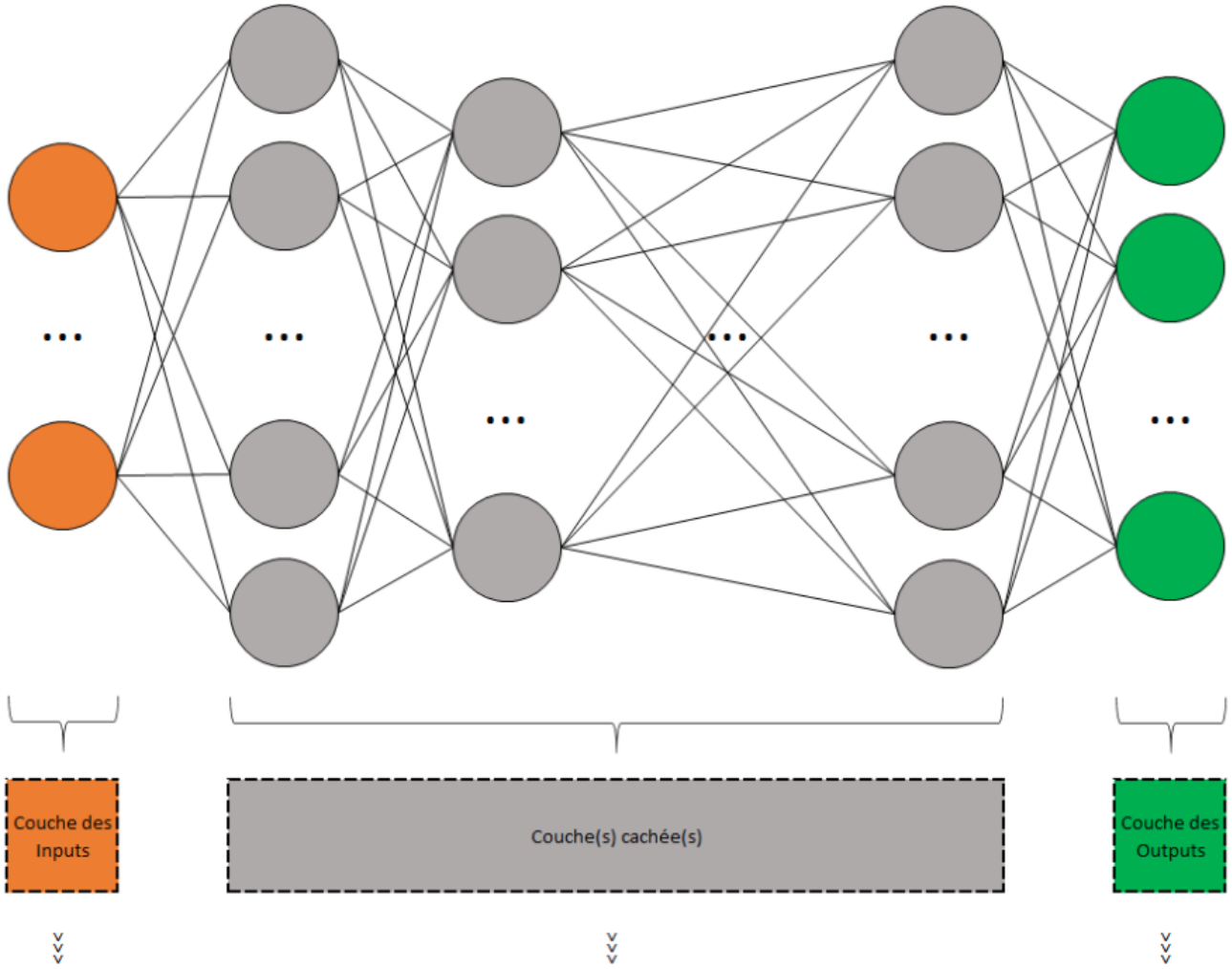
# Petite introduction aux méthodes de l'IA

Deuxième partie

G. Barmarin



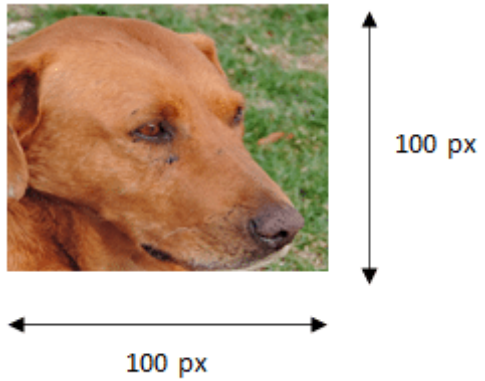

Ici nous avons 2 neurones, mais ça peut être 1, 2, 10, 20... cela dépend du contexte, du cadre dans lequel on utilise le réseau

Il peut y avoir un nombre infini de couches (à partir de 2 couches, on peut parler de réseau profond). Chaque couche peut également contenir une infinité de neurones. Ces choix (nombre de couches, et nombre de neurones par couches) sont, la plupart du temps, arbitraires.

Le nombre de neurones dans la couche de sortie dépend également du contexte, du cadre dans lequel on utilise le réseau.

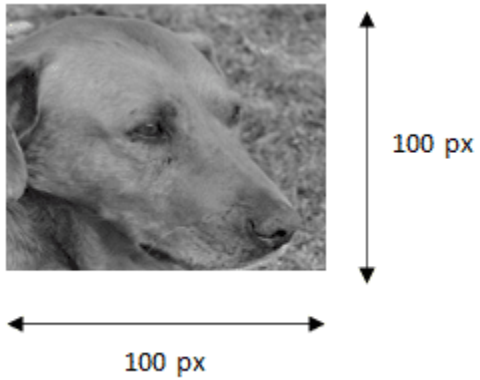
# 1er contexte : la reconnaissance d'images

1.1. L'image en couleur



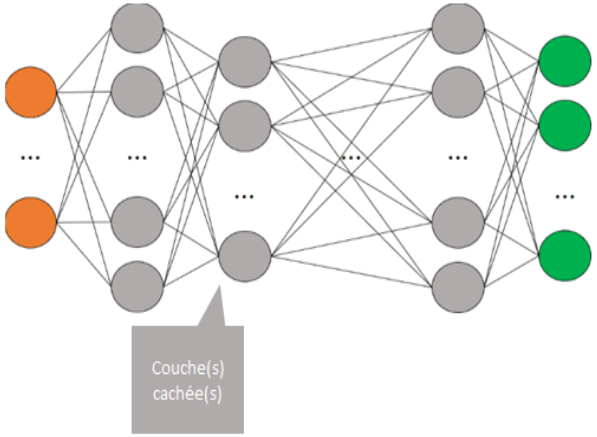

Cette image a une taille de 100 px sur 100 px.  Un pixel contient 3 valeurs (R, G, B). Cette image est donc composée de 100 x 100 x 3 = 30 000 valeurs. Ces 30 000 valeurs vont représenter notre input layer (couche d'entrée). Il y a aura donc 30 000 neurones dans la première couche. (note: Il existe aussi les réseaux neuronaux convolutif (CNN), qui modifient la façon d'appréhender la valeur des couches en entrée d'un NN (Neural Network).

1.2. L'image en noir et blanc



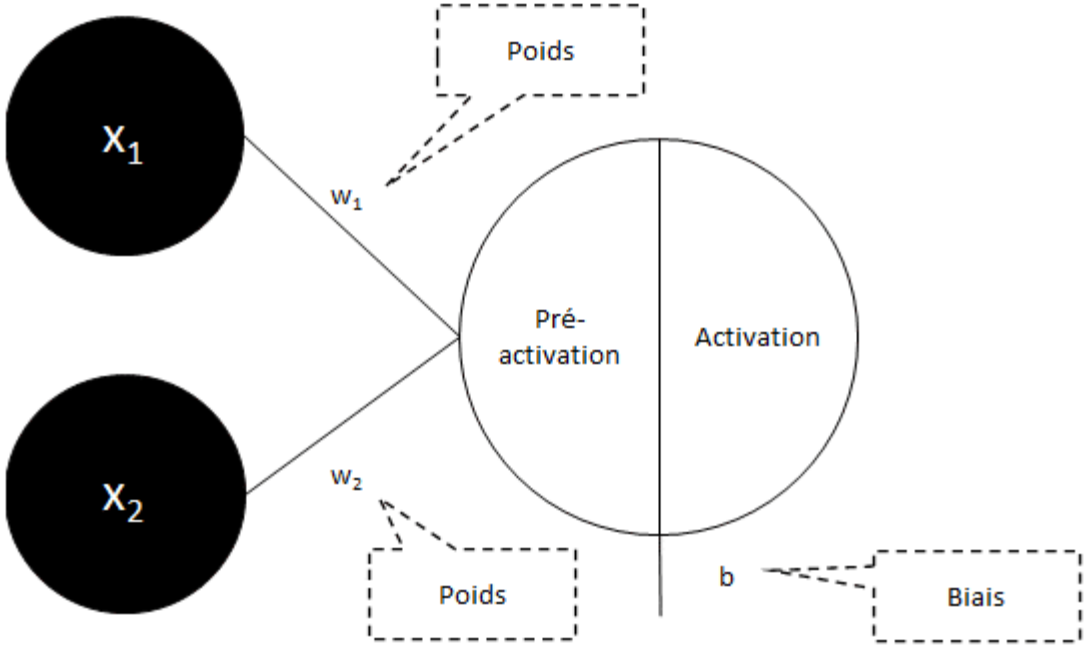

Cette image a une taille de 100 px sur 100 px. Un pixel contient 1 valeur comprise entre 0 et 1 (0 = blanc, 1 = noir, gris = 0,5…). Cette image est donc composée de 100 x 100 = 10 000 valeurs. Ces 10 000 valeurs vont représenter notre input layer (couche d'entrée). Il y aura donc 10 000 neurones dans la première couche.

# Couche(s) cachée(s)




Cette fois, chaque « rond » représente un neurone artificiel.
Comme indiqué précédemment, le nombre de couches est variable et variera en fonction du contexte  mais aussi de la puissance de calcul disponible pour entraîner et utiliser le réseau.
C'est la même chose pour le nombre de neurones par couches.

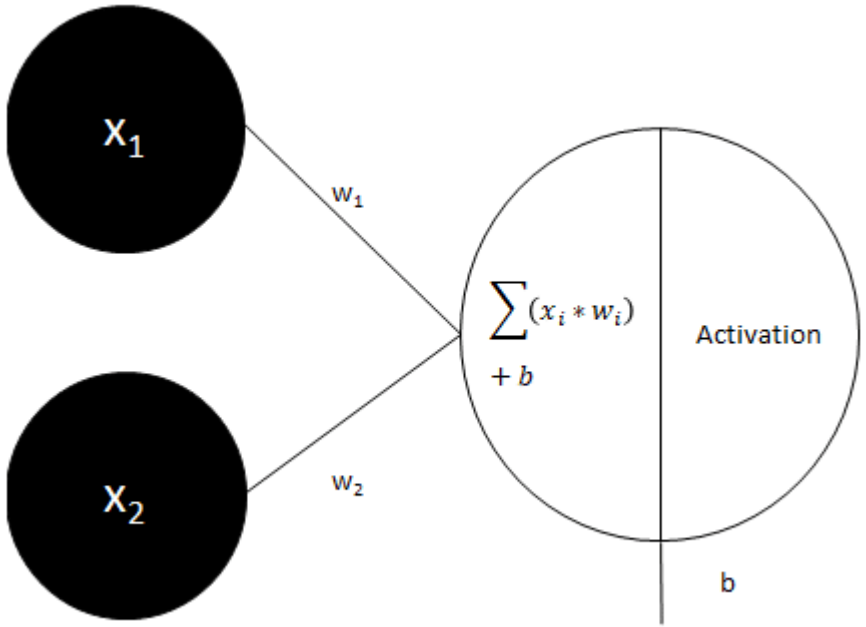
1. Tous les neurones de la (ou des) couche(s) cachée(s) se décomposent comme suit.



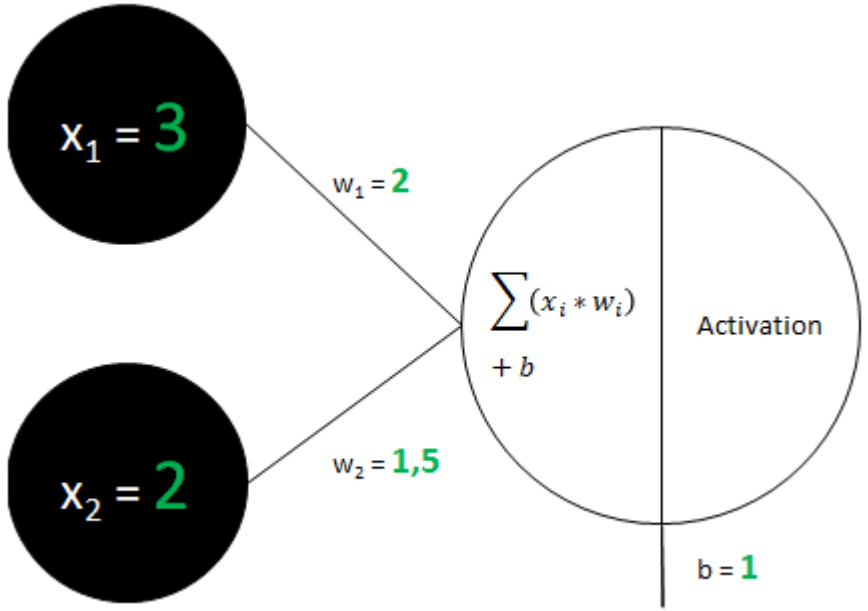

Un neurone est composé de 2 fonctions, la pré-activation et l'activation.

2. La formule de pré-activation est toujours la même.

Réseau de neurones - Couches cachées - préactivation
Cette formule multiplie tout simplement tous les x avec les w, additionne les résultats obtenus, puis ajoute un biais.

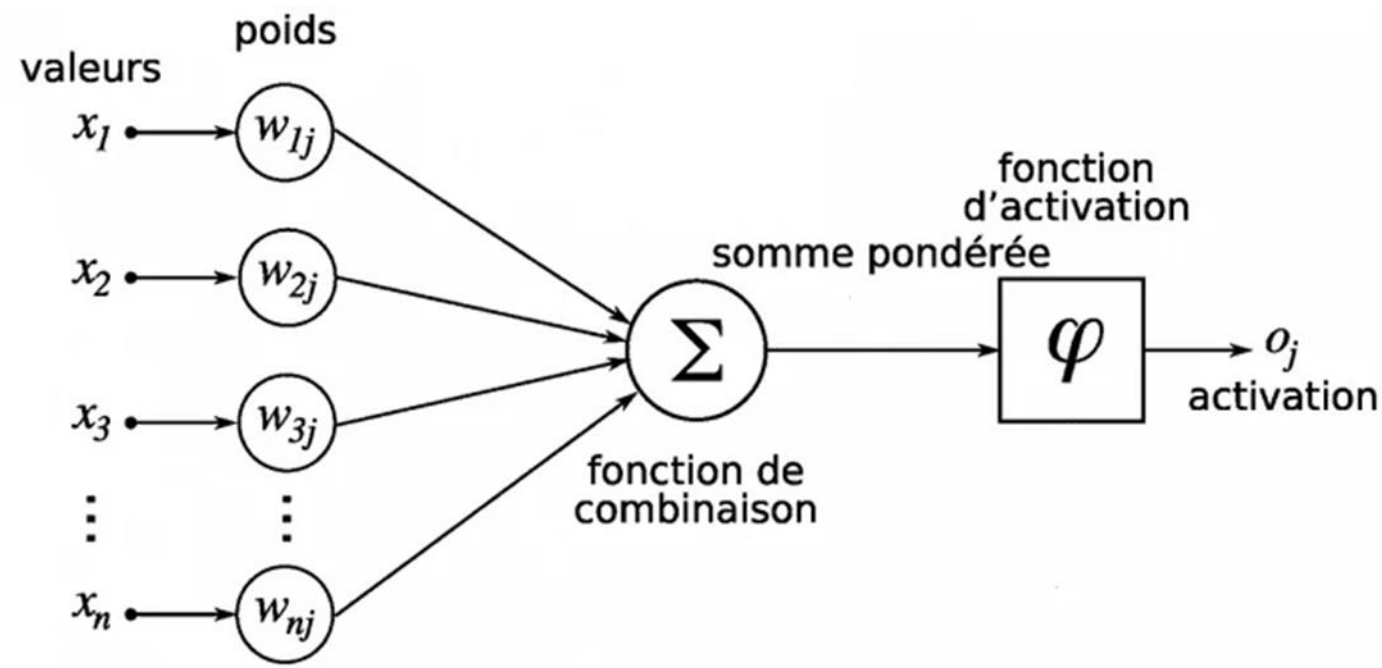



Exemple :



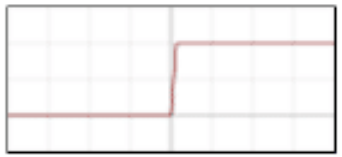

Dans la situation décrite dans l'image, notre pré-activation est égale à :
= x1 * w1 + x2 * w2 + b
= 3×2 + 2×1,5 + 1
= 10

Cela ne change rien si on a plus de 2 neurones :



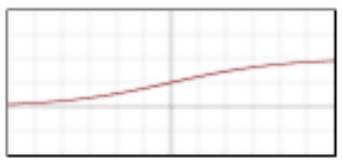
3. Passons désormais à la formule d'activation, celle-ci change en fonction du contexte, du cas d'usage concerné par le réseau. Voici les principales formules d'activation utilisées.

Fonction "Marche/Heaviside"



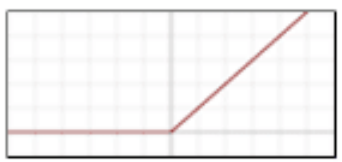
$$f(x) = \begin{cases} 0 & \text{si} \quad x < 0 \\ 1 & \text{si} \quad x \geq 0 \end{cases}$$

Fonction Sigmoïde



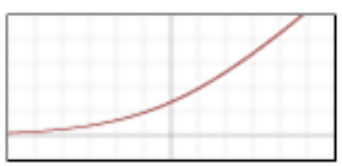
$$f(x) = \frac{1}{1 + e^{-x}}$$

Fonction "Unité de rectification linéaire" (ReLU)



$$f(x) = \begin{cases} 0 & \text{si} \quad x < 0 \\ x & \text{si} \quad x \geq 0 \end{cases}$$

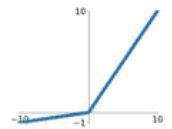
Fonction "Unité de rectification linéaire douce" (SoftPlus)



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

Il y en a bien d'autres:

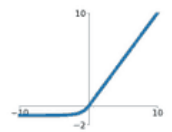
**Leaky ReLU**
$\max(0.1x, x)$



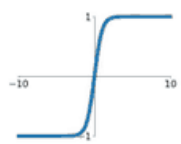
**Maxout**
$\max(w_1^T x + b_1, w_2^T x + b_2)$

**ELU**

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



**tanh**
$\tanh(x)$

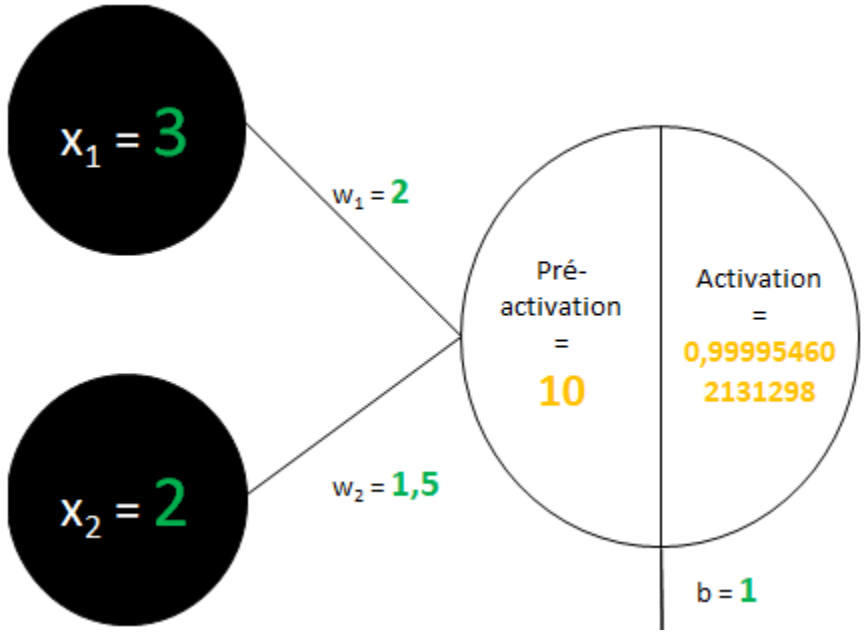


4. Pour l'exemple, nous allons utiliser la fonction sigmoïde.

Précédemment, nous avions calculé la pré-activation (x1 * w1 + x2 * w2 + b = 3*2 + 2*1,5 + 1 = 10). Nous allons appliquer la fonction sigmoïde à ce résultat (nous allons remplacer notre « x » par « le résultat de la pré-activation »).

$$\textbf{Activation} = f(x) = \frac{1}{(1 + e^{-x})}) = \frac{1}{(1 + e^{-10})} = \mathbf{0,999954602131298}$$

Notre neurone devient :



Couche de l’output (des outputs)

1. Le nombre de neurones dans cette couche est variable et correspond au contexte du réseau.

Pour rappel, dans le contexte de la reconnaissance d’images on souhaite connaître par exemple le pourcentage de chance que l’image représente un chat, et le pourcentage de chance que l’image représente un chien. Nous aurons donc 2 neurones de sortie: probabilité d’avoir une photo de chat et probabilité d’avoir une photo de chien.

2. Quel que soit le nombre de neurones dans cette couche de sortie, les neurones fonctionnent comme ceux des couches cachées à la seule différence qu’il y a des contextes où seule la fonction de pré-activation est utilisée, et d’autres (c’est majoritairement le cas), où les 2 fonctions sont utilisées (pré-activation et activation).

# L’entraînement

L’entraînement d’un réseau de neurones consiste à faire en sorte que la machine trouve, seule par elle-même :

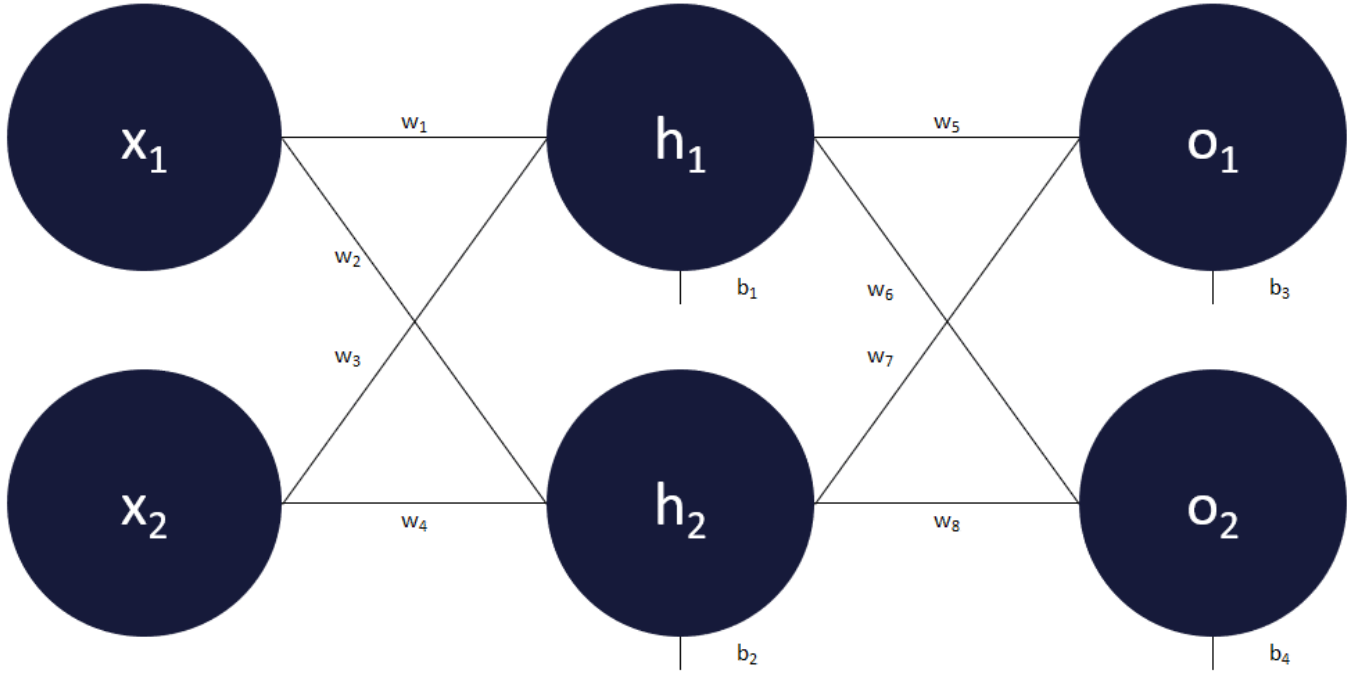
- les bonnes valeurs pour tous les biais (b) du réseau
- les bonnes valeurs pour tous les poids (w) du réseau

Concrètement, nous allons dans l’ordre :

- Fournir des données aléatoires pour tous les b et w du réseau,
- Fournir en entrée (dans la couche d’entrée) des données labellisées (i.e. des données dont on connaît la sortie attendue – ex. on sait que cette image fournie en entrée représente un chien, on attend que le réseau prédise qu’il s’agit d’un chien, on sait que cette fleur est bleue, on attend que le réseau nous réponde que la fleur est bleue…),
- Laisser la machine faire ses calculs (i.e. calculer les résultats des neurones (pré-activation et activation) jusqu’à la couche de sortie),
- Laisser la machine comparer les résultats obtenus avec les résultats attendus,
- Laisser la machine corriger plus ou moins fortement les b et w afin de minimiser l’erreur (l’écart entre ce qui a été prédit et ce que l’on espérait voir être prédit),
- Recommencer (i.e. fournir en entrée des données labellisées, laisser la machine faire ses calculs jusqu’à la couche de sortie, calculer l’erreur… etc.)

## Comment mettre en place l'apprentissage d'un réseau de neurones ?

Voici le réseau de neurones que nous allons utiliser afin de comprendre comment un réseau de neurones apprend par lui-même à corriger ses erreurs. La fonction d'activation choisie sera la fonction *sigmoïde*.



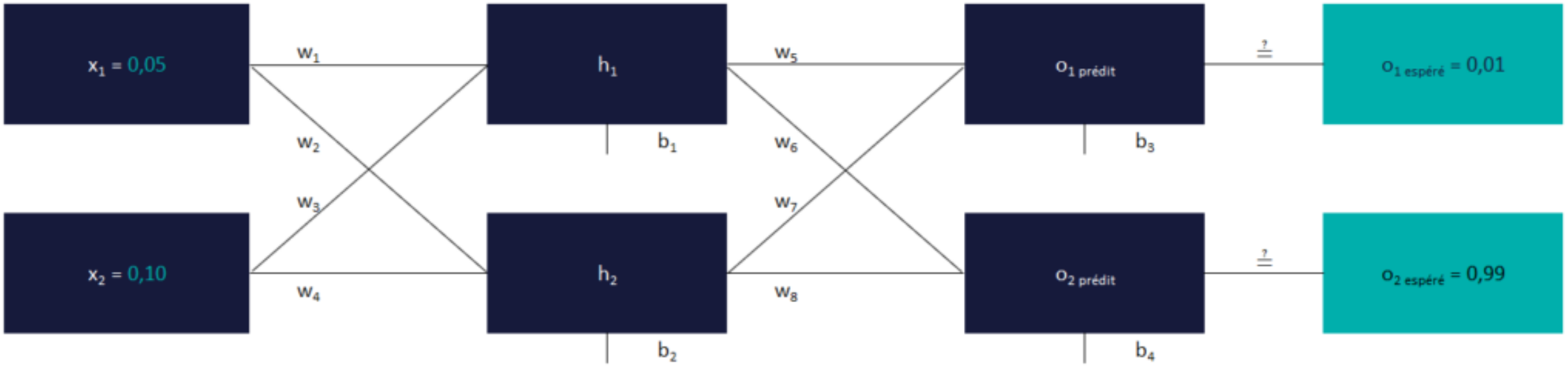

Ce réseau est donc composé :

- d'une couche d'entrée qui comprend 2 entrées x1 et x2,
- d'une couche cachée qui comprend 2 neurones h1 et h2,
- d'une couche de sortie qui comprend également 2 neurones o1 et o2, ces 2 sorties seront respectivement désignées comme « o1 prédit » et « o2 prédit »,
- il y a en tout 8 poids et 4 biais.

**Nos données d'entrées et nos sorties attendues, ce que nous souhaitons faire avec ce réseau :**

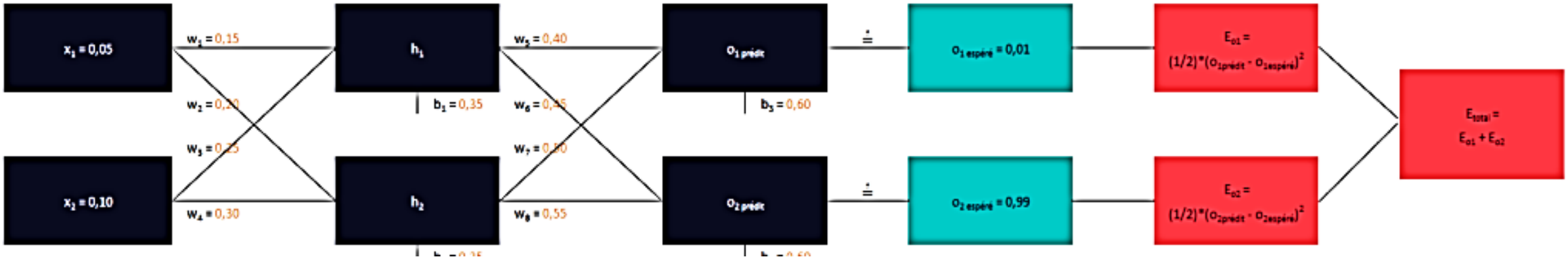
Quelles que soient les valeurs x1 et x2 renseignées en entrée, nous souhaitons que o1 soit le plus proche possible de 0,01 et o2 le plus proche possible de 0,99.
(Remarque: cet objectif n'a aucun intérêt dans un environnement de production, il est simplement utilisé ici pour des raisons pédagogiques.)

Nous allons utiliser, pour la phase d'apprentissage, des valeurs fixes pour x1 et x2 (respectivement 0,05 et 0,10). Puis, nous changerons ces valeurs après la phase d'apprentissage afin de voir si les sorties prédites restent respectivement proches de 0,01 et de 0,99.



Nos poids et biais aléatoires

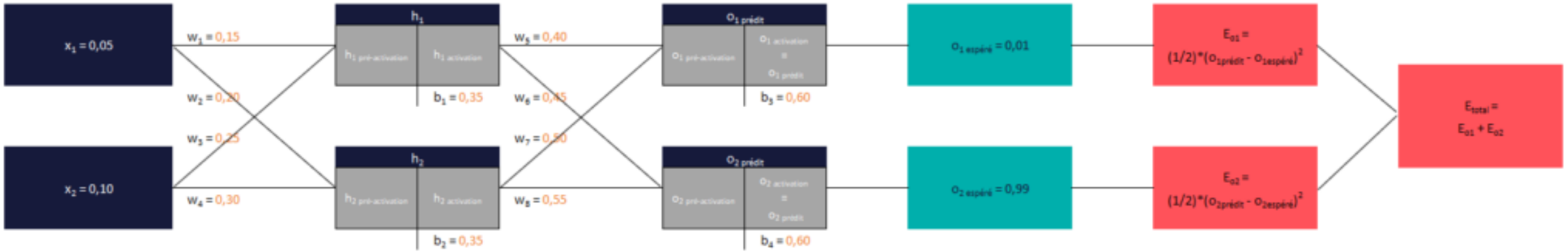
Nous allons commencer avec des poids et biais aléatoires, ce sont ces valeurs que la machine va faire évoluer, seule, afin que x1 et x2 soient, respectivement, e plus proche de 0,01 et 0,99, et ce, indépendamment de leur valeur.
Notre réseau, dans son état initial, est donc le suivant :




Comme vous pouvez le voir, 3 éléments ont été ajoutés :

Eo1 --> l'erreur o1 représente la moitié de la différence, au carré, entre o1prédit et o1espéré (ex. o1prédit = 0,5 ; o1espéré = 0,01 ; Eo1 = (1/2) * (0,5 – 0,01)2 = 0,12005)
Eo2 --> l'erreur o2 représente la moitié de la différence, au carré, entre o2prédit et o2espéré (ex. o2prédit = 0,3 ; o1espéré = 0,99 ; Eo2 = (1/2) * (0,3 – 0,99)2 = 0,23805)
Etotal --> l'erreur totale est tout simplement la somme de Eo1 et de Eo2.

Dans la mesure où nos 2 dernières couches (couche cachée et couches de sortie) comprennent une formule de pré-activation et une formule d'activation, notre réseau se présente sous la forme suivante :




Pour rappel, si l'on prend h1, il est calculé en 2 temps :

$$h_{1préactivation} = x_1 * w_1 + x_2 * w_3 + b_1$$

$$h_{1activation} = sigm(h_{1préactivation}) = \frac{1}{1 + e^{-h_{1préactivation}}} = h_1$$

C'est la même chose pour h2, o1, o2.

1ère itération

En faisant les calculs, voici les résultats obtenus :

| $x_1$ | $x_2$ | $w_1$ | $w_2$ | $w_3$ | $w_4$ | $b_1$ |
|---|---|---|---|---|---|---|
| 0,05 | 0,10 | 0,15000000 | 0,20000000 | 0,25000000 | 0,30000000 | 0,35000000 |

| $b_1$ | $h_1$ - Pré-activation | $h_1$ - Activation | $h_1$ | $b_2$ | $h_2$ - Pré-activation | $h_2$ - Activation | $h_2$ |
|---|---|---|---|---|---|---|---|
| 0,35000000 | 0,38250000 | 0,59447593 | 0,59447593 | 0,35000000 | 0,39000000 | 0,59628270 | 0,59628270 |

| $w_5$ | $w_6$ | $w_7$ | $w_8$ | $b_3$ | $o_1$ - Pré-activation | $o_1$ - Activation | $b_4$ | $o_2$ - Pré-activation |
|---|---|---|---|---|---|---|---|---|
| 0,40000000 | 0,45000000 | 0,50000000 | 0,55000000 | 0,60000000 | 1,13593172 | 0,75693192 | 0,60000000 | 1,19546965 |

| $o_1$ - Prédit | $o_2$ - Prédit | $o_1$ - Espéré | $o_2$ - Espéré | $E_{o1}$ | $E_{o2}$ | $E_{total}$ |
|---|---|---|---|---|---|---|
| 0,75693192 | 0,76771788 | 0,01 | 0,99 | 0,27895364 | 0,02470467 | 0,30365831 |

En noir : des données fixes, qui ne changeront pas entre 2 itérations,
En vert : les poids et biais qui vont être optimisés à chacune des itérations, et donc être modifiés
En orange : les résultats des formules de pré-activation et d'activation qui seront calculés à chaque itération en fonction des nouveaux poids et biais modifiés à chaque itération

h1préactivation par exemple, est égal à : x1 * w1 + x2 * w3 + b1 = 0,05 * 0,15 + 0,10 * 0,25 + 0,35 = 0,3825

h1activation, est donc égal à : 1 / (1 + e-0,3825) = 0,594475931

L'action consistant à faire les calculs de la gauche vers la droite, au regard des poids et biais à un instant T, est appelée forward.

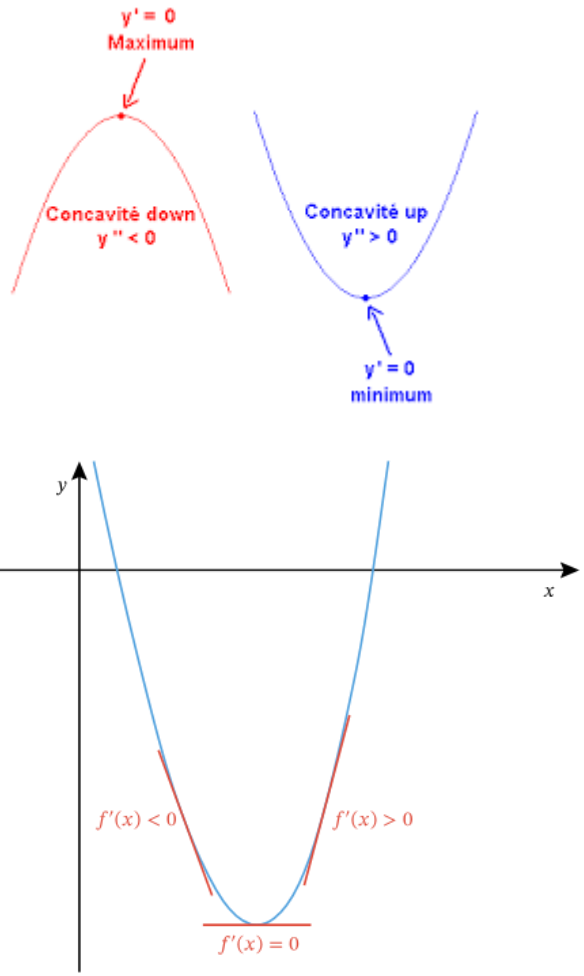
## Descente de gradient

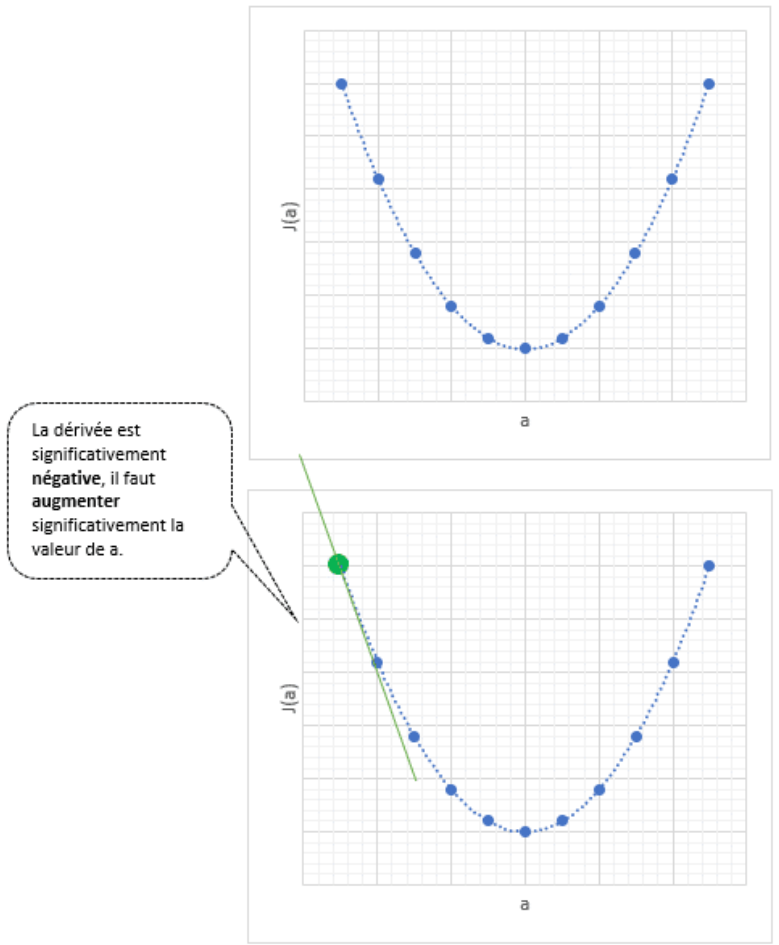
Rappel, nous souhaitons trouver des poids et biais qui permettent, quelles que soient les entrées x1 et x2, d'avoir un o1prédit le plus proche de 0,01 et o2prédit le plus proche de 0,99.

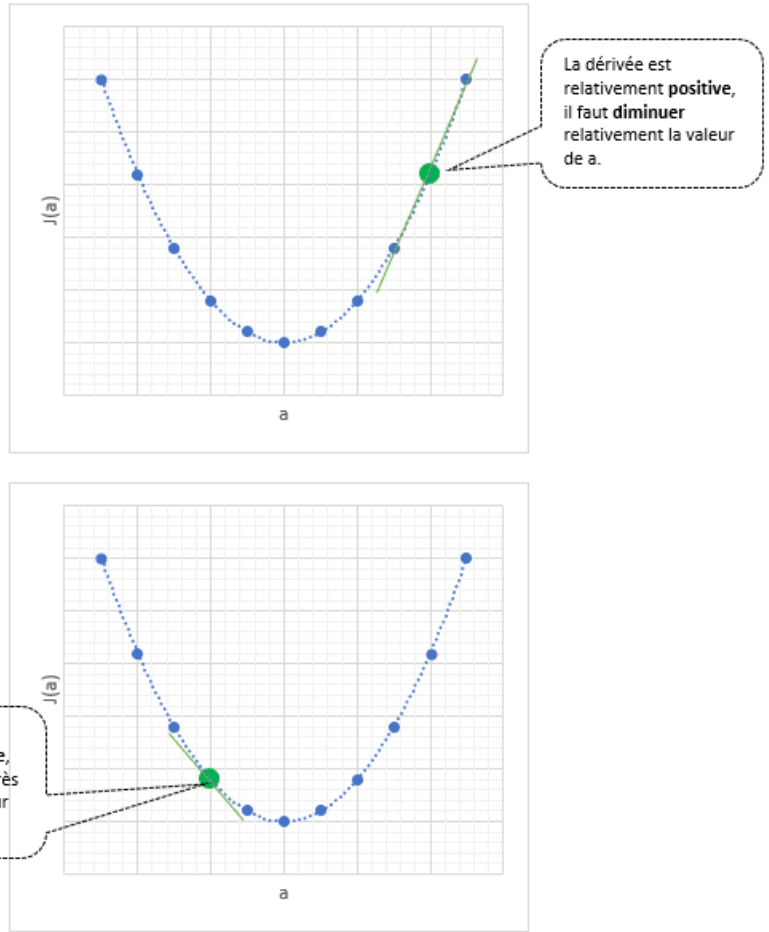
Nous allons utiliser la méthode de la descente de gradient, appelée rétropropagation du gradient dans le cas d'un réseau de neurones.

Notre fonction coût Etotal dépend des valeurs de :
– w1
– w2
– w3
– w4
– w5
– w6
– w7
– w8
– b1
– b2
– b3
– b4
Ce sont les 12 seules valeurs qui peuvent être modifiées dans ce réseau de neurones (en vert sur la dia précédente).

Rappel:  la dérivée mesure la pente d'un point située sur une courbe.
À gauche d'un minimum relatif, la dérivée est négative, au minimum relatif, la dérivée est nulle et à droite du minimum relatif, la dérivée est positive.
Pour minimiser la fonction coût, nous pouvons calculer le minimum de sa dérivée descendante avant sa remontée, dont l'abscisse correspondra à l'endroit du minimum de la fonction…



**La descente de gradient aléatoire est une suite d'opérations que voici:**

Etape a. La machine choisit aléatoirement un nombre pour a.  Exemple : a = -758

Etape b. La machine calcule la dérivée afin de savoir si elle est à gauche ou à droite du point minimum.

Dérivée de J(a) par rapport à a = $\frac{\partial J(a)}{a} = \frac{452\,381}{7} * a - 12\,867{,}114$ = (452381 / 7) * (-758) – 12 867,114 = – 48999266,83

Ici la dérivée de « a = -758 » est très négative, cela signifie que la pente descend très fort et qu'on se situe à gauche du minimum. On peut donc continuer et proposer un « a » plus grand que « -758 ».

Étape c. La machine propose aléatoirement un nouveau chiffre plus grand que « -758 ». Exemple : a = -432

Étape d. La machine calcule la dérivée afin de savoir si elle est à gauche ou à droite du point minimum.
La dérivée de « a = -432 » est toujours négative, cela signifie que la pente descend, on se situe à gauche du minimum.
On peut donc continuer et proposer un a plus grand que « -432 ».

Étape e. La machine propose aléatoirement un nouveau chiffre plus grand que « -432 ». Exemple : a = 23

Étape f. La machine calcule la dérivée afin de savoir si elle est à gauche ou à droite du point minimum.
La dérivée de « a = 23 » est positive, cela signifie que la pente monte, on se situe à droite du minimum.
On peut donc continuer et proposer un a situé entre « -432 » et « 23 ».

Etape … n. La machine se stabilise et trouve la valeur de a permettant de minimiser la fonction de coût.

Cette méthode fonctionne mais peut prendre beaucoup de temps et nécessite de nombreux calculs…

## Utilisation de la descente de gradient « optimisée »

Comment gagner du temps et minimiser le nombre de calculs à effectuer?

Plutôt que de proposer aléatoirement à chacune des étapes une nouvelle valeur pour a, on va soustraire à a un certain pourcentage de la dérivée calculée à l'étape précédente, pourcentage que l'on appelle « learning rate ».

Cette technique est appelée communément la descente de gradient.

La nouvelle valeur de a n'est cette fois plus aléatoire mais elle est calculée de la manière suivante :

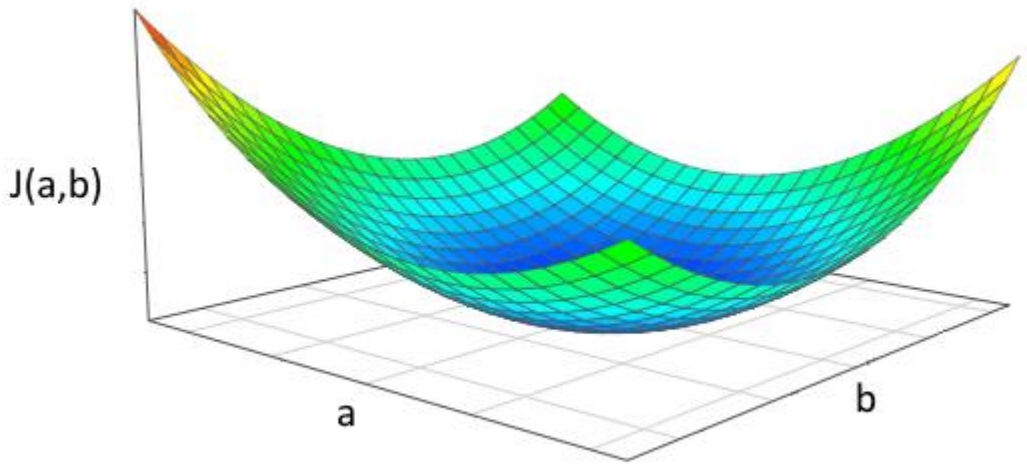
« Nouveau a » = « Ancien a » – « learning_rate » * « Dérivée »

Le « learning rate » vaut typiquement 0,01

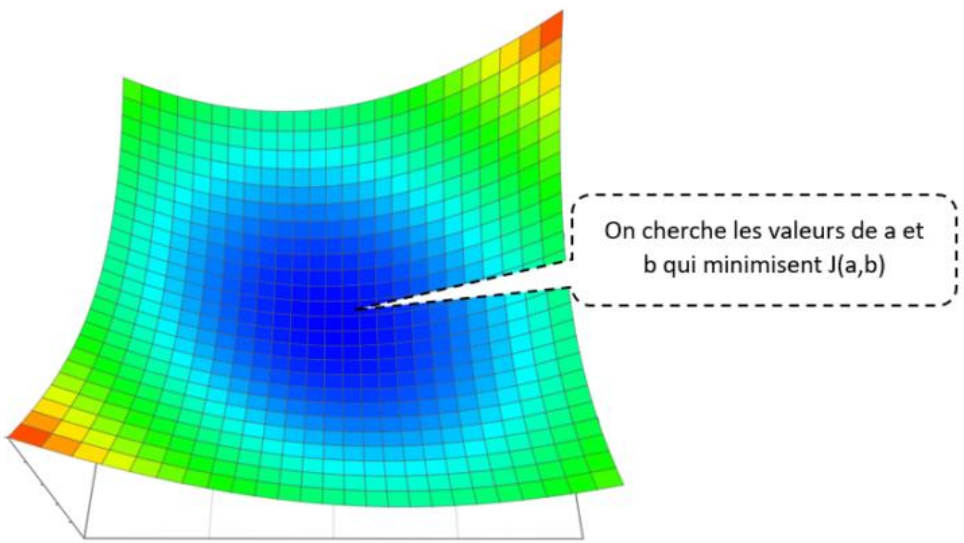
Utiliser la descente de gradient « optimisée » pour trouver, cette fois, a et b donc deux paramètres liés par une fonction est à peu près identique.

Il faut seulement comprendre que l'ajout d'une nouvelle variable, en l'occurrence b, transforme notre fonction coût en un graphique en 3D dimensions.

En 3 dimensions, une fonction coût pourrait se présenter sous cette forme.




Vue du dessus:



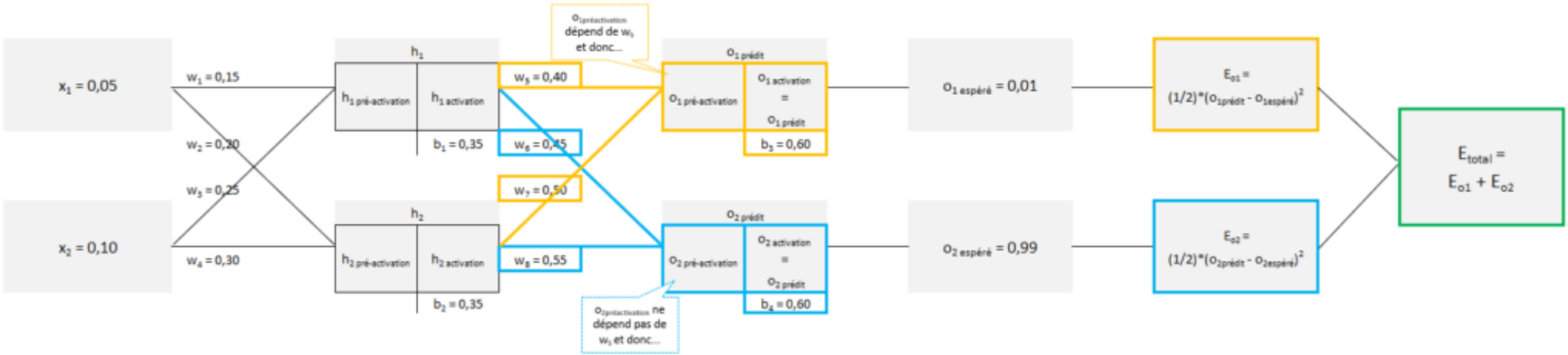

On va donc dériver la fonction selon a puis selon b,
On aura des « dérivées partielles »

- la dérivée de J(a,b) par rapport à *a* $(\frac{\partial\, j(a,b)}{\partial a})$
- la dérivée de J(a,b) par rapport à *b* $(\frac{\partial\, j(a,b)}{\partial b})$

Revenons à notre réseau de neurones et ses 12(!) paramètres…

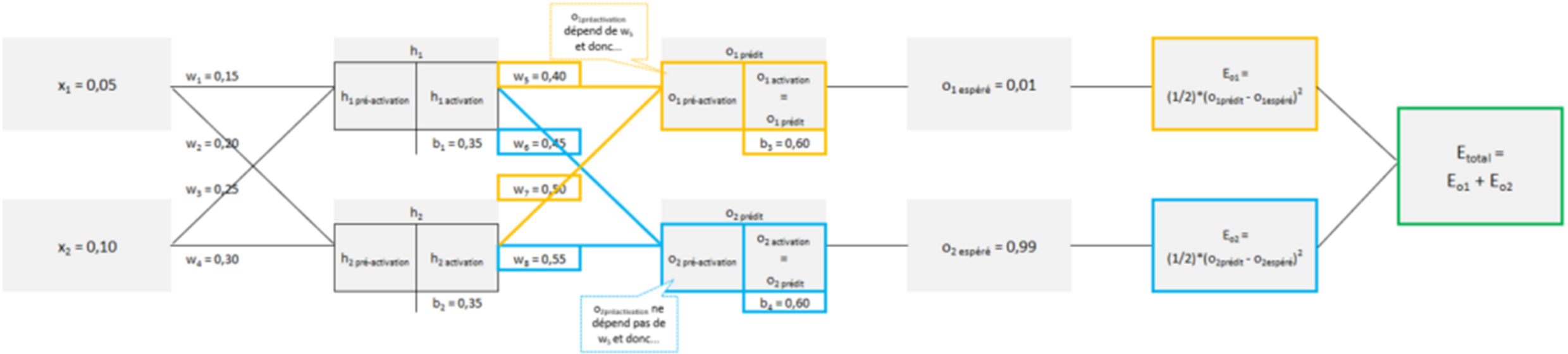
Pour calculer la dérivée et minimiser l'erreur, nous allons partir de la couche de sortie puis revenir vers la couche d'entrée en calculant, pour chacune des couches, les dérivées de chacun des poids et biais la constituant.

Exemple: Dérivée de Etotal par rapport à w5 $\frac{\partial E_{total}}{\partial w_5} = ?$ :




Etotal est la somme de Eo1 et de Eo2.

Prenons Eo1. Les éléments qui permettent de calculer Eo1 sont indiqués en orange ci-dessus.
En effet, Eo1 dépend de o1prédit, o1prédit (qui est égal à o1activation) dépend de o1préactivation
o1préactivaion dépend de b3, w5 et de w7.
Si l'on suit le « chemin » de Eo1, on voit bien que Eo1 dépend de w5 donc la dérivée de Eo1 par rapport à w5 n'est pas nulle.




Prenons maintenant Eo2.

Les éléments qui permettent de calculer Eo2 sont indiqués en bleu ci-dessus.
Eo2 dépend de o2prédit, O2prédit (qui est égal à o2activation) dépend de o2préactivation, o2préactivation dépend de b4, w6 et de w8.

Si l'on suit le chemin de $E_{o2}$, on voit, à la différence de $E_{o1}$, que $E_{o2}$ ne dépend pas de $w_5$. La dérivée de $E_{o2}$ par rapport à $w_5$ est donc nulle, w5 n'influence pas $E_{o2}$.

Revenons à nos calculs, si on développe notre formule, voici ce que cela donne:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial(E_{o1} + E_{o2})}{\partial w_5} = \frac{\partial(E_{o1})}{\partial w_5} + \frac{\partial(E_{o2})}{\partial w_5} = \frac{\partial(E_{o1})}{\partial w_5} + 0 = \frac{\partial(\frac{1}{2} * (o_{1prédit} - o_{1espéré})^2)}{\partial w_5}$$ en effet: $$\frac{\partial(E_{o2})}{\partial w_5} = 0$$

$$= \frac{\partial(\frac{1}{2} * \left(\frac{1}{1 + e^{-o_{1préactivation}}} - o_{1espéré}\right)^2)}{\partial w_5} = \frac{\partial(\frac{1}{2} * \left(\frac{1}{1 + e^{-(h_1 * w_5 + h_2 * w_7 + b_3)}} - o_{1espéré}\right)^2)}{\partial w_5}$$

Dérivation des fonctions composées: si nous avons une fonction : f(g(h(x))) alors sa dérivée partielle est : $$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} * \frac{\partial g}{\partial h} * \frac{\partial h}{\partial x}$$

Ce qui se traduit par: $$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{o1}}{\partial w_5} = \frac{\partial E_{total}}{\partial o_{1activation}} * \frac{\partial o_{1activation}}{\partial o_{1préactivation}} * \frac{\partial o_{1préactivation}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial o_{1activation}} = \frac{\partial(E_{o1} + E_{o2})}{\partial o_{1activation}} = \frac{\partial E_{o1}}{\partial o_{1activation}} + \frac{\partial E_{o2}}{\partial o_{1activation}} = \frac{\partial E_{o1}}{\partial o_{1activation}} + 0 = \frac{\partial E_{o1}}{\partial o_{1activation}} = \frac{\partial(\frac{1}{2} * (o_{1prédit} - o_{1espéré})^2)}{\partial o_{1activation}}$$

$$= \frac{\partial(\frac{1}{2} * (o_{1prédit} - o_{1espéré})^2)}{\partial o_{1activation}} = \frac{\partial(\frac{1}{2} * (o_{1activation} - o_{1espéré})^2)}{\partial o_{1activation}} = 2 * \frac{1}{2} * (o_{1activation} - o_{1espéré}) = (o_{1activation} - o_{1espéré}) = (o_{1prédit} - o_{1espéré})$$

$$\frac{\partial o_{1activation}}{\partial o_{1préactivation}} = \frac{\partial sigm(o_{1préactivation})}{\partial o_{1préactivation}} = sigm(o_{1préactivation}) * \left(1 - sigm(o_{1préactivation})\right)$$

$$= o_{1activation} * (1 - o_{1activation}) = o_{1prédit} * (1 - o_{1prédit})$$

(la dérivée d'une sigmoïde est égale à : sigmoïde * (1 – sigmoïde) )

$$\frac{\partial o_{1préactivation}}{\partial w_5} = \frac{\partial(h_1 * w_5 + h_2 * w_7 + b_3)}{\partial w_5} = \frac{\partial(h_1 * w_5)}{\partial w_5} + \frac{\partial(h_2 * w_7)}{\partial w_5} + \frac{\partial(b_3)}{\partial w_5} = h_1 * 1 + 0 + 0 = h_1$$

## Forward et Backward

L'activité consistant à faire les calculs de la gauche vers la droite, c'est-à-dire de la couche d'entrée vers la couche de sortie, est appelé Forward.

A la fin de celle-ci, une erreur est calculée. Dans notre cas, il s'agit de Etotal.
Cette erreur va nous permettre de faire une marche arrière, et de corriger légèrement à chaque itération nos poids et biais.

Nous allons donc faire des itérations successives afin que les dérivées partielles de Etotal par rapport à tous les poids et biais soient le plus proche possible de 0.

Concrètement, voici les étapes que nous allons suivre :

Etape initiale : donner des valeurs aléatoires aux poids et biais. Cette étape n'est à faire qu'une seule fois.
Etape 1 : calcul de toutes les dérivées partielles de tous les poids (w) et biais (b) par rapport à Etotal, et ce au regard de toutes les valeurs à un instant T du réseau,
Etape 2 : si celles-ci ne tendent pas vers 0, les variables associées peuvent être optimisées selon la descente de gradient.
De nouveaux poids (w) et biais (b) seront calculés, de la même manière que dans le cas de la régression linéaire, à savoir :
nouveau wi= ancien wi – learning rate * ∂Etotal/∂w (idem pour les biais).
Etape 3 : les nouveaux poids (w) et biais (b) calculés seront utilisés dans le cadre d'une nouvelle itération,
On recommence les étapes 1, 2, 3 un très grand nombre de fois jusqu'à trouver :
Eo1 très proche de 0
Eo2 très proche de 0

Les algorithmes de rétro-propagation et d'optimisation permettent d'exploiter la grande capacité du réseau en contraignant les poids des réseaux vers un ensemble des valeurs qui mène à une bonne performance de classification : les caractéristiques sont apprises spécifiquement pour la tâche considérée.

Les caractéristiques extraites, simples si on les considère dans les premières couches, se complexifient peu à peu au fil des opérations faites par les couches. C'est cette dimension de couches empilées, qui permet la création de caractéristiques complexes, à laquelle on fait référence lorsqu'on parle **d'apprentissage « profond ».**

Cela mène à des caractéristiques qui peuvent être extrêmement raffinées et subtiles, bien plus que ce qu'un humain ou un algorithme d'extraction de caractéristiques classique peuvent réaliser au prix précisément que l'humain ne puisse plus « comprendre » le « raisonnement » de la machine.

## Inconvénients

La grande capacité des réseaux peut entraîner cependant un grand inconvénient : le sur-apprentissage (overfit en anglais). Pour l'illustrer, plaçons-nous dans le cadre d'une application d'apprentissage automatique simple : la régression non-linéaire.
Considérons que nous avons n points d'entraînement qui sont des paires de nombres réels (xi, yi) et que nous souhaitons ajuster un polynôme de degré m à ces points, c'est-à-dire trouver les coefficients qui permettent au polynôme de passer au plus près de ces points. Si m ≥ n, il existe forcément un polynôme qui soit exactement ajusté à ces données : un tel modèle a assez de capacité pour représenter complètement le jeu de données.

Note:

A force de vouloir trop bien faire, on fait pire!

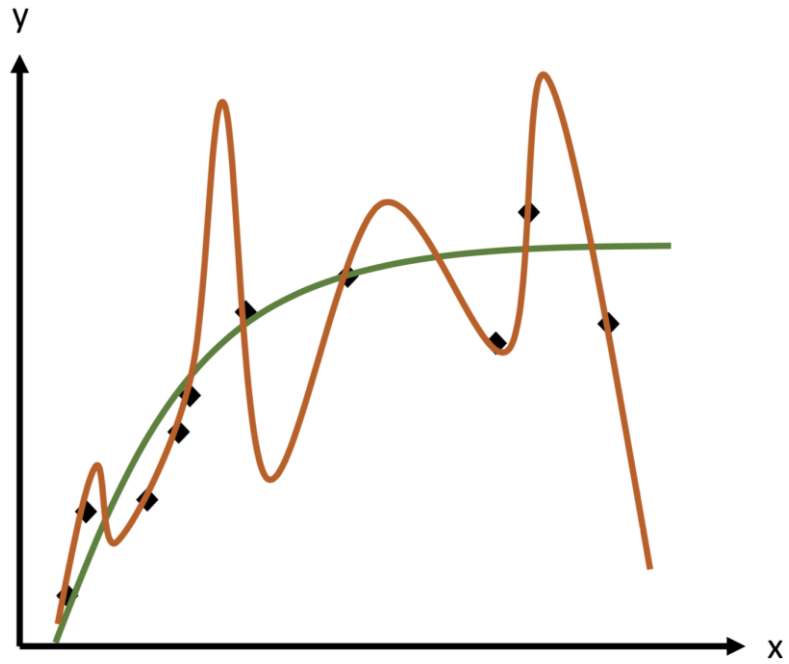



Illustration sur un cas de régression non-linéaire du phénomène de sur-apprentissage

L’optimisation des coefficients du polynôme mène donc à une fonction qui passe exactement par tous les points d’entraînement.

Cette situation, alléchante sur le papier, est en fait souvent très fâcheuse. En effet, il faut se souvenir que, comme pour toute procédure de statistique inférentielle, le jeu d’entraînement ne constitue qu’un échantillon de la distribution que l’on cherche à ajuster. À ce titre, les points d’entraînement représentent la distribution mais avec un certain bruit (écart au signal). En utilisant un polynôme de degré élevé, l’ajustement se fera sur ce bruit plutôt que sur la véritable distribution.

Dans la figure précédente, les points semblent provenir d’une distribution qui pourrait être ajustée par un polynôme de degré 2 (courbe verte). L’utilisation d’un polynôme de degré 10 (courbe marron) mène à un ajustement plus précis des points d’entraînement, mais nous constatons aisément que la capacité «excédentaire» du modèle (les 8 paramètres supplémentaires) a servi à ajuster le bruit.

Une telle configuration d’entraînement mène souvent en outre à un modèle ajusté très éloigné de la fonction génératrice dans les plages où il n’y a pas suffisamment de données d’entraînement (tout à droite de la figure par exemple). Lors d’une phase de prédiction d’une nouvelle donnée, l’ajustement proposé risque de ne pas être satisfaisant. On parle alors de sur-apprentissage : l’algorithme a appris « par cœur » les données d’entraînement sans en tirer la « substantifique moelle » [Rabelais, 1534].

# Exemple avec sklearn

Exemple avec sklearn: Création du réseau et préparation de l'échantillon d'entraînement ¶

**scikit-learn** permet de créer, d'entraîner, et d'évaluer un réseau de neurones en quelques lignes de code.

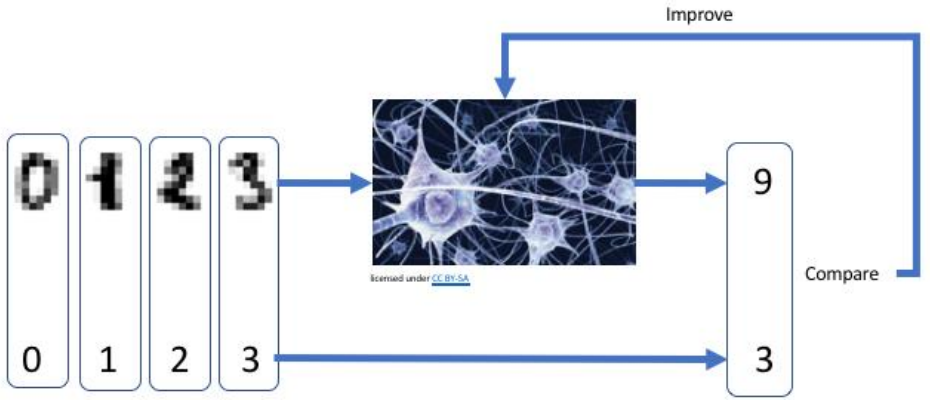
Nous allons créer un réseau extrêmement simple, avec seulement trois couches:

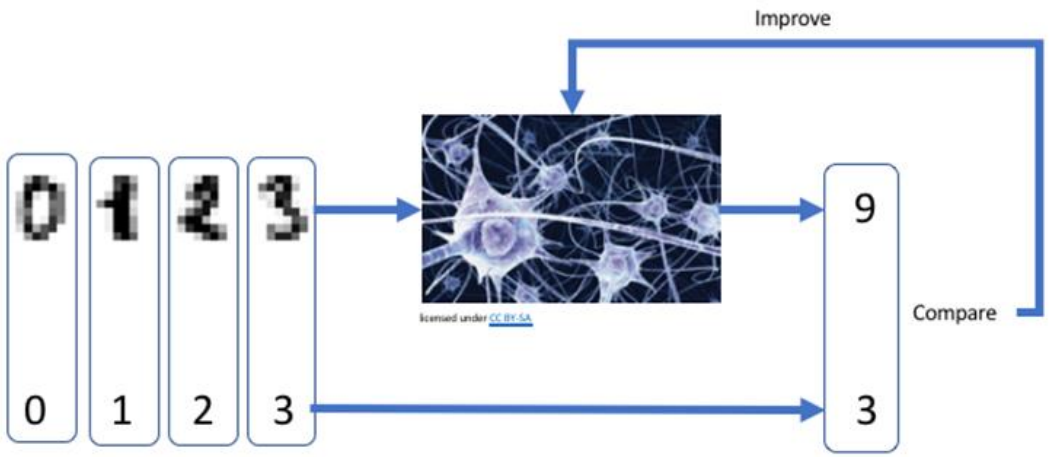
une couche d'entrée ( input layer ), avec un noeud par pixel dans les images 8x8, 64 noeuds, donc) (neurone qui ne fait rien, il se contente de prendre la valeur en entrée et de la transmettre aux neurones de la couche suivante).

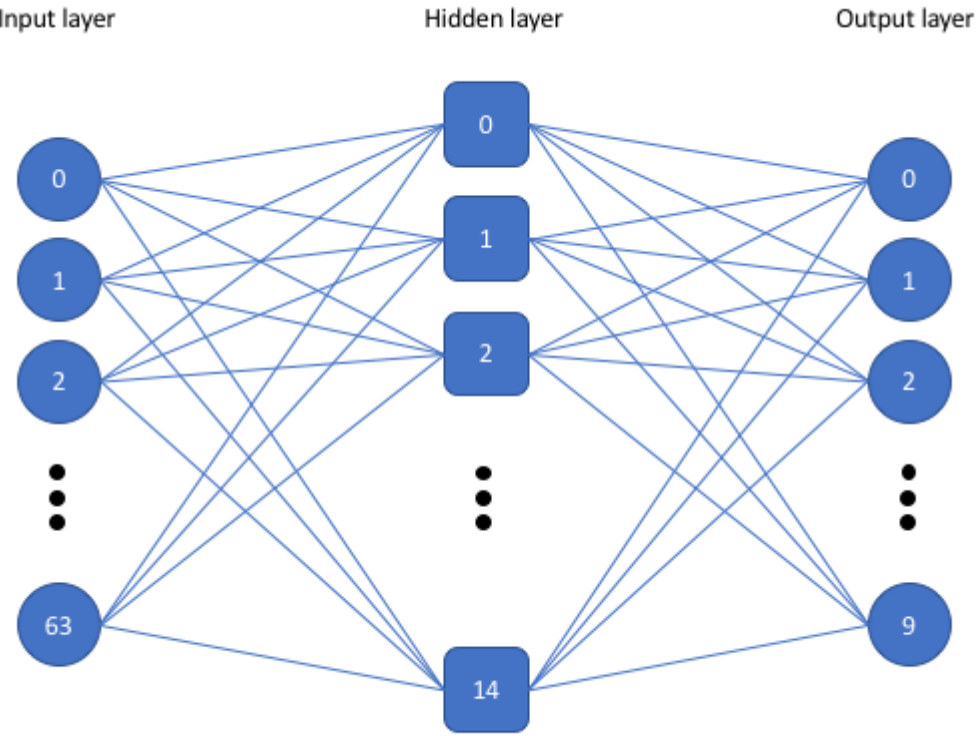
une couche cachée ( hidden layer ) avec 15 neurones.
Nous aurions pu choisir un autre nombre de neurones, et ajouter des couches avec différents nombres de neurones.

une couche de sortie ( output layer ) avec 10 neurones, correspondant à nos 10 classes de chiffres, de 0 à 9

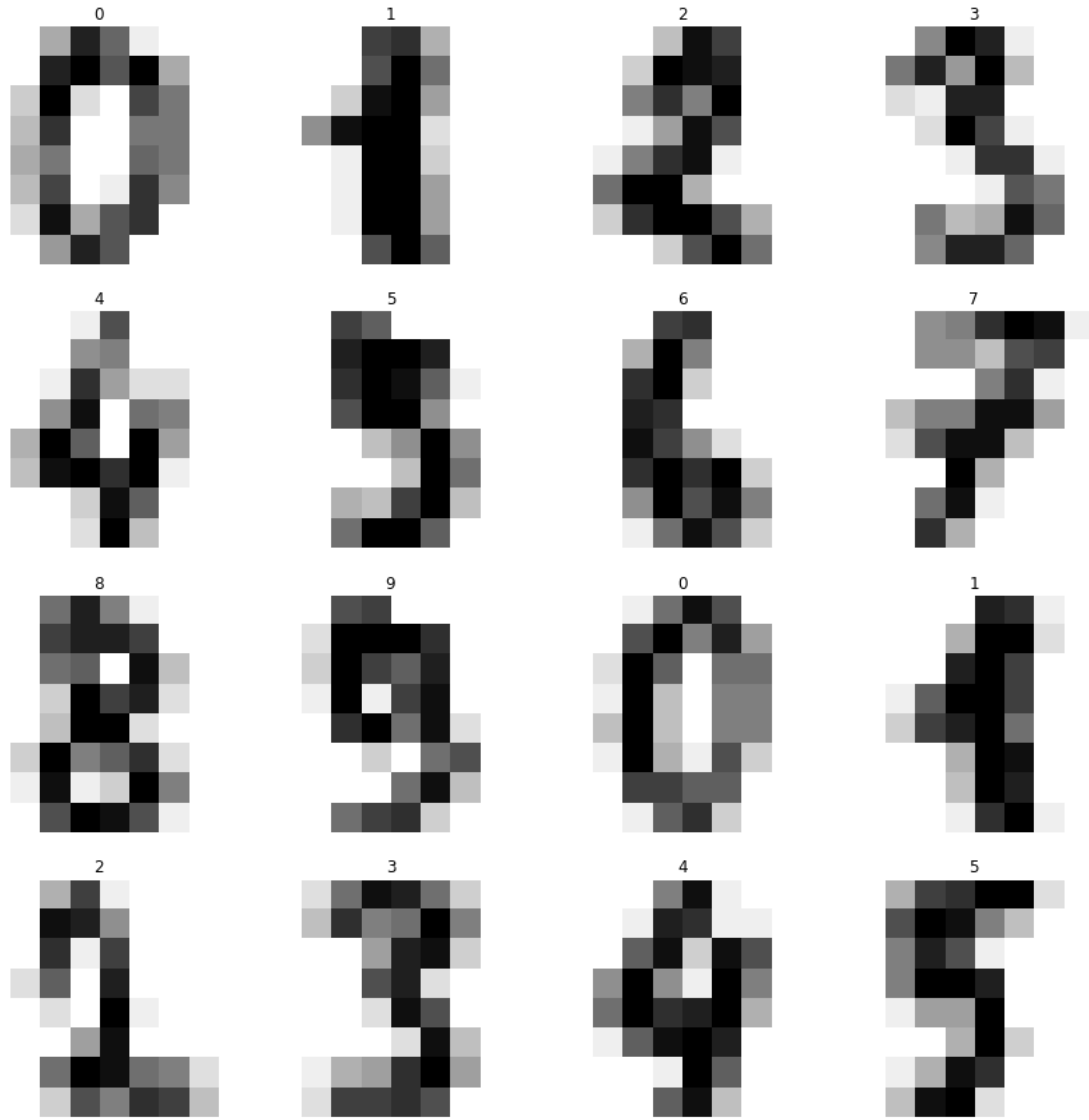
On dit que ce type de réseau est dense , ce qui veut dire que dans chaque couche, chaque neurone est connecté avec tous les neurones des couches précédentes et suivantes.




Improve
0
1
2
3
licensed under CC BY-SA
9
3
Compare


Input layer
Hidden layer
Output layer
0
1
2
63
0
1
2
14
0
1
2
9

Un échantillon des données du dataset représenté sous forme d’images



## Sources:


https://fr.blog.businessdecision.com/tutoriel-machine-learning-comprendre-ce-quest-un-reseau-de-neurones-et-en-creer-un/

https://fr.blog.businessdecision.com/tutoriel-machine-learning-comment-mettre-en-place-lapprentissage-dun-reseau-de-neurones/

https://thedatafrog.com/fr/articles/handwritten-digit-recognition-scikit-learn/

https://culturesciencesphysique.ens-lyon.fr/ressource/IA-Bernet-2.xml

https://culturesciencesphysique.ens-lyon.fr/ressource/IA-apprentissage-Rousseau.xml