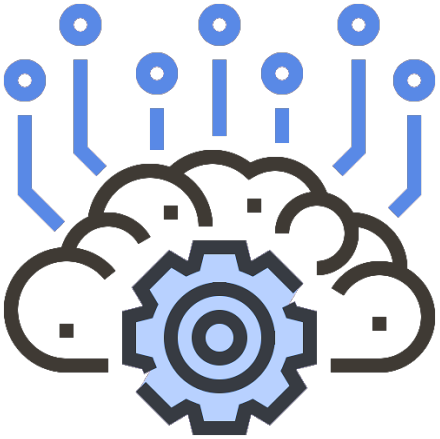
INPT

المعهد الوطني للبريد والمواصلات

ⵙⵓⵔⵜ ⵏⵓⵎⵓⵔⵜ ⵏ ⵜⵓⵎⵓⵔⵜ ⴰ ⵜⵉⵎⵙⵙⵓⵔⵉⵏ

Institut National des Postes et Télécommunications



Deep Learning

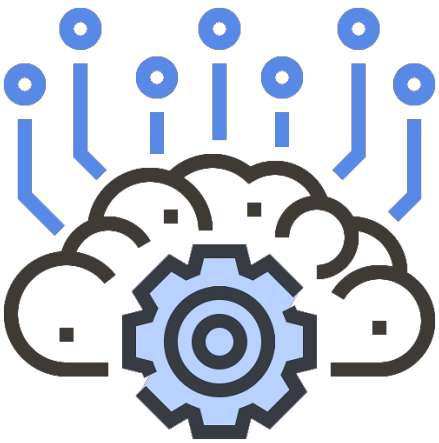
RAPPORT – TP2

REALISE PAR :

- *EL-ASRI NOSSAIBA*
- *CHARAFI ASMAA*
- *AITJILLALI NOUHAILA*
- *BOUJIDA HAFSA*

ENCADRE PAR :

- *M. IBN ELHAJ*



Deep Learning

TP2 : Construisez votre premier réseau neuronal MLP sous Python

Sommaire :

1. Les notions de base :

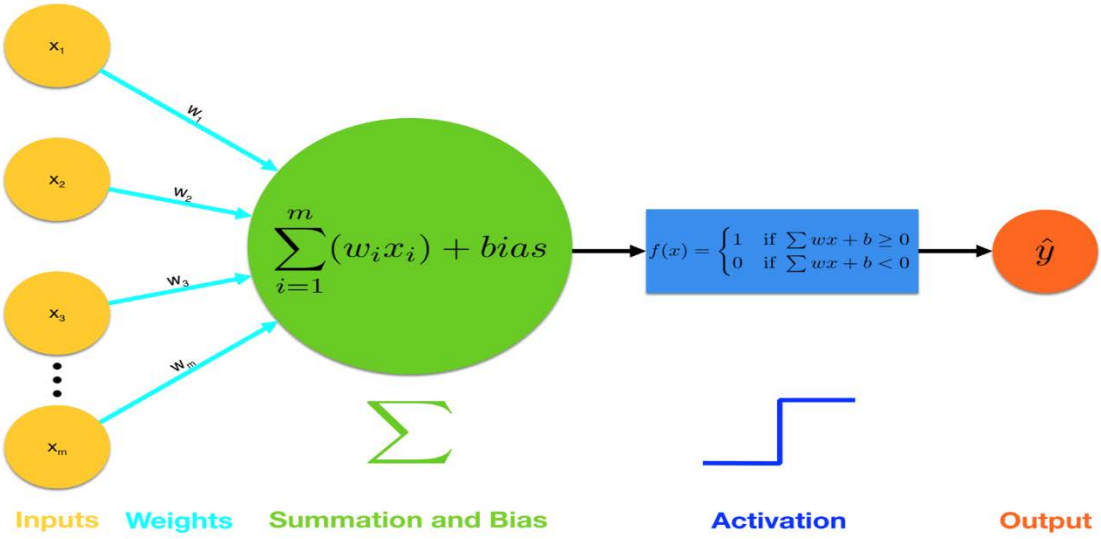
- **Un réseau neuronal :**

Un système dont la conception est à l'origine schématiquement inspiré du fonctionnement des neurones biologiques, et qui par la suite s'est rapproché des méthodes statistiques

- **Les Caractéristiques des réseaux de neurones artificiels :**
  - Un grand nombre de neurones de traitement très simples
  - Un grand nombre de liaisons pondérées entre les éléments
  - Représentation distribuée des connaissances sur les connexions
  - Les connaissances sont acquises en réseau grâce à un processus d'apprentissage

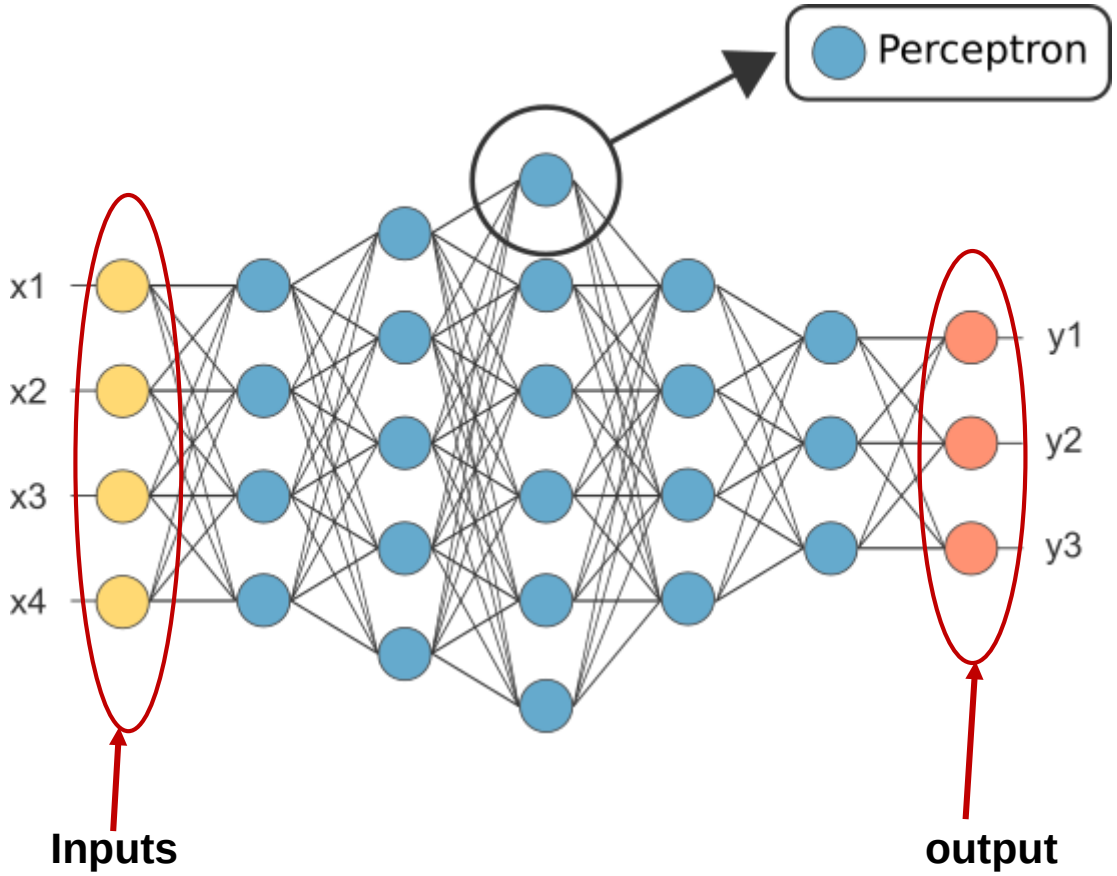
- **Un perceptron :**

Un perceptron peut être compris comme tout ce qui prend plusieurs entrées et produit une seule sortie



- **Multi-layer perceptron (MLP) :**

  Le perceptron multicouche un type de réseau neuronal artificiel organisé en plusieurs couches (une couche d'entrée, une couche cachée, une couche de sortie) au sein desquelles une information circule de la couche d'entrée vers la couche de sortie uniquement ; il s'agit donc d'un réseau à propagation directe. Chaque couche est constituée d'un nombre variable de neurones, les neurones de la dernière couche étant les sorties du système global



2. Le code Python :

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Feb 15 09:14:38 2021

@author: group2
"""
import numpy as np

#Input array
X=np.array([[1,0,1,0],[1,0,1,1],[0,1,0,1]])

#Output
y=np.array([[1],[1],[0]])

#Sigmoid Function
def sigmoid (x):  return 1/(1 + np.exp(-x))

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):  return x * (1 - x)


#Variable initialization
epoch=5000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = X.shape[1] #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer


#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
```

```python

for i in range(epoch):

    #Forward Propogation
    hidden_layer_input1=np.dot(X,wh)
    hidden_layer_input=hidden_layer_input1 + bh
    hiddenlayer_activations = sigmoid(hidden_layer_input)
    output_layer_input1=np.dot(hiddenlayer_activations,wout)
    output_layer_input= output_layer_input1+ bout
    output = sigmoid(output_layer_input)


    #Backpropagation
    E = y-output
    slope_output_layer = derivatives_sigmoid(output)
    slope_hidden_layer = derivatives_sigmoid(hiddenlayer_activations)
    d_output = E * slope_output_layer
    Error_at_hidden_layer = d_output.dot(wout.T)
    d_hiddenlayer = Error_at_hidden_layer * slope_hidden_layer
    wout += hiddenlayer_activations.T.dot(d_output) *lr
    bout += np.sum(d_output, axis=0,keepdims=True) *lr
    wh += X.T.dot(d_hiddenlayer) *lr
    bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr


print (output)
```

3 . Explication du code :

- D’abord la déclaration de bibliothèque Numpy

```
import numpy as np
```

- On a défini X comme matrice d’entré et Y comme matrice de sortie

```
#Input array
X=np.array([[1,0,1,0],[1,0,1,1],[0,1,0,1]])

#Output
y=np.array([[1],[1],[0]])
```

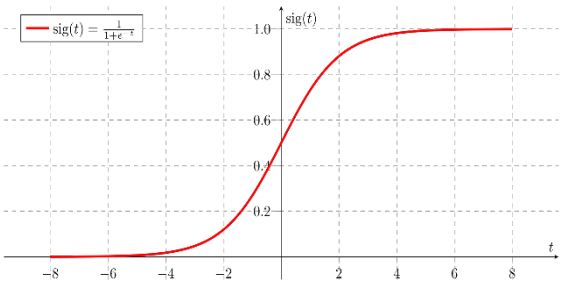
- On a ajouté la fonction sigmoïde (on l’utilise dans forward propagation) et son dérivé (on l’utilise dans backpropagation) :

```
#Sigmoid Function
def sigmoid (x):  return 1/(1 + np.exp(-x))

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):  return x * (1 - x)
```

  - La fonction sigmoïde : une fonction de répartition de la loi logistique. Elle est souvent utilisée dans les réseaux de neurones parce qu'elle est dérivable.

$$f(x) = \frac{1}{1 + \mathrm{e}^{-x}} \text{ pour tout réel } x$$



- We initialize the number of features in our dataset (1), the number of hidden layer neurons (3), the number of output neurons (1) and set how many epochs we want, what our learning rate lr will be.

```
#Variable initialization
epoch=5000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = X.shape[1] #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer
```

- Après, on a initialize wh, bh, wout, bout

```
#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
```

  - Wh : matrice de poids à la couche cachée.
  - Bh : matrice de biais vers la couche cachée.
  - Wout : matrice de poids sur la couche de sortie.
  - Bout : matrice de polarisation à la couche de sortie.

- **Forward Propagation:**

  Def : NN prend plusieurs entrées, les traite via plusieurs neurones de plusieurs couches cachées et renvoie le résultat en utilisant une couche de sortie.

  Les étapes :
  - Tout d'abord, nous obtiendrons le produit scalaire matriciel du poids et des valeurs d'entrée.
  - Ensuite, nous ajouterons le biais aux produits.
  - Ensuite, Effectuer une transformation non linéaire en utilisant une fonction d'activation (Sigmoid)

- répéter cela avec le résultat de la couche de sortie

```
for i in range(epoch):

    #Forward Propogation
    hidden_layer_input1=np.dot(X,wh)
    hidden_layer_input=hidden_layer_input1 + bh
    hidden_layer_activations = sigmoid(hidden_layer_input)
    output_layer_input1=np.dot(hidden_layer_activations,wout)
    output_layer_input= output_layer_input1+ bout
    output = sigmoid(output_layer_input)
```

- **Back Propagation :**

  Def : obtenir le poids des neurones tel que l'erreur totale soit minimisée

  Les étapes :

  - Calculer l'erreur, la différence entre y et la sortie.
  - Calculez la pente / le gradient des neurones des couches cachées et de sortie
  - Calculer le facteur de changement (delta) à la couche de sortie, en fonction du gradient d'erreur multiplié par la pente d'activation de la couche de sortie
  - nous prendrons le produit scalaire du delta de la couche de sortie avec les paramètres de poids des bords entre la couche cachée et la couche de sortie, car l'erreur se propage de nouveau dans le réseau
  -) Calculer le facteur de changement (delta) au niveau de la couche cachée, multiplier l'erreur au niveau de la couche cachée avec la pente d'activation de la couche cachée
  - Mettre à jour les poids à la sortie et la couche cachée

```
#Backpropagation
E = y-output
slope_output_layer = derivatives_sigmoid(output)
slope_hidden_layer = derivatives_sigmoid(hiddenlayer_activations)
d_output = E * slope_output_layer
Error_at_hidden_layer = d_output.dot(wout.T)
d_hiddenlayer = Error_at_hidden_layer * slope_hidden_layer
wout += hiddenlayer_activations.T.dot(d_output) *lr
bout += np.sum(d_output, axis=0,keepdims=True) *lr
wh += X.T.dot(d_hiddenlayer) *lr
bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr
```

- Afficher la sortie :

```
print (output)
```

Le résultat :

```
In [3]: runfile('C:/Users/Pc/untitled0.py', wdir='C:/Users/Pc')
[[0.97700189]
 [0.96933046]
 [0.03880442]]
```

- Nos sorties d'origine étaient 1, 1, 0, et ces sorties sont très proches.

- Si on veut voir l'erreur

```
print(E)
```

Le résultat :

```
[[ 0.02429638]
 [ 0.03383288]
 [-0.04958817]]
```

4 . Conclusion :

L’objectif de ce TP et de savoir comment Construire un réseau neuronal MLP sous Python et cela pour faciliter notre assimilation du cours Deep Learning et pour s'habituer de résoudre des problèmes en utilisant le langage Python