DEMOCRATIC AND POPULAR REPUBLIC OF ALGERIA
الجمهورية الجزائرية الديمقراطية الشعبية

MINISTRY OF HIGHER EDUCATION
AND SCIENTIFIC RESEARCH

HIGHER SCHOOL IN APPLIED SCIENCES
--T L E M C E N--

وزارة التعليم العالي والبحث العلمي

المدرسة العليا في العلوم التطبيقية
-تلمسان-

Higher School of Applied Sciences of Tlemcen

Second-cycle department

Program: Industrial Engineering

Practical Work Handout

Methods of Artificial Intelligence

Prepared by

Dr. Zoulikha KOUDAD, Ep. BELMEKKI

Academic Year: 2025/2026

Preface

This practical work booklet is part of the course *Methods in Artificial Intelligence* designed for 4^{th} year Industrial Engineering students at the Higher School of Applied Sciences. Its main objective is to introduce students to the fundamental concepts of artificial intelligence through a practical and progressive approach.

The booklet contains five lab sessions aimed at enabling students to master key concepts and develop their own intelligent systems, initially limiting the use of libraries and predefined functions. This approach seeks to deepen understanding of the underlying algorithms and encourage a reflective approach to data processing.

The purpose of the first two labs is to implement an intelligent system step by step, without using predefined functions, whether in MATLAB or Python.

- The first lab guides students through classification using the k-nearest neighbors (k-NN) method, including data cleaning and preparation from a downloaded dataset.
- The second lab involves creating and using a perceptron, also fully programmed by the students without relying on built-in functions.

The third and fourth labs deal with handwritten character recognition using a dataset constructed by the students themselves.

- The third lab, conducted in Matlab, covers data preparation and the use of a multilayer neural network with the help of MATLAB toolbox and its predefined functions.
- The fourth lab addresses the same problem in Python, this time utilizing Python libraries such as Scikit-learn and TensorFlow.

Finally, the fifth lab focuses on designing a fuzzy system for controlling an inverted pendulum. This is done in MATLAB by first using the dedicated toolbox, then programming the fuzzy system using predefined functions, and finally using the Scikit-Fuzzy library of Python.

This booklet aims to provide a balance between theoretical understanding and practical skills, offering students the necessary foundations to design, program, and analyze complete intelligent systems.

We hope this work will effectively contribute to the training of future industrial engineers by developing their analytical, programming, and innovation capabilities in the field of artificial intelligence.

Contents

List of Figures

Lab 0

Introduction to MatLab / Python

0.1 MatLab

Matlab (Matrix laboratory), a language for scientific computing, data analysis, visualization, and development of algorithms.

Matlab offers different frameworks (toolbox), of which we cite ; Neural Network Toolbox, Fuzzy Logic Toolbox, Image Processing Toolbox, Signal Processing Toolbox.

0.1.1 MATLAB Environment

MATLAB provides several windows such as the **Command Window**, **Current Directory**, **Workspace**, and **Command History**.

Command Window

The Command Window is the MATLAB command interpreter. It allows you to perform calculations, test functions, and execute instructions interactively.

Functions

To create a function in MATLAB, open a new script page and define the function header. For example:

```
function StudentKPPV()
...
end
```

The file must be saved with the same name as the function, in this case: `StudentKPPV.m`.

0.1.2 Numbers, Vectors, and Matrices

Before starting, it is important to note that indices in **MATLAB** always begin at 1, which implies that index 0 does not exist.

The function `csvread` allows us to read a CSV file and transform it into a matrix, if the first line contains non-numeric data (headers), it must be ignored:

```
data = csvread('Students.csv', 1, 0); % Ignore the 1st line
```

The function `round` allows us to keep the integer part of a number. For example, to compute 75% of a number `T`:

```
trn = round(T * 0.75);
```

Examples of vectors:

```
v = [1 2 3 4];      % Row vector
v = [1; 2; 3; 4];   % Column vector
v = 0:0.2:1;        % From 0 to 1 with step 0.2;
                    % 0  0.2  0.4  0.6  0.8  1
```

Accessing elements:

```
v(3)        % Returns 0.4
v(2:4)      % Returns 0.2 0.4 0.6
```

Useful Functions on Vectors

- `length(v)`: number of elements in vector `v`.
- `max(v)`: maximum value of `v`.
- `min(v)`: minimum value of `v`.
- `mean(v)`: average value of `v`.
- `sum(v)`: sum of all elements in `v`.
- `prod(v)`: product of all elements in `v`.
- `sort(v)`: sorts `v` in ascending order.

Definition of a Matrix

```
M = [1 2 3 ; 4 5 6 ; 7 8 9];
% or
M = [1,2,3;4,5,6;7,8,9];
```

This produces:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
M(2,3)    % Returns 6
```

Special Matrices

```
Z = zeros(2,3);    % Null matrix
U = ones(4,3);     % Matrix full of ones
I = eye(3);        % Identity matrix
```

Extracting a submatrix:

```
M(1:2, 2:3)
% Returns:
% [2 3
%  5 6]
```

Using the colon : operator:

```
M(1,:)      % First row -> [1 2 3]
```

==========> [1 2 3]

```
M(1:2,:)    % First two rows
```

==========> $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

```
M(:,2)      % Second column -> [2; 5; 8]
```

==========> $\begin{bmatrix} 2 \\ 5 \\ 8 \end{bmatrix}$

Useful Functions on Matrices

- `size(M)`: dimensions of `M`.
- `max(M)`: row vector of column-wise maxima.
- `min(M)`: row vector of column-wise minima.
- `rank(M)`: rank of `M`.
- `det(M)`: determinant of `M`.
- `diag(M)`: diagonal elements of `M`.
- `triu(M)`: upper triangular part of `M`.

0.1.3 The `for` Loop

```
for i = 1:10
...
end
```

0.1.4 The graphical interface

Selection from a list

We start by preparing the list of choices that we call **cs**. To prepare **cs** to contain choices from 1 to 9, the function **num2str(i)** allows transforming an integer i into a string:

```
for i = 1:9
cs{i} = num2str(i);
end
```

After that we use the command **listdlg** to display a window with the list of choice (**cs**) prepared in advance as a list of strings.

```
[sel, ok] = listdlg('ListString', cs, ...
'SelectionMode', 'Single');
```

The output **sel** contains the user's choice.

Dialog box

Display a dialog box for the result *cl*:

```
h = msgbox(cl, 'The␣category␣is');
```

Window to select a file

Open a window to select a file (or image):

```
[fname, fpath, fltidx] = uigetfile( ...
{'*.gif;*.png;*.jpg;*.xcf', 'Image␣formats'});
```

uigetfile opens a dialog box that lists the files in the current folder (in this case the function displays all image files). **uigetfile** returns the name of the file selected when the user clicks on Open.

The complete file name can be built as:

```
fileName = [fpath '/' fname];
```

0.1.5 Image Processing

- **Reading image files into a matrix in MATLAB**

```
img = imread('football.jpg');
```

The function **imread** reads the image from the file specified by the filename 'football.jpg', and returns a matrix: img ..

- **Save the matrix as an image**

```
imwrite(img, 'img_football.jpg');
```

- **Show an image**

```
imshow(img);
```

- **Resize an image** (img) to have the dimensions $x2 \times y2$

```
img2 = imresize(img, [x2 y2]);
```

- **Binarization of an image** (black = 0, white = 1)

```
img2 = im2bw(img);
```

0.2 Python

Python is the most widely used programming language in the world. It is free and cross-platform, meaning that it works on many operating systems: Windows, Mac OS X, Linux, Android, ...

0.2.1 Python Environment

There are several environments for programming in Python. Among the complete IDEs, we can mention **PyCharm** (JetBrains), **Visual Studio Code (VS Code)**, and **Spyder**.

There are also interactive notebooks such as **Jupyter Notebook / JupyterLab**, **Google Colab** (with GPU/TPU available), and **Kaggle Notebooks** (with available datasets).

The Interactive Console

The Python console (REPL) allows executing instructions line by line. For instance:

```
>>> 2 + 3
5
>>> print("Hello, Python!")
Hello, Python!
```

0.2.2 indentation

Unlike MATLAB, in **Python** there is no need to use `end` to close a block (`function`, `if`, `for`, `while`, ...). The end of a block is indicated only by indentation. Inside a block, each line must be indented (shifted to the right). To exit the block, simply return to a new line without indentation.

0.2.3 Defining Functions

A function in Python is created using the keyword `def`. For example :

```
def add_numbers(a, b):
    """Return the sum of two numbers."""
      return a + b():
```

The file must be saved with the extension `.py`, e.g. `student_knn.py`.

0.2.4 Lists in Python

In **Python**, a list is a data structure that allows storing a collection of elements, which can be of different types (integers, strings, floats, etc.). Lists are defined using square brackets `[ ]`, and elements are separated by commas.

Example.

```
# Creating a list
my_list = [10, 20, "pomme", 40]
# Accessing elements
print(my_list[0])    # First element : 10
print(my_list[2])    # Third element : "pomme"
# Modifying an element
my_list[1] = 25      # Now the list is [10, 25, 30, 40]
# Adding a new element
my_list.append(50)   # The list becomes [10, 25, 30, 40, 50]
```

Note. The indices in Python always start at 0. Therefore, the first element of a list is accessed with index 0, and the last element can be accessed with index `len(list) - 1`.

0.2.5 Creating and Copying Lists in Python

Empty List. In **Python**, an empty list can be created in two ways:

```
L = []           # Using square brackets
L = list()       # Using the keyword list
```

Copying a List. When you assign a list to another variable, Python does not create a real copy of the list as in MatLab. Instead, both variables point to the same object in memory.

```
L1 = [1, 2, 3]
L2 = L1
L2[0] = 100
print(L1)    # Output: [100, 2, 3]
print(L2)    # Output: [100, 2, 3]
```

To create an actual copy of a list, you can use one of the following methods:

- **Slicing notation:**

```
L1 = [1, 2, 3]
L2 = L1[:]
L2[0] = 100
print(L1)    # Output: [1, 2, 3]
print(L2)    # Output: [100, 2, 3]
```

- **Using the list() function:**

```
L1 = [1, 2, 3]
L2 = list(L1)
```

- **Using the copy module:**

```
import copy
L1 = [1, 2, 3]
L2 = copy.copy(L1)
```

0.2.6 Numbers, Lists, and Arrays

In **Python**, indices always begin at 0, which means that the first element of a list or array is accessed with index 0.

To read a CSV file and transform it into a matrix, the `pandas` or `numpy` libraries are commonly used. For example, with `pandas`:

```
import pandas as pd
data = pd.read_csv("Students.csv")
print(data.head())
```

If the first line contains headers, it will be ignored automatically by `pandas`. For numerical arrays, `numpy` is more efficient:

```
import numpy as np
data = np.loadtxt("Students.csv", delimiter=",", skiprows=1)
```

Rounding a number:

```
T = 100
trn = round(T * 0.75)  # 75% of T
```

Examples of lists:

```
v = [1, 2, 3, 4]          # List
v[2]                      # Access third element (index 2) -> 3
v[1:4]                    # Slice -> [2, 3, 4]
```

Using NumPy arrays:

```
import numpy as np
v = np.arange(0, 1.1, 0.2)    # From 0 to 1 with step 0.2
print(v)                      # [0.  0.2 0.4 0.6 0.8 1. ]
```

Useful Functions on Arrays

- `len(v)`: number of elements in list/array.
- `np.max(v)`: maximum value.
- `np.min(v)`: minimum value.
- `np.mean(v)`: average value.
- `np.sum(v)`: sum of all elements.
- `np.prod(v)`: product of all elements.
- `np.sort(v)`: sorts the array in ascending order.

Definition of a Matrix

```
M = np.array([[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
```

gives the following matrix: M= $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

```
print(M)
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]
```

```
print(M[1,2])    # Returns 6
```

Special Matrices

```
Z = np.zeros((2,3))    # Null matrix
U = np.ones((4,3))     # Matrix full of ones
I = np.eye(3)          # Identity matrix
```

Extracting a submatrix:

```
M[0:2, 1:3]
# [[2 3]
#  [5 6]]
```

Using slicing:

```
M[0,:]     # First row -> [1 2 3]
```

==========> [1 2 3]

```
M[0:2,:]   # First two rows
```

==========> $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

```
M[:,1]     # Second column -> [2 5 8]
```

==========> $\begin{bmatrix} 2 \\ 5 \\ 8 \end{bmatrix}$

Useful Functions on Matrices

- `M.shape`: dimensions of the matrix.
- `np.max(M, axis=0)`: column-wise maxima.
- `np.min(M, axis=0)`: column-wise minima.
- `np.linalg.matrix_rank(M)`: rank of the matrix.
- `np.linalg.det(M)`: determinant of the matrix.
- `np.diag(M)`: diagonal elements of the matrix.
- `np.triu(M)`: upper triangular part of the matrix.

Note that as in **list**, in **NumPy**, assigning one array to another variable does not create a true copy. Both variables will point to the same memory location.

0.2.7 The `for` Loop

```
for i in range(1, 11):
    print(i)
```

0.2.8 The `range()` Function

In Python, `range()` generates a sequence of integers, often used in `for` loops.

```
range(stop)           # 0 ... stop-1
range(start, stop)    # start ... stop-1
range(start, stop, step)
```

Examples:

```
list(range(5))         # [0,1,2,3,4]
list(range(2,7))       # [2,3,4,5,6]
list(range(0,10,2))    # [0,2,4,6,8]
```

0.2.9 Graphical User Interface (GUI)

The standard `tkinter` library in Python allows us to create a graphical interface.

Selection from a List

The `Listbox` widget is used to create a list window where we can select an element, but it is not as straightforward as in MATLAB. Of course, we must first create the list of choices and then create a `tkinter` window using the `Tk` method, while also giving it a title:

```
import tkinter as tk
from tkinter import messagebox

lst = ["Python", "Java", "C++", "JavaScript"]
fen = tk.Tk()
fen.title("Choose a language")
```

Next, we can specify that the window type is a `Listbox` using the corresponding method. We then insert each element of the list one by one using the `insert` method. The `pack` method is a geometry manager that arranges widgets relative to each other. The option `anchor='w'` places the list elements on the west (left side) of the window.

```
listb = tk.Listbox(fen)
for item in lst:
    listb.insert(tk.END, item)
listb.pack(anchor='w')
```

Storing the Selection Result

To store the selection result, we use a mutable variable (for example, a dictionary as follow : `result = { "val" : None }`), which behaves like a global variable when used inside a nested function. This way, its value is not lost when leaving the function.

We define the method `selec`, which is called when clicking the `Select` button of the window. Inside this function, the `curselection` method retrieves the index of the selected element, and the `get` method returns the chosen element. This element is then stored in our dictionary under the key `"val"`. If no element is selected, a small `messagebox` window is displayed to require the user to make a choice. After that, the `tk.Button` method allows us to create the `Select` button.

```
result = {"val": None}
def selec():
    sel = listb.curselection()
    if not sel:
        messagebox.showwarning("Warning", "Please␣select␣an␣item.")
        return
    result["val"] = listb.get(sel[0])
    fen.destroy()
butt1 = tk.Button(fen, text="Select", command=selec)
butt1.pack()
```

Finalizing the Tkinter Window

Since the code is now complete, we need to add the `mainloop` method to keep the window active during the selection process. After the window is closed, we can retrieve the selected value from the dictionary and use it later in the program.

```
fen.mainloop()
X = result["val"]
```

Lab 1

k-Nearest Neighbor (KNN)

1.1 Required Work

Implement the k-nearest neighbor algorithm to predict the category of student performance, in both the **MATLAB** and **Python** environments.

1.1.1 Data Set Presentation

In this workshop, we will use a learning base "StudentsPerformance.csv" already downloaded from the **Kaggle** site and located in the **Student.csv** file, it contains comprehensive information on **2,392** high school students, detailing their demographic characteristics, study habits, parental involvement, extracurricular activities, and academic performance. The target variable, GradeClass, classifies students' grades into distinct categories.

Student number	unique identifier assigned to each student (from 1001 to 3392).
Age	students' age ranges from 15 to 18 years old.
Gender	0 represents male and 1 represents female.
Ethnicity	0: Caucasian, 1: African American, 2: Asian, 3: Other
ParentalEducation	0: None, 1: High School, 2: College, 3: Bachelor's Degree, 4: Graduate
StudyTime	Weekly study time in hours, ranging from 0 to 20.
Absences	Number of absences during the school year, ranging from 0 to 30.
Tutoring	Tutoring Status, where 0: No and 1: Yes.
Parental Support	0: None, 1: Low, 2: Moderate, 3: High, 4: Very High
Extracurricular	Extracurricular Activities: 0 : No and 1: Yes.
Sports	0: No and 1: Yes.
Music	0: No and 1: Yes.
Volunteer	0: No and 1: Yes.
GPA	grade point average on a scale of 2.0 to 4.0
GradeClass	**0: A** (GPA>= 3.5), **1: B** (3-3.5), **2: C** (2.5-3), **3: D** (2-2.5), **4: F** (< 2)

1.1.2 Work Steps

- Start by creating a folder for the workshps, and copy the file **"Student.csv"** inside.
- In the Matlab environment open a new file containing a function StudentKPPV().
- read the csv file into a matrix **"data"**.
- Clean the data; eliminate attributes (columns that are of no use in the classification).
- Split the data into inputs **(Xdata)** and outputs **(Ydata)**, then split them into training and test data. **(xtrain, xtst, ytrain, ytst).**
- Write the KPPV algorithm to predict the category of a single student.
- Modify the program to predict the categories of all students in **xtst** and calculate the number of correct and incorrect predictions

1.1.3 The k-nn Algorithm.

Implementation of the k-ppv algorithm.

Algorithm 1 k-ppv

parameters: k: number of neighbors. d: similarity measure (distance)
$D = (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_m)$the training set
$Y = y_1, y_2, \ldots, y_m$the set of classes
$\overline{x_i}$the vector representation of $x_i, i = 1, \ldots, n$.
$\overline{x}$ the vector representation of x .
$N_k(x)$ the set of the k nearest neighbors of x, initially empty.
For each x_i compute $d(x_i, x)$
endfor
$N_k(x) = argmax(d(x_i, x))_k$ %the k elements with maximum similarity
count the number of'occurrences of each class in $N_k(x)$
assign x the majority class

- The user must be able to choose ***k***, for the number of neighbors taken into account, (use ***listdlg***).
- Use ***msgbox*** to visualize the final result.

1.2 Solution: 1 with MatLab

Before starting, I want to remind you to execute each step individually to prevent error accumulation and ensure the program behaves correctly. Whenever students wish to verify the result of an instruction, they should write it without a semicolon (;) and execute it.

1.2.1 Function Creation and Dataset Loading

We begin by creating the KnnStudent function and reading the Student.csv file into a data matrix as follows. Note that the first line of the file contains alphabetic information, which should be removed because the csvread function only processes numeric data. To skip the header line, we add ,1,0 in the function call.

```
function KnnStudent()
data = csvread('Student.csv', 1, 0);
end
```

1.2.2 Data Preparation

Before data cleaning, it is essential to determine the dataset's dimensions using the function "*size*". Applying this function to the first column provides the number of rows, corresponding to the number of students in the dataset. We then select 75% of these rows for training and 25% for testing. For the input features, we retain only columns 2 to 13, as the first column (student number) does not influence the results. The 14th column, representing the student's grade, is also excluded because we aim to predict the range of this grade, which corresponds to the 15th column—the output of our KNN model.

```
function KnnStudent()
data = csvread('Student.csv', 1, 0);
T=size(data(:,1))
trn=round(T*0.75)
tst=T-trn
xdata=data(:,2:13)
ydata=data(:,15)
xtrain=xdata(1:trn,:)
ytrain=ydata(1:trn,:)
xtst=xdata(trn+1:T,:)
ytst=ydata(trn+1:T,:)
end
```

1.2.3 Choice of the Value of k

To implement the K-Nearest Neighbors (KNN) algorithm, we've chosen to allow the user to select the value of `k` using MATLAB's graphical user interface component, `listdlg`. We prepared a list of numbers from 1 to 9 within a cell array and provided it to the listdlg component. The user's selection is then assigned to the variable sel as follows:

```
for i=1:9
    cs{i}=num2str(i);
end
[sel,ok]=listdlg('PromptString', 'Select␣the␣value␣of␣k:' ,'
   ListString',cs,'SelectionMode','Single');
k=sel;
```

We will obtain the window as in figure 1.1:

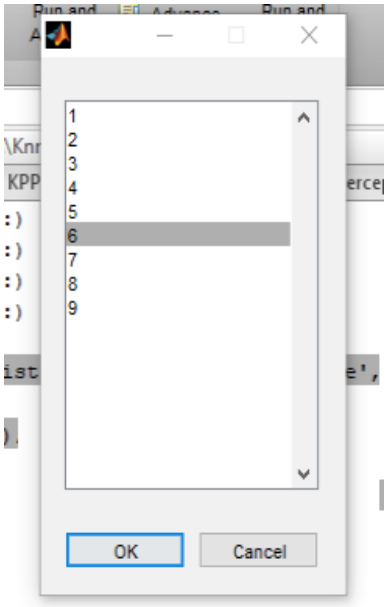

Figure 1.1: The selection of K

1.2.4 Computation of Euclidean Distances

After this, we proceed to the steps of the KNN algorithm. We classify each element in the test set and calculate the number of correct classifications (pos) and incorrect classifications (neg). For each test element, we compute the Euclidean distance separating it from each training element, resulting in a distance array for each as follows:

```
pos=0;
neg=0;
for i=1:tst
   x=xtst(i,:)
   for j=1:trn
```

```
        s=0;
      for m=1:12
         dif=x(i,m)-xtrain(j,m);
         dif=dif*dif;
         s=s+dif;
      end
        dis(j)=sqrt(s);
  end
```

1.2.5 Finding the k Nearest Neighbors

Next, within the initial loop that iterates over the test elements, we search for the k smallest distances without exiting this loop.

```
y=max(dis);
for ii=1:k
    [x1,ind]=min(dis);
    kpp(ii)=ytrain(ind);
    dis(ind)=y;
end
```

1.2.6 Majority Class Recognition

After identifying the `k` nearest neighbors for each test element, the next step is to determine the most frequent class among these neighbors. Given that we have 5 distinct classes, we count the occurrences of each class label among the `k` neighbors and assign the test element to the class with the highest count, as follows:

```
tab=zeros(1,5);
for ii=1:k
   jj=kpp(ii);
   tab(jj+1)=tab(jj+1)+1;
end
[x1,jj]=max(tab);
```

1.2.7 Comparison with the True Class

To verify the accuracy of our KNN classification, we compare its output to the desired output in the ytest array. Since our dataset's classes start from 0, but MATLAB arrays are 1-based indexed, we need to add 1 to the class labels to align them correctly.

```
if (ytst(i)+1==jj)
```

```
    pos=pos+1;
  else neg=neg+1;
  end
```

1.2.8 Displaying the Result with a Message Box

Finally, we can display the results using the disp function or by utilizing a message box (msgbox) in MATLAB's graphical user interface.

```
res=num2str(pos)
res=[' The number of correst classification is: ' res]
h=msgbox(res,' The result');
```

What will give the figure 1.2

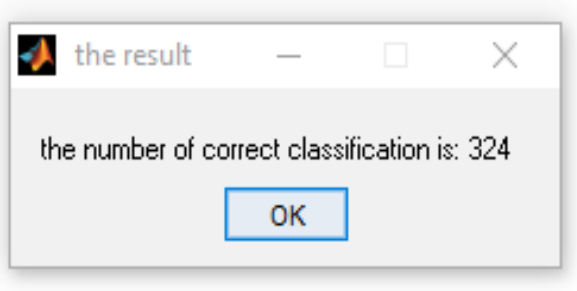

Figure 1.2: The msgbox to display the result

1.3 Solution 2: with Python

1.3.1 File creation and dataset loading

In Python, we do not need to create a main function; it is enough to open a new file named "**knnStudent.py**" and start coding.

As already mentioned in Lab 0, in Python most functionalities require libraries. Therefore, it is better to start by importing the libraries we will need, as follows:

```
import pandas as pd
import numpy as np
import tkinter as tk
from tkinter import messagebox
import math
```

We can then open our dataset file 'Student.csv' using the **read_csv** function from Pandas. The result of this operation is an object of type pandas.DataFrame, so it must be converted into a numerical array using the **to_numpy** method, as follows:

```
df = pd.read_csv("Student.csv", skiprows=1)
data = df.to_numpy()
```

Alternatively, we can do the same work by directly using the **loadtxt** method from the NumPy library. In both methods, we removed the first line of the Student.csv file by using the option **skiprows=1**, since it is the header line, and we only need the numerical data for processing, as follows:

```
data = np.loadtxt("Student.csv", delimiter=",", skiprows=1)
```

1.3.2 Data Preparation

We will calculate the dimension of the array `data` using the method `shape`, which returns an array of two elements. The first element (index 0) indicates the number of rows, and the second element (index 1) represents the number of columns.

We will take 75% of the rows for training and 25% for testing. Then, we will remove column 0, which contains the student ID number, since it should not be considered as reliable feature for classification.

Thus, we will use columns 1 to 12 as the input features, and the 15^{th} column (column index 14 in Python) as the classifier output, as shown in the following code:

```
T=data.shape[0]
trn = round ( T*0.75)
tst =T- trn
xdata = data [: ,1:13]
ydata = data [: ,14]
```

```
xtrain = xdata [0: trn ,:]
ytrain = ydata [0: trn ]
xtst = xdata [trn : T ,:]
ytst = ydata [trn : T ]
```

1.3.3 Choice of the Value of k

To choose the value of k interactively, we will use a `Listbox` from the `tkinter` library, to which we provide a list of numbers from 1 to 9. We create the window, insert the elements of the list, and write the `selec` method, which is triggered when the `Select` button is clicked.

If the user has not selected any element, a message will be displayed asking them to make a choice. The `Listbox` window is shown in Figure 1.3.

After a successful selection, the `Listbox` will be destroyed, and we retrieve the chosen value of k as follows:

```
lst = list(range(1, 10))  # numbers from 1 to 9
fen = tk.Tk()
fen.title("choose␣a␣value␣for␣K")
listb = tk.Listbox(fen)
for itm in lst:
listb.insert(tk.END,itm)
listb.pack(anchor='w')
result = {"k": None}
def selec():
sel = listb.curselection()
if not sel:
messagebox.showwarning("Warning", "Please␣select␣an␣item.")
return
result["k"] = listb.get(sel[0])
fen.destroy()
butt1=tk.Button(fen, text="select", command=selec)
butt1.pack()
fen.mainloop()
K=result["k"]
```

1.3.4 Computation of Euclidean Distances

We now move on to the classification of the test examples using the KNN method. To achieve this, we begin by computing the Euclidean distance between each element of the test set and all the elements of the training set, considering that each element contains 12 attributes.

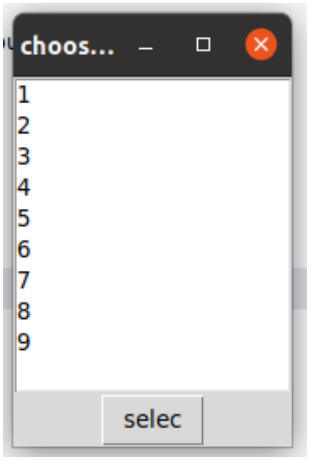

Figure 1.3: The Lisbox for choosing a value for K

Thus, for each test element, we will need a distance array of size `trn`. This array, denoted as `dis`, must be initialized with zeros before iterations over the test elements.

The variables `pos` and `neg` are used as counters for positive and negative classifications, respectively, as illustrated below:

```
pos =0
neg =0
dis=np.zeros((trn,1))
for i in range(tst):
    x = xtst [i ,:]
    for j in range(trn):
        s =0
        for m in range(12):
            dif = x [m ] - xtrain [j , m ]
            dif = dif * dif
            s = s + dif
        dis [j ] = math.sqrt ( s )
```

1.3.5 Finding the k Nearest Neighbors

While still in the first loop iterating over the test elements, we search for each test element's k nearest neighbors from the distance matrix.

Each time we find a minimum value in the distances, we record its class in the `knn` array and replace this minimum by the maximum value, in order to be able to find the next minimum.

The procedure is illustrated as follows:

```
    knn=np.zeros(( K,1))
    y = max(dis)
    for n in range(K):
        ind = np.argmin(dis)
```

```
        knn[n] = ytrain[ind]
        dis[ind] = y
```

1.3.6 Majority Class Recognition

After retrieving the k nearest neighbors of each test example, we need to identify the most frequent class. Since we have five classes, we propose to create an array of five elements initialized to zero.

We then iterate through the array of k nearest neighbors. For each class encountered, we increment the element at the corresponding index in the class array. Finally, we take the maximum value, which represents the majority class, as shown below:

```
    tab = np.zeros(( 5,1))
    for n in range(K):
        m = int(knn[n])
        tab[m] = tab[m]+ 1
    m = np.argmax(tab)
```

1.3.7 Comparison with the True Class

Now that we have the class predicted by our KNN, we compare it with the true class. If the prediction is correct, we increment `pos`; otherwise, we increment `neg`.

All of this is done inside the first loop that iterates over the test elements. At this point, we exit the loop and display the accumulated results with a simple `print`, as shown below:

```
    if (ytst[i] == m):
        pos = pos + 1
    else:
        neg = neg +1
print("pos=", pos, "neg=", neg)
```

1.3.8 Displaying the Result with a Message Box

We can also use a `messagebox` to display the result. In this case, it is better to prepare the string to be displayed in advance, as follows:

```
res="The␣number␣of␣correct␣classification␣is:"+str(pos)
bx = tk.Tk()
bx.withdraw()  # hide the main window
messagebox.showinfo("Result", res)
```

The displayed result is shown in Figure 1.4.

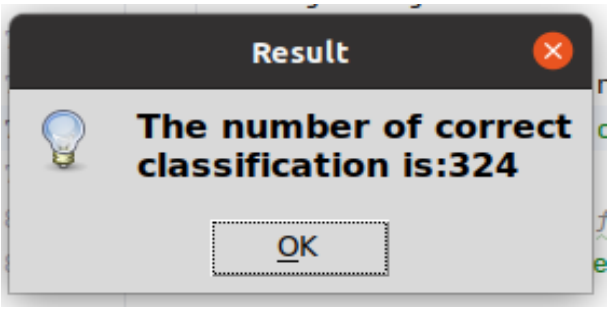

Figure 1.4: A messagebox for displaying the result of knn classification

Lab 2

The Perceptron

2.1 Requested Work

The task is to develop two implementations of a perceptron: one using Matlab and another using Python. The perceptron should be designed to classify flowers based on four characteristics: Sepal Length (x_1), Sepal Width (x_2), Petal Length (x_3), Petal Width (x_4), using the provided dataset.

2.2 Approach

To implement the perceptron algorithm, start by creating a file named `perceptron1` (.m/.py) in your working directory.

2.2.1 Preparing the Example Base

In the same file, write the function `base` to manually enter the dataset. This function returns a matrix $\mathbf{X}$, which will contain the input data for training, and a matrix $\mathbf{Y}$, which will contain the corresponding classification target values. Another row can be added to matrix $\mathbf{X}$ in order to implicitly implement the neuron biases.

2.2.2 Receiving the Training Data

Return to the main function/program and call the function `base` to make the dataset available for the perceptron.

2.2.3 The Neuron Structure

- Create a Neuron structure, containing 2 fields: the output (integer) and the weight table (real) the implicit bias will be the weight w_{n+1}.
- The weights (w) and biases are randomly initialized between 0 and 1, directly when creating the neuron structure.

- The number of weights at the input of a neuron corresponds to the number of inputs (p) of the perceptron.
- After defining a neuron, we can initialize (n) neurons, such that (n) is the number of outputs, (number of flower classes.)
- The numbers p and n correspond to the number of rows in the input matrix X and the number of rows in the output matrix Y of the learning base respectively.

2.2.4 The Learning Step

Define a constant α corresponding to the learning step at 0.01.

2.2.5 The Iterations

- Use a *while* loop to combine two stopping criteria, number of iterations < 100 and $error > 0$,
- In each iteration we make a pass through all the learning examples;
- For each training example, the output of each neuron is computed, and if an error occurs, the p weights of that neuron are updated.

2.2.6 Output of a Neuron

Write a function that calculates the internal value of a neuron: $v = \sum_{i=1}^{p+1} w_i x_i$, and the output of a neuron; for our TP we choose the threshold function (Heaviside): If $v > 0$ the output y of the neuron is 1 otherwise 0.

2.2.7 Updating the Weights

Apply the perceptron learning rule.

$$w_i = w_i + \alpha \left(y - h\left(X_t\right)\right) x_i$$

2.2.8 Saving

At the end of the learning, save the neural network, to be able to use it later.

2.2.9 Using the Perceptron for Classification

Write a function `use()`, in another file, that loads the already recorded neural network, and calculates the output from the unlearned data, then calculates the number of errors.

2.3 Example Base for Flower Classification

setosa class:

$x1$	4.9	5.1	4.7	4.6	5	5.4	4.6	5	4.4	4.9	5.4	4.8
$x2$	3.0	3.5	3.2	3.1	3.6	3.9	3.4	3.4	2.9	3.1	3.7	3.4
$x3$	1.4	1.4	1.3	1.5	1.4	1.7	1.4	1.5	1.4	1.5	1.5	1.6
$x4$	0.2	0.2	0.2	0.2	0.3	0.4	0.3	0.2	0.2	0.1	0.2	0.2

Versicolor class

$x1$	5.2	5	5.9	6	6.1	5.6	6.7	5.6	5.8	5.6	5.9	6.1
$x2$	2.7	2.0	3.0	2.2	2.9	2.9	3.1	3.0	2.7	2.5	3.2	2.8
$x3$	3.9	3.5	4.2	4.0	4.7	3.6	4.4	4.5	4.1	3.9	4.8	4.0
$x4$	1.4	1.0	1.5	1.0	1.4	1.3	1.4	1.5	1.0	1.1	1.8	1.3

virginica class

$x1$	6.3	6.5	7.6	4.9	7.3	6.7	7.2	6.5	6.4	6.8	5.8	6.4
$x2$	2.9	3.0	3.0	2.5	2.9	2.5	3.6	3.2	2.7	3.0	2.8	3.2
$x3$	5.6	5.8	6.6	4.5	6.3	5.8	6.1	5.1	5.3	5.5	5.1	5.3
$x4$	1.8	2.2	2.1	1.7	1.8	1.8	2.5	2.0	1.9	2.1	2.4	2.3

- Build the basis as a matrix X for the inputs, and a matrix Y for the outputs. X and Y must have the same number of columns.
- Matrix X is drawn directly from the data above, and Y, you will deduce it from the same data, without forgetting that the values of Y are 0 and 1.
- Don't forget to use a large part for training and a small part for testing.

2.4 Useful Matlab Informations and Functions

- In Matlab, you can create several functions in the same file. The first function is the main function and must match the name of the file.
- The '**save/load** functions are used to save data in the current directory and reload them when needed as follows:

  a- if you do **save('FD');** with a single parameter, all the data in the workspace will be saved in a file "FD.mat". Whereas;

  b- **save('FD','X','Y');** is used to save the variables X and Y in a file "FD.mat".

  c- to load all the contents of a file "FD.mat", we write: **load('FD')**.

  d- to load the variables X and Y of a file FD we write: **load('FD','X','Y')**.

- **the structures**: to explicitly define a structure, we use the following syntax:
  s=struct(field1,value1,..., fieldN, valueN)
- **rand(i,j)** returns a matrix of i*j random numbers between 0 and 1.
- **disp(x);** displays the value of x
- **net=perceptron;** allows to create a perceptron without configuration.
- **net = train(net,X,Y);** allows to configure the perceptron according to the dimensions of the inputs X and the outputs Y, and to do the training on these same data.
- The while loop, example:
  while(condition1 & condition2)
  .....
  .....
  end

2.5 Useful Python Informations and Functions

- In Python, we create the file `perceptron1.py`, where the main program is written. We can directly write the code into this file, starting by importing the necessary libraries. Inside the script, we can define several functions and call them as needed.
- To save and load numerical data of type `numpy`, we use `np.save` and `np.load`.
- For object-type data, we use the `pickle` library with (`pk.dump`, `pk.load`). The `pickle` library requires opening a file with `open` before saving or loading data.
- For structured data, we can create a class of object and use it. For example, we can define a class as follows:

```
class MyClass:
    att1 = value1
    att2 = value2
```

- We can save multiple instantiated objects of the class in a list by adding them one by one using the `append` method as follows:

```
list = []
for i in range(n):
    list.append(MyClass())
```

- The function `np.random.rand(p)` generates an array of `p` random numbers from a uniform distribution between 0 and 1.

- The `while` loop with two conditions :

```
while condition1 and condition2:
    ....
    ....
```

2.6 Solution 1: with Matlab

We will start by creating the function perceptron1 and the function base in the same script.

2.6.1 Dataset Preparation

In the base function, which returns the input data X and the output Y of our perceptron, we will copy the input data X by concatenating the rows of the three classes while maintaining the order. This will result in a matrix with 4 rows and 36 columns (representing 36 flowers).

For the outputs, we simply assign the first 12 flowers the column vector $[1; 0; 0]$, representing the first class, the next 12 flowers the vector $[0; 1; 0]$, and the last 12 flowers the vector $[0; 0; 1]$.

Then, we shuffle the order of X and Y in the same way to ensure that each element of X maintains its corresponding Y, using the randperm function as follows:

```
function perception1()

end
function [X, Y]=base()
X=[4.9 5.1 4.7 4.6 5 5.4 4.6 5 4.4 4.9 5.4 4.8 5.2 5 5.9 6 6.1
   5.6 6.7 5.6 5.8 5.6 5.9 6.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5
   6.4 6.8 5.8 6.4;
3.0 3.5 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 3.7 3.4 2.7 2.0 3.0 2.2
   2.9 2.9 3.1 3.0 2.7 2.5 3.2 2.8 2.9 3.0 3.0 2.5 2.9 2.5 3.6
   3.2 2.7 3.0 2.8 3.2;
1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 3.9 3.5 4.2 4.0
   4.7 3.6 4.4 4.5 4.1 3.9 4.8 4.0 5.6 5.8 6.6 4.5 6.3 5.8 6.1
   5.1 5.3 5.5 5.1 5.3;
0.2 0.2 0.2 0.2 0.3 0.4 0.3 0.2 0.2 0.1 0.2 0.2 1.4 1.0 1.5 1.0
   1.4 1.3 1.4 1.5 1.0 1.1 1.8 1.3 1.8 2.2 2.1 1.7 1.8 1.8 2.5
   2.0 1.9 2.1 2.4 2.3]
for i=1:12
   Y(:,i)=[1;0;0];end
for i=13:24
   Y(:,i)=[0;1;0];end
for i=25:36
   Y(:,i)=[0;0;1];end
t=size(X)
n=t(2)
ind=randperm(n)
X=X(:,ind)
Y=Y(:,ind)
end
```

2.6.2 Perceptron Initialization

We return to our function perceptron1, where we call base to obtain the data. We extract the dimensions such that the number of rows in X corresponds to the number of inputs P, and the number of rows in Y corresponds to the number of outputs n, which is also the number of neurons.

Thus, we define the neuron structure, which contains a weight vector w with p elements initialized randomly. The bias b is also initialized randomly. We can optionally choose to include s in the structure.

Then, we create our perceptron, which corresponds to a vector of n neurons:

```
function perception1()
[X, Y]=base();
t1=size(X);
t2=size(Y);
p=t1(1)  %number of input
n=t2(1)  %number of  neurones
neurone=struct('w',rand(p,1),'b',rand(1),'s',0);
for i=1:n
   neuro(i)=neurone;  %vecteur des neurones
end
```

2.6.3 Training the Perceptron

Next, we start the perceptron learning algorithm with iterations that calculate the output for each neuron and each input flower, then compare the output with the corresponding vector in Y. If there is an error, we update the weights by applying the perceptron learning rule.

The training is performed on 30 flowers only, while 6 flowers are reserved for testing.

```
alpha=0.05;
i=0;
er=1
while (i<100 && er>0)
   er=0;
   i=i+1;
   for j=1:30
      x1=X(:,j);
      y1=Y(:,j);
      for k=1:n
         S=srti(neuro(k),x1,p);
         if (S~=y1(k))
           er=er+1
           for m=1:p      %updating weights
```

```
                neuro(k).w(m)=neuro(k).w(m)+alpha*(y1(k)-S)*x1(m);
            end
            neuro(k).b=neuro(k).b+alpha*(Y(k)-S);    %updating b
          end
       end
   end
end
```

Knowing that the function srti must also be written either before or after the base function, this function calculates the weighted sum of a neuron's inputs and applies the threshold function.

```
function srt=sorti(ner,x,p)
s=0;
for i=1:p
   s=s+ner.w(i)*x(i);
end
if(s>0) srt=1;
 else srt=0;
end
end
```

2.6.4 Testing and Saving the Perceptron

To calculate the number of correct classifications on the 6 test flowers, we return to the perceptron1 function, right after the end of the training while loop. We compute the output for these flowers and count the number of errors, which can then be displayed.

```
ert=0;
for j=31:36
   x1=X(:,j);
   y1=Y(:,j);
   for k=1:n
      S(k)=srti(neuro(k),x1,p);
   end
   if (S~=y1)
      ert=ert+1
   end
end
disp('the number of error is :')
disp(ert)
save('fper','neuro','X','Y');
end
```

2.6.5 Using the Perceptron

After training, we save our perceptron and dataset in a new file *fper* and load them into a useper function in a new script, which allows us to use the perceptron when needed. The srti function must also be copied into the same script.

```
function S=useper(x)
load fper
t1=size(X);
t2=size(Y);
p=t1(1)  %number of input
n=t2(1)  %number of neurons
for k=1:n
   S(k)=srti(neuro(k),x,p);
End
End
```

Compile this function, and in the Command Window, you can load the data, select a data point, and predict its classification by calling the *useper* function.

```
>> load fper
>> x=X(:,9)
>> y=Y(:,9)
>> s=useper(x)
```

2.7 Solution 2: Using the Matlab perceptron

In a new script, we define a function `percep()` and save the file as `percep.m`. Inside this function, we either copy or load the matrices `X` and `Y`, as they are stored in the `fper` file.

Next, the command `net = perceptron` is used to create a perceptron object with its initial configuration. The command `net = train(net, X, Y)` configures the network `net` using the provided data and performs the training process.

Once the perceptron is trained, it is ready to be saved and used in other scripts or applications.

```
function percep()
X=[4.9 5.1 4.7 4.6 ...
.....
.....
.... 2.1 2.4 2.3];
for i=1:12
Y(:,i)=[1;0;0];
end
for i=13:24
Y(:,i)=[0;1;0];
end
for i=25:36
Y(:,i)=[0;0;1];
end
save('data','X','Y');
net=perceptron;
net = train(net,X,Y);
save('net')
end
```

2.8 Solution 3: with Python

We will start by creating the file `perceptron1.py`, loading the necessary libraries, and then defining the base function in the same script using the keyword `def`.

2.8.1 Dataset Preparation

In the base function, which returns the input data X and the output Y of our perceptron, to construct the matrix X, we begin by creating a list containing 4 rows. Each row corresponds to the concatenation of the data from the three flower classes, which gives 36 elements per row. We then transform this list into a `np.array`. To implicitly handle the bias term in the perceptron, we add an additional row of ones to the matrix X.

```
import pickle as pk
import numpy as np

def data():
   D = [[4.9, 5.1, 4.7, 4.6, 5, 5.4, 4.6, 5, 4.4, 4.9, 5.4, 4.8,
      5.2, 5, 5.9, 6, 6.1,   5.6, 6.7, 5.6, 5.8, 5.6, 5.9, 6.1,
      6.3, 6.5, 7.6, 4.9, 7.3, 6.7, 7.2, 6.5,6.4, 6.8, 5.8, 6.4],
   [3.0, 3.5 ,3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.4,
      2.7, 2.0, 3.0 ,2.2,   2.9, 2.9, 3.1, 3.0, 2.7, 2.5, 3.2,
      2.8, 2.9, 3.0, 3.0, 2.5, 2.9, 2.5, 3.6,3.2, 2.7,   3.0,
      2.8, 3.2],\
   [1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5 ,1.4, 1.5, 1.5, 1.6,
      3.9, 3.5,  4.2, 4.0,4.7, 3.6, 4.4, 4.5, 4.1, 3.9, 4.8, 4.0,
       5.6, 5.8, 6.6, 4.5, 6.3, 5.8, 6.1,5.1,   5.3, 5.5, 5.1,
      5.3],\
   [0.2, 0.2, 0.2, 0.2, 0.3, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.2,
      1.4, 1.0,   1.5, 1.0,1.4, 1.3 ,1.4, 1.5, 1.0, 1.1, 1.8,
      1.3, 1.8, 2.2, 2.1, 1.7, 1.8, 1.8, 2.5,   2.0, 1.9, 2.1,
      2.4, 2.3]]

   X=np.array(D)
   d=np.ones((1,36))
   X=np.vstack([X, d])
   print(X.shape, X)
```

For the outputs, we create a `np.array` of size 3×36. For the first row, we replace the first 12 elements with ones. For the second row, the second dozen elements are replaced with ones. Finally, for the third row, the last 12 elements are replaced with ones. Then, we shuffle the order of X and Y in the same way to ensure that each element of X maintains its correspondence with Y, using the `np.random.permutation` function as follows:

```
    Y=np.zeros((3,36))
    for i in range(12):
        Y[0,i]=1
        Y[1,i+12]=1
        Y[2,i+24]=1
    ind = np.random.permutation(X.shape[1])
    X = X[:, ind]
    Y = Y[:, ind]
    return X,Y
```

2.8.2 Perceptron Initialization

We return to our main script and call the function `data` in order to retrieve the two matrices X and Y. We then extract their dimensions, such that the number of rows in X corresponds to the number of inputs p, and therefore to the number of weights connected to each neuron. Similarly, the number of rows in Y corresponds to the number of outputs, and thus to the number of neurons n.

Next, we create the class `Neuron`, which defines a neuron by its p randomly initialized weights w and its output s, initialized to 0. Finally, we define a list `neuro` of n neurons, each with reinitialized weights.

```
X,Y=data()
p = X.shape[0]
n = Y.shape[0]
class neurone:
    w=np.random.rand(p)
    s=0
neuro=[]
for i in range(n):
    neuro.append(neurone ())    # vector of neurons
    neuro[i].w=np.random.rand(p)
```

2.8.3 Training the Perceptron

We begin by writing the function `out`, which computes the output of a neuron *ner* given an input vector x of size p. The function calculates the weighted sum $\sum_{i=1}^{p} x_i w_i$. If this sum is greater than 0, the output of the neuron is 1; otherwise, the output of the neuron is 0.

```
def out ( ner ,x , p ):
    s =0
    for i in range(p):
```

```
        s = s + ner. w[i] * x[i]
    if (s >0):
        srt =1
    else:
        srt =0
    return srt
```

The learning iterations will stop either after 100 iterations or when there are no more training errors. In each iteration, we go through the 30 input vectors x_i (flowers), keeping 6 aside for testing. For each input vector, we compute the output of each neuron using the `out` function and compare the result with the desired output vector y_i. In case of an error, we increment the error counter and update the weights by applying the perceptron learning rule, with the learning rate α fixed at 0.05.

```
alpha=0.05
i =0
er =1
x1=np.zeros((p,1))
y1=np.zeros((n,1))
while i <100 and er > 0 :
    er =0
    i = i +1
    for j in range(30):
        x1 = X [: , j ]
        y1 = Y [: , j ]
        for k in range(n):
            S = out ( neuro [ k ] ,x1 , p )
            if not( S == y1 [k ] ):
                er = er +1
                for m in range(p): # updating weights
                    neuro[k].w[m]=neuro[k].w[m]+alpha*(y1[k]-S)*x1[m]
```

2.8.4 Testing and Saving the Perceptron

After completing the training loop, we evaluate the classification performance of our perceptron on the remaining six flower examples. We calculate the number of misclassifications, which can then be displayed.

```
ert =0
Sr=np.zeros((n,))
for j in range(30,36):
    x1 = X[:, j]
```

```
    y1 = Y[:, j]
    for k in range(n):
        Sr[k] = out ( neuro [ k ] ,x1 , p )
    if not( np.all(Sr == y1) ):
        ert = ert +1
print ( "␣the␣number␣of␣error␣is␣:␣",ert)
```

To enable future reuse of the trained model for flower classification tasks, we need to save it. Since our model is object-oriented, the most suitable library for this task is `pickle`. We open a binary file (e.g., `'fperceptron.pkl'`) in write mode using `'wb'` to preserve the internal structure of the perceptron object.

The perceptron object (named `neuro`) is saved to the file using `pickle.dump()`, and the file is then closed. Additionally, the feature and label data (`X` and `Y`) can be saved using `numpy`'s `np.save()` function.

```
fper=open ("fperceptron.pkl",'wb')
pk.dump(neuro,fper)
np.save("Xdata",X)
np.save("Ydata",Y)
fper.close()
```

2.8.5 Using the Perceptron

We create a new file named `use1.py` in which we import the `numpy` and `pickle` libraries. We also import the class `neurone` and the `out` function from the previous file, `perceptron1`.

Next, we define the function `use1`, which takes a vector `x1` as input, representing a flower, and returns its classification. Inside the function, we open the binary file `fperceptron.pkl` in read mode, and load the saved perceptron (named `neuro`) using `pickle.load()`. We also load the data matrices `X` and `Y` in order to retrieve the necessary dimensions. The function then loops over the number of neurons and calls the function `out`for each one to compute the output.

```
import numpy as np
import pickle as pk
from perceptron1 import neurone,out
def use1(x1):
    fper=open('fperceptron.pkl','rb')
    percept=pk.load(fper)
    X=np.load("Xdata.npy")
    Y=np.load("Ydata.npy")
    fper.close()
    p = X.shape[0]
```

```
    n = Y.shape[0]
    Sr = np.zeros((n,))
    for k in range(n):
        Sr[k] = out(percept[k], x1, p)
    return Sr
```

To classify a new flower, we go to the terminal or a separate Python file, import `numpy` and the `use1` function, prepare an input vector `x`, and call `use1(x)`. The function will return the classification result for the flower represented by `x`.

```
In [2]: import numpy as np
In [3]: from use1 import use1
In [4]: X = np.load("Xdata.npy")
In [5]: Y = np.load("Ydata.npy")
In [6]: x1=X[:,31]
In [7]: y1=Y[:,31]
In [8]: use1(x1)
```

The result will be displayed directly in the terminal, for example:

```
Out[11]:
array([[1.],
       [0.],
       [0.]])
```

Lab 3

Multilayer Neural Network with Matlab

3.1 Required work

Implement a gradient backpropagation neural network in MATLAB for handwritten letter classification using our dataset bs, that consists of a collection of images, where each image contains a letter of the alphabet, bs was created by me and the students of the first GI (Industrial Engineering) cohort of our school ESSAT.

1. The image database is located in the **bs** folder, it contains **12*26** images, each image contains a handwritten letter, the file name indicates the letter concerned. There are **12 images** of each letter.

2. Start by copying the **bs** folder inside your work folder and creating a new function "function net1() " and save it as net1.m in the work folder.

3. Data collection: File names make it easier for us to read images automatically by loops **for** to build the input and output tables of our network. To do this, each image must be resized to the size [**20 14**] then binarized to reduce the amount of data. For image processing.

4. Transform the binary images into vectors xdata, and prepare the output vectors ydata.

5. Use 10 images of each letter for training, and leave two images of each letter for testing.

6. Save the matrix xdata, ydata, xtest.

7. Load the data into the workspace, create a neural network graphically using the Matlab Toolbox, train it, and then visualize the training results.

8. Create a layered neural network using Matlab's predefined functions, train it, and visualize the results. Perform an implementation with a single hidden layer and another with two

hidden layers. Change the parameters each time (the number of neurons in each layer, the activation functions and see the best results) Train it and visualize the learning results.

9. save the models and the splited data, and use them in another script or in command windows.

10. To improve the results, split the images into 4 rows and 3 columns, and sum each piece, and feed these results to the RNs for training and testing.

Useful functions and informations

1. save, load, reshape;

2. sm(k,l)=sum(sum(imb(p:p+4,q:q+4))); to sum a portion of the image (matrix).

3. net=feedforwardnet(nbr); this function creates a multi-layer network.

4. net=configure(net...

5. net.trainFcn=... define the learning algorithm (gradient back propagation)

6. net.layers{1}.transferFcn=..., define the activation function (logistic or threshold or other)

7. [net,tr]=train(net...) start the training by specifying the data (X,Y)

8. y=sim(net,data) simulation for the test.

3.2 Solution

We start by extracting our image database into a folder named bs inside our working directory. Set the path to our working directory and create a new function function data() in a file named data.m.

3.2.1 Data Preparation

We proceed by reading the images one by one using a for loop that reads the 12 images of each letter inside another for loop that iterates through the letters from ‘a’ to ‘z’. We construct the name of an image file by concatenating the folder ’bs/’ with the letter, the number, and the extension ’.jpg’.

We use the imread function to read the image file and convert it into a matrix, the function imresize to resize the image to 20x14, the function im2bw for image binarization (as a result, the image will contain only ‘0’s and ‘1’s), and then the reshape function to transform matrices into vectors.

Neural networks in Matlab receive input and output data in the form of vectors, so we will construct the output vectors as we process the inputs and save the resulting matrices to be used by neural networks as follows:

```
function data()
j=0;     %number of classes
n=0;     %number of images in the set
for c='a':'z'
j=j+1;
for i=1:12
n=n+1;
nm=['bs/' c num2str(i) '.jpg'];
im=imread(nm);
m=imresize(im,[20,14]);
imb=im2bw(m);
xdata(:,n)=reshape(imb,1,280);
ydata(j,n)=1;
end
end
save ('Data','xdata','ydata');
end
```

Unlike Python, we are not required to initialize matrices to zero before progressively filling them. When we set ydata(j,n) = 1;, Matlab automatically expands the matrix to size [j, n], sets the last position to 1, and fills the rest with zeros. The repetition of this instruction results in a matrix where each column contains a single 1 and 25 zeros. The 1 corresponds to the position of the respective letter.

For xdata, we add a new column each time, corresponding to the image vector. Both matrices are then saved in a file named Data.mat.

3.2.2 Using Matlab's Toolbox to Create a Neural Network

After executing the previous code, the Data.mat file appears in the Current Folder. Double-click on it to load it into the Workspace. You will notice two matrices: xdata (280x312) and ydata (26x312).

To use Matlab's Toolbox, go to the APP menu, click on the arrow on the right to see all applications, and select "Neural Net Pattern Recognition". The result will be as shown in figure 3.1: The Toolbox suggests a two-layer network: a hidden layer and an output layer. For the configuration, click Next, and the result will be as shown in figure 3.2.

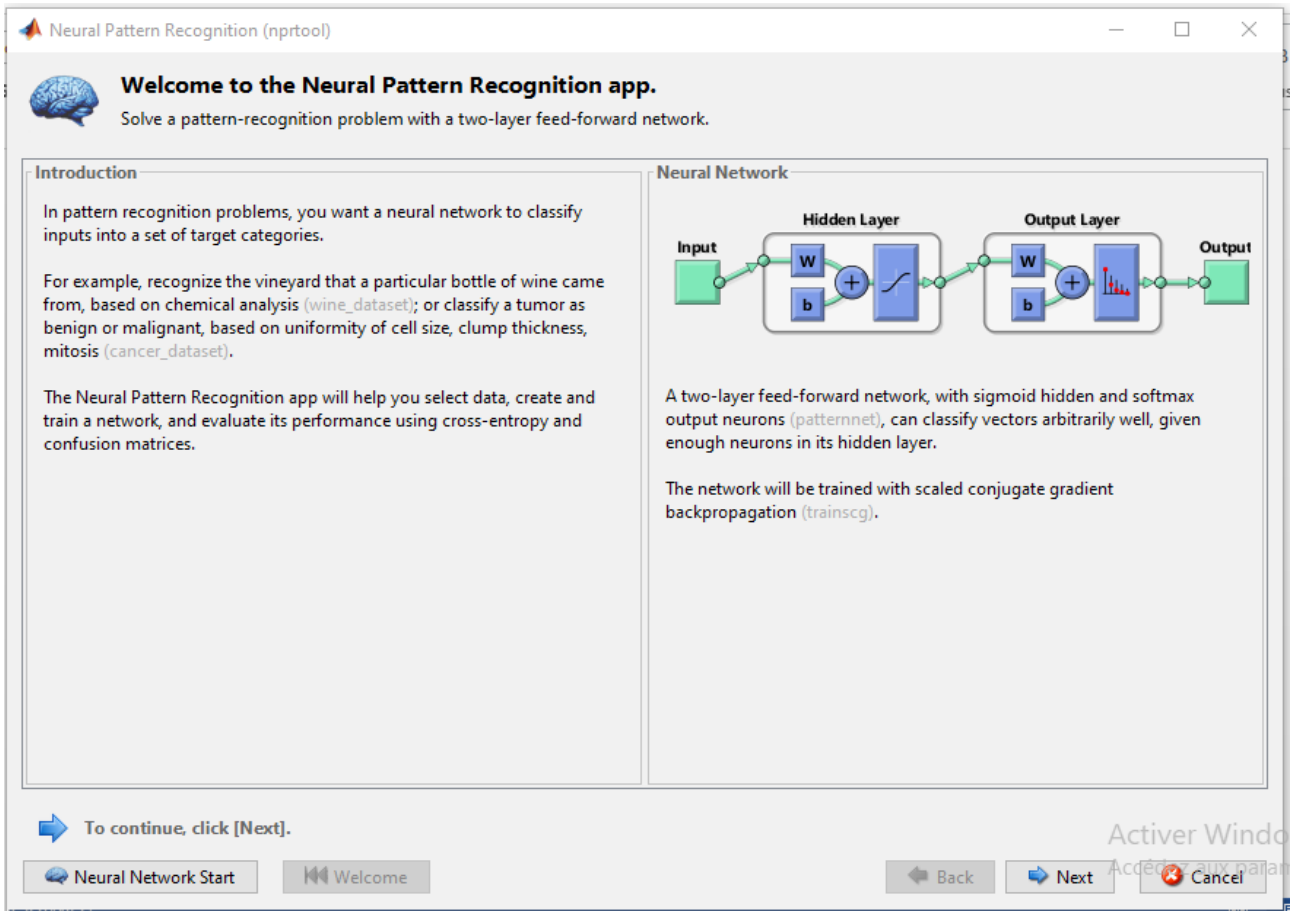

Figure 3.1: The pattern recognition neural network offered by default by Mtalab Toolbox

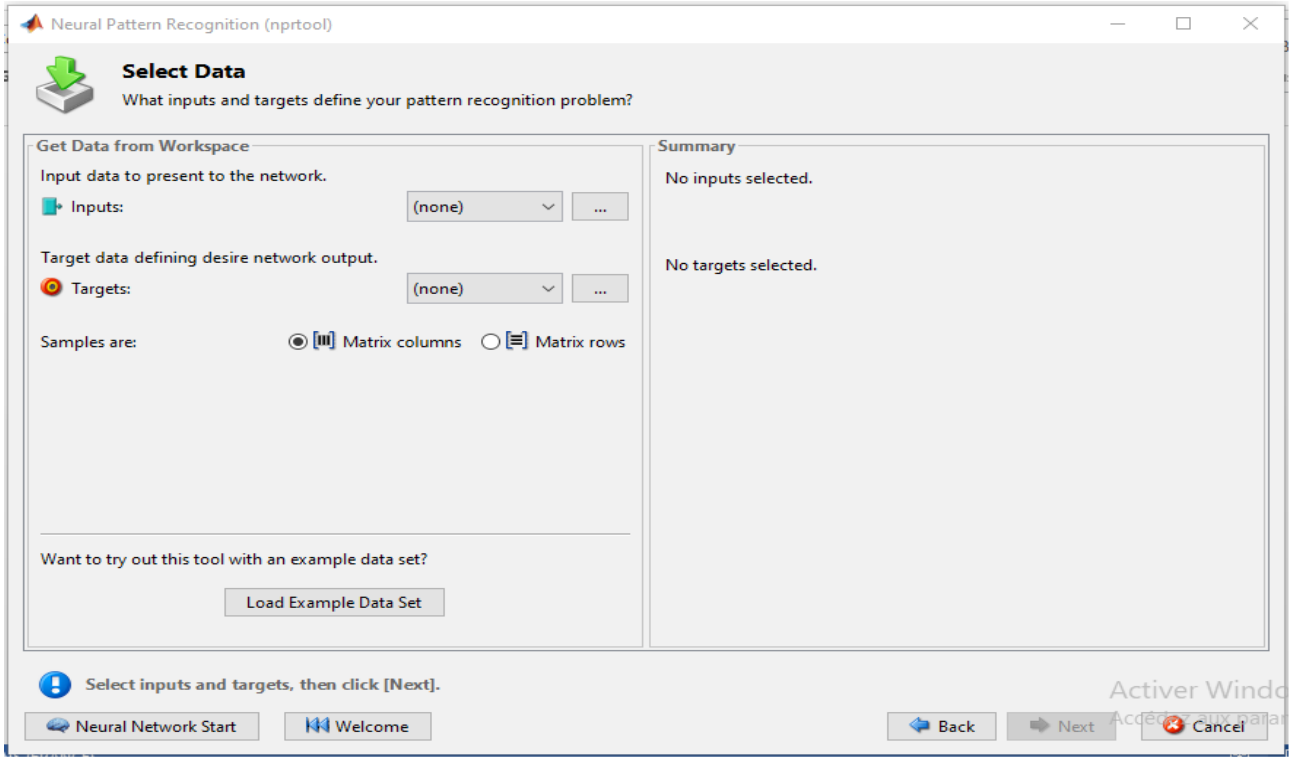

Figure 3.2: The second window for selecting inputs and outputs

Go to the Inputs dropdown menu and select xdata, then go to the Targets dropdown menu

and select ydata. Information about these two matrices will be displayed in the Summary section on the right side of the window as shown in figure 3.3.

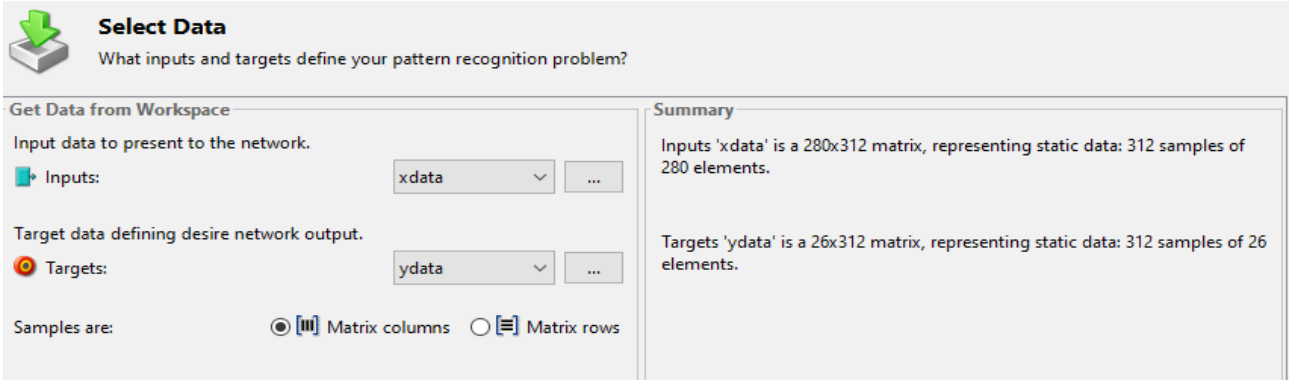

Figure 3.3: The third window after selecting inputs and outputs

Click Next, and you will see that our dataset is divided into three parts, with adjustable percentages: 70% for training, 15% for validation, and 15% for testing. Explanations about the role of each part are provided on the right, as presented in figure 3.4.

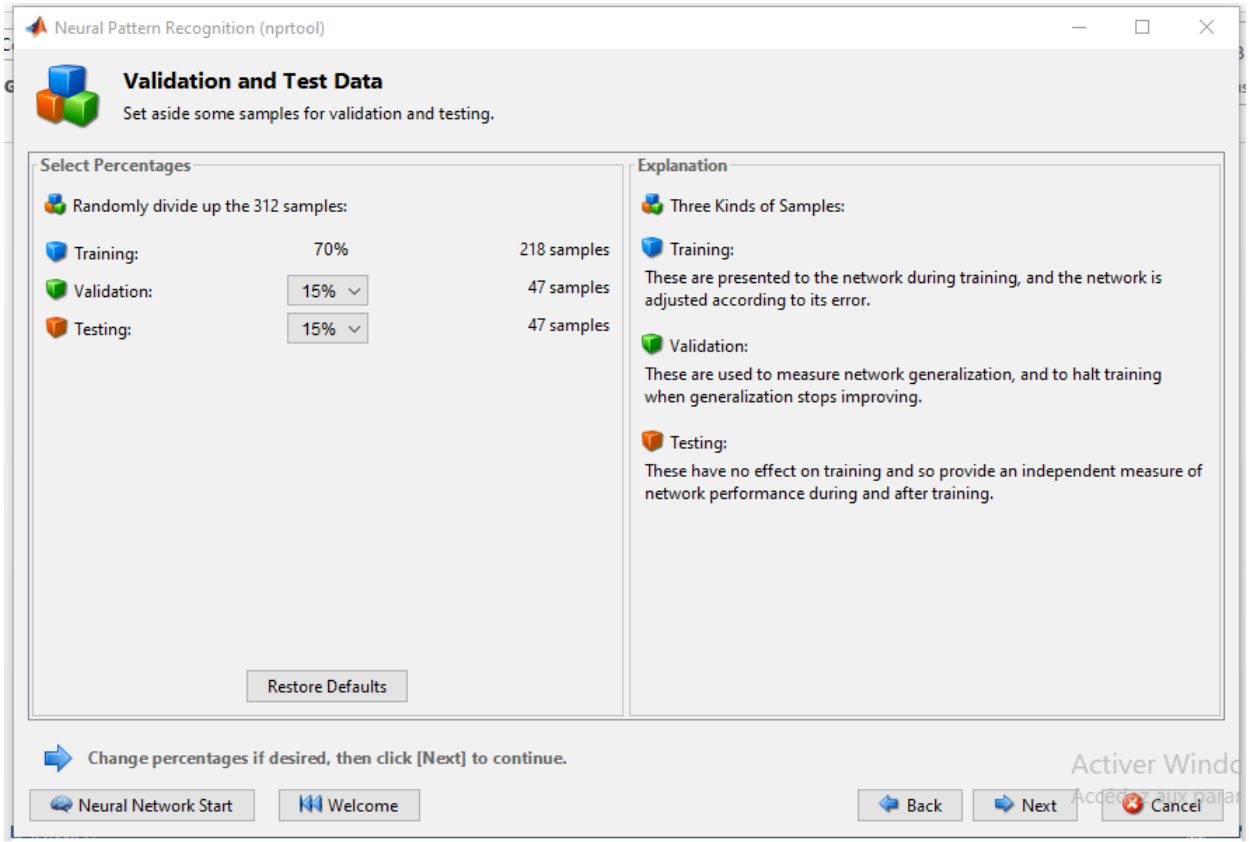

Figure 3.4: Division of the dataset into three parts (training, validation, and testing)

By clicking on "Next," our network configuration is displayed with 10 neurons proposed by default in the hidden layer, 280 inputs, and 26 neurons in the output layer. The last two pieces of information are derived from the xdata and ydata matrices, as shown in the figure 3.5.

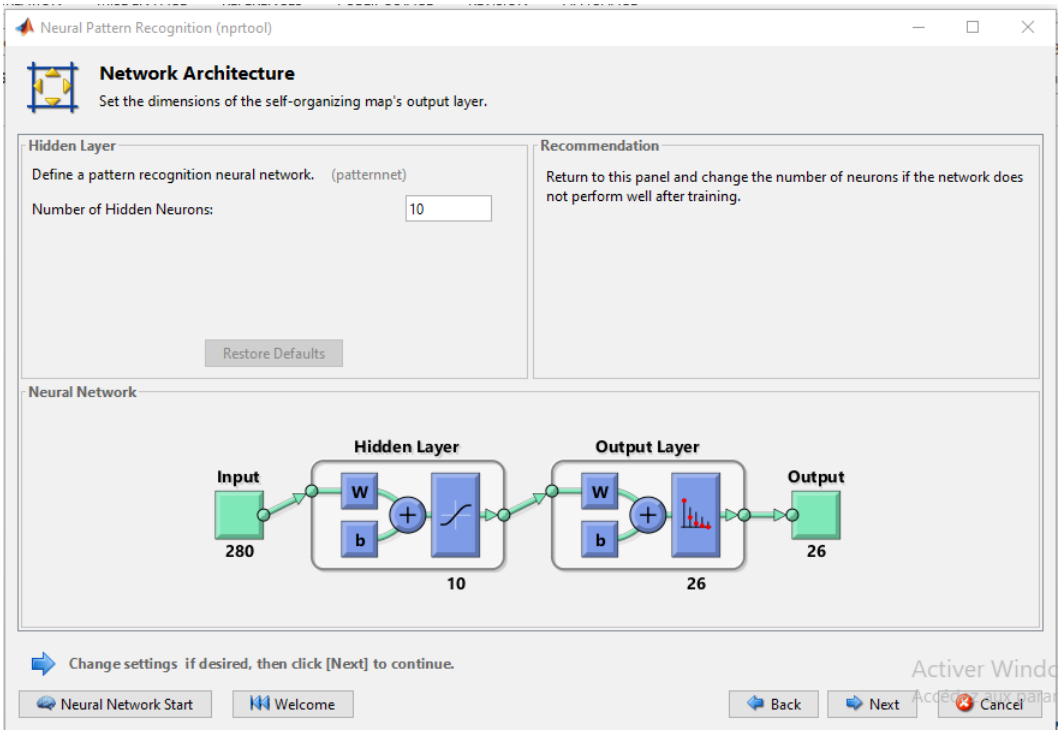

Figure 3.5: Configuration of the neural network

After the configuration, by clicking on "Next," the following figure 3.6 allows us to start the training by clicking on "Train."

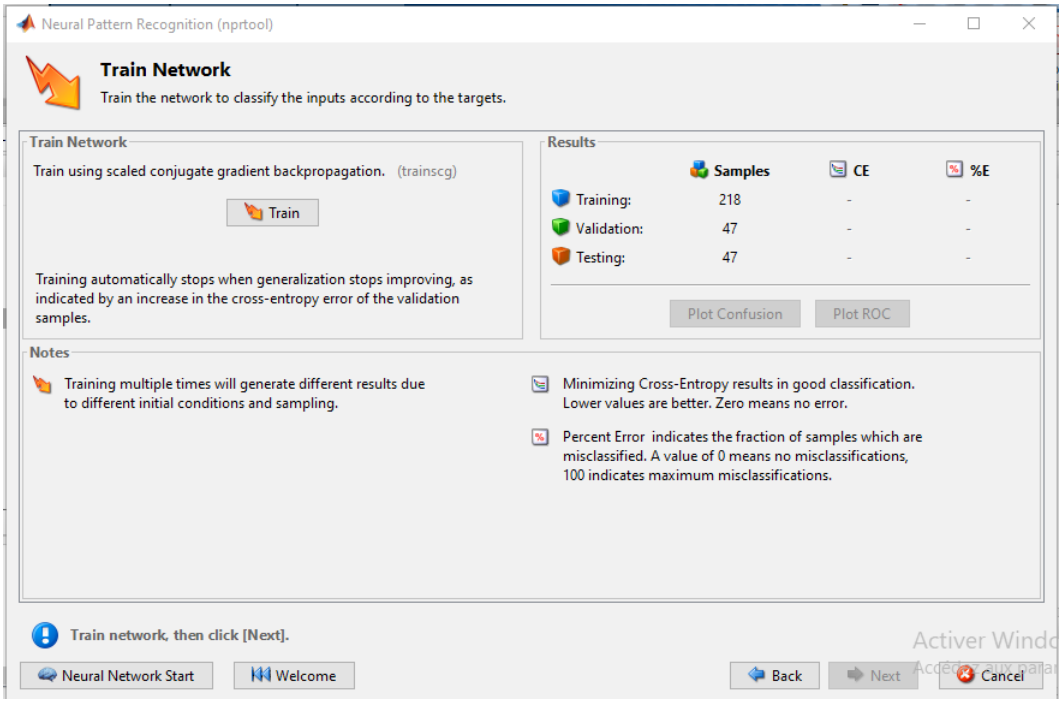

Figure 3.6: The network is ready for training

After the training, a window displaying statistics about the training results is shown (see figure 3.7). It includes the number of iterations taken by the training, the training time, the gradient, the performance, etc.

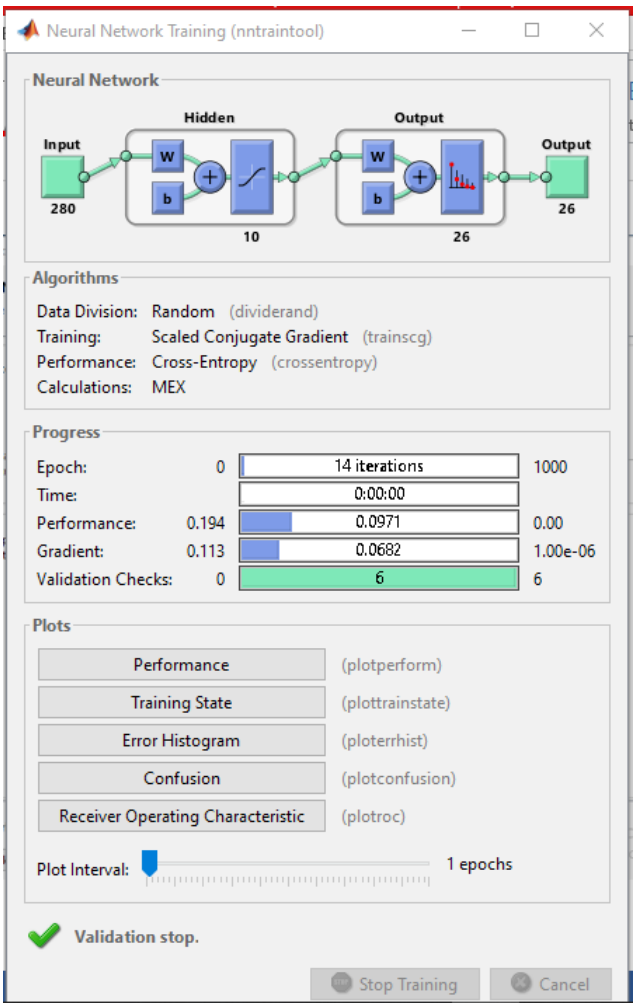

Figure 3.7: Resuts and statistics after training

By clicking on the performance button, we get more details, as shown in the figure 3.8

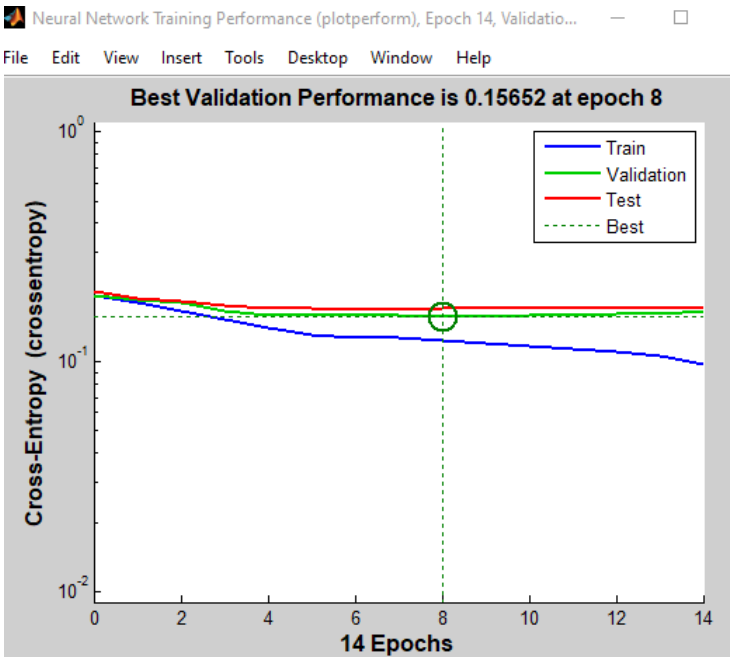

Figure 3.8: Evolution of the training, validation, and test errors

This image (figure 3.8) shows the details of the training epochs. The three curves indicate the evolution of the training, validation, and test errors. All three curves are decreasing, and the decision to stop the training is made when the validation curve stops decreasing and starts to rise to avoid overfitting.

3.2.3 Using MATLAB's Predefined Functions

To create a neural network using MATLAB's predefined functions, we can either continue with the same script or create a new one. In the latter case, the data must be loaded using load Data.

We will create a new function net1 in a file named net1.m and load the data. Then, we will create a neural network using the function feedforwardnet([60,30]). The parameter of this function specifies the number of neurons in the hidden layers. If it is a scalar, it indicates a single hidden layer. In our case, it is a network with two hidden layers, containing 60 and 30 neurons, respectively.

The configure function is used to set the number of inputs and output layer neurons according to the training data xdata and ydata.

We can modify the activation function of the desired layers or change the training algorithm as follows:

```
function net1()
load Data
net = feedforwardnet([60,30]);
net = configure(net,xdata,ydata);
net.layers{3}.transferFcn='softmax';
net.trainFcn = 'trainlm';
[net,tr] = train(net,xdata ,ydata);
save('net');
view(net)
end
```

Next, a training window is displayed (figure 3.9), showing information about the training performance. The view(net) function displays the overall structure of our neural network as shown in figure 3.10.

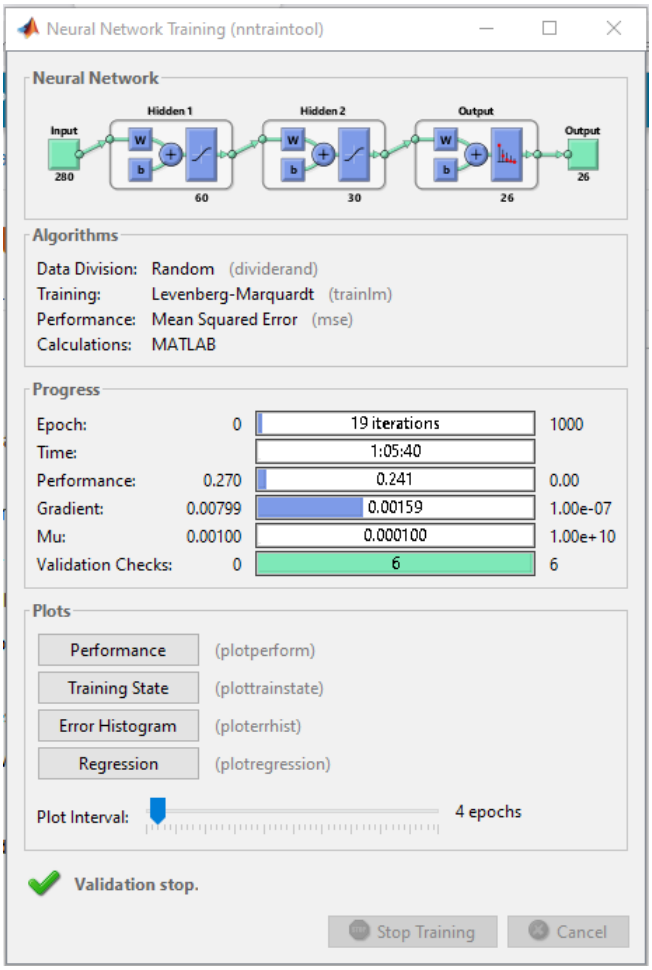

Figure 3.9: Resluts of training of the new neural network

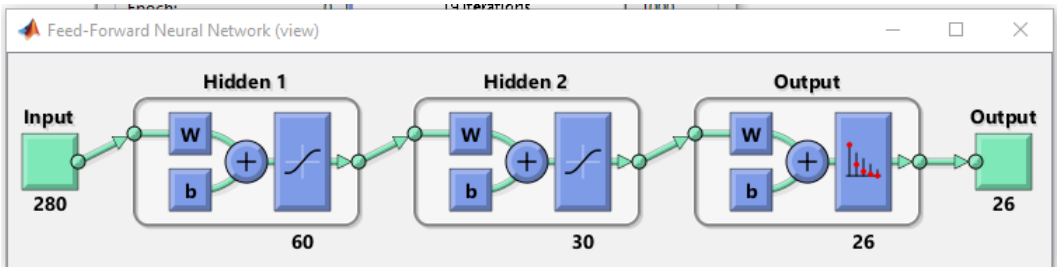

Figure 3.10: Configuration detail of the new neural network

Once the neural network is saved, it can be loaded and used in another script or in the command interpreter using the sim function or by calling the neural network by its name as follows:

```
function usenetwork()
load Data;
load net;
x=xdata(:,29);
y=net(x);
y1=sim(net,x);
```

```
[n,clas]=max(y);
cl=char(clas+96);
h=msgbox(['The␣handwritten␣caracter␣is␣classified␣as␣:␣␣␣' cl],'
   Result');
end
```

In this function, we selected a single column from the xdata matrix, specifically column 29, which actually represents a 'C'. By presenting this data to the neural network (net), either by its name or using the sim function, the result is a 26-element vector containing the probability for each letter. We then identified the element with the highest probability, converted it into a letter, and displayed it in a message box, which produced the following result.

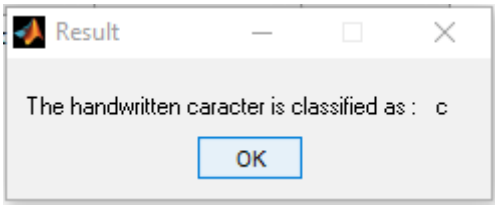

Figure 3.11: The result of our NN classification in msgbox

Lab 4

Multilayer Neural Network with Python

4.1 Presentation of libraries

For this workshop dedicated to multilayer neural networks, we will use two libraries widely used for machine learning. Scikit-Learn and TensorFlow are two major libraries for Machine Learning and artificial intelligence in Python.

Scikit-Learn is more oriented towards classic models (MLP, SVM, decision trees, KNN, regressions, etc.).

TensorFlow is designed for deep neural networks (Deep Learning), and can also be used to create an MLP network.

Each of these libraries generates datasets that can be imported and used for our neural networks.

sklearn datasets	
load_iris()	Flower classification, 150 samples
load_digits()	1797 images of handwritten digits 8×8
load_breast_cancer()	Breast cancer detection. 150
tensorflow datasets	
mnist	70,000 handwritten digit images 28×28
fashion_mnist	70,000 clothing images
cifar10	60,000 color images. 10 classes
cifar100	60,000 color images. 100 classes

4.1.1 Project Creation and Installation

1. Open PyCharm and Click on File → New Project
2. Choose a folder where to save your project → type the name of your-project and Click on "Create"

3. Go to File →Settings → Project : click on your-project
4. In the window on the right click on: Python Interpreter and Click on + (Add a package)
5. Search for scikit-learn, tensorflow and opencv-python and Click on "Install Package"

4.1.2 Get familiar with the scikit-learn and tensorflow libraries

scikit-learn

Start by creating a new file scikitTP.py and importing the libraries:

```
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report
```

The **MLPClassifier** function is used to create a multilayer neural network for classification. The **StandardScaler** function is used to center and scale the data. To evaluate the performance of a classification model, three metrics are commonly used: **accuracy-score**, **confusion-matrix** and **classification-report**.
To test sklearn we will use the dataset of handwritten digits load-digits(), we will import it as follows:

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
```

To split the dataset into two parts, a 'train' part for training, and a test part, sklearn has a function *train_test_split*.
Then, we will load the dataset of handwritten digits

```
digits = load_digits()
X, y = digits.data, digits.target
```

The following code displays the first ten images as subplots organized in a grid of 2 rows and 5 columns:

```
plt.figure(figsize=(8, 4))
for i in range(10):
plt.subplot(2, 5, i + 1)
plt.imshow(digits.images[i], cmap="gray")
plt.axis("off")
plt.show()
```

To normalize the data and put the pixels between -1 and 1;

```
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

Separate training data and test data:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)
```

Create the neural network with two hidden layers that contain 64 and 32 neurons respectively.

```
mlp = MLPClassifier(hidden_layer_sizes=(64, 32), activation='
    relu', solver='adam', max_iter=500, random_state=42)
```

Then start training

```
mlp.fit(X_train, y_train)
```

Afterwards, we can test our network by performing the classification of the test data.

```
y_pred = mlp.predict(X_test)
```

At the end, we can display the statistics on the results; the precision and the confusion matrix...

```
print(f"Precision : \{accuracy_score(y_test, y_pred) * 100:.2
    f\}%")
print("\nMatrice confusion :\n", confusion_matrix(y_test,
    y_pred))
print("\nRapport de classification :\n",
    classification_report(y_test, y_pred))
```

tensorflow

Then we do the same work with the tensorflow library in a new file tensorTP.py :

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

(X_train, y_train), (X_test, y_test) = mnist.load_data()
print(X_train.shape)  #(60000, 28, 28)  60k images de 28x28
    pixel
# Normaliser et aplatir les images
X_train = X_train.reshape((60000, 28*28)).astype("float32") /
    255.0
X_test = X_test.reshape((10000, 28*28)).astype("float32") /
    255.0
```

```
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
# Definir le modele MLP
model = Sequential([
Dense(128, activation='relu', input_shape=(28*28,)),  #
   Entree aplatie
Dense(64, activation='relu'),
Dense(10, activation='softmax')  # 10 classes (chiffres 0-9)
])
# Compiler et entrainer
model.compile(optimizer='adam', loss='
   categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=32,
   validation_data=(X_test, y_test))
```

4.2 Work requested

Realize a multi-layer neural network implementation with gradient backpropagation for handwritten character recognition.

1. Using an image database created by former students; The image database is located in the **bs** folder, it contains **12*26** images, each image contains a handwritten letter, the file name indicates the letter concerned. There are **12 images** of each letter.

2. Start by copying the **bs** folder inside your project folder and creating a new file "*net*1.*py*".

3. Data collection: File names make it easier for us to read images automatically by loops **for** to build the input and output tables of our network. To do this, each image must be resized to the size [**20 14**] then binarized to reduce the amount of data. For image processing, you can use the *OpenCv* or *matplotlib* library

4. Transform the binary images into vectors, and prepare the output vectors.

5. Use 10 images of each letter for training, and leave two images of each letter for testing.

6. Create a neural network using the **sklearn** library, then visualize the results on the test data.

7. Create a neural network using the **tensorflow** library, then visualize the results on the test data.

8. save both models and the data, and use them in another script.

9. To improve the results, split the images into 4 rows and 3 columns, and sum each piece, and feed these results to the RNs for training and testing.

Useful informations

- To read, resize, binarize and display an image use the **imread**, **resize**, **threshold** and **imshow** functions of OpenCV as follows:

```
import numpy as np
import cv2
nm="bs/a1.jpg"
m=cv2.imread(nm,cv2.IMREAD_GRAYSCALE)
mr=cv2.resize(m,(14,20))
ret, imbin = cv2.threshold(mr, 127, 1, cv2.
   THRESH_BINARY)
cv2.imshow('image',imbin*255)
cv2.waitKey()
```

- To save and load an RN created by sklearn, use the dump and load functions of the joblib library as follows:

```
import joblib
joblib.dump(modele1, 'nomFichier.pkl')
modele12 = joblib.load('nomFichier.pkl')
```

- To save and load a RN created by tensorflow, use the save and load functions of tensorflow.keras as follows:

```
from tensorflow.keras.models import load_model
modele2.save('nomFichier2')
modele22 = load_model('nomFichier2')
```

- To save and load a matrix of data;

```
import numpy as np
np.save('xtest.npy', xtest)
xtest=np.load('xtest.npy')
```

4.3 Solution

After creating the two files scikitTP.py and tensorTP.py in the project and testing the scikit-learn and TensorFlow libraries for training and classifying handwritten digits using the datasets

available in the libraries, we will now move on to the classification of handwritten characters using both libraries and our dataset prepared by the students.

Our dataset, bs, consists of a collection of images, where each image contains a letter of the alphabet. We have 12 images for each letter, resulting in a total of 12 × 26 images. The name of each image file is composed of the corresponding letter followed by its sequence number and the .jpg extension. For example: (a1.jpg, ..., a12.jpg), which facilitates processing.

Start by extracting the bs dataset, copying it into the project folder, and opening a new script named net1.py in the same project folder.

Then, copy the previously used libraries into the file and add OpenCV (cv2) and joblib as follows:

```
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np
import cv2
import joblib
```

4.3.1 Preparation of Input Vectors

Since we have 12 images per letter, we can use 10 images per letter for training ($10 \times 26 = 260$ images) and $2 \times 26 = 52$ images for testing.

Each image has a resolution of 20 × 14 pixels, meaning the images will be converted into vectors of $20 * 14 = 280$ elements.

The images will be presented to the neural network in row vector format, so the input training data will be a matrix of 260 rows (images) and 280 columns. Similarly, the input test data will be a matrix of 52 × 280.

We will initialize these matrices and a vector for a single image with zeros using np.array before filling them as follows:

```
xtrain=np.zeros((260,280))
xtest=np.zeros((52,280))
imvec=np.zeros((1,280))
```

4.3.2 Preparation of Output Vectors

The output vector of Sklearn's MLP is a column vector, where the size corresponds to the number of examples : 260 for training (ytr) and 52 for testing (yts). The values range from 0 to 25, where 0 corresponds to 'a' and 25 corresponds to 'z'.

Since the output layer of our neural network contains 26 neurons, the MLPClassifier in Sklearn automatically transforms the y vector into a matrix of 0s and 1s using one-hot encoding.

For TensorFlow's neural network output, we must prepare the labels using one-hot encoding or call the to_categorical function from Keras to handle this transformation.

For this task, we will manually prepare ytrain and ytest. So, we will initialize vectors of size 260×26 for training and 52×26 for testing.

```
ytr=np.zeros((260,1))
yts=np.zeros((52))
ytrain=np.zeros((260,26))
ytest=np.zeros((52,26))
```

4.3.3 Image Preparation

To iterate through the entire set of images, we use a for loop that goes through the letters from 'a' to 'z', which corresponds to ASCII codes 97 to 123. However, since Python does not include the last element in a loop range, it will stop at 122.

Inside this loop, we use another loop to iterate over the 10 images for each letter. Within this second loop, we can construct the image filenames by concatenating the letter, the number, and the extension '.jpg'.

Once the filenames are prepared, we can load the images into matrices using OpenCV's imread function and display them to ensure that the reading process is error-free. This can be done using the following code:

```
j=0;
i=0
for c in range(97,123):
    for d in range(1,11):
        nm="bs/" + chr(c)+str(d)+".jpg"
          print(nm)
        m=cv2.imread(nm,cv2.IMREAD_GRAYSCALE)
        cv2.imshow('image',m)
        cv2.waitKey()
```

If everything works as expected, we will no longer need to display the images, so the last two lines should be commented out using #.

Next, we proceed with resizing, binarization, and then transforming each image into a vector as follows:

```
for c in range(97,123):
    for d in range(1,11):
        nm="bs/" + chr(c)+str(d)+".jpg"
        m=cv2.imread(nm,cv2.IMREAD_GRAYSCALE)
```

```
        #cv2.imshow('image',m)
        #cv2.waitKey()
        mr=cv2.resize(m,(14,20))
        ret, imbin = cv2.threshold(mr, 127, 1, cv2.
           THRESH_BINARY)
        imvec=imbin.reshape(( 20*14)).astype("float32")
        xtrain[i]=imvec
        ytrain[i,j]=1
        ytr[i]=j
        i=i+1
    j=j+1;
```

Where i counts the number of examples (images) and j represents the number of classes. The row vector of each image is added to the xtrain matrix, while the outputs for both networks, ytr and ytrain, are constructed in parallel.

Similarly, we fill the test matrices and vectors: xtest, yts, and ytest, as follows:

```
j=0;
i=0
for c in range(97,123):
    for d in range(11,13):
        nm="bs/" + chr(c)+str(d)+".jpg"
          m=cv2.imread(nm,cv2.IMREAD_GRAYSCALE)
        mr=cv2.resize(m,(14,20))
        ret, imbin = cv2.threshold(mr, 127, 1, cv2.
           THRESH_BINARY)
        imvec=imbin.reshape(( 20*14)).astype("float32")
        xtest[i]=imvec
        ytest[i,j]=1
        yts[i]=j
        i=i+1
    j=j+1;
```

4.3.4 Use of MLPClassifier from Sklearn

We create the neural network, choosing two hidden layers: the first with 64 neurons and the second with 32 neurons. Students can experiment with these values, as well as with the choice of the optimization function (solver), activation function, number of iterations, etc.

The fit function (the training function) automatically adjusts the number of neurons in the output layer based on the values of ytr.

```
mlp1= MLPClassifier(hidden_layer_sizes=(64, 32), activation='
   relu', solver='adam', max_iter=500, random_state=42)
```

```
    mlp1.fit(xtrain, ytr)
```

We make predictions on the test data and analyze the results and the effectiveness of our network.

```
y_pred = mlp1.predict(xtest)
print(f"Precision : {accuracy_score(yts, y_pred) * 100:.2f}%")
print("\nMatrice de confusion :\n", confusion_matrix(yts, y_pred)
   )
print("\nRapport de classification :\n", classification_report(
   yts, y_pred))
```

The results for this configuration are as follows.

```
Precision : 34.62%

Matrice de confusion :
[[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0]
[0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0]]
```

```
Rapport de classification :
precision    recall  f1-score   support

0.0         1.00      0.50      0.67         2
1.0         1.00      0.50      0.67         2
2.0         0.50      0.50      0.50         2
3.0         0.50      0.50      0.50         2
4.0         1.00      1.00      1.00         2
5.0         0.00      0.00      0.00         2
6.0         0.00      0.00      0.00         2
7.0         0.33      0.50      0.40         2
8.0         0.00      0.00      0.00         2
9.0         0.00      0.00      0.00         2
10.0        0.00      0.00      0.00         2
11.0        0.00      0.00      0.00         2
12.0        0.00      0.00      0.00         2
13.0        1.00      0.50      0.67         2
14.0        1.00      0.50      0.67         2
15.0        1.00      0.50      0.67         2
16.0        0.67      1.00      0.80         2
17.0        0.50      1.00      0.67         2
18.0        0.00      0.00      0.00         2
19.0        0.00      0.00      0.00         2
20.0        0.00      0.00      0.00         2
21.0        0.33      0.50      0.40         2
22.0        0.33      0.50      0.40         2
23.0        0.33      1.00      0.50         2
24.0        0.00      0.00      0.00         2
25.0        0.00      0.00      0.00         2

accuracy                           0.35        52
macro avg         0.37      0.35      0.33        52
weighted avg      0.37      0.35      0.33        52
```

4.3.5 Use of a Neural Network with TensorFlow

The implementation of a sequential neural network with TensorFlow is done step by step. For the first layer, we specify the number of neurons (e.g., 128), the activation function, and the input shape (a row vector of 280 elements). For the other layers, we only specify the number of neurons and the activation function. Then, we compile the model by choosing the optimization

algorithm, the loss function, and the metric for statistical evaluation. Finally, we start the training using the fit function.

```
model2 = Sequential([
Dense(128, activation='relu', input_shape=(20*14,)),  # flat
   input
Dense(64, activation='relu'),
Dense(26, activation='softmax')  # 26 classes
])
model2.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
model2.fit(xtrain, ytrain, epochs=150, batch_size=32,
   validation_data=(xtest, ytest))
```

After that, we can save our models and data files to use them later in another script.

```
joblib.dump(mlp1, 'mlp1.pkl')
model2.save('mod2.h5')
np.save('xtrain.npy', xtrain)
np.save('xtest.npy', xtest)
np.save('ytrain.npy', ytrain)
np.save('ytest.npy', ytest)
np.save('ytr.npy', ytr)
np.save('xts.npy', yts)
```

4.3.6 The use of our Neural Networks

We can create a new script, use.py, load our neural networks, and use them to predict the outputs of the images we want. The output of each network and the predicted letter will be displayed.

```
import joblib
import numpy as np
from tensorflow.keras.models import load_model
xtest=np.load('xtest.npy')
mdel2=load_model('mod2.h5')
mlp1=joblib.load('mlp1.pkl')
x=xtest[5:6,:]   # for example the image 5 of the test set
y1=mlp1.predict(x)
print ('y1=', y1)
y=int(y1[0])
print("prediction␣de␣mlp1␣:␣"+ chr(y+97))
y2=mdel2.predict(x)
```

```
print('y2= ', y2)
y=np.argmax(y2)
print("prediction de  model2 : "+ chr(y+97))
```

As a result, both networks successfully made a correct prediction (letter 'c' for test image 5) as follows:

```
y1= [2.]
prediction de mlp1 : c
1/1 [==============================] - 0s 123ms/step
y2=  [[1.1501058e-04 8.4992089e-06 9.7744691e-01 3.0962310e-06
   2.2244127e-02 1.0939131e-04 1.7796359e-06 4.4667909e-06
   1.0639982e-05 8.4583100e-08 4.2569845e-08 3.2522948e-06
   3.6532111e-09 5.6504522e-08 3.6053651e-05 6.1070938e-07
   3.1297717e-10 6.9386618e-07 1.0498806e-05 1.1955022e-08
   6.9436732e-07 2.2996599e-06 1.4050500e-08 5.0859818e-08
   2.4745783e-10 1.8001140e-06]]
prediction de  model2 : c
```

4.3.7 Cropping images

As an attempt to improve the results, we were asked to divide each image into 4 rows and 3 columns and compute the sum of these image segments. This would provide the neural networks with row vectors of size 4×3 only and allow us to observe the results. To perform the segmentation and prepare the new dataset, we only need to modify the first part of the program as follows:

```
for c in range(97,123):
    for d in range(1,11):
        nm="bs/" + chr(c)+str(d)+".jpg"
        print(nm)
        m=cv2.imread(nm,cv2.IMREAD_GRAYSCALE)
        mr=cv2.resize(m,(14,20))
        ret, imbin = cv2.threshold(mr, 127, 1, cv2.THRESH_BINARY)
        print(imbin.shape)
        for p in range(4):
            for q in range (3):
                if q<1:
                    mat1=np.zeros((5,5))
                    mat1=imbin[p*5:(p+1)*5,q*5:(q+1)*5]
                else:
                    mat1 = np.zeros((5, 4))
```

```
                mat1 = imbin[p * 5:(p + 1)* 5, q * 5: (q + 1)*4]
            matsom[p,q]=np.sum(mat1)
        vecsom=matsom.reshape(( 4*3)).astype("float32")
        smtrain[i]=vecsom
```

Lab 5

Fuzzy Logic and Fuzzy System

5.1 Required work

Implement a fuzzy system for controlling an inverted pendulum in Matlab.

5.1.1 The Inverted Pendulum

The studied inverted pendulum (see figure 5.1)consists of a mobile cart that moves along a horizontal axis while supporting a freely rotating pendulum. The pendulum is attached to the cart and can rotate within an angle range of$[-90^o, +90^o]$. The primary objective is to keep the pendulum balanced in an upright (vertical) position despite disturbances and the inherent instability of the system.

This system is a classic example in control theory and robotics, often used to test and develop advanced control strategies such as fuzzy logic, PID controllers, or reinforcement learning algorithms. Maintaining the pendulum in a vertical position requires continuously adjusting the cart's movement to counteract the pendulum's tendency to fall.

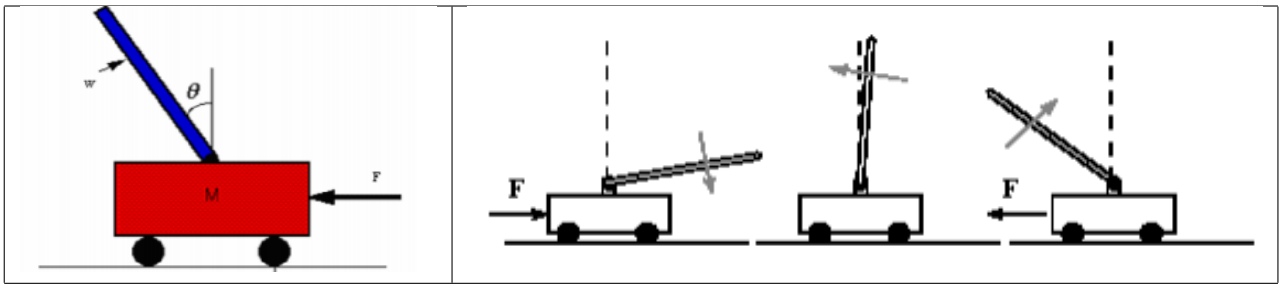

Figure 5.1: The Inverted Pendulum

5.1.2 Inputs

The system's inputs are the angle θ of the pendulum and its angular velocity w. The angle θ represents the deviation of the pendulum from the vertical position, while the angular velocity w indicates the speed and direction of its movement.

5.1.3 Outputs

The system's output is the force F applied to the cart. This force is responsible for adjusting the cart's motion along the horizontal axis to counteract the pendulum's movement and stabilize it. The control system must determine the appropriate magnitude and direction of F based on the input values to ensure that the pendulum remains upright.

5.1.4 Membership Functions

To model the fuzzy logic control system, triangular membership functions are used to represent the linguistic variables associated with the inputs and outputs as chown in figure 5.2. These functions allow for a smooth transition between different fuzzy sets, enabling the system to make gradual adjustments rather than abrupt changes. The membership functions are defined as follows:

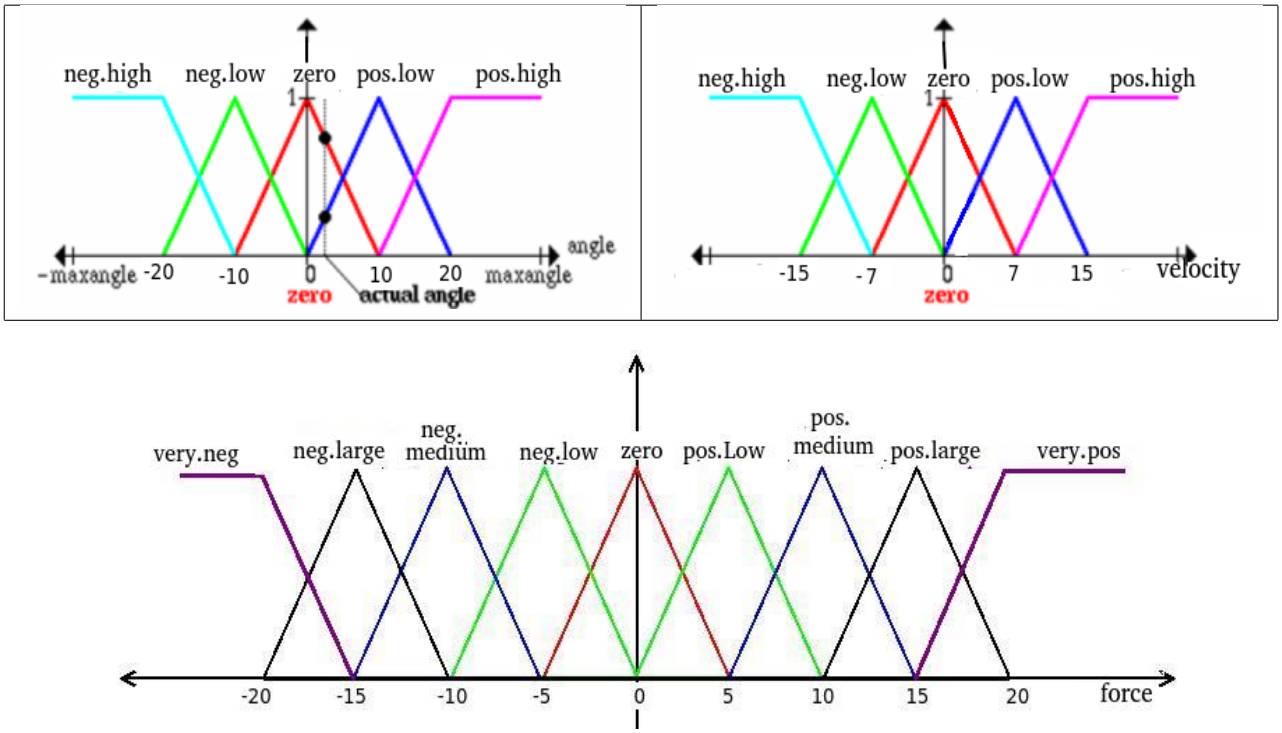

Figure 5.2: Membership Functions

5.1.5 Fuzzy Rules

For the fuzzy control system, a set of rules defines how the inputs (angle θ and angular velocity w) influence the output (force F). These rules follow an "IF-THEN" structure, where different conditions on the inputs determine the necessary corrective force to stabilize the pendulum.

Example Rules

- If the angle is zero and the velocity is zero, then the force is zero.

- If the angle is zero and the velocity is negative low, then the force is positive low.

The complete set of fuzzy rules is represented in the table 5.1 as follows :

	neg.high	neg.low	zero	pos.low	pos.high
neg.high	very.pos	pos.large	pos.medium	pos.low	zero
neg.low	pos.large	pos.medium	pos.low	zero	neg.low
zero	pos.medium	pos.low	zero	neg.low	neg.medium
pos.low	pos.low	zero	neg.low	neg.medium	neg.large
pos.high	zero	neg.low	neg.medium	neg.large	very.neg

Table 5.1: Fuzzy rules of the inverted pendulum control system

5.2 Steps to Follow

1. Implement a fuzzy controller using the Fuzzy Logic Toolbox in MATLAB.
2. Utilize MATLAB's predefined functions.
3. Implement a fuzzy controller with Python

5.3 Useful Functions

Here are some useful functions for the lab work:

- fz = newfis('pendul'); → Creates a fuzzy system named pendul.
- fz = addvar(fz, 'input', 'angle', [-25 25]); → Adds a linguistic variable.
- fz = addmf(fz, 'input', 1, 'neg.high', 'trapmf', [-30 -25 -20 -10]); → Adds a fuzzy set (membership function) to the first input variable.

5.3.1 Writing Fuzzy Rules in Matrix Form

```
ruleList = [1 1 9 1 1
1 2 8 1 1
1 3 7 1 1
...
...];
```

- The first column indicates the membership function number of the first variable.
- The second column represents the membership function number of the second variable.
- The third column corresponds to the membership function number of the output variable.

- The fourth column specifies the rule weight (between 0 and 1).
- The fifth column; (1) for AND (conjunction) between premise variables, and (2) for OR (disjunction) between premise variables.

5.3.2 Integrate the rules system

To integrate the rules into our fuzzy system, we use the function addrule as follow:

```
fz = addrule(fz, ruleList); % Integrate the rules into our fuzzy
    system
```

5.3.3 Other functions

- showrule(fz); view the list of rules.
- x=evalfis([15 -10], fz); test the system.
- plotmf(fz,'input',1); plot the membership functions.
- showfis(fz); display system characteristics.
- writefis(fz,'flou1'); save the system.
- ff=readfis('flou1'); load the file.
- surfview('flou1'); view the surface.
- surfview(fz); view the surface.
- ruleview('flou1'); view the rules graphically .
- ruleview(fz); view the rules.

5.4 Solution 1: using the Fuzzy Logic Toolbox of MATLAB

We start by going to the APP menu in MATLAB, then click on the arrow on the right to view all the applications and select 'Fuzzy Logic Designer'. A window called 'FIS Editor' will open, as shown in Figure 5.3

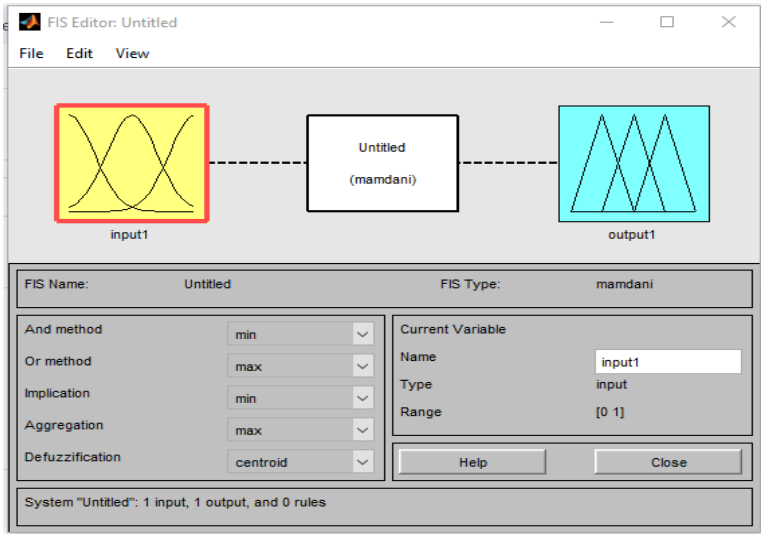

Figure 5.3: Fuzzy Logic Designer

Click on the input 1 curve, then change its name in the 'Name' field on the left, and rename the first input to 'angle', as shown in Figure 5.4.

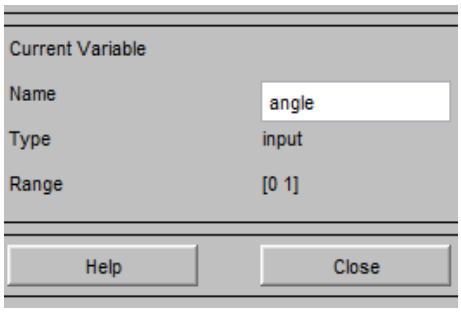

Figure 5.4: The 'Name' field of the fuzzy variable

Then, double-click on the 'angle' curve to open a new window that allows you to edit the membership functions for the variable 'angle', as shown in Figure 5.5

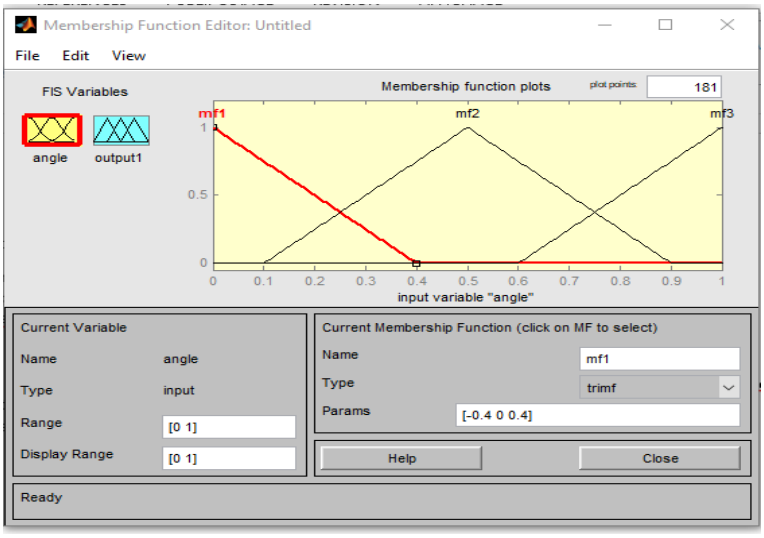

Figure 5.5: The membership functions editor

Go to the bottom-left panel and modify the range of the first input to [-35 35]. Then click on the first curve 'mf1' and go to the 'Name' field to rename it 'neg.high'. Change the 'Type' field to 'trapmf' since the first membership function is a trapezoidal function. Then, go to the 'Parameters' field and define the trapezoidal function using the vector [-35 -30 -20 -10]. The result is shown in Figure 5.6.

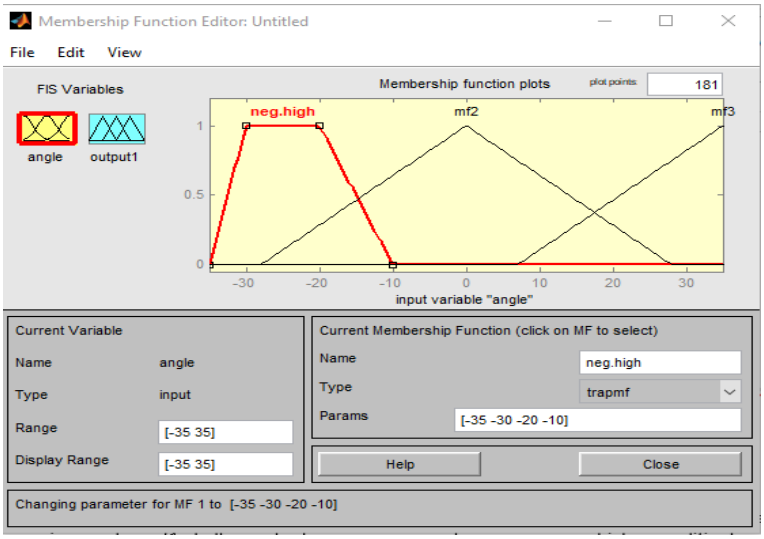

Figure 5.6: The definition of the first membership functions for the first input variable

Then do the same for the two membership functions of type 'trimf' (triangular). However, in our lab session, we need five membership functions. So, go to the 'Edit' menu in the same window and choose 'Add MFs', then select 2 since we already have 3. The result is shown in Figure 5.7.

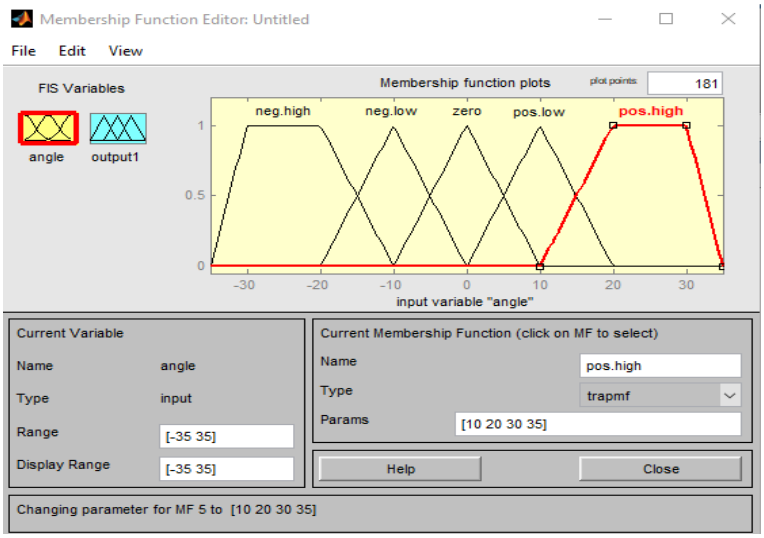

Figure 5.7: The membership functions of the variable 'angle'

Return to the main window of the Fuzzy Logic Designer (Figure 5.3), click on the 'Edit' menu, then choose 'Add Variable' $\Longrightarrow$ 'Input'. Next, click on the input2 curve and change its name to 'velocity' in the 'Name' field. The result is shown in Figure 5.8.

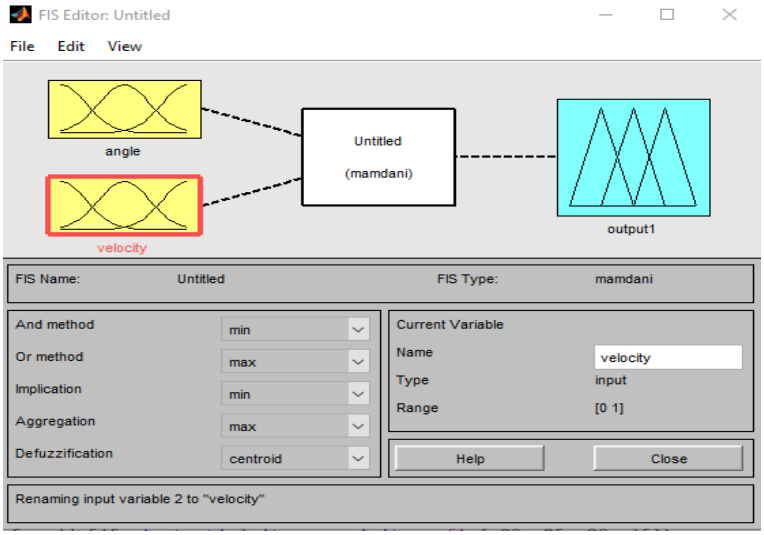

Figure 5.8: Creating the second membership function

Double-click on the 'velocity' curve to open a new membership function editor window. Click on the 'velocity' curve, define the range of the variable, then set the membership functions in the same way as we did for 'angle'. The result is shown in Figure 5.9.

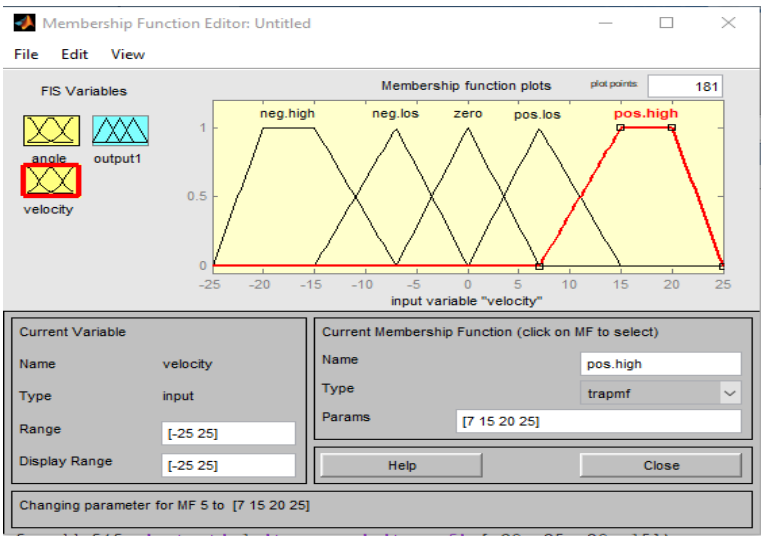

Figure 5.9: The membership functions of the variable 'velocity'

Select the output variable, and in the same way, define its membership functions. The result is shown in Figure 5.10.

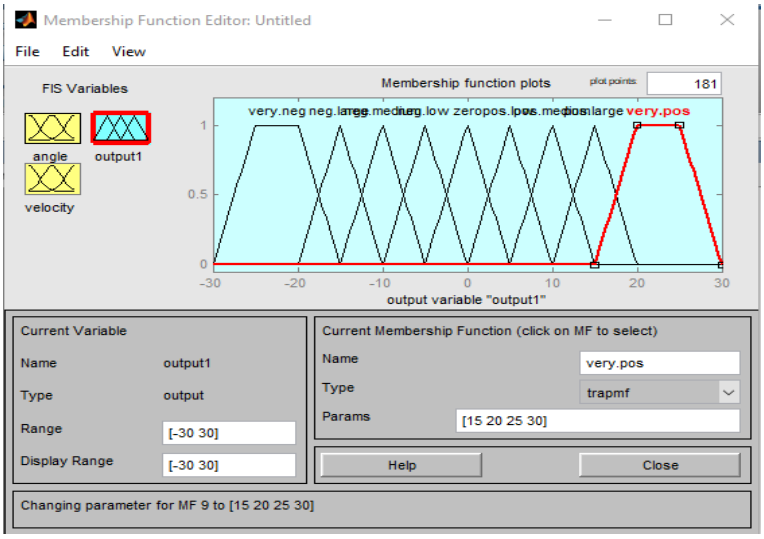

Figure 5.10: The membership functions of the output variable 'force'

Click on 'Edit' and choose 'Rules' to define the rules of our fuzzy system. In the rule editor window, the membership functions are already available — you just need to select and add them to write our 25 rules. The result is shown in Figure 5.11.

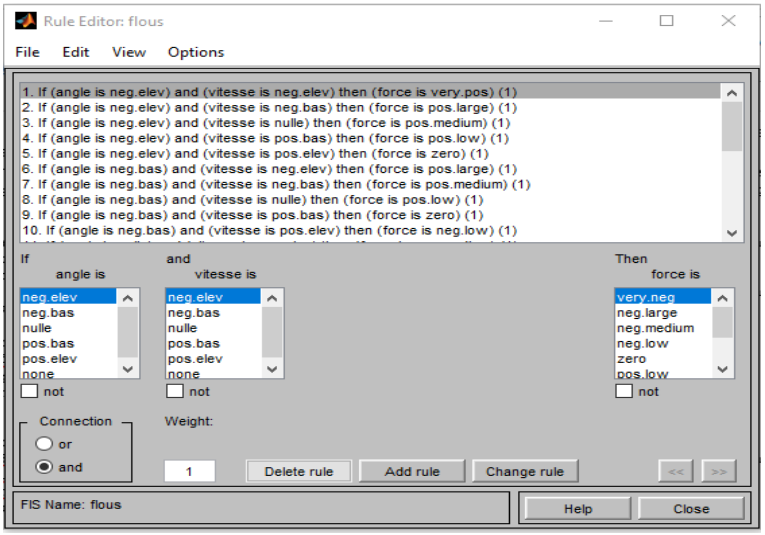

Figure 5.11: The rules edited

If we choose the 'View' menu and select 'Surface', a window will appear displaying a 3D surface. This surface shows how the output of our fuzzy system varies based on all possible input combinations, as shown in Figure 5.12.

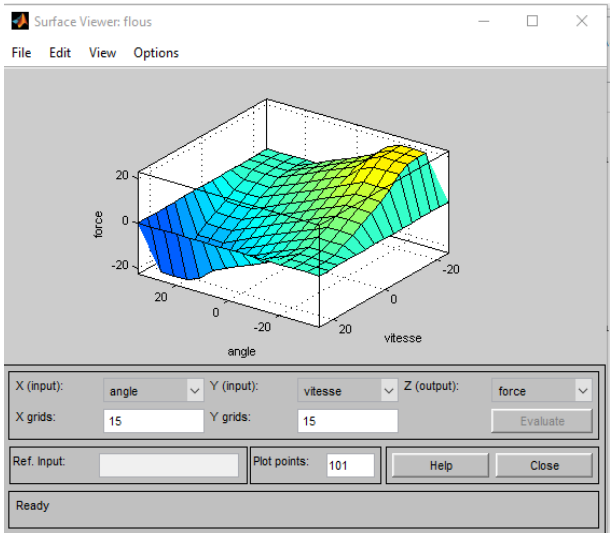

Figure 5.12: The 3D plot of the surface

If we choose the 'View' menu and select 'Rules', a window will appear showing the results and the activated curves each time the system inputs are changed, as shown in Figure 5.13.

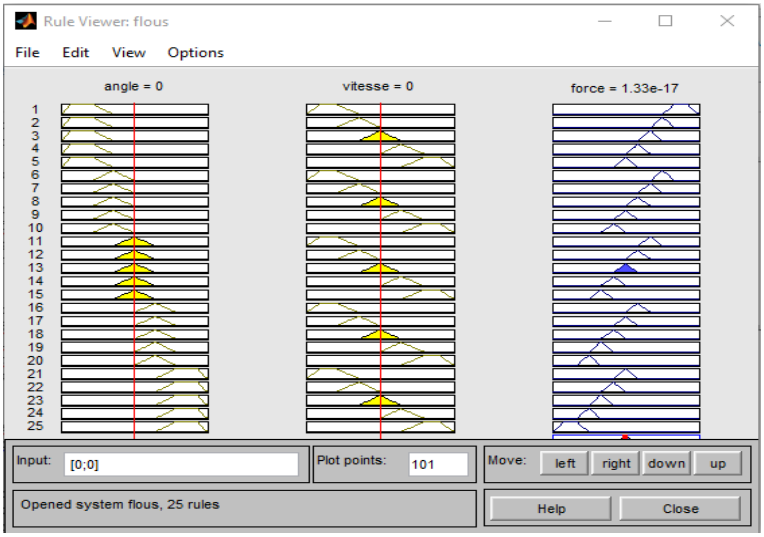

Figure 5.13: The rules activation

5.5 Solution 2: using MATLAB's predefined functions

For this second approach, without using the MATLAB toolbox, open a new script, define a new function 'fuzzy', and save it. We create a new fuzzy system using the newfis function and store it in a variable called fz. Then, we add the first input variable called 'angle' with its range using the addvar function, as follows :

```
function Fuzzy()
fz=newfis('pendulum');
fz=addvar(fz,'input','angle',[-35 35]);
```

Then, we define the five membership functions by specifying for each one: the variable type ('input'), the variable number, the name of the membership function, its type (trapezoidal, triangular, etc.), and its range, as follows:

```
fz=addmf(fz,'input',1,'neg.high','trapmf',[-35 -30 -20 -10]);
fz=addmf(fz,'input',1,'neg.low','trimf',[ -20 -10 0]);
fz=addmf(fz,'input',1,'zero','trimf',[ -10 0 10]);
fz=addmf(fz,'input',1,'pos.low','trimf',[0  10  20]);
fz=addmf(fz,'input',1,'pos.high','trapmf',[10 20 30 35]);
```

Repeat the same process for the second input variable, 'velocity', and for the first output variable, 'force'. For each of them, use the addvar function to define the variable type ('input' or 'output'), the variable name, and its range. Then, define the membership functions by specifying the appropriate parameters, as shown below:

```
fz=addvar(fz,'input','vitesse',[-25 25]);
fz=addmf(fz,'input',2,'neg.high','trapmf',[-25 -20 -15 -7]);
fz=addmf(fz,'input',2,'neg.low','trimf',[ -15 -7 0]);
fz=addmf(fz,'input',2,'zero','trimf',[ -7 0 7]);
fz=addmf(fz,'input',2,'pos.low','trimf',[0  7  15]);
fz=addmf(fz,'input',2,'pos.high','trapmf',[7 15 20 25]);

fz=addvar(fz,'output','force',[-30 30]);
fz=addmf(fz,'output',1,'very.neg','trapmf',[-30 -25 -20 -15]);
fz=addmf(fz,'output',1,'neg.large','trimf',[ -20 -15 -10]);
fz=addmf(fz,'output',1,'neg.medium','trimf',[ -15 -10 -5]);
fz=addmf(fz,'output',1,'neg.low','trimf',[ -10 -5 0]);
fz=addmf(fz,'output',1,'zero','trimf',[ -5 0 5]);
fz=addmf(fz,'output',1,'pos.low','trimf',[ 0 5 10]);
fz=addmf(fz,'output',1,'pos.medium','trimf',[ 5 10 15]);
fz=addmf(fz,'output',1,'pos.large','trimf',[10  15  20]);
fz=addmf(fz,'output',1,'very.pos','trapmf',[15 20 25 30]);
```

To display the membership functions, we use the plotmf function as follows:

```
igure()
plotmf(fz,'input',1);
figure()
plotmf(fz,'input',2);
figure()
plotmf(fz,'output',1);
```

The corresponding result is shown in Figure 5.14.

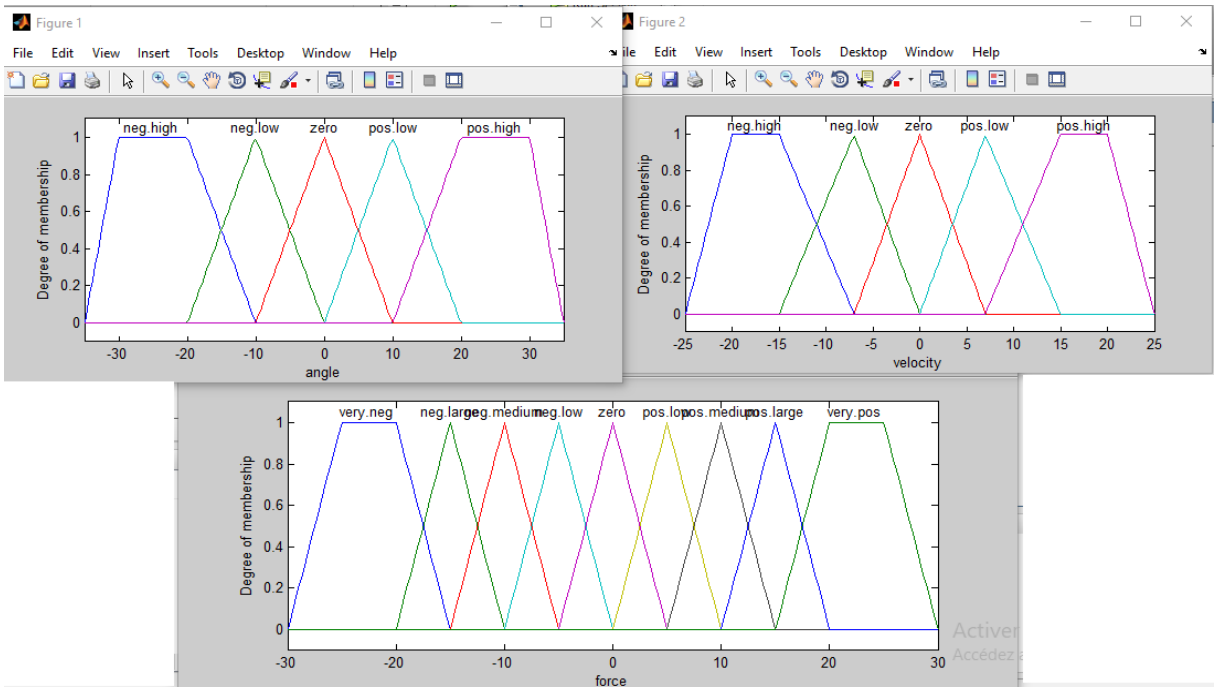

Figure 5.14: The plot of the membership functions of the three variables.

To write the fuzzy rules, the order in which the fuzzy variables and their fuzzy sets are defined is crucial. The 25 rules are represented as a matrix with 25 rows and 5 columns. For each row ; the first element indicates the index of the membership function for the first input variable, the second element indicates the index for the second input variable, the third element corresponds to the index of the output variable's membership function, the fourth element represents the weight of the rule, all the rules have the same weight, equal to 1. the fifth element is fixed to 1, since all rules use an "AND" operator. This matrix is then integrated into the fuzzy system using the addrule function as follows:

```
ruleList=[1 1 9 1 1
1 2 8 1 1
1 3 7 1 1
1 4 6 1 1
1 5 5 1 1
2 1 8 1 1
```

```
2 2 7 1 1
2 3 6 1 1
2 4 5 1 1
2 5 4 1 1
3 1 7 1 1
3 2 6 1 1
3 3 5 1 1
3 4 4 1 1
3 5 3 1 1
4 1 6 1 1
4 2 5 1 1
4 3 4 1 1
4 4 3 1 1
4 5 2 1 1
5 1 5 1 1
5 2 4 1 1
5 3 3 1 1
5 4 2 1 1
5 5 1 1 1];
fz=addrule(fz,ruleList);
```

The functions ruleview and surfview allow visualization of the rules and the surface, similar to the MATLAB toolbox, as shown in figures 5.12 and 5.13. The writefis function enables saving the fuzzy system for later use, as follows:

```
ruleview(fz)
surfview(fz)
writefis(fz, 'flous');
end
```

Subsequently, we can load our fuzzy system into another function or in the command window using the readfis function, to use or evaluate it on new input data with the evalfis function, as follows:

```
>> f=readfis('flous');
>> x=evalfis([15 -10],f)

x =

-0.4086

>>
```

5.6 Solution 3: using the library scikit-fuzzy with Python

Start by installing the libraries scikit-fuzzy and networkx the same way we did in the previous lab for scikit-learn.

5.6.1 Import the libraries and define the fuzzy variables

We begin by creating a new script, which we'll call flou.py, and then import the libraries numpy, matplotlib.pyplot, as well as skfuzzy and skfuzzy.control.

Next, we create the three fuzzy variables for the inverted pendulum problem. The input variables (or antecedents) are angle and velocity, and the output variable (or consequent) is force. We define the range (or universe) of each variable based on the curves given in the problem statement, as shown in the following code:

```
import numpy as np
import matplotlib.pyplot as plt
import skfuzzy as fz
from skfuzzy import control as ctrl
angle = ctrl.Antecedent(np.arange(-35, 36, 1), 'angle')
velocity = ctrl.Antecedent(np.arange(-25,26,1), 'velocity')
force = ctrl.Consequent(np.arange(-30,31,1), 'force')  # Newton
```

5.6.2 Creation of Membership Functions (Fuzzy Sets)

Based on the curves of each variable, we can determine the fuzzy subsets. For example, for the variable angle, we have five fuzzy sets: neg.high, neg.low, zero, pos.low, and pos.high.

The first and last sets have a trapezoidal shape, defined by four points, while the other three have a triangular shape, defined by three points. The definition of these intervals is as follows:

```
angle['negHi'] = fz.trapmf(angle.universe, [-35, -30, -20,
   -10])
angle['negLo']= fz.trimf(angle.universe, [-20, -10, 0])
angle['Zero'] = fz.trimf(angle.universe, [-10, 0, 10])
angle['posLo'] = fz.trimf(angle.universe, [0, 10, 20])
angle['posHi'] = fz.trapmf(angle.universe, [10, 20, 30, 35])
```

The same applies to the second variable (velocity).

```
velocity['negHi'] = fz.trapmf(velocity.universe,
   [-25,-20,-15,-7])
velocity['negLo'] = fz.trimf(velocity.universe, [-15, -7, 0])
velocity['Zero'] = fz.trimf(velocity.universe, [-7, 0, 7])
velocity['posLo'] = fz.trimf(velocity.universe, [0, 7, 15])
velocity['posHi'] = fz.trapmf(velocity.universe, [7,15,20,25])
```

And for the variable force, which contains nine fuzzy sets, the coordinates are taken from the force curves in the problem statement.

```
force['Vneg'] = fz.trapmf(force.universe, [-30, -25, -20, -15])
force['negLa'] = fz.trimf(force.universe, [-20, -15, -10])
force['negMe'] = fz.trimf(force.universe, [-15, -10, -5])
force['negLo'] = fz.trimf(force.universe, [-10, -5, 0])
force['Zero'] = fz.trimf(force.universe, [-5, 0, 5])
force['posLo'] = fz.trimf(force.universe, [0, 5, 10])
force['posMe']= fz.trimf(force.universe, [5, 10, 15])
force['posLa'] = fz.trimf(force.universe, [10, 15, 20])
force['Vpos'] = fz.trapmf(force.universe, [15, 20, 25, 30])
```

To ensure that the membership functions are correctly defined, we can visualize them using the view method as follows:

```
angle.view()
velocity.view()
force.view()
plt.show()
```

This will display the membership functions as defined in the lab assignment, as shown in Figure 5.15.

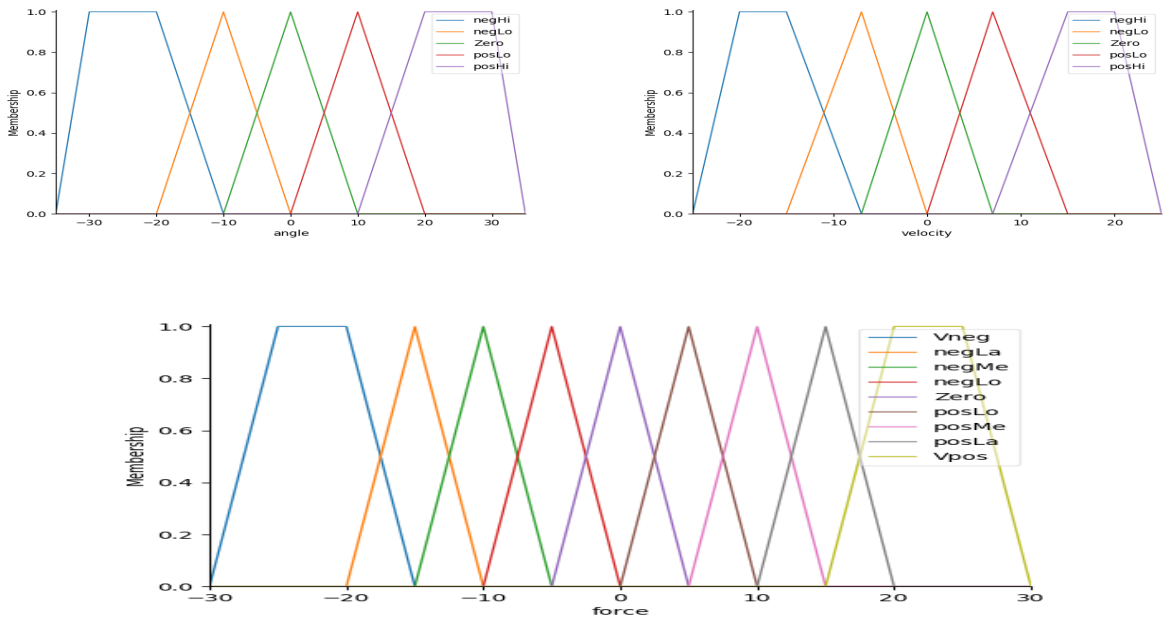

Figure 5.15: Membership Functions viewed by view method

5.6.3 Definition of fuzzy rules

There are 25 fuzzy rules, provided in Table 5.1. For example, the first rule is:
"If angle is neg.high and velocity is neg.high, then force should be very.pos."
To write the fuzzy rules, we use the control module from skfuzzy. The 25 rules are then written as follows:

```
r1=ctrl.Rule(angle['negHi'] & velocity['negHi'],force['Vpos'])
r2=ctrl.Rule(angle['negHi'] & velocity['negLo'],force['posLa'])
r3=ctrl.Rule(angle['negHi'] & velocity['Zero'],force['posMe'])
r4=ctrl.Rule(angle['negHi'] & velocity['posLo'],force['posLo'])
r5=ctrl.Rule(angle['negHi'] & velocity['posHi'],force['Zero'])
r6=ctrl.Rule(angle['negLo'] & velocity['negHi'],force['posLa'])
r7=ctrl.Rule(angle['negLo'] & velocity['negLo'],force['posMe'])
r8=ctrl.Rule(angle['negLo'] & velocity['Zero'],force['posLo'])
r9=ctrl.Rule(angle['negLo'] & velocity['posLo'],force['Zero'])
r10=ctrl.Rule(angle['negLo']& velocity['posHi'],force['negLo'])
r11=ctrl.Rule(angle['Zero'] & velocity['negHi'],force['posMe'])
r12=ctrl.Rule(angle['Zero'] & velocity['negLo'],force['posLo'])
r13=ctrl.Rule(angle['Zero'] & velocity['Zero'],force['Zero'])
r14=ctrl.Rule(angle['Zero'] & velocity['posLo'],force['negLo'])
r15=ctrl.Rule(angle['Zero'] & velocity['posHi'],force['negMe'])
r16=ctrl.Rule(angle['posLo']& velocity['negHi'],force['posLo'])
r17=ctrl.Rule(angle['posLo']& velocity['negLo'],force['Zero'])
r18=ctrl.Rule(angle['posLo']& velocity['Zero'],force['negLo'])
r19=ctrl.Rule(angle['posLo']& velocity['posLo'],force['negMe'])
r20=ctrl.Rule(angle['posLo']& velocity['posHi'],force['negLa'])
r21=ctrl.Rule(angle['posHi']& velocity['negHi'],force['Zero'])
r22=ctrl.Rule(angle['posHi']& velocity['negLo'],force['negLo'])
r23=ctrl.Rule(angle['posHi'] & velocity['Zero'],force['negMe'])
r24=ctrl.Rule(angle['posHi']& velocity['posLo'],force['posLa'])
r25=ctrl.Rule(angle['posHi'] & velocity['posHi'],force['Vneg'])
```

5.6.4 Fuzzy inference system

The control.ControlSystem module from skfuzzy is used to build the fuzzy inference system by taking the set of fuzzy rules as input and returning the fuzzy system as output. This module contains the system's inputs and outputs and combines the fuzzy rules using the implication **minimum**, aggregation **maximum**, and **Mamdani**-type inference mechanism.

The default defuzzification method is centroid, but it can be changed to one of the following:

- 'centroid': center of gravity (default)

- 'bisector': splits the area into two equal parts
- 'mom': mean of maxima
- 'som': smallest of maxima
- 'lom': largest of maxima

The following command creates the fuzzy inference system, which we name force_sys, and which is not yet executable:

```
force_sys = ctrl.ControlSystem([r1, r2, r3,r4, r5, r6,r7, r8,
    r9,r10,r11,r12,r13,r14,r15,r16,r17,r18,r19,r20,r21,r22,r23,
    r24,r25])
```

To use the fuzzy system, we need to build the simulator using the control.ControlSystemSimulation module, which takes the fuzzy system as input and returns the simulator, which we named force_sim.

To execute our fuzzy system, we assign values to the angle and velocity inputs using the .input attribute, then compute the result using the .compute() method. The output can then be viewed via the .output attribute, as shown below:

```
force_sim = ctrl.ControlSystemSimulation(force_sys)
force_sim.input['angle'] = -10
force_sim.input['velocity'] = 2  # as example
force_sim.compute()
print(f"Force must be : {force_sim.output['force']:.2f} N")
```

As result we have:

```
Force must be : 3.38 N
```

We can visualize the rule execution using the view method on the simulation, as shown below:

```
force.view(sim=force_sim)
plt.show()
```

The result displayed is shown in figure 5.16.

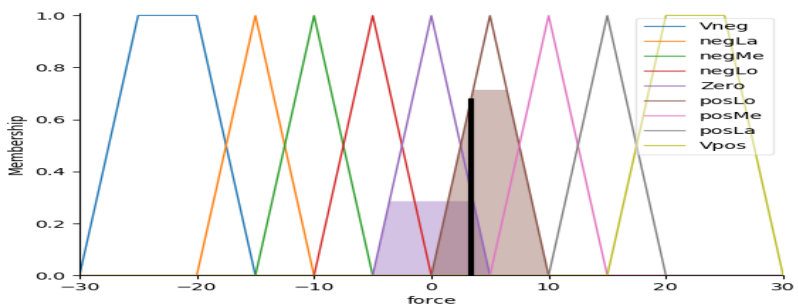

Figure 5.16: the force calculated for angle=-10 and velocity = 2

5.6.5 Saving and Reusing a Fuzzy System

Using the pickle library is the simplest method to save and reload a complete Python object, including a ControlSystem object and a ControlSystemSimulation object. So we open a binary file in write mode, which we call, for example, 'myFuzzy', and save the fuzzy system in it as follows:

```
import pickle
fuzz1=open("myFuzzy.pkl", "wb")
pickle.dump(force_sys, fuzz1)
```

To use our fuzzy system in another script, we need to load it and then build its simulator as follows:

```
import pickle
import skfuzzy as fz
from skfuzzy import control as ctrl

fuzz= open("myFuzzy.pkl", "rb")
system = pickle.load(fuzz)
sim = ctrl.ControlSystemSimulation(system)
```