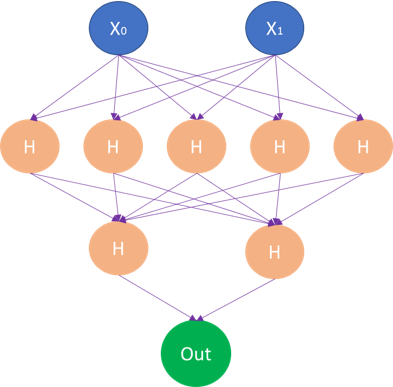
**NN - Classificateur Perceptron multicouche (MLPClassifier)**

1. Introduction
2. Chargement des librairies
3. MLPClassifier pour la classification multiclasse
4. Chargement des données
5. Prétraitement des données
6. MLPClassifier
7. Évaluation du modèle
8. Réglage des paramètres hyper

# Introduction



Quand on pense au Deep Learning, on pense immédiatement aux bibliothèques bien connues telles que Keras, PyTorch ou TensorFlow. Mais il existe une très populaire bibliothèque Scikit-Learn, qui est également capable de modéliser l'apprentissage en profondeur de base.

Pour ce rapport, l’ensemble de données Iris\_data.csv est utilisée.

# Chargement des librairies

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import accuracy\_score

from sklearn.metrics import plot\_confusion\_matrix

from sklearn.metrics import classification\_report

from sklearn.model\_selection import GridSearchCV

# MLPClassifier pour la classification binaire

Le perceptron multicouche (MLP) est un modèle de réseau de neurones artificiels à action directe qui mappe des ensembles de données d'entrée à un ensemble de sorties appropriées.

Un MLP se compose de plusieurs couches et chaque couche est entièrement connectée à la suivante.

Les nœuds des couches sont des neurones à fonctions d'activation non linéaires, à l'exception des nœuds de la couche d'entrée.

Entre la couche d'entrée et la couche de sortie, il peut y avoir une ou plusieurs couches cachées non linéaires.

## Chargement des données

df = pd.read\_csv('Iris\_Data.csv')

print(df)

|  | **sepal\_length** | **sepal\_width** | **petal\_length** | **petal\_width** | **species** |
| --- | --- | --- | --- | --- | --- |
| 0 | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |
| 1 | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| 2 | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| 4 | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |
| ... | ... | ... | ... | ... | ... |
| 145 | 6.7 | 3.0 | 5.2 | 2.3 | Iris-virginica |
| 146 | 6.3 | 2.5 | 5.0 | 1.9 | Iris-virginica |
| 147 | 6.5 | 3.0 | 5.2 | 2.0 | Iris-virginica |
| 148 | 6.2 | 3.4 | 5.4 | 2.3 | Iris-virginica |
| 149 | 5.9 | 3.0 | 5.1 | 1.8 | Iris-virginica |

150 rows × 5 columns

Examinons la variable cible :

df['species'].value\_counts()

Iris-setosa 50

Iris-versicolor 50

Iris-virginica 50

Name: species, dtype: int64

## Pré-traitement des données

x = df.drop('species', axis=1)

y = df['species']

trainX, testX, trainY, testY = train\_test\_split(x, y, test\_size = 0.2)

Pour former un réseau MLP, les données doivent toujours être mises à l'échelle car elles y sont très sensibles.

sc = StandardScaler()

scaler = sc.fit(trainX)

trainX\_scaled = scaler.transform(trainX)

testX\_scaled = scaler.transform(testX)

## MLPClassifier

Supposons que nous ayons deux variables prédictives et que nous souhaitions effectuer une classification. Pour cela, on saisi les paramètres suivants au niveau du modèle :

Entraînons donc notre premier MLP (avec un plus grand nombre de couches) :

mlp\_clf =  MLPClassifier(hidden\_layer\_sizes=(5,2),

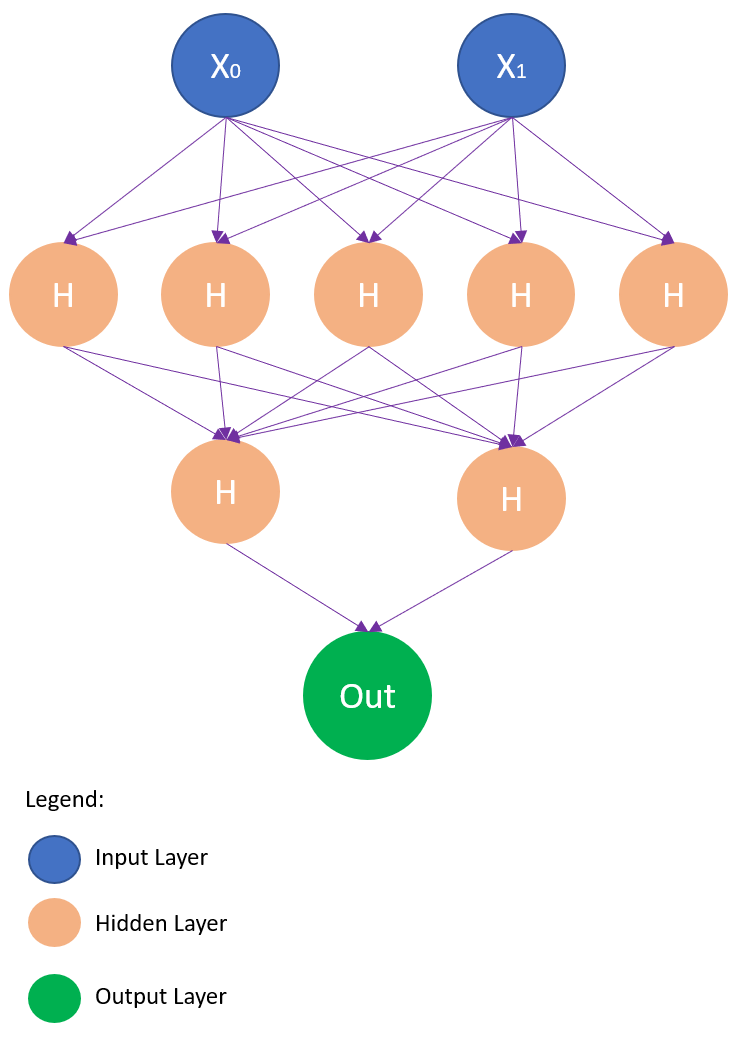
                        max\_iter = 300,activation = 'relu',

                        solver = 'adam')

mlp\_clf.fit(trainX\_scaled, trainY)

* hidden\_layer\_sizes : Avec ce paramètre, nous pouvons spécifier le nombre de couches et le nombre de nœuds que nous voulons avoir dans le Neural Network Classifier. Chaque élément du tuple représente le nombre de nœuds à la ième position, où i est l'index du tuple. Ainsi, la longueur du tuple indique le nombre total de couches cachées dans le réseau de neurones.
* max\_iter : indique le nombre d'époques.
* activation : la fonction d'activation des couches masquées.
* solveur : ce paramètre spécifie l'algorithme d'optimisation du poids sur les nœuds.

La structure de réseau créée dans le processus ressemblerait à ceci :



## Évaluation du modèle

Les métriques pouvant être utilisées pour mesurer les performances des algorithmes de classification doivent être connues.

y\_pred = mlp\_clf.predict(testX\_scaled)

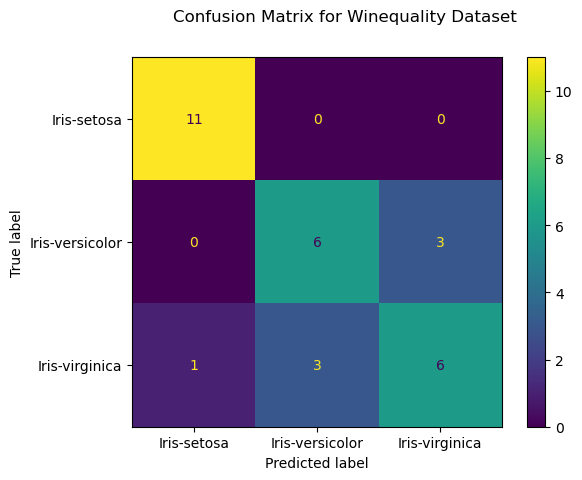
print('Accuracy: {:.2f}'.format(accuracy\_score(testY, y\_pred)))

Accuracy: 0.77

fig = plot\_confusion\_matrix(mlp\_clf, testX\_scaled, testY, display\_labels=mlp\_clf.classes\_)

fig.figure\_.suptitle("Confusion Matrix for Winequality Dataset")

plt.show()



print(classification\_report(testY, y\_pred))

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Precision** | **Recall** | **F1-score** | **Support** |
| **Iris-setosa** | 0.33 | 0.44 | 0.38 | 9 |
| **Iris-versicolor** | 0.22 | 0.29 | 0.25 | 7 |
| **Iris-virginica** | 0.56 | 0.36 | 0.43 | 14 |
| **accuracy** |  |  | 0.37 | 30 |
| **macro avg** | 0.37 | 0.36 | 0.36 | 30 |
| **weighted avg** | 0.41 | 0.37 | 0.38 | 30 |

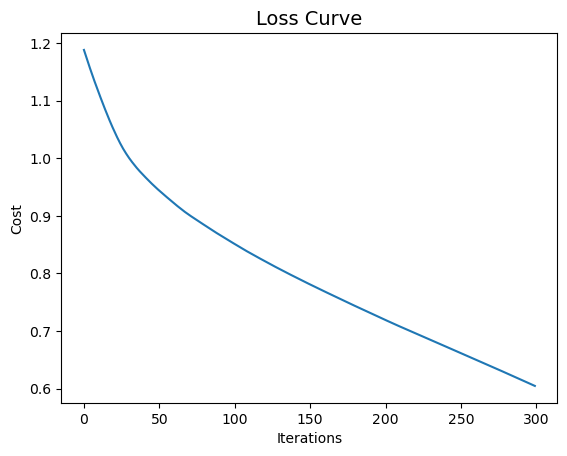
plt.plot(mlp\_clf.loss\_curve\_)

plt.title("Loss Curve", fontsize=14)

plt.xlabel('Iterations')

plt.ylabel('Cost')

plt.show()



## Hyper réglage des paramètres

param\_grid = {

    'hidden\_layer\_sizes': [(150,100,50), (120,80,40), (100,50,30)],

    'max\_iter': [50, 100, 150],

    'activation': ['tanh', 'relu'],

    'solver': ['sgd', 'adam'],

    'alpha': [0.0001, 0.05],

    'learning\_rate': ['constant','adaptive'],

}

grid = GridSearchCV(mlp\_clf, param\_grid, n\_jobs= -1, cv=5)

grid.fit(trainX\_scaled, trainY)

print(grid.best\_params\_)

{'activation': 'tanh', 'alpha': 0.0001, 'hidden\_layer\_sizes': (150, 100, 50), 'learning\_rate': 'constant', 'max\_iter': 150, 'solver': 'adam'}

grid\_predictions = grid.predict(testX\_scaled)

print('Accuracy: {:.2f}'.format(accuracy\_score(testY, grid\_predictions)))

Accuracy: 0.93