, and

the results are compared.

Listing 1.2: Extract from the program used for processing and model selection of the yTtrain and yVtrain datasets

```
    # DATAFRAME SPLITTING INTO VALIDATION-TRAIN VALUE
# split the dataframe into train and validation
   sections
yT_train = read_csv(name_y_file_train)
yV_train = np.array(yT_train.iloc[:, colY])  # extract
    the column related to pm10 concentration values
   from the file IT0187A_PM10_day.xlsx

# create a matrix with tempmin and temp values to use
   as training data
uT_train = read_csv(name_u_file_train)
uV_train = np.matrix(uT_train.iloc[:, colU])

yT_val = read_csv(name_y_file_val)
yV_val = np.array(yT_val.iloc[:, colY])

uT_val = read_csv(name_u_file_val)
uV_val = np.matrix(uT_val.iloc[:, colU])

# y_train, X_train = dataset_shift(yV_train, uV_train,
    ar_ord, ex_ord, ex_del)
y_train_clean, X_train_clean = clean(y_train, X_train)
# y_val, X_val = dataset_shift(yV_val, uV_val, ar_ord,
    ex_ord, ex_del)
y_val_clean, X_val_clean = clean(y_val, X_val)
```

Similarly, another data-splitting technique called **k-cross validation** involves dividing the dataset into *train* and *test*, where the *test* partition is used to determine the approximations. The *train* partition is further divided into k partitions, each approximating the model parameters separately. The results are then averaged on the *train* and compared with those of the *test*.

Depending on the type of model considered (linear, NN, etc.), data processing is required according to the model's requirements. In neural networks (Chapter: Multi-layer Perceptron), for example, a probabilistic estimate is provided, and the output will be between $0 < y < 1$. Similarly, the inputs are considered probabilities and must adhere to the property of y. Normalization can be performed according to various parameters. In this project, normalization with respect to

the maximum value was chosen.

$$\boldsymbol{X} = \frac{\boldsymbol{X}}{x_{\max}}$$

Listing 1.3: Used the `normalize` function provided by the `sklearn preprocessing` library

```
from sklearn import preprocessing
X_train_clean_nn = preprocessing.normalize(
    X_train_clean, norm='max')
y_train_clean_nn = preprocessing.normalize([
    y_train_clean], norm='max')
y_train_clean_nn = np.transpose(y_train_clean_nn)
X_val_clean_nn = preprocessing.normalize(X_val_clean,
    norm='max')
y_val_clean_nn = preprocessing.normalize([y_val_clean
    ], norm='max')
y_val_clean_nn = np.transpose(y_val_clean_nn)
```

Chapter 2

Autoregressive and Exougenous systems

2.1 ARX Models with Exogenous Inputs

ARX models are presented in the form

$$\begin{aligned} A(z) &= 1 - (\alpha_1 z^{-1} + \cdots + \alpha_n z^{-n}) \\ B(z) &= \beta_0 + \beta_1 z^{-1} + \cdots + \beta_m \\ A(z)y(t) &= B(z)u(t) \end{aligned} \tag{2.1}$$

where the operator z^{-i} is the time-shift operator in discrete-time.
Expanding the vector product in 2.1, we obtain

$$y(t) = \alpha_1 y(t-1) + \cdots + \alpha_n y(t-n) + \beta_0 u(t) + \cdots + \beta_m u(t-m) \tag{2.2}$$

where n is the order of the autoregressive part, and m is the order of the exogenous part. The **exogenous** variables "$\boldsymbol{u}$" are considered independent input variables, while the **autoregressive** variables "$\boldsymbol{y}$" depend on their previous states in time ($y(t-i)$ with $1 \leq i \leq n$).
Let's construct a matrix $\boldsymbol{M}$ and the coefficient matrix $\boldsymbol{\theta}$ such that

$$M(t) = \big(y(t-1) \dots y(t-n)\big)$$

$$\theta = \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \\ \beta_0 \\ \vdots \\ \beta_m \end{pmatrix} \tag{2.3}$$

$$y(t) = M(t) \times \theta \tag{2.4}$$

In a **black box** problem (introduced in Chapter 1), we can work in either *prediction* or *simulation*. Estimating the parameters of a model in **prediction** assumes knowing every state of the system up to n before the one being estimated. Calculating in **simulation** means starting with knowledge of the model and estimating subsequent states using the estimated states; this option involves greater error propagation.
Let's assume it is possible to measure the system outputs. Then, we distinguish in the matrix M between the measured and predicted parts:

$$\hat{y}(t) = M(t) \times \theta = \begin{cases} \hat{M}(t) \times \theta & \text{predicted} \\ \bar{M}(t) \times \theta & \text{measured} \end{cases} \tag{2.5}$$

Following Theorem 1.2 (Chapter 1), a metric D, called distance (which in Chapter 4 will be identified as the *cost function* $f(x)$), is fixed. It indicates the distance between the estimated output $\hat{y}$ and the measured output $\bar{y}$. This distance must be minimized to obtain the most optimal parameters.

Example 1 (Prediction and Simulation)**.** *Suppose the ARX model is in the form*

$$y(t) = \alpha_1 y(t-1) + \beta_0 u(t) + \beta_1 u(t-1)$$

and the measured output $\bar{y}$ for the input $\bar{u}$ at time t is:

t	$\bar{y}$	$\bar{u}$
0	1	1
1	1	3
2	2	5
3	3	7
4	5	9
5	8	11
6	13	12

*Construct the table applying the calculation in **prediction** to be able to apply least squares:*

t	$\bar{y}(t)$	$\bar{y}(t-1)$	$\bar{u}(t)$	$\bar{u}(t-1)$
0	1	—	1	—
1	1	1	3	1
2	2	1	5	3
3	3	2	7	5
4	5	3	9	7
5	8	5	11	9
6	13	8	12	11

The dashed table corresponds to $\bar{M} = [M_y M_u]$.
*In **simulation**, it turns out to be*

2.2 Least Squares Method

Let's use some concepts expanded in the chapter 3. Define the distance function as

$$f(x) = D(\hat{y}, \bar{y}) = \sum_t \left(\hat{y}_\theta(t) - \bar{y}(t)\right)^2$$

seeking parameters $\theta = (\alpha_1 \dots \alpha_n \beta_0 \dots \beta_m)$ that minimize $D(\hat{y}, \bar{y})$

$$argmin_\theta D(\hat{y}, \bar{y}) = argmin_\theta \sum_t \left(\hat{y}_\theta(t) - \bar{y}(t)\right)^2$$

of the system

$$y(t) = \left[\hat{M}\bar{M}\right] \times \theta$$

The matrix of inputs and outputs can be rewritten in the form 2.5, explicitly showing the measured and predicted parts

$$\begin{aligned} \hat{y}_\theta(t) - \bar{y}(t) = \hat{M} \times \theta - \bar{y} = \left(\hat{M}_y \times \vec{\alpha} + \bar{M}_u \times \vec{\beta}\right) = \\ = M^T M\theta = M^T Y \\ \theta = (M^T M)^{-1} M^T Y \end{aligned} \tag{2.6}$$

(Take a look at eq3.5)

2.3 Coding

Let's define some generic functions that implement the construction from matrix M by shifting the data based on the order of the autoregressive and exogenous parts. We call:

- `y`: matrix/vector of measured values
- `U`: matrix/vector of inputs (measured)
- `na`: order of the autoregressive part
- `nb`: order of the exogenous part
- `nk`: order of the time delay between output and input

$$y(t) = \alpha_1 y(t-1) + \dots + \alpha_{\mathbf{na}} y(t - \mathbf{na}) + \beta_0 u(t - \mathbf{nk}) + \dots + \beta_{\mathbf{nb}} u(t - \mathbf{nb} - \mathbf{nk})$$

ipothesis: there is no delay between inputs and outputs, so **nk**=0.

Listing 2.1: This function computes the shifting of dataset based on the degree of ARX parameter

```
def dataset_shift (y,U,na,nb,nk):
        inizio_out,inizio_ar,inizio_ex,
           inizio_out_traslato,inizio_ar_traslato,
           inizio_ex_traslato=start_shift(na,nb,nk)
        yout=y[int(inizio_out_traslato):]
        L=len(yout)
        X=np.zeros([L,na+np.sum(nb)])
        cont=0
        if len(inizio_ex_traslato):
                for i in range(len(nb)):
                        for j in range(nb[i]):
                                X[:,cont]=U[int(
                                   inizio_ex_traslato[
                                   i,j]):int(
                                   inizio_ex_traslato[
                                   i,j])+L,i].flatten
                                   ()
                                cont=cont+1

        if len(inizio_ar_traslato):
                for j in range(na):
                        X[:,cont]=y[int(
                           inizio_ar_traslato[j]):int(
                           inizio_ar_traslato[j]+L)]
                        cont=cont+1

        X=np.array(X)
        return(yout,X)
```

Listing 2.2: This or funtion create a mask used from the previous method. The "mask" created tells who are the first significant datas (look at Example 1); receives na, and the two vectors nb and nk,also receives two vectors y and u (outputs and inputs).Returns matrix M and vector theta

```
def start_shift(na,nb,nk):
    min_u=0
    min_a=0
    if na>0:
        inizio_ar=np.array(range(0,-na,-1))-1
                min_a=np.min(inizio_ar)
    else:
                inizio_ar=[]
    if type(nb) is int:
                nb=np.append(nb,np.nan)
                nk=np.append(nk,np.nan)
                num_ex=1
    else:
                num_ex=len(nb)
    if len(nb)==0 or np.nanmax(nb)<=0:
                inizio_ex=[]
        else:
                inizio_ex=np.full([num_ex,int(np.
                   nanmax(nb))],np.nan)
                for i in range(0,num_ex):
                        max_ord=int(nb[i])
                        max_del=int(nk[i])
                        inizio_ex[i,0:max_ord]=np.
                           array(range(0-max_del,-
                           max_ord-max_del,-1))
                min_u=np.nanmin(inizio_ex)
    minimo=np.min([min_a,min_u])
    if len(inizio_ar):
                inizio_ar_traslato=inizio_ar-minimo
    else:
                inizio_ar_traslato=[]
    if len(inizio_ex):
                inizio_ex_traslato=inizio_ex-minimo
    else:
                inizio_ex_traslato=[]
    inizio_out=0
    inizio_out_traslato=-minimo

return(inizio_out,inizio_ar,inizio_ex,
   inizio_out_traslato,inizio_ar_traslato,
   inizio_ex_traslato)
```

Chapter 3

Linear Regression

Machine learning revolves around optimization, which requires the selection of a model and parameters that allow fitting the response in the most plausible manner.
This is the purpose of all curve fitting techniques. The most well-known curve fitting techniques in the context of machine learning are regression or polynomial fitting techniques. Below, a broader view of the curve fitting concept will be provided, delving more into detail on linear regression, a technique utilized for the project.

3.1 Introduction

Regression aims to estimate the relationship between independent variables $\mathbf{X}$ and dependent variables $\mathbf{Y}$, along with unknown parameters $\boldsymbol{\omega}$:

$$\mathbf{Y} = f(\mathbf{X}, \boldsymbol{\omega})$$

When optimizing the goodness-of-fit of a function, the goal is to find the parameters $\boldsymbol{\omega}$.

3.2 Linear Regression with Least-Squares Fitting Methods

Linearization involves fitting points in the plane or space to a line, and fitting is performed using various techniques. In this section, we will analyze the method of least squares.
Consider n points

$$(x_1, y_1), \ldots, (x_i, y_i), \ldots, (x_n, y_n)$$

We want to construct a function $\tilde{f}(x) = \sum_i^m \beta_i \phi_i(x)$ with $m = 2$, $\phi(x)$ being a *shape function* such that

$$\tilde{f}(x) = \omega_1 x + \omega_2$$

and $\tilde{f}(x_i) = y_i$, implying

$$y_i = \omega_1 x_i + \omega_2$$

Let B be the matrix:

$$B = \begin{pmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix}$$

where ω_1, ω_2 are the parameters of the vector $\boldsymbol{\omega}$ chosen to minimize the error associated with the fit.

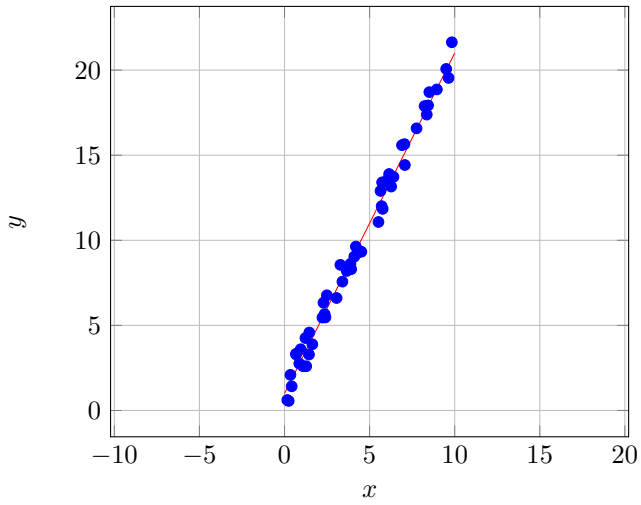


Figure 3.1: Linear regression of a set of non-aligned points

The points are not aligned, and thus, a linear relationship does not connect them. We need to construct a line that minimizes the mean squared error:

$$LS(f) = E_2(f) = \left(\sum_i^n |f(x_i) - y_i|^2 \right) = \left(\sum_i^n |\omega_1 x + \omega_2 - y_i|^2 \right) \tag{3.1}$$

To minimize the sum:

$$\phi(\omega_1, \omega_2) = \sum_i^n |f(x_i) - y_i|^2$$

we need to differentiate with respect to the constants to be determined, ω_1 and ω_2, such that

$$\begin{cases} \dfrac{\partial \phi}{\partial \omega_1} = 0 \\ \dfrac{\partial \phi}{\partial \omega_2} = 0 \end{cases} \tag{3.2}$$

Expanding the partial derivatives and organizing the equations, the problem reduces to solving a linear system $\boldsymbol{Ax} = \boldsymbol{B}$

$$\begin{cases} \dfrac{\partial \phi}{\partial \omega_1} = 2\sum_i^n (y_i - w_1 x - w_2)(-x_i) = 0 \\ \dfrac{\partial \phi}{\partial \omega_2} = 2\sum_i^n (y_i - w_1 x - w_2)(-1) = 0 \end{cases} \tag{3.3}$$

$$\begin{cases} \dfrac{\partial \phi}{\partial \omega_1} = (\sum_i^n x_i^2)\omega_1 + (\sum_i^n x_i)\omega_2 = (\sum_i^n y_i x_i) \\ \dfrac{\partial \phi}{\partial \omega_2} = (\sum_i^n x_i)\omega_1 + \omega_2 = (\sum_i^n y_i) \end{cases}$$

Transitioning from a system with $B = \begin{pmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix}$ (size $n \times 2$) to a 2x2 system:

$$\begin{pmatrix} (\sum_i^n x_i^2) & (\sum_i^n x_i) \\ (\sum_i^n x_i)\omega_1 & n \end{pmatrix} \begin{pmatrix} \omega_1 \\ \omega_2 \end{pmatrix} = \begin{pmatrix} (\sum_i^n y_i x_i) \\ (\sum_i^n y_i) \end{pmatrix} \tag{3.4}$$

$$\begin{pmatrix} x_1 & \dots & x_n \\ 1 & \dots & 1 \end{pmatrix} \begin{pmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} \omega_1 \\ \omega_2 \end{pmatrix} = \begin{pmatrix} x_1 & \dots & x_n \\ 1 & \dots & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

Expressible as

$$B^T B\boldsymbol{\omega} = B^T \mathbf{y}$$

where $B^T B$ is a 2×2 matrix.

The vector $\boldsymbol{\omega}$ is found using the pseudo-inverse of $B^T B$:

$$\boldsymbol{\omega} = (B^T B)^{-1} B^T \boldsymbol{y} \tag{3.5}$$

3.3 Coding

`Sklearn` provides a library for using linear regression models.

```
from sklearn.linear_model import LinearRegression
model=LinearRegression()
model.fit(X_train_clean,y_train_clean)
```

Once created a model `LinearRegression()` we are ready to specify on which variable fit this model by the command `model.fit(var1, var2)`; the method do all mathemathics shown before by itself.

Chapter 4

Artificial Neural Networks

In this chapter, we will strive to comprehend the functioning of Multi-Layer Perceptrons (MLP), commonly known as Neural Networks. We will meticulously study each aspect, starting from the perceptron and progressing to more intricate neural networks.

4.1 The Perceptron

The perceptron algorithm is grounded in the concept of simulating a single neuron in the human brain. A human neuron performs a straightforward task: receiving inputs and activating if the inputs surpass a certain threshold, transmitting the signal to the next neuron. The perceptron emulates this process by utilizing input data as the neuron input, with weights representing the connection strength between input neurons and the output neuron.
However, limitations were identified in the perceptron algorithm, particularly its incapability to learn certain types of non-linear patterns.
The perceptron acts as a linear binary classifier, capable of classifying linearly separable data. As a binary classifier, it determines whether a data point belongs to a specific class, analogous to a binary "Yes" or "No."

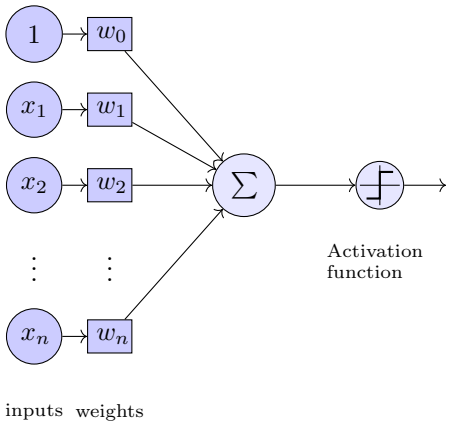


Figure 4.1: Perceptron

As illustrated in Figure 2.1, the perceptron comprises input links ($X1, X2, X3$) with corresponding weights ($W1, W2, W3$). These weights are the core elements determining the strength of each input signal.
The perceptron computes the weighted sum of inputs ($z = \sum ix_i w_i$), and these sums are passed through an activation function, also known as the step function. The activation function determines whether the perceptron needs activation.
The final output is determined by the activation function (see Activation Functions section) based on the state of the sum z. For instance, using a Heaviside step function $\mathcal{H}(z)$

$$\mathcal{H}(z) = \begin{cases} 0 & \text{se } z < 0 \\ 1 & \text{se } z \geq 0 \end{cases} \tag{4.1}$$

4.2 Training Perceptrons

The perceptron is trained by identifying algorithmic errors and adjusting weights to minimize this error. This aligns with the primary goal of training machine learning algorithms.
The algorithm predicts based on given training data, compares predictions with actual outputs, calculates errors for each instance, and reduces these errors during training.
Weights corresponding to correct predictions are adjusted accordingly.

$$w_{n+1} = w_n + \eta(y_n - \hat{y}) \tag{4.2}$$

(To understand this equation, refer to the Gradient Descent chapter and the Backpropagation section). Where η is the **learning rate**, qnd

$$y_n = \phi\left(\sum x_i * w_i + b\right)$$

ϕ is the activation function and b is the *bias*) The bias is an additional necessary input as one of the perceptron's inputs is null. As long as this input and the output y_n are null, the bias serves to shift activation during the perceptron's learning process.

The learning rate regulates the size of steps the model takes during training, influencing how quickly or slowly the model learns from data. A too high learning rate may cause model oscillation without convergence, while a too low one may lead to slow convergence or getting stuck in local minima.

Choosing the learning rate is a compromise between rapid convergence and training stability.

4.3 Multi Layer Perceptron

While the perceptron serves well in simple tasks, real-world problems often demand more nuanced solutions. Recognizing the limitations of a single-layer perceptron in handling complex patterns, researchers introduced the concept of a Multi-Layer Perceptron (MLP). The MLP represents a significant advancement, incorporating multiple layers of interconnected neurons, allowing for the model to capture intricate relationships within the data.
In this chapter we'll be concerned about feed-forward NN.

4.4 From Perceptron to MLP

By combining multiple units (perceptrons), we create Multi-Layer Perceptrons (MLPs). In this chapter, we will delve into the fundamentals of *feed-forward networks*, which serve as a gateway to more intricate structures such as *recurrent* and beyond.

Feed-forward networks exhibit a configuration where basic units are organized in layers, and there are no feedback loops closing the graph. The simplest form of MLP is depicted in the figure, where the initial layer is the input layer, the final layer is the output layer, and any intermediary layer(s) is/are referred to as the Hidden Layer(s).
The depth of a network is determined by the number of layers, while the width is defined by the number of units within each layer.

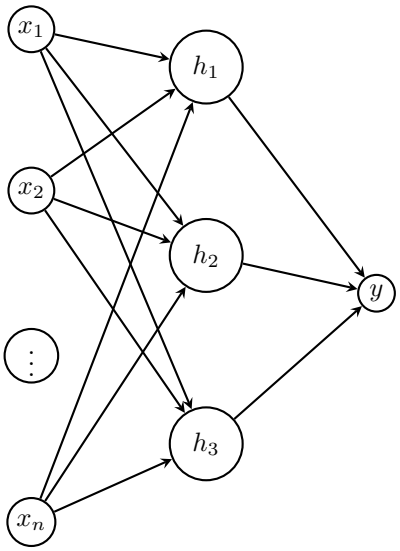


Figure 4.2: MLP with one hidden layer

4.5 Training MLPs

All units in the network are perceptrons; thus, we extend the behavior to all units comprising the network.

> **Notation**: As with the linear case, we'll refer to the activations of the input units as xj and the activation of the output unit as y. The units i_{th} in the l_{th} hidden layer will be denoted h_i^l.

$$\begin{aligned} h_i^{(1)} &= \phi^{(1)}(\sum x_i * w_{ij} + b_i^1) \\ h_i^{(2)} &= \phi^{(2)}(\sum h_i * w_{ij} + b_i^2) \\ y_i &= \phi^{(l)}(\sum x_i * w_{ij} + b_i^{(l)}) \end{aligned} \tag{4.3}$$

It is necessary to specify the superscript for the activation functions ϕ since it may vary from unit to unit.
Having specified the behavior of each perceptron inside and outside the hidden layers, we can move on to vector notation. This notation allows us to consider every interaction of perceptrons in the hidden layer with those from the previous layer.

$$\begin{aligned} \mathbf{h}^{(1)} &= \phi^{(1)}(\sum \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^1) \\ \mathbf{h}^{(2)} &= \phi^{(2)}(\sum \mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^2) \\ \mathbf{y} &= \phi^{(3)}(\sum \mathbf{W}^{(l)}\mathbf{h}^{(2)} + \mathbf{b}^l) \end{aligned} \tag{4.4}$$

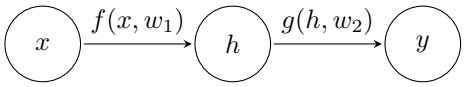
Since there is a weight for every pair of units in two consecutive layers, we represent each layer's weights with a weight matrix **W**. Each layer also has a bias vector **b**.
The greater the complexity of the network, the more challenging it becomes to determine an accurate approximation of weights and biases, leading to an increased expression of error.
In this regard, specialized algorithms are employed to optimize the search for parameters that minimize the error function ε, irrespective of the network's structure. The most well-known algorithm for this purpose is the **Gradient Descent**.

$$w_{n+1} = w_n - \gamma \nabla f(w_n)$$

explored in more detail in chapter 6. Since the computation of the gradient is often a computationally expensive operation, it is necessary to optimize it, and this is achieved through the *backpropagation* algorithm.

4.5.1 Backpropagation

This algorithm optimizes the computation of the gradient and, consequently, determines the weights of the network. This algorithm specifically computes the gradient of the error, which will then be utilized by the gradient descent.



$$y = g(f(x, w_1), w_2)$$

Figure 4.3: Neural network with one input node, one hidden layer, and one output node.

Now let's analyze the simplest scenario, considering an MLP network with one input node, one hidden layer, and one output node; the relationship between input x and output y is given by

$$y = g(f(x, w_1), w_2)$$

Given the functions $f(\cdot)$ and $g(\cdot)$, the error produced by the network, in accordance with the principle of least squares, will be

$$E = \frac{1}{2}(y - \hat{y})^2$$

where y is the correct output value, and $\hat{y}$ is the estimated output. The goal is to determine w_1 and w_2 to minimize the error E, which requires the application

of the ***chain rule*** (for partial derivatives)

$$\frac{\partial E}{\partial w_1} = -(y - \hat{y})\frac{\partial y}{\partial h}\frac{\partial h}{\partial w_1} \tag{4.5}$$

Back-prop results in an iterative, gradient descent update rule as cited before in that chapter:

$$\begin{aligned} w_1^{n+1} &= w_1^{(n)} - \gamma \frac{\partial E}{\partial w_1^{(n)}} \\ w_2^{n+1} &= w_2^{(n)} - \gamma \frac{\partial E}{\partial w_2^{(n)}} \end{aligned} \tag{4.6}$$

Example 2. *Suppose now that $f(\cdot)$ and $g(\cdot)$ are two ReLU activation functions*

$$f(i, w) = g(i, w) = i * w$$

in 4.3 we have

$$z = w_1 x$$

$$y = w_2 z$$

Applying the gradient as shown in eq. 4.5:

$$\frac{\partial E}{\partial w_1} = -(y - \hat{y})\frac{\partial y}{\partial z}\frac{\partial z}{\partial w_1} = -(y - \hat{y})w_2 x$$

$$\frac{\partial E}{\partial w_2} = -(y - \hat{y})\frac{\partial y}{\partial z}\frac{\partial z}{\partial w_2} = -(y - \hat{y})w_1 x$$

and the weights w_n are updated according to eq. 4.6

Generalizing, the backpropagation for a deeper network with L hidden layers labeled $h_1, \ldots, h_l$ with the first connection weight w_1 between x and h_1 (as in fig. 4.3) is given by:

$$\frac{\partial E}{\partial w_1} = -(y - \hat{y})\frac{\partial y}{\partial h_l}\frac{\partial h_l}{\partial h_{l-1}}\frac{h_{l-1}}{h_{l-2}} \cdots \frac{h_1}{w_1}$$

4.6 Coding

`Sklearn` provides a library for using and configuring a MLP network

```
#NEURAL NETWORK MODEL
from sklearn.neural_network import MLPRegressor
    model_nn = MLPRegressor(hidden_layer_sizes=(50,20,)
    , solver='sgd', max_iter=100, activation = 'tanh',
    learning_rate_init=0.001).fit(X_train_clean_nn,
    y_train_clean_nn)
```

The constructor follow the criterion according to which for n inputs , neurons in hidden layers must be within $[\frac{n}{2}, 20n]$

Chapter 5

Activation Functions

5.1 The importance of Activation Function

An **Activation Function** decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations. The activation function introduce non-linearity to the perceptron or MML behaviour. It is foundamental to have non linearity because if not the output of every Perceptron is a linear trasformation on inputs using weights and biases

$$y_n = \sum(x_i w_i) + b$$

(as shown in the previuous chapter), ogni rete neurale quindi si comporterà sempre allo stesso modo perchè componendo funzioni lineari otterremo sempre altre funzioni lineari, riducendo cosi la complessità dei task apprendibili dalla rete.

5.2 A quick look of Activation functions

Activation functions are functions used in Neural Network to compute the weighted sum of inputs and biases, which limits the permissible amplitude of outputs in a limited range . It manipulates the presented data through some gradient processing usually **gradient descent** (capitolo Gradient Descent) to produce an output for the NN, that contains the parameters in the data. In this section will have a quick look to the most used AFs for non-linear problems:

- **Rectified Linear Unit (ReLu) function**
- **Sigmoid**
- **Hyperbolic Tangent Function**

5.2.1 Rectified Linear Unit function (ReLu)

The ReLU AF is among the most widely used AFs.It offers better performance and generalization in DL compared to other AFs [34], [35]. The ReLU represents a nearly linear function but has derivates function, and therefore preserves the properties of linear models that made them easy to optimize, with gradientdescent methods [4]. The ReLU AF performs a threshold operation to each input element where values less than zero are set to zero thus the ReLU is given by

$$f(x) = max(0, x) = \begin{cases} x_i & x_i \geq 0 \\ 0 & x_i < 0 \end{cases} \tag{5.1}$$

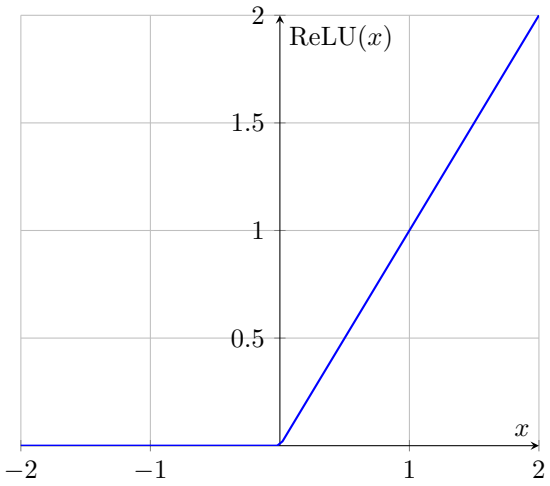


Figure 5.1: ReLu function

ReLu is computationally efficient and accelerates the convergence of gradient descent due to his linear properties.
The negative side of this functions is that due to its derivate

$$f'(x) = \begin{cases} 1 & x_i \geq 0 \\ 0 & x_i < 0 \end{cases}$$

the weight and the biases of some perceptron are not updated reducing the models ability to train and fit properly datas.

5.2.2 Sigmoid

The Sigmoid is a non-linear AF used mostly in feedforward NNs. It is a bounded differentiable real function, defined for real input value with positive derivatives everywhere. The Sigmoid function is given by

$$f(x) = \frac{1}{1 + e^{-x}} \tag{5.2}$$

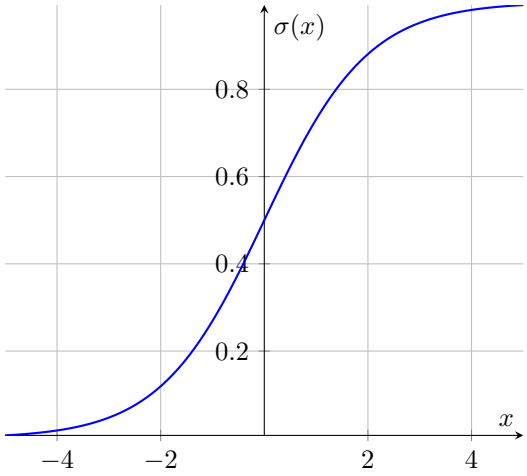


Figure 5.2: Sigmoid

Since the output is limited in a range of $(0, 1)$ this AF is most commonly used when the output is a probability, which in its definition is a value ranged from 0 to 1.

5.2.3 Hyperbolic Tangent Function

The tanh is another AF used in ML, it is similar to the sigmoi, in this case the more positive the inputs are the closer the output will be to 1, otherwhise the more negative they are the closer the output will be to -1 . This is given by the equation

$$f(x) = \frac{sinh}{cosh} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5.3}$$

Its use is prefered from the sigmoid function but it present the same problem like the **saturation** or the **vanishin gradient** (*Section 2.4*).
This is a centered function so it comes usefull when the datas has to be centered aroud a mean values, it is commonly used in the hidden layers so the training will be much easier.

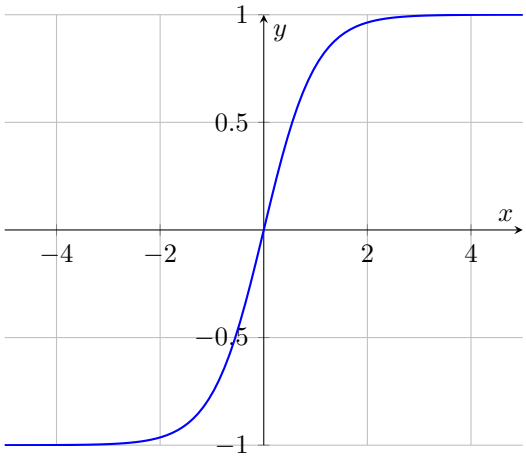
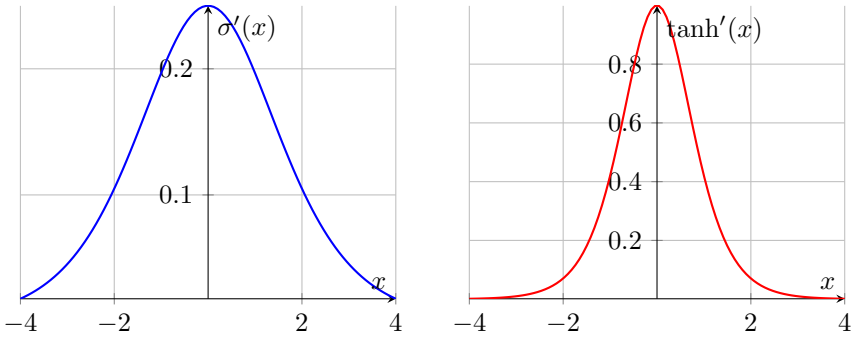


Figure 5.3: tanh

5.3 The problem of Saturation and the Vanishing Gradient

A low gradient of an activation function, particularly in the context of neural networks, can lead to training difficulties and is often considered undesirable. When the gradient of the activation function is very small, it can lead to the **vanishing gradient** problem. This problem occurs during backpropagation can cause the weights in the early layers to be updated very slowly or not at all. As a result, those layers may not effectively learn from the training data.

In some activation functions, particularly those like the sigmoid function or hyperbolic tangent (tanh), the gradients become very small for extreme input values. This is known as **saturation**.



(a) Derivative of the sigmoid function (b) Derivative of the tanh function

Figure 5.4: Derivatives of Sigmoid and Tanh Functions

When neurons become saturated, they effectively "turn off" during training, contributing little to the learning process. This can slow down or even halt the learning process for certain parts of the network.

Chapter 6

Optimisation Algorithsm for local minimum

In Machine Learning, it is necessary to define the weights and biases of a Perceptron, which are updated whether the network follows a feed-forward or back-propagation approach, as expressed by the formula:

$$w_{n+1} = w_n - \eta(y_n - \hat{y})$$

To derive the corrected weight w_{n+1}, it is essential to determine the value $\hat{y}$ that minimizes $y - \hat{y}$. Minimizing the error leads to a more accurate prediction. The primary algorithms used for minimizing a cost function are **Gradient Descent** and **Stochastic Gradient Descent**.
The optimization problem takes the form:

$$\min_x f(x)$$

where

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

is differentiable with $\mathbf{dom}(f)$=$\mathbb{R}$. The goal is to find $f^* = \min_x f(x)$, and the optimal solution is $x^* = \arg\min_x f(x)$.
Instead of solving the often complex system $\nabla f = 0$ to find this solution, an iterative scheme is employed. It computes a minimizing sequence of points $x^0, x^1, \ldots, x^n$ as $n \to \infty$, ensuring $f(x^n) \to f(x^*)$. The algorithm terminates when the residuals between the current state and the optimal value fall within an acceptable tolerance ε:

$$f(x^n) - f^* < \varepsilon$$

6.1 Gradient Descent

Gradient Descent is one of the most used optimisation algorithm for determine local minumum in $\mathbb{R}^n$ space, instead other algorithms like *linear regression*, *logic*

regression or *t-SNE*, which each application is limited in limited fiels.
Starting from an initial point y_0,we obtain approximations which over time, as the iterations increase, must tend towards the solution we are looking for,obtaining approximations of the argument which minimize $f(x)$ to which we can refers to as *cost function* or *loss function*.

$$y_0 \in \mathbb{R}$$

$$x^* = argmin_x f(x)$$

x^* is obtained as:

$$x_{n+1} = x_n - \gamma \nabla f(x_n)$$

where γ is the step's widht, called **learning rate**. The principle behind Gradient Descent is, given the sum of least squares, i.e., $f(x)$, to calculate its derivative at its points to determine the direction of minimum slope. This means finding the coordinates (the solution) that allow minimizing the error.
Since the gradient is the direction of maximum growth (locally) of the function, and considering the goal is to find the minimum of the function, one moves in the opposite direction **-**$\nabla f(x)$.

A quick example of iteration steps:

```
k = 0
x(0)=0 #c.i
learning_rate = 0.1
while gradient(k)=0
    compute gradient(k)
    descent = -gradient(k)
    x(k+1) = x(k) - learning_rate*descent
    k++
end
```

Gradient descent does few calculations far from the optimal solution and increases the number of calculation when the gradient become lower, closer to the optimal value. This concept is instantiated by the product $\gamma \nabla f$, setting a value of $0 < \gamma < 1$. In fact, the smaller $\nabla f(x_n)$ is, the smaller the step taken by the algorithm will be.
The way the Learning rate change from relatively large to relatively small is called **schedule**

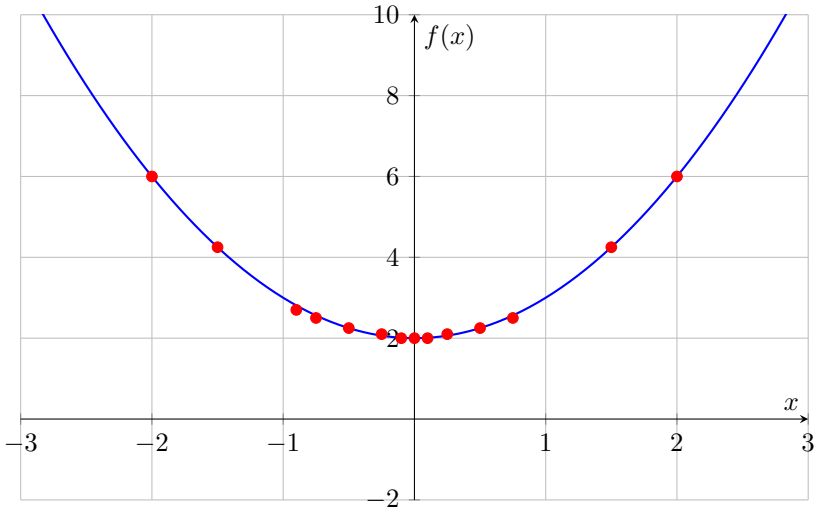


Figure 6.1: $[f : \mathbb{R} \to \mathbb{R}, x \in \mathbb{R}]$ Red dots represent the schedule of learning rate

6.1.1 Gradient descent in multiple dimensions

Since a system generally has multiple inputs $(x^0, x^1, \ldots, x^n)$ In many cases, the form of the function is often unknown, and thus, we cannot readily determine the local minimum. This is why pseudo-random samples are considered, following the logic introduced earlier.

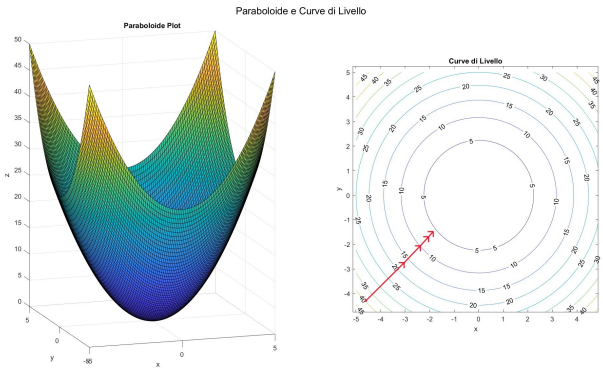


Figure 6.2: $\mathbb{R}^3 \to \mathbb{R}$ Paraboloid and level curves

The gradient $\nabla f(x_n)$ represents the vector orthogonal to the level curves. Starting from the initial condition $\vec{x_0}$, the module of the vector decreases as we approach the optimal solution, indicating that the step size diminishes progressively in accordance with the previous theory.

When the dataset is too heavy this process can take a large amount of time making the process of training slower. In this case is more likely to use **stochastic gradient descent**

6.2 Stochastic Gradient Descent

Unlike GD, SGD determines the local minimum by randomly selecting the data sample for gradient measurement instead of choosing it progressively. This new algorithm addresses the slow convergence issue and finds its maximum utility in scenarios where datasets are excessively large or exhibit redundancies.

$$x^k = x^{k-1} - t_k \nabla f_{i_k}(x^{k-1}) \tag{6.1}$$

k=1,2,3,...
$i_k \in (1, \ldots, n)$
$t_k : learningrate$
Similar to the GD algorithm but the loss function f is taken randomly (f_{i_k})

Given the algorithm's emphasis on speed as its key advantage, to enhance parameter estimation stability within a limited number of steps, the **mini-batching** technique is employed. This technique involves utilizing multiple samples for each individual step.

$$x^k = x^{k-1} - \frac{t_k}{b} \sum_{i \in I_k} \nabla f_{i_k}(x^{k-1}) \tag{6.2}$$

where I_k is the mini-batch and b is the number of samples ($|I_k| = b << n$)

6.3 How to avoid Local Minimum

Achieving a local minimum is not sufficient to ensure the minimization of the cost function f; the goal is to reach an absolute minimum of this function. The methods previously presented do not guarantee that the parameters $\boldsymbol{x} = (x_1, ..., x_n) \equiv \boldsymbol{\beta}$ satisfy

$$\boldsymbol{x} \implies \min f(\boldsymbol{x})$$

To avoid this issue, a rather brute-force technique that can be adopted is to iterate the algorithm k times with different initial conditions y_0, comparing the obtained results $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^k$ and selecting the optimal result $\boldsymbol{x}^i$.

Chapter 7

Performance Metrics in Machine Learning

Performance metrics are essential components of machine learning pipelines, offering quantifiable assessments of model performance during training and testing. Whether dealing with basic linear regression or advanced techniques, the need for a performance metric is fundamental.

In machine learning tasks like Regression and Classification, specific performance metrics evaluate different aspects of model predictions. This discussion focuses on key metrics and their informative value regarding model performance.

It's crucial to distinguish between **metrics** and **loss functions**.
Loss functions, used during training with optimization techniques like Gradient Descent, measure model performance and are typically differentiable concerning the model's parameters.
Metrics, on the other hand, serve to monitor and measure performance during training and testing without the need for differentiability.

In cases where a performance metric is differentiable, it can potentially be employed as a loss function, possibly incorporating additional regularization (e.g., Mean Squared Error - MSE).

In this chapter we'll have a look on some metrics used for regression.

- mean-absolute error
- mean-square error
- root-mean-square error
- Coefficient of determination

Notation: in the next sections will be used y as the real value of the function and $\hat{y}$ as the predicted value, i the current data and n the number of data.

7.1 MAE [mean-absolute error]

Mathematically, the Mean Absolute Error is calculated as the average of the absolute differences between the predicted values (or model outputs) and the actual values (ground truth) of the target variable.

$$MAE = \frac{1}{n}\sum_{i}^{n}|y_i - \hat{y}_i| \tag{7.1}$$

Pros

- **Robust to Outliers**: MAE is less sensitive to outliers compared to MSE, as errors are treated linearly.
- **Interpretability**: MAE's error is measured directly in the same units as the response variable, making the evaluation of model performance more interpretable.

Cons

- **Sensitivity to Medium Errors**: MAE may underestimate the severity of larger errors, as it does not amplify them as MSE does. This can be problematic in situations where large errors are significant.

7.2 MSE [mean-square error]

Mathematically, the Mean Squared Error is calculated as the average of the squared differences between the predicted values (or model outputs) and the actual values (ground truth) of the target variable.

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2 \tag{7.2}$$

Pros

- **Sensitive to Large Errors**: MSE heavily penalizes larger errors, meaning the model is strongly influenced by any predictions significantly deviating from the expected values.
- **Differentiability**: The MSE error function is differentiable everywhere, facilitating the use of gradient-based optimization algorithms such as Gradient Descent.

Cons:

- **Outlier Sensitivity**: The high penalty for squared errors can make MSE more sensitive to outliers, significantly impacting model performance.
- **Higher Measurement Units**: The error is measured in squared units of the response variable, which might not be intuitively interpretable.

7.2.1 MAE vs MSE

When to Choose MSE:

- When you want to heavily penalize larger errors.
- When using gradient-based optimization algorithms.

When to Choose MAE:

- When robustness to outliers is crucial.
- When preferring an error measure that is interpretable in the same units as the response variable.

In general, the choice between MSE and MAE depends on the specific requirements of the problem and the tolerance for prediction errors. If handling outliers is critical, and an intuitively interpretable error measure is desired, MAE may be the preferred choice. However, if emphasizing prediction accuracy and using gradient-based algorithms, MSE might be more suitable.

7.3 RMSE [root-mean-square error]

RMSE is a variant of the Mean Squared Error (MSE), and it is widely used as an evaluation metric in regression problems. The RMSE is calculated by taking the square root of the average of the squared differences between predicted and actual values

$$RMSE = \left(\frac{1}{n} \sum_{i}^{n} (y_i - \hat{y}_i)^2 \right)^{\frac{1}{2}} \tag{7.3}$$

Pros:

- **Sensitivity to Errors**: RMSE retains the sensitivity to larger errors inherent in MSE, making it useful for penalizing significant deviations in predictions.
- **Same Unit as Response Variable**: Like MAE, RMSE is measured in the same units as the response variable, enhancing interpretability.
- **Mathematical Properties**: RMSE is well-behaved mathematically, being a proper metric with desirable properties.

Cons:

- **Outlier Sensitivity**: RMSE is still sensitive to outliers, as it inherits this characteristic from MSE.
- **Amplification of Large Errors**: Similar to MSE, RMSE amplifies the impact of larger errors due to the squaring operation, which can be a drawback in the presence of outliers.

7.4 R^2 [Coefficient of determination]

R-squared is a statistical measure that represents the proportion of the variance in the dependent variable that is explained by the independent variables in a regression model. In other words, it indicates how well the independent variables explain the variability in the dependent variable. R-squared values range from 0 to 1, with 0 indicating that the model does not explain any variance, and 1 indicating that the model explains all the variance. It is often used in conjunction with other metrics and diagnostic tools for a comprehensive evaluation of a regression model

$$\sigma^2(\hat{Y}) = \sum_{i}^{n} (y - \hat{y})^2$$

$$\sigma^2(line) = \sum_{i}^{n} (y - mean(\hat{y}))^2$$

Where:
$\sigma^2(\hat{Y})$ is the sum of the squares of the differences between the observed values and the values predicted by the model
$\sigma^2(line)$ is the sum of the squares of the differences between the observed values and the mean of the observed values

Percentage of variation described the regression line is

$$\frac{\sigma^2(line)}{\sigma^2(\hat{Y})}$$

and R^2 is defined by

$$1 - \frac{\sigma^2(line)}{\sigma^2(\hat{Y})}$$

Pros:

- **Interpretability**: R-squared is easy to interpret. A higher R-squared value suggests a best meaning the regression was able to capture a higher percentage of the variance in the target variable.
- **Comparative Measure**: R-squared can be used to compare different models. When comparing models with the same dependent variable, the model with a higher R-squared is generally considered a better fit.

- **Global Fit Indicator**: R-squared provides a global measure of how well the model fits the data, which can be useful for assessing the overall performance of the model.

Cons:

- **Misleading in Some Cases**: R-squared can be misleading if the model is overfitting the data. A high R-squared does not necessarily imply a good model if the model is too complex and fits noise in the data.
- **Doesn't Convey Causation**: R-squared indicates correlation, not causation. Even if R-squared is high, it doesn't imply a causal relationship between the variables.
- **Sensitive to Outliers**: R-squared can be sensitive to outliers, and the presence of outliers can significantly impact its value.
- **Inappropriate for Non-Linear Relationships**: R-squared may not be a good measure of fit for models with non-linear relationships, as it is designed for linear regression models.

7.5 Coding

Creation of a `method` that compute all the metrics using `sklearn.metrics`:

```
def valid_perf(y_true,y_pred):
        import sklearn.metrics as mt
        r2=mt.r2_score(y_true, y_pred)
        me=np.average(y_true-y_pred)
        nme=me/np.average(y_true)
        mae=mt.mean_absolute_error(y_true, y_pred)
        mse=mt.mean_squared_error(y_true, y_pred)
        msle=mt.mean_squared_log_error(y_true,
           y_pred)
        mape=mt.mean_absolute_percentage_error(
           y_true, y_pred)
        nmae=mae/np.average((np.abs(y_true)))
        medae=mt.median_absolute_error(y_true,
           y_pred)
        maxe=mt.max_error(y_true, y_pred)
        evar=mt.explained_variance_score(y_true,
           y_pred)
        perf_dict={}
        for variable in ["r2","me","nme","mae","
           mse","msle","mape","nmae","medae","maxe
           ","evar"]:
                perf_dict[variable] = eval(
                   variable)

        return(perf_dict)
```

Chapter 8

Conclusions

In this chapter we'll resume all steps and results obtained describing the system, datas and all methods used for this analysis, with personal considerations and theoric notions from the previous chapters.

8.1 Project's Purpose

The aim is to estimate the performance of multiple models for predicting the concentration of particolar matter PM10 in the air using an approach based on ARX systems.

8.2 ARX model based

ARX models relate independent input variables u to dependent output variables y, as discussed in Chapter 2.

$$y(t) = \alpha_1 y(t-1) + \cdots + \alpha_n y(t-n) + \beta_0 u(t) + \cdots + \beta_m u(t-m)$$

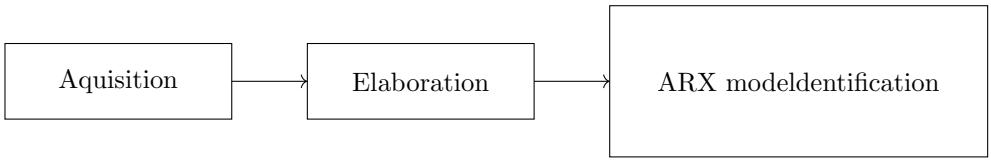
(eq 2.2 chap 2).
In particular, the analyzed system is of the *MISO* (Multiple-Inputs Single-Output) type and has three first-order exogenous inputs and a second-order autoregressive input. Rewriting the equation:

$$y(t) = \alpha_1 \boldsymbol{y(t-1)} + \alpha_2 \boldsymbol{y(t-2)} + +\beta_0 \boldsymbol{u_1(t)} + \beta_1 \boldsymbol{u_2(t)} + \beta_2 \boldsymbol{u_3(t)} \qquad (8.1)$$

The process for determining the future value $y(t)$ of a system based on the ARX model involves estimating the parameters β_i and α_i that constitute it
The estimation is carried out fixing at the very first step the distance $\boldsymbol{D}$, then minimised from an optimisation algorithm of regression.
In this specif case, the error will be minimazed using the least square method and a particular non linear method that involves the cross-entropy error function and a particular optimisated algorithm called SGD.

8.3 Data Analysis

At the core, a data acquisition and processing system is essential to make the prediction phase as effective as possible.



The original datas, suppied from a methereologic station near Brescia, rapresent the value of emission and concentration of PM10 from 2013 to 2021.

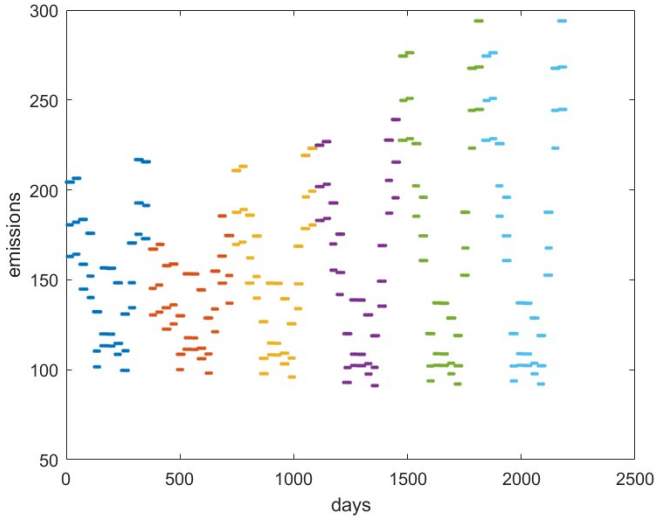


Figure 8.1: Emission rapresentatoin from 2013 to 2019

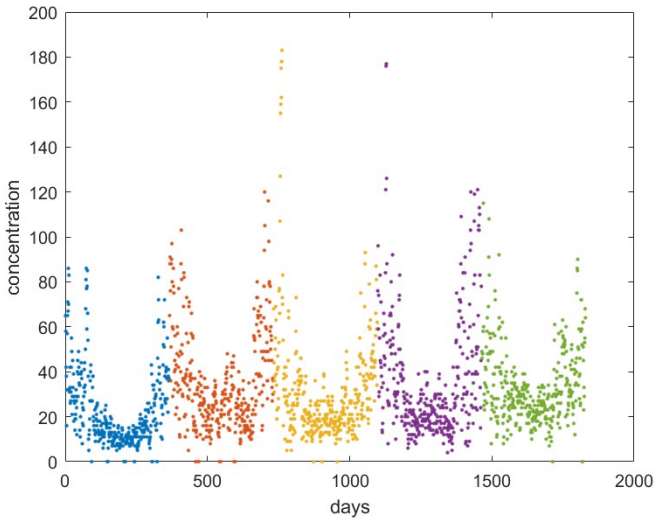


Figure 8.2: Concentration of PM10 from 2013 to 2019

During the preprocessing phase, carried out automatically, to make datas clean and usable, they were initially divided in a *Train* an *Test* partition in $(40\% - 60\%)$ portion, then cleaned up from *null* or insignificant values.

train	test

Figure 8.3: 60% test - 40% train Datasets

On each dataset partition will be estimated the ARX parameters and then compared.

8.4 Linear Regression

The first technique used to solve the ARX system is linear regression performed to minimize the quadratic error. The distance $\boldsymbol{D}$ is fixed as

$$D(f) = LS(f) = \sum_{i}^{n} |f(x_i) - y_i|^2$$

(eq. 3.1 chapt. 3)
Let's assume that the data follows a linear trend; in this way, it is possible to easily determine the next value of concentration.

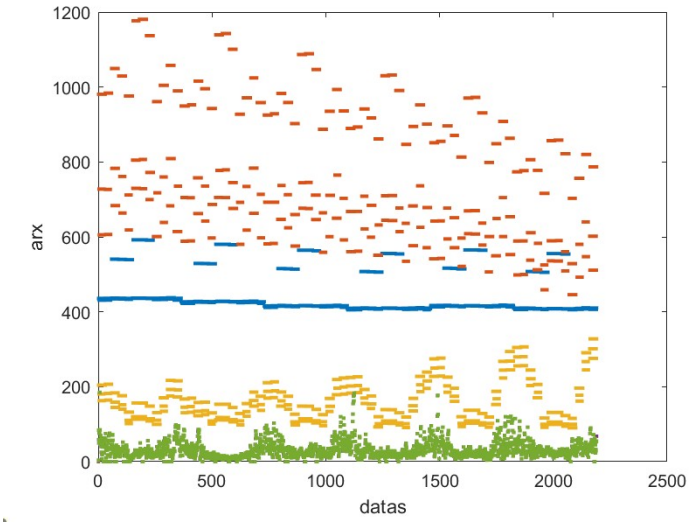


Figure 8.4: ARX data visualization

However, it is noteworthy that the data is not precisely arranged in a linear manner (fig. 8.4). This is where the fixed distance D comes into play; indeed, the line will be constructed in such a way as to minimize the quadratic residuals LS.

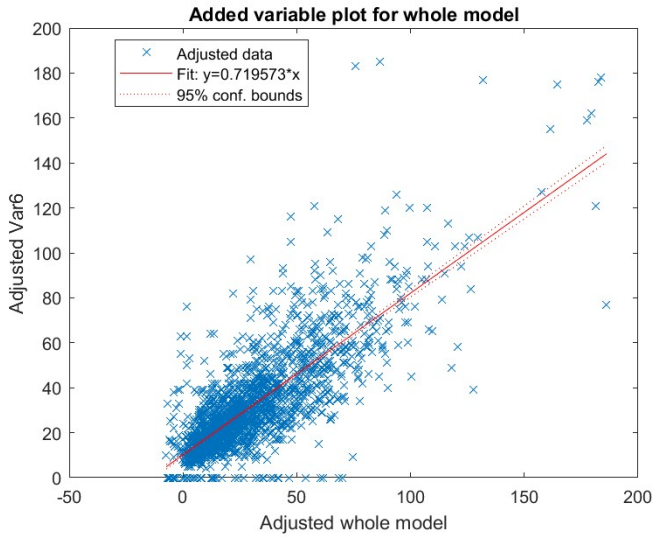


Figure 8.5: Linear regression on Arx datas

In the conducted study, the utilized data spans from the year 2013 to 2021.

However, if we had access to an acquisition system to enrich the study data, the linear regression method allows iteratively adjusting the characteristics of the line based on the current data. This enables a system that is continuously evolving. Despite the apparent simplicity of the approximation, it provides more than satisfactory results.

evar	0.651
mae	8.647
maxe	69.586
me	0.201
medae	5.567
mse	155.84
nmae	0.280
nme	0.006
r2	0.807

Table 8.1: Linear Regression performance

8.5 Artificial Neural Network

The second method employed to solve the parameters of the ARX system is the artificial neural network. To construct and manipulate these tools, it is necessary to understand every detail, starting from the fundamental unit, the Perceptron, the activation functions, and then grouping these units into complete artificial networks (chap. 4). The constructed neural network has the 5 inputs specified by the ARX system

$$(u_1(t), u_2(t), u_3(t), y(t-1), y(t-2))$$

.

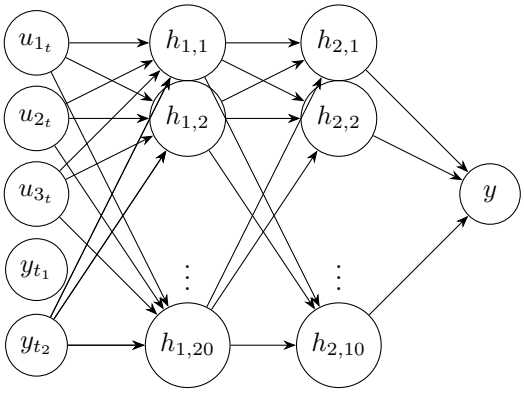


Figure 8.6: Final Architecture

Through the constructor provided for the `MPLRegressor` method from the `sklearn` library, it is possible to set every parameter of the network, including the number of hidden layers, the number of nodes in each hidden layer, the activation function, and the optimizer to minimize the error. After some trial and error, the optimal solution was reached, which includes two hidden layers with 20 and 10 neurons each (fig. 8.6), with each neuron activated by the *Tanh* function.
Among the functions discussed in chap. 5, the arctangent was chosen because abrupt changes in the data could lead to premature saturation of the output.

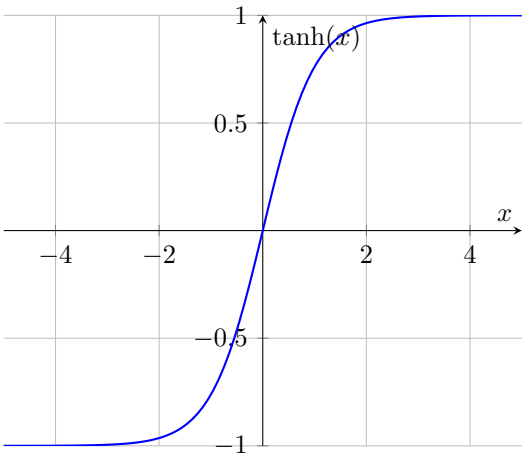


Figure 8.7: fig.5.3 chap.5

The gentle trend of this function allows for the detection of every variation in the input.
The cross-entropy function

$$D(P*|P) = \sum_{y} P^*(y|x_i) log \frac{P^*(y|x_i)}{P(y|x_i;\theta)} \tag{8.2}$$

fixed for this constructor.
It is minimized by choosing the stochastic gradient descent as the optimizer, as detailed in Chapter 6. The obtained performance metrics are as follows:

evar	0.629
mae	8.941
maxe	70.629
me	-0.117
medae	5.809
mse	165.876
nmae	0.290
nme	-0.003
r2	0.794

Table 8.2: Neural Network performance

8.6 Comparison

The overall performance obtained is not entirely satisfactory; both linear regression and the neural network only moderately follow the trend:

$$r2(LR) = 0.807$$

$$r2(NN) = 0.794$$

The parameter *evar* (explained variance score, equivalent to the r^2 parameter) indicates how well the independent variables (inputs) can fit the dependent ones (outputs). The closer this value is to 1, the better is the fit. Other parameters such as *maxe*, *mae*, *mse*, and *nmse* represent singular or overall evaluations of the error.

-	linear regression	Neural Network
mae	8.647	8.941
maxe	69.586	70.629
medae	5.567	5.809
mse	155.84	165.876
nme	0.006	-0.003
nmae	0.280	0.290

Table 8.3: Performance comparison

The performances are comparable, leading us to prefer the simpler evaluation method: the linear regression.
The complexity of an architecture, such as that of a neural network,thus, does not guarantee high performance. Indeed, performance depends on the quality and quantity of the data.

Bibliography

[1] Claudio Carnevale *elaerning*, https://elearning.unibs.it

[2] Steven Brunton *Data Driven Modelling and Ingeneering 2nd edition*

[3] Steven Brunton https://www.youtube.com/@Eigensteve

[4] *neural network modeks*, https://scikit-learn.org/stable/modules/neuralnetworks

[5] *Multy-layer perceptron*, https://www.cs.upc.edu/ mmartin/ml-mds/ml-06.pdf

[6] *functional MLPS*, https://arxiv.org/pdf/0709.3642.pdf

[7] *Quadratic approximation*, https://www.mit.edu/ hlb/StantonGrant/Lecture9/quadratic.pdf

[8] *Activation Function*, $https://pure.strath.ac.uk/ws/portalfiles/portal/118946797/Nwankpa_etal_ICCST_2021_Activation_functions_comparison_of_trends_in_practice.pdf$

[9] Ryan Tibshirani, *Gradient Descent*, https://www.stat.cmu.edu/ ryantibs/convexopt-S15/scribes/05-grad-descent-scribed.pdf

[10] Ryan Tibshirani, *stochastic gradient descent*, $https://www.stat.cmu.edu/ ryantibs/convexopt-F18/scribes/Lecture_24.pdf$

[11] *Gradient Descent*, https://www.niser.ac.in/ smishra/teach/cs460/23cs460/lectures/lec7.pdf

[12] *MLPs*, https://lightning.ai/courses/deep-learning-fundamentals/training-multilayer-neural-networks-overview

[13] *the Perceptron*, $https://www.pycodemates.com/2022/12/perceptron-algorithm-understanding-and-implementation-python.htmle$

[14] Autore2, Nome2, *Titolo del Libro o Documento*, Casa editrice, Anno di pubblicazione.

[15] Autore2, Nome2, *Titolo del Libro o Documento*, Casa editrice, Anno di pubblicazione.