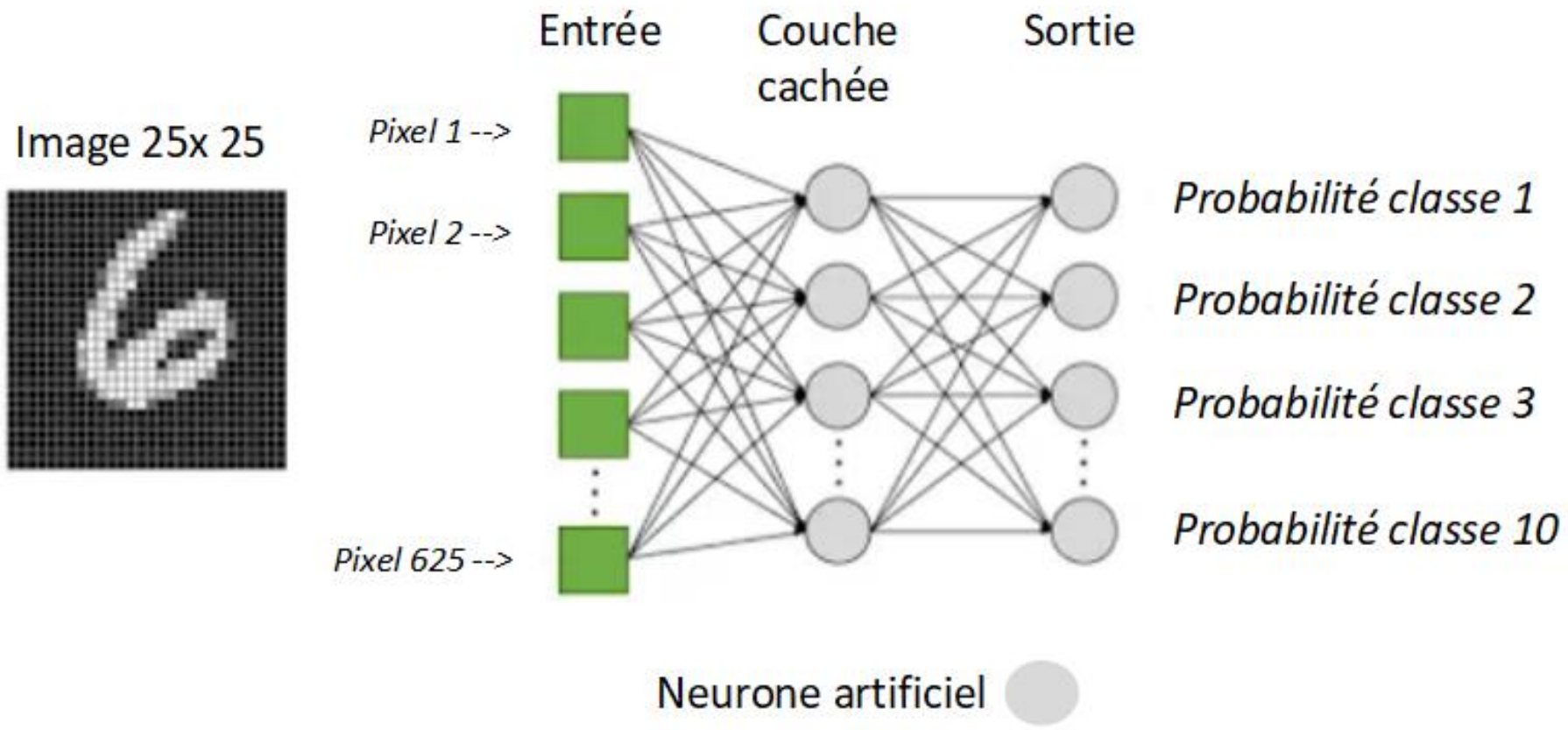
# Introduction aux réseaux de neurones

Adrien Lescourt & Tarik Garidi

# Présentation

Un réseau neuronal est l'association plus ou moins complexe, d'objets élémentaires, les neurones artificiels. Par exemple:

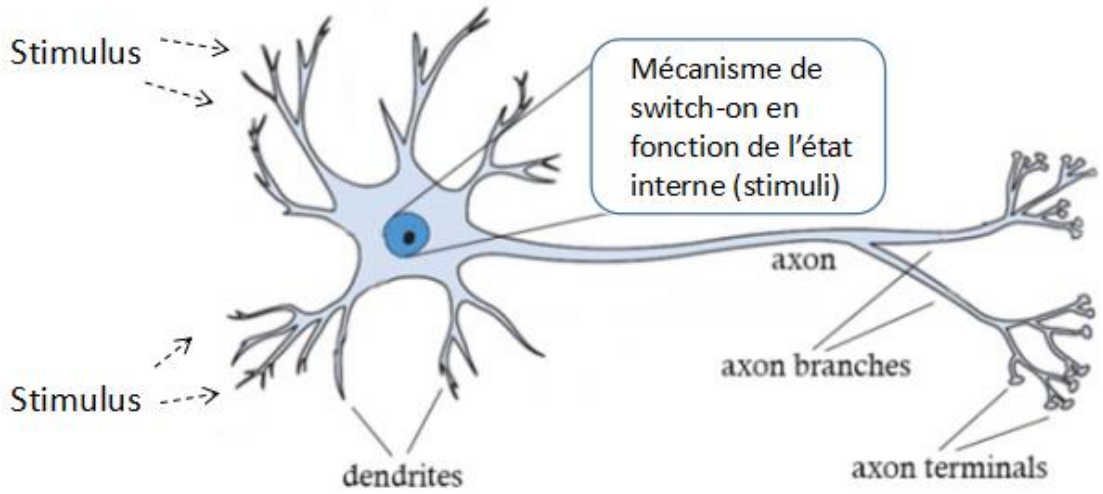


# Analogie (lointaine) avec les neurones biologiques

Paradigme *connexionniste:* vision parallèle et distribuée de l'activité du cerveau

- un neurone traite l'information indépendamment puis communique un résultat à d'autres neurones (synapses)
- Parties du cerveau (groupes de neurones) se spécialisent dans certaines tâches

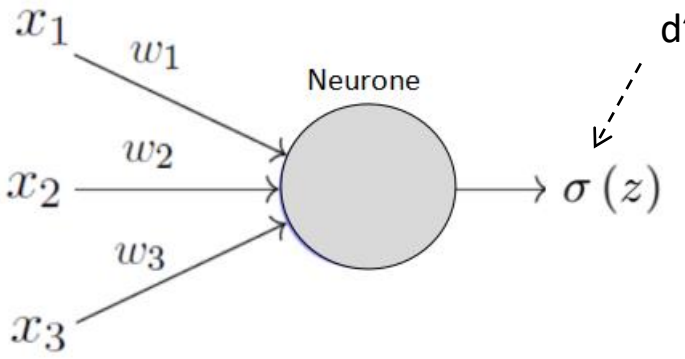
Traitement d'un signal par un neurone:




Sortie: le neurone communique sa décision d'activer ou non à d'autres neurones

# Neurone artificiel

Un neurone implémente le mécanisme de *switch-on* à l'aide d'une représentation de son état et d'une fonction d'activation:




Fonction d'activation

Etat interne : combinaison (souvent linéaire) des variables d'entrées (les stimuli)

avec $z = \sum_i \omega_i \cdot x_i + b$

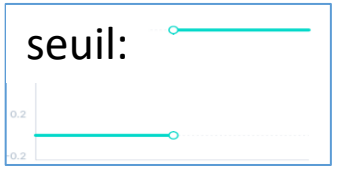
- Les paramètres ω,b sont la mémoire du réseau, ils sont estimés dans la phase d'apprentissage. Ils permettent d'ajuster les états (moduler l'importances des variables) en fonction de la réponse attendue

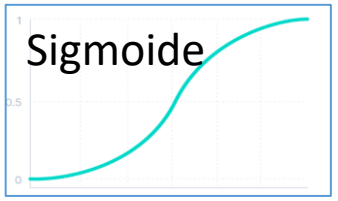
- Les fonctions d'activations déterminent les conditions du *switch-on*

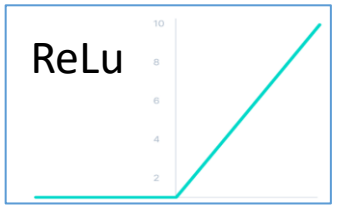
# Fonctions d'activations

Sans activation, un reseau de neurone est une application linéaire incapable de résoudre un problème complexe

- $la\ fonction\ seuil : \sigma(z, \rho) = 1_{z \geq \rho}$
- $la\ fonction\ sigmoide : \sigma(z) = \frac{1}{1 + e^{-z}}$
- $la\ fonction\ softmax : \sigma(z_i) = \frac{\exp z_i}{\sum_k \exp z_k}$
- $la\ fonction\ ReLu\ (rectified\ linear\ unit) : \sigma(z) = max(0, z)$




Choix en fonction de:

- Type de problème à résoudre (classification/régression)
- La dérivabilité (gradient) qui joue un rôle clé dans l'apprentissage (rétro-propagation)

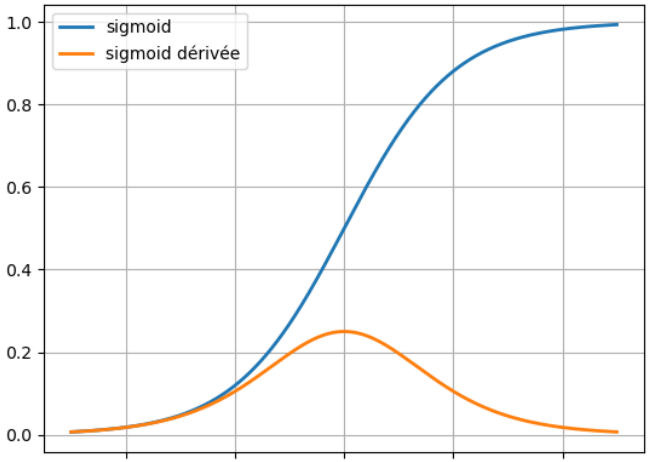
# Activation par une Sigmoide

La fonction sigmoide est populaire car dérivable et interprétable en terme de probabilités.

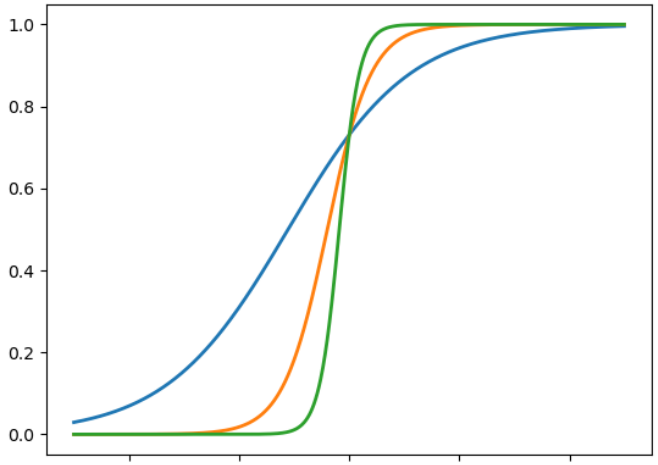
$$\sigma(x, \omega, b) = \frac{1}{1 + e^{-(x\omega + b)}}$$

Les poids ω et le biais b permettent de contrôler la pente et le seuil d'activation dans son ensemble.

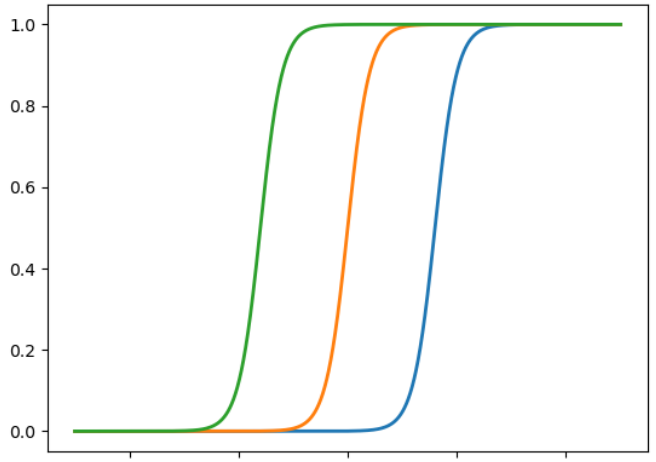
Activation et sa dérivée:




Variation du poid ω:



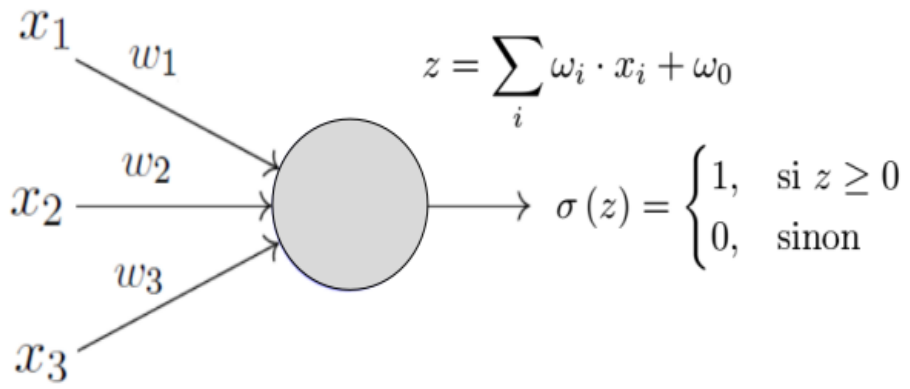

Variation de b (le biais):



# Les débuts, unité de base perceptron

Le perceptron est formé d'un neurone unique avec activation binaire*. Son inventeur Rosenblatt (1957) voulait en faire une machine (plutôt qu'un algo):

Navy Press Conf 1958: *..the embryo of an electronic computer that will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.*

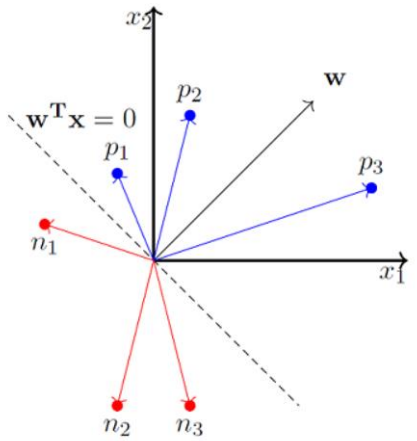



**Algorithm:** Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
**while** !*convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$ ;
  **if** $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
  **end**
  **if** $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
  **end**
**end**
//the algorithm converges when all the inputs are classified correctly

Le but est de trouver un plan qui classe les points en étudiant le cosinus de l'angle entre chaque point (x1,x2,x3) avec (ω1,ω2,ω3) en ajustant celui-ci de manière à :

- bleu --> cosinus positif
- rouge --> cosinus négatif



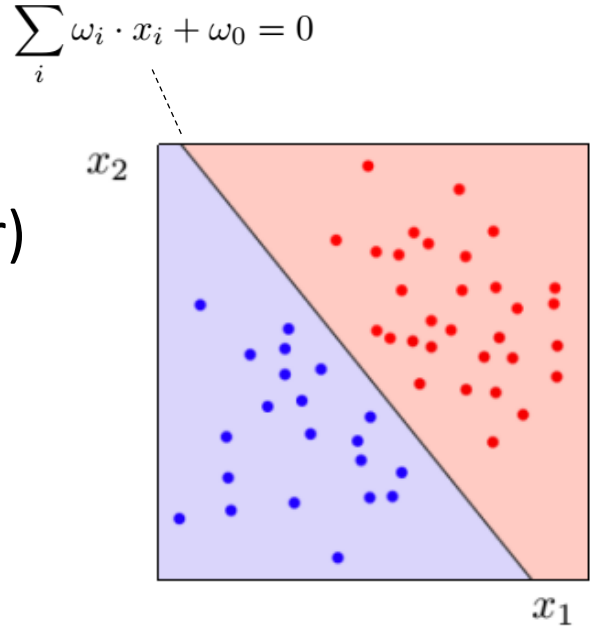

*Avec une activation sigmoide le perceptron est une regression Logistique

# Evolution vers les réseaux de neurones multi-couches

En 1969, Minsky-Papert  pointent certaines limitations du perceptron:

- Classification binaire seulement
- Problèmes linéairement séparables
- Fonction logique XOR (or exclusif) non couverte

En pratique lorsque l'algo du perceptron converge, il permet de trouver les poids qui définisent un hyper-plan séparant les données:



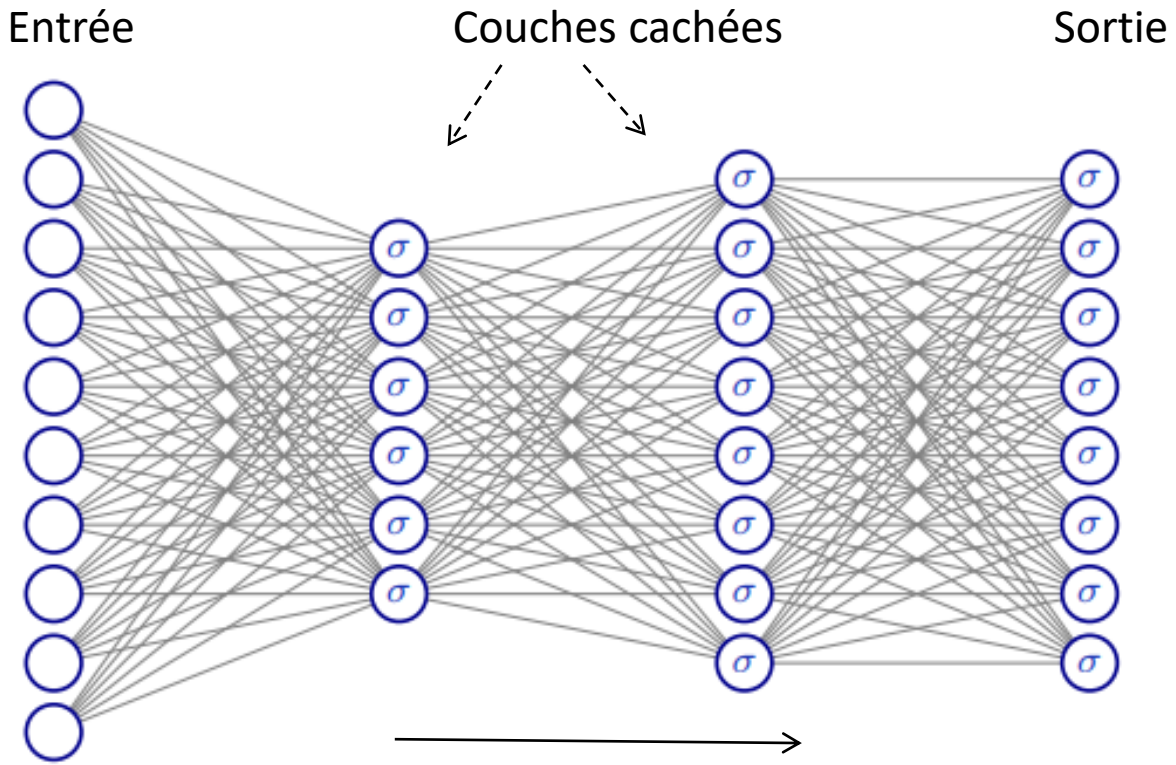

Ralentissement  dans la recherche jusqu'à ce qu'on montre:

- Un assemblage de perceptron permet de traîter XOR (multi-layer)
- Un assemblage de neurones dans un *réseau de neurones à propagation avant* (ou multi-layer perceptron) permet le traitement de problèmes complexes
- Werbos (1974) et LeCun (1989) entre autres montrent comment un réseau peut être entrainé par rétro-propagation

# Multi-Layer Perceptron

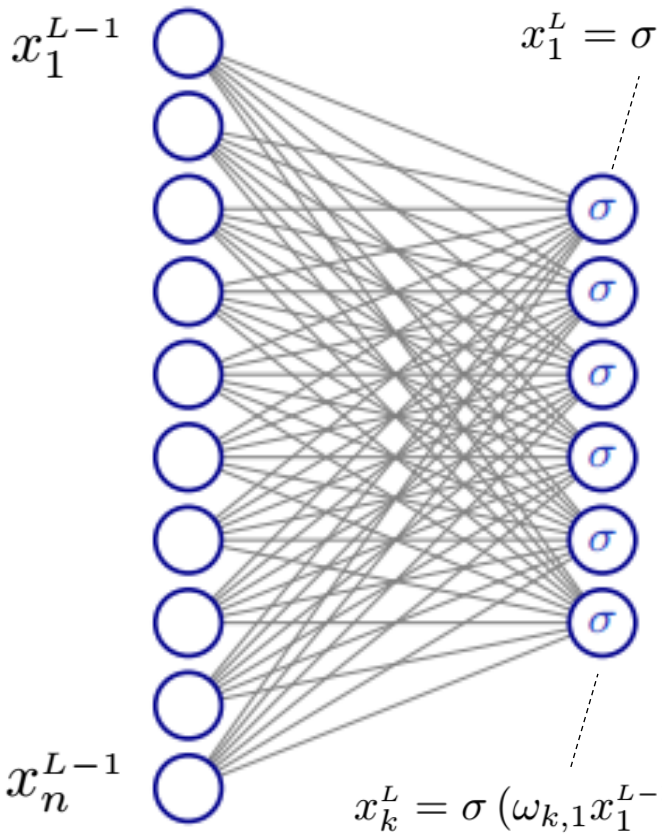
Réseau composé de couches successives:

- Une couche est un ensemble de neurones n'ayant pas de connexion entre eux
- Une couche communique un résultat à la prochaine couche (propagation avant)



# Propagation avant - une étape

A chaque étape, le réseau transforme les variables de la couche L-1 vers de nouvelles variables sur la couche L:




$$x_1^L = \sigma\left(\omega_{1,1}x_1^{L-1} + .. + \omega_{1,n}x_n + b_1\right)$$

$$x_k^L = \sigma\left(\omega_{k,1}x_1^{L-1} + .. + \omega_{k,n}x_n^{L-1} + b_k\right)$$

En notation matricielles on peut écrire:

$$\begin{pmatrix} x_1^L \\ \vdots \\ x_k^L \end{pmatrix} = \sigma\left( \begin{bmatrix} \omega_{1,1} & .. & \omega_{1,n} \\ & \vdots & \\ \omega_{k,1} & .. & \omega_{k,n} \end{bmatrix} \begin{bmatrix} x_1^{L-1} \\ \vdots \\ x_n^{L-1} \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_k \end{bmatrix} \right)$$

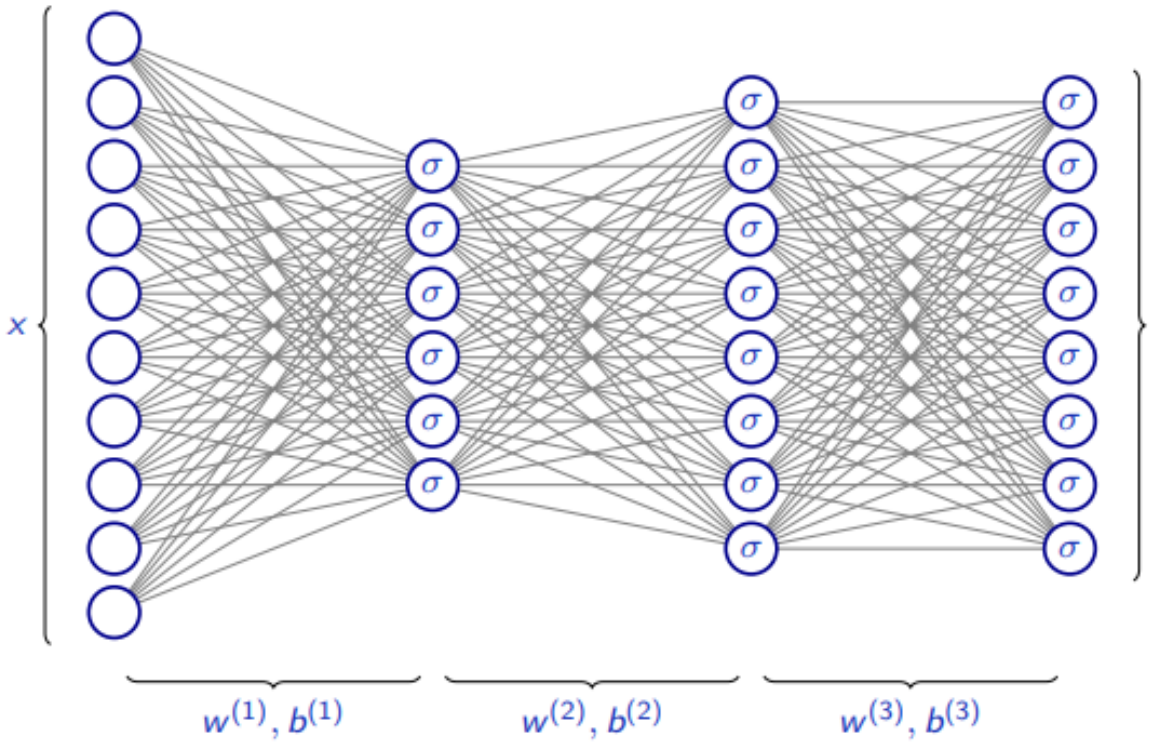
Donc pour une couche la transformation est:

$$\boxed{x^L = \sigma(wx^{L-1} + b)}$$

# Forward pass (Propagation avant)

Pour l'ensemble du réseau on peut formaliser la propagation de la manière suivante:
Pour les couches $L = 0, .., L_{out}$ avec $x^0 = x$ (la couche zéro est l'entrée) on propage selon

$$x^L = \sigma(w^L x^{L-1} + b^L)$$




$$\begin{aligned} x^3 &= \sigma\left(w^3 x^2 + b^3\right) \\ &= \sigma\left(w^3 \sigma\left(w^2 x^1 + b^2\right) + b^3\right) \\ &= \sigma\left(w^3 \sigma\left(w^2 \sigma\left(w^1 x + b^1\right) + b^2\right) + b^3\right) \\ &= f(x, w, b) \end{aligned}$$

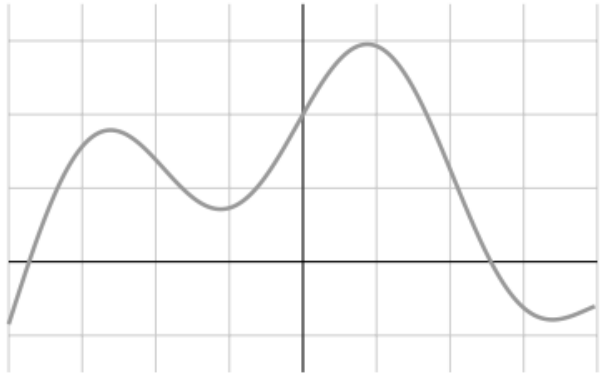
- Le réseau peut-être vu comme une fonction de x, avec paramètres ω, b
- Le *forward pass* est succession de compositions de fonctions

Note: Si l'activation est linéaire alors le réseau est simplement une transformation linéaire avec une paramétrisation bizarre

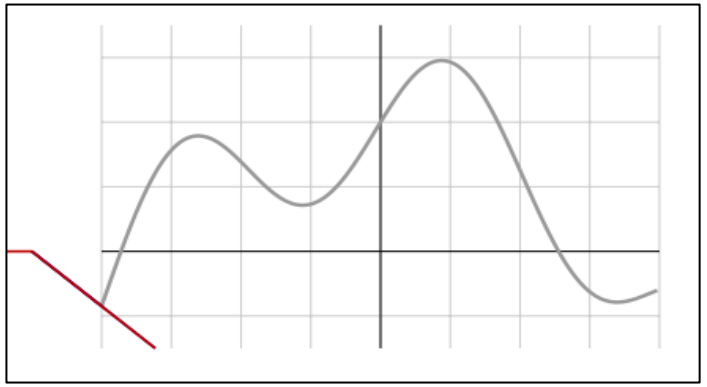
# Approximation universelle

Il est démontré  (Hornik 1991 et Cybenko 1989 pour la sigmoide) que toute fonction suffisamment régulière (continue,..) peut-être approximée par un réseau de neurones (type MLP) avec une unique couche cachée et un nombre de neurones finis.

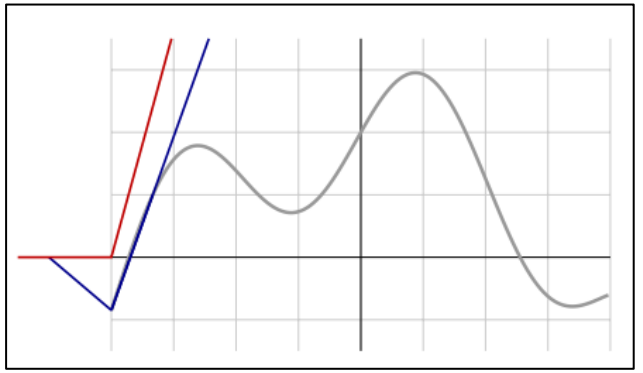
Supposons qu'on veuille approximer le comportement suivant en dimension une.
On utilise l'activation ReLu:



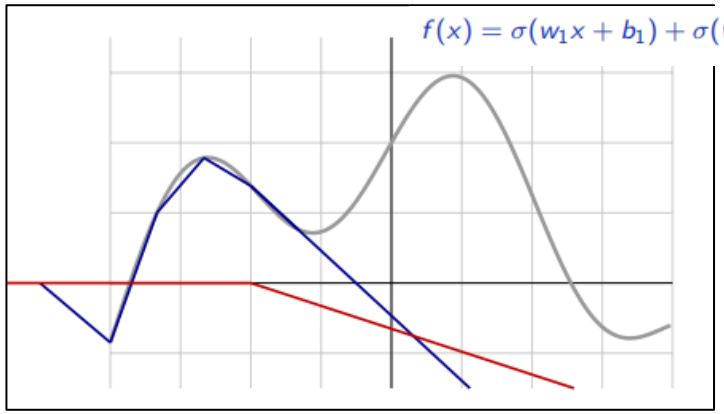
$f(x) = \sigma(w_1 x + b_1)$

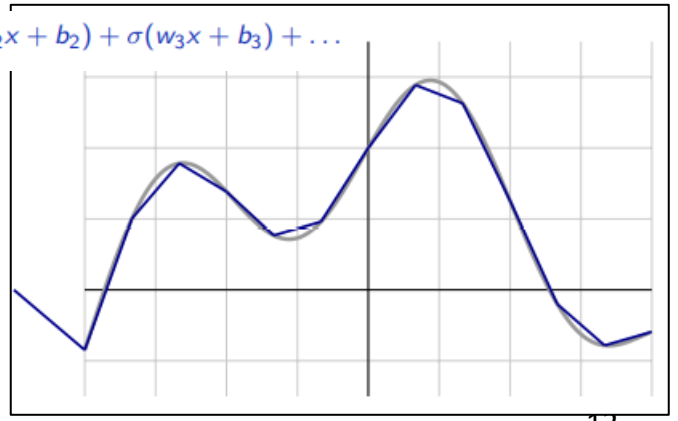


$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2)$



$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$





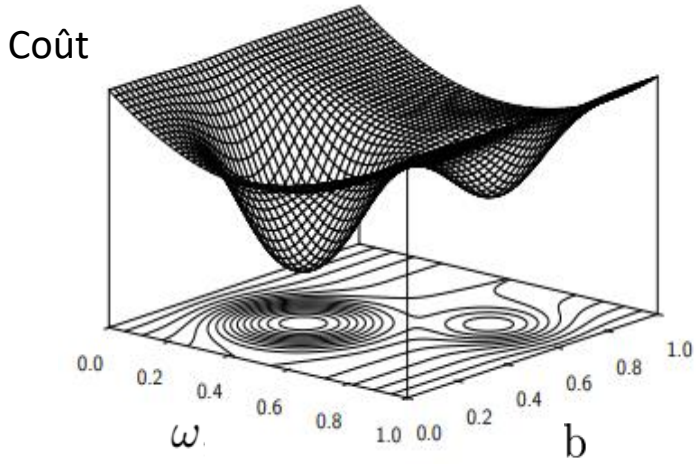
# Apprentissage

Entraîner un réseau de neurones consiste à déterminer les paramètres w et b en minimisant une fonction de coût à l'aide d'un échantillon d'apprentissage:

1. On a donc le modèle qui pour chaque *x* fournit une prédiction: $\hat{y} = f(x, \omega, b)$

2. On compare cette prédiction à la valeur cible *y* puisque l'on dispose d'un ensemble d'entraînement $\mathcal{A} = (x_i, y_i)$ . Par exemple on peut minimiser la fonction des moindres carrés (ou d´autres):

$$\mathcal{L}(\omega, b) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - f(x_i, \omega, b))^2$$

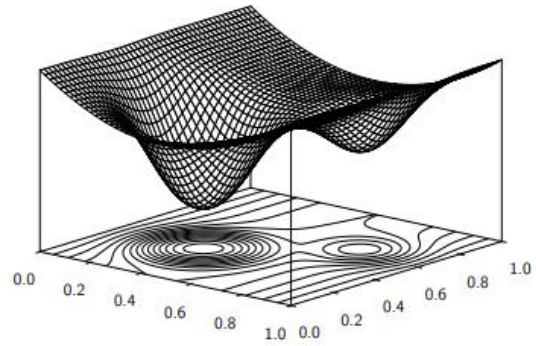


# Descente de Gradient

La descente de gradient est un algorithme qui utilise la connaissance locale (dérivée première) de la variation d'une surface pour décider de la direction de progression à chaque pas dans le but d'atteindre le point le plus bas

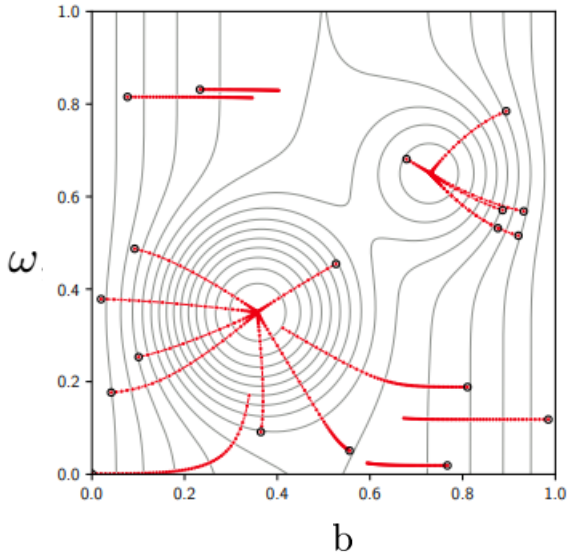
Le gradient indique la direction de la plus grande pente, donc on empreinte la direction opposé.

$$\nabla \mathcal{L} = \left( \frac{\partial \mathcal{L}}{\partial x_1}, \frac{\partial \mathcal{L}}{\partial x_2}, \ldots, \frac{\partial \mathcal{L}}{\partial x_n} \right)^{\mathrm{T}}$$



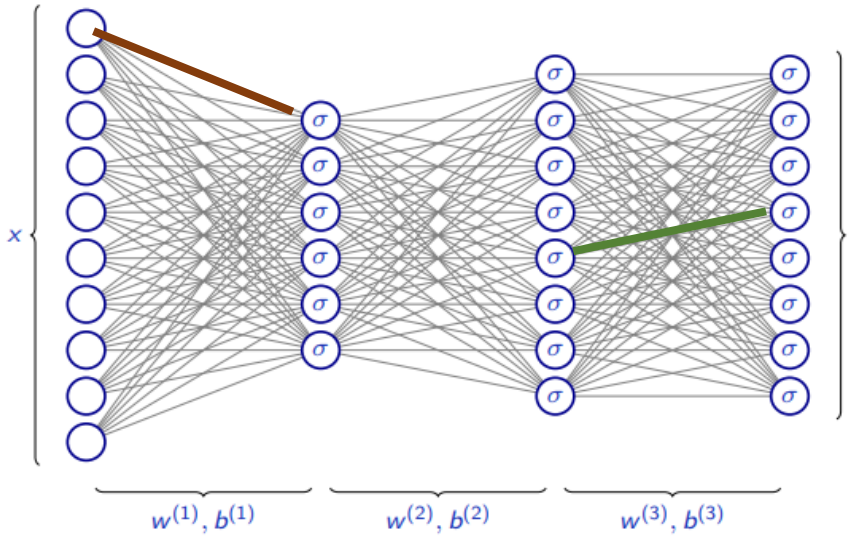

**Algorithme:** (Descente de Gradient)
Choisir un point initial $\mathbf{x}_0$ (arbitraire)
**Répéter :**

1. Déterminer une direction de descente : $-\nabla f(\mathbf{x}_k)$,
2. Choisir une taille de pas $\lambda_k > 0$,
3. Actualiser $\mathbf{x}_{k+1} = \mathbf{x}_k - \lambda_k \cdot \nabla f(\mathbf{x}_k)$,
4. Tester le critère d'arrêt.



# Calcul du gradient pour un réseau de neurones

On cherche à calculer la dérivée de la fonction de coût pour tous les paramètres du réseau le but étant de réduire l'erreur de prédiction en sortie




$$\ell = \frac{1}{2}(y - f(x, \omega, b))^2 \qquad \frac{\partial \ell}{\partial \omega_{ij}^L}, \qquad \frac{\partial \ell}{\partial b_i^L}$$

Approche naive: dérivée numérique pour chaque variable

$$\frac{\partial \ell}{\partial \omega_{ij}^L} \simeq \frac{\ell(\omega_{i,j}, b) - \ell(\omega_{i,j} + \epsilon, b)}{\epsilon}$$

Nécessite un nouveau *forward pass* pour chaque paramètre!

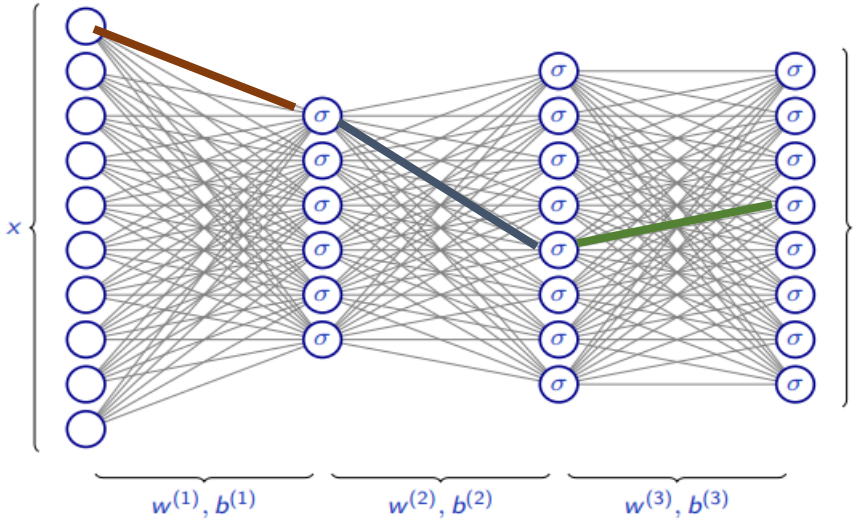
# Calcul du gradient par composition de fonctions

Les paramètres dans les couches cachées influencent l'erreur de manière indirecte car le *forward pass* est une composition de fonctions: $x^L = \sigma(s^L) \quad s^L = w^L x^{L-1} + b^L$

$$f(x, w, b) = \sigma\left(s^3\right) = \sigma\left(w^3\sigma\left(s^2\right) + b^3\right) = \sigma\left(w^3\sigma\left(w^2\sigma\left(s^1\right) + b^2\right) + b^3\right) = \sigma \circ h \circ \ g \circ s^1$$

Le gradient pour les couches de sorties sont accessibles plus facilement

Le gradient pour les couches internes s'obtiennent par dérivation composée




Dérivation de fonctions composée:

$$(g \circ f) = g(f(x)) = \sin(x^3) \qquad g(y) = \sin(y) \qquad f(x) = x^3$$

$$(g \circ f)' = (g' \circ f)f' = \cos(x^3)3x^2$$

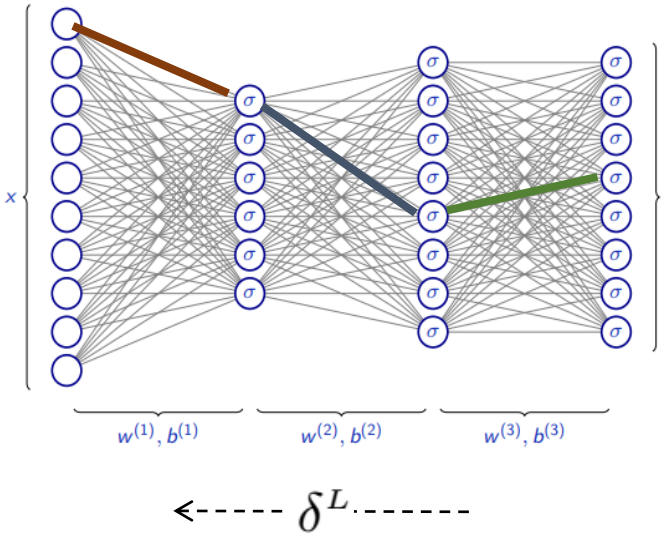
La dérivation par rapport à la variable *x* la plus *profonde,* nécessite de connaitre la dérivé des fonctions *externes:* on procède en quelque sorte de la fin vers le début

# Rétro-propagation de l'erreur

En pratique on implémente la rétro-propagation en appliquant la *chain rule* sur $s^L$

$$\frac{\partial \ell}{\partial \omega_{ij}^L} = \frac{\partial \ell}{\partial s^L}\frac{\partial s^L}{\partial \omega_{ij}^L} = \delta^L x^{L-1}$$

Terme d'erreur




Calcul du terme erreur pour chaque couche:

- Couche externe (niveau 3) : $\delta^3 = (y - f(x, \omega, b))\sigma'(s^3)$
- Couches internes:

$$\delta_j^L = \frac{\partial \ell}{\partial s^{(L)}} = \sum_{neuron} \frac{\partial \ell}{\partial s^{L+1}}\frac{\partial s^{L+1}}{\partial s^L} = \sum_{neuron} \delta^{L+1}\frac{\partial s^{L+1}}{\partial s^L} = \sigma'(s_j^L)\sum_k \delta_k^{L+1}\omega_{jk}^{L+1}$$

Back-prop équation:

$$\delta^L = \sigma'(s^L) \sum_{neuron} \delta^{L+1}\omega^{L+1}$$

# Algorithme - Descente de gradient avec back-prop

**Algorithme:** (Back-Prop)
Initialiser les paramètres $(\omega, b)$
**Répéter :**

1. Pour chaque échantillon $u$, exécuter le *Forward pass*. Stocker $f(x_u, \omega, b)$, $s^L$ et $x^L$ pour chaque couche $L$.
2. Pour chaque échantillon $u$, exécuter le *Backward pass* : calculer les composantes du gradient $\nabla \ell_u$

$$\frac{\partial \ell_u}{\partial \omega_{ij}^L} \quad \text{et} \quad \frac{\partial \ell_u}{\partial b_i^L} \quad \text{avec} \quad \ell_u = \frac{1}{2}(y_u - f(x_u, \omega, b))^2,$$

   en partant de l'erreur obtenue à la sortie :
   a. Calculer l'erreur de la couche externe.
   b. Propager cette erreur vers l'arrière en utilisant l'équation de back-prop pour chaque couche,
   c. Calculer le gradient pour chaque couche.
3. Calculer $\nabla \mathcal{L}(\omega, b)$ en appliquant la moyenne par composantes du gradient pour chaque échantillon $u$ :

$$\nabla \mathcal{L}(\theta, x, y) = \frac{1}{m} \sum_u \nabla \ell_u.$$

4. Actualiser tout les paramètres :

$$(\omega_{t+1}, b_{t+1}) = (\omega_t, b_t) - \lambda \nabla \mathcal{L}(\omega, b).$$

4. Tester critère d'arrêt.