14 - Réseaux neuronaux, structure, poids et matrices

Introduction

Nous avons présenté les idées de base sur les réseaux neuronaux dans le chapitre précédent de notre tutoriel sur l'apprentissage automatique.

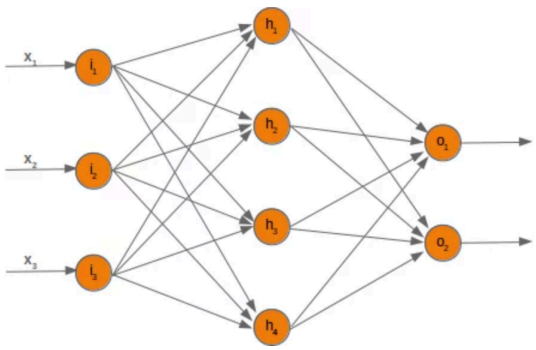
Nous avons souligné la similitude entre les neurones et les réseaux de neurones en biologie. Nous avons également présenté de très petits réseaux neuronaux artisanaux et introduit les frontières de décision et le problème XOR.

Dans les exemples simples que nous avons présentés jusqu'à présent, nous avons vu que les poids sont les éléments essentiels d'un réseau neuronal. Avant de commencer à écrire un réseau neuronal à couches multiples, nous devons examiner de plus près les poids.

Nous devons voir comment initialiser les poids et comment multiplier efficacement les poids avec les valeurs d'entrée.

Dans les chapitres suivants, nous allons concevoir un réseau neuronal en Python, qui se compose de trois couches, à savoir la couche d'entrée, une couche cachée et une couche de sortie. Vous pouvez voir la structure de ce réseau neuronal dans le diagramme suivant. Nous avons une couche d'entrée avec trois nœuds i_1, i_2, i_3. Ces noeuds reçoivent les valeurs d'entrée correspondantes x_1, x_2, x_3. La couche intermédiaire ou cachée comporte quatre nœuds h_1, h_2, h_3, h_4. L'entrée de cette couche provient de la couche d'entrée. Nous discuterons bientôt de ce mécanisme. Enfin, notre couche de sortie se compose des deux noeuds suivants o_1 et o_2.

La couche d'entrée est différente des autres couches. Les nœuds de la couche d'entrée sont passifs. Cela signifie que les neurones d'entrée ne modifient pas les données, c'est-à-dire qu'aucun poids n'est utilisé dans ce cas. Ils reçoivent une valeur unique et dupliquent cette valeur à leurs nombreuses sorties.



La couche d'entrée est constituée des nœuds i_1, i_2 et i_3. En principe, l'entrée est un vecteur unidimensionnel, comme (2, 4, 11). Un vecteur unidimensionnel est représenté dans `numpy` comme ceci :

Entrée [1]:
```
import numpy as np

input_vector = np.array([2, 4, 11])
print(input_vector)
```

```
[ 2  4 11]
```

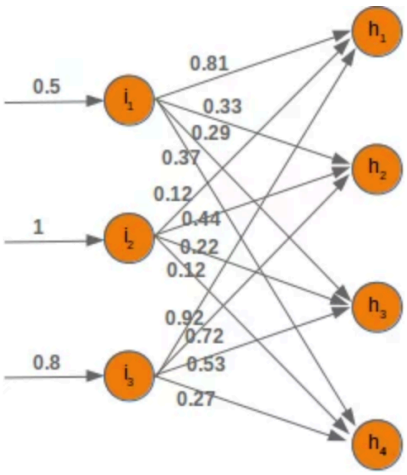
Dans l'algorithme, que nous écrirons plus tard, nous devrons le transposer en un vecteur colonne, c'est-à-dire un tableau bidimensionnel avec une seule colonne :

Entrée [2]:
```
import numpy as np

input_vector = np.array([2, 4, 11])
input_vector = np.array(input_vector, ndmin=2).T
print("The input vector:\n", input_vector)

print("The shape of this vector: ", input_vector.shape)
```

```
The input vector:
 [[ 2]
 [ 4]
 [11]]
The shape of this vector:  (3, 1)
```

Poids et matrices

À chacune des flèches de notre diagramme de réseau est associée une valeur de poids. Pour l'instant, nous n'examinerons que les flèches entre la couche d'entrée et la couche de sortie.



La valeur x_1 entrant dans le nœud i_1 sera distribué en fonction des valeurs des poids. Dans le diagramme suivant, nous avons ajouté quelques valeurs d'exemple. En utilisant ces valeurs, les valeurs d'entrée (Ih_1, Ih_2, Ih_3, Ih_4) dans les noeuds (h_1, h_2, h_3, h_4) de la couche cachée peut être calculée comme suit :

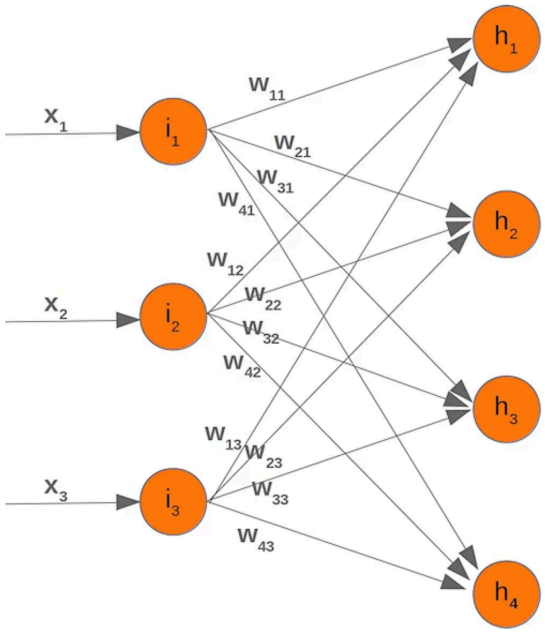
$$Ih_1 = 0.81 * 0.5 + 0.12 * 1 + 0.92 * 0.8$$

$$Ih_2 = 0.33 * 0.5 + 0.44 * 1 + 0.72 * 0.8$$

$$Ih_3 = 0.29 * 0.5 + 0.22 * 1 + 0.53 * 0.8$$

$$Ih_4 = 0.37 * 0.5 + 0.12 * 1 + 0.27 * 0.8$$

Ceux qui sont familiers avec les matrices et la multiplication matricielle verront à quoi cela se résume. Nous allons redessiner notre réseau et désigner les poids par w_{ij} :



Afin d'exécuter efficacement tous les calculs nécessaires, nous allons organiser les poids dans une matrice de poids. Dans le diagramme ci-dessus, les poids forment un tableau, que nous appellerons "poids_in_hidden" dans notre classe de réseau neuronal. Ce nom doit indiquer que les poids relient les nœuds d'entrée et les nœuds cachés, c'est-à-dire qu'ils se trouvent entre l'entrée et la couche cachée. Nous abrégerons également le nom par " wih ". La matrice des poids entre la couche cachée et la couche de sortie sera désignée par "who" :

$$wih = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix}$$

$$who = \begin{pmatrix} wh_{11} & wh_{12} & wh_{13} & wh_{14} \\ wh_{21} & wh_{22} & wh_{23} & wh_{24} \end{pmatrix}$$

Maintenant que nous avons défini nos matrices de poids, nous devons passer à l'étape suivante. Nous devons multiplier la matrice par le vecteur d'entrée. En fait, c'est exactement ce que nous avons fait manuellement dans notre exemple précédent.

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} w_{11} \cdot x_1 + w_{12} \cdot x_2 + w_{13} \cdot x_3 \\ w_{21} \cdot x_1 + w_{22} \cdot x_2 + w_{23} \cdot x_3 \\ w_{31} \cdot x_1 + w_{32} \cdot x_2 + w_{33} \cdot x_3 \\ w_{41} \cdot x_1 + w_{42} \cdot x_2 + w_{43} \cdot x_3 \end{pmatrix}$$

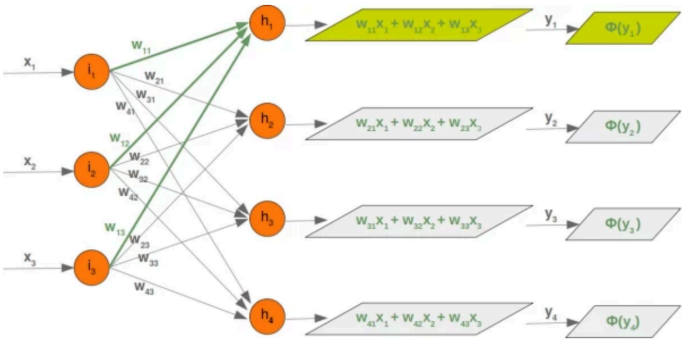
Nous avons une situation similaire pour la matrice 'qui' entre la couche cachée et la couche de sortie. Ainsi, la sortie z_1 et z_2 à partir des noeuds o_1 et o_2 peuvent également être calculés à l'aide de multiplications matricielles :

$$\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} wh_{11} & wh_{12} & wh_{13} & wh_{14} \\ wh_{21} & wh_{22} & wh_{23} & wh_{24} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} wh_{11} \cdot y_1 + wh_{12} \cdot y_2 + wh_{13} \\ wh_{21} \cdot y_1 + wh_{22} \cdot y_2 + wh_{23} \end{pmatrix}$$

Vous avez peut-être remarqué qu'il manque quelque chose dans nos calculs précédents. Nous avons montré dans notre chapitre d'introduction Réseaux de neurones à partir de zéro en Python que nous devons appliquer une fonction d'activation ou de pas Φ sur chacune de ces sommes.

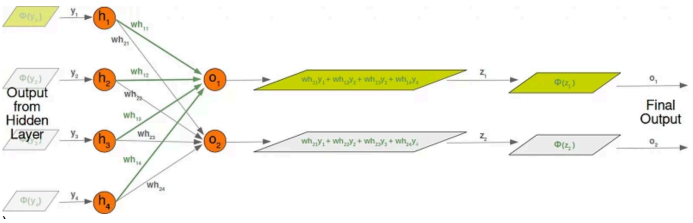
L'image suivante représente l'ensemble du flux de calcul, c'est-à-dire la multiplication matricielle et l'application successive de la fonction d'activation.

La multiplication matricielle entre la matrice wih et la matrice des valeurs des nœuds d'entrée x_1, x_2, x_3 calcule la sortie qui sera transmise à la fonction d'activation.



La sortie finale y_1, y_2, y_3, y_4 est l'entrée de la matrice de poids qui :

Même si le traitement est complètement analogique, nous allons également examiner en détail ce qui se passe entre notre couche cachée et la couche de sortie :



Initialisation des matrices de poids

L'un des choix importants à faire avant de former un réseau neuronal consiste à initialiser les matrices de poids. Nous ne savons rien des poids possibles, lorsque nous commençons. Nous pourrions donc commencer par des valeurs arbitraires ?

Comme nous l'avons vu, l'entrée de tous les nœuds, à l'exception des nœuds d'entrée, est calculée en appliquant la fonction d'activation à la somme suivante :

$$y_j = \sum_{i=1}^{n} w_{ji} \cdot x_i$$

(avec n étant le nombre de nœuds dans la couche précédente et y_j est l'entrée d'un nœud de la couche suivante)

Nous pouvons facilement voir que ce ne serait pas une bonne idée de mettre toutes les valeurs de poids à 0, car dans ce cas, le résultat de cette sommation sera toujours zéro. Cela signifie que notre réseau sera incapable d'apprendre. Il s'agit du pire choix, mais initialiser une matrice de poids à 1 est également un mauvais choix.

Les valeurs des matrices de poids doivent être choisies de manière aléatoire et non arbitraire. En choisissant une distribution normale aléatoire, nous avons brisé des situations symétriques possibles, qui peuvent et sont souvent mauvaises pour le processus d'apprentissage.

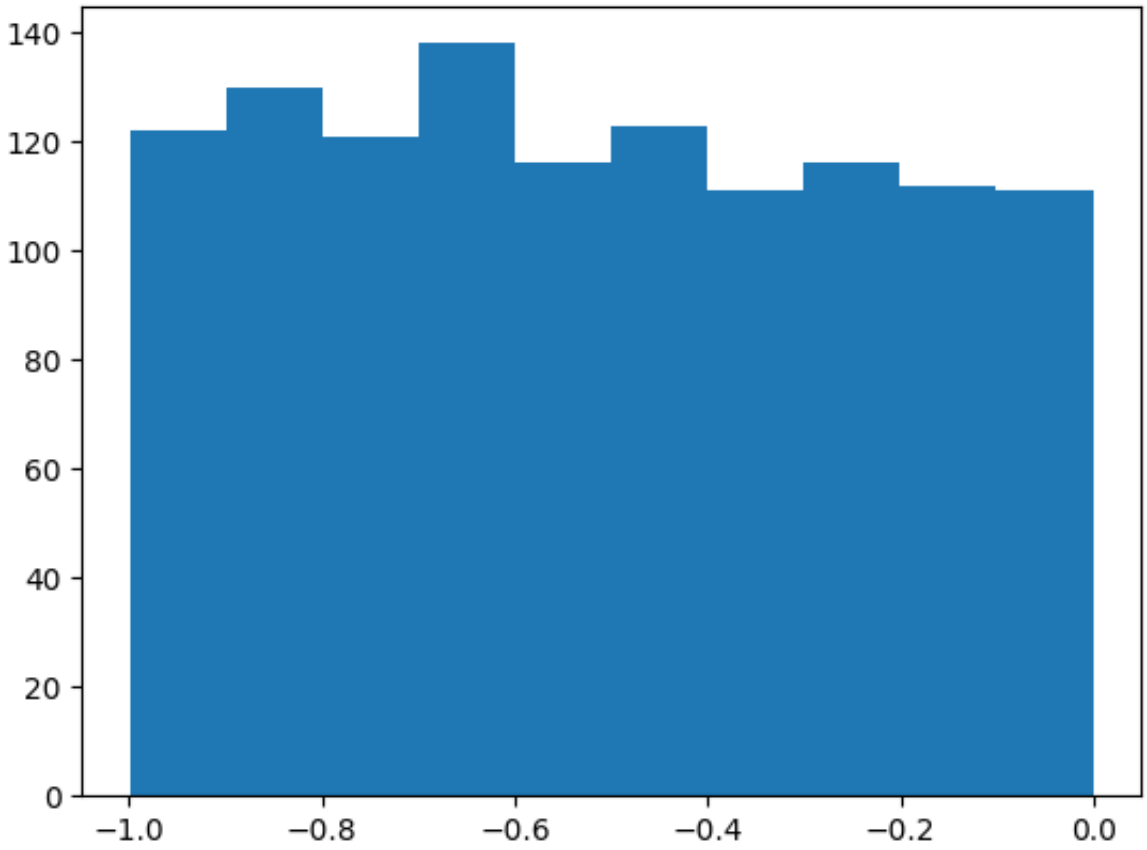
Il existe plusieurs façons d'initialiser les matrices de poids de manière aléatoire. La première que nous allons présenter est la fonction unity de numpy.random. Elle crée des échantillons qui sont uniformément distribués sur l'intervalle semi-ouvert `[low, high]`, ce qui signifie que la valeur basse est incluse et la valeur haute est exclue. Chaque valeur dans l'intervalle donné a la même probabilité d'être tirée par 'uniform'.

Entrée [3]:
```
import numpy as np

number_of_samples = 1200
low = -1
high = 0
s = np.random.uniform(low, high, number_of_samples)

# all values of s are within the half open interval [-1, 0) :
print(np.all(s >= -1) and np.all(s < 0))
```

```
True
```

L'histogramme des échantillons, créé avec la fonction uniforme dans notre exemple précédent, ressemble à ceci :

Entrée [4]:
```
import matplotlib.pyplot as plt
plt.hist(s)
plt.show()
```

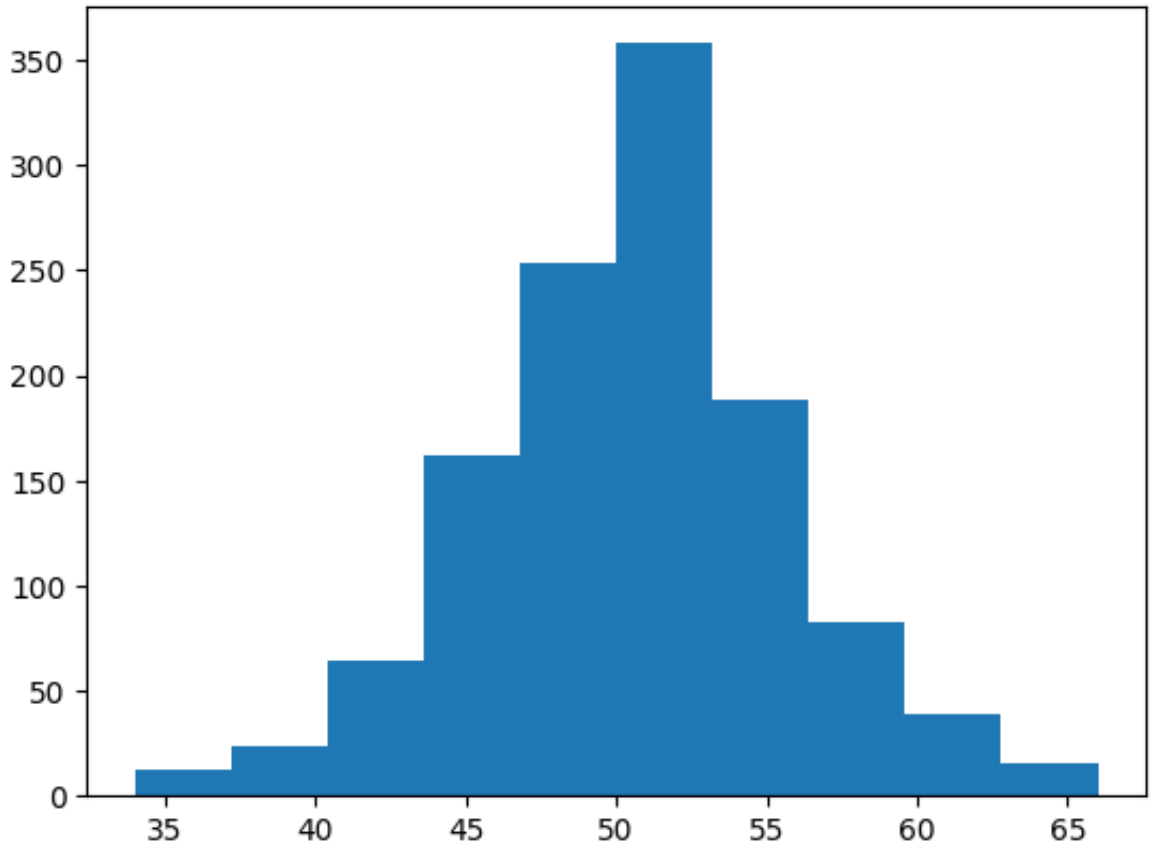


La prochaine fonction que nous allons examiner est 'binomial' de numpy.binomial :

```
binomial(n, p, size=None)
```

Elle tire des échantillons d'une distribution binomiale avec les paramètres spécifiés, n essais et la probabilité p de succès où n est un entier >= 0 et p est un flottant dans l'intervalle [0,1]. (n peut être entré sous la forme d'un flottant, mais il est tronqué en entier lors de son utilisation).

Entrée [5]:
```
s = np.random.binomial(100, 0.5, 1200)
plt.hist(s)
plt.show()
```



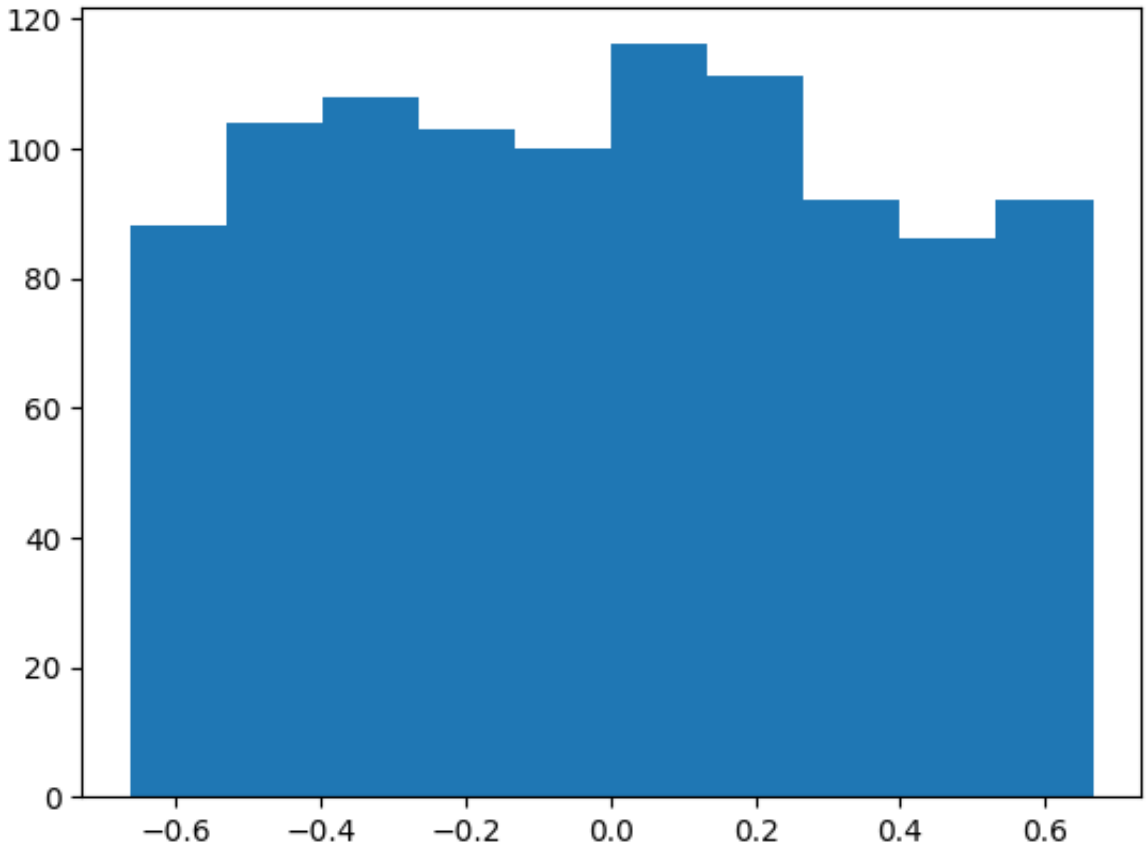
Nous aimons créer des nombres aléatoires avec une distribution normale, mais les nombres doivent être bornés. Ce n'est pas le cas avec np.random.normal(), car il n'offre aucun paramètre de limite.

Nous pouvons utiliser `truncnorm` de `scipy.stats` dans ce but.

La forme standard de cette distribution est une normale standard tronquée à l'intervalle `[a, b]` - remarquez que a et b sont définis sur le domaine de la normale standard. Pour convertir les valeurs de clip pour une moyenne et un écart-type spécifiques, utilisez :

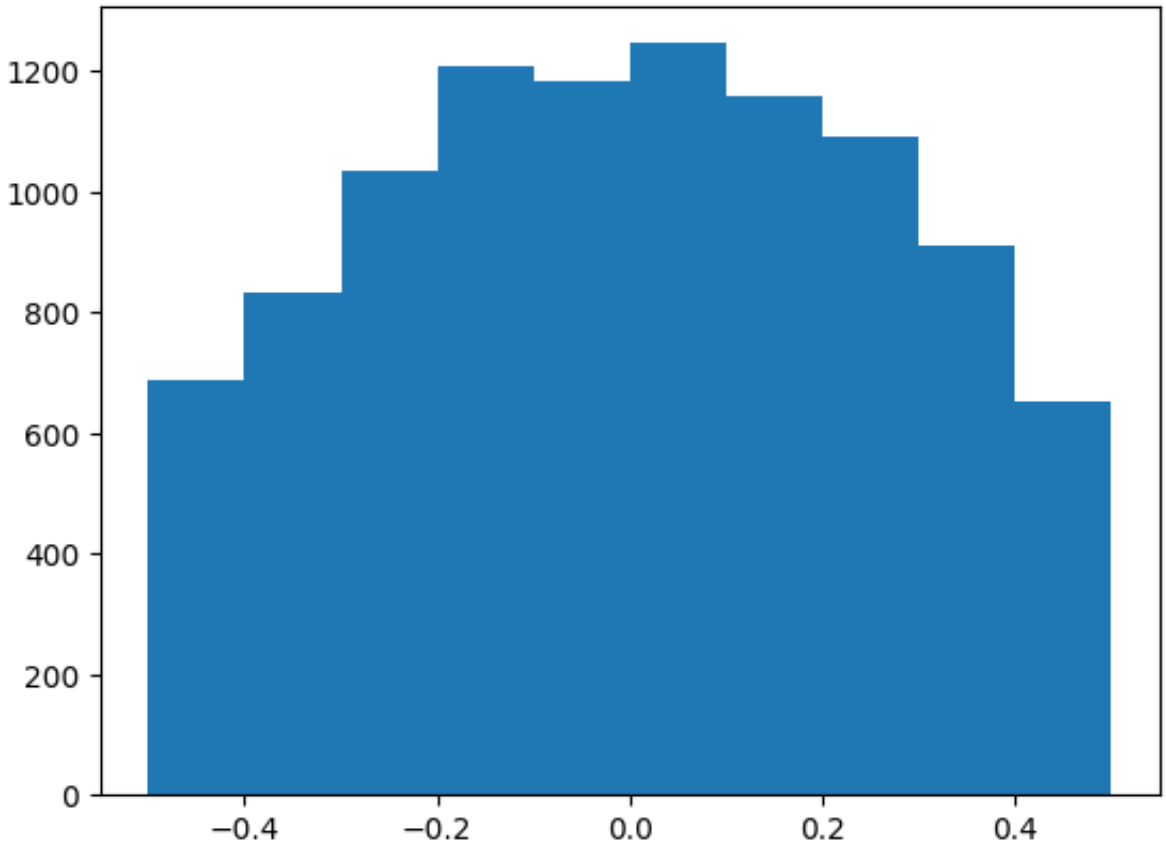
```
a, b = (myclip_a - my_mean) / my_std, (myclip_b - my_mean)
/ my_std
```

Entrée [6]:
```
from scipy.stats import truncnorm

s = truncnorm(a=-2/3., b=2/3., scale=1, loc=0).rvs(size=1000)

plt.hist(s)
plt.show()
```



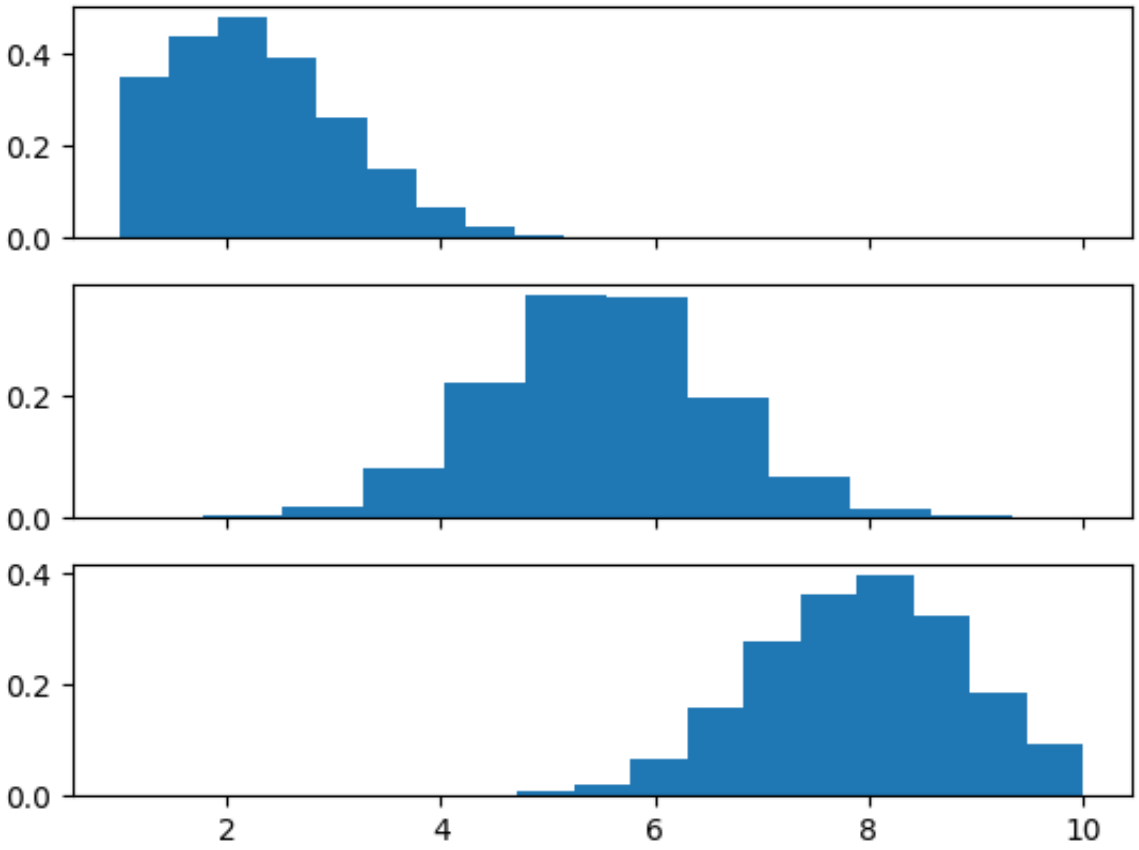
La fonction 'truncnorm' est difficile à utiliser. Pour nous faciliter la vie, nous définissons une fonction truncated_normal dans la suite pour faciliter cette tâche :

Entrée [7]:
```
def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

X = truncated_normal(mean=0, sd=0.4, low=-0.5, upp=0.5)
s = X.rvs(10000)

plt.hist(s)
plt.show()
```



Autres exemples :

Entrée [8]:
```
X1 = truncated_normal(mean=2, sd=1, low=1, upp=10)
X2 = truncated_normal(mean=5.5, sd=1, low=1, upp=10)
X3 = truncated_normal(mean=8, sd=1, low=1, upp=10)

import matplotlib.pyplot as plt
fig, ax = plt.subplots(3, sharex=True)
ax[0].hist(X1.rvs(10000), density=True)
ax[1].hist(X2.rvs(10000), density=True)
ax[2].hist(X3.rvs(10000), density=True)
plt.show()
```



Nous allons maintenant créer la matrice des poids des liens. `truncated_normal` est idéal à cet effet. C'est une bonne idée de choisir des valeurs aléatoires à l'intérieur de l'intervalle.

$$(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}})$$

où n désigne le nombre de nœuds d'entrée.

Nous pouvons donc créer notre matrice "wih" avec :

Entrée [9]:
```
no_of_input_nodes = 3
no_of_hidden_nodes = 4
rad = 1 / np.sqrt(no_of_input_nodes)

X = truncated_normal(mean=2, sd=1, low=-rad, upp=rad)
wih = X.rvs((no_of_hidden_nodes, no_of_input_nodes))
wih
```

Out[9]:
```
array([[ 0.54832644,  0.32498369,  0.41544843],
       [ 0.53493027, -0.57658063, -0.40036726],
       [-0.52154612,  0.57051283,  0.07832974],
       [ 0.36566932,  0.46409181,  0.29629894]])
```

De même, nous pouvons maintenant définir la matrice de poids "qui" :

Entrée [10]:
```
no_of_hidden_nodes = 4
no_of_output_nodes = 2
rad = 1 / np.sqrt(no_of_hidden_nodes)  # this is the input in this

X = truncated_normal(mean=2, sd=1, low=-rad, upp=rad)
who = X.rvs((no_of_output_nodes, no_of_hidden_nodes))
who
```

Out[10]:
```
array([[ 0.33411038, -0.08018864,  0.19919053, -0.40061747],
       [ 0.34498795,  0.33822084, -0.11220133,  0.16798007]])
```

Entrée [ ]: