15 - Construction d'un réseau de neurones avec Python

Une classe de réseau de neurones

Nous avons appris dans le chapitre précédent de notre tutoriel sur les réseaux de neurones les faits les plus importants concernant les poids. Nous avons vu comment ils sont utilisés et comment nous pouvons les implémenter en Python. Nous avons vu que la multiplication des poids avec les valeurs d'entrée peut être réalisée avec des tableaux de Numpy en appliquant la multiplication matricielle.

Cependant, ce que nous n'avions pas fait, c'était de les tester dans un environnement réel de réseau neuronal. Nous devons d'abord créer cet environnement. Nous allons maintenant créer une classe en Python, implémentant un réseau neuronal. Nous allons procéder par petites étapes afin que tout soit facile à comprendre.

Les méthodes les plus essentielles dont notre classe a besoin sont

- `__init__` pour initialiser une classe, c'est-à-dire que nous allons définir le nombre de neurones pour chaque couche et initialiser les matrices de poids.
- `run` : Une méthode qui est appliquée à un échantillon, que nous voulons classer. Elle applique cet échantillon au réseau neuronal. On pourrait dire que l'on " exécute " le réseau pour " prédire " le résultat. Dans d'autres implémentations, cette méthode est souvent appelée " predict ".
- `train` : Cette méthode reçoit en entrée un échantillon et la valeur cible correspondante. Avec cette entrée, elle peut ajuster les valeurs de poids si nécessaire. Cela signifie que le réseau apprend à partir d'une entrée. Du point de vue de l'utilisateur, nous "entrainons" le réseau. Dans `sklearn` par exemple, cette méthode est appelée `fit` .

Nous remettrons à plus tard la définition de la méthode `train` and `run` . Les matrices de poids doivent être initialisées à l'intérieur de la méthode `__init__` . Nous le faisons indirectement. Nous définissons une méthode `create_weight_matrices` et l'appelons dans `__init__` . De cette façon, la méthode init reste claire.

Nous allons également reporter l'ajout de nœuds de biais aux couches.

Le code Python suivant contient une implémentation d'une classe de réseau neuronal appliquant les connaissances que nous avons développées dans le chapitre précédent :

Entrée [1]:
```
import numpy as np
from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,  # corresponds to the number of c
                 no_of_hidden_nodes,
                 learning_rate):
        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.learning_rate = learning_rate
        self.create_weight_matrices()

    def create_weight_matrices(self):
        rad = 1 / np.sqrt(self.no_of_in_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_in_hidden = X.rvs((self.no_of_hidden_nodes,
                                        self.no_of_in_nodes))
        rad = 1 / np.sqrt(self.no_of_hidden_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_hidden_out = X.rvs((self.no_of_out_nodes,
                                         self.no_of_hidden_nodes))


    def train(self):
        pass

    def run(self):
        pass
```

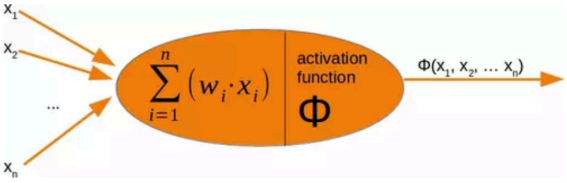
Nous ne pouvons pas faire grand chose avec ce code, mais nous pouvons au moins l'initialiser. Nous pouvons également jeter un coup d'oeil aux matrices de poids :

Entrée [2]:
```
simple_network = NeuralNetwork(no_of_in_nodes = 3,
                               no_of_out_nodes = 2,
                               no_of_hidden_nodes = 4,
                               learning_rate = 0.1)
print(simple_network.weights_in_hidden)
print(simple_network.weights_hidden_out)
```

```
[[-0.17464757 -0.35153434  0.24434159]
 [-0.47085764  0.42745673 -0.41198079]
 [-0.35140133 -0.17978873  0.57095284]
 [ 0.2037165   0.36719322 -0.16214184]]
[[ 0.06428107 -0.39045835  0.20770618 -0.02467576]
 [ 0.05831799  0.13057882  0.17464291  0.46415516]]
```

Fonctions d'activation, Sigmoïde et ReLU

Avant de pouvoir programmer la méthode d'exécution, nous devons nous occuper de la fonction d'activation. Nous avions le diagramme suivant dans le chapitre d'introduction aux réseaux de neurones :



Les valeurs d'entrée d'un perceptron sont traitées par la fonction de sommation et suivies par une fonction d'activation, transformant la sortie de la fonction de sommation en une sortie souhaitée et plus appropriée. La fonction de sommation signifie que nous aurons une multiplication matricielle des vecteurs de poids et des valeurs d'entrée.

Il existe de nombreuses fonctions d'activation différentes utilisées dans les réseaux neuronaux. L'une des vues d'ensemble les plus complètes des fonctions d'activation possibles se trouve sur Wikipedia.

La fonction sigmoïde est l'une des fonctions d'activation les plus utilisées. La fonction sigmoïde, que nous utilisons, est également connue sous le nom de fonction logistique.

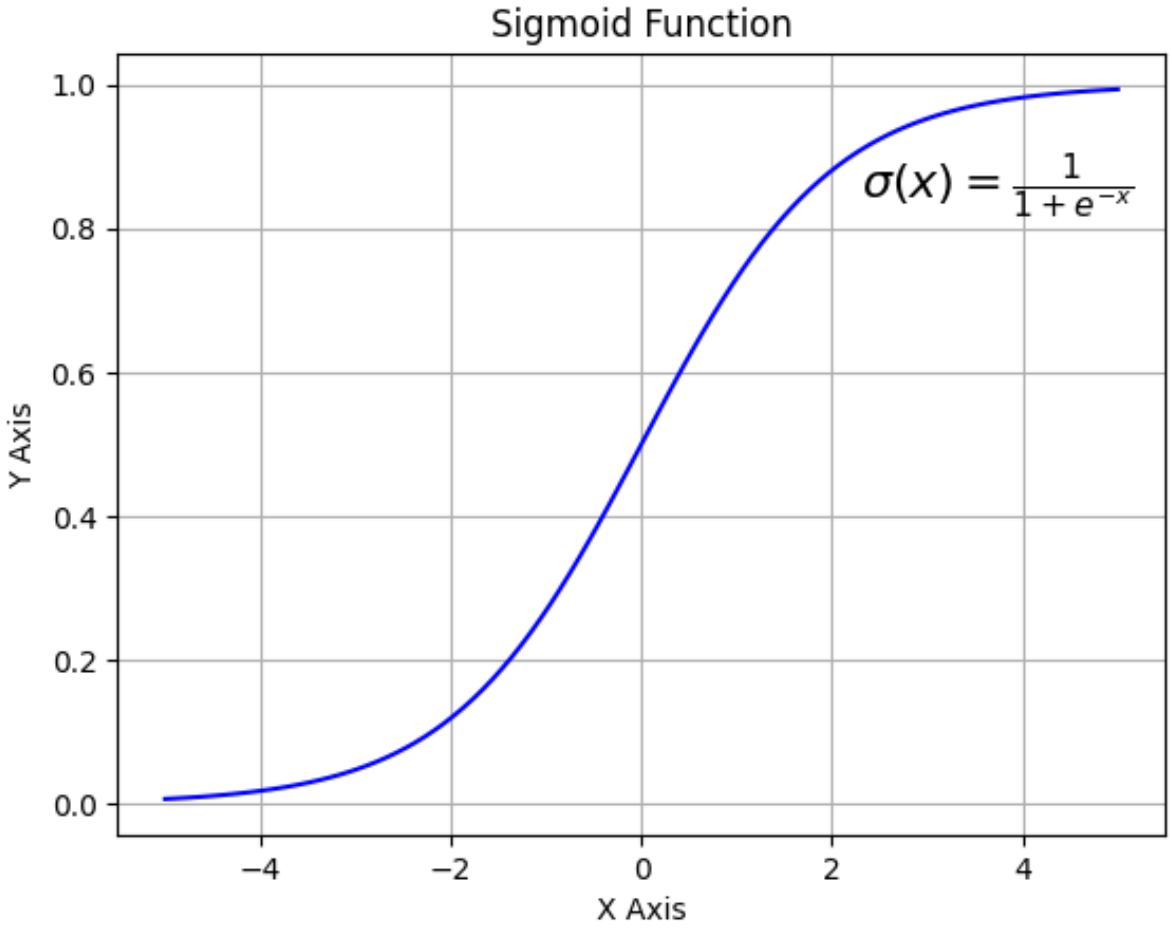
Elle est définie comme suit:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Regardons le graphique de la fonction sigmoïde. Nous utilisons `matplotlib` pour tracer la fonction sigmoïde :

Entrée [3]:

```
import numpy as np
import matplotlib.pyplot as plt
def sigma(x):
    return 1 / (1 + np.exp(-x))

X = np.linspace(-5, 5, 100)


plt.plot(X, sigma(X),'b')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Sigmoid Function')

plt.grid()

plt.text(2.3, 0.84, r'$\sigma(x)=\frac{1}{1+e^{-x}}$', fontsize=16)


plt.show()
```



En observant le graphique, nous pouvons voir que la fonction sigmoïde fait correspondre un nombre donné x à une plage de nombres comprise entre 0 et 1. 0 et 1 ne sont pas inclus ! Plus la valeur de x est grande, plus la valeur de la fonction sigmoïde se rapproche de 1 et plus x est petit, plus la valeur de la fonction sigmoïde se rapproche de 0.

Au lieu de définir nous-mêmes la fonction sigmoïde, nous pouvons également utiliser la fonction expit de scipy.special, qui est une implémentation de la fonction sigmoïde. Elle peut être appliquée à différentes classes de données comme int, float, list, numpy,ndarray et ainsi de suite. Le résultat est un ndarray de la même forme que les données d'entrée x.

Entrée [4]:
```
from scipy.special import expit
print(expit(3.4))
print(expit([3, 4, 1]))
print(expit(np.array([0.8, 2.3, 8])))
```

```
0.9677045353015494
[0.95257413 0.98201379 0.73105858]
[0.68997448 0.90887704 0.99966465]
```

La fonction logistique est souvent utilisée dans les réseaux neuronaux pour introduire une non-linéarité dans le modèle et pour adapter les signaux dans une plage spécifiée, c'est-à-dire 0 et 1. Elle est également trés utilisée car dérivable, ce qui est nécessaire dans la méthode de rétropropagation.

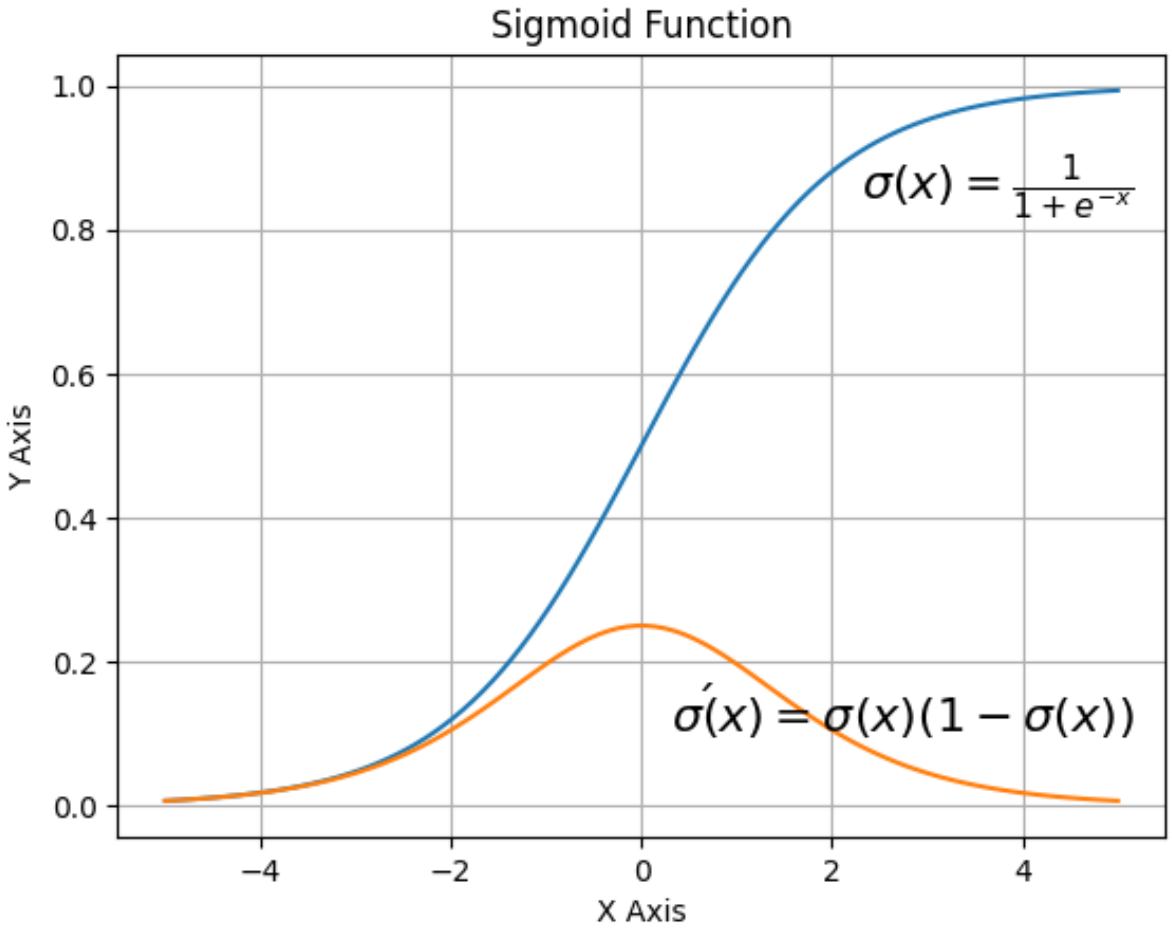
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

et sa dérivée :

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Entrée [5]:

```
import numpy as np
import matplotlib.pyplot as plt
def sigma(x):
    return 1 / (1 + np.exp(-x))

X = np.linspace(-5, 5, 100)

plt.plot(X, sigma(X))
plt.plot(X, sigma(X) * (1 - sigma(X)))

plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Sigmoid Function')

plt.grid()

plt.text(2.3, 0.84, r'$\sigma(x)=\frac{1}{1+e^{-x}}$', fontsize=16)
plt.text(0.3, 0.1, r'$\sigma\'(x) = \sigma(x)(1 - \sigma(x))$', fon

plt.show()
```



Nous pouvons également définir notre propre fonction sigmoïde avec le décorateur `vectorize` de numpy :

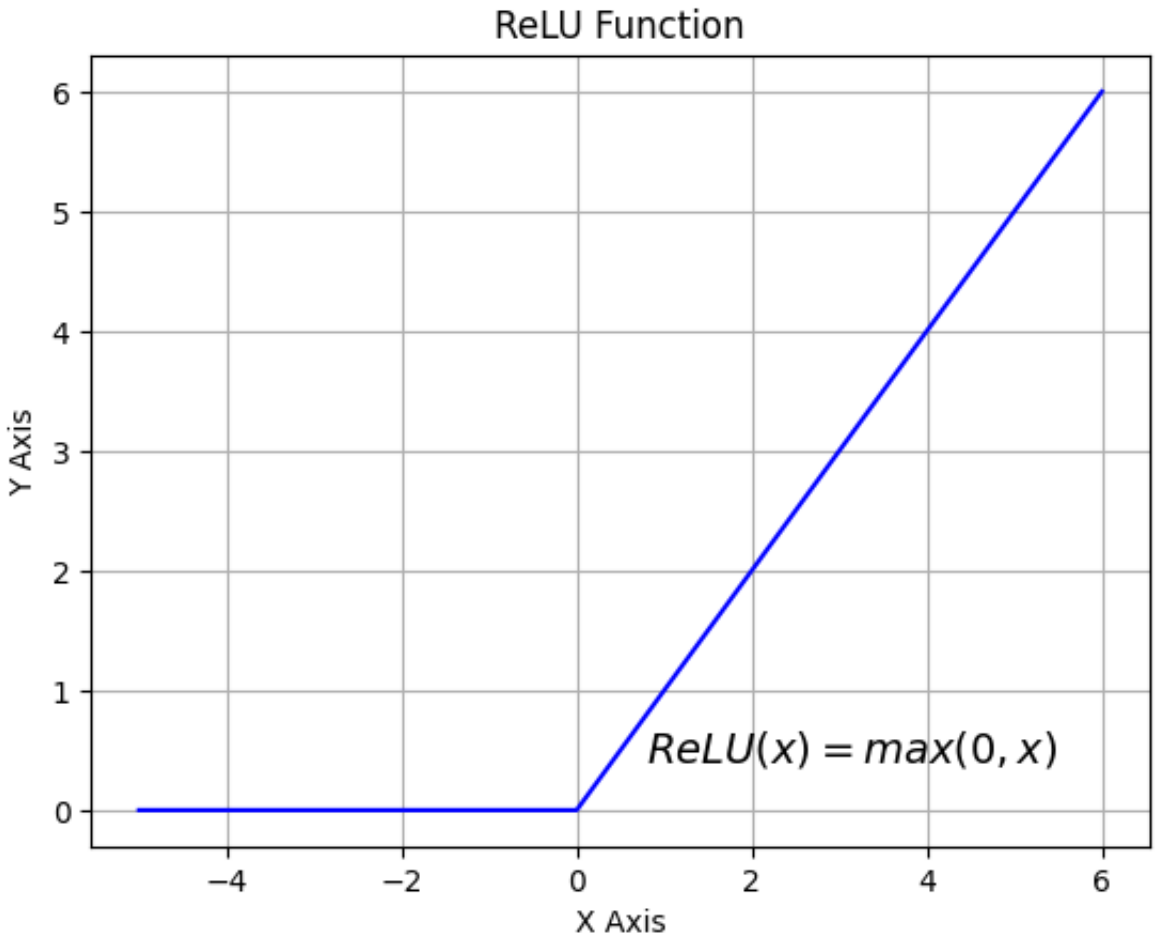
Entrée [6]:
```
@np.vectorize
def sigmoid(x):
    return 1 / (1 + np.e ** -x)

#sigmoid = np.vectorize(sigmoid)
sigmoid([3, 4, 5])
```

Out[6]:
```
array([0.95257413, 0.98201379, 0.99330715])
```

Une autre fonction d'activation facile à utiliser est la fonction ReLU. ReLU signifie unité linéaire rectifiée. Elle est également connue sous le nom de fonction de rampe. Elle est définie comme la partie positive de son argument, c'est-à-dire $y = \max(0, x)$. C'est "actuellement, la fonction d'activation la plus réussie et la plus utilisée est l'unité linéaire rectifiée (ReLU). La fonction ReLu est plus efficace du point de vue du calcul que les fonctions de type Sigmoïde, car Relu signifie seulement choisir le maximum entre 0 et l'argument x, alors que les Sigmoïdes doivent effectuer des opérations exponentielles coûteuses.

Entrée [7]:
```
# alternative activation function
def ReLU(x):
    return np.maximum(0.0, x)

# derivation of relu
def ReLU_derivation(x):
    if x <= 0:
        return 0
    else:
        return 1
```

Entrée [8]:
```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-5, 6, 100)
plt.plot(X, ReLU(X),'b')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('ReLU Function')
plt.grid()
plt.text(0.8, 0.4, r'$ReLU(x)=max(0, x)$', fontsize=14)
plt.show()
```



Ajout d'une méthode `run`

Nous avons maintenant tout réuni pour implémenter la méthode run (ou predict) de notre classe de réseau neuronal. Nous allons utiliser scipy.special comme fonction d'activation et la renommer activation_function :

Entrée [9]:
```
from scipy.special import expit as activation_function
```

Tout ce que nous avons à faire dans la méthode d'exécution est le suivant.

1. Multiplication matricielle du vecteur d'entrée et de la matrice weights_in_hidden.
2. Application de la fonction d'activation au résultat de l'étape 1.
3. Multiplication matricielle du vecteur résultat de l'étape 2 et de la matrice des poids_in_hidden.
4. Pour obtenir le résultat final : Application de la fonction d'activation au résultat de l'étape 3

Entrée [10]:

```
import numpy as np
from scipy.special import expit as activation_function
from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)


class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,
                 no_of_hidden_nodes,
                 learning_rate):
        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.learning_rate = learning_rate
        self.create_weight_matrices()

    def create_weight_matrices(self):
        """ A method to initialize the weight matrices of the neura
        rad = 1 / np.sqrt(self.no_of_in_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_in_hidden = X.rvs((self.no_of_hidden_nodes,
                                        self.no_of_in_nodes))
        rad = 1 / np.sqrt(self.no_of_hidden_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_hidden_out = X.rvs((self.no_of_out_nodes,
                                         self.no_of_hidden_nodes))


    def train(self, input_vector, target_vector):
        pass


    def run(self, input_vector):
        """
        running the network with an input vector 'input_vector'.
        'input_vector' can be tuple, list or ndarray
        """
        # turning the input vector into a column vector
        # turn one-dimensional into 2-dimensional column vector:
        input_vector = np.array(input_vector, ndmin=2).T
        input_hidden = activation_function(self.weights_in_hidden @
        output_vector = activation_function(self.weights_hidden_out
        return output_vector
```

Nous pouvons instancier une instance de cette classe, qui sera un réseau neuronal. Dans l'exemple suivant, nous créons un réseau avec deux nœuds d'entrée, quatre nœuds cachés et deux nœuds de sortie.

Entrée [11]:
```
simple_network = NeuralNetwork(no_of_in_nodes=2,
                               no_of_out_nodes=2,
                               no_of_hidden_nodes=4,
                               learning_rate=0.6)
```

Nous pouvons appliquer la méthode run à tous les tableaux dont la forme est (2,), ainsi qu'aux listes et aux tuples comportant deux éléments numériques. Le résultat de l'appel est défini par les valeurs aléatoires des poids :

Entrée [12]:
```
simple_network.run([(3, 4)])
```

Out[12]:
```
array([[0.49791969],
       [0.39456095]])
```

Entrée [ ]:
```

```