

IA Embarquée

Introduction 2

P. Bakowski



Intelligence Artificielle (IA) et IA Embarquée

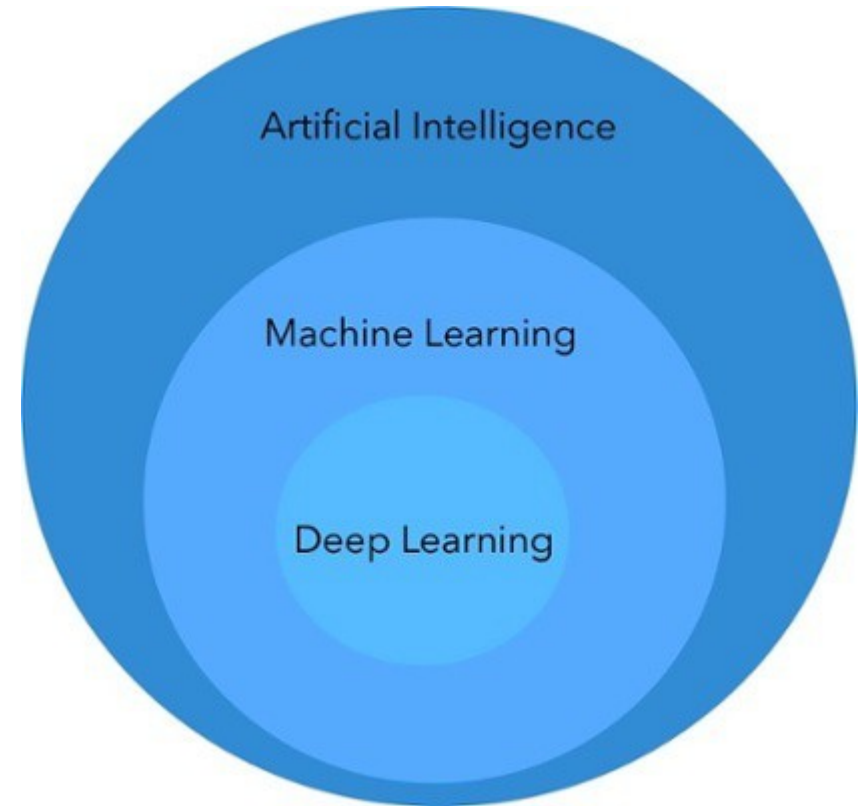
- Introduction
- Matériel pour l'IA Embarquée
- Logiciel pour l'IA Embarquée
- **Lab 0 – Introduction au Réseaux Neuronaux**
- Lab 1 – CPU-NEON, Inférence sur RPI 4
- Lab 2 – GPU – Jetson Nano, Inférence
- Lab 3 – GPU – Jetson Nano, Reconnaissance faciale
- Lab 4 – GPU – Jetson Nano, Voix
- Lab 5 – TPU – Google CORAL, Inférence



Introduction – IA, ML et DL

■ Le **Machine Learning** est une technologie d'intelligence artificielle permettant aux ordinateurs d'apprendre **sans avoir été programmés** explicitement à cet effet.

■ Pour apprendre et se développer, les ordinateurs ont toutefois **besoin de données** à analyser et sur lesquelles ils doivent **s'entraîner**.





Introduction – IA, ML et DL

- L'**apprentissage profond** est un ensemble de méthodes d'apprentissage automatique tentant de modéliser avec un haut niveau d'abstraction des données grâce à des architectures articulées de différentes **transformations non linéaires**.
- Ces techniques ont permis des progrès importants et rapides dans les domaines de l'analyse du **signal sonore ou visuel** et notamment de la **reconnaissance faciale**, de la **reconnaissance vocale**, de la **vision par ordinateur**, du traitement automatisé du langage.



Introduction – IA Embarquée

■ Grâce à notre capacité à construire des machines intelligentes qui simulent l'intelligence humaine, les implications pour le progrès technologique dans de nombreux secteurs sont infinies. Alors, **quoi de mieux que l'intelligence artificielle ?**

■ **Intelligence Artificielle Embarquée.**

■ L'intelligence artificielle (IA) embarquée est l'application de **ML** et **DL** au **niveau de l'appareil** (*embedded device*)

■ L'Appareil ou *embedded device* est un **SoC** ou un **FPGA** intégré sur une carte autonome – **SCB** (*Single Computer Board*)

■ Exemples : Nvidia -Jetson Nano, Xavier, Google - CORAL, ..

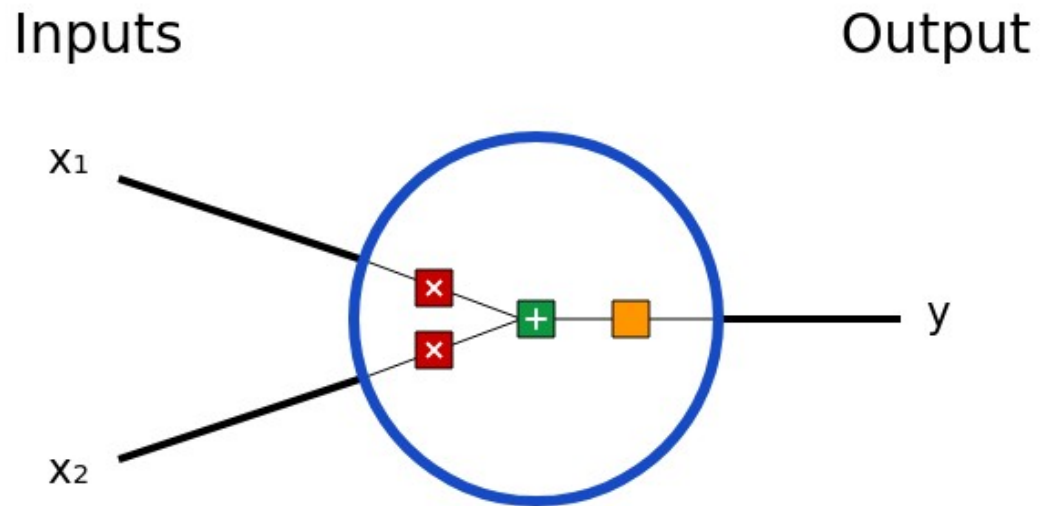


Introduction – Logiciel

- Les logiciels fonctionnant sur les équipements IA embarqués sont (presque) les mêmes que ceux sur les équipements standards (PC, serveurs,..)
- **Langages** : C/C++ et Python
- **Packages** : PyNum, Ski-Learn, ..
- **Couches applicatives** : Keras, TensorFlow, PyTorch,..
TensorFlow => TensorFlow Lite
- **Couches d'exécution** : CUDA, cuDNN, ..
- **OS** : Linux, FreeRTOS, ..

Introduction – Un Neurone

■ Un **Neurone** est un mécanisme essentiel dans le développement des applications d'IA. Voici sa structure fonctionnelle :




■
$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$

■ où : **w1** et **w2** sont des **poids** (*weights*) et **b** est un **biais**




Introduction – Un Neurone


$$y = f(x1*w1+x2*w2+b)$$



3 choses se passent ici :




Tout d'abord, chaque entrée (**x1** et **x2**) est multipliée par un poids **w1** et **w2**:


$$x1*w1 \text{ et } x2*w2$$




Ensuite, toutes les **entrées pondérées** sont additionnées avec un **biais b** :


$$(x1*w1)+(x2*w2)+b$$



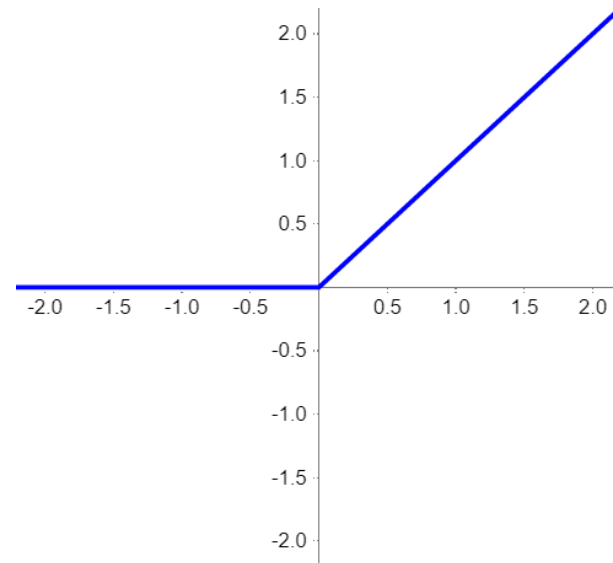
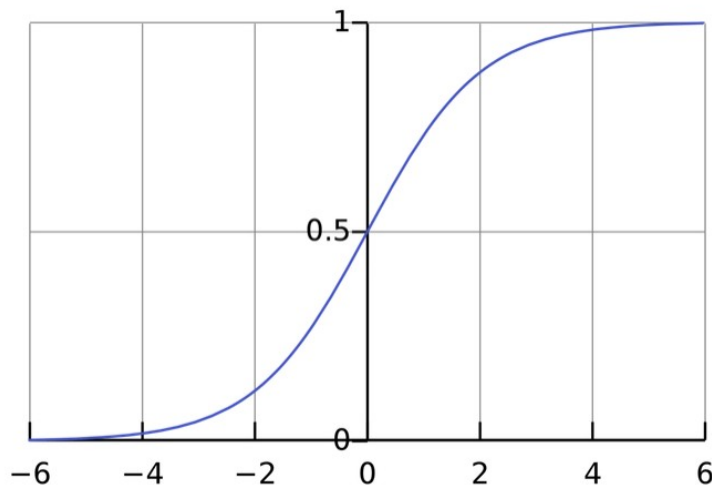
Enfin, la somme passe par une **fonction d'activation** :


$$y=f(x1*w1+x2*w2+b)$$

Introduction – Un Neurone

$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$

- La **fonction d'activation** est utilisée pour transformer une entrée illimitée en une sortie qui a une forme agréable et prévisible.
- Une fonction d'activation couramment utilisée est la fonction **sigmoïde** (ou **ReLU - Rectified Linear Unit**):





Un exemple simple

- Supposons **un neurone à 2 entrées** qui utilise la fonction d'activation **sigmoïde** et possède les paramètres suivants :
- **$w=[0,1]$ et $b=4$ ou $w=[0,1]$ signifie $w_1=0$ et $w_2=1$**
- Donnons au neurone une entrée de **$x=[2,3]$** . Nous utiliserons le **produit scalaire** (\cdot - **dot**) pour écrire les choses de manière plus concise :
- **$(w \cdot x) + b = ((w_1 * x_1) + (w_2 * x_2)) + b = 0 * 2 + 1 * 3 + 4 = \dots$**
- **$y = f(w \cdot x + b) = f(7) = \dots$**
- En sachant que **$f(x)$ est $1/(1-e^{-x})$** calculez vous même le résultat.



Codage d'un neurone

```
import numpy as np

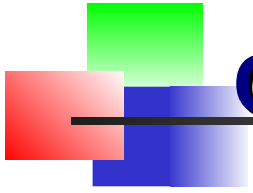
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

class Neuron:
    def __init__(self, weights, bias):
        self.weights = weights
        self.bias = bias

    def feedforward(self, inputs):
        total = np.dot(self.weights, inputs) + self.bias
        return sigmoid(total)

weights = np.array([0, 1]) # w1 = 0, w2 = 1
bias = 4                  # b = 4
n = Neuron(weights, bias)

x = np.array([2, 3])      # x1 = 2, x2 = 3
print(n.feedforward(x))   # 0.9990889488055994
```



Codage d'un neurone

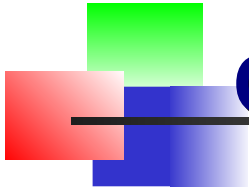
■ **Pour cet exemple** nous utiliserons la plateforme Google Colab

Qu'est-ce que **Colaboratory** ?

Colaboratory, souvent raccourci en "Colab", vous permet d'écrire et d'exécuter du **code Python** dans **votre navigateur**.

Il offre les **avantages** suivants :

1. Aucune configuration requise
2. Accès gratuit aux **GPU/TPU**
3. Partage facile



Codage d'un neurone

Page d'accueil



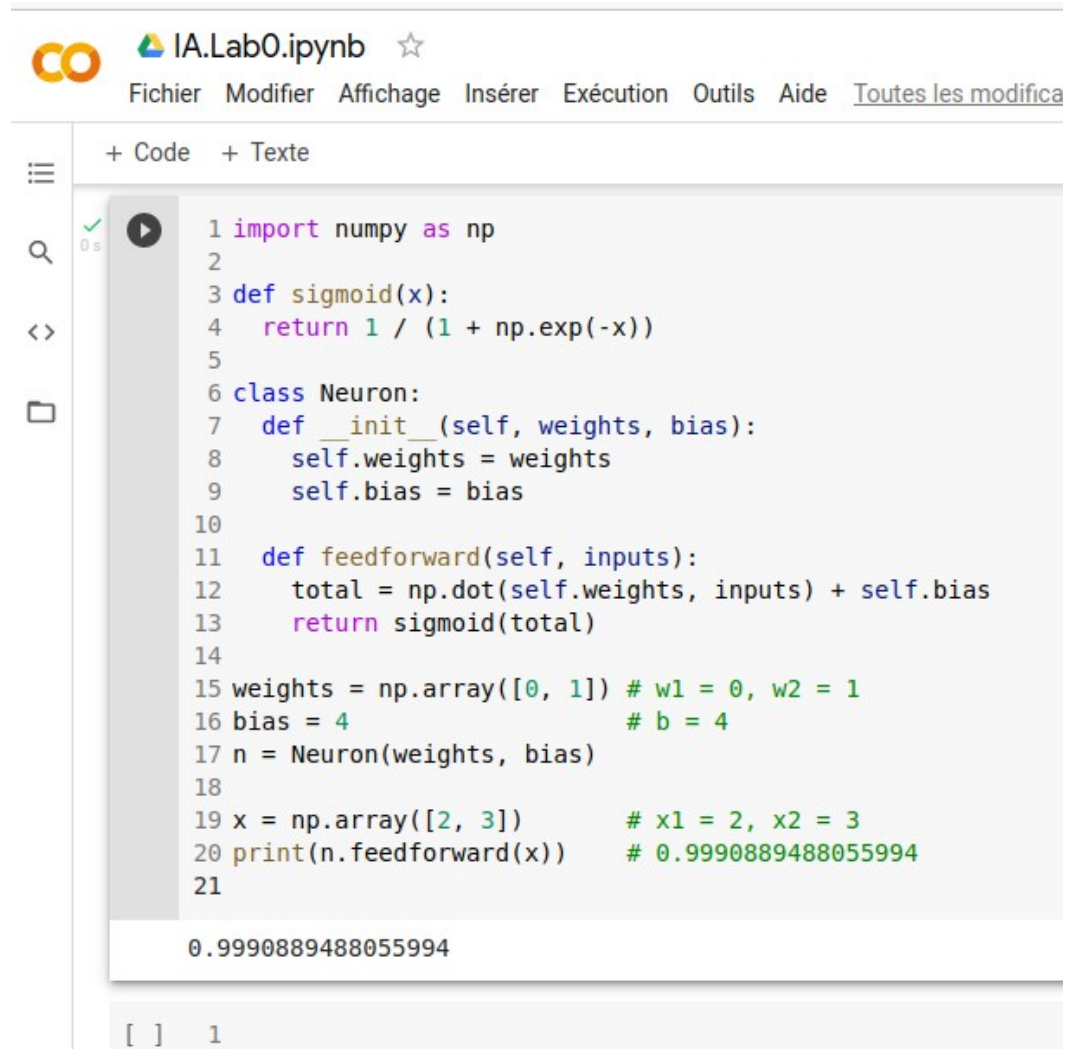
Votre notebook



Codage d'un neurone

Notre premier exemple -
neurone

A vous de référer cet
exemple sur votre compte
Google Colab



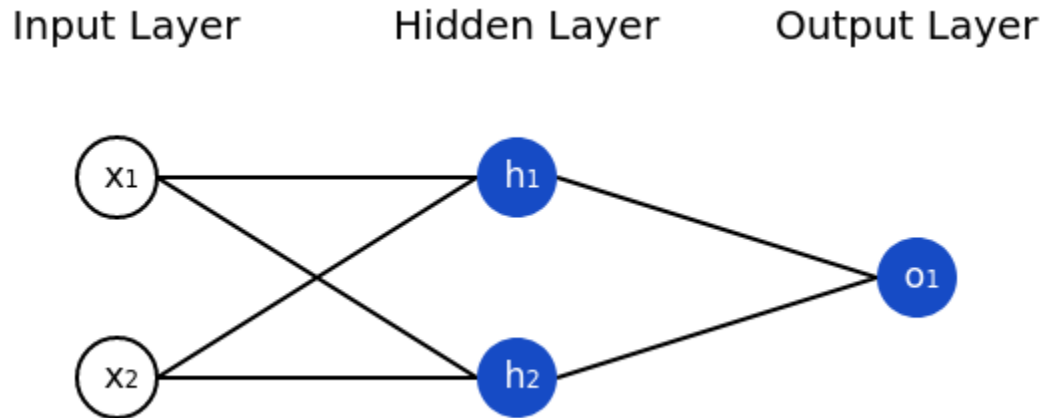
```
1 import numpy as np
2
3 def sigmoid(x):
4     return 1 / (1 + np.exp(-x))
5
6 class Neuron:
7     def __init__(self, weights, bias):
8         self.weights = weights
9         self.bias = bias
10
11     def feedforward(self, inputs):
12         total = np.dot(self.weights, inputs) + self.bias
13         return sigmoid(total)
14
15 weights = np.array([0, 1]) # w1 = 0, w2 = 1
16 bias = 4 # b = 4
17 n = Neuron(weights, bias)
18
19 x = np.array([2, 3]) # x1 = 2, x2 = 3
20 print(n.feedforward(x)) # 0.9990889488055994
21
```

0.9990889488055994

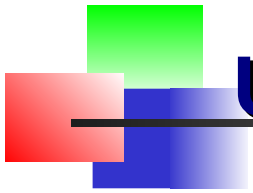
[] 1

Un réseau de neurones (NN)

Un réseau de neurones n'est rien de plus qu'un **groupe de neurones connectés** entre eux. Voici à quoi pourrait ressembler un simple réseau de neurones :



Ce réseau a 2 entrées, une couche cachée avec 2 neurones (**h_1** et **h_2**), et une couche de sortie avec 1 neurone (**o_1**). Notez que les entrées de **o_1** sont les sorties de **h_1** et **h_2** - c'est ce qui en fait un réseau.



Un exemple : Feed-Forward

Utilisons le réseau illustré ci-dessus et supposons que tous les neurones ont les mêmes poids $\mathbf{w}=[0,1]$, le même biais $\mathbf{b}=0$ et la même fonction d'activation sigmoïde. Soit $\mathbf{h1}, \mathbf{h2}, \mathbf{o1}$ les sorties des neurones qu'ils représentent.

Que se passe-t-il si on passe l'entrée $\mathbf{x}=[2,3]$?

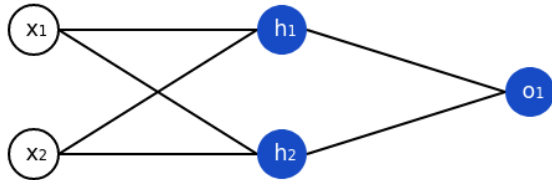
$$\mathbf{h1}=\mathbf{h2}=f(\mathbf{w}\cdot\mathbf{x}+\mathbf{b})=f((0*2)+(1*3)+0)=f(3)=0.9526$$

$$\mathbf{o1}=f(\mathbf{w}\cdot[\mathbf{h1},\mathbf{h2}]+\mathbf{b})=f((0*\mathbf{h1})+(1*\mathbf{h2})+0)=f(0.9526)=0.7216$$

La sortie $\mathbf{o1}$ du réseau neuronal pour l'entrée $\mathbf{x}=[2,3]$ est de **0,72160**.

Codage du réseau : Feed-Forward

Input Layer Hidden Layer Output Layer



IA.Lab0.ipynb

Fichier Modifier Affichage Insérer Exécution Outils Aide [Toutes les modifica](#)

+ Code + Texte

```
1 import numpy as np
2
3 def sigmoid(x):
4     return 1 / (1 + np.exp(-x))
5
6 class Neuron:
7     def __init__(self, weights, bias):
8         self.weights = weights
9         self.bias = bias
10
11     def feedforward(self, inputs):
12         total = np.dot(self.weights, inputs) + self.bias
13         return sigmoid(total)
14
15 weights = np.array([0, 1]) # w1 = 0, w2 = 1
16 bias = 4 # b = 4
17 n = Neuron(weights, bias)
18
19 x = np.array([2, 3]) # x1 = 2, x2 = 3
20 print(n.feedforward(x)) # 0.9990889488055994
21
```

0.9990889488055994

[] 1

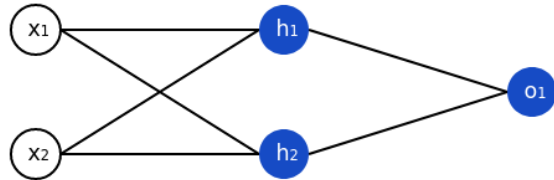
+ Code + Texte

```
1 import numpy as np
2
3 # ... code from previous section here
4
5 class OurNeuralNetwork:
6     '''
7     A neural network with:
8     - 2 inputs
9     - a hidden layer with 2 neurons (h1, h2)
10    - an output layer with 1 neuron (o1)
11    Each neuron has the same weights and bias:
12    - w = [0, 1]
13    - b = 0
14    '''
15    def __init__(self): # no arguments, only internal values
16        weights = np.array([0, 1])
17        bias = 0
18
19    # The Neuron class here is from the previous section
20    self.h1 = Neuron(weights, bias)
21    self.h2 = Neuron(weights, bias)
22    self.o1 = Neuron(weights, bias)
23
24    def feedforward(self, x):
25        out_h1 = self.h1.feedforward(x)
26        out_h2 = self.h2.feedforward(x)
27
28        # The inputs for o1 are the outputs from h1 and h2
29        out_o1 = self.o1.feedforward(np.array([out_h1, out_h2]))
30
31        return out_o1
32
33 network = OurNeuralNetwork()
34 x = np.array([2, 3])
35 print(network.feedforward(x)) # 0.7216325609518421
```

0.7216325609518421

Formation d'un réseau de neurones, partie 1

Input Layer Hidden Layer Output Layer



Disons que nous avons les mesures suivantes :

Name	Weight (lb)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F



Fonction de perte (loss)

Avant de former notre réseau, nous avons d'abord besoin d'un moyen de **quantifier à quel point** il se comporte « bien » afin qu'il puisse essayer de faire « mieux ». C'est ça la **perte**.

Nous utiliserons la perte **erreur quadratique moyenne** (*Mean Square Error - MSE*) :

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_{\text{true}} - y_{\text{pred}})^2$$

n est le **nombre d'échantillons**, qui est de **4** (Alice, Bob, Charlie, Diana).

y représente la **variable prédite**, qui est le **sexe**.

ytrue est la **vraie valeur** de la variable (la "bonne réponse").

Par exemple, **ytruey** pour **Alice** serait **1** (Femme).

ypred est la **valeur prédite** de la variable. C'est quelle que soit la sortie de notre réseau.

(ytrue-ypred)^2 est connu comme l'**erreur quadratique**. Notre fonction de perte prend simplement **la moyenne de toutes les erreurs au carré**.



Un exemple de calcul de perte

Disons que notre réseau affiche (prédit) **toujours 0** - en d'autres termes, il est sûr que tous les humains sont des hommes .

Quelle serait notre perte ?

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_{\text{true}} - y_{\text{pred}})^2$$

Name	y_{true}	y_{pred}	$(y_{\text{true}} - y_{\text{pred}})^2$
Alice	1	0	1
Bob	0	0	0
Charlie	0	0	0
Diana	1	0	1

$$\text{MSE} = \frac{1}{4}(1 + 0 + 0 + 1) = \boxed{0.5}$$



Code : Perte MSE

Voici un code pour calculer la perte pour nous:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_{\text{true}} - y_{\text{pred}})^2$$

```
1 import numpy as np
2
3 def mse_loss(y_true, y_pred):
4     # y_true and y_pred are numpy arrays of the same length.
5     return ((y_true - y_pred) ** 2).mean()
6
7 y_true = np.array([1, 0, 0, 1])
8 y_pred = np.array([0, 0, 0, 0])
9
10 print(mse_loss(y_true, y_pred)) # 0.5
```

0.5



Formation d'un réseau de neurones, partie 2

Notre objectif est de **minimiser la perte du réseau** de neurones.

Nous pouvons **modifier les pondérations et les biais** du réseau pour influencer ses prévisions, mais comment le faire de manière à réduire les pertes ?

Pour simplifier, supposons que nous n'ayons qu'Alice dans notre ensemble de données :

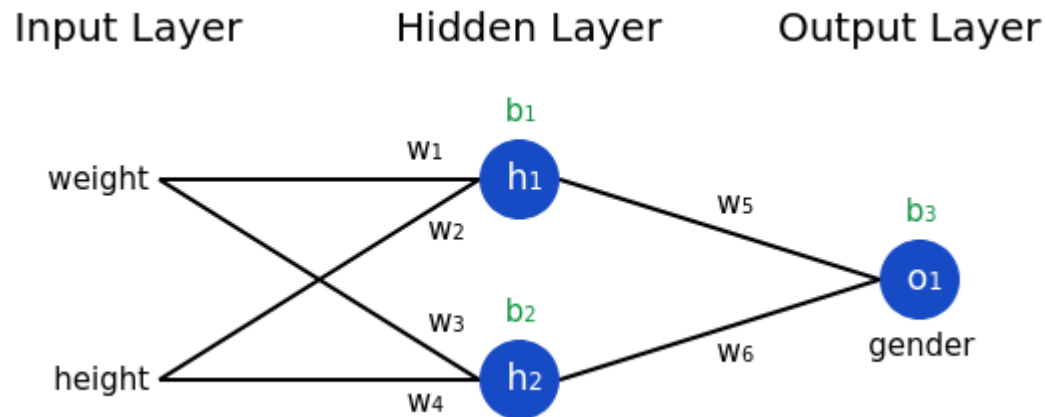
Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

Dans ce cas, la perte d'erreur quadratique moyenne n'est que l'erreur quadratique d'Alice = 1 :

$$\begin{aligned}\text{MSE} &= \frac{1}{1} \sum_{i=1}^1 (y_{\text{true}} - y_{\text{pred}})^2 \\ &= (y_{\text{true}} - y_{\text{pred}})^2 \\ &= (1 - y_{\text{pred}})^2\end{aligned}$$

Formation d'un réseau de neurones, partie 2

Une autre façon de penser à la perte est en fonction des **poids** et des **biais**.
Étiquetons chaque poids et biais de notre réseau :



Ensuite, nous pouvons écrire la perte **L** sous la forme d'une **fonction multivariable** :

$$L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$$



Formation d'un réseau de neurones, partie 2

$$L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$$

Imaginez que nous voulions modifier **w1**. Comment la perte **L** changerait-elle si on changeait **w1** ?

C'est une question à laquelle la **dérivée partielle** $\partial L / \partial w_1$ peut répondre.
Comment le calcule-t-on ?

Pour commencer, réécrivons la **dérivée partielle** en termes de $\partial y_{pred} / \partial w_1$ à la place :

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1}$$



Formation d'un réseau de neurones, partie 2

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1}$$

Nous pouvons calculer $\frac{\partial L}{\partial y_{pred}}$ car nous avons calculé $L = (1 - y_{pred})^2$ ci-dessus :

$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial (1 - y_{pred})^2}{\partial y_{pred}} = \boxed{-2(1 - y_{pred})}$$

Voyons maintenant quoi faire avec $\frac{\partial y_{pred}}{\partial w_1}$. Tout comme avant, soit h_1 , h_2 , o_1 les sorties des neurones qu'ils représentent. Puis:

$$y_{pred} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$

f is the sigmoid activation function, remember?



Formation d'un réseau de neurones, partie 2

Voyons maintenant quoi faire avec $\partial y_{pred} / \partial w_1$. Tout comme avant, soit h_1 , h_2 , o_1 les sorties des neurones qu'ils représentent. Puis:

$$y_{pred} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$

f is the sigmoid activation function, remember?

Puisque w_1 n'affecte que h_1 (pas h_2), nous pouvons écrire:

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = \boxed{x_1 * f'(w_1 x_1 + w_2 x_2 + b_1)}$$

Nous faisons la même chose pour $\partial h_1 / \partial w_1$:

$$\frac{\partial y_{pred}}{\partial w_1} = \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial y_{pred}}{\partial h_1} = \boxed{w_5 * f'(w_5 h_1 + w_6 h_2 + b_3)}$$

Formation d'un réseau de neurones, partie 2

C'est la deuxième fois que nous voyons $f'(x)$ (le dérivé de la fonction sigmoïde) maintenant ! **Dérivons-le** :

Nous utiliserons cette belle forme pour $f'(x)$ plus tard.

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

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x) * (1 - f(x))$$

Succès. Nous avons réussi à décomposer $\frac{\partial L}{\partial w_1}$ en plusieurs parties que nous pouvons calculer :

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$
$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial (1 - y_{pred})^2}{\partial y_{pred}} = -2(1 - y_{pred})$$
$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$
$$\frac{\partial h_1}{\partial w_1} = x_1 * f'(w_1 x_1 + w_2 x_2 + b_1)$$
$$\frac{\partial y_{pred}}{\partial h_1} = w_5 * f'(w_5 h_1 + w_6 h_2 + b_3)$$

Exemple : Calcul de la dérivée partielle

Nous allons continuer à prétendre que seule Alice est dans notre ensemble de données. (poids =135 (lb) et taille=66 (in) sont des valeurs moyennes)

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

Initialisons **tous les poids à 1** et **tous les biais à 0**. Si nous effectuons une propagation en avant (***feedforward pass***) à travers le réseau, nous obtenons :

$$\begin{aligned}h_1 &= f(w_1x_1 + w_2x_2 + b_1) \\&= f(-2 + -1 + 0) \\&= 0.0474\end{aligned}$$

$$h_2 = f(w_3x_1 + w_4x_2 + b_2) = 0.0474$$

$$\begin{aligned}o_1 &= f(w_5h_1 + w_6h_2 + b_3) \\&= f(0.0474 + 0.0474 + 0) \\&= 0.524\end{aligned}$$



Exemple : Calcul de la dérivée partielle

Le réseau génère **ypred=0,524**, ce qui ne favorise pas fortement **Male (0)** ou **Female (1)**.

Calculons $\frac{\partial L}{\partial w_1}$:

Nous l'avons fait! Cela nous indique que si nous augmentons **w1**, **L** augmenterait un tout petit peu en conséquence (2% de **w1**)

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\begin{aligned}\frac{\partial L}{\partial y_{pred}} &= -2(1 - y_{pred}) \\ &= -2(1 - 0.524) \\ &= -0.952\end{aligned}$$

$$\begin{aligned}\frac{\partial y_{pred}}{\partial h_1} &= w_5 * f'(w_5 h_1 + w_6 h_2 + b_3) \\ &= 1 * f'(0.0474 + 0.0474 + 0) \\ &= f(0.0948) * (1 - f(0.0948)) \\ &= 0.249\end{aligned}$$

$$\begin{aligned}\frac{\partial h_1}{\partial w_1} &= x_1 * f'(w_1 x_1 + w_2 x_2 + b_1) \\ &= -2 * f'(-2 + -1 + 0) \\ &= -2 * f(-3) * (1 - f(-3)) \\ &= -0.0904\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial w_1} &= -0.952 * 0.249 * -0.0904 \\ &= \boxed{0.0214}\end{aligned}$$



Entraînement : descente de gradient stochastique

Maintenant nous pouvons former (entraîner) un réseau de neurones !

Nous utiliserons un **algorithme d'optimisation** appelé **descente de gradient stochastique** (*Stochastic Gradient Descent*) qui nous indique comment modifier nos poids et biais pour minimiser les pertes.

Il s'agit essentiellement de cette **équation de mise à jour** :

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

η est une constante appelée **taux d'apprentissage** qui contrôle la vitesse à laquelle nous entraînons le réseau.

Tout ce que nous faisons est de **soustraire $\eta * \partial L / \partial w_1$** de **$w_1$** :

Si $\partial L / \partial w_1$ est **positif**, **w_1** diminuera, ce qui fait **diminuer L**.

Si $\partial L / \partial w_1$ est **négatif**, **w_1** augmentera, ce qui fera **diminuer L**.

Si nous faisons cela **pour chaque poids et biais** du réseau, la perte diminuera lentement et notre réseau s'améliorera.



Entraînement : descente de gradient stochastique

Notre processus de formation (**entraînement**) ressemblera à ceci :

1. Choisissez un **échantillon de notre ensemble de données**.

C'est ce qui en fait une descente de gradient stochastique - nous n'opérons que sur **un échantillon à la fois**.

2. Calculer **toutes les dérivées partielles** de la perte par rapport aux **poids** ou aux **biais** (par exemple $\partial L \partial w_1$, $\partial L \partial w_2$, etc.).

3. Utilisez l'équation de **mise à jour** pour mettre à jour chaque poids et biais.

4. Revenez à l'**étape 1**.

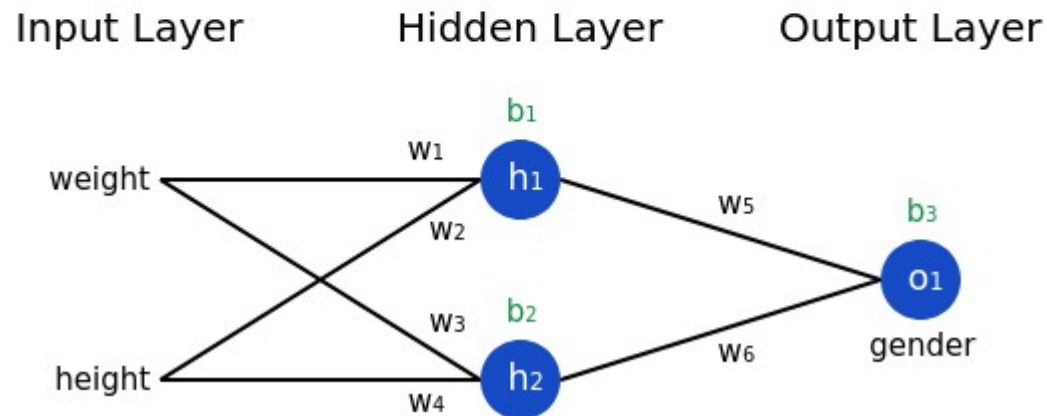
Voyons-le en **action** !

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

Code : Un réseau de neurones complet

Il est enfin temps de mettre en place un réseau de neurones complet :

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1



Code : fonctions utilitaires

IA.Lab0.ipynb ☆

Fichier Modifier Affichage Insérer Exécution Outils Aide [Toutes les modifications c](#)

Code + Texte

[] 1

0.5

```
1 import numpy as np
2
3 def sigmoid(x):
4     # Sigmoid activation function:  $f(x) = 1 / (1 + e^{-x})$ 
5     return 1 / (1 + np.exp(-x))
6
7 def deriv_sigmoid(x):
8     # Derivative of sigmoid:  $f'(x) = f(x) * (1 - f(x))$ 
9     fx = sigmoid(x)
10    return fx * (1 - fx)
11
12 def mse_loss(y_true, y_pred):
13     # y_true and y_pred are numpy arrays of the same length.
14     return ((y_true - y_pred) ** 2).mean()
15
```

Code : la classe OurNeuralNetwork

Méthodes :
__init__ et
feedforward :

IA.Lab0.ipynb ☆

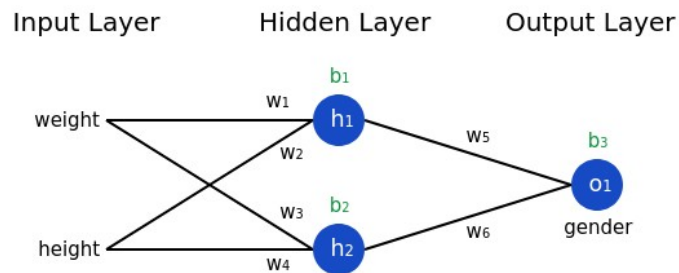
Fichier Modifier Affichage Insérer Exécution Outils Aide [Toutes les modifications on](#)

Code + Texte

```
15
16 class OurNeuralNetwork:
17
18     def __init__(self):
19         # Weights
20         self.w1 = np.random.normal()
21         self.w2 = np.random.normal()
22         self.w3 = np.random.normal()
23         self.w4 = np.random.normal()
24         self.w5 = np.random.normal()
25         self.w6 = np.random.normal()
26
27         # Biases
28         self.b1 = np.random.normal()
29         self.b2 = np.random.normal()
30         self.b3 = np.random.normal()
31
32     def feedforward(self, x):
33         # x is a numpy array with 2 elements.
34         h1 = sigmoid(self.w1 * x[0] + self.w2 * x[1] + self.b1)
35         h2 = sigmoid(self.w3 * x[0] + self.w4 * x[1] + self.b2)
36         o1 = sigmoid(self.w5 * h1 + self.w6 * h2 + self.b3)
37         return o1
38
```

Code : les datasets et la fonction zip()

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1



```
1 # Define dataset
2 data = np.array([
3     [-2, -1], # Alice
4     [25, 6],  # Bob
5     [17, 4],  # Charlie
6     [-15, -6], # Diana
7 ])
8 all_y_trues = np.array([
9     1, # Alice
10    0, # Bob
11    0, # Charlie
12    1, # Diana
13 ])
14
15 list(zip(data, all_y_trues))
```

```
[(array([-2, -1]), 1),
 (array([25, 6]), 0),
 (array([17, 4]), 0),
 (array([-15, -6]), 1)]
```



Code : faire une propagation - y_pred

Nous calculons la prédiction **y_pred**, par une **propagation** avec les valeurs de **w1,w2,b1** , **w3,w4,b2**, **w5,w6,b3** de la dernière époque :

```
38
39 def train(self, data, all_y_trues):
40     learn_rate = 0.1
41     epochs = 100 # number of times to loop through the entire dataset
42     for epoch in range(epochs):
43         for x, y_true in zip(data, all_y_trues):
44             # --- Do a feedforward (we'll need these values later)
45             sum_h1 = self.w1 * x[0] + self.w2 * x[1] + self.b1
46             h1 = sigmoid(sum_h1)
47             sum_h2 = self.w3 * x[0] + self.w4 * x[1] + self.b2
48             h2 = sigmoid(sum_h2)
49             sum_o1 = self.w5 * h1 + self.w6 * h2 + self.b3
50             o1 = sigmoid(sum_o1)
51             y_pred = o1
52
```


Code : calculer les dérivées partielles

Nous calculons les **dérivées partielles** pour les nouvelles valeurs de poids et de biais. **d_L_d_w1** représente la dérivée partielle de **L** sur **w1** :

```
53     # --- Calculate partial derivatives.
54     # --- Naming: d_L_d_w1 represents "partial L / partial w1"
55     d_L_d_ypred = -2 * (y_true - y_pred)
56     # Neuron o1
57     d_ypred_d_w5 = h1 * deriv_sigmoid(sum_o1)
58     d_ypred_d_w6 = h2 * deriv_sigmoid(sum_o1)
59     d_ypred_d_b3 = deriv_sigmoid(sum_o1)
60
61     d_ypred_d_h1 = self.w5 * deriv_sigmoid(sum_o1)
62     d_ypred_d_h2 = self.w6 * deriv_sigmoid(sum_o1)
63
64     # Neuron h1
65     d_h1_d_w1 = x[0] * deriv_sigmoid(sum_h1)
66     d_h1_d_w2 = x[1] * deriv_sigmoid(sum_h1)
67     d_h1_d_b1 = deriv_sigmoid(sum_h1)
68
69     # Neuron h2
70     d_h2_d_w3 = x[0] * deriv_sigmoid(sum_h2)
71     d_h2_d_w4 = x[1] * deriv_sigmoid(sum_h2)
72     d_h2_d_b2 = deriv_sigmoid(sum_h2)
73
```

Code : application d'un pas d'apprentissage

Nous modifions les valeurs de poids et de biais avec la valeur de taux d'apprentissage - $\dot{\eta}$ (learn_rate=0.1):

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

```
73
74 # --- Update weights and biases
75 # Neuron h1
76 self.w1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w1
77 self.w2 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w2
78 self.b1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_b1
79
80 # Neuron h2
81 self.w3 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w3
82 self.w4 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w4
83 self.b2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_b2
84
85 # Neuron o1
86 self.w5 -= learn_rate * d_L_d_ypred * d_ypred_d_w5
87 self.w6 -= learn_rate * d_L_d_ypred * d_ypred_d_w6
88 self.b3 -= learn_rate * d_L_d_ypred * d_ypred_d_b3
89
```

Code : application d'un pas d'apprentissage

Nous modifions les valeurs de poids et de biais avec la valeur de taux d'apprentissage - $\dot{\eta}$ (learn_rate=0.1):

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

```
73  
74 # --- Update weights and biases  
75 # Neuron h1  
76 self.w1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w1  
77 self.w2 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w2  
78 self.b1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_b1  
79  
80 # Neuron h2  
81 self.w3 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w3  
82 self.w4 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w4  
83 self.b2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_b2  
84  
85 # Neuron o1  
86 self.w5 -= learn_rate * d_L_d_ypred * d_ypred_d_w5  
87 self.w6 -= learn_rate * d_L_d_ypred * d_ypred_d_w6  
88 self.b3 -= learn_rate * d_L_d_ypred * d_ypred_d_b3  
89
```



Code : calcul de perte, affichage

A la fin d'une (ou plusieurs) époque on fait une prédiction pour calculer la perte avec les poids et biais actuellement disponibles:

```
89
90     # --- Calculate total loss at the end of each epoch
91     if epoch % 10 == 0:
92         y_preds = np.apply_along_axis(self.feedforward, 1, data)
93         loss = mse_loss(all_y_trues, y_preds)
94         print("Epoch %d loss: %.3f" % (epoch, loss))
95
```

Notez que dans le cas ci-dessus on fait ce calcul toutes les 10 époques.




Code : instantiation et entraînement

Pour lancer le programme :

on **prépare les données**,
data, all_y_trues

on **instancie** notre classe
OurNeuralNetwork comme **network**

on **execute** la méthode
.train(data,all_y_trues) de cette
classe.



```
96 # Define dataset
97 data = np.array([
98     [-2, -1], # Alice
99     [25, 6],  # Bob
100    [17, 4],   # Charlie
101    [-15, -6], # Diana
102 ])
103 all_y_trues = np.array([
104     1, # Alice
105     0, # Bob
106     0, # Charlie
107     1, # Diana
108 ])
109
110 # Train our neural network!
111 network = OurNeuralNetwork()
112 network.train(data, all_y_trues)
113
```



Code : résultat d'apprentissage

```
96 # Define dataset
97 data = np.array([
98     [-2, -1], # Alice
99     [25, 6],  # Bob
100    [17, 4],   # Charlie
101    [-15, -6], # Diana
102 ])
103 all_y_trues = np.array([
104     1, # Alice
105     0, # Bob
106     0, # Charlie
107     1, # Diana
108 ])
109
110 # Train our neural network!
111 network = OurNeuralNetwork()
112 network.train(data, all_y_trues)
113
```

Epoch 0 loss: 0.315
Epoch 10 loss: 0.261
Epoch 20 loss: 0.204
Epoch 30 loss: 0.080
Epoch 40 loss: 0.058
Epoch 50 loss: 0.046
Epoch 60 loss: 0.037
Epoch 70 loss: 0.031
Epoch 80 loss: 0.027
Epoch 90 loss: 0.024

Code : modification des hyper-paramètres

Expérimentez avec le programme en modifiant ces **hyper-paramètres**:

- **taux d'apprentissage** : `learn_rate = 0.1`

- **nombre d'époques** : `epochs = 1000`

plus la fréquence d'affichage : `disp=epochs/10`

Ajoutez affichage avec les éléments du code:

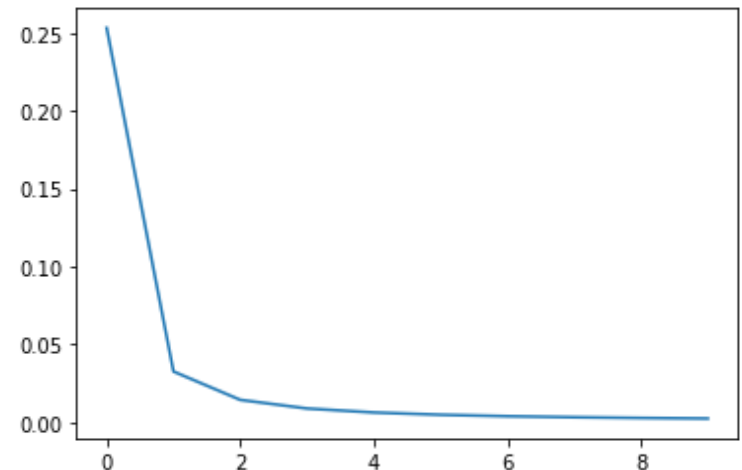
```
import matplotlib.pyplot as plt
```

```
loss_arr=[]
```

```
loss_arr.append(loss)
```

```
plt.plot(loss_arr)  
plt.show()
```

```
Epoch 0 loss: 0.253  
Epoch 100 loss: 0.033  
Epoch 200 loss: 0.015  
Epoch 300 loss: 0.009  
Epoch 400 loss: 0.006  
Epoch 500 loss: 0.005  
Epoch 600 loss: 0.004  
Epoch 700 loss: 0.003  
Epoch 800 loss: 0.003  
Epoch 900 loss: 0.003
```






Code : prédictions

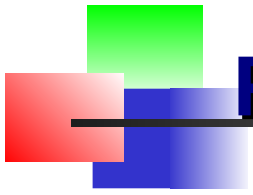
Nous pouvons maintenant utiliser le réseau pour prédire les sexes :

Attention : les valeurs de poids/taille de personnes dans les commentaires ne sont pas correctes !



```
1 # # Make some predictions
2 maria = np.array([-2, -1]) # 128 pounds, 63 inches
3 hugo = np.array([25, 6]) # 155 pounds, 68 inches
4 sophie = np.array([2, -1]) # 128 pounds, 63 inches
5 nico = np.array([32, 7]) # 155 pounds, 68 inches
6 print("Maria: %.3f" % network.feedforward(maria)) # 1 - F
7 print("Hugo: %.3f" % network.feedforward(hugo)) # 0 - M
8 print("Sophie: %.3f" % network.feedforward(sophie)) # 1 - F
9 print("Nico: %.3f" % network.feedforward(nico)) # 0 - M
```

```
Maria: 0.949
Hugo: 0.060
Sophie: 0.808
Nico: 0.060
```



Résumé

Un petit récapitulatif de ce que nous avons fait :

1. Nous avons introduit les **neurones**, les éléments constitutifs des réseaux de neurones.
2. On a utilisé la **fonction d'activation sigmoïde** dans nos neurones.
3. Les réseaux de neurones ne sont que des **neurones connectés entre eux**.
4. La création d'un réseaux - jeu de données avec le **poids** et la **taille** comme **entrées** (ou **caractéristiques**) et le **sexe** comme **sortie** (ou **étiquette**).
5. Nous avons pris la connaissance des **fonctions de perte** et la perte d'**erreur quadratique moyenne** (MSE).
6. Nous avons réalisé que la **formation d'un réseau** ne fait que **minimiser sa perte**.
7. La **rétro-propagation** est utilisée pour calculer les **dérivées partielles**.
8. Nous avons utilisé la **descente de gradient stochastique** (SGD) pour entraîner notre réseau.
9. Nous avons fait une **prédiction**