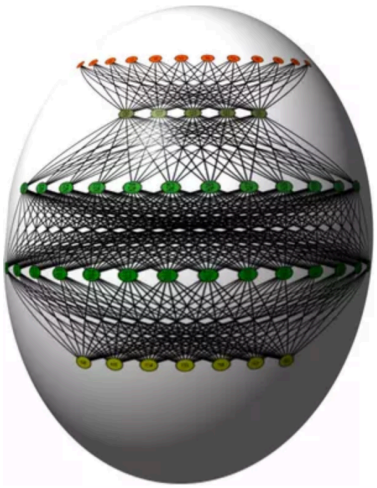
12 - Un réseau de neurones simple à partir de zéro en Python

Ensembles de données linéairement séparables



Comme nous l'avons montré dans le chapitre précédent de notre tutoriel sur l'apprentissage automatique, un réseau neuronal composé d'un seul perceptron a suffi à séparer les classes de notre exemple. Bien entendu, nous avons soigneusement conçu ces classes pour que cela fonctionne. Il existe de nombreux groupes de classes pour lesquels cela ne fonctionnera pas. Nous allons examiner d'autres exemples et discuter des cas où il ne sera pas possible de séparer les classes.

Nos classes ont été linéairement séparables. La séparabilité linéaire a un sens en géométrie euclidienne. Deux ensembles de points (ou classes) sont appelés linéairement séparables, si au moins une ligne droite dans le plan existe de sorte que tous les points d'une classe sont d'un côté de la ligne et tous les points de l'autre classe sont de l'autre côté.

Plus formellement :

Si deux clusters (classes) de données peuvent être séparés par une frontière de décision sous la forme d'une équation linéaire:

$$\sum_{i=1}^{n} x_i \cdot \omega_1 = 0$$

elles sont dites linéairement séparables.

Dans le cas contraire, c'est-à-dire si une telle frontière de décision n'existe pas, les deux classes sont dites linéairement inséparables. Dans ce cas, on ne peut pas utiliser un simple réseau de neurones.

Perceptron pour la fonction AND

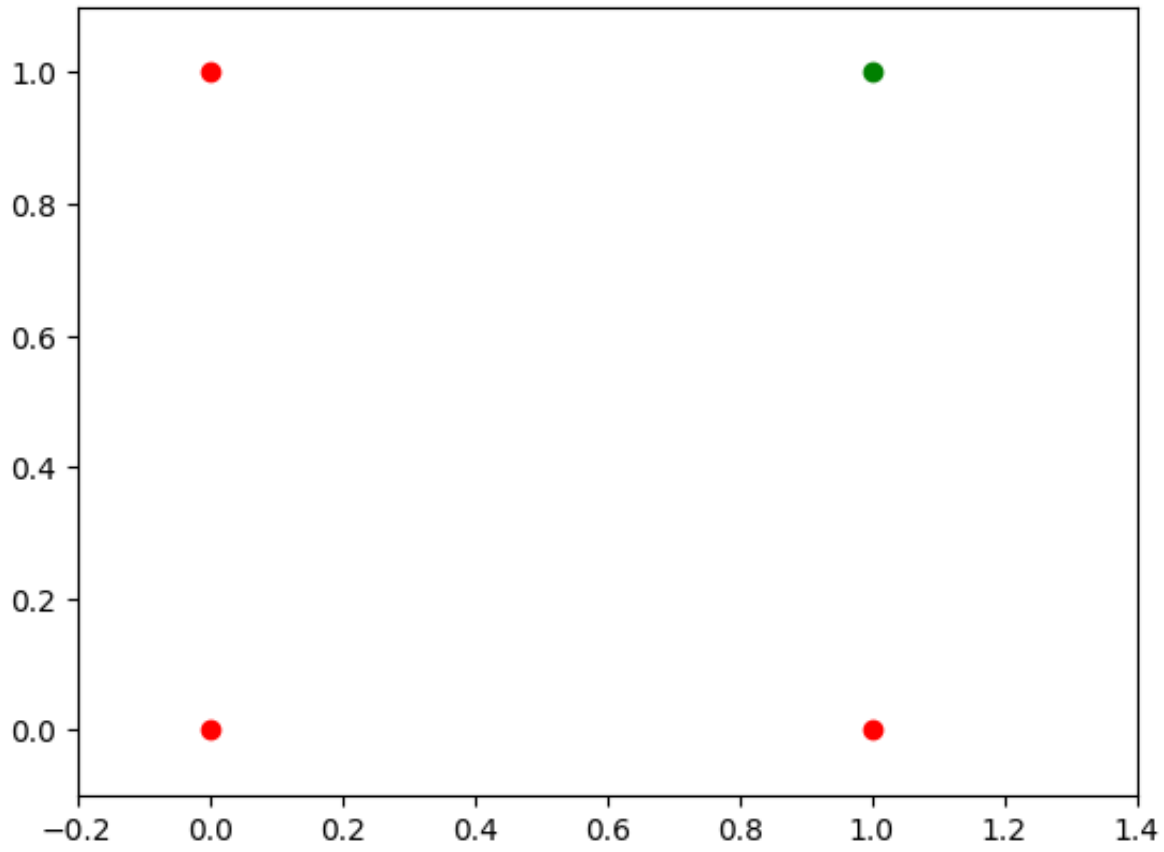
Dans notre prochain exemple, nous allons programmer un réseau de neurones en Python qui implémente la fonction logique "Et". Elle est définie pour deux entrées de la manière suivante :

Input1	Input2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Nous avons appris dans le chapitre précédent qu'un réseau neuronal avec un perceptron et deux valeurs d'entrée peut être interprété comme une frontière de décision, c'est-à-dire une ligne droite divisant deux classes. Les deux classes que nous voulons classer dans notre exemple ressemblent à ceci :

Entrée [1]:
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m = -1
#ax.plot(X, m * X + 1.2, label="decision boundary")
plt.plot()
```

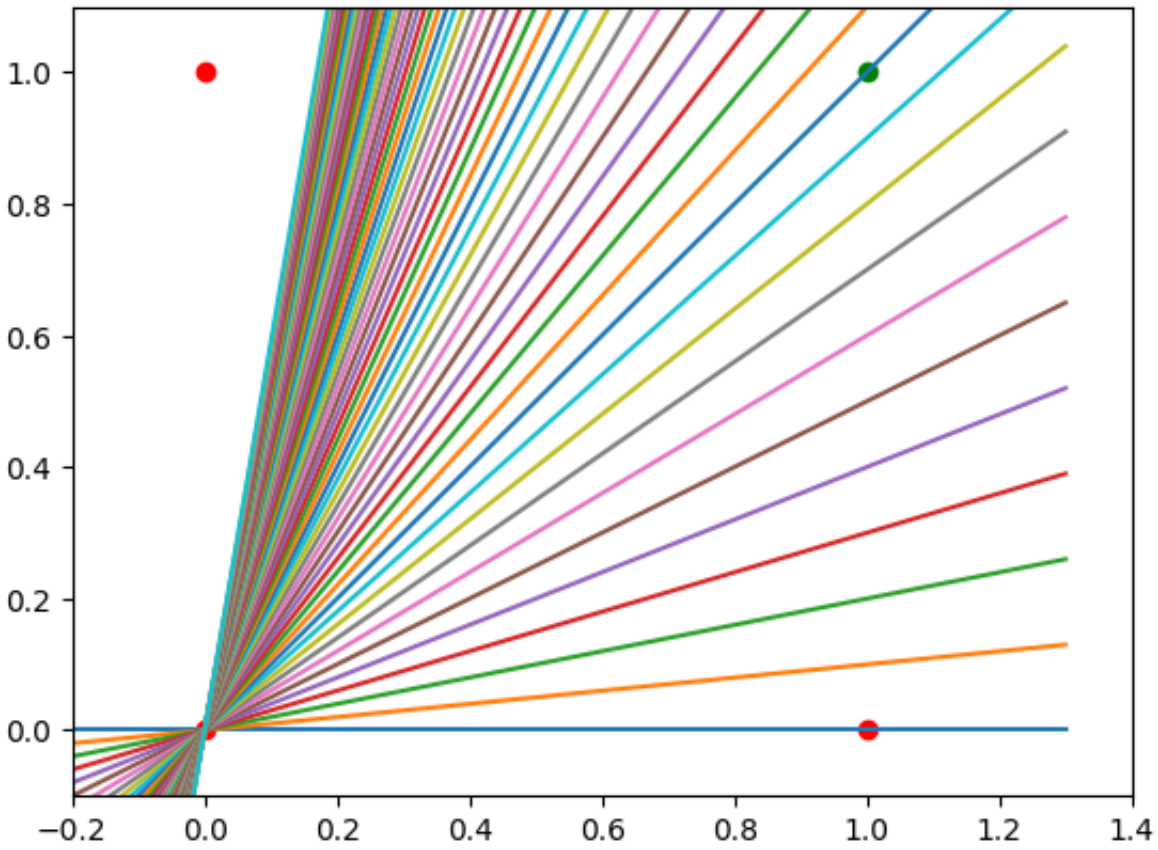
Out[1]: []



Nous avons également découvert qu'un réseau neuronal aussi primitif n'est capable de créer que des lignes droites passant par l'origine. Donc des lignes de division comme celle-ci :

Entrée [2]:
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m = -1
for m in np.arange(0, 6, 0.1):
    ax.plot(X, m * X )
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
plt.plot()
```

Out[2]: []



Nous pouvons voir qu'aucune de ces lignes droites ne peut être utilisée comme limite de décision, ni aucune autre ligne passant par l'origine.

Nous avons besoin d'une ligne

$$y = m \cdot x + c$$

où l'ordonnée à l'origine c n'est pas égale à 0.

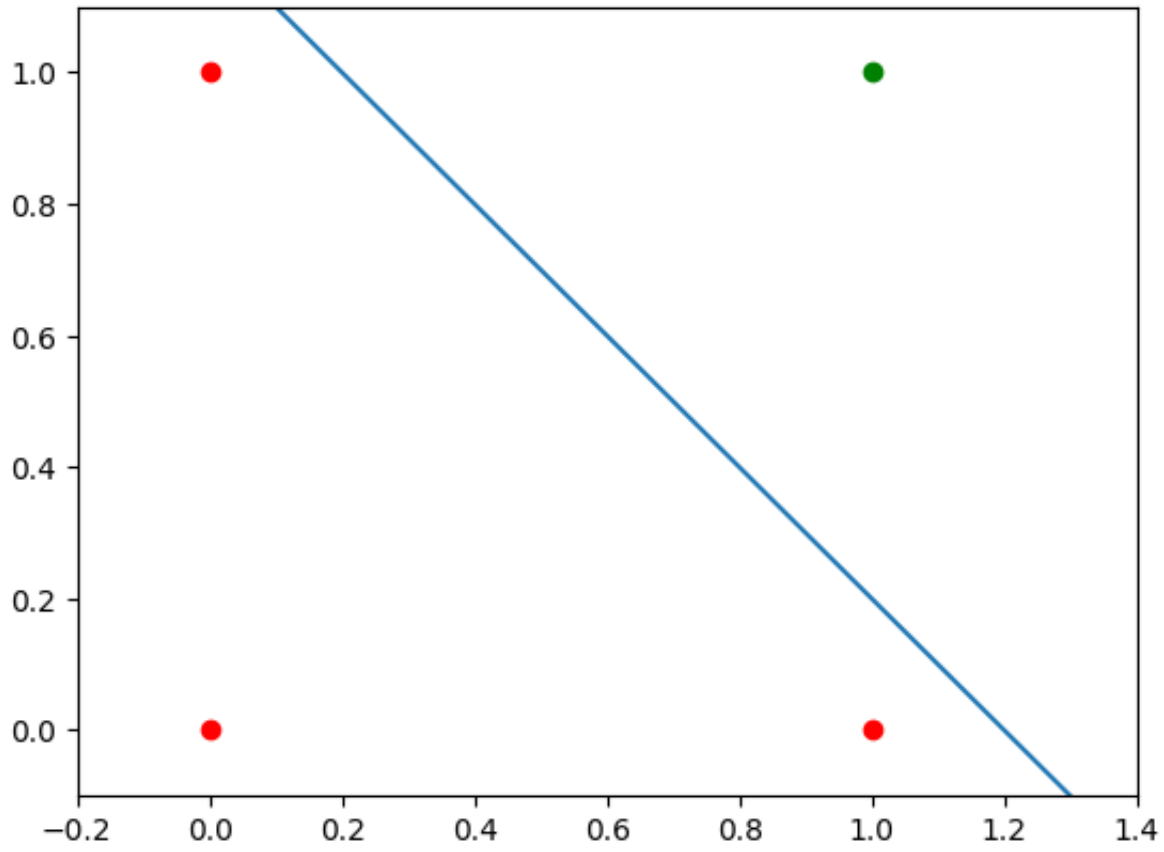
Par exemple, la droite

$$y = -x + 1,2$$

pourrait être utilisé comme ligne de séparation pour notre problème :

Entrée [3]:
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m, c = -1, 1.2
ax.plot(X, m * X + c )
plt.plot()
```

Out[3]: []

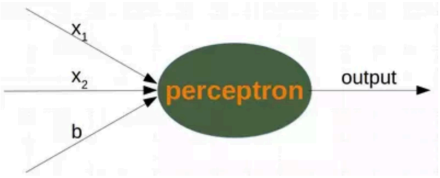


La question est maintenant de savoir si nous pouvons trouver une solution avec des modifications mineures de notre modèle de réseau ? Ou en d'autres termes : Pouvons-nous créer un perceptron capable de définir des frontières de décision arbitraires ?

La solution consiste en l'ajout d'un nœud de biais.

Perceptron simple avec un biais

Un perceptron avec deux valeurs d'entrée et un biais correspond à une ligne droite générale. À l'aide de la valeur de biais b, nous pouvons entraîner le perceptron à déterminer une frontière de décision avec une interception non nulle c.



Alors que les valeurs d'entrée peuvent changer, une valeur de biais reste toujours constante. Seul le poids du nœud de biais peut être adapté.

Maintenant, l'équation linéaire d'un perceptron contient un biais :

$$\sum_{i=1}^{n} \omega_i \, x_i + \omega_{n+1} \cdot b = 0$$

Dans notre cas, cela ressemble à ceci :

$$\omega_1 \cdot x_1 + \omega_2 \cdot x_2 + \omega_3 \cdot b = 0$$

ceci est équivalent à

$$x_2 = -\frac{\omega_1}{\omega_2} \cdot x_1 - \frac{\omega_3}{\omega_2} \cdot b$$

Cela signifie :

$$m = -\frac{\omega_1}{\omega_2}$$

et

$$c = -\frac{\omega_3}{\omega_2} \cdot b$$

Entrée [4]:
```
%%capture
%%writefile perceptrons.py

import numpy as np
from collections import Counter

class Perceptron:
```

```
class Perceptron:

    def __init__(self,
                 weights,
                 bias=1,
                 learning_rate=0.3):
        """
        'weights' can be a numpy array, list or a tuple with the
        actual values of the weights. The number of input values
        is indirectly defined by the length of 'weights'
        """
        self.weights = np.array(weights)
        self.bias = bias
        self.learning_rate = learning_rate

    @staticmethod
    def unit_step_function(x):
        if  x <= 0:
            return 0
        else:
            return 1

    def __call__(self, in_data):
        in_data = np.concatenate( (in_data, [self.bias]) )
        result = self.weights @ in_data
        return Perceptron.unit_step_function(result)

    def adjust(self,
               target_result,
               in_data):
        if type(in_data) != np.ndarray:
            in_data = np.array(in_data)  #
        calculated_result = self(in_data)
        error = target_result - calculated_result
        if error != 0:
            in_data = np.concatenate( (in_data, [self.bias]) )
            correction = error * in_data * self.learning_rate
            self.weights += correction

    def evaluate(self, data, labels):
        evaluation = Counter()
        for sample, label in zip(data, labels):
            result = self(sample) # predict
            if result == label:
                evaluation["correct"] += 1
            else:
                evaluation["wrong"] += 1
        return evaluation
```

Nous supposons que le code Python ci-dessus avec la classe Perceptron est stocké dans votre répertoire de travail actuel sous le nom `perceptrons.py` .

Entrée [5]:

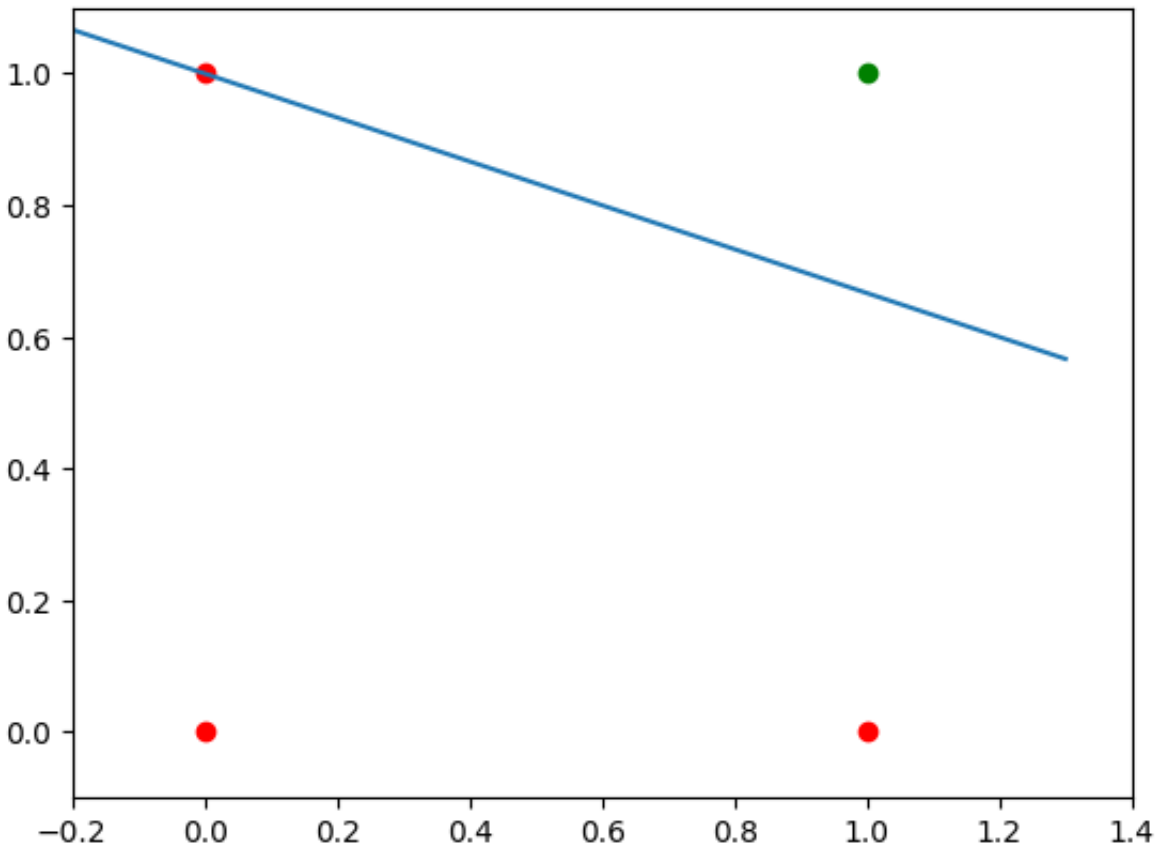
```
import numpy as np
from perceptrons import Perceptron

def labelled_samples(n):
    for _ in range(n):
        s = np.random.randint(0, 2, (2,))
        yield (s, 1) if s[0] == 1 and s[1] == 1 else (s, 0)

p = Perceptron(weights=[0.3, 0.3, 0.3],
               learning_rate=0.2)

for in_data, label in labelled_samples(30):
    p.adjust(label,
             in_data)

test_data, test_labels = list(zip(*labelled_samples(30)))

evaluation = p.evaluate(test_data, test_labels)
print(evaluation)
```

```
Counter({'correct': 30})
```

Entrée [6]:
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m = -p.weights[0] / p.weights[1]
c = -p.weights[2] / p.weights[1]
print(m, c)
ax.plot(X, m * X + c )
plt.plot()
```

```
-0.33333333333333326 1.0000000000000002
```

Out[6]:
```
[]
```

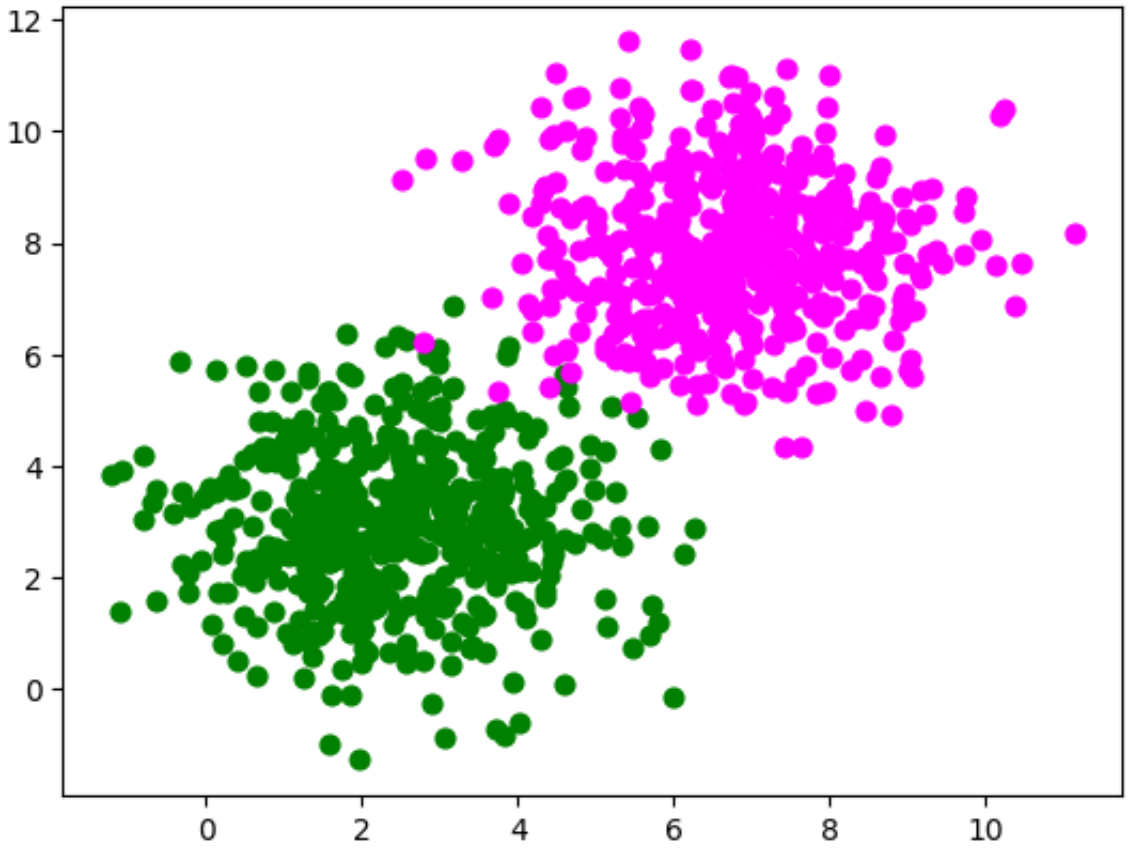


Nous allons créer un autre exemple avec des ensembles de données linéairement séparables, qui nécessitent un nœud de biais pour être séparables. Nous allons utiliser la fonction `make_blobs` de `sklearn.datasets` :

Entrée [7]:
```
from sklearn.datasets import make_blobs

n_samples = 1000
samples, labels = make_blobs(n_samples=n_samples,
                             centers=([2.5, 3], [6.7, 7.9]),
                             cluster_std=1.4,
                             random_state=0)
```

Visualisons les données précédemment créées :

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

colours = ('green', 'magenta', 'blue', 'cyan', 'yellow', 'red')
fig, ax = plt.subplots()


for n_class in range(2):
    ax.scatter(samples[labels==n_class][:, 0], samples[labels==n_cl
               c=colours[n_class], s=40, label=str(n_class))
```



Entrée [9]:
```
from sklearn.model_selection import train_test_split
res = train_test_split(samples, labels,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=1)

train_data, test_data, train_labels, test_labels = res
from perceptrons import Perceptron

p = Perceptron(weights=[0.3, 0.3, 0.3],
               learning_rate=0.8)

for sample, label in zip(train_data, train_labels):
    p.adjust(label,
             sample)

evaluation = p.evaluate(train_data, train_labels)
print(evaluation)
```

```
Counter({'correct': 784, 'wrong': 16})
```

Entrée [10]:
```
evaluation = p.evaluate(test_data, test_labels)
print(evaluation)
```

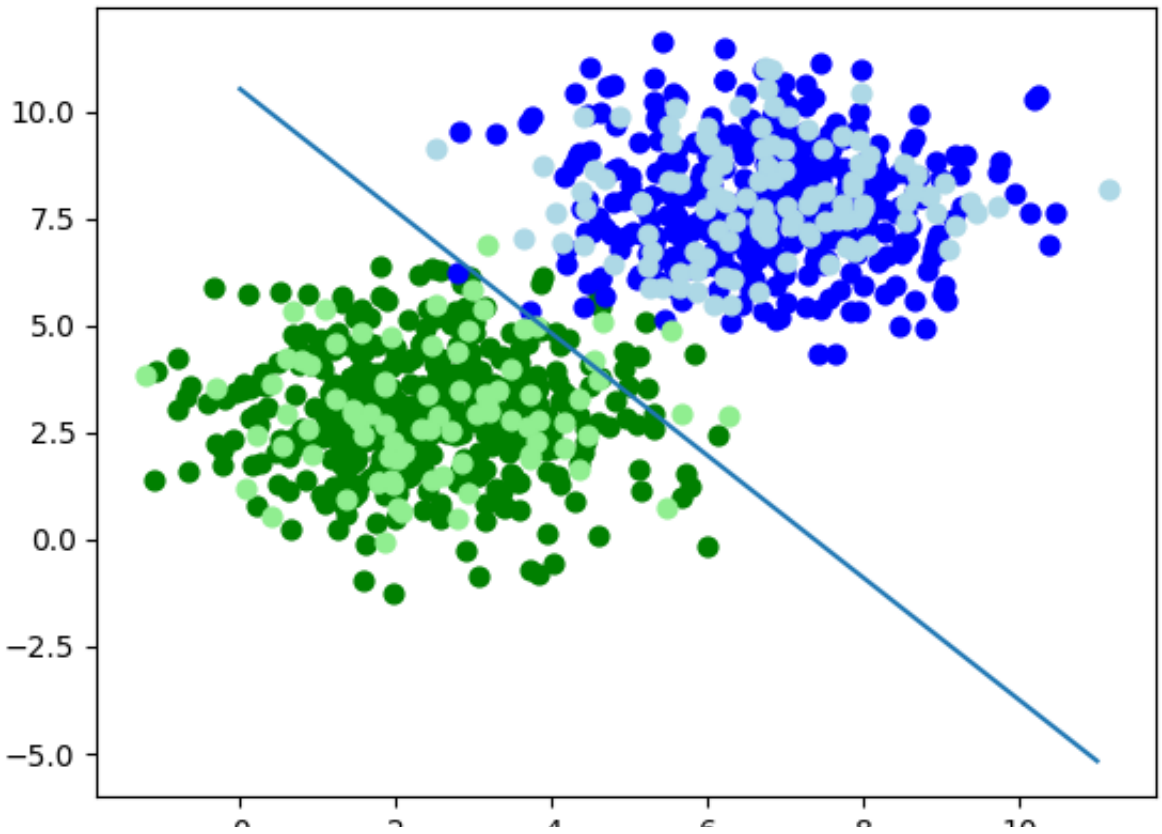
```
Counter({'correct': 194, 'wrong': 6})
```

Visualisons la frontière de décision :

Entrée [11]:

```
import matplotlib.pyplot as plt


fig, ax = plt.subplots()

# plotting learn data
colours = ('green', 'blue')
for n_class in range(2):
    ax.scatter(train_data[train_labels==n_class][:, 0],
               train_data[train_labels==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))

# plotting test data
colours = ('lightgreen', 'lightblue')
for n_class in range(2):
    ax.scatter(test_data[test_labels==n_class][:, 0],
               test_data[test_labels==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))



X = np.arange(np.max(samples[:,0]))
m = -p.weights[0] / p.weights[1]
c = -p.weights[2] / p.weights[1]
print(m, c)
ax.plot(X, m * X + c )
plt.plot()
plt.show()
```

```
-1.4277135509226737 10.516023065099064
```



0 2 4 6 8 10

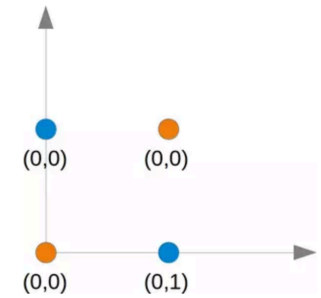
Dans la section suivante, nous allons présenter le problème XOR pour les réseaux neuronaux. Il s'agit de l'exemple le plus simple de réseau neuronal non linéairement séparable. Il peut être résolu à l'aide d'une couche supplémentaire de neurones, appelée couche cachée.

Le problème XOR pour les réseaux neuronaux

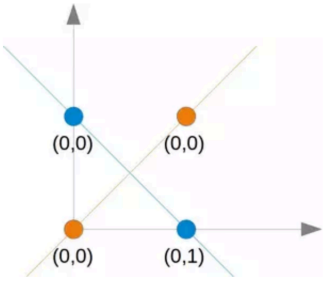
La fonction XOR (ou exclusif) est définie par la table de vérité suivante :

Input1	Input2	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

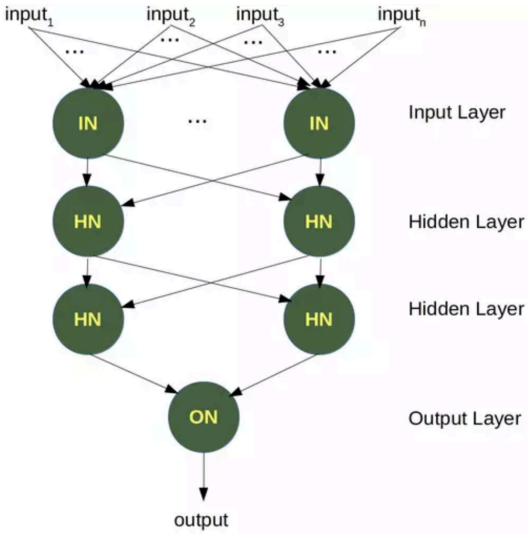
Ce problème ne peut pas être résolu avec un simple réseau de neurones, comme on peut le voir dans le diagramme suivant :



Quelle que soit la ligne droite que vous choisissez, vous ne réussirez jamais à avoir les points bleus d'un côté et les points orange de l'autre. C'est ce que montre la figure suivante. Les points orange sont sur la ligne orange. Cela signifie qu'il ne peut s'agir d'une ligne de séparation. Si nous déplaçons cette ligne parallèlement - quelle que soit la direction - il y aura toujours deux points orange et un point bleu d'un côté et un seul point bleu de l'autre côté. Si nous déplaçons la ligne orange de manière non parallèle, il y aura un point bleu et un point orange de chaque côté, sauf si la ligne passe par un point orange. Il est donc impossible qu'une seule ligne droite sépare ces points.

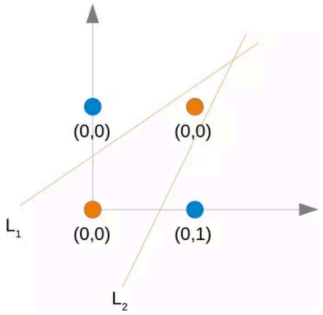


Pour résoudre ce problème, nous devons introduire un nouveau type de réseaux neuronaux, un réseau avec des couches dites cachées. Une couche cachée permet au réseau de réorganiser ou de réarranger les données d'entrée.

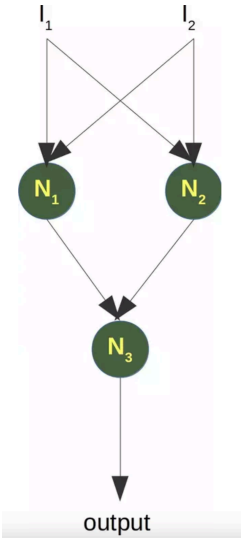


Nous n'aurons besoin que d'une couche cachée avec deux neurones. L'un fonctionne comme une porte ET et l'autre comme une porte OU. La sortie se déclenchera lorsque la porte OU se déclenchera et que la porte ET ne se déclenchera pas.

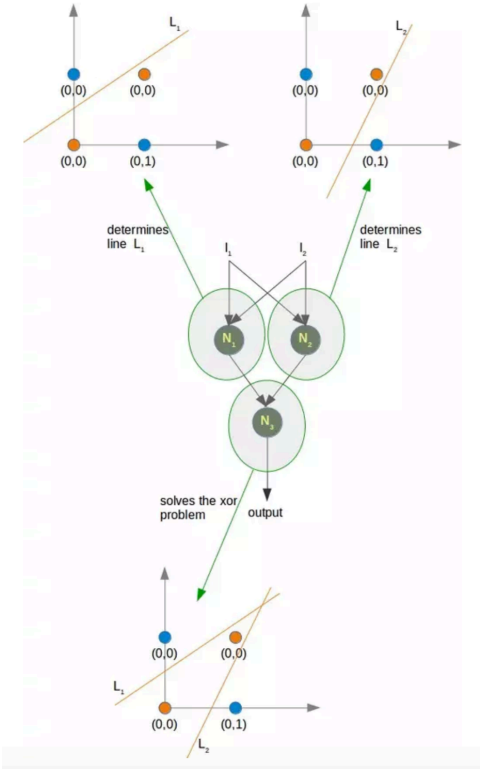
Comme nous l'avons déjà mentionné, nous ne pouvons pas trouver une ligne qui sépare les points orange des points bleus. Mais ils peuvent être séparés par deux lignes, par exemple L1 et L2 dans le schéma suivant :



Pour résoudre ce problème, nous avons besoin d'un réseau du type suivant, c'est-à-dire avec une couche cachée N1 et N2



Le neurone N1 va déterminer une ligne, par exemple L1 et le neurone N2 va déterminer l'autre ligne L2. N3 résoudra finalement notre problème :



L'implémentation de cette méthode en Python devra attendre le prochain chapitre de notre tutoriel sur l'apprentissage automatique.

Exercices

Exercice 1

Nous pourrions étendre le ET logique aux valeurs flottantes entre 0 et 1 de la manière suivante :

Input1	Input2	Output
x1 < 0.5	x2 < 0.5	0
x1 < 0.5	x2 >= 0.5	0
x1 >= 0.5	x2 < 0.5	0
x1 >= 0.5	x2 >= 0.5	1

Essayez de former un réseau neuronal avec un seul perceptron. Pourquoi cela ne fonctionne-t-il pas ?

Exercice 2

Un point appartient à une classe 0, si $x_1 < 0.5$ et appartient à la classe 1, si $x_1 \geq 0.5$. Entraînez un réseau avec un perceptron pour classer des points arbitraires. Que pouvez-vous dire de la limite de décision ? Qu'en est-il des valeurs d'entrée x_2

Solutions aux exercices

Solution du 1er exercice

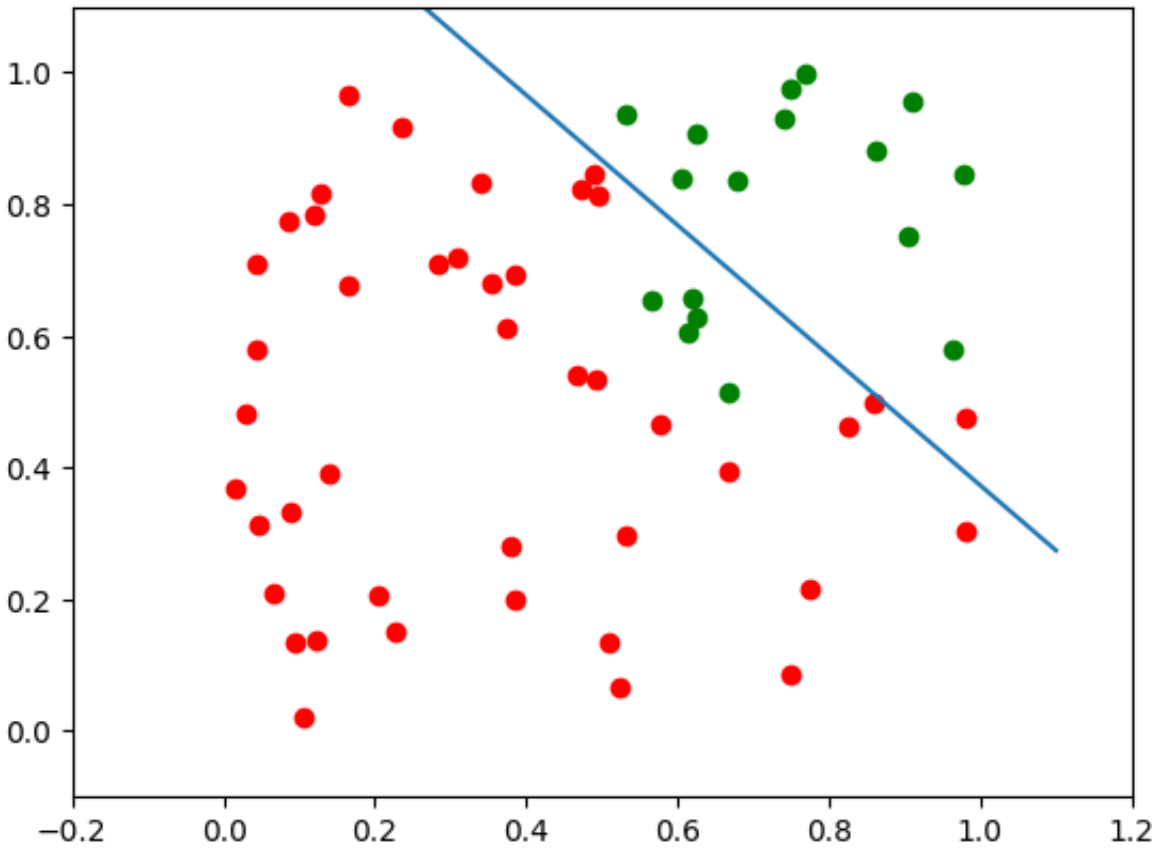
Entrée [12]:
```
from perceptrons import Perceptron

p = Perceptron(weights=[0.3, 0.3, 0.3],
               bias=1,
               learning_rate=0.2)

def labelled_samples(n):
    for _ in range(n):
        s = np.random.random((2,))
        yield (s, 1) if s[0] >= 0.5 and s[1] >= 0.5 else (s, 0)

for in_data, label in labelled_samples(30):
    p.adjust(label,
             in_data)

test_data, test_labels = list(zip(*labelled_samples(60)))

evaluation = p.evaluate(test_data, test_labels)
print(evaluation)
```

```
Counter({'correct': 54, 'wrong': 6})
```

Le moyen le plus simple de voir pourquoi cela ne fonctionne pas est de visualiser les données.

Entrée [13]:
```
import matplotlib.pyplot as plt
import numpy as np

ones = [test_data[i] for i in range(len(test_data)) if test_labels[
zeroes = [test_data[i] for i in range(len(test_data)) if test_label

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.2
X, Y = list(zip(*ones))
ax.scatter(X, Y, color="g")
X, Y = list(zip(*zeroes))
ax.scatter(X, Y, color="r")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
c = -p.weights[2] / p.weights[1]
m = -p.weights[0] / p.weights[1]
X = np.arange(xmin, xmax, 0.1)
ax.plot(X, m * X + c, label="decision boundary")
```

Out[13]:
```
[<matplotlib.lines.Line2D at 0x7f98c005f2e0>]
```



On voit que les points verts et les points rouges ne sont pas séparables par une seule droite.

Solution du 2ème exercice

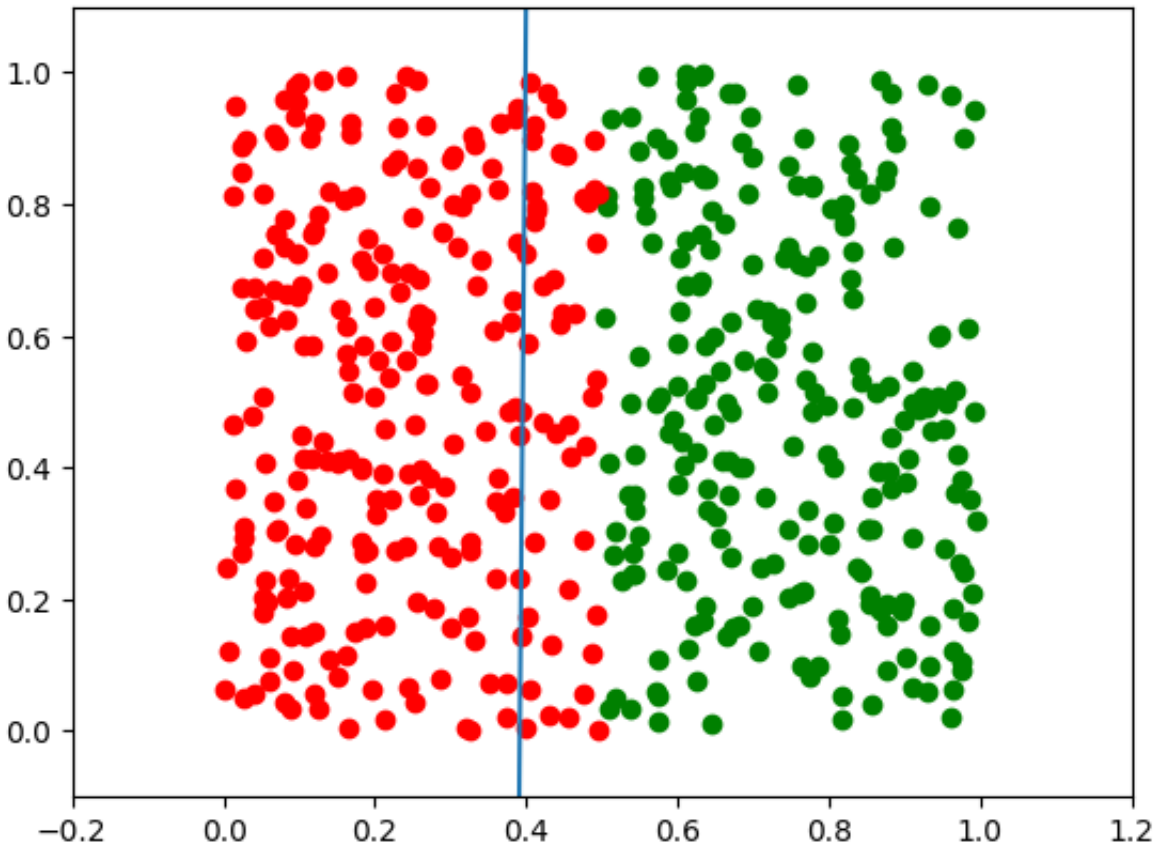
Entrée [14]:
```
from perceptrons import Perceptron

import numpy as np
from collections import Counter

def labelled_samples(n):
    for _ in range(n):
        s = np.random.random((2,))
        yield (s, 0) if s[0] < 0.5 else (s, 1)


p = Perceptron(weights=[0.3, 0.3, 0.3],
               learning_rate=0.4)

for in_data, label in labelled_samples(300):
    p.adjust(label,
             in_data)

test_data, test_labels = list(zip(*labelled_samples(500)))

print(p.weights)
p.evaluate(test_data, test_labels)
```

```
[ 2.2988437 -0.01718   -0.9      ]
```

Out[14]:
```
Counter({'correct': 455, 'wrong': 45})
```

Entrée [15]:
```
import matplotlib.pyplot as plt
import numpy as np

ones = [test_data[i] for i in range(len(test_data)) if test_labels[
zeroes = [test_data[i] for i in range(len(test_data)) if test_label

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.2
X, Y = list(zip(*ones))
ax.scatter(X, Y, color="g")
X, Y = list(zip(*zeroes))
ax.scatter(X, Y, color="r")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
c = -p.weights[2] / p.weights[1]
m = -p.weights[0] / p.weights[1]
X = np.arange(xmin, xmax, 0.1)
ax.plot(X, m * X + c, label="decision boundary")
```

Out[15]:
```
[<matplotlib.lines.Line2D at 0x7f9910379570>]
```



Entrée [16]:
```
p.weights, m
```

Out[16]:
```
(array([ 2.2988437, -0.01718  , -0.9      ]), 133.80930682863217)
```

La pente m devra être de plus en plus grande dans des situations comme celle-ci.

Entrée [ ]: