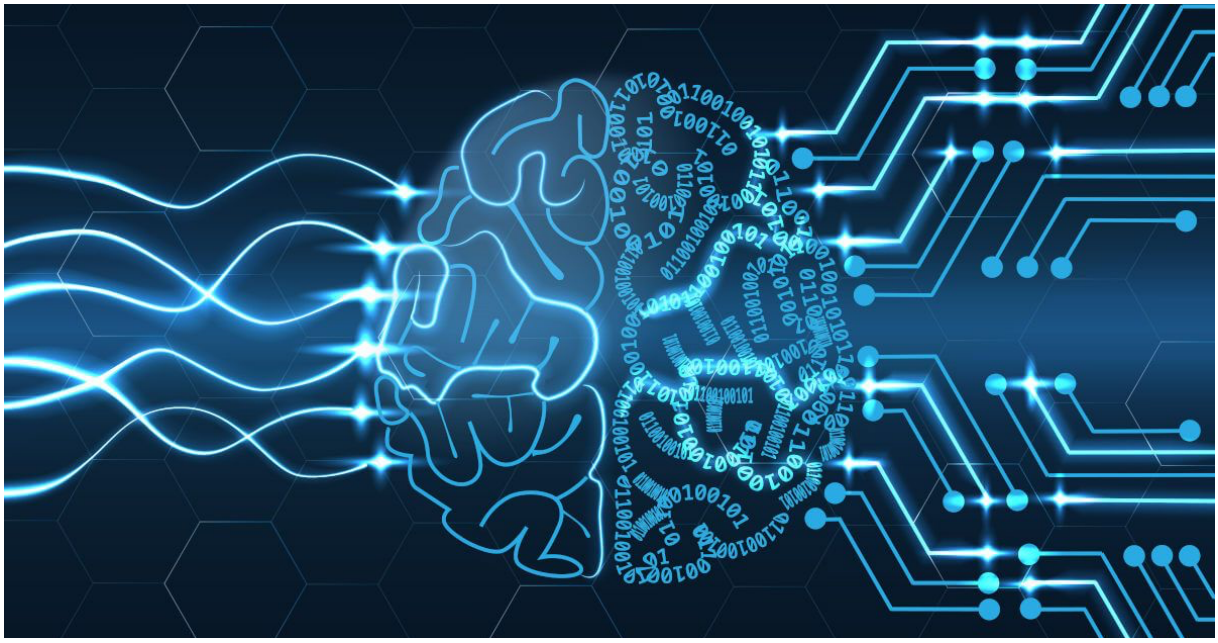
ISSASSIS
Grégoire
254715

BRNO UNIVERSITY OF TECHNOLOGY

MPA-MLF

Machine Learning Project

Classification of wireless transmitters



Student: Grégoire ISSASSIS (254715)

Study program: **Electronics and Communication Technologies**

Year of study: 2022 - 2023

Title of the module: Machine Learning (MPA-MLF 22/23L)

Roman MARŠÁLEK – Jan BOLCEK

Summary

Content table

Introduction

As an exchange student, I came to work at the BRNO University of Technology. Among the courses taught, a technical course, called MPA-MLF, is about Machine Learning (ML). In addition to the theoretical courses, we were able to do a lot of LABs to put into practice what we saw during the lectures. Thus, we could see what were the notions of K-means, Support Vector Machine (SVM), Principal Component Analysis (PCA), or Neural Networks (NN).

Now it is time for a project, which will review and assimilate everything we have seen during the lectures and applications in the different LABs since the beginning of the semester. The task will be to classify wireless transmitters based on measurement data. For this purpose, we have data collected in different CSV files. We will have to perform different tasks on these files in order to create a complete ML model from scratch.

These tasks are splitted this way. Firstly, we will examine the data, find which file contains our main dataset, see what the characteristics of the data inside this file are and understand how to work on this file. Then, we will have to preprocess these data so that it can be used in the creation of a ML model. The next step will be to build and train the ML model. Once the model is created, we will try to make it perform better, doing performance tuning, playing with the parameters of the functions inside this model. Finally, we will evaluate this model and create a CSV file from the model prediction. This file will be uploaded into a KAGGLE repository in order to compete with other students and see how the model of each student performed in comparison to the others.

One of the other main goals of this project is to compare different method of ML modeling, such as Multi-Layer Perceptron (MLP) and SVM.

Environment settings

First of all, I have imported into GOOGLE COLLABORATORY all the required libraries, in order to perform the project. I began to import all the libraries and modules we used during all the LABs from the beginning of the semester. Then, at the end of the project, I removed all the libraries and modules I have not used during the project advancement.

Here are all the libraries and modules I have used during the project:

```
from keras import layers
from keras.layers import Input, Dense, Activation, LSTM, RepeatVector, TimeDistributed
from keras.models import Sequential
from keras.utils import to_categorical

import matplotlib.image as mpimg
import matplotlib.pyplot as plt

import numpy as np
from numpy import quantile, where, random

import os

import pandas as pd

from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.svm import OneClassSVM
from sklearn.svm import SVC

import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import RMSprop, SGD
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

***Figure 1**: Libraries and modules used for the project*

Then, I have linked my GOOGLE DRIVE account to GOOGLE COLLABORATORY. Indeed, I have put all the needed files for the project into a dedicated folder in my GOOGLE DRIVE account. When linking my account to GOOGLE COLLABORATORY, I can then import the files into my code.

Here is the code to link both tools together:

```
[ ]  # Drive mounting in Google Colab
     from google.colab import drive
     drive.mount("/content/gdrive")

     Mounted at /content/gdrive
```

***Figure 2**: GOOGLE DRIVE and GOOGLE COLLABORATORY accounts linking*

Data examination

Once the code environment has been set up, I have imported the dataset into GOOGLE COLLABORATORY, and checked some basic information about this dataset in order to know it better. After an examination of the CSV files in MICROSOFT EXCEL, it seemed that x_train was the main file to use for the project. The basic information we will check are **dataset length**, **number of null or NaN values** into each column, dataset **feature names**, **data from the 5 first and last rows**, etc.

Here is the code I have used to perform this task:

```
[ ] # Define the path to the dataset
    path_to_dataset = '/content/gdrive/My Drive/ML - Projet/x_train.csv'

    # Load the dataset
    dataset = pd.read_csv(path_to_dataset)

    # Get the dataset length
    print('\nDataset length (rows, columns) :')
    print(dataset.shape)

    # Check the number of null values for each column
    print('\nNumber of null values :')
    print(dataset.isnull().sum())

    # Get dataset feature names
    print('\nDataset feature names :')
    print(dataset.columns)

    # Get data from the 5 first rows
    print('\nData from 5 first rows :')
    print(dataset.head(5))

    # Get data from the 5 last rows
    print('\nData from 5 last rows :')
    print(dataset.tail(5))

    # Get the whole data
    print('\nWhole dataset :')
    print(dataset)
```

***Figure 3**: Dataset examination code*

```
Dataset length (rows, columns) :
(15360, 12)

Number of null values :
Unnamed: 0    0
cfo_demod     0
gain_imb      0
iq_imb        0
or_off        0
quadr_err     0
m_power       0
ph_err        0
mag_err       0
evm           0
Tosc          0
Tmix          0
dtype: int64

Dataset feature names :
Index(['Unnamed: 0', 'cfo_demod', 'gain_imb', 'iq_imb', 'or_off', 'quadr_err',
       'm_power', 'ph_err', 'mag_err', 'evm', 'Tosc', 'Tmix'],
      dtype='object')

Data from 5 first rows :
   Unnamed: 0   cfo_demod  gain_imb     iq_imb     or_off  quadr_err  \
0           0  592.234802  0.048079 -35.082729 -28.560846   1.993170
1           1 -103.302032 -0.019917 -29.946953 -35.798664  -3.642311
2           2 -582.331299  0.036379 -32.096672 -31.905628   2.835839
3           3 -630.611267  0.063928 -38.216297 -30.084171   1.346316
4           4 -415.526978 -0.055761 -29.180740 -36.601025  -3.963526

    m_power    ph_err   mag_err       evm  Tosc  Tmix
0 -0.499721  1.107926  1.507550  2.423943  39.9  47.6
1 -0.928193  1.236059  2.741568  3.458056  14.8  23.1
2 -1.272485  1.282163  2.140096  3.013522  42.5  48.6
3 -0.596438  1.154848  1.093465  2.254514  26.1  35.4
4  0.113055  1.498889  3.608737  4.286684  24.2  40.8

Data from 5 last rows :
       Unnamed: 0   cfo_demod  gain_imb     iq_imb     or_off  quadr_err  \
15355       15355 -476.575653  0.039874 -39.792461 -29.962997   1.143736
15356       15356 -345.645508  0.077925 -32.963398 -27.998444   2.524010
15357       15357 -613.989807  0.090771 -38.951530 -28.111988   1.145893
15358       15358 -997.769531  0.099840 -39.532818 -27.819826   1.014136
15359       15359  450.446838  0.068571 -34.959385 -28.346176   1.996514
```

***Figure 4**: Dataset examination results*

As we can see in the figures above:

- The dataset contains **15360 rows** and **12 columns**;
- **There is no null or NaN value in the dataset**. So, there is no need to remove or replace any value that can be corrupted in this file;
- The columns are:
  - **Unnamned: 0**. This column corresponds to the index of each row. It will not be needed during the project. So, it will be dropped from the dataset during the project;
  - **cfo_demod**: Carrier Frequency Offset (CFO) between transmitter and receiver, measured after demodulation, in Hz;
  - **gain_imb**: gain imbalance of modulator;
  - **iq_imb**: combination of gain and quadrature imbalances aggregated into one parameter;
  - **or_off**: origin offset, known as DC offset;
  - **quadr_err**: quadrature error imbalance;
  - **m_power**: measured signal power. This column is just for information and will be dropped from the dataset during the project;
  - **ph_error**: phase difference between received and ideal constellation points;
  - **mag_error**: magnitude error between received and ideal constellation points;
  - **evm**: Error Vector Magnitude (EVM), representing the Root Mean Square (RMS) error between received and ideal constellation points;
  - **Tosc**: temperature of the software-defined radio, measured at the local oscillator. This column is just for information and will be dropped from the dataset during the project;
  - **Tmix:** temperature of the software-defined radio, measured at the RF mixer. This column is just for information and will be dropped from the dataset during the project;

Data preprocessing

Before working on the dataset, I will have to modify it accordingly to the data examination made in the step before. Here is how I have proceeded to prepare the data for the project:

```
[ ]  # Import all the data
     x_test_original = pd.read_csv('/content/gdrive/My Drive/ML - Projet/x_test.csv')
     x_train_original = pd.read_csv('/content/gdrive/My Drive/ML - Projet/x_train.csv')
     y_train_original = pd.read_csv('/content/gdrive/My Drive/ML - Projet/y_train.csv')

     # Work on copies of the original files
     x_test = x_test_original.copy()
     x_train = x_train_original.copy()
     y_train = y_train_original.copy()
```

***Figure 5**: Data importing and copies creation*

First of all, we will import all the data required for the project. Then, in order to avoid any issue, I have made copies of all the files. We will work only on the copies of the files and not the original files during all the project.

```
y_train :
       Unnamed: 0  target
0               0       5
1               1       1
2               2       6
3               3       3
4               4       2
...           ...     ...
15355       15355       3
15356       15356       8
15357       15357       7
15358       15358       7
15359       15359       5

[15360 rows x 2 columns]
```

***Figure 6**: y_train file examination*

```
# Get only y_train useful data (column 2)
print('\ny_train :\n', y_train)
y_train_column2 = y_train['target']

# Classifying y_train values into 8 different classes (from 1 to 8 (max value))
#y_train_encoded = to_categorical(y_train_column2, num_classes=None)  # 9 different classes (from 0 to 8 (max value))
y_train_encoded = pd.get_dummies(y_train_column2)
print('\ny_train_encoded :\n', y_train_encoded)
```

***Figure 7**: y_train file preprocessing*

Then, I have examinated the y_train file to see how it is made and what type of data are composing it. From data examination in Figure 6, we can assume:

- There are two columns on this file, but only the second is relevant. Indeed, **the first column of the y_train file represents the rows indexes, which is not relevant to use for the project**. So, as we can see in Figure 7, I have created a variable, called **y_train_column2**, and containing only the data of the second column of y_train;
- **The y_train file target column values are between 1 and 8**. There is no value out of this interval. So, I have encoded the previously created variable (containing the target column of y_train) into another variable, called **y_train_encoded**. I have used the **pandas *get_dummies*** function rather than **keras *to_categorical*** because this function starts at the index 1 and not 0. As we have no 0 value, it is better to encode the values from 1 to 8 than from 0 to 8 or 0 to 7.

```
y_train_encoded :
        1  2  3  4  5  6  7  8
0       0  0  0  0  1  0  0  0
1       1  0  0  0  0  0  0  0
2       0  0  0  0  0  1  0  0
3       0  0  1  0  0  0  0  0
4       0  1  0  0  0  0  0  0
...    .. .. .. .. .. .. .. ..
15355   0  0  1  0  0  0  0  0
15356   0  0  0  0  0  0  0  1
15357   0  0  0  0  0  0  1  0
15358   0  0  0  0  0  0  1  0
15359   0  0  0  0  1  0  0  0

[15360 rows x 8 columns]
```

***Figure 8**: y_train_encoded results*

As we can see in the Figure 8, the encoded y_train target column gives a boolean representation of the data, only composed of a table with 0 and 1 numbers. If I would have used to_categorical rather than get_dummies, we would have 9 columns, with an irrelevant 0 column which would have never been used.

```
# Remove unwanted features (m_power, tosc, tmix)
x_test = x_test.drop(['Unnamed: 0','m_power', 'Tosc', 'Tmix'], axis=1)
x_train = x_train.drop(['Unnamed: 0','m_power', 'Tosc', 'Tmix'], axis=1)

# Check the features removing
print('\nx_test features :')
print(x_test.columns)
print('\nx_train features :')
print(x_train.columns)
```

***Figure 9**: Unwanted features dropping*

```
x_test features :
Index(['cfo_demod', 'gain_imb', 'iq_imb', 'or_off', 'quadr_err', 'ph_err',
       'mag_err', 'evm'],
      dtype='object')

x_train features :
Index(['cfo_demod', 'gain_imb', 'iq_imb', 'or_off', 'quadr_err', 'ph_err',
       'mag_err', 'evm'],
      dtype='object')
```

***Figure 10**: Features dropping checks*

As we have seen during the Data Examination part of the project, some features are not required. Indeed, the next step of the project has been to remove the "Unnamed: 0", "m_power", "Tosc" and "Tmix" columns on both x_test and x_train files. We can see the code I have used to do this above, in Figure 9. In Figure 10, we can see that after the code has been run, there are only 8 features remaining in the x_test and x_train files, instead of 12.

Finally, I have **classified x_train values into different classes**, the same way we have seen before, with y_train_encoded. To do this, I have created a new variable, called x_train_encoded, with the help of the **pandas *get_dummies*** function.

```
# Classifying x_train values into different classes
x_train_encoded = pd.get_dummies(x_train)
print('\nx_train_encoded :', x_train_encoded)
```

***Figure 11**: x_train values classification*

Model building

Now that all the files have been preprocessed, comes the time of model building. Here are the code lines I have written to build the model:

```
[ ] # Creating the model
    model = Sequential()

    # Add input layer
    model.add(Input(shape=(8,)))

    # Add hidden layers
    model.add(Dense(8, activation = 'relu'))
    model.add(Dense(4, activation = 'relu'))

    # Add output layer
    model.add(Dense(8, activation='softmax'))

    # Summarize the model
    model.summary()
```

***Figure 12**: Model building*

First of all, I have created the model, using the ***Sequential()*** function we have used during LABs. Then, the model is splitted in three different parts:

- The first one is the **input layer**. To create it, I have used the ***model.add(Input)*** function, with 8 neurons, as we have 8 possible values in y_train_encoded;
- The second part are the **hidden layers**, using the ***model.add(Dense)*** function. As we will see later, in the Performance tuning step of the project, the best solution seemed to have 2 hidden layers, containing respectively 8 and 4 neurons, and both using the relu activation function;
- The third and last part is the **output layer**. As for the hidden layers, I have used the ***model.add(Dense)*** function and the softmax activation function. As for the input layer, I have used 8 neurons.

Finally, I have printed a summary of the model, using the ***model.summary()*** function.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 8)                 72

 dense_1 (Dense)             (None, 4)                 36

 dense_2 (Dense)             (None, 8)                 40

=================================================================
Total params: 148
Trainable params: 148
Non-trainable params: 0
```

***Figure 13**: Model summary*

As we can see in the Figure above, the model summary allows us to better understand the output shape and the parameters of each layer. With the parameters I have set before, we can see that the model uses 72 parameters in the input layer, 36 in the first hidden layer and 40 in the second hidden layer. In total, there are 148 trainable parameters.

Model training

The next big step of the project is the model training, which can be decomposed into three main parts. These parts are **splitting the training data into train and validation**, **compiling the model and setting the loss function**, and finally, **training the model**.

Splitting the training data

Here is the code I have used to perform this task, and the results I have obtained:

```
[ ] # Split training data into train and validation
    # (here 20% of the data will be used for the testing and 80% for the checking)
    X_train, X_test, Y_train, Y_test = train_test_split(x_train_encoded, y_train_encoded, test_size=0.2)
    print('\nX_train shape :', X_train.shape)
    print('X_test shape :', X_test.shape, '\n')
```

***Figure 14**: Splitting the training data*

```
X_train shape : (12288, 8)
X_test shape : (3072, 8)
```

***Figure 15**: Results*

The main goal of the ***train_test_split()*** function, that we can see in Figure 14, is to **split arrays or matrices into random train and test subsets**. As we will see later, in the Performance tuning step of the project, the best test_size ratio seemed to be 0.2. **This number significates that 80% of the data will be used for the training and 20% of the data will be used for the testing**. We can see in Figure 15 that from the 15360 of the dataset, 12288 will be used for the training and 3072 for the testing. These numbers correspond respectively to 80% and 20% of 15360.

Compiling the model and setting the loss function

Here is the code I have used to perform this task, and the results I have obtained:

```
# Compile the model & Set the loss function
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

***Figure 16**: Compiling the model*

I have used the ***optimizers.SGD()*** function from **tensorflow keras** library, the same way we have used it during the LABs. In this function, I have set the ***learning_rate*** to 0.01 because, as we will see later in the Performance tuning step of the project, it is the number that gave me the best results when testing. Finally, I have used the ***model.compile()*** function to compile the model. As seen during the LABs, I have used **binary_crossentropy** as loss function in parameters.

Training the model

Here is the code I have used to perform this task, and the results I have obtained:

```
# Model training
history = model.fit(X_train, Y_train, epochs=200, batch_size=16, verbose=1)
```

***Figure 17**: Compiling the model*

```
Epoch 100/200
768/768 [==============================] - 2s 3ms/step - loss: 0.3262 - accuracy: 0.2508
Epoch 101/200
768/768 [==============================] - 3s 4ms/step - loss: 0.3260 - accuracy: 0.2507
Epoch 102/200
768/768 [==============================] - 2s 3ms/step - loss: 0.3259 - accuracy: 0.2509
Epoch 103/200
768/768 [==============================] - 2s 3ms/step - loss: 0.3257 - accuracy: 0.2510
Epoch 104/200
768/768 [==============================] - 2s 3ms/step - loss: 0.3256 - accuracy: 0.2509
Epoch 105/200
768/768 [==============================] - 2s 3ms/step - loss: 0.3254 - accuracy: 0.2509
```

***Figure 18**: Sample of the results*

As we can see in the Figure 17, I have used the ***model.fit()*** function in order to train the model. X_train and Y_train are given from the ***train_test_split()*** function used during the data splitting into training and testing categories. We will see later in the Performance tuning step of the project, why I have set ***epochs*** and ***batch_size*** parameters respectively to 200 and 16, who seemed to be the best values when testing. When set to 1, the verbose parameter is used to display the history of all events, epoch by epoch.

In the Figure 18, we can see a sample of the results, between the epochs 100 and 105. We can see that the ***model.fit()*** function **computes for each epoch the loss and the accuracy percentages**. Also, we can see that with a training ratio of 80% and a batch_size of 16, there are 768 samples per epoch. Finally, the function also displays the time spent by the machine for each epoch.

For example, the total dataset size is 15360 rows.
When we use 80% of the data for training, it remains 12288 samples.
With a batch_size of 16, we compute $\frac{number\_of\_training\_samples}{batch\_size} = \frac{12288}{16} = 768$.
This number of samples per epoch is a variable, changing in function of the given parameters in the ***model.fit()*** and ***train_test_split()*** functions.

Performance tuning

Once I was able to check that my model was working correctly, without errors or slowdowns, I could start optimizing its performance. I handled this task in the same way as we have made during LAB6, where we had to check the impact of each parameter within the functions used.

Here are the 6 feature parameters that I found useful to modify to improve the performance my ML model for this project:

- **Number of hidden layers**;
- **Number of neurons in each hidden layer**;
- **learning_rate** parameter in the ***tf.keras.optimizers.SGD()*** function;
- **number of epochs** in the ***model.fit()*** function;
- **batch_size** parameter in the ***model.fit()*** function;
- **test_size** parameter in the ***train_test_split()*** function;
- **activation function** in ***model.add()*** function during hidden layer creation;

Number of hidden layers

For testing this parameter, I have set the other parameters with fixed values who was 16 neurons in each hidden layer, softmax activation function, test_size=0.2, learning_rate=0.01, epochs=200 and batch_size=16. Then I have run the entire GOOGLE COLLABORATORY file with these parameters for the number of hidden layers:

- 1 hidden layer : Loss = 33.08% ; Accuracy = 25.21%
- **2 hidden layers: Loss = 37.27% ; Accuracy = 26.58%**
- 3 hidden layers: Loss = 37.68% ; Accuracy = 12.75%
- 5 hidden layers: Loss = 37.68% ; Accuracy = 12.87%
- 8 hidden layers: Loss = 37.68% ; Accuracy = 12.66%

As we can see above, even if the results seemed to be quite similar, the more I have added hidden layers, the lower the scores were. **Finally, the best score was obtained with 2 hidden layers**.

Number of neurons in each hidden layers

For testing this parameter, I have set the other parameters with fixed values who was 2 hidden layers, softmax activation function, test_size=0.2, learning_rate=0.01, epochs=200 and batch_size=16. Then I have run the entire GOOGLE COLLABORATORY file with these parameters for the number of neurons in each hidden layer (X neurons in 1st hidden layer, Y neurons in 2nd hidden layer):

- 4, 4: Loss = 32.06 % ; Accuracy = 25.74%
- 4, 8: Loss = 32.11 % ; Accuracy = 26.16%
- 4, 16: Loss = 32.17% ; Accuracy = 27.61%
- **8, 4: Loss = 26.61% ; Accuracy = 37.28%**
- 8, 8: Loss = 33.24% ; Accuracy = 25.09%
- 8, 16: Loss = 32.20% ; Accuracy = 26.48%
- **16, 4: Loss = 26.64% ; Accuracy = 37.63%**

- **16, 8: Loss = 26.60%% ; Accuracy = 37.00%**
- 16, 16: Loss = 32.54% ; Accuracy = 24.92%

Once again, the scores seemed to be quite similar between 8 and 4 neurons, 16 and 4 neurons, and finally, 16 and 8 neurons. However, I have chosen the 8 and 4 neurons configuration because the less neurons we have, the faster and easier to compute the algorithm is. **Finally, the best score was with 8 neurons on the 1st layer and 4 neurons in the 2nd layer**.

learning_rate

For testing this parameter, I have set the other parameters with fixed values who was 2 hidden layers with respectively 8 and 4 neurons, softmax activation function, test_size=0.2, epochs=100 (to optimize time spent on tests, this is why values are lower than for other tests) and batch_size=16. Then I have run the entire GOOGLE COLLABORATORY file with these parameters for the learning_rate:

- 0.001: Loss = 38.12% ; Accuracy = 13.80%
- **0.01: Loss = 32.66% ; Accuracy = 27.40%**
- 0.1: Loss = 31.55% ; Accuracy = 24.66%
- 0.5: Loss = 31.46% ; Accuracy = 24.76%
- 1.0: Loss = 31.48% ; Accuracy = 24.34%

Once again, results are quite similar. But the best score in terms of accuracy has been found with 0.01. However, it seems that the higher the learning_rate is, the lower the loss score is. As loss scores are very similar but Accuracy score is different, I have chosen the learning_rate giving the best Accuracy score. **Finally, the best score has been obtained with learning_rate=0.01**.

Number of epochs

For testing this parameter, I have set the other parameters with fixed values who was 2 hidden layers with respectively 8 and 4 neurons, softmax activation function, learning_rate= 0.01, test_size=0.2 and batch_size=16. Then I have run the entire GOOGLE COLLABORATORY file with these parameters for the number of epochs:

- 10: Loss = 38.20% ; Accuracy = 13.07%
- 20: Loss = 36.60% ; Accuracy = 24.99%
- 50: Loss = 36.95% ; Accuracy = 32.71%
- 100: Loss = 32.66% ; Accuracy = 27.40%
- **200: Loss = 26.61% ; Accuracy = 37.28%**
- 1000: Loss = 31.55% ; Accuracy = 25.18%
- 2000: Loss = 31.49% ; Accuracy = 25.11%

This parameter was the most complicated to test because results were very randoms. However, after multiple tests, it seemed that when increasing epochs number, Loss tends to 0% and Accuracy to 37.5%. Also, 200 epochs were the best score in terms of both Loss and Accuracy scores. **So, I have decided to set the number of epochs at 200 for the rest of the project**.

batch_size

For testing this parameter, I have set the other parameters with fixed values who was 2 hidden layers with respectively 8 and 4 neurons, softmax activation function, learning_rate= 0.01, epochs=100 (to optimize time spent on tests, this is why values are lower than for other tests) and test_size=0.2. Then I have run the entire GOOGLE COLLABORATORY file with these parameters for the batch_size:

- 8: Loss = 31.89% ; Accuracy = 25.38%
- **16: Loss = 32.66% ; Accuracy = 27.40%**
- 32: Loss = 36.01% ; Accuracy = 25.05%
- 64: Loss = 37.39% ; Accuracy = 24.79%
- 128: Loss = 37.62% ; Accuracy = 12.82%

It seemed that the lower the batch_size was, the lower the Loss score was. However, I found Accuracy scores very similar, and the best Accuracy score was with a batch_size set to 16. **Finally, I have decided to set the batch_size to 16 for the next of the project**.

test_size

For testing this parameter, I have set the other parameters with fixed values who was 2 hidden layers with respectively 8 and 4 neurons, softmax activation function, learning_rate= 0.01, epochs=100 (to optimize time spent on tests, this is why values are lower than for other tests) and batch_size=16. Then I have run the entire GOOGLE COLLABORATORY file with these parameters for the test_size:

- 0.1: Loss = 32.60% ; Accuracy = 25.05%
- 0.15: Loss = 33.09% ; Accuracy = 24.70%
- **0.2: Loss = 32.66% ; Accuracy = 27.40%**
- 0.5: Loss = 34.49% ; Accuracy = 24.57%
- 1.0: Loss = 32.41% ; Accuracy = 25.37%

Once again scores were very similar. **But I have found that the best score was obtained with a test_size set to 0.2**.

Activation function

For testing this parameter, I have set the other parameters with fixed values who was 2 hidden layers with respectively 8 and 4 neurons, learning_rate= 0.01, epochs=200, test_size=0.2 and batch_size=16. Then I have run the entire GOOGLE COLLABORATORY file with these parameters for the test_size:

- elu: Loss = 37.68% ; Accuracy = 12.75%
- selu: Loss = 30.27% ; Accuracy = 32.36%
- sigmoïd: Loss = 31.80% ; Accuracy = 24.88%
- softmax: Loss = 32.66% ; Accuracy = 27.40%
- softplus: Loss = 23.45% ; Accuracy = 65.65%
- softsign: Loss = 23.92% ; Accuracy = 49.36%
- **relu: Loss = 6.91% ; Accuracy = 93.37%**

Here, the results can be very different in function of the chosen activation function. **I have found that the best score was obtained with the relu activation function**. Also, I have found that the results were better if I let softmax as activation function for the output layer, and change just the activation function of the hidden layers

Summary of results

Here is a summary of the parameters that gave me the best scores and that I decided to use after this performance tuning:

- **2 hidden layers**:
  - **First one with 8 neurons**;
  - **Second one with 4 neurons**;
- **test_size = 0.2**;
- **learning_rate = 0.01**;
- **epochs = 200**;
- **batch_size = 16**;
- **activation function: relu for hidden layers and softmax for output layer**.

Schema of the final model

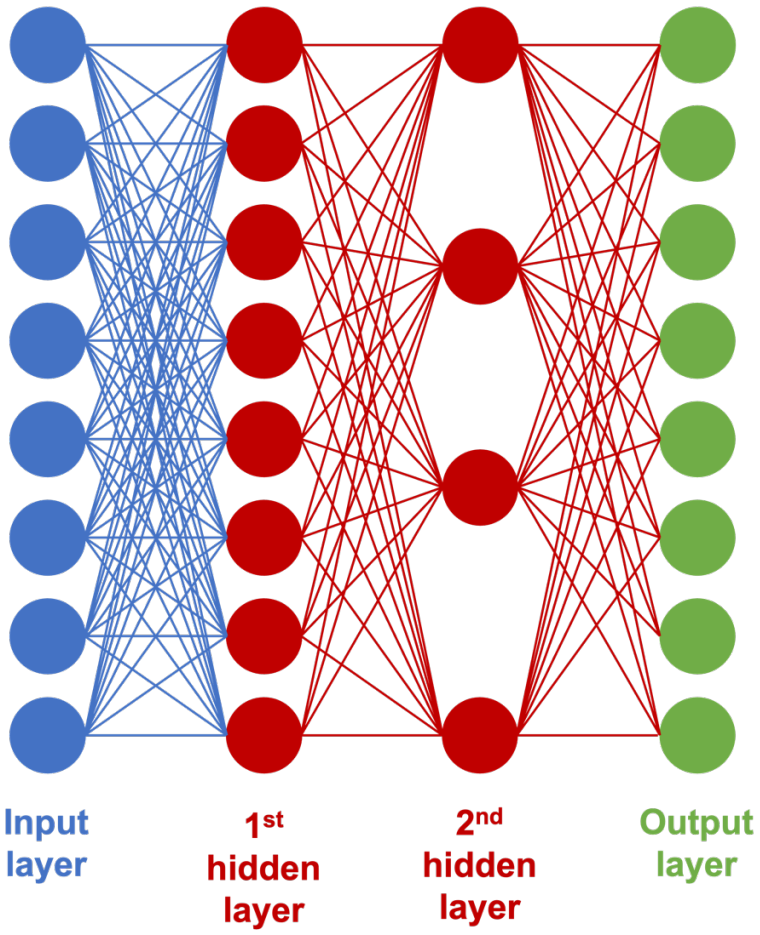


***Figure 19**: Final model characteristics*

Model evaluation

The model evaluation can be splitted in two main parts, computing the Loss and Accuracy score, and plotting the results.

Computing Loss and Accuracy

Here is the code I have used to perform this task, and the results I have obtained:

```
# Evaluate the model
loss, accuracy = model.evaluate(X_train, Y_train, verbose=0)

# Print final accuracy and loss
print('\nLoss: {:.2f}%'.format(loss*100))
print('Accuracy: {:.2f}%\n'.format(accuracy*100))
```

***Figure 20**: Loss and Accuracy computation code*

```
Loss: 31.92%
Accuracy: 25.29%
```

***Figure 21**: Results*

As we can see in the Figure 20, I have used the ***model.evaluate()*** function to compute the Loss and Accuracy scores. I have used, as before in the Model training step, X_train and Y_train values. Finally, I have set verbose to 0 in order to just display the last results in output and not the results for each epoch. As the score is computed between 0 and 1, we multiply them by a factor 100, in order to get a percentage. The Figure 21 displays the results of one of my tests. It changes every time I run the code, and each time I change a parameter, which is normal.

Plotting the results

Once we have calculated the Loss and Accuracy score, and reformatted them in percentages, we should display the results in two different plots. One plot represents the Loss values, and the other one represents the Accuracy values. Here is the code I have used to perform this task, and the results I have obtained:

```
# Plot the Loss graphic
plt.figure()
plt.plot(history.history['loss'])
plt.xlabel('N° of epochs')
plt.ylabel('Loss')

# Plot the Accuracy graphic
plt.figure()
plt.plot(history.history['accuracy'])
plt.xlabel('N° of epochs')
plt.ylabel('Accuracy')
```

***Figure 22**: Loss and Accuracy plots code*

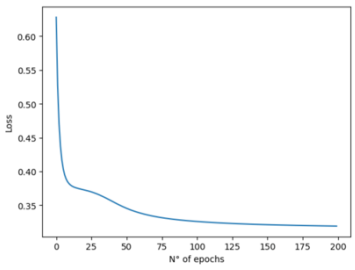


***Figure 23**: Loss plot*

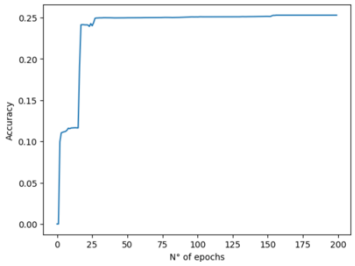


***Figure 24**: Accuracy plot*

As we can see in the Figure 23, when the epoch number increases, the Loss score decreases. After multiple tests, it seems that Loss score is tending to 0%. Inversely, the Accuracy score tends to 35.7%. The figures here are taken with some parameters that are not the optimal ones we have discussed during the Performance tuning step, this is why the accuracy score tends to 25% here.

Compare MLP to SVM

The next big step of the project was to create a SVM model. Once the SVM was created, I had to compare its results with previous results, during MLP model implementation. Here is how I have proceeded to make the SVM:

```
# Split training data into train and validation
# (here 20% of the data will be used for the training and 80% for the testing)
X_train, X_test, Y_train, Y_test = train_test_split(x_train_encoded, y_train_column2, test_size=0.2)

# Create a SVM model
SVMmodel=SVC(kernel='linear') # Linear kernel will search for separate lines --> Definition of the ke

# Feed the SVM
SVMmodel.fit(X_train, Y_train)

# Check what are the parameters
SVMmodel.get_params()
print('SVM parameters :', SVMmodel.get_params())

# Check how good it classifies
SVMmodel.score(X_test, Y_test)
print('\nSVM score :', SVMmodel.score(X_test, Y_test))
```

***Figure 25**: SVM implementation code*

```
SVM parameters : {'C': 1.0, 'break_ties': False,

SVM score : 0.9993489583333334
```

***Figure 26**: SVM results*

As we can see in the Figure 25, which is above, I have firstly reused the ***train_test_split()*** function. It was for two main reasons. The first reason is that I have used an encoded variable of y_train for the first implementation of this function. The second reason is that I would be sure that the data coming from this function was from scratch and not modified during previous code blocks. To be sure the results can be comparable between MLP and SVM, I have set the test_size to 0.2 again. Then I have created the model, using ***SVC()*** function, fitted the model, using ***SVModel.fit()*** function, as we have made during previous LABs. Finally, I have checked parameters, using the ***get_params()*** function and displayed the score using the ***score()*** function.

The Figure 26 shows the score obtained by this method. As we can see, with 0.9993, the score is really high, and better than the score obtained with MLP implementation (best at 0.9337).

Feature engineering

Here comes the final part of the project, which consists in creating a CSV file with all the y_test values and importing it in KAGGLE website. Firstly, I have predicted the model created in previous steps, with the x_test file that was given for the project, using the ***model.predict()*** function. As we have seen in Data preprocessing part, I have encoded the values on 8 different classes. So, the results obtained at the end are, for each row, a percentage for each of the classes. When we check the prediction and its shape, we see that the results are well a (3840,8) matrix, that is to say 8 columns of 3840 rows.

```
y_pred :
(3840, 8)
[[0.0709556  0.08265392 0.03230019 ... 0.02660263 0.00729199 0.05406735]
 [0.15909642 0.09986307 0.20318723 ... 0.17166075 0.05742444 0.09034553]
 [0.0709556  0.08265392 0.03230019 ... 0.02660263 0.00729199 0.05406735]
 ...
 [0.23285976 0.2315229  0.13684058 ... 0.12590042 0.03884295 0.1464555 ]
 [0.1291323  0.0695876  0.21220241 ... 0.18560086 0.06642041 0.06606736]
 [0.20181246 0.17205723 0.16165751 ... 0.07999001 0.01730726 0.23779428]]
```

***Figure 27**: Original prediction matrix before processing*

The goal is now to get, for each row, the index of the column with the highest value. It is important that the prediction be a (3840,1) matrix, that is to say a single column of 3840 rows. This is because KAGGLE will only accept CSV files of this size. To do this, I have used the **numpy *argmax()*** function. But it gave me a (1,3840) matrix, so I had to transpose it with the **reshape()** function. The last processing to do on the results is to add 1 to each value. Indeed, as I have retrieved the indexes from an array and indexing starts at 0 and not 1 in Python, we need to add 1 to each of the retrieved index values to get the right results. To do this, I have made a for loop with the length of y_test as number of iterations. For each iteration, we get y_test[i] value, which is the value at the index i, and we add 1.

Finally, the last part was to get all the values in a dataframe, called y_test_dataframe, with a column name which was 'target'. Then, I have created a CSV file, containing the dataframe previously created, and added an index column, called 'id'. At this moment, I had a complete code to perform the project. The very last part was to run the code, get the CSV fil and import it in KAGGLE to check the score.

Here is the code of this part of the project, and its final results:

```
# Predict the model
y_pred = model.predict(x_test)

# Display y_pred
print('\ny_pred :')
print(y_pred.shape)
print(y_pred)

# Get column index of highest value for each line
y_test = np.argmax(y_pred, axis=1)
y_test = y_test.reshape(3840, 1)

# Add 1 à chaque ligne (Python index begins at 0 and not 1)
for i in range(len(y_test)):
  y_test[i] += 1

# Display y_test
print('\ny_test :')
print(y_test.shape)
print(y_test)

# Create the dataframe
y_test_dataframe = pd.DataFrame(y_test, columns=['target'])

# Save the dataframe into a CSV file
y_test_dataframe.to_csv('y_test.csv', index_label='id')
```

***Figure 28**: Model prediction, processing and CSV creation*

```
y_test :
(3840, 1)
[[5]
 [4]
 [5]
 ...
 [1]
 [4]
 [8]]
```

***Figure 29**: Prediction matrix after processing*

Conclusion

To conclude, this project was the occasion to create a complete Machine Learning model, which would have seemed to me unthinkable just a few weeks ago.

This project was broken down into a few major parts, who are, setting up an environment on GOOGLE COLLABORATORY, examining data from several CSV files, preprocessing the data, creating a Machine Learning model, training this model, improving its performance and, finally, evaluating this model. This project also allowed me to compare different methods to implement a Machine Learning model.

It was a very interesting project, which allowed me to review everything we had done in LABs, but also what we have seen during the Machine Learning courses.