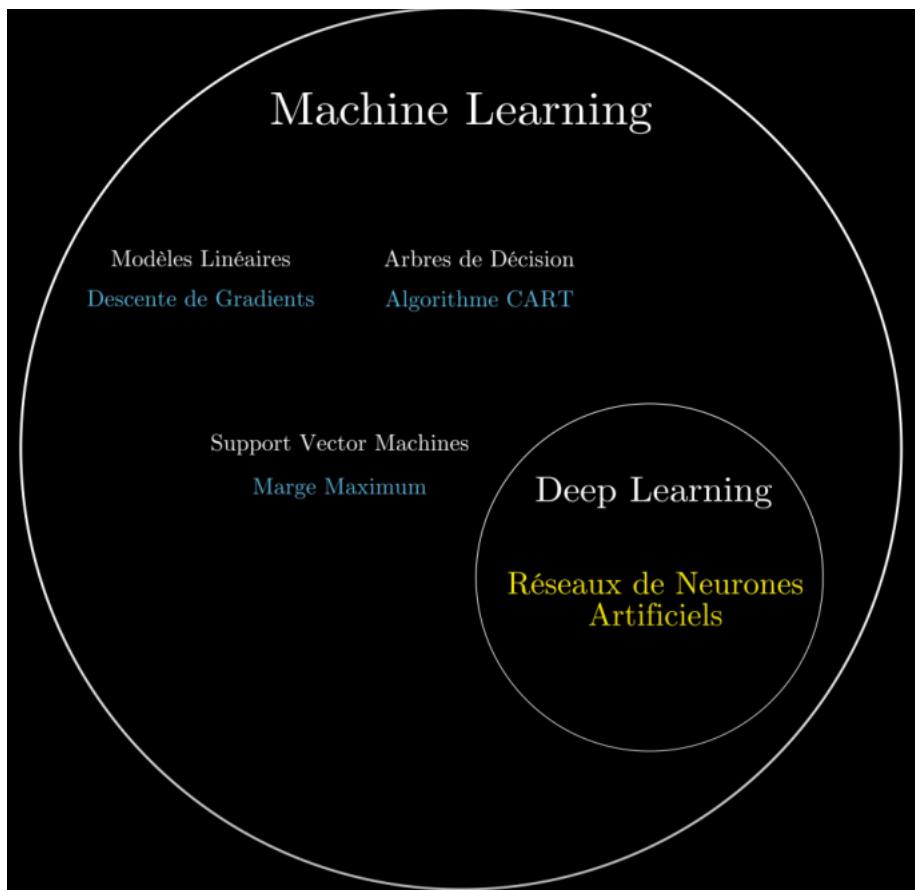
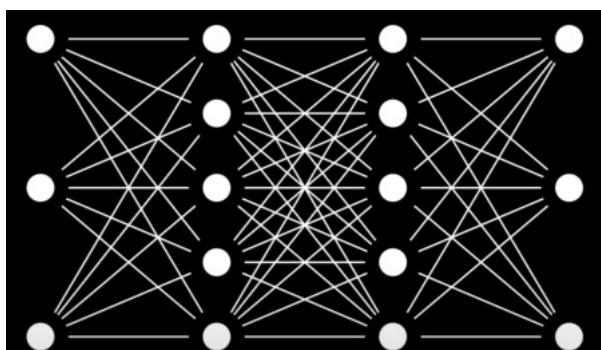


Deep Learning

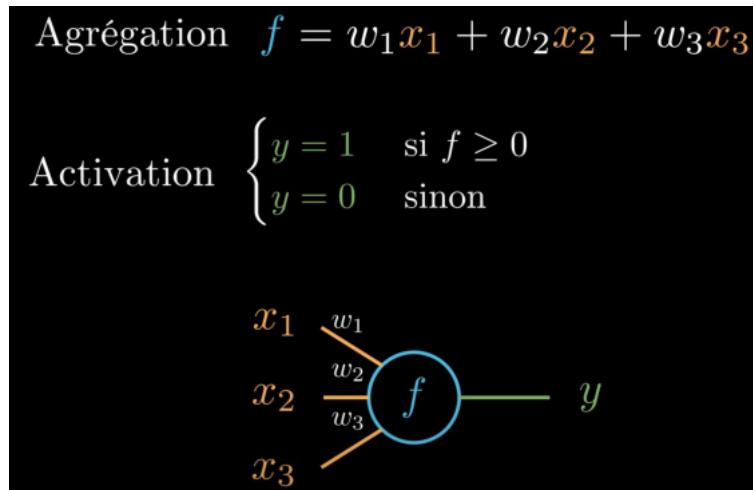


Le **Deep Learning** est une branche dans le **Machine Learning**. Le principe est le même qu'en ML, on fournit à notre modèle des données (un Dataset), et on utilise un algorithme d'optimisation pour ajuster le modèle pour avoir une Fonction Coût la plus faible possible sur les différents Set (Train, Validation & Test).

Mais cette fois-ci notre modèle n'est pas une simple fonction de type $f(x) = ax + b$, mais plutôt un réseau de fonction connecté les unes aux autres :



Et plus le réseau contient de **fonctions**, plus la machine va être capable d'apprendre à réaliser des **tâches complexes** (identifier une personne sur une photo, conduire une voiture etc).



Pitts et McCulloch (inventeurs des premiers neurones) se sont inspirés des réseaux de neurones humaines pour construire ceux artificielles.

Il y a 2 étapes, d'abord on fait **l'agrégation** :

- On fait la somme de toutes les entrées du neurones, en multipliant chaque entrée par un coefficient **w** (un poids, weight), ce coefficient représente l'activité synaptique, c-a-d le fait que le signal soit excitateur (donc $w = +1$) ou inhibiteur (donc $w = -1$).
- Ensuite, on passe à la phase **d'activation**, on regarde le résultat du calcul effectué précédemment, et si celui-ci dépasse un certain seuil (souvent 0), alors le neurone va s'activer et retourne une sortie $y=1$ sinon $y=0$ il ne s'active pas.

Avec ça, Rosenblatt, va inventer le perceptron, qui va rajouter par dessus cette théorie de fonctionnement de neurones artificiels, un algorithme d'apprentissage qui va permettre de trouver les valeurs pour les poids **w** pour avoir les sorties qui nous convienne.

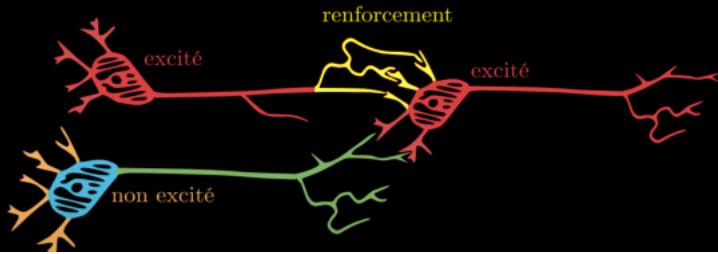
Il c'est inspiré de la **théorie de Hebb**, qui dit que lorsque 2 neurones biologiques sont excités conjointement, alors ils renforcent leur lien synaptique.

Théorie de Hebb :

Lorsque 2 neurones biologiques sont **excités** conjointement, alors ils **renforcent** leur lien synaptique.



Donald Hebb

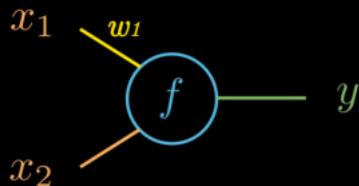


En neuroscience c'est ce qu'on appelle la **plasticité synaptique**. Et c'est ce qui permet à notre cerveau de construire notre mémoire, d'apprendre de nouvelles choses ou encore de faire de nouvelles associations.

Et c'est avec cela que le perceptron a été construit :

Le Perceptron (1957) :

Entraîner un neurone artificiel sur des **données de références** (X, y) pour que celui-ci **renforce** ses paramètres W à chaque fois qu'une **entrée X est activée** en même temps que la **sortie y** présente dans ces données.



$$W = W + \alpha(y_{true} - y)X$$

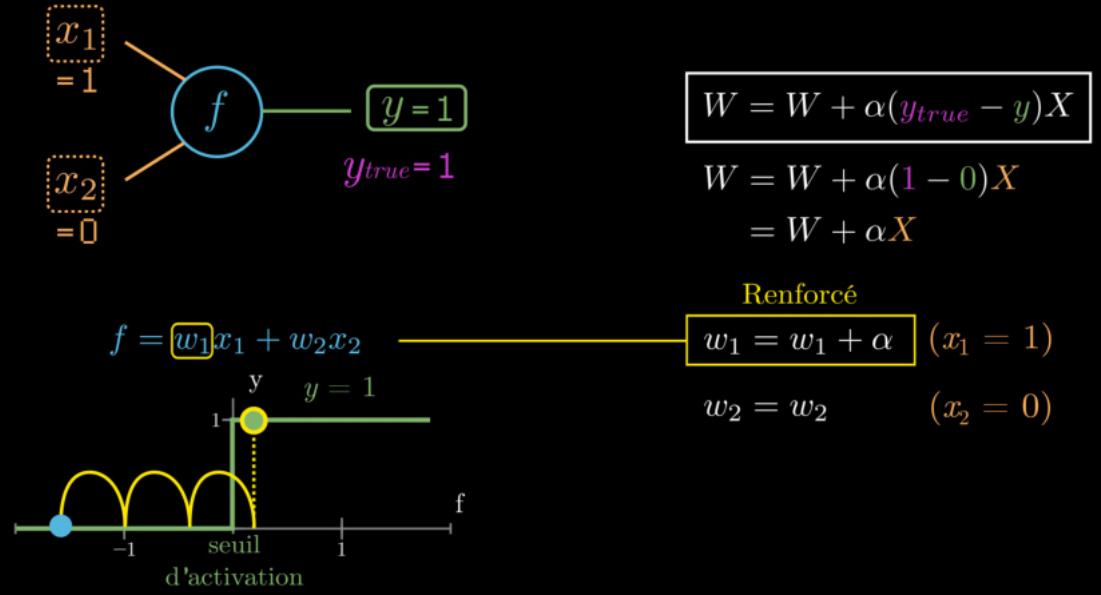
y_{true} : sortie de référence

y : sortie produite par le neurone

X : entrée du neurone

α : vitesse d'apprentissage

Le Perceptron (1957) :



On voit que si notre sortie nous donne 0 alors qu'on souhaite avoir 1, on aura $w = w + \text{Alpha} * X$, donc pour les entrées X qui valent 1 le coefficient w se verra **augmenter d'un petit pas (Alpha)** et sera alors **renforcé** ce qui provoquera une **augmentation de la fonction $w_1x_1 + w_2x_2$** et qui **rapprochera donc notre neurone de son seuil d'activation petit à petit**.

Aussi longtemps qu'on sera en dessous de ce seuil, donc aussi longtemps que le neurone produira une mauvaise sortie, alors le coefficient w continuera d'augmenter grâce à notre formule, jusqu'au moment où y_true vaudra y (1-1) et à ce moment là notre formule donnera $w = w+0$, ce qui fait que nos paramètres arrêterons d'évoluer et on aura donc les meilleurs paramètres.

Plus tard, après ça, il y a eu l'invention du perceptron multicouches de Hinton.

Car le perceptron tout seul est un modèle linéaire, avec w_1, w_2 pour ajuster la droite et le biais le $+b$ pour déplacer la droite, mais la plupart des problèmes qui existent ne sont pas linéaires, donc le perceptron à lui seul n'est pas très utile.

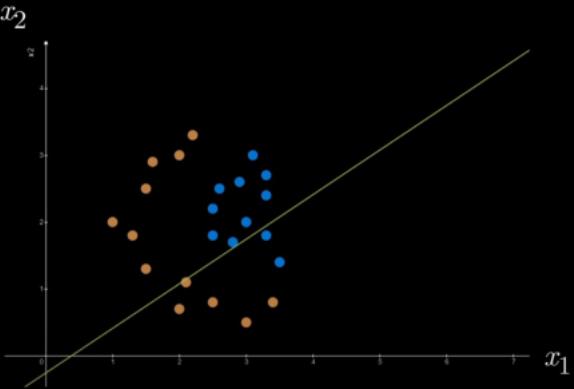
Le Perceptron est un modèle linéaire

$$f = w_1x_1 + w_2x_2 + b$$

$w_1 = -0.26$

 $w_2 = 0.39$

 $b = 0.1$

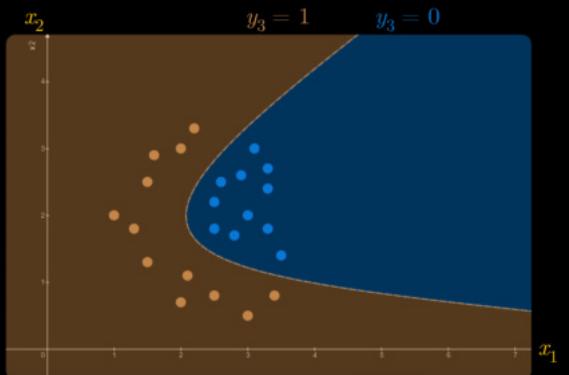
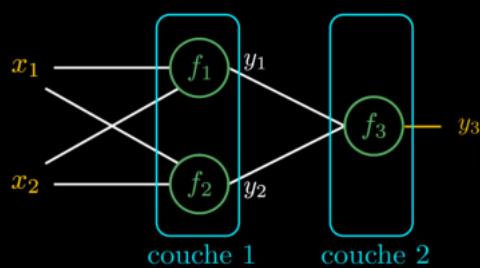


$$\begin{cases} y = 1 & \text{si } f \geq 0 \\ y = 0 & \text{sinon} \end{cases}$$

Mais si on calcul pour chaque perceptron on fait le calcul qu'on va ensuite ajouter à un autre perceptron puis un autre etc etc, alors on aura un modèle qui ne sera plus linéaire :

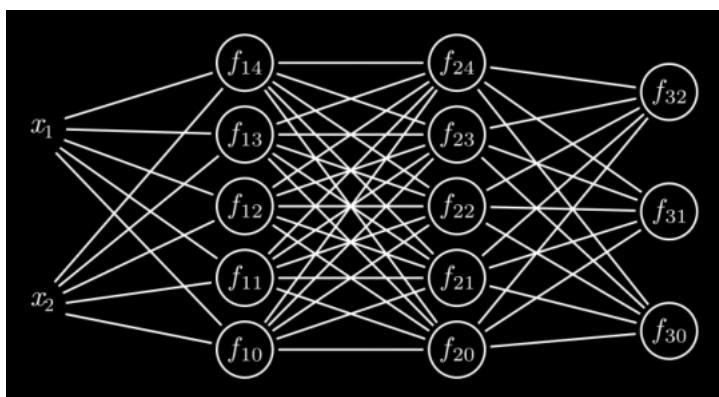
Réseau de Neurones Artificiels

- 3 neurones
- 2 couches



Perceptron Multi-Couche

Et les couches ont peut en mettre 4-8-10 etc :



Plus on en met et plus le résultat en sortie sera complexe et intéressant.

Mais comment entraîner un tel réseau de neurone pour qu'il fasse ce qu'on lui demande de faire ?

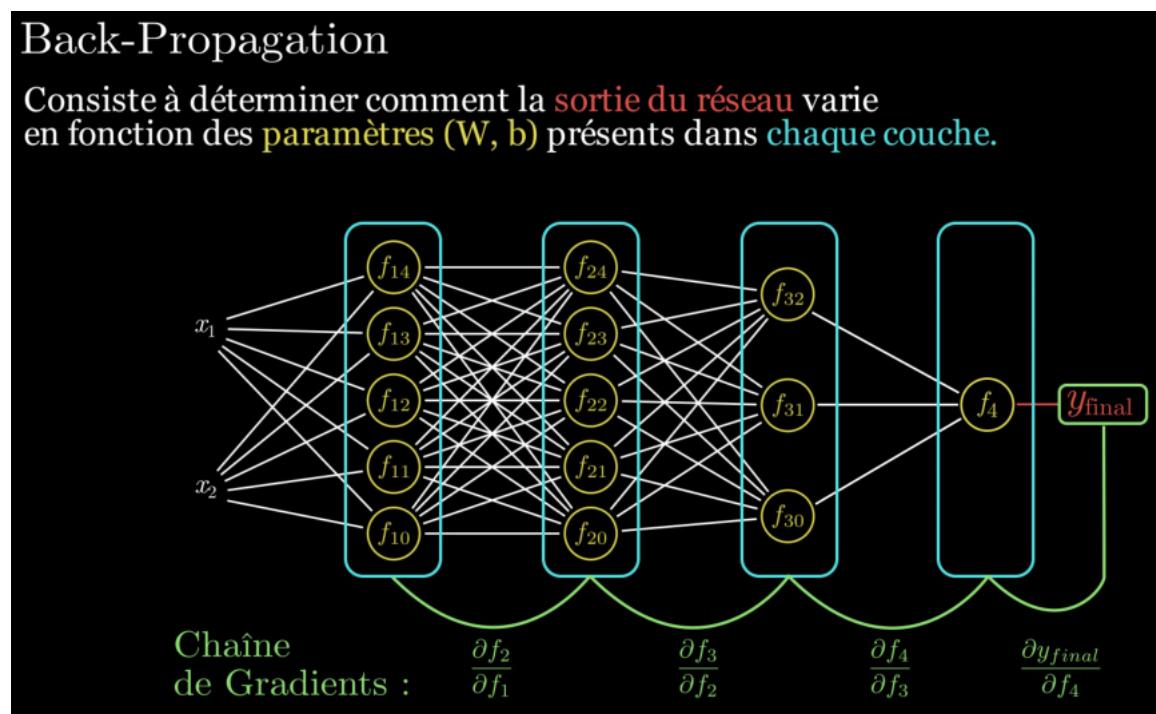
Pour ça on utilise une technique qui s'appelle la **backpropagation**.

Elle consiste à déterminer comment la sortie du réseau **varie** en fonction des paramètres (**w, b**) présents dans **chaque couche**.

Pour ça on **calcul une chaîne de Gradient** indiquant comment la sortie varie en fonction de la dernière couche, puis comment la dernière couche varie en fonction de la couche précédente etc etc jusqu'à retourner au début aux inputs.

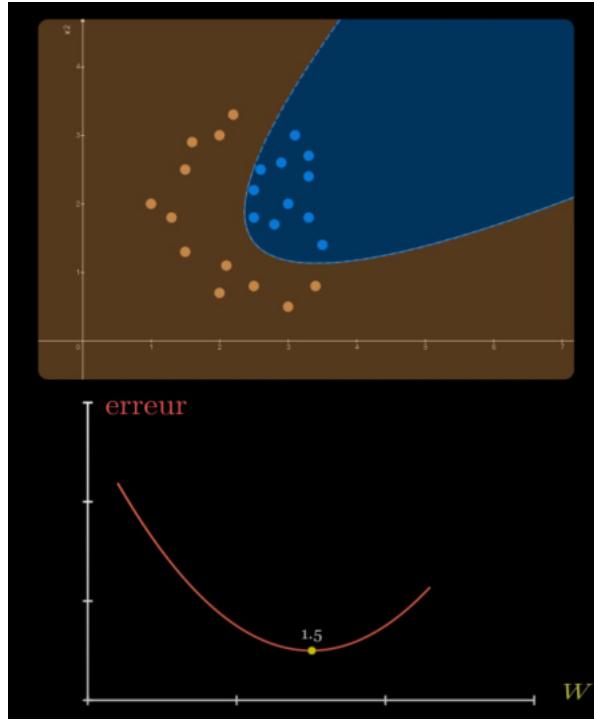
Back-Propagation

Consiste à déterminer comment la **sortie du réseau** varie en fonction des **paramètres (W, b)** présents dans **chaque couche**.



On propage vers l'arrière.

Avec ces informations, ces **gradients**, on peut alors mettre à jour les **paramètres (w, b)** de **chaque couche** de telle sorte à ce qu'ils **minimisent l'erreur** entre la **sortie** du modèle et la **réponse attendue (y_true)**.



Et pour faire ça, on utilise l'algorithme de **Gradient Descent**.

$$W = W - \alpha \frac{\partial E_{\text{erreur}}}{\partial W}$$

En Résumé pour développer un réseau de neurone artificiels :

1- Forward Propagation : on fait circuler les données de la première couche jusqu'à la dernière afin de produire une sortie y .

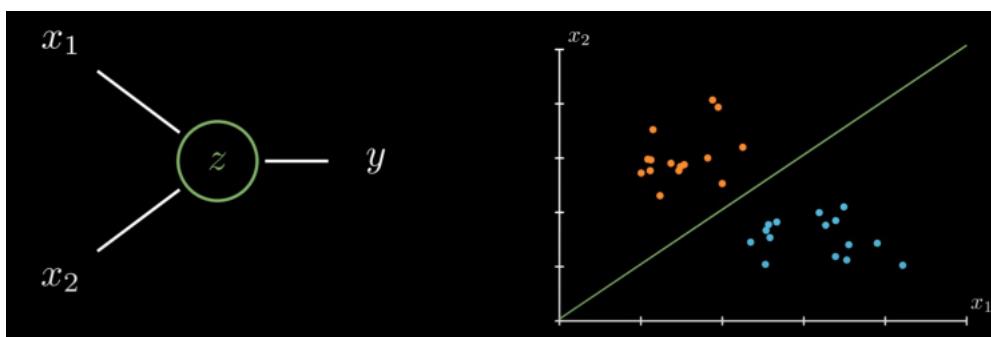
2- Cost Function : on calcule l'erreur entre cette sortie et la sortie de référence y_{true} que l'on désire avoir, pour ça on utilise donc une fonction coût.

3- Backward Propagation : on mesure comment cette fonction coût varie par rapport à chaque couche de notre modèle, en partant de la dernière jusqu'à la première.

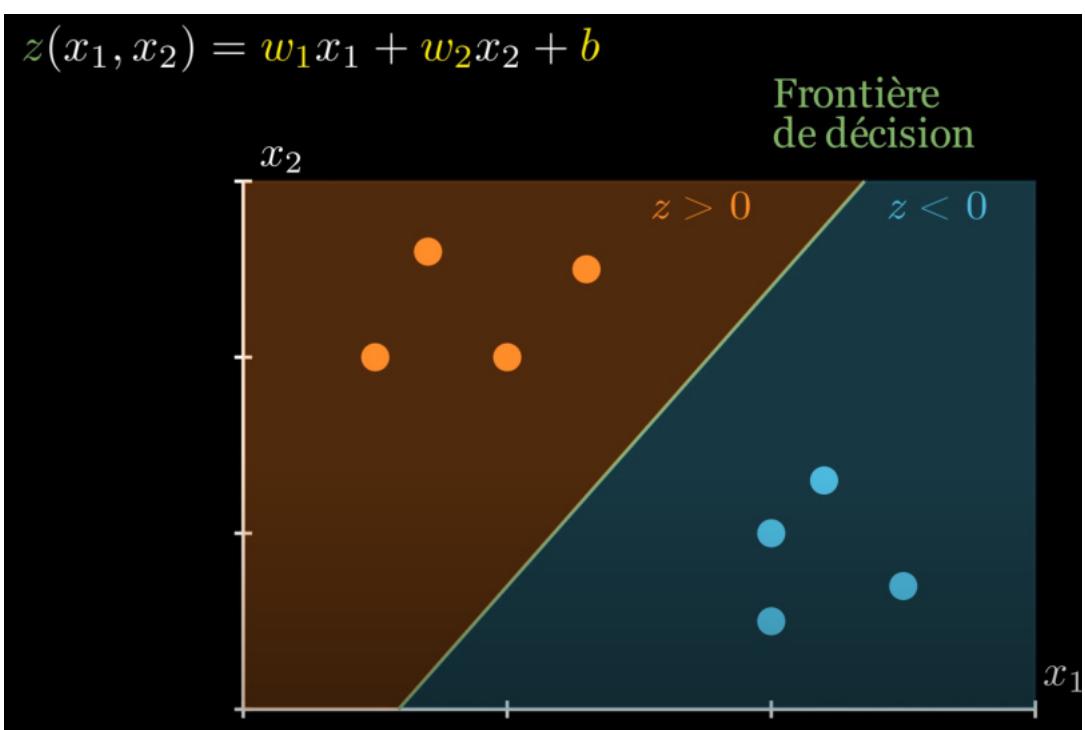
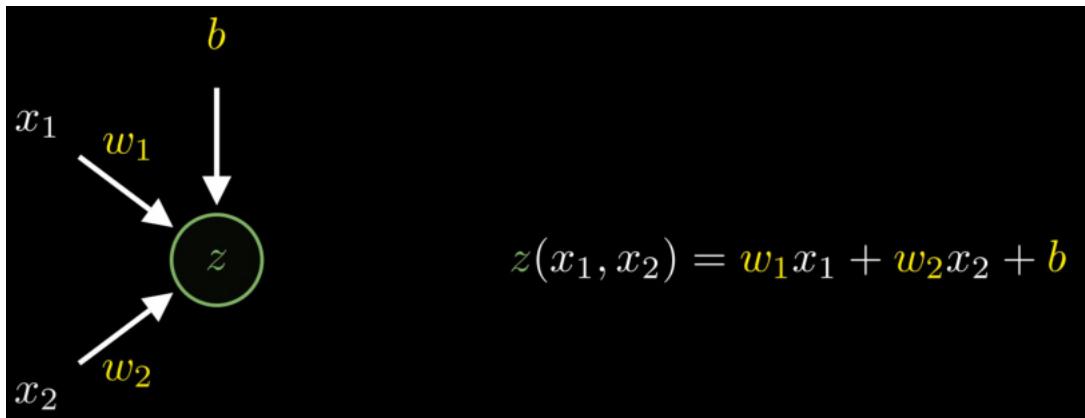
4- Gradient Descent : on corrige chaque paramètres du modèle grâce à l'algorithme de Descente du Gradient avant de reboucler vers la première étape (Forward Propagation) pour recommencer un cycle d'entraînement.

I-Le Perceptron :

Le perceptron est l'unité de base des réseaux de neurones. Il s'agit d'un modèle de classification binaire, capable de séparer linéairement 2 classes de point.



C'est un modèle linéaire et il aura pour équation :



On constate que la frontière de décision correspond aux valeurs de **x1 et x2** pour lesquelles **z = 0**.

Du coup pour prédire la classe d'appartenance ici d'un élément en orange ou bleu, il va falloir régler **les paramètres w et b**, de façon à séparer du mieux possible nos 2 classes et après on pourra prédire si un élément appartient dans la classe bleu ou orange en regardant simplement le signe de **z**.

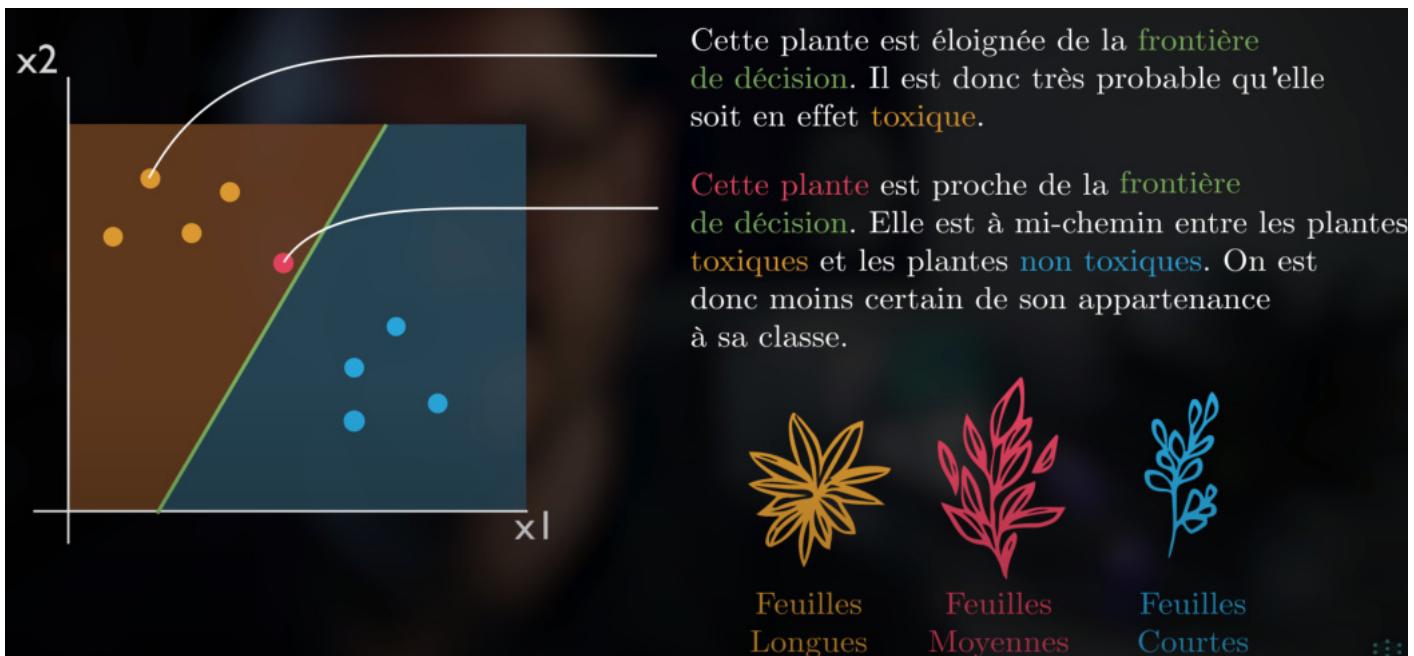
Si **z < 0** alors bleu sinon orange.

Prédictions :

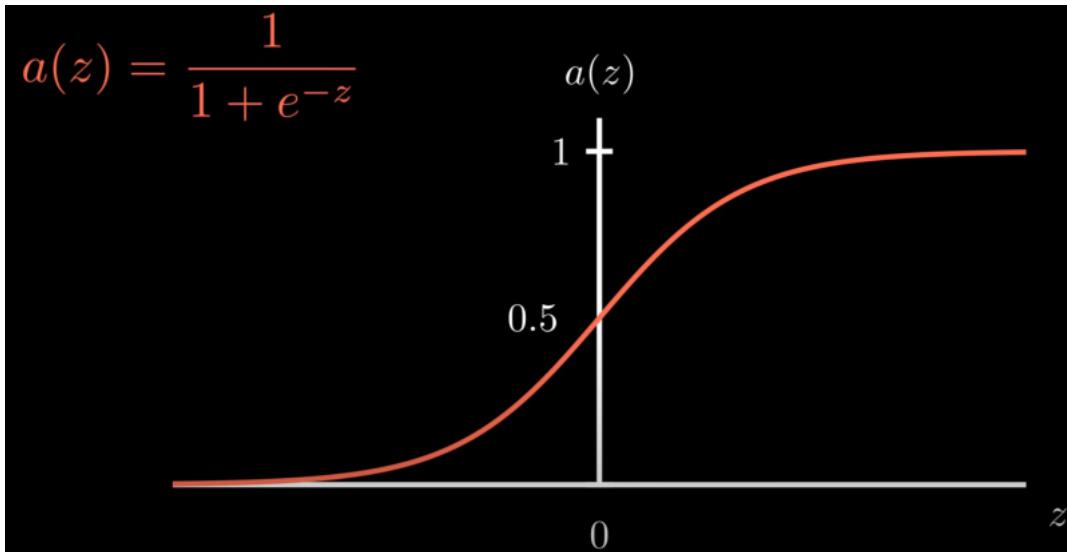
y_pred = 0 si z < 0

y_pred = 1 si z >= 1

Pour peaufiner le modèle on rajoute de la **probabilité**, exemple pour des plantes toxiques ou non :



Pour ça on pourrait utiliser une fonction d'activation nous retournant une sortie qui s'approche de 0 ou 1 au fur et à mesure qu'on s'éloigne de la **frontière de décision (là où $z = 0$)**.
La **fonction Sigmoïde (fonction Logistique)** permet de faire ça :

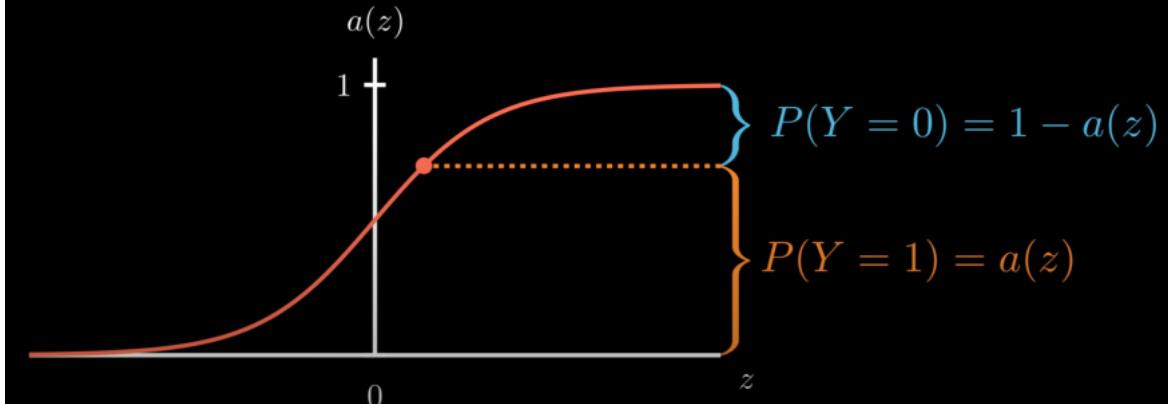


Cette fonction permet de convertir la sortie z en une probabilité $a(z)$, ici celle qu'une plante appartienne à la classe 1 (toxique). Exemple, si on a une plante avec un $z=1,4$, on aura une probabilité de **0,8** donc 80 % de chance que ça soit une plante toxique.

Et ces probabilités suivent en réalité une loi de Bernoulli.

Loi de Bernoulli :

$$P(Y = y) = a(z)^y \times (1 - a(z))^{1-y}$$



Si on décompose les 2 cas, $y=1$ et $y=0$ on a :

$$P(Y = 0) = a(z)^0 \times (1 - a(z))^{1-0}$$

$$P(Y = 1) = a(z)^1 \times (1 - a(z))^{1-1}$$

$$P(Y = 0) = 1 \times (1 - a(z))^{1-0}$$

$$P(Y = 1) = a(z)^1 \times 1$$

(rappel : $x^0 = 1$)

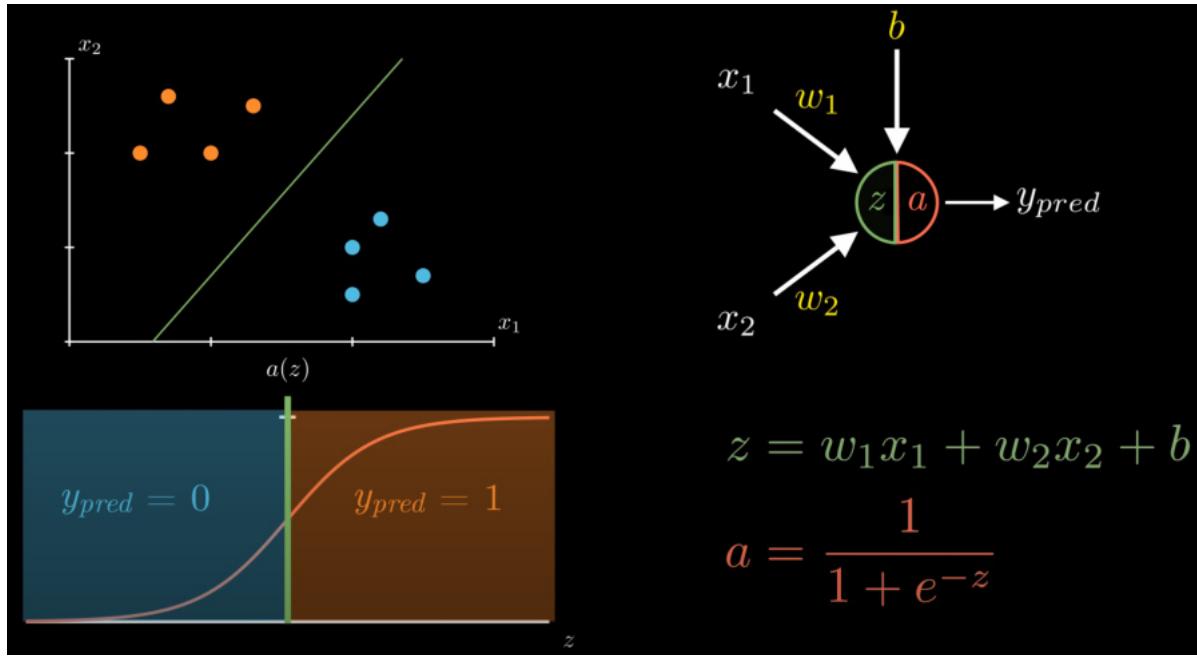
On retombe sur les expressions vu au-dessus sur le graphique.

En Résumé :

On trouve dans un neurone une fonction linéaire :

$$z = x_1 w_1 + x_2 w_2 + b$$

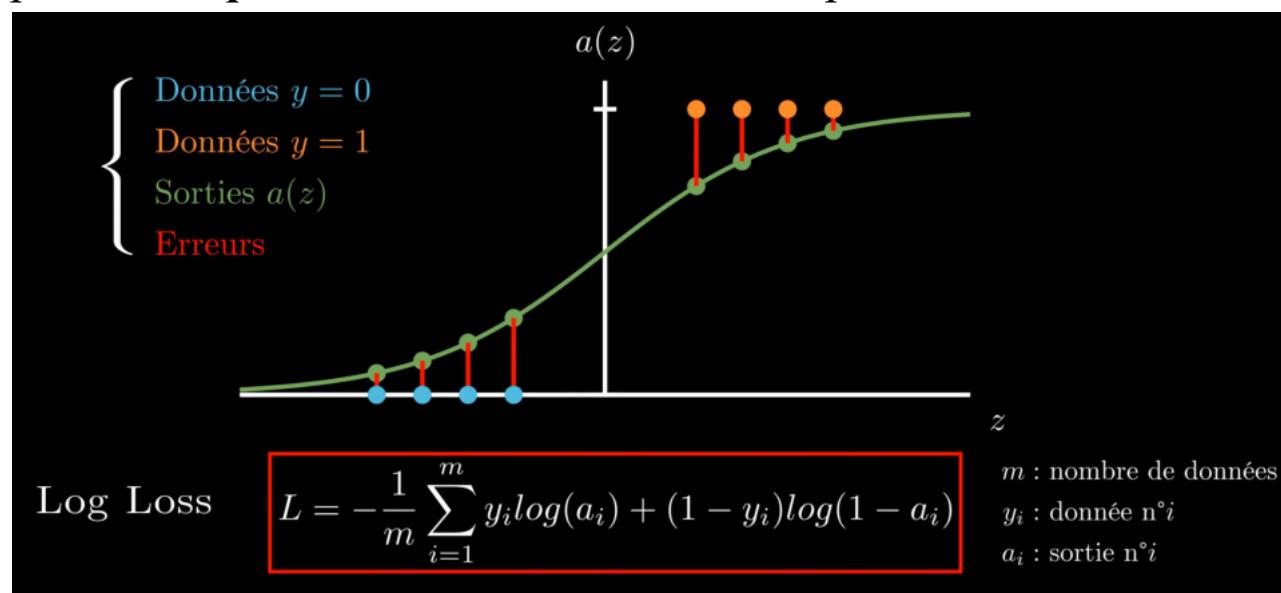
Suivi d'une fonction d'activation, il en existe beaucoup, la plus simple étant la fonction **Sigmoïde**, qui nous retourne une probabilité suivant une loi de Bernoulli.



Le but ici, va être de régler les paramètres w et b de façon à obtenir le **meilleur modèle possible**, c-a-d le modèle qui fait les plus petites erreurs entre les **sorties $a(z)$** et les **vraies données y** . Pour ça, on va commencer par définir une **Fonction Coût** qui va permettre de **mesurer ces erreurs**.

La Fonction Coût :

En ML, une Fonction Coût (Cost Function), c'est une fonction qui permet de **quantifier** les **erreurs effectués** par un modèle.



(ici la Log Loss)

Un façon d'évaluer la performance de son modèle c'est de calculer sa **vraisemblance**, en statistique la vraisemblance indique la **plausibilité** du modèle **vis-à-vis de vraies données**.

Par Analogie :

Une histoire est vraisemblable lorsqu'elle est en accord avec des faits qui se sont vraiment déroulés.

Et pour voir si notre modèle est vraisemblable, on va donc faire le produit de toutes nos probabilités en utilisant la loi de Bernoulli :

$$L = \prod_{i=1}^m a_i^{y_i} \times (1 - a_i)^{1-y_i}$$

L = Likelihood (vraisemblance en anglais)

Si le résultat de ce calcul est de 100 % ça signifiera que notre modèle est **vraisemblable** à 100 % car il est en **accord parfait** avec les données que l'on considère comme '**vrai**'.

A l'inverse si le résultat est **proche de 0**, alors le modèle est **invraisemblable**.

Le problème avec la vraisemblance :

Comme on effectue un produit de probabilité cela fait que plus on a de nombre et plus le résultat tend vers 0 (exemple : $0,5 * 0,5 = 0,25$).

Donc il faut une astuce pour calculer cette vraisemblance sans pour autant converger vers 0.

Et cette astuce c'est d'utiliser une fonction **logarithme** !

Car $\log(a*b) = \log(a) + \log(b)$

Le logarithme d'un produit nous donne la somme des logarithmes.

On applique alors un logarithme à notre fonction de vraisemblance :

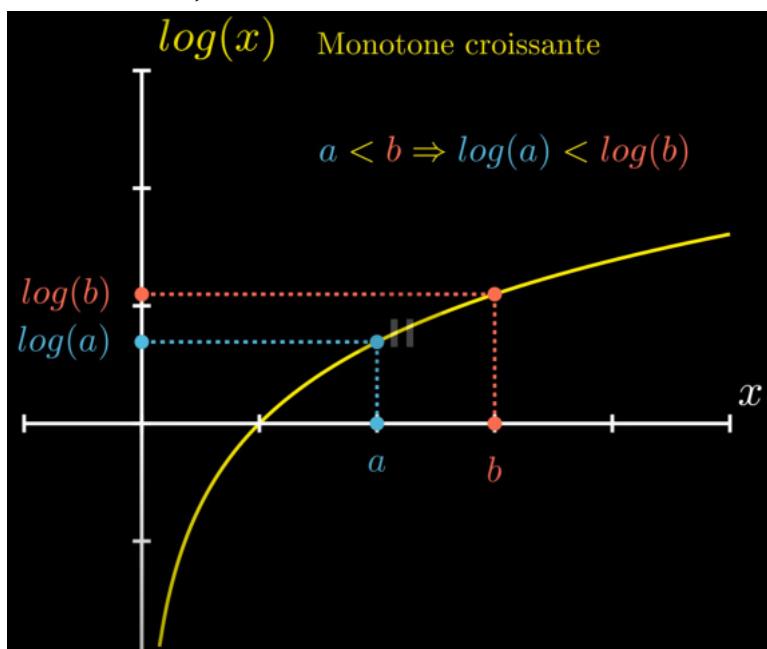
$$L = \prod_{i=1}^m a_i^{y_i} \times (1 - a_i)^{1-y_i}$$

$$\log(ab) = \log(a) + \log(b)$$

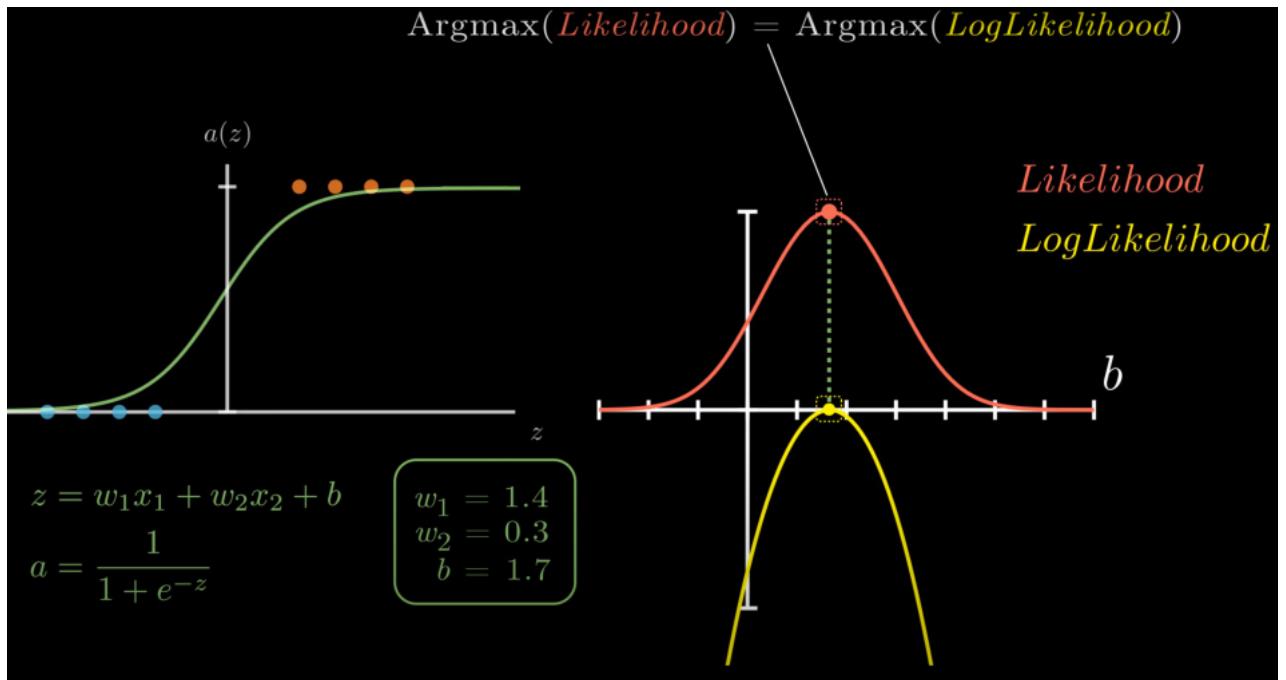
$$L = 0.7 \times 0.8 \times 0.95 \times 0.8 \times 0.9 \times 0.75 \times \dots \times 0.8 = 0.000\dots$$

$$\begin{aligned} \log(L) &= \log\left(\prod_{i=1}^m a_i^{y_i} \times (1 - a_i)^{1-y_i}\right) \\ &= \log(0.7 \times 0.8 \times 0.95 \times 0.8 \times 0.9 \times 0.75) \\ &= \log(0.7) + \log(0.8) + \log(0.95) + \log(0.8) + \log(0.9) + \log(0.75) \\ &= -0.35 - 0.22 - 0.05 - 0.22 - 0.1 - 0.35 \\ &= -1.24 \end{aligned}$$

Et comme la fonction logarithme est une fonction monotone croissante, elle va conserver l'ordre de nos termes :



Le maximum de la vraisemblance est le même que le log du maximum de cette vraisemblance, on aura le même résultat :



Mathématiquement si on développe le tout, on aura :

$$LL = \log\left(\prod_{i=1}^m a_i^{y_i} \times (1 - a_i)^{1-y_i}\right)$$

$$= \sum_{i=1}^m \log(a_i^{y_i} \times (1 - a_i)^{1-y_i})$$

$$= \sum_{i=1}^m \log(a_i^{y_i}) + \log((1 - a_i)^{1-y_i})$$

Rappel :
 $\log(ab) = \log(a) + \log(b)$

$$= \sum_{i=1}^m y_i \log(a_i) + (1 - y_i) \log(1 - a_i)$$

Rappel :
 $\log(a^y) = y \log(a)$

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m y_i \log(a_i) + (1 - y_i) \log(1 - a_i)$$

Log Loss

⋮

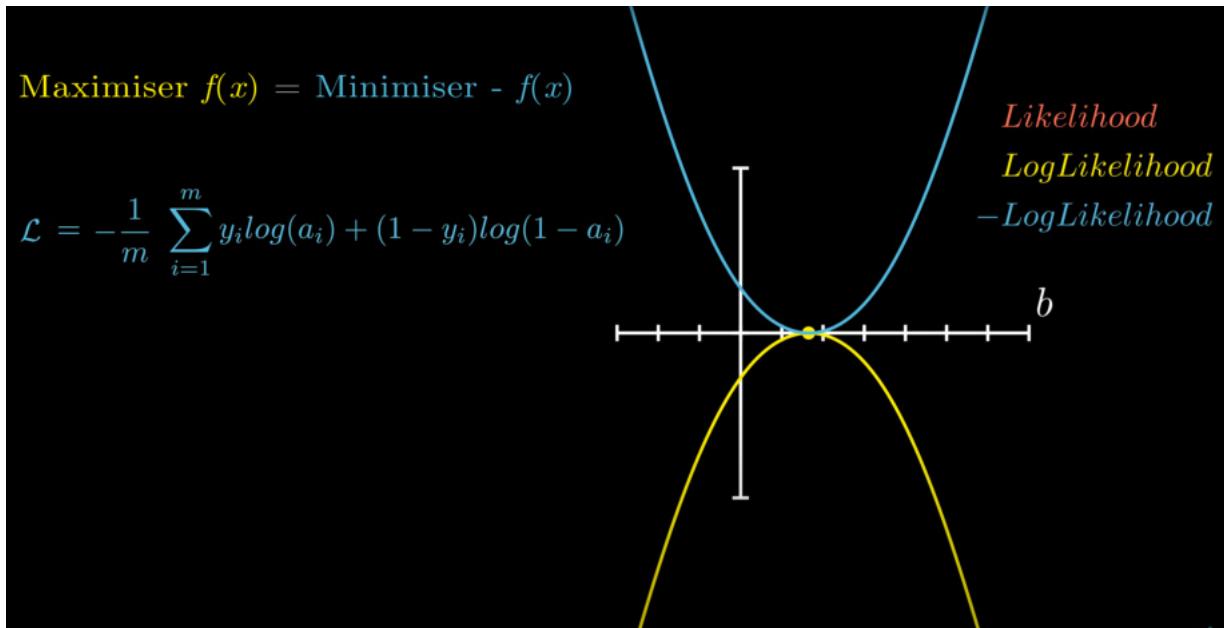
On voit que le résultat final ressemble beaucoup à l'équation du Log Loss.

Il manque juste le $-1/m$ mais c'est normal.

Jusqu'à maintenant on a calculé le log de la vraisemblance, et la vraisemblance on cherche à la maximiser pour avoir le meilleur modèle possible.

Or en mathématique les algorithmes de maximisation n'existe pas. En général on utilise des algorithmes de minimisation.

Mais maximiser une fonction $f(x)$ ça revient à faire l'inverse de la minimisation de $f(x)$, donc max de $f(x) = \min - f(x)$.

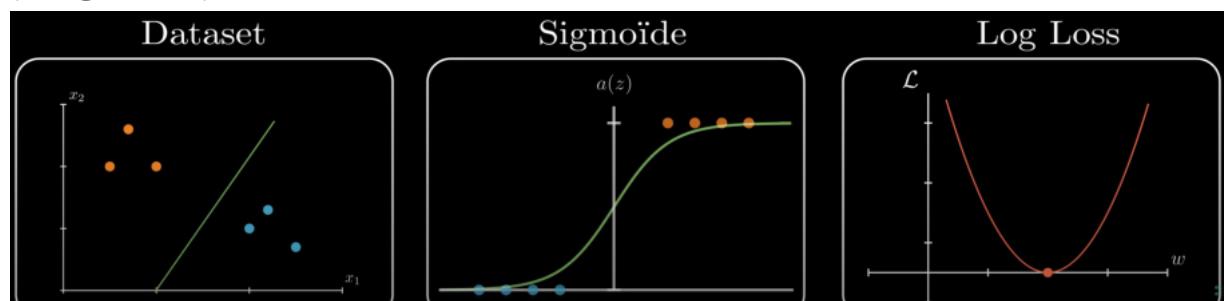


C'est pour ça qu'on a le $-$ au début et le $1/m$ est juste un petit facteur pour normaliser notre résultat.

Le Gradient Descent :

C'est l'un des algorithmes d'apprentissage les plus utilisés en ML et DL.

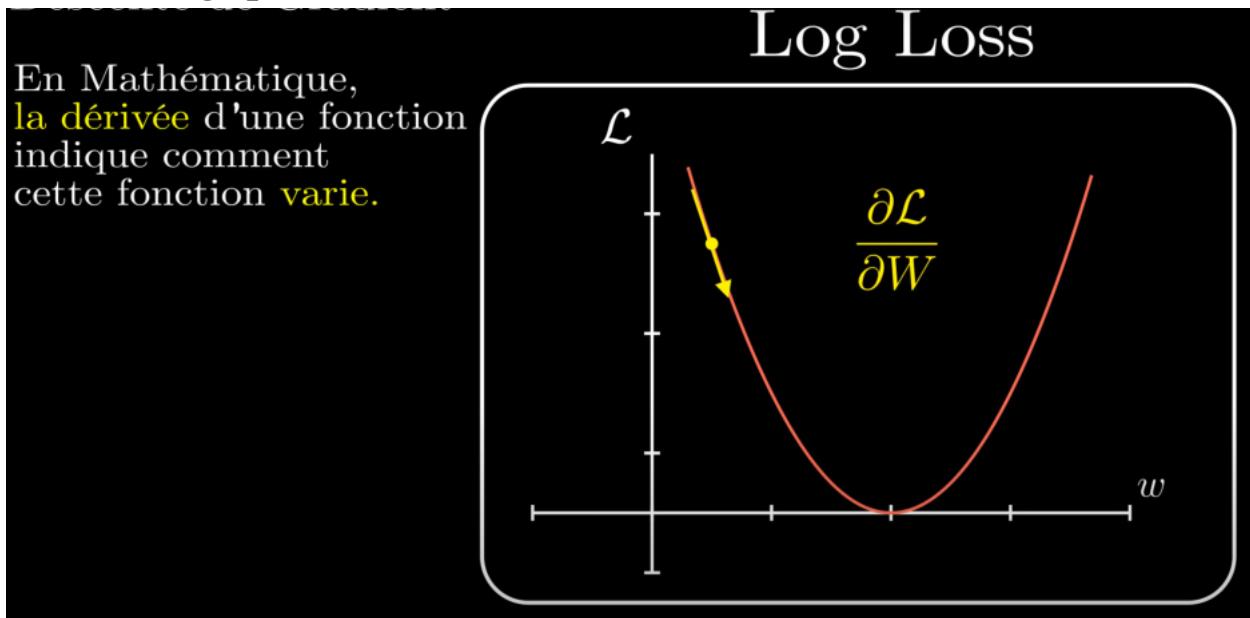
Il consiste à ajuster les paramètres w et b de façon à minimiser les erreurs du modèle, c-a-d à minimiser la Fonction Coût (Log Loss).



Pour ce faire, il faut déterminer **comment est-ce-que cette fonction varie en fonction des différents paramètres**.

C'est pourquoi on calcule le **Gradient** (= la dérivée) de la **Fonction Coût**.

Et c'est logique car :



Si la dérivé est négative ça nous indique que la fonction diminue quand w augmente, et qu'il va falloir augmenter w si l'on veut réduire nos erreurs, à l'inverse, si la dérivé est positive, cela indique que la Fonction Coût augmente quand w augmente, il faut alors diminuer w .

Pour faire ça on va utiliser cette formule :

$$W_{t+1} = W_t - \alpha \frac{\partial \mathcal{L}}{\partial W_t}$$

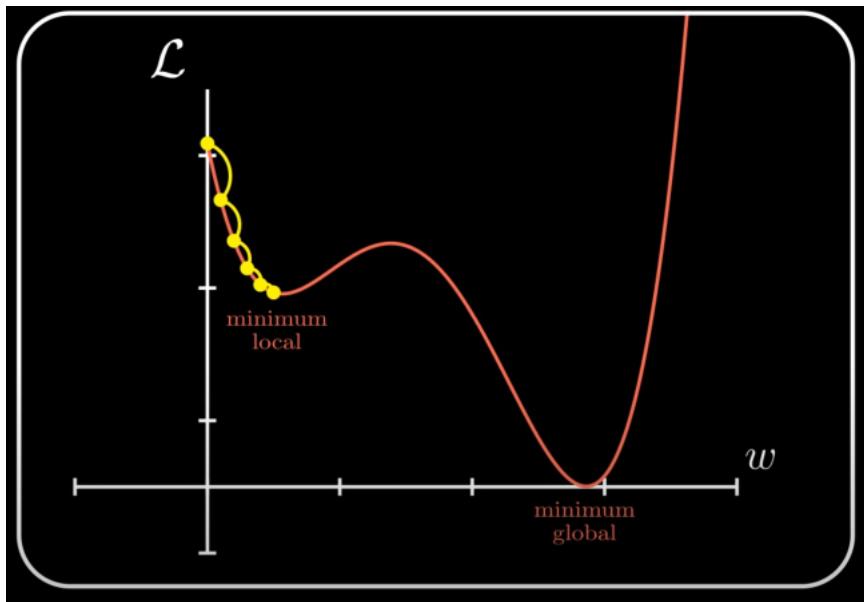
W_{t+1} : Paramètre W à l'instant $t+1$

W_t : Paramètre W à l'instant t

α : Pas d'apprentissage positif

$\frac{\partial \mathcal{L}}{\partial W_t}$: Gradient à l'instant t

Il faut par contre avoir une **Fonction Convexe**, donc une fonction qui ne contient qu'un seul minimum, s'il existe des minimum local cela ne sera pas possible.



On voit ici sur cette Fonction qui **n'est pas Convexe**, que l'algorithme de Descente de Gradient **est bloqué dans un minimum local**.

Mais la Fonction Coût Log Loss est une fonction Convexe.

Formule Mathématique à retenir et à comprendre :

$$z = w_1x_1 + w_2x_2 + b$$

$$a = \frac{1}{1 + e^{-z}}$$

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m y_i \log(a_i) + (1 - y_i) \log(1 - a_i)$$

$$W = W - \alpha \frac{\partial \mathcal{L}}{\partial W}$$

On a ici d'abord notre Fonction Linéaire, notre Fonction Sigmoïde, notre Fonction Coût et notre Descente de Gradient.

La Vectorisation

En programmation, cela consiste à mettre nos données dans des **vecteurs**, des **matrices** ou des **tableaux à N-dimension** afin d'effectuer des opérations mathématiques sur l'ensemble de ces données.

Exemple :
multiplier les éléments d'une liste

A la place :
Multiplier un vecteur tout entier

$$\begin{bmatrix} 4, & 6, & 2, & \dots, & 7 \end{bmatrix} \times 2$$
$$\begin{bmatrix} 8, & 12, & 4, & \dots, & 14 \end{bmatrix}$$

$$\begin{bmatrix} 4 \\ 6 \\ 2 \\ \vdots \\ 7 \end{bmatrix} \times 2 = \begin{bmatrix} 8 \\ 12 \\ 4 \\ \vdots \\ 14 \end{bmatrix}$$

C'est bien plus simple et efficace de multiplier un vecteur par un coefficient multiplicateur que de multiplier ce coefficient un à un pour chaque élément dans une liste.

```
liste_A = [4, 6, 2, 7]  
  
liste_B = [i * 2 for i in liste_A]
```

```
A = np.array([4, 6, 2, 7])  
  
B = A * 2
```

Code plus simple
Execution plus rapide

En plus de ça, le code est plus simple.

La vectorisation est une technique indispensable à connaître en ML et en DL. Car il y a de très grosses quantités de données de ces domaines.

On va convertir nos équations vu précédemment pour les adapter en calcul matriciel mais avant voyons un petit rappel sur le calcul matriciel :

Les Matrices sont des **tableaux à 2 dimensions** dont on se sert pour résoudre **facilement et rapidement** une **grande quantité de problèmes** mathématiques.

$$\begin{matrix} & 1 & 2 & \dots & n \\ \begin{matrix} 1 \\ 2 \\ \vdots \\ m \end{matrix} & \left[\begin{matrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{matrix} \right] \end{matrix} \in \mathbb{R}^{m \times n}$$

exemple de matrice de dimension (m, n)
(m lignes, n colonnes)

En DeepLearning voici 3 opérations élémentaires à connaître sur le calcul matriciel :

1- Additions et Soustractions

2- Les Transposées

3- Les Multiplications

1-Additions et Soustractions :

Pour additionner ou soustraire 2 matrices, il faut que leurs **dimensions soient égales**.

$$\overbrace{\begin{bmatrix} 2 & 3 \\ 1 & 4 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 1 \\ 2 & 3 \\ 1 & 0 \end{bmatrix}}^{\text{mêmes dimensions}} = \begin{bmatrix} 5 & 4 \\ 3 & 7 \\ 4 & 1 \end{bmatrix}$$

$$\overbrace{\begin{bmatrix} 2 & 3 \\ 1 & 4 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}}^{\text{dimensions différentes}} = \text{impossible}$$

2-Transposition :

Consiste à faire pivoter la matrice sur **sa diagonale**, ce qui a pour effet d'interchanger ses dimensions (le **nombre de lignes** devient le **nombre de colonnes** et inversement).

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \in \mathbb{R}^{4 \times 3}$$

$$A^T = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix} \in \mathbb{R}^{3 \times 4}$$

3-Multiplication Matricielle :

Pour multiplier ensemble 2 matrices, il ne s'agit pas de multiplier leurs termes un-à-un comme pour les additions / soustractions.

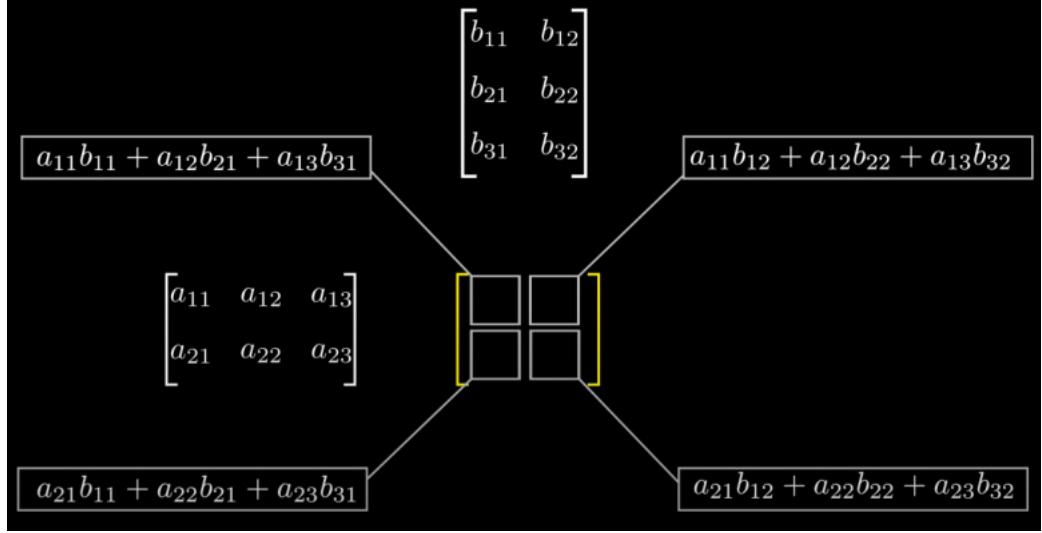
A la place, on effectue une **combinaison linéaire** entre les lignes de la matrice de gauche et les colonnes de la matrice de droite.

Ainsi, pour multiplier 2 matrices, il faut que le **nombre de colonnes** de la première soit égal au **nombre de lignes** de la deuxième :

$$\underbrace{\begin{bmatrix} A \\ B \end{bmatrix}}_{\substack{2 \times 3}} \cdot \underbrace{\begin{bmatrix} C \\ D \end{bmatrix}}_{\substack{3 \times 2}} = \underbrace{\begin{bmatrix} E \\ F \end{bmatrix}}_{\substack{2 \times 2}}$$

The diagram shows two matrices, A and B, being multiplied by another matrix C and D. Matrix A has 2 rows and 3 columns, indicated by a green bracket below it. Matrix B has 2 rows and 2 columns, indicated by a pink bracket below it. The result of the multiplication is a 2x2 matrix, indicated by a blue bracket below it. Below the matrices, their dimensions are written: 2 x 3, 3 x 2, and 2 x 2. Green and pink curved arrows point from the respective brackets to the corresponding dimensions of the matrices.

Le résultat du produit est une **combinaison linéaire** entre les lignes de la matrice de gauche et les colonnes de celle de droite.



Maintenant, Vectorisation de nos équations :
Rappel de nos équations précédentes

$$z = w_1x_1 + w_2x_2 + b$$

Modèle

$$a = \frac{1}{1 + e^{-z}}$$

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \times \log(a^{(i)}) + (1 - y^{(i)}) \times \log(1 - a^{(i)})$$

Fonction Coût

$$w_1 = w_1 - \alpha \frac{\partial \mathcal{L}}{\partial w_1} \quad \left(\frac{\partial \mathcal{L}}{\partial w_1} = \frac{1}{m} \sum_{i=1}^m (a^{(i)} - y^{(i)})x_1 \right)$$

$$w_2 = w_2 - \alpha \frac{\partial \mathcal{L}}{\partial w_2} \quad \left(\frac{\partial \mathcal{L}}{\partial w_2} = \frac{1}{m} \sum_{i=1}^m (a^{(i)} - y^{(i)})x_2 \right)$$

$$b = b - \alpha \frac{\partial \mathcal{L}}{\partial b} \quad \left(\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum_{i=1}^m (a^{(i)} - y^{(i)}) \right)$$

Descente de Gradient

Vectorisation de z :

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} \end{bmatrix}$$

$$Z = \begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(m)} \end{bmatrix} = \underbrace{\begin{bmatrix} w_1 x_1^{(1)} + w_2 x_2^{(1)} + b \\ w_1 x_1^{(2)} + w_2 x_2^{(2)} + b \\ \vdots \\ w_1 x_1^{(m)} + w_2 x_2^{(m)} + b \end{bmatrix}}_{\substack{(m, 2) \\ (m, 1)}} = \underbrace{\begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} \end{bmatrix}}_{(m, 1)} \cdot \underbrace{\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}}_{(2, 1)} + \underbrace{\begin{bmatrix} b \\ b \\ \vdots \\ b \end{bmatrix}}_{(m, 1)}$$

Rappel :

$$z^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + b$$

Donc la formule final de l'équation Z sera :

$$Z = X \cdot W + b$$

Vectorisation de a :

$$A = \begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix} = \sigma \left(\begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(m)} \end{bmatrix} \right) = \sigma(Z)$$

Rappel :

$$a^{(i)} = \sigma(z^{(i)}) = \frac{1}{1 + e^{-z^{(i)}}}$$

Vectorisation de la Fonction Coût :

Ici on veut comparer directement le vecteur A au vecteur y.

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \times \log(a^{(i)}) + (1 - y^{(i)}) \times \log(1 - a^{(i)})$$

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \times \log \left(\begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix} \right) + \left(1 - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \right) \times \log \left(1 - \begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix} \right)$$

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \begin{bmatrix} y^{(1)} \times \log(a^{(1)}) + (1 - y^{(1)}) \times \log(1 - a^{(1)}) \\ y^{(2)} \times \log(a^{(2)}) + (1 - y^{(2)}) \times \log(1 - a^{(2)}) \\ \vdots \\ y^{(m)} \times \log(a^{(m)}) + (1 - y^{(m)}) \times \log(1 - a^{(m)}) \end{bmatrix}$$

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \boxed{y^{(1)} \times \log(a^{(1)}) + (1 - y^{(1)}) \times \log(1 - a^{(1)}) + y^{(2)} \times \log(a^{(2)}) + (1 - y^{(2)}) \times \log(1 - a^{(2)}) + \dots + y^{(m)} \times \log(a^{(m)}) + (1 - y^{(m)}) \times \log(1 - a^{(m)})}$$

↑

La somme de ce vecteur donne un nombre réel

On a le résultat de la somme qui donne un nombre réel ce qui est logique car vu que la **Fonction Coût** nous donne un **coût**, c-a-d une **mesure de l'erreur** de notre modèle.

$$\mathcal{L} = -\frac{1}{m} \sum y \times \log(A) + (1 - y) \times \log(1 - A)$$

Vectorisation de la Descente de Gradient :

$$w_1 = w_1 - \alpha \frac{\partial \mathcal{L}}{\partial w_1}$$

$$w_2 = w_2 - \alpha \frac{\partial \mathcal{L}}{\partial w_2}$$

$$b = b - \alpha \frac{\partial \mathcal{L}}{\partial b}$$

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \end{bmatrix}$$

$$W = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \quad \frac{\partial \mathcal{L}}{\partial W} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \end{bmatrix}$$

Jacobien (excluant b)

$$b = b - \alpha \frac{\partial \mathcal{L}}{\partial b}$$

$$W = W - \alpha \frac{\partial \mathcal{L}}{\partial W}$$

$$W = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \quad \frac{\partial \mathcal{L}}{\partial W} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \end{bmatrix}$$

Jacobien (excluant b)

note : D'un point de vue mathématique, la notation la plus correcte est $W_{t+1} = W_t - \alpha \frac{\partial \mathcal{L}}{\partial W_t}$
 La notation utilisée ici est celle que nous emploierons pour la partie programmation

$$b = b - \alpha \frac{\partial \mathcal{L}}{\partial b}$$

Rappel : b est un nombre réel (pas un vecteur).

Donc la dérivée partielle $\frac{\partial \mathcal{L}}{\partial b}$ est aussi un nombre réel.

Pas besoin de vectoriser cette équation.

Vectorisation des Gradients :

$$\begin{aligned}
 \frac{\partial \mathcal{L}}{\partial W} &= \underbrace{\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \end{bmatrix}}_{(2, 1)} = \frac{1}{m} \begin{bmatrix} (a^{(1)} - y^{(1)})x_1^{(1)} + (a^{(2)} - y^{(2)})x_1^{(2)} + \dots + (a^{(m)} - y^{(m)})x_1^{(m)} \\ (a^{(1)} - y^{(1)})x_2^{(1)} + (a^{(2)} - y^{(2)})x_2^{(2)} + \dots + (a^{(m)} - y^{(m)})x_2^{(m)} \end{bmatrix} \\
 &= \frac{1}{m} \underbrace{\begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(m)} \end{bmatrix}}_{(2, m)} \cdot \underbrace{\begin{pmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{pmatrix} - \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}}_{(m, 1)}
 \end{aligned}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{m} X^T \cdot (A - y)}$$

Maintenant pour b , ce paramètre étant un nombre réel (et non un vecteur) sa dérivée est elle aussi un nombre réel.

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum_{i=1}^m (a^{(i)} - y^{(i)})$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum_{i=1}^m \left(\begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix} - \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix} \right)$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum_{i=1}^m (a^{(1)} - y^{(1)} + a^{(2)} - y^{(2)} + \dots + a^{(m)} - y^{(m)})}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum (A - y)}$$

Résumé de la vectorisation de nos équations :

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} \end{bmatrix} \in \mathbb{R}^{m \times 2} \quad W = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \in \mathbb{R}^{2 \times 1} \quad b \in \mathbb{R}$$

$$Z = X \cdot W + b$$

$$A = \frac{1}{1 + e^{-Z}}$$

Modèle

$$\mathcal{L} = -\frac{1}{m} \sum y \times \log(A) + (1 - y) \times \log(1 - A)$$

Fonction Coût

$$W = W - \alpha \frac{\partial \mathcal{L}}{\partial W} \quad \left(\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{m} X^T \cdot (A - y) \right)$$

$$b = b - \alpha \frac{\partial \mathcal{L}}{\partial b} \quad \left(\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum (A - y) \right)$$

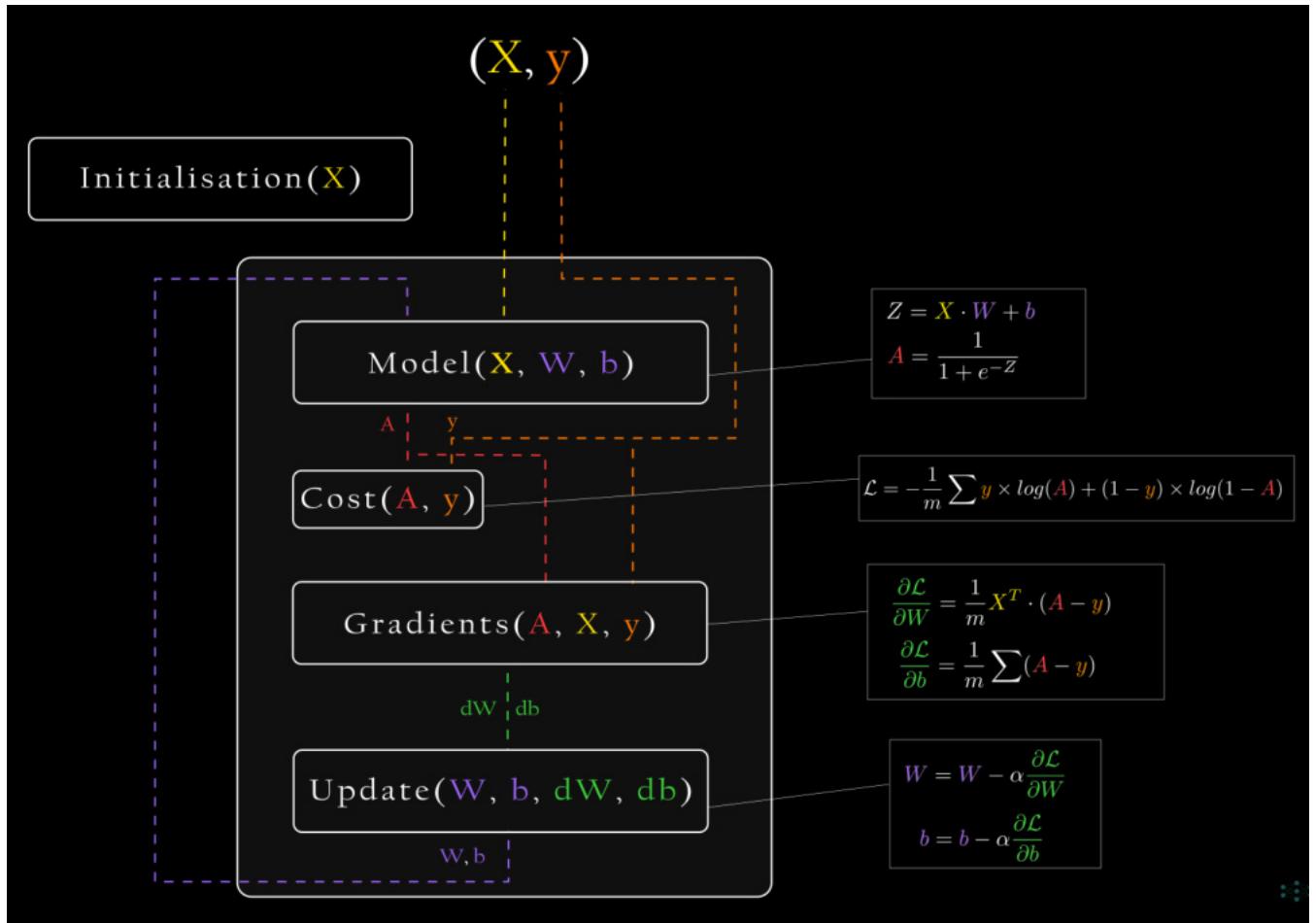
Descente de Gradient

Et grâce à cette vectorisation, on a des équations universelles, si on rajoute d'autres éléments **x1, x2..x20..** cela reviendra au même les équations ne changent pas, juste ici si on prend **x20** on aura une matrice de **dimension x de (m, 20)** multiplié par **w** (les poids) **(20, 1)** ce qui donnera toujours la **dimension (m, 1)**.

Et pour les gradients rien ne changera si ce n'est pas que les paramètres iront de **w1 jusqu'à w20** et le gradient sera toujours le même mais avec la **transposé de X qui sera de (20, m)** multiplié par **A-y qui est de dimension (m, 1)** donc le **résultat sera de dimension (20, 1)**.

Grâce à la vectorisation on peut maintenant traiter toutes nos données en une seule opération sans avoir à passer par des boucles for, mais aussi de traiter des problèmes avec autant de variables que nécessaires.

Programmation d'un neurone artificiel :



Ici on va générer un simple Dataset (X, y) pour l'exemple :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
```

```
X, y = make_blobs(n_samples=100, n_features=2, centers=2,
random_state=0)
y = y.reshape((y.shape[0], 1))
```

```
print('dimension de X :', X.shape)
print('dimension de y :', y.shape)
```

```
plt.scatter(X[:,0], X[:,1], c=r, cmap='summer')
```

```
plt.show()
```

Création de la fonction d'initialisation :

```
def initialisation(X):
    W = np.random.randn(X.shape[1], 1)
    b = np.random.randn(1)
    return [W, b]
```

Création de la fonction de notre modèle :

```
def model(X, W, b):
    Z = X.dot(W) + b
    A = 1 / (1 + np.exp(-Z))
    return A
```

Création de la fonction Coût :

```
def log_loss(A, y):
    return 1 / len(y) * np.sum([-y * np.log(A) - (1 - y) * np.log(1 - A)])
```

Création de la fonction des Gradients :

```
def gradients(A, X, y):
    dW = 1 / len(y) * np.dot(X.T, A - y)
    db = 1 / len(y) * np.sum(A - y)
    return [dW, db]
```

Création de la fonction Update :

```
def update(dW, db, W, b, learning_rate):
    W = W - learning_rate * dW
    b = b - learning_rate * db
    return [W, b]
```

Création de la fonction Predict :

```
def predict(X, W, b):
    A = model(X, W, b)
    print(A)
    return A >= 0.5
```

Assemblage final :

```
def artificial_neuron(X, y, learning_rate = 0.1, n_iter = 100):
    # initialisation W, b
    W, b = initialisation(X)

    Loss = []

    for i in range(n_iter):
        A = model(X, W, b)
        Loss.append(log_loss(A, y))
        dW, db = gradients(A, X, y)
        W, b = update(dW, db, W, b, learning_rate)

    y_pred = predict(X, W, b)
    print(accuracy_score(y, y_pred))

    plt.plot(Loss)
    plt.show()

    return [W, b]
```

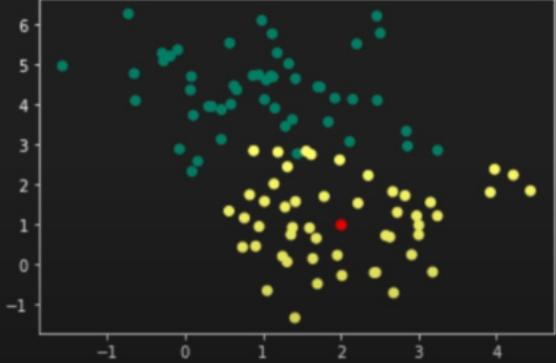
(ne pas oublier d'importer avant : from sklearn.metrics import accuracy_score)

On retourne les paramètres **W** et **b** qu'on va obtenir, et s'ils sont de qualités on peut **sauvegarder ses paramètres** et les réutiliser dès que besoin sans avoir besoin de ré-entraîner le modèle pour faire des prédictions.

On a juste ici à utiliser la fonction de prédiction par exemple avec un plante qui aurait comme propriété (2,1) on la passe dans notre modèle et on a la prédiction avec le seuil établit au préalable.

```
new_plant = np.array([2, 1])
plt.scatter(X[:,0], X[:, 1], c=y, cmap='summer')
plt.scatter(new_plant[0], new_plant[1] , c='r')
plt.show()
predict([new_plant, W, b])
```

✓ 0.1s



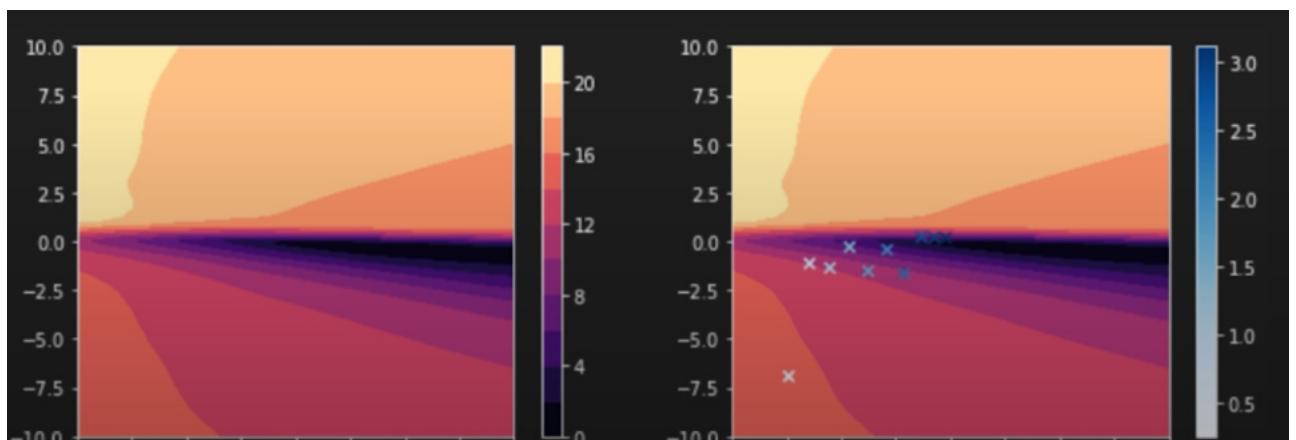
La Normalisation de Données

Cela signifie de mettre à même échelle toutes les variables d'un Dataset afin d'éviter que les plus grandes ne viennent écraser les plus petites.

Quand les variables sont sur une même échelle, la fonction Coût évolue de façon similaire sur tous ses paramètres. Cela permet une bonne convergence de l'algorithme du Gradient Descent.

Car sans la normalisation, si une variable est plus imposante que l'autre, alors la fonction Coût est compressée, et ça aura un impact sur la sortie $A(Z)$ et donc complique la convergence de la Gradient Descent.

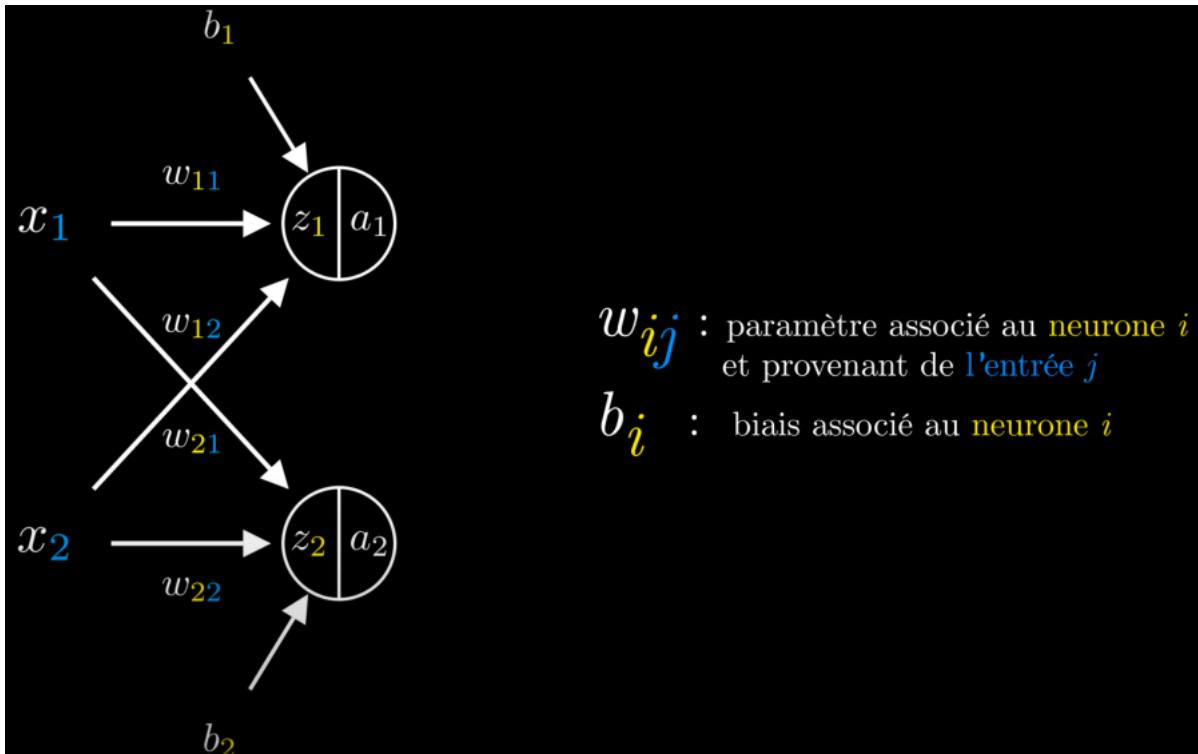
Illustration d'une Fonction Coût compressé à cause de la non normalisation des variables du Dataset :



Il existe plusieurs façons de normaliser, ici voyons la MinMax :
Ici on va mettre toutes nos variables sur une échelle de [0 à 1]

$$X = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Création d'un réseau de neurones :

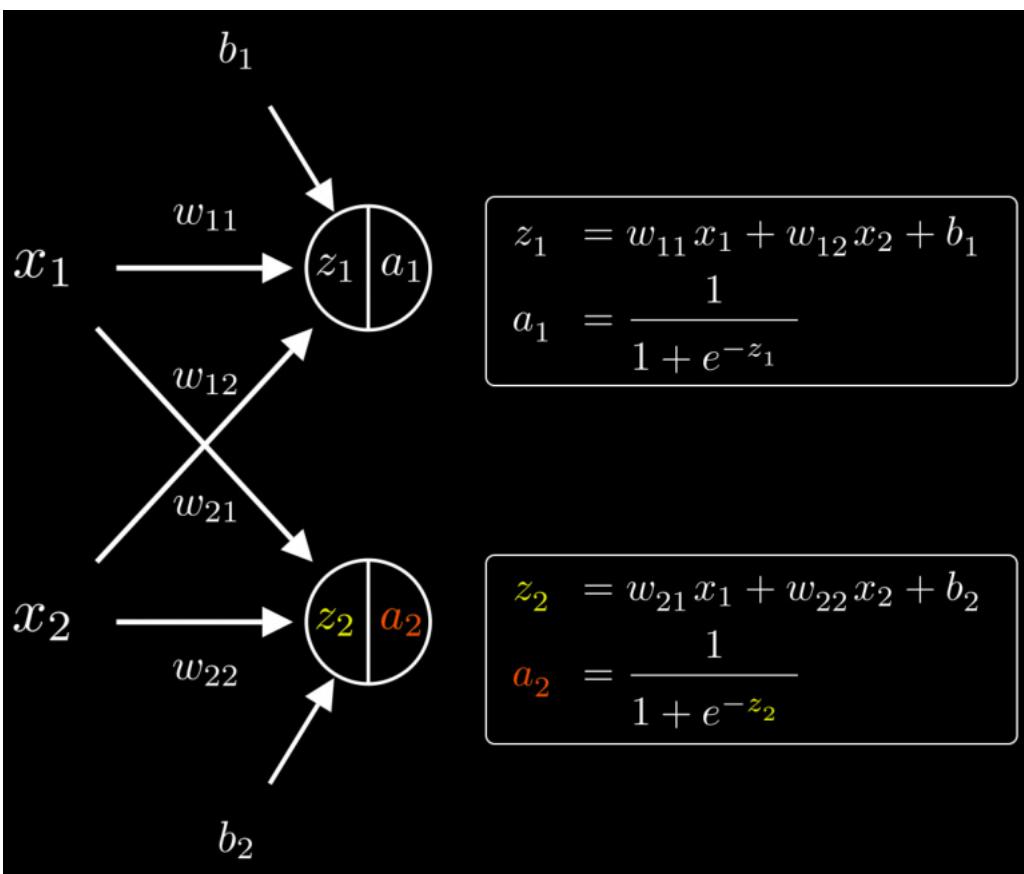


w_{ij} : paramètre associé au neurone i et provenant de l'entrée j

b_i : biais associé au neurone i

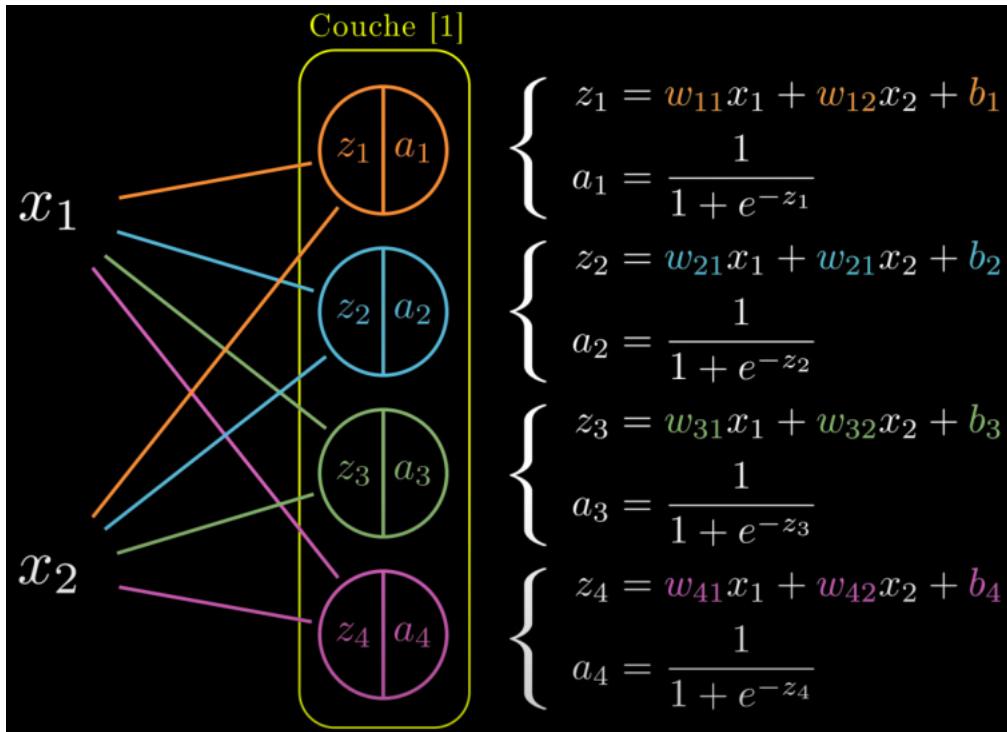
Ici on a 2 neurones avec leurs connexions.

On aura donc :



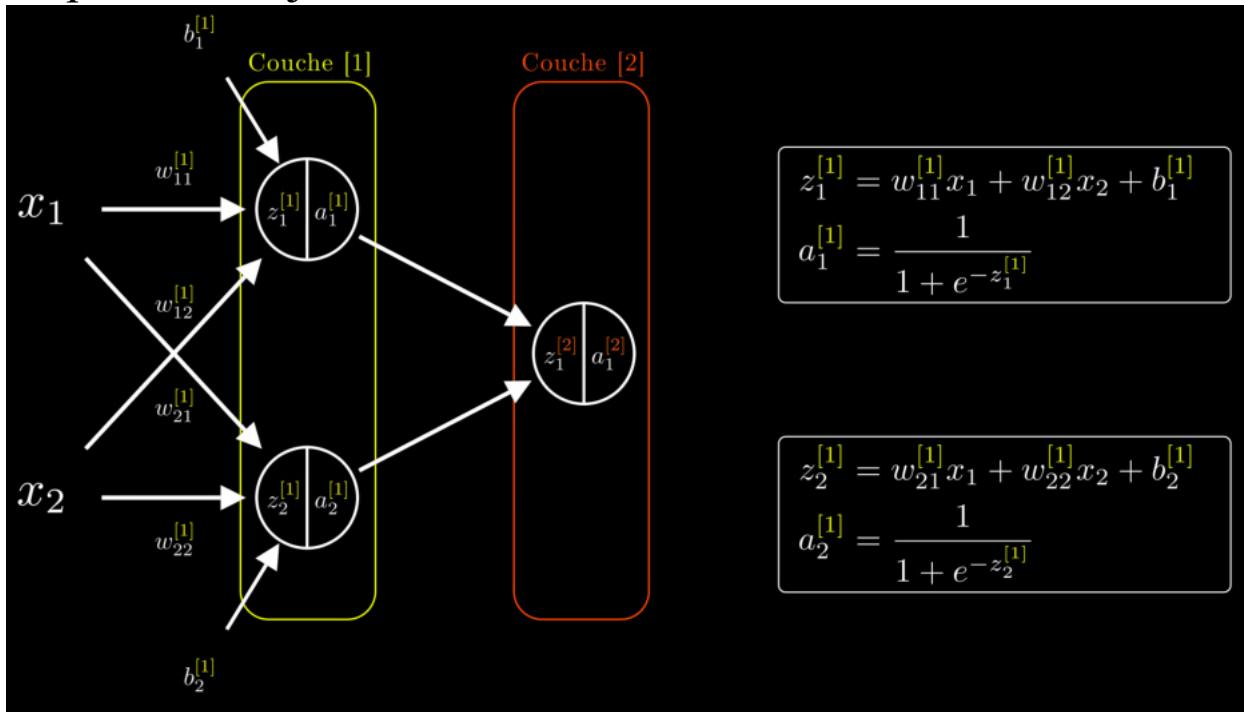
Et à présent on a notre premier réseau de neurone à 1 couche !

On peut rajouter autant de neurones qu'on désire dans cette couche :



La puissance d'un réseau augmente avec le nombre de neurones ajoutés mais ralenti le modèle !

On peut aussi ajouter une autre couche :



On va différencier les couches ici avec les [1] & [2].

Relation et équations entre la couche 1 et 2 :

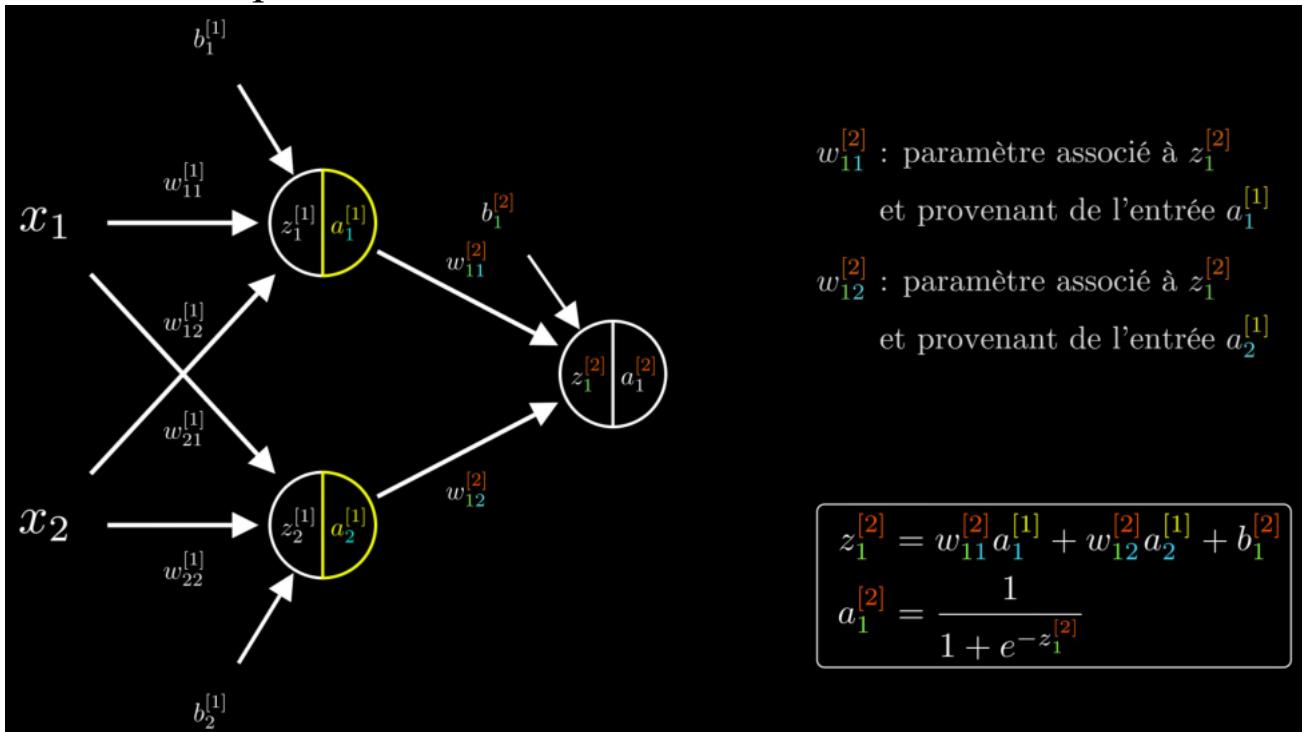
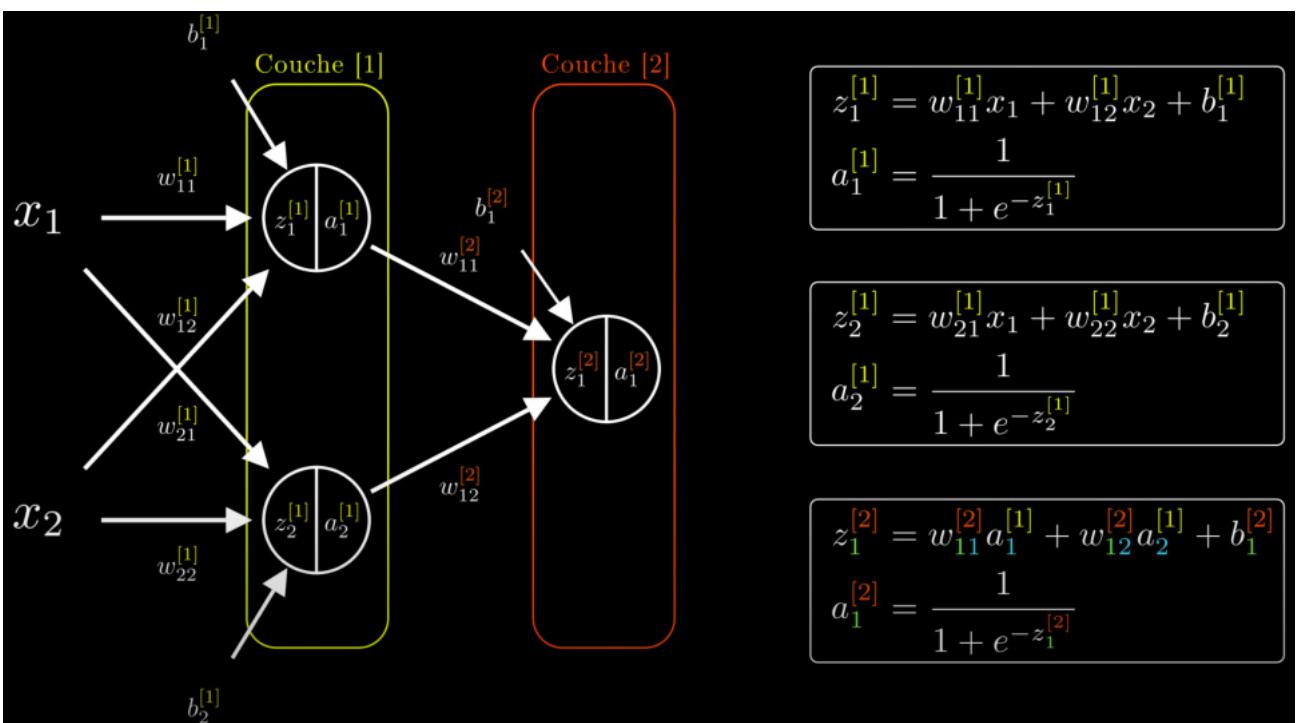
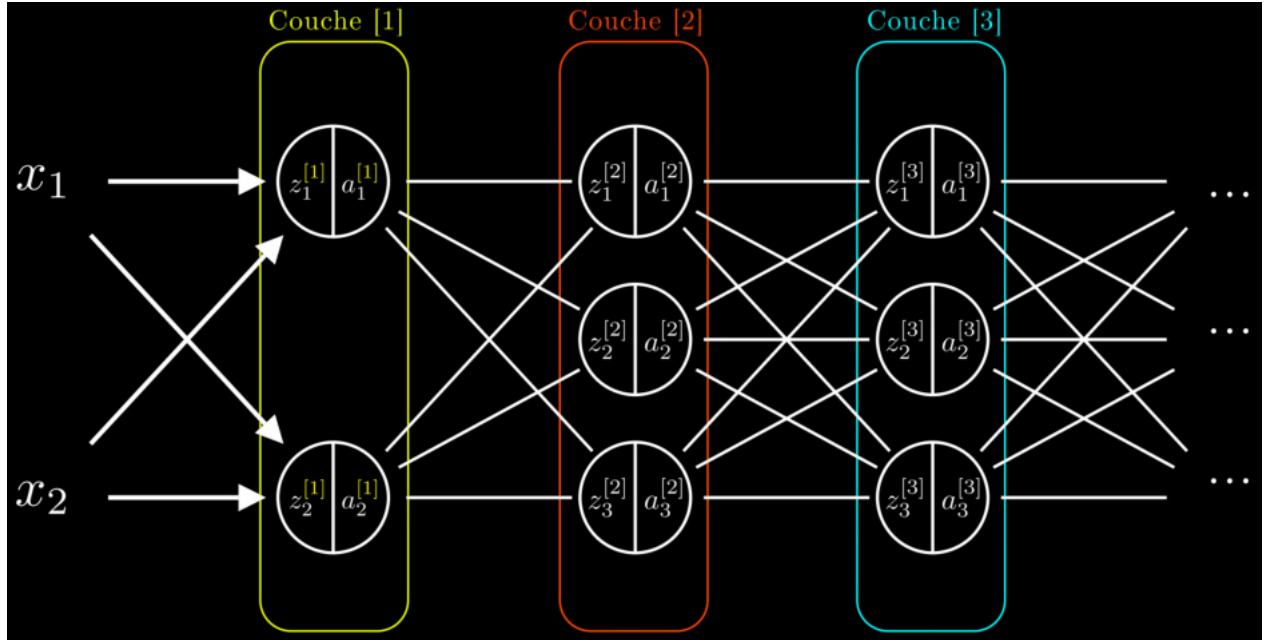


Illustration final de notre réseau à 2 couches :



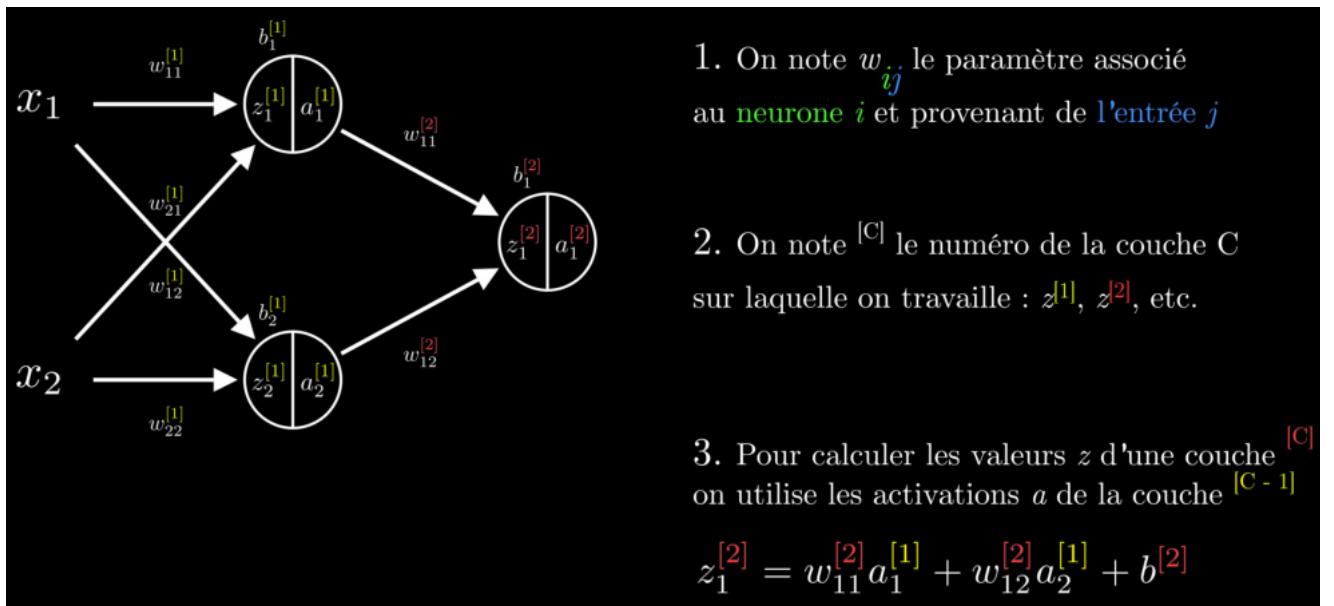
Et chaque couche va être construite sur le même principe, c-a-d en prenant en entrée les activations des couches précédentes et va les multiplier par les poids w et ajouter un biais (+b) :



Plus notre réseau sera **profond** (Deep Neural Network), plus il sera capable d'apprendre des choses **compliquées**. Mais cela rendra **l'apprentissage plus long**.

Il faut donc toujours trouver un juste équilibre selon le besoin spécifique.

Pour résumer, voici les 3 points les plus importants :

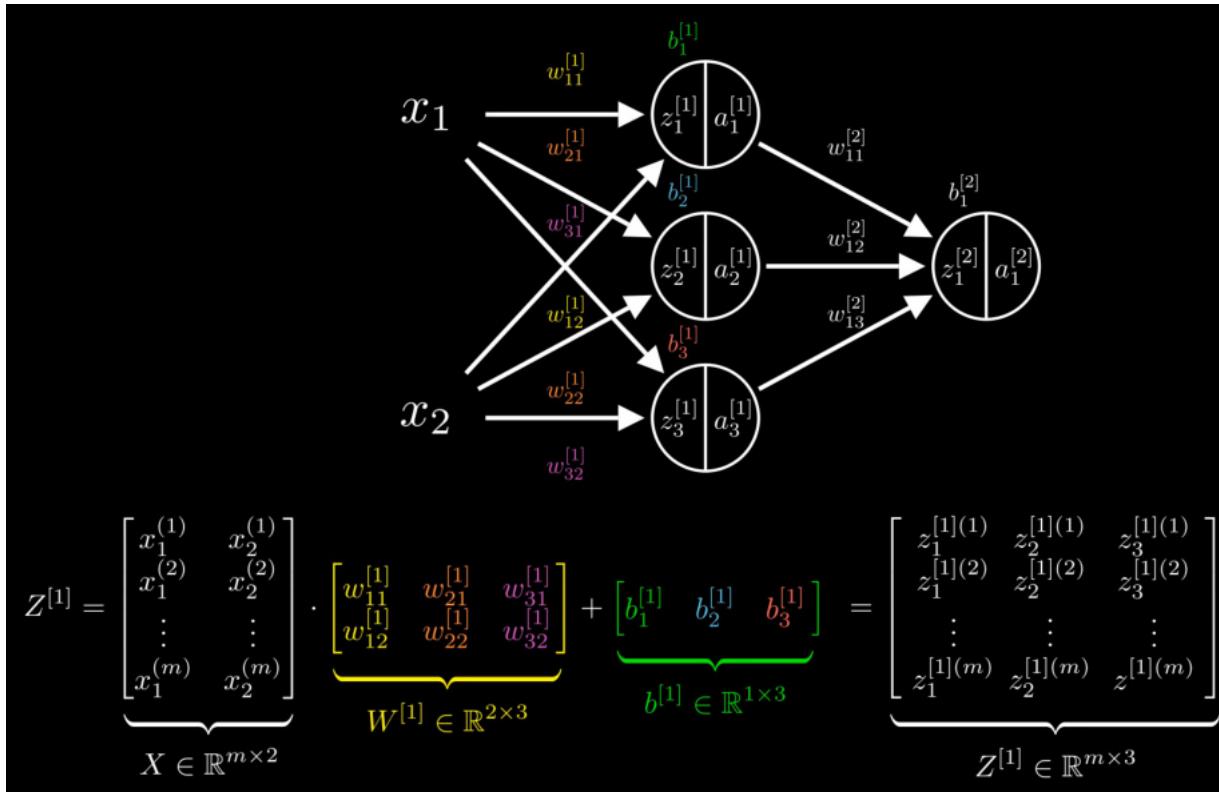


Pour implémenter de tels modèles, il n'est pas pratique d'écrire les équations de chaque neurone...

L'idéal est donc de **vectoriser** ces équations.

Afin de représenter chaque couche du réseau par des matrices.

Vectorisation d'un réseau de neurone

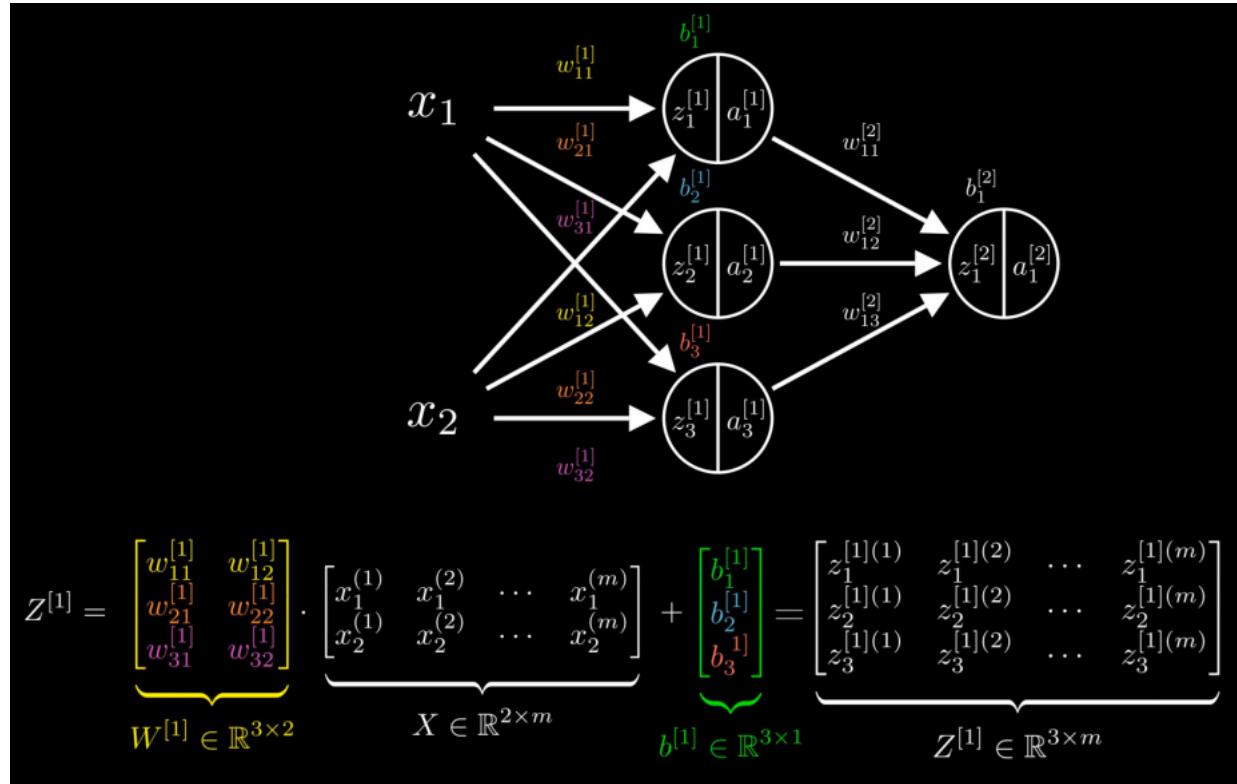


On place les coefficients **w** au sein d'une **matrice W**, puis également on place les **biais b** au sein d'un vecteur **b**, pour ensuite faire le même calcul qu'on a vu donc $\mathbf{z} = \mathbf{X} * \mathbf{W} + \mathbf{b}$, ce qui nous donne une matrice **Z[1]** de **m lignes et de 3 colonnes** (1 pour chaque neurone).

Grâce à la vectorisation on est capable de réunir le résultat de nos **2 neurones au sein d'une seule matrice** et tout ça avec **qu'un seul calcul**.

L'avantage si on veut rajouter une neurone on a juste à **rajouter une colonne à la matrice \mathbf{W}** , ainsi qu'un autre **paramètre** dans le vecteur \mathbf{b} et de part le **calcul matriciel $\mathbf{X}^*\mathbf{W}+\mathbf{b}$** on obtiendra automatiquement une **3ième colonne dans $\mathbf{Z}[1]$** qui correspondra à notre **3ième neurone**.

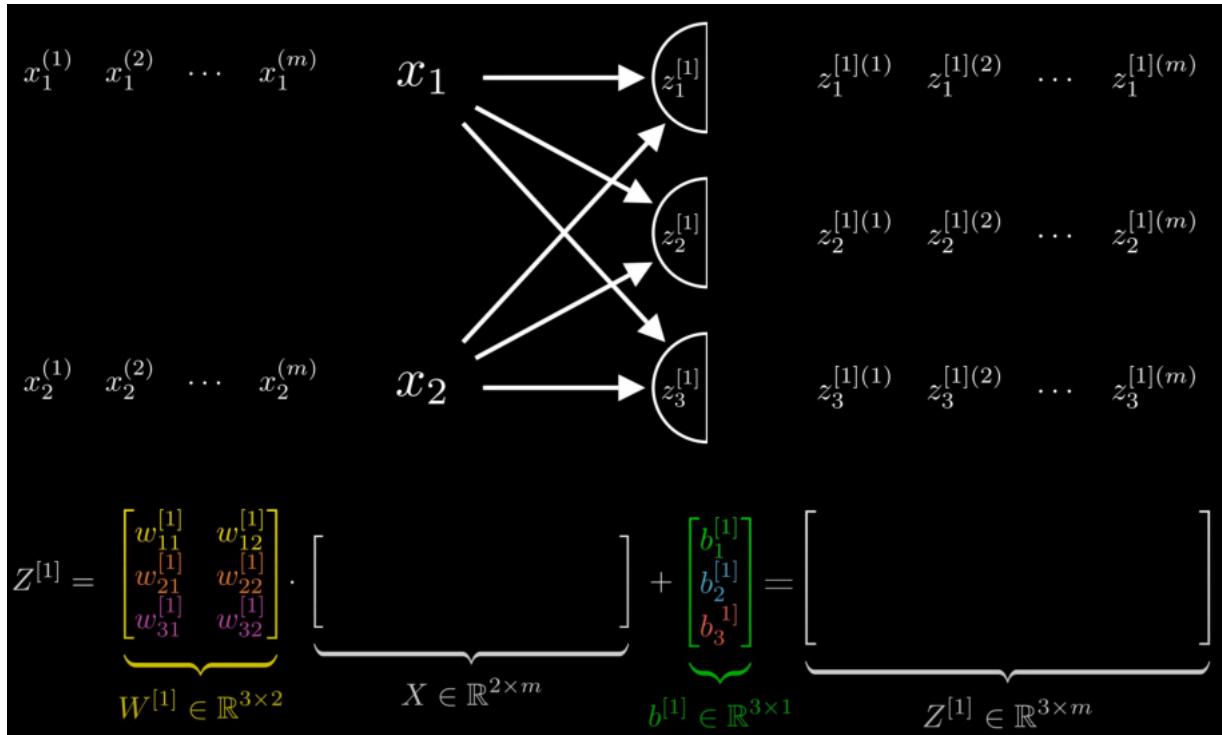
Parfois dans les livres ou autres on écrit différemment ces calculs :



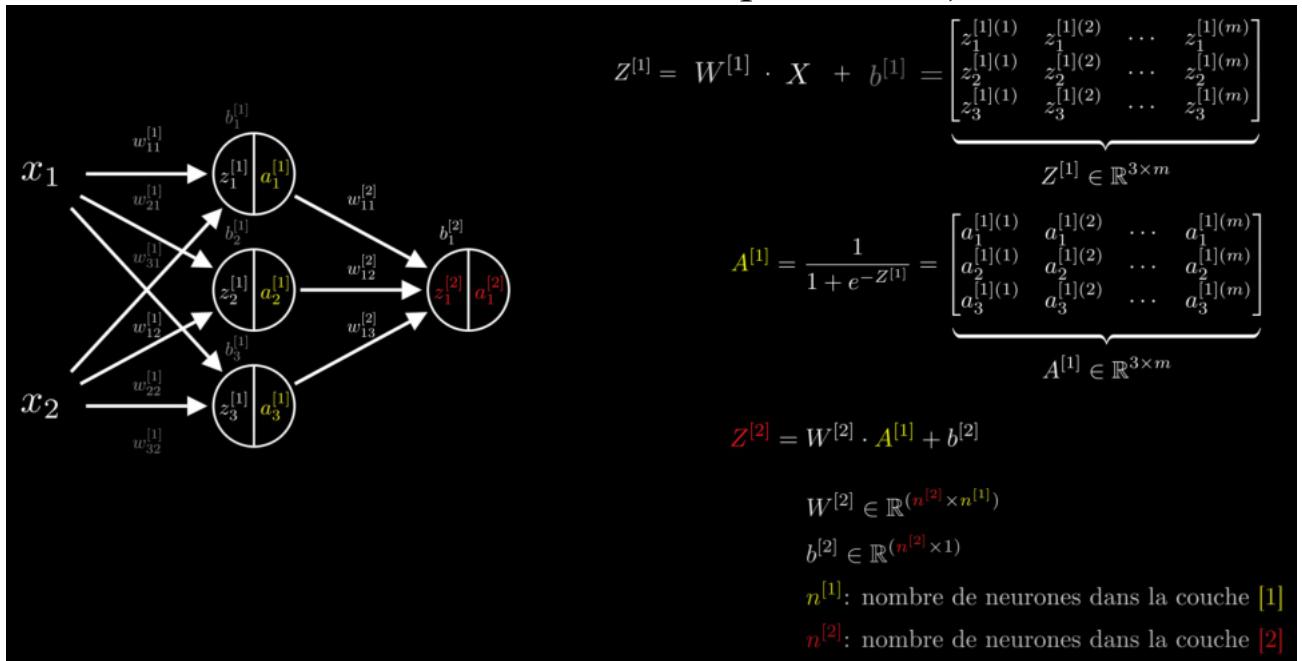
Il arrive qu'on préfère organiser les coefficients **\mathbf{W} et \mathbf{b} en lignes** (et non pas en colonne), ce qui impose de **transposer la matrice \mathbf{X}** et de réorganiser les termes de notre calcul matriciel, au lieu d'avoir **$\mathbf{X}^*\mathbf{W}+\mathbf{b}$** on aura donc **$\mathbf{W}^*\mathbf{X}+\mathbf{b}$** pour **respecter la loi des produits matriciels** et donc pour avoir des **dimensions correctes** pour effectuer ce produit.

Cela change rien à notre calcul, si ce n'est le fait de **transposer la matrice \mathbf{Z}** .

L'intérêt de faire ça c'est de pouvoir aligner chaque ligne de nos matrices avec leur neurone respectif :



Par la suite après si on a plusieurs couches et une fois qu'on a passé $Z[1]$ par exemple dans notre **fonction d'activation** alors pour $Z[2]$ il suffit de **remplacer X par A** (les fonctions d'activations de la couche de neurone précédente) :

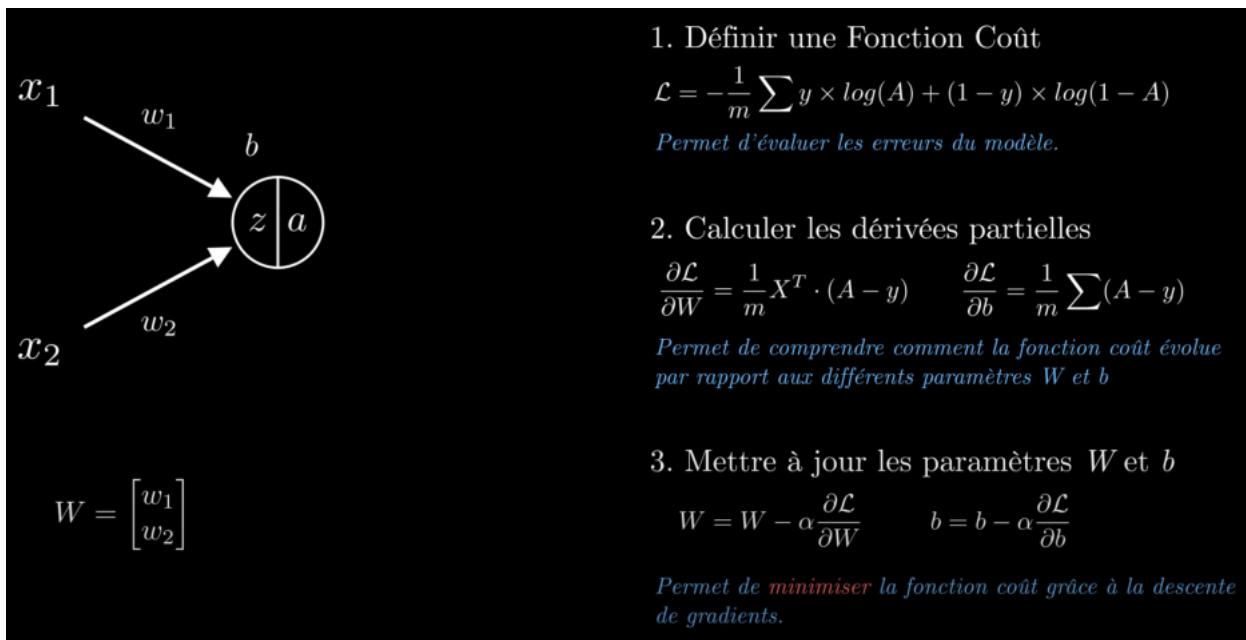


On va ensuite calculer **A[2]** (les fonctions d'activations de la nouvelle couche) **à partir de Z[2]** puis si on a d'autres couches on va calculer **Z[3]** à partir de **A[2]** et **A[3]** à partir de **Z[3]** etc etc

Et cela forme ce qu'on appelle l'étape de Forward Propagation.

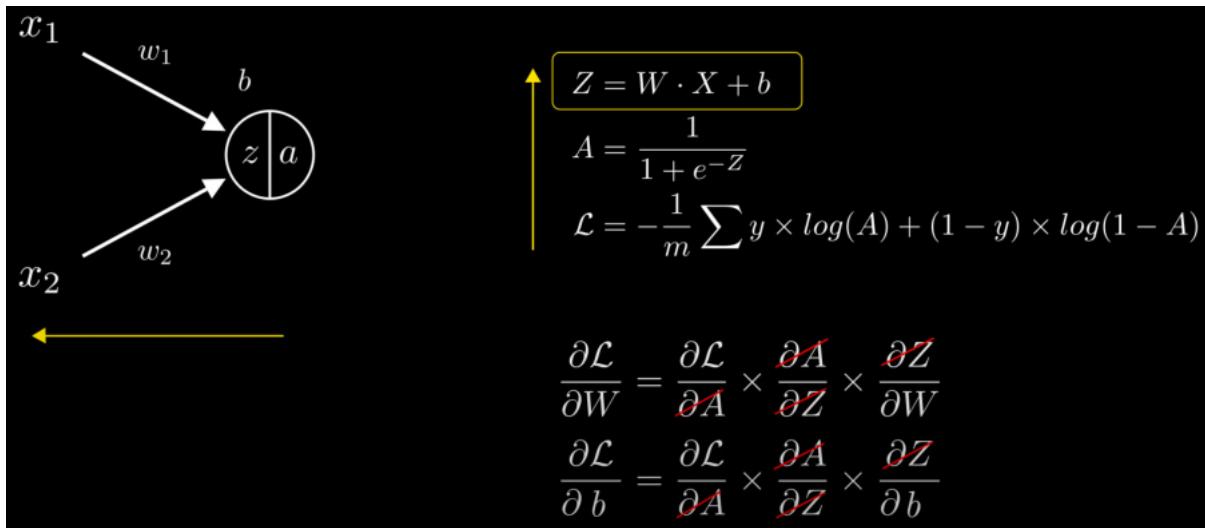
Pour l'entraînement on utilise la technique de Back-Propagation

Rappel : Entraînement d'un neurone artificiel



Dans le cas d'un réseau de neurone on fait la même chose, mais on va utiliser la technique de la **back-propagation**, cela consiste à **retracer** comment la Fonction Coût évolue de la dernière à la première couche du réseau.

C'est un petit peu comme ce qu'on a fait au début quand on a calculé les gradients d'un modèle de neurone artificiel :



On va faire pareil mais pour un réseau de neurone, donc on va calculer la **dérivée partielle de L par rapport à A[2]** puis la **dérivée partielle de A[2] par rapport à Z[2]** et pour finir la **dérivée partielle de Z[2] par rapport à W[2] et b[2]**.

Ce qui permet de calculer les **gradients de la 2ième couche**.
 Ensuite pour calculer les gradients de la première couche on va continuer à revenir en arrière, on fait **dérivée partielle de L par rapport à A[2]** puis de **A[2] par rapport à Z[2]** et ensuite afin de passer de la **2ième couche à la première**, on va calculer la **dérivée partielle de Z[2] par rapport à A[1]** !

Après il suffit de calculer la **dérivée partielle de A[1] par rapport à Z[1]** et pour finir la **dérivée partielle de Z[1] par rapport à W[1] et b[1]**.

C'est ça qu'on appelle la Back-Propagation, puisqu'on retrace tout notre calcul de la dernière équation jusqu'à la toute première, et **on calcul la dérivée partielle de chaque variable pour voir comment chaque variable évolue et impact l'apprentissage** de sorte à trouver après les meilleurs paramètres pour l'apprentissage et ainsi maximiser chaque variable pour qu'elle soit le mieux possible !

On va obtenir alors les équations de nos différents gradients :

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W^{[2]}} &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial W^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial b^{[2]}} &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial b^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial W^{[1]}} &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial W^{[1]}} \\ \frac{\partial \mathcal{L}}{\partial b^{[1]}} &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial b^{[1]}}\end{aligned}$$

Mais on peut les simplifier, on voit que les gradients de la 2ième couche commencent toujours par la même base :

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W^{[2]}} &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial W^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial b^{[2]}} &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial b^{[2]}}\end{aligned}$$

On peut alors poser une variable qui va simplifier le tout :

$$\begin{aligned}dZ2 &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial W^{[2]}} &= dZ2 \times \frac{\partial Z^{[2]}}{\partial W^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial b^{[2]}} &= dZ2 \times \frac{\partial Z^{[2]}}{\partial b^{[2]}}\end{aligned}$$

Idem pour la 1ière couche où d'ailleurs on va retrouver dans notre dZ_1 notre dZ_2 :

$$\begin{aligned}
 dZ_2 &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \\
 \frac{\partial \mathcal{L}}{\partial W^{[2]}} &= dZ_2 \times \frac{\partial Z^{[2]}}{\partial W^{[2]}} \\
 \frac{\partial \mathcal{L}}{\partial b^{[2]}} &= dZ_2 \times \frac{\partial Z^{[2]}}{\partial b^{[2]}} \\
 dZ_1 &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \\
 \frac{\partial \mathcal{L}}{\partial W^{[1]}} &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial W^{[1]}} \\
 \frac{\partial \mathcal{L}}{\partial b^{[1]}} &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial b^{[1]}}
 \end{aligned}$$

Donc simplifions :

$$\begin{aligned}
 dZ_2 &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \\
 \frac{\partial \mathcal{L}}{\partial W^{[2]}} &= dZ_2 \times \frac{\partial Z^{[2]}}{\partial W^{[2]}} \\
 \frac{\partial \mathcal{L}}{\partial b^{[2]}} &= dZ_2 \times \frac{\partial Z^{[2]}}{\partial b^{[2]}} \\
 dZ_1 &= dZ_2 \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \\
 \frac{\partial \mathcal{L}}{\partial W^{[1]}} &= dZ_1 \times \frac{\partial Z^{[1]}}{\partial W^{[1]}} \\
 \frac{\partial \mathcal{L}}{\partial b^{[1]}} &= dZ_1 \times \frac{\partial Z^{[1]}}{\partial b^{[1]}}
 \end{aligned}$$

Rappel de notion : le gradient d'une fonction c'est en fait le vecteur ou la matrice qui contient l'ensemble des dérivées partielles de cette fonction.

On va avoir après calcul, les équations finales des gradients :

$$dz_2 = (A^{[2]} - y)$$

$$\frac{\partial \mathcal{L}}{\partial w^{[2]}} = \frac{1}{m} dz_2 \cdot A^{[1]T}$$

$$\frac{\partial \mathcal{L}}{\partial b^{[2]}} = \frac{1}{m} \sum_{\text{axe}1} dz_2$$

$$dz_1 = \omega^{[2]T} \cdot dz_2 \times A^{[1]}(1 - A^{[1]})$$

$$\frac{\partial \mathcal{L}}{\partial w^{[1]}} = \frac{1}{m} dz_1 \cdot X^T$$

$$\frac{\partial \mathcal{L}}{\partial b^{[1]}} = \frac{1}{m} \sum_{\text{axe}1} dz_1$$

On a juste fait les calculs des dérivées partielles par rapport aux différentes variables en remontant tout le réseau de neurone petit à petit en **back propagation** (inverse de la forward propagation), voir sur internet le détail des calculs de matrices et d'équations si besoin.

Programmation d'un réseau de neurone à 2 couches :

Comme précédemment on va faire notre fonction d'initialisation :

```
def initialisation(n0, n1, n2):
    ...
    W1 = np.random.randn(n1, n0)
    b1 = np.random.randn(n1, 1)
    W2 = np.random.randn(n2, n1)
    b2 = np.random.randn(n2, 1)

    parametres = {
        'W1': W1,
        'b1': b1,
        'W2': W2,
        'b2': b2
    }

    return parametres
```

$n^{[0]}$: nombre d'entrées du réseau

$n^{[1]}$: nombre de neurones dans la couche 1

$n^{[2]}$: nombre de neurones dans la couche 2 (la sortie)

Maintenant notre modèle auquel on va appliquer une **forward-propagation** :

```
def forward_propagation(X, parametres):
    ...
    W1 = parametres['W1']
    b1 = parametres['b1']
    W2 = parametres['W2']
    b2 = parametres['b2']

    Z1 = W1.dot(X) + b1
    A1 = 1 / (1 + np.exp(-Z1))
    Z2 = W2.dot(A1) + b2
    A2 = 1 / (1 + np.exp(-Z2))

    activations = {
        'A1': A1,
        'A2': A2
    }

    return activations
```

Note : Nous aurons besoin de $A^{[1]}$
pour calculer les gradients $dW^{[2]}$ et $dZ^{[1]}$

$$dW^{[2]} = \frac{1}{m} \times dZ^{[2]} \cdot A^{[1]^T}$$

On va à présent passer à la fonction des gradients en utilisant la **back-propagation** :

ATTENTION Les effets de Broadcasting :

$$b^{[2]} = b^{[2]} - \alpha \times db^{[2]} \\ (n^{[2]}, 1) \quad (n^{[2]}, 1)$$

$$b^{[2]} = b^{[2]} - \alpha \times db^{[2]} \\ (n^{[2]}, 1) \quad (n^{[2]})$$

```
n2 = 3

b2 = np.random.randn(n2, 1)
db2 = np.random.randn(n2, 1)

b2 = b2 - 0.1 * db2
print(b2)

✓ 0.3s
```

résultat $(n^{[2]}, 1)$ --- correct

```
n2 = 3

b2 = np.random.randn(n2, 1)
db2 = np.random.randn(n2)

b2 = b2 - 0.1 * db2
print(b2)

✓ 0.4s
```

[[-0.10238776 0.38093192 0.34732375]
 [0.56623349 0.84477765 0.81116948]
 [-1.334333781 -1.05579364 -1.08940182]]

résultat $(n^{[2]}, n^{[2]})$ --- incorrect

```
def back_propagation(X, y, activations, parametres):
    A1 = activations['A1']
    A2 = activations['A2']
    W2 = parametres['W2']

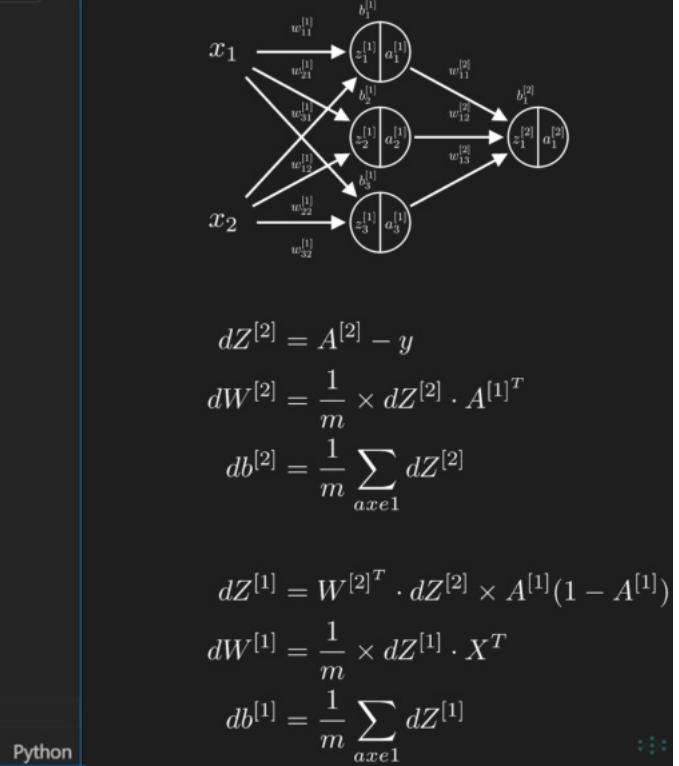
    m = y.shape[1]

    dZ2 = A2 - y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2, axis=1, keepdims=True)

    dZ1 = np.dot(W2.T, dZ2) * A1 * (1 - A1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1, axis=1, keepdims=True)

    gradients = {
        'dW1': dW1,
        'db1': db1,
        'dW2': dW2,
        'db2': db2
    }

    return gradients
```



Pour finir, il reste plus que la fonction update :

```
def update(gradients, parametres, learning_rate):  
    ... W1 = parametres['W1']  
    ... b1 = parametres['b1']  
    ... W2 = parametres['W2']  
    ... b2 = parametres['b2']  
  
    ... dW1 = gradients['dW1']  
    ... db1 = gradients['db1']  
    ... dW2 = gradients['dW2']  
    ... db2 = gradients['db2']  
  
    ... W1 = W1 - learning_rate * dW1  
    ... b1 = b1 - learning_rate * db1  
    ... W2 = W2 - learning_rate * dW2  
    ... b2 = b2 - learning_rate * db2  
  
    ... parametres = {  
    ...     'W1': W1,  
    ...     'b1': b1,  
    ...     'W2': W2,  
    ...     'b2': b2  
    ... }  
  
    ... return parametres
```

On fait une fonction predict :

```
def predict(X, parametres):  
    ... activations = forward_propagation(X, parametres)  
    ... A2 = activations['A2']  
    ... return A2 >= 0.5
```

Il reste plus qu'à tout assembler en une fonction final :

```
def neural_network(X_train, y_train, n1, learning_rate = 0.1, n_iter = 1000):

    # initialisation W, b
    n0 = X_train.shape[0]
    n2 = y_train.shape[0]
    parametres = initialisation(n0, n1, n2)

    train_loss = []
    train_acc = []

    for i in range(n_iter):

        activations = forward_propagation(X_train, parametres)
        gradients = back_propagation(X_train, y_train, activations, parametres)
        parametres = update(gradients, parametres, learning_rate)

        if i % 10 == 0:
            # Train
            train_loss.append(log_loss(y_train, activations['A2']))
            y_pred = predict(X_train, parametres)
            current_accuracy = accuracy_score(y_train.flatten(), y_pred.flatten())
            train_acc.append(current_accuracy)

    plt.figure(figsize=(14, 4))

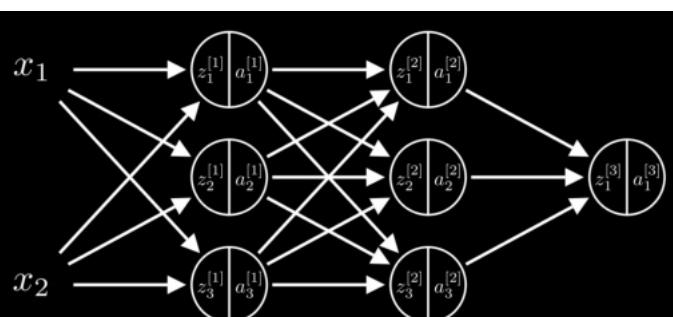
    plt.subplot(1, 2, 1)
    plt.plot(train_loss, label='train loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(train_acc, label='train acc')
    plt.legend()
    plt.show()

    return parametres
```

Généralisation des équations d'un réseau à 2 couches :

I/- Les Paramètres :



$$W^{[c]} \in \mathbb{R}^{n^{[c]} \times n^{[c-1]}}$$

$$b^{[c]} \in \mathbb{R}^{n^{[c]} \times 1}$$

```

def initialisation(dimensions):
    parametres = {}
    C = len(dimensions)

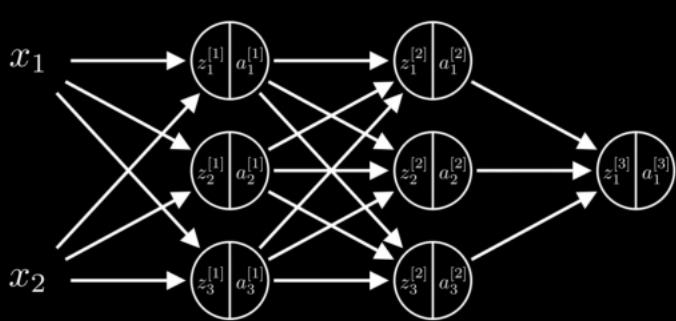
    for c in range(1, C):
        parametres['W' + str(c)] = np.random.randn(dimensions[c], dimensions[c - 1])
        parametres['b' + str(c)] = np.random.randn(dimensions[c], 1)

    return parametres

```

II-/ Forward-Propagation :

2. Forward Propagation



On pose $X = A^{[0]}$

$$Z^{[1]} = W^{[1]} \cdot X + b^{[1]}$$

$$A^{[1]} = \frac{1}{1 + e^{-Z^{[1]}}}$$

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + b^{[2]}$$

$$A^{[2]} = \frac{1}{1 + e^{-Z^{[2]}}}$$

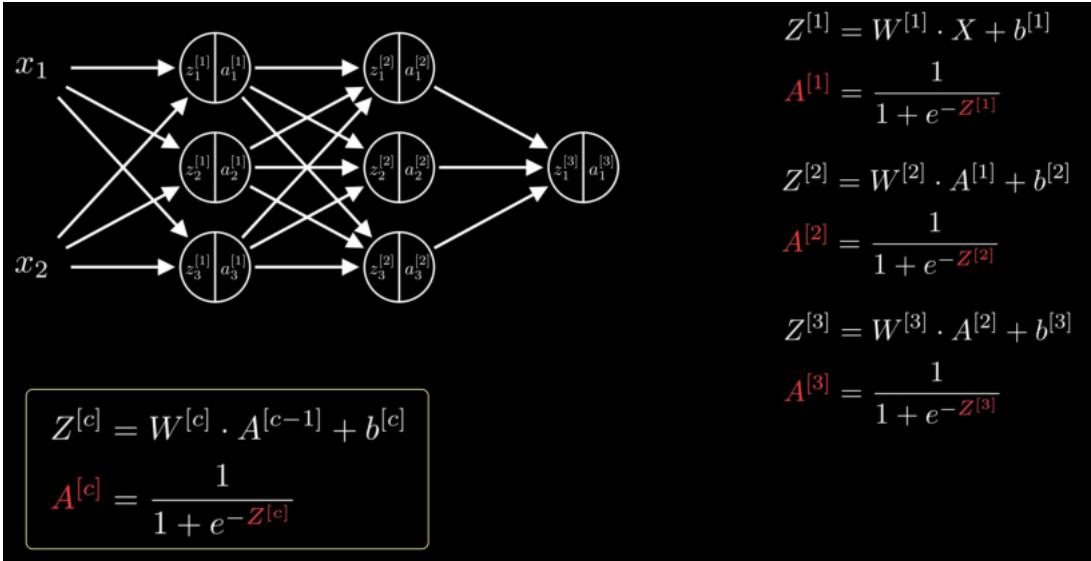
$$Z^{[3]} = W^{[3]} \cdot A^{[2]} + b^{[3]}$$

$$A^{[3]} = \frac{1}{1 + e^{-Z^{[3]}}}$$

$$Z^{[c]} = W^{[c]} \cdot A^{[c-1]} + b^{[c]}$$

$$A^{[c]} = \frac{1}{1 + e^{-Z^{[c]}}}$$

Ici on pose $X = A^{[0]}$ pour permettre à l'équation généralisée $Z^{[c]}$ de fonctionner avec $Z^{[1]}$ qui a X , donc on convertit le X en $A^{[0]}$ pour ne pas avoir de problème et pour que ça fonctionne pour toutes les équations même la première.

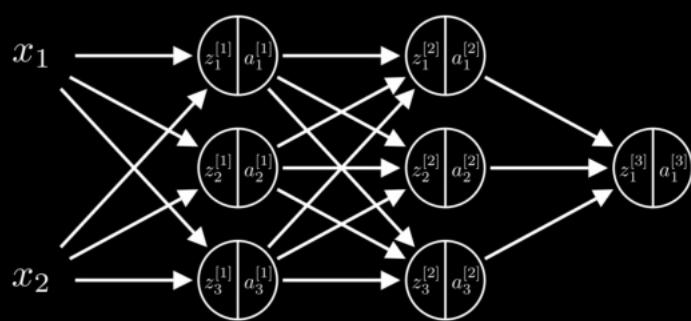


Pour $A[c]$ rien ne change on met juste c .

```
def forward_propagation(X, parametres):
    activations = {'A0': X}
    C = len(parametres) // 2
    for c in range(1, C + 1):
        Z = parametres['W' + str(c)].dot(activations['A' + str(c - 1)]) + parametres['b' + str(c)]
        activations['A' + str(c)] = 1 / (1 + np.exp(-Z))
    return activations
```

III-/ Back-Propagation :

3. Back-Propagation



$$\boxed{\begin{aligned} dZ^{[C_f]} &= A^{[C_f]} - y \\ dW^{[c]} &= \frac{1}{m} \times dZ^{[c]} \cdot A^{[c-1]^T} \\ db^{[c]} &= \frac{1}{m} \sum_{axe1} dZ^{[c]} \\ dZ^{[c-1]} &= W^{[c]^T} \cdot dZ^{[c]} \times A^{[c-1]}(1 - A^{[c-1]}) \end{aligned}}$$

$$dZ^{[3]} = A^{[3]} - y$$

$$dW^{[3]} = \frac{1}{m} \times dZ^{[3]} \cdot A^{[2]^T}$$

$$db^{[3]} = \frac{1}{m} \sum_{axe1} dZ^{[3]}$$

$$dZ^{[2]} = W^{[3]^T} \cdot dZ^{[3]} \times A^{[2]}(1 - A^{[2]})$$

$$dW^{[2]} = \frac{1}{m} \times dZ^{[2]} \cdot A^{[1]^T}$$

$$db^{[2]} = \frac{1}{m} \sum_{axe1} dZ^{[2]}$$

$$dZ^{[1]} = W^{[2]^T} \cdot dZ^{[2]} \times A^{[1]}(1 - A^{[1]})$$

$$dW^{[1]} = \frac{1}{m} \times dZ^{[1]} \cdot X^T$$

$$db^{[1]} = \frac{1}{m} \sum_{axe1} dZ^{[1]}$$

⋮⋮⋮

```
def back_propagation(y, activations, parametres):
    m = y.shape[1]
    C = len(parametres) // 2

    dZ = activations['A' + str(C)] - y
    gradients = {}

    for c in reversed(range(1, C+1)):
        gradients['dW' + str(c)] = 1 / m * np.dot(dZ, activations['A' + str(c-1)].T)
        gradients['db' + str(c)] = 1 / m * np.sum(dZ, axis=1, keepdims=True)
        if c > 1:
            dZ = np.dot(parametres['W' + str(c)].T, dZ) * activations['A' + str(c-1)] * (1 - activations['A' + str(c-1)])
    return gradients
```

IV-/ Mise à jour des paramètres :

$$\boxed{W^{[c]} = W^{[c]} - \alpha \times dW^{[c]}}$$

$$b^{[c]} = b^{[c]} - \alpha \times db^{[c]}$$

```
def update(gradients, parametres, learning_rate):
    C = len(parametres) // 2
    for c in range(1, C + 1):
        parametres['W' + str(c)] = parametres['W' + str(c)] - learning_rate * gradients['dW' + str(c)]
        parametres['b' + str(c)] = parametres['b' + str(c)] - learning_rate * gradients['db' + str(c)]
    return parametres
```

Fonction Final :

```
def neural_network(X, y, hidden_layers = (32, 32, 32), learning_rate = 0.1, n_iter = 1000):
    np.random.seed(0)
    # initialisation W, b
    dimensions = list(hidden_layers)
    dimensions.insert(0, X.shape[0])
    dimensions.append(y.shape[0])
    parametres = initialisation(dimensions)

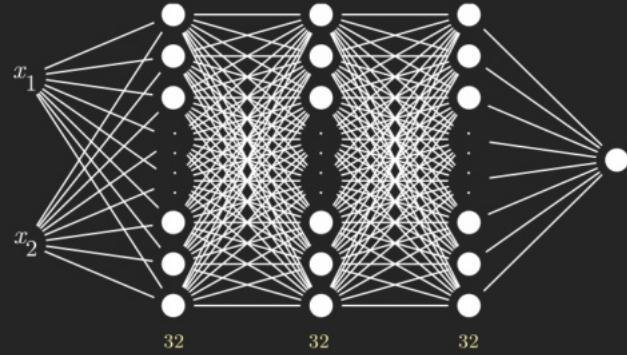
    train_loss = []
    train_acc = []

    for i in tqdm(range(n_iter)):
        activations = forward_propagation(X, parametres)
        gradients = back_propagation(y, activations, parametres)
        parametres = update(gradients, parametres, learning_rate)

        if i % 10 == 0:
            C = len(parametres) // 2
            train_loss.append(log_loss(y, activations['A' + str(C)]))
            y_pred = predict(X, parametres)
            current_accuracy = accuracy_score(y.flatten(), y_pred.flatten())
            train_acc.append(current_accuracy)

    # Visualisation des résultats
    fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))
    ax[0].plot(train_loss, label='train loss')
    ax[0].legend()
    ax[1].plot(train_acc, label='train acc')
    ax[1].legend()
    visualisation(X, y, parametres, ax)
    plt.show()

    return parametres
```



Pour calculer les prédictions :

```
def predict(X, parametres):
    activations = forward_propagation(X, parametres)
    C = len(parametres) // 2
    AF = activations['A' + str(C)]
    return AF >= 0.5
```

Petite précision :

Plus un réseau de neurone est profond (a de couches) et plus celui-ci a de chance de se perdre dans son apprentissage !

En Deep Learning c'est un problème qu'on appelle : le Vanishing Gradients ou le Exploding Gradients