Introduction à l'apprentissage et aux réseaux de neurones

Apprentissage Profond

A. Carlier

2021

Plan du cours

Une définition de l'apprentissage

L'apprentissage machine est un domaine d'étude qui donne aux ordinateurs la capacité d'apprendre sans être explicitement programmé.

Arthur Samuel (1959)

On dit d'un programme informatique qu'il apprend de son expérience E, par rapport à une tâche T et une mesure de performance P, si sa performance P sur la tâche T augmente avec l'expérience E.

Tom Mitchell (1998)

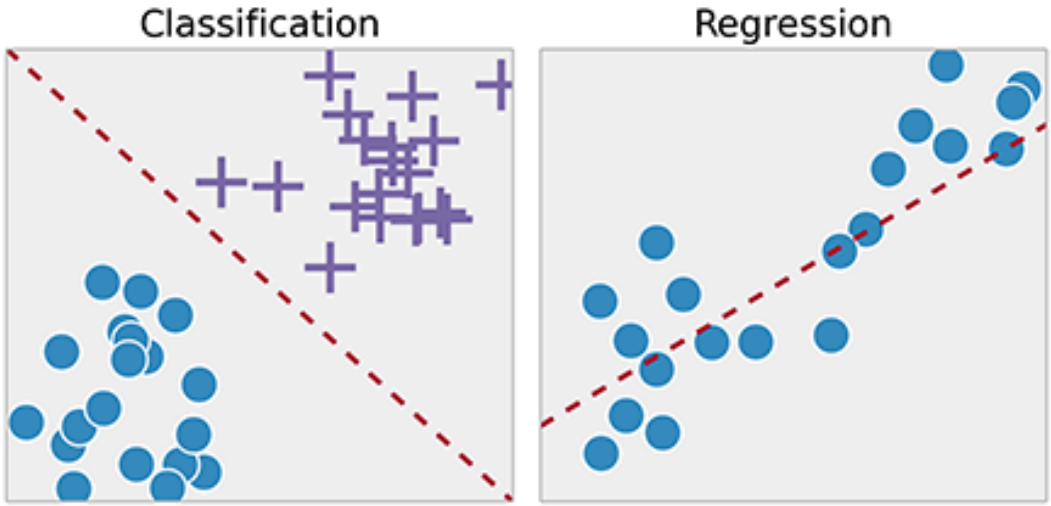
3 grands paradigmes d'apprentissage

Apprentissage supervisé

- Un oracle (expert) fournit un ensemble d'apprentissage (données, labels) :

$$\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), .., (\mathbf{x}^{(m)}, y^{(m)})\}.$$

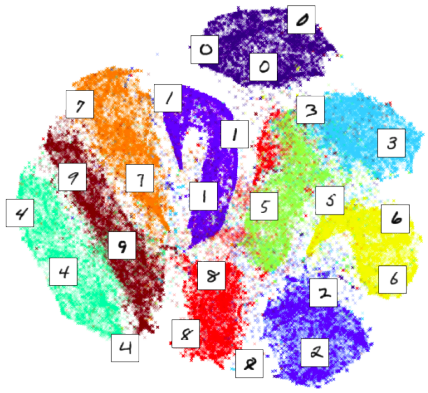
- Recherche d'un prédicteur qui minimise la différence entre labels réels et prédits
- Souvent coûteux car nécessite l'annotation de larges bases de données



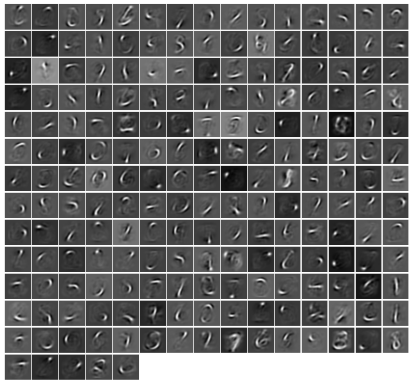
3 grands paradigmes d'apprentissage

Dans le cadre de l'**apprentissage non supervisé**, on cherche à inférer de l'information à partir d'observations uniquement :

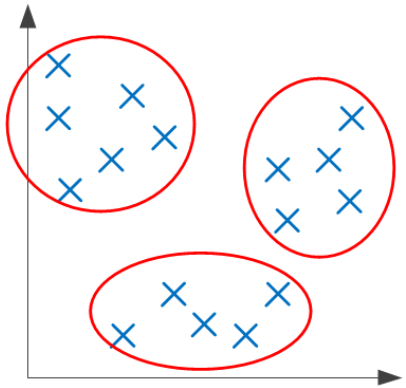
$$\mathcal{D} = \{\mathbf{x}^{(1)}, .., \mathbf{x}^{(m)}\}.$$



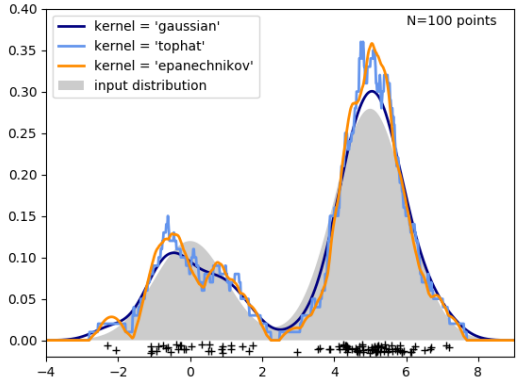
Réduction de dimension



Extraction de caractéristiques



Clustering

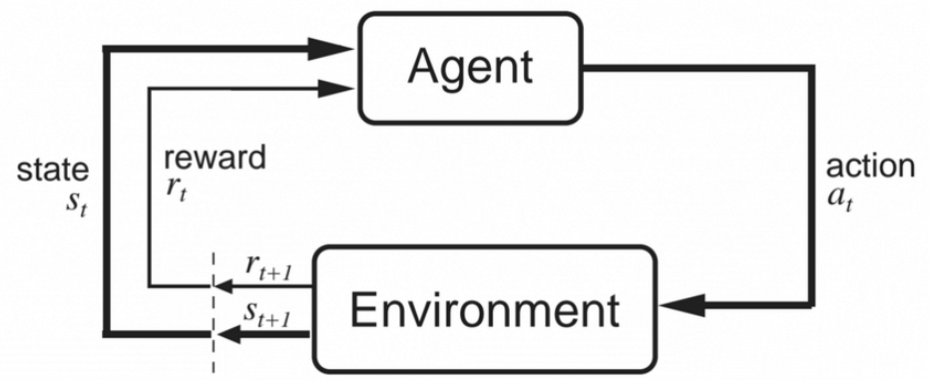


Estimation de densité

3 grands paradigmes d'apprentissage

Apprentissage par renforcement

- Définition d'un **agent** comportemental, qui peut prendre un ensemble de décisions (**actions**) en fonction de l'**état** d'un certain système
- L'agent obtient des **récompenses** pour chacune de ses actions
- L'objectif est d'apprendre une **politique**, c'est-à-dire une fonction pour déterminer l'action optimale à effectuer en fonction du contexte (état)



Mais aussi...

- **Apprentissage faiblement supervisé.** La faible supervision peut avoir plusieurs origines :
  - Un trop petit nombre d'annotations (*few-shots*/*one-shot learning*)
  - Les annotations sont trop bruitées, ou imprécises.
- **Apprentissage semi-supervisé**.
  - On dispose à la fois d'un faible nombre de données annotées et d'un grand nombre de données non-annotées.
- **Apprentissage actif**.
  - On part d'un ensemble de données dont seulement certaines sont annotées.
  - Le modèle doit déterminer, au travers d'un certain critère à définir, quelles données vont potentiellement lui fournir le plus d'informations pour évoluer vers de meilleures performances.
  - Un "oracle" (annotateur humain) peut alors annoter ces données, et le modèle s'adapter grâce à ces nouvelles informations.

Plan du cours

Apprentissage supervisé

Dans le cadre de l'**apprentissage supervisé**, on dispose d'observations et de leurs étiquettes (appelées encore cibles (*targets*), catégories ou *labels*) qui constituent un ensemble d'apprentissage. On le note :

$$\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), .., (\mathbf{x}^{(m)}, y^{(m)})\}.$$

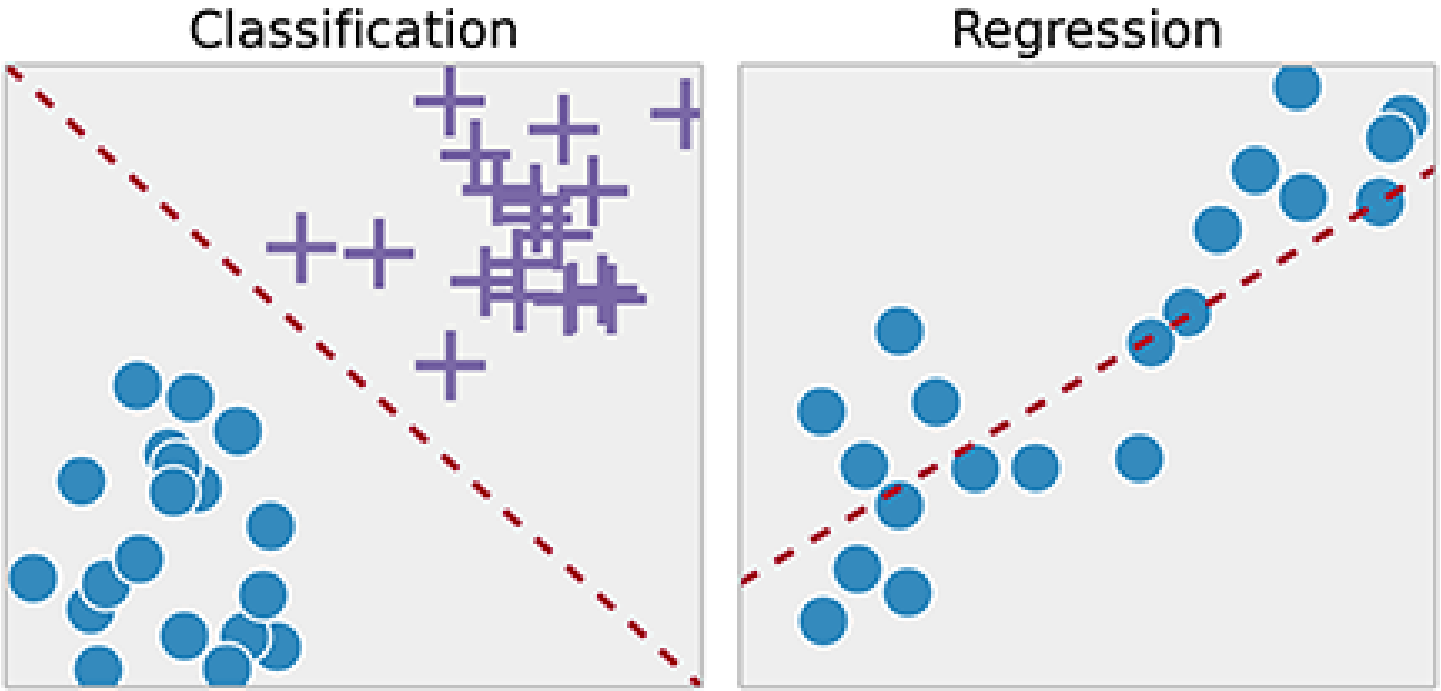
Les labels permettent d'enseigner à l'algorithme à établir des correspondances entre les observations et les labels.

Apprentissage supervisé

Il existe deux principaux types d'apprentissage supervisé :

- **Classification :** Assigner une catégorie à chaque observation :
  - Les catégories sont discrètes
  - La cible est un indice de classe : $y \in \{0, .., K-1\}$
  - Exemple : reconnaissance de chiffres manuscrits :
    - ★ $\mathtt{x}$: vecteur ou matrice des intensités des pixels de l'image
    - ★ y : identité du chiffre
- **Régression :** Prédire une valeur réelle à chaque observation :
  - les catégories sont continues
  - la cible est un nombre réel $y \in \mathbb{R}$
  - Exemple : prédire le cours d'une action
    - ★ $\mathtt{x}$: vecteur contenant l'information sur l'activité économique
    - ★ y : valeur de l'action le lendemain

Classification et Régression



$\Rightarrow$ Régression

Régression linéaire

Soit l'ensemble $\mathcal{D}$ contenant m exemples d'apprentissage.

Pour l'exemple, $\mathbf{x}^{(i)}$, le modèle linéaire s'écrit :

$$\hat{y}^{(i)} = \theta^T \mathbf{x}^{(i)}$$

et la fonction de coût quadratique s'écrit :

$$(\hat{y}^{(i)} - y^{(i)})^2$$

$\rightarrow$ Formulation aux **Moindres Carrés** !

Trouver θ qui minimise la perte sur tous les exemples d'apprentissage (fonction objectif $J(\theta)$) :

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2 = J(\theta) \tag{1}$$

Régression linéaire

Soit l'ensemble $\mathcal{D}$ contenant m exemples d'apprentissage en dimension d (d variables), on définit :

- $\mathtt{X} \in \mathbb{R}^{m \times d}$ matrice des données
- $\mathtt{y} \in \mathbb{R}^m$: vecteur de cibles
- $\hat{\mathtt{y}} \in \mathbb{R}^m$: vecteur de prédictions avec $\hat{y} = \mathtt{X}\theta$
- $\theta \in \mathbb{R}^d$ vecteur des paramètres du modèle à estimer

Régression aux moindres carrés

Estimer le modèle linéaire θ entre les données et les cibles en minimisant la somme des résidus quadratiques :

$$\min_{\theta} \|\mathtt{X}\theta - \mathtt{y}\|^2 = J(\theta)$$

Régression linéaire

Résolution des problèmes aux moindres carrés :

Condition Nécessaire du premier ordre :

$$\frac{dJ(\theta)}{d\theta} = 0 = 2\mathrm{X}^T(\mathrm{X}\theta - \mathrm{y}).$$

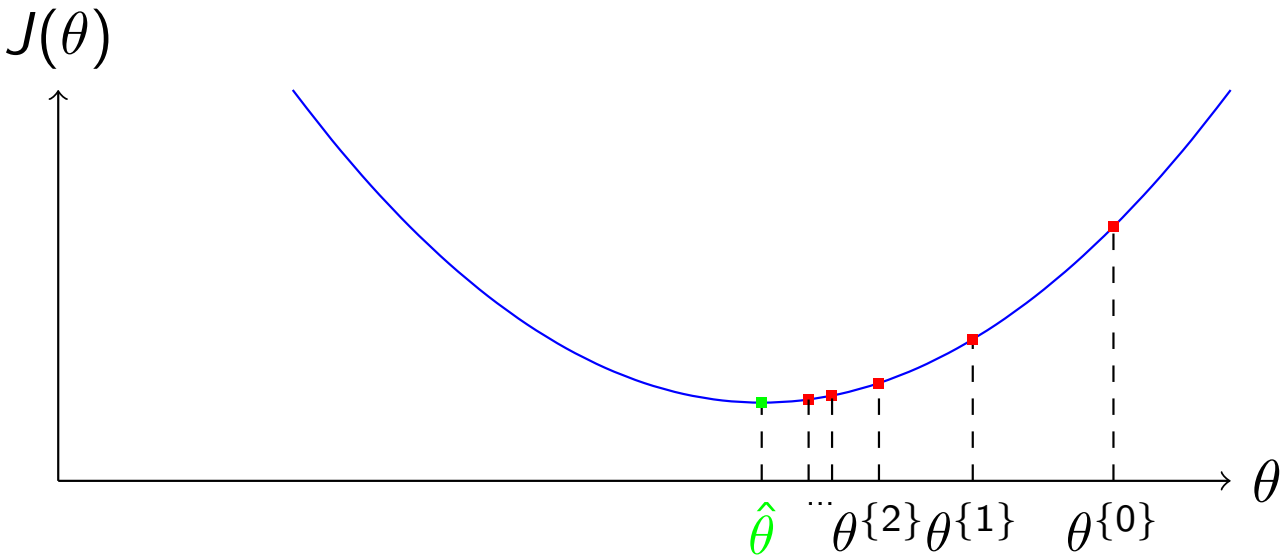
Si $det(\mathrm{X}^T\mathrm{X}) \neq 0$, la solution analytique est :

$$\hat{\theta} = (\mathrm{X}^T\mathrm{X})^{-1}\mathrm{X}^T\mathrm{y}$$

En pratique, calculs coûteux (inversion de matrice) donc solution itérative : **descente de gradient** !

Remarque : les problèmes aux moindres carrés sont **convexes** $\rightarrow$ minimum local est global !

Régression linéaire



Algorithme : Descente de gradient $(\mathcal{D}, \alpha)$

Initialiser $\theta^{\{0\}} \leftarrow 0$, $k \leftarrow 0$

TANT QUE pas convergence **FAIRE**

  POUR j de 1 à d **FAIRE**

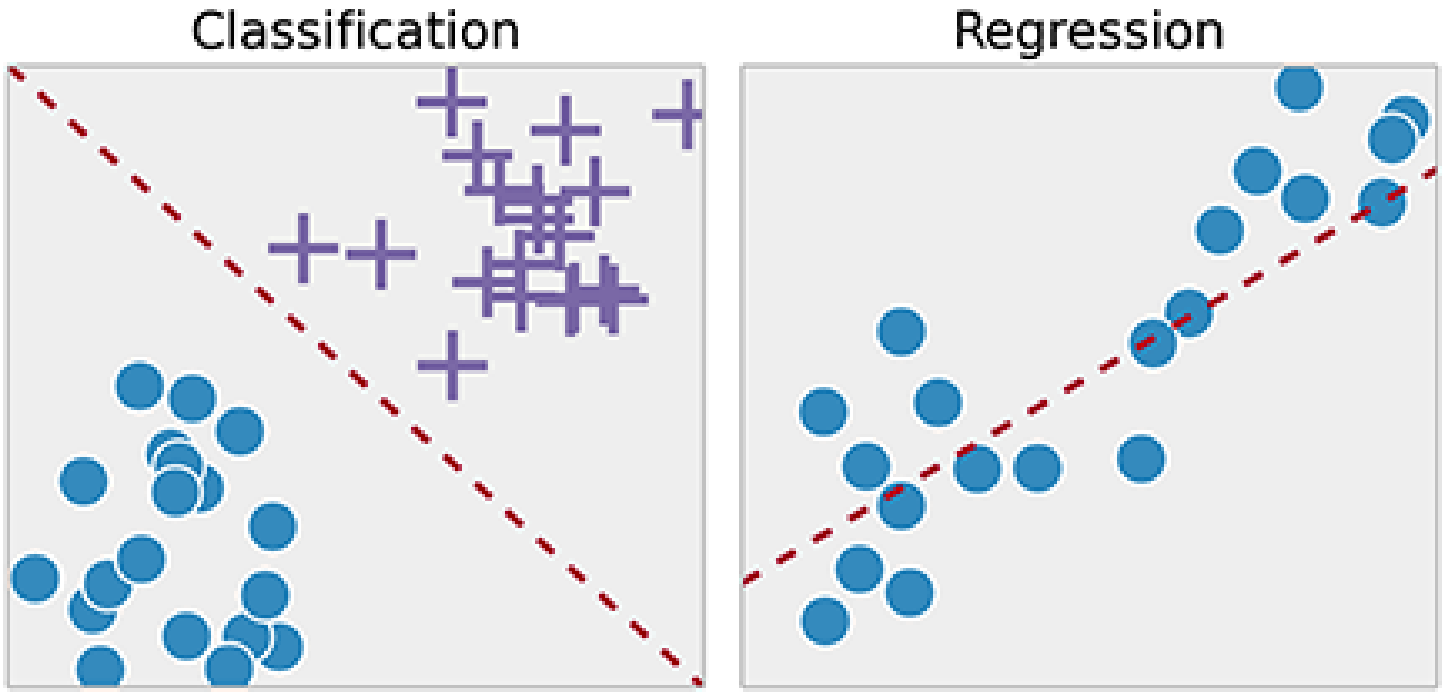
$$\theta_j^{\{k+1\}} \leftarrow \theta_j^{\{k\}} - \alpha \frac{\partial J(\theta^{\{k\}})}{\partial \theta_j}$$

  FIN POUR

  $k \leftarrow k + 1$

FIN TANT QUE

Classification et Régression



$\Rightarrow$ Classification

Régression logistique

En sortie, la sortie du modèle peut être :

- **binaire** : échec/succès, 0/1, -/+
  $\Rightarrow$ fonction **sigmoïde ou logistique**
- **multinomiale** (multi-classes) : chiffres
  $\Rightarrow$ fonction **softmax**

Régression logistique : cas binaire

Comme avec la régression linéaire, on prend un modèle linéaire type :

$$z = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n = \theta^T \mathbf{x}$$

Ce modèle linéaire agit comme **séparateur** des 2 classes.
Puis on veut une probabilité : $0 \leq h_\theta(x) = g(z) \leq 1$ telle que :

- si $h_\theta(x) \leq 0.5$, alors la classe est 0 ($y = 0$)
- si $h_\theta(x) > 0.5$, alors la classe est 1 ($y = 1$)

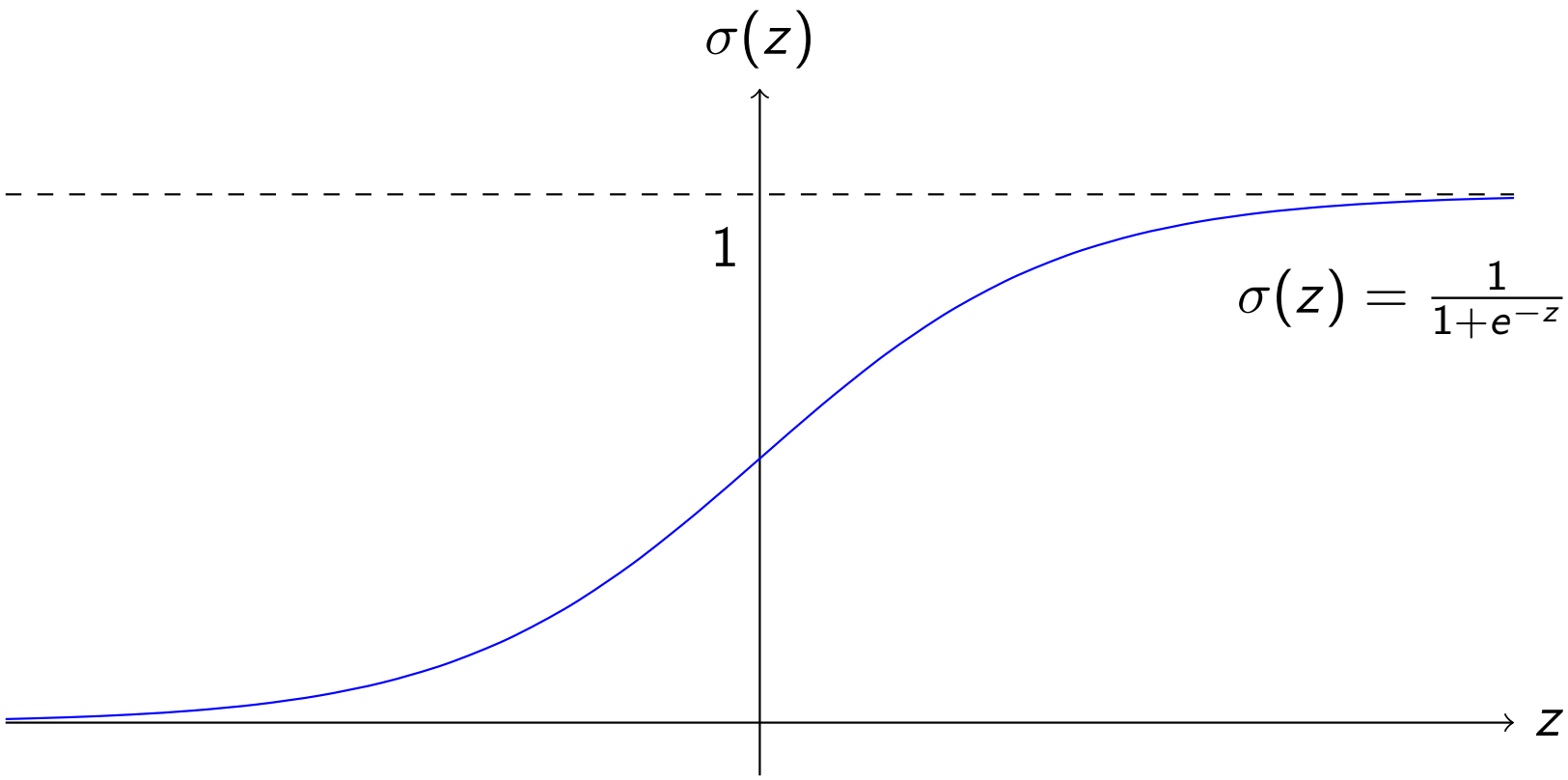
Quelle est cette fonction g telle que :

$$h_\theta(x) = g(z) = g(\theta^T \mathbf{x}) = P(y = 1|x; \theta) \ ?$$

Régression logistique : cas binaire

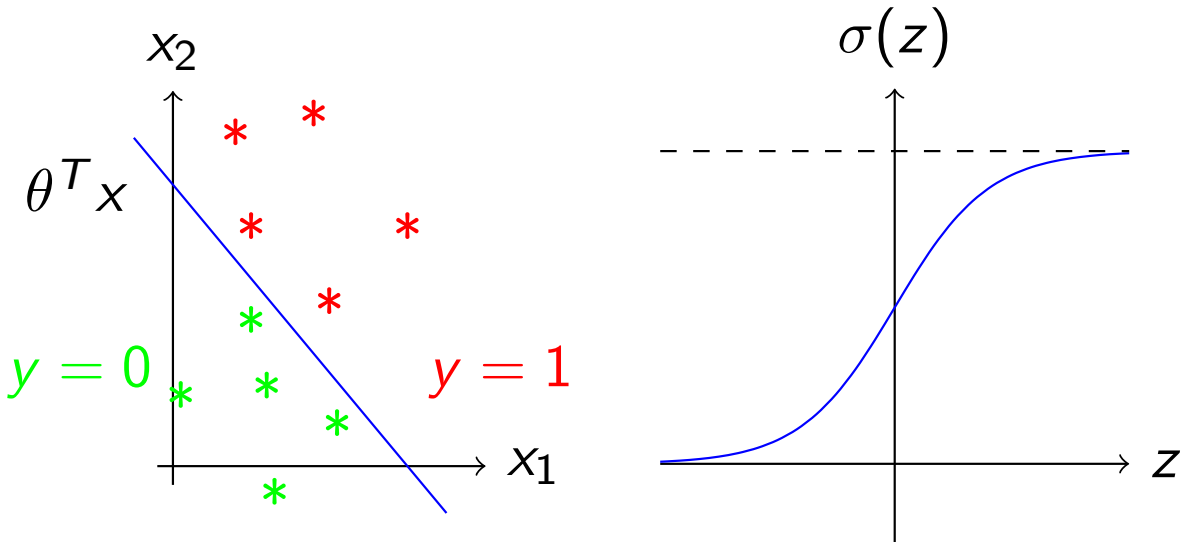
La **sigmoïde** ou **fonction logistique** définie par :

$$\sigma(z) = \frac{1}{1 + exp(-z)}$$



Régression logistique : cas binaire

Cas 2D où les données sont linéairement séparables

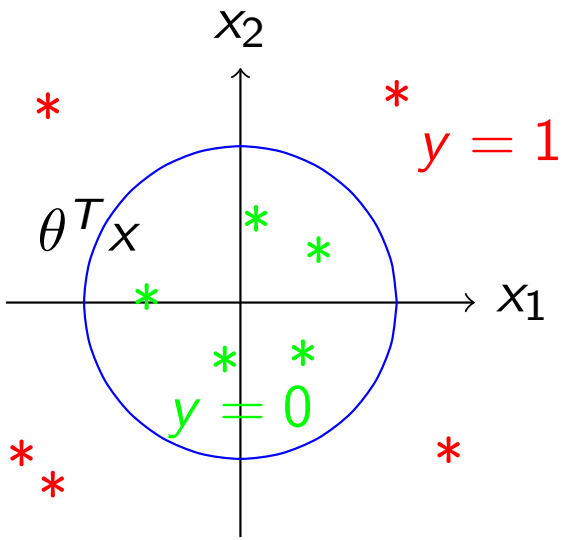


$$h_\theta(x) = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

- on prédit $y = 1$ si $-3 + x_1 + x_2 > 0$
- on prédit $y = 0$ si $x_1 + x_2 \leq 3$

Régression logistique : cas binaire

Cas 2D où les données sont linéairement séparables



$$h_\theta(x) = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$$

- on prédit $y = 1$ si $x_1^2 + x_2^2 > 1$
- on prédit $y = 0$ si $x_1^2 + x_2^2 \leq 1$

Régression logistique : cas binaire

Comment estimer automatiquement θ ?

On a un **corpus d'apprentissage** $\mathcal{D}$, contenant m exemples avec :

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \vdots \\ x_d^{(i)} \end{bmatrix} \in \mathbb{R}^{d+1} \text{ et } y^{(i)} \in \{0,1\}, \forall i \in \{1,..,m\}$$

et le **modèle** est défini par la fonction sigmoïde :

$$P(y=1|x;\theta) = \frac{1}{1+e^{-\theta^T x}}$$

Régression logistique : cas binaire

Il faut minimiser une **fonction objectif** à l'aide d'une technique d'optimisation (descente de gradient).

Peut on utiliser la même fonction de perte que pour la régression linéaire ?

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} perte\left(h_\theta(x^{(i)}), y^{(i)}\right)$$

avec

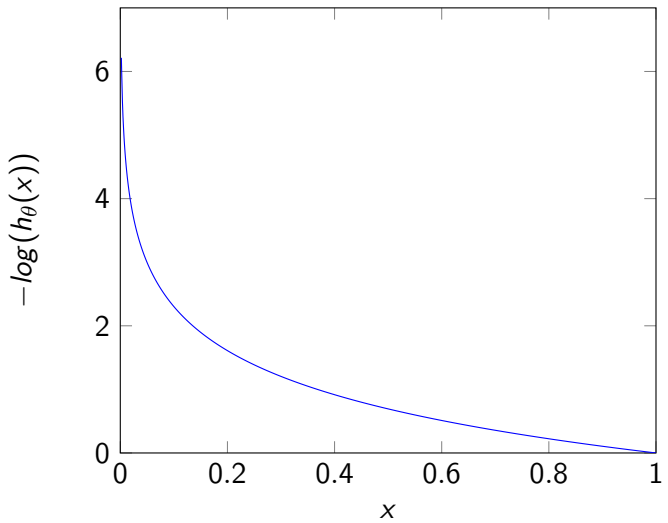
$$perte(h_\theta(x^{(i)}), y^{(i)}) = \frac{1}{2}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 = \frac{1}{2}\left(\frac{1}{1+e^{-\theta^T x^{(i)}}} - y^{(i)}\right)^2$$

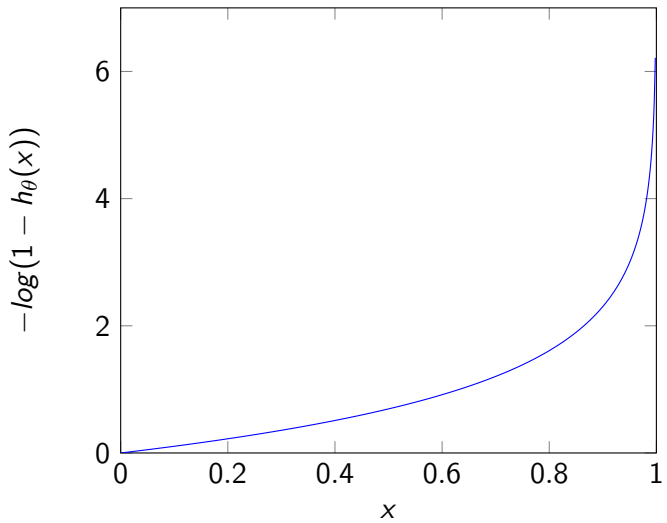
Cette fonction n'est **pas convexe** donc la descente de gradient ne garantit pas un minimum global !

Régression logistique : cas binaire

On introduit donc la **fonction de perte logistique ou entropie croisée (cross-entropy)** définie par :

$$\begin{aligned} perte(h_{\theta(x)}, y) &= \begin{cases} -log(h_\theta(x)) \text{ si } y = 1 \\ -log(1 - h_\theta(x)) \text{ si } y = 0 \end{cases} \\ &= -ylog(h_\theta(x)) - (1 - y)log(1 - h_\theta(x)) \end{aligned}$$





Régression logistique : cas binaire

Sur les m exemples, la fonction de perte devient :

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} -y^{(i)} log(h_\theta(x^{(i)})) - (1-y^{(i)}) log(1-h_\theta(x^{(i)}))$$

Pour minimiser cette fonction, on applique la descente de gradient.

$$\begin{aligned}
\frac{\partial J}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j}\frac{1}{m}\sum_{i=1}^{m} -y^{(i)} log(h_\theta(x^{(i)})) - (1-y^{(i)}) log(1-h_\theta(x^{(i)})) \\
&= \frac{1}{m}\sum_{i=1}^{m} -y^{(i)} \frac{\partial}{\partial \theta_j}(log(h_\theta(x^{(i)}))) - (1-y^{(i)}) \frac{\partial}{\partial \theta_j} log(1-h_\theta(x^{(i)})) \\
&= \frac{1}{m}\sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)} \qquad \text{(tous calculs faits)}
\end{aligned} \tag{2}$$

Précision sur le calcul précédent

Pour le calcul de $\frac{\partial}{\partial \theta_j}(log(h_\theta(x)))$, on pose $z = \theta^T x$, $u = \sigma(z)$ and $v = log(u)$

$$\begin{aligned}
\frac{\partial}{\partial \theta_j}(log(h_\theta(x))) &= \frac{\partial}{\partial \theta_j}(log(\sigma(\theta^T x))) \\
&= \frac{\partial v}{\partial u}\frac{\partial u}{\partial z}\frac{\partial z}{\partial \theta_j} && \text{chain-rule} \\
&= \frac{1}{\sigma(z)}\sigma(z)(1-\sigma(z))x_j && \text{car } \sigma'(z) = \sigma(z)(1-\sigma(z)) \\
&= (1-h_\theta(x))x_j
\end{aligned}$$

On obtient par le même procédé $\frac{\partial}{\partial \theta_j}(log(1-h_\theta(x))) = -h_\theta(x)x_j$

Régression logistique : cas multiclasse

Comment faire quand on a k **classes avec** $k > 2$?

On utilise la **fonction softmax** :

$$P(y = i|x, \theta) = \frac{e^{\theta_i^T x}}{\sum_{j=1}^{k} e^{\theta_j^T x}}$$

avec $y^{(i)} = [0 \ \ldots \ 0 \ 1 \ 0 \ldots \ 0]^T$ avec 1 à la $i$ème coordonnée.

Régression logistique : cas multiclasse

Pour chaque vecteur de données de test $x^{(i)}$, on calcule un vecteur de probabilités d'obtenir l'une des k classes.

$$h_\theta(x^{(i)}) = \begin{bmatrix} p(y^{(i)} = 1|x^{(i)};\theta) \\ p(y^{(i)} = 2|x^{(i)};\theta) \\ \vdots \\ p(y^{(i)} = k|x^{(i)};\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix}$$

Régression logistique : cas multiclasse

La fonction objectif devient :

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k}\mathbb{I}(y^{(i)}=j)log\left(\frac{e^{\theta_j^T x^{(i)}}}{\sum_{p=1}^{k} e^{\theta_p^T x^{(i)}}}\right) \quad (3)$$

Le gradient s'écrit :

$$\nabla_{\theta_j} J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\left(\mathbb{I}(y^{(i)}=j) - P(y^{(i)}=j|x^{(i)};\theta)\right)$$

Plan du cours

Perceptron monocouche

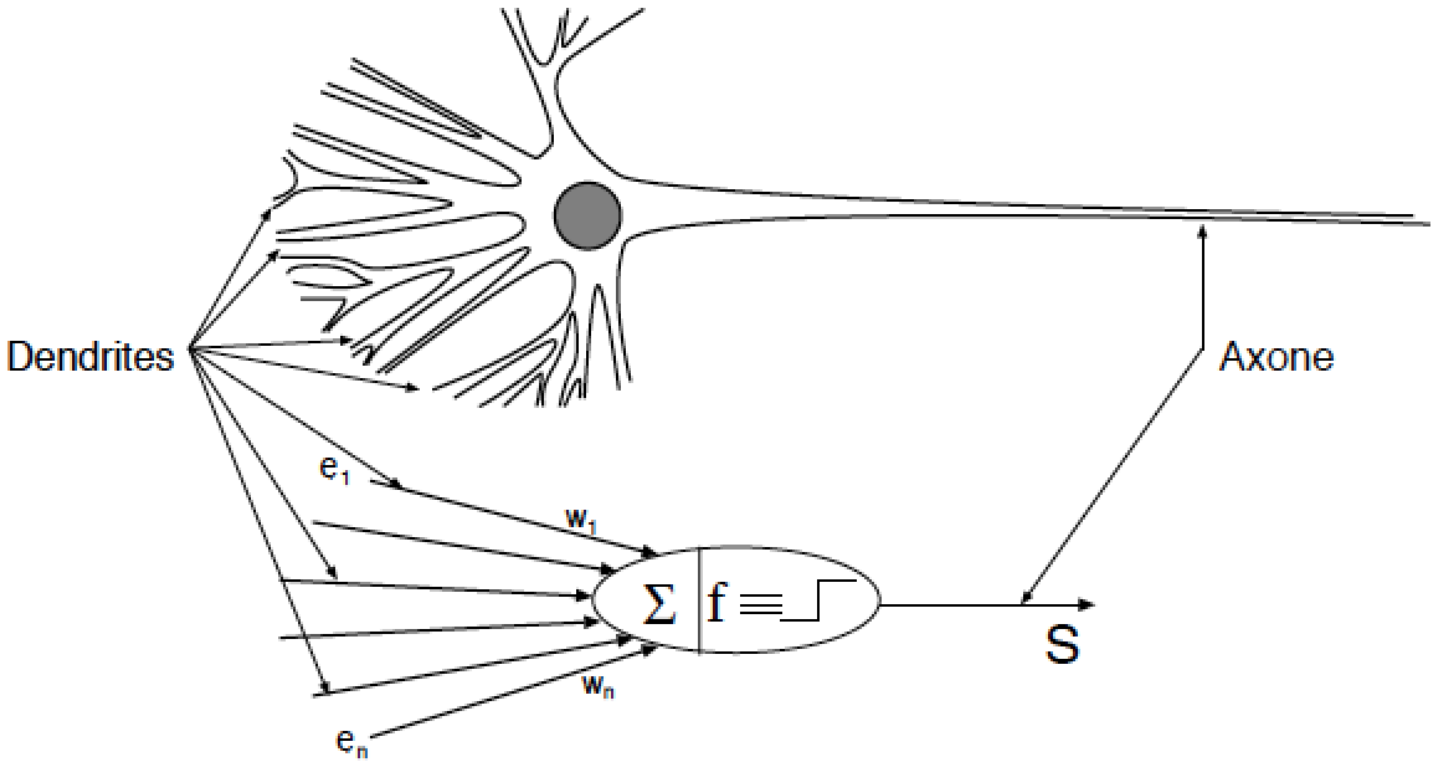
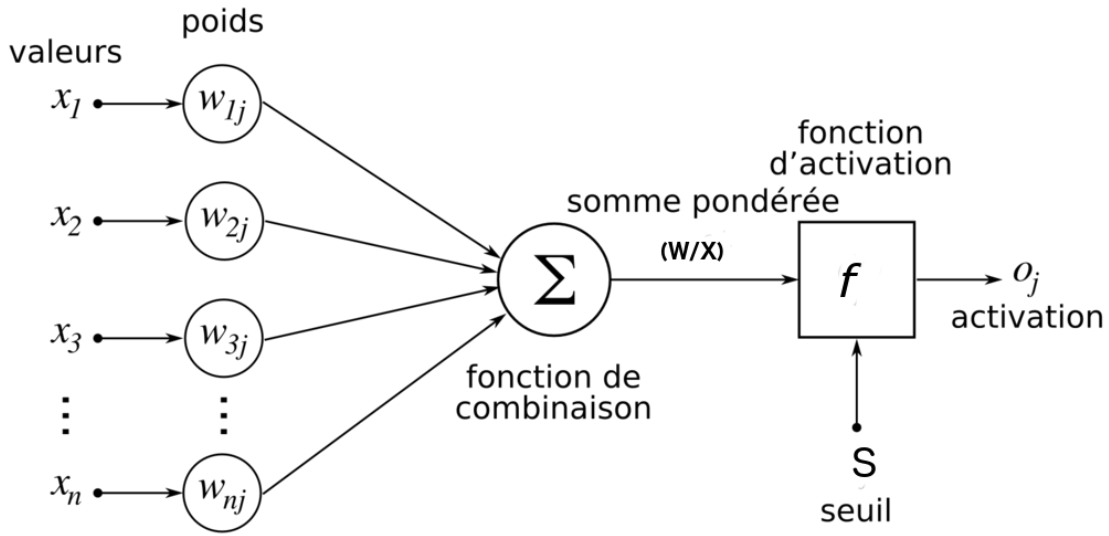


FIGURE – Structure d'un neurone biologique/artificiel

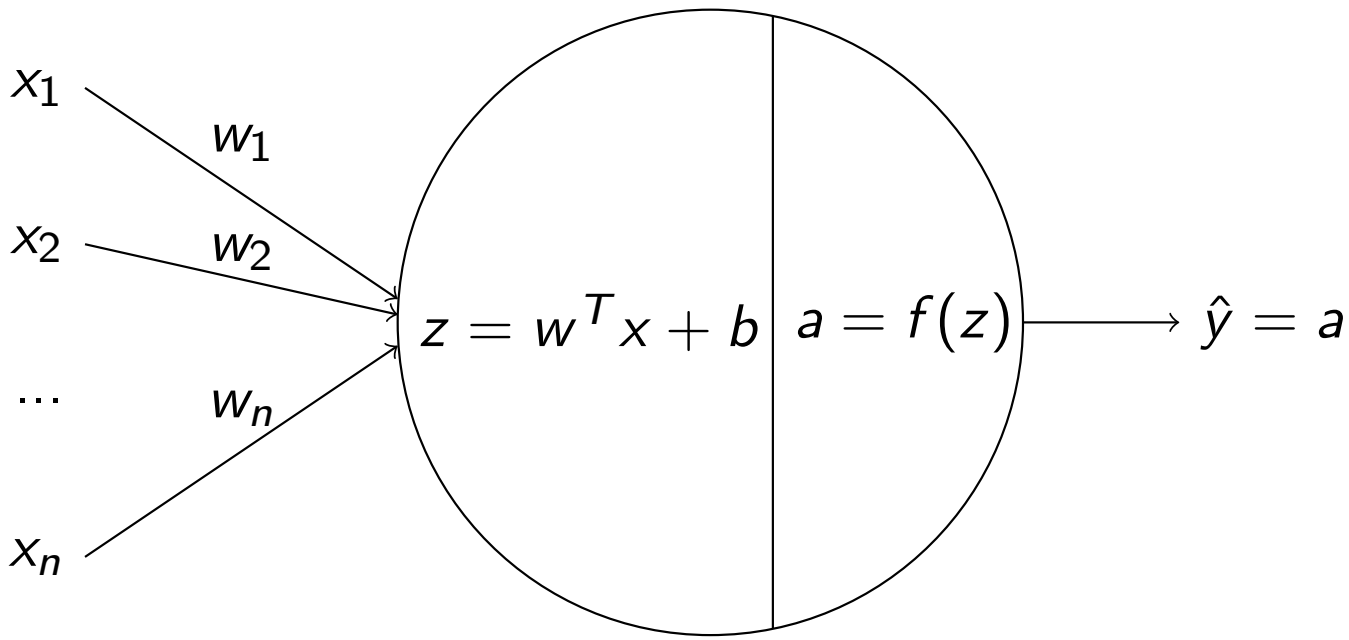
Perceptron monocouche : Fonctionnement



On assimile un neurone à un triplet : poids synaptique w, seuil S (ou biais b), fonction d'activation f.

1. produit scalaire entre les entrées x et les poids synaptiques w : $w^T x$;
2. ajout d'une valeur de référence (biais b) : $z = w^T x + b$
3. application de la fonction d'activation à la valeur obtenue z : $a = f(z)$

Perceptron monocouche : Fonctionnement



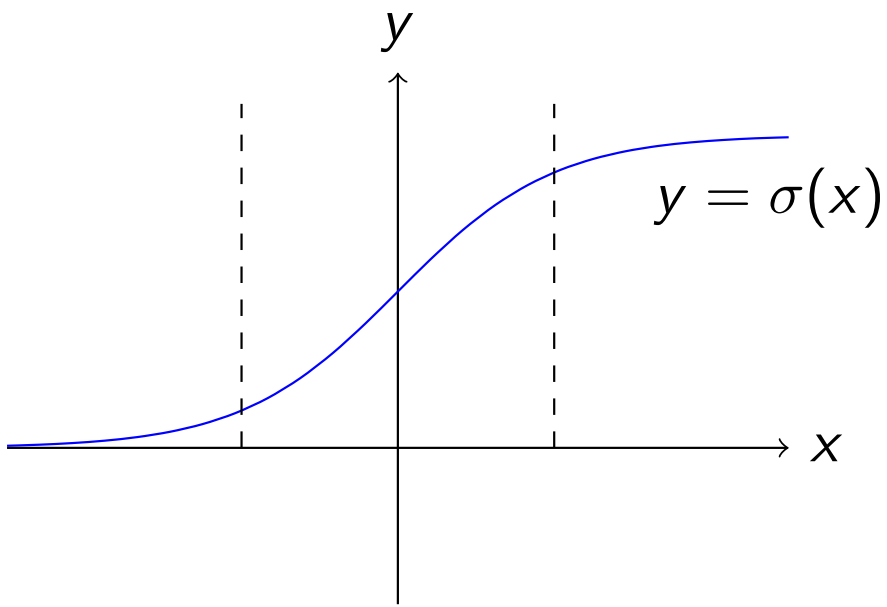
1. produit scalaire entre les entrées x et les poids synaptiques w : $w^T x$;
2. ajout d'une valeur de référence (biais b) : $z = w^T x + b$
3. application de la fonction d'activation à la valeur obtenue z : $a = f(z)$

Représentation "neurone"

Perceptron monocouche

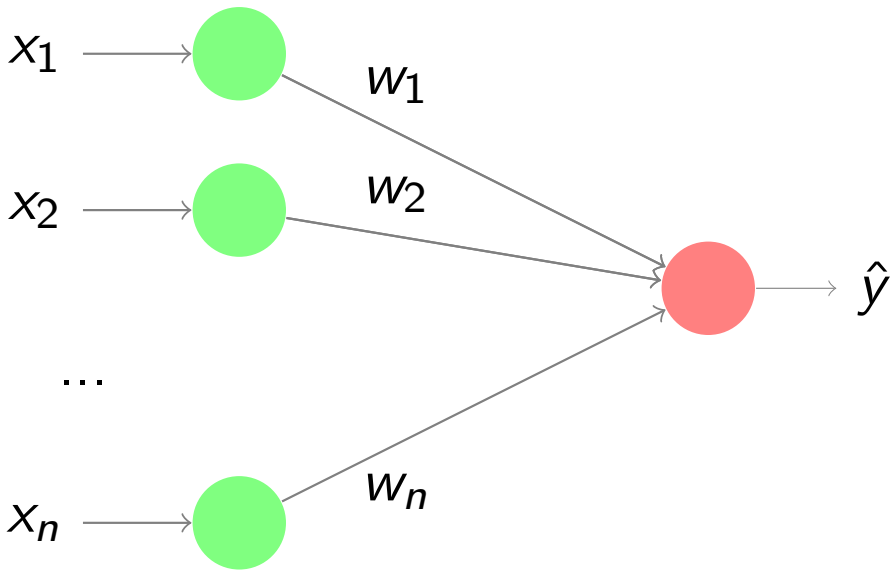
Les **fonctions d'activation**, notées f, sont des fonctions de seuillage qui peuvent souvent se décomposer en trois parties, comme c'est le cas pour la fonction sigmoïde (ci-dessous) :

- une partie non-activée, en dessous du seuil ;
- une phase de transition, aux alentours du seuil ;
- une partie activée, au dessus du seuil.



Rappel des notations

Entrées $x^{(1)}, ..., x^{(m)}$



Perceptron monocouche

Algorithm 1 Algorithme du perceptron monocouche

1. Initialisation des poids $w_j^{\{0\}}$
2. Présentation d'un vecteur d'entrées $x^{(1)}, ..., x^{(m)}$ et du vecteur de sortie correspondantes $y^{(1)}, .., y^{(m)}$
3. Calcul de la sortie prédite et de la fonction objectif :

$$\hat{y}^{(i)\{k\}} = f\left(\sum_{j=1}^{n} w_j^{\{k\}} x_j^{(i)}\right) \text{ et } J = \sum_{i=1}^{m} \text{perte}(\hat{y}^{(i)\{k\}}, y^{(i)})$$

4. Mise à jour des poids

$$w_j^{\{k+1\}} = w_j^{\{k\}} - \alpha \frac{\partial J}{\partial w_j}$$

où α est le taux d'apprentissage ($0 \leq \alpha \leq 1$).

5. Retourner en 2 jusqu'à la converence (c'est-à-dire $\hat{y}^{(i)\{k\}} \approx y^{(i)}$).

Réseau de neurones : perceptron monocouche

- Fonction d'activation linéaire : $f(x) = x$
- Fonction de coût quadratique :

$$J = \sum_{i=1}^{m} (\hat{y}^{(i)\{k\}} - y^{(i)})^2$$

- Descente de gradient :

$$w_j^{\{k+1\}} \leftarrow w_j^{\{k\}} - \alpha \frac{\partial J}{\partial w_j}$$

$$w_j^{\{k+1\}} \leftarrow w_j^{\{k\}} - \alpha \sum_{i=1}^{m} (\hat{y}^{(i)\{k\}} - y^{(i)}) x_j^{(i)}$$

$\Rightarrow$ Perceptron monocouche **équivalent à la régression linéaire** !

Réseau de neurones : perceptron monocouche

- Fonction d'activation sigmoïde : $\sigma(x) = \frac{1}{1+e^{-x}}$
- Fonction de coût entropie croisée :

$$J = -y^{(i)\{k\}} log(\hat{y}^{(i)}) - (1 - y^{(i)\{k\}}) log(1 - \hat{y}^{(i)})$$

- Descente de gradient :

$$w_j^{\{k+1\}} \leftarrow w_j^{\{k\}} - \alpha \frac{\partial J}{\partial w_j}$$

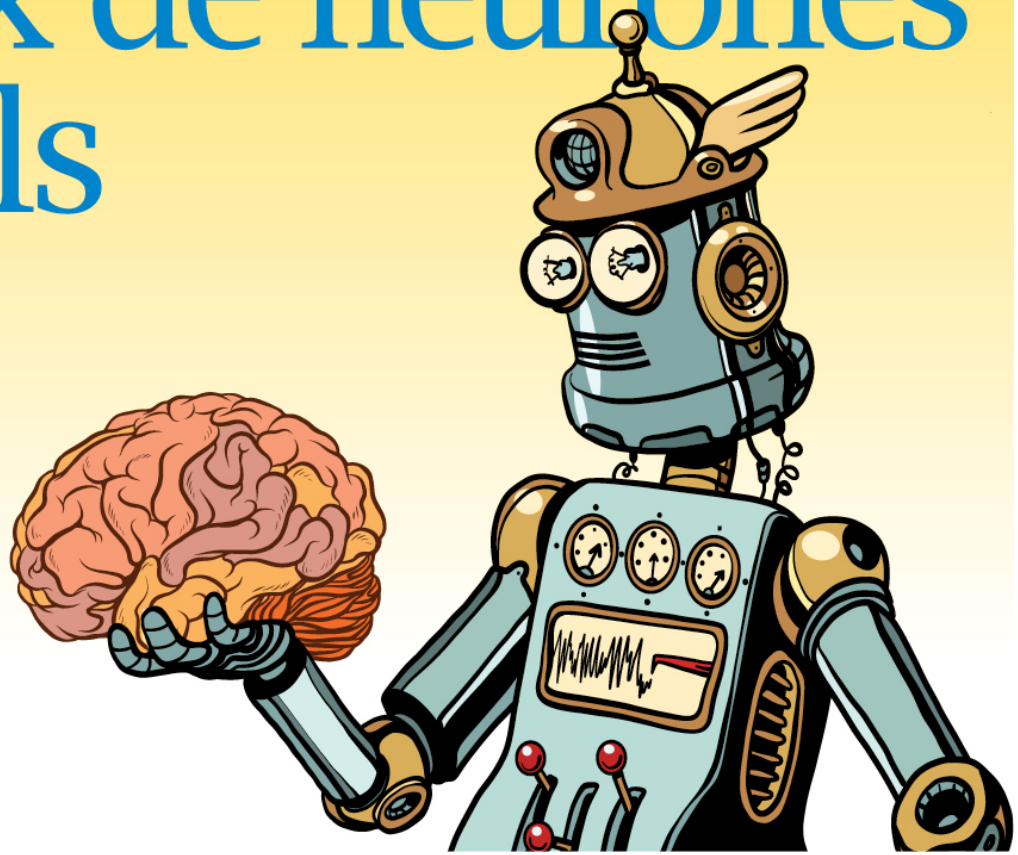
$$w_j^{\{k+1\}} = w_j^{\{k\}} - \alpha \sum_{i=1}^{m} (\hat{y}^{(i)\{k\}} - y^{(i)}) x_j^{(i)}$$

où α est le taux d'apprentissage (fixe ou variable).

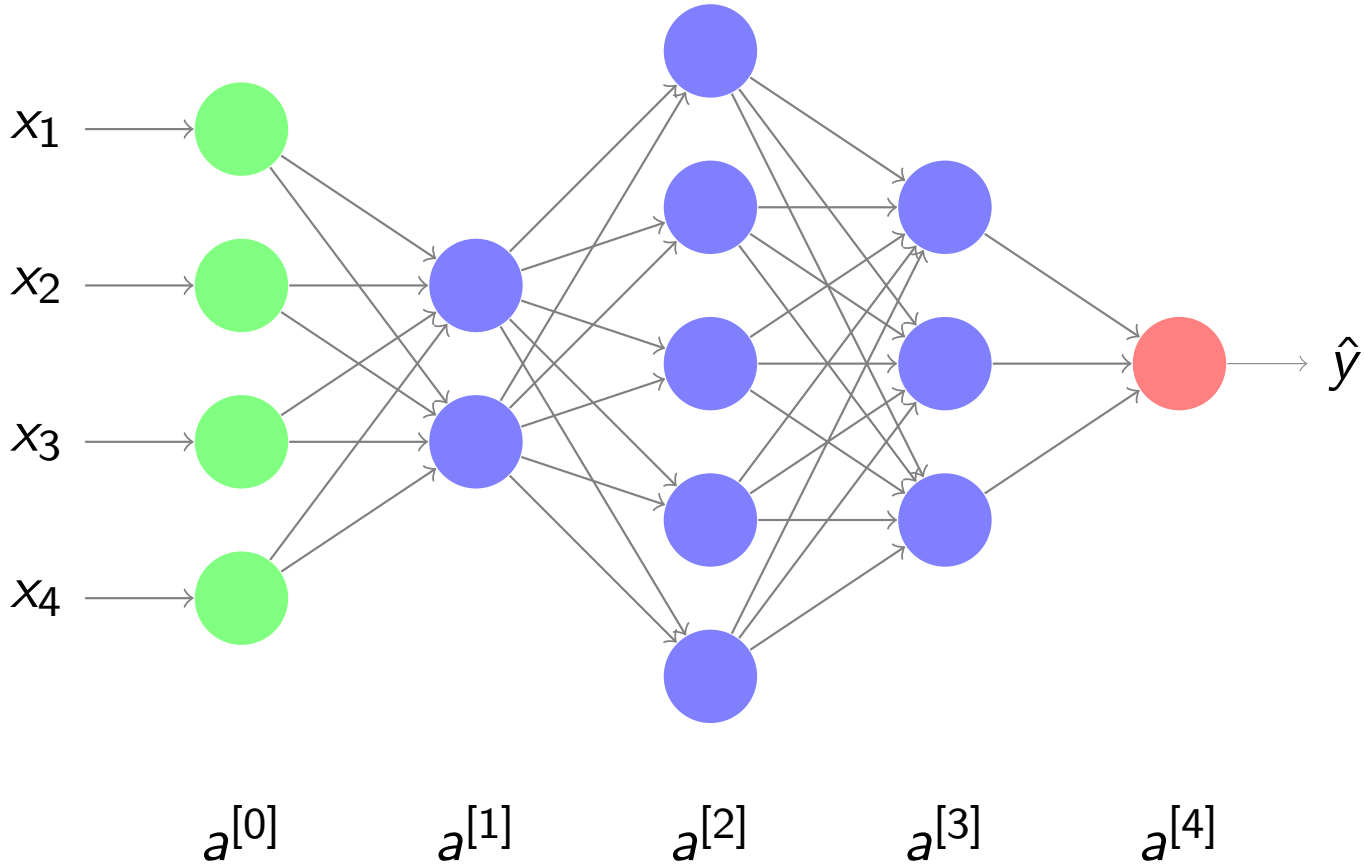
$\Rightarrow$ Perceptron monocouche **équivalent à la régression logistique** !

Plan du cours

Réseaux de neurones artificiels



Réseau de neurones : Perceptrons multicouche



Un perceptron multicouche se décompose en une couche d'entrée, une couche de sortie, et des couches cachées intermédiaires.
La **profondeur** du réseau est ici de 4 : 3 couches cachées plus une couche de sortie.

Point sur les notations

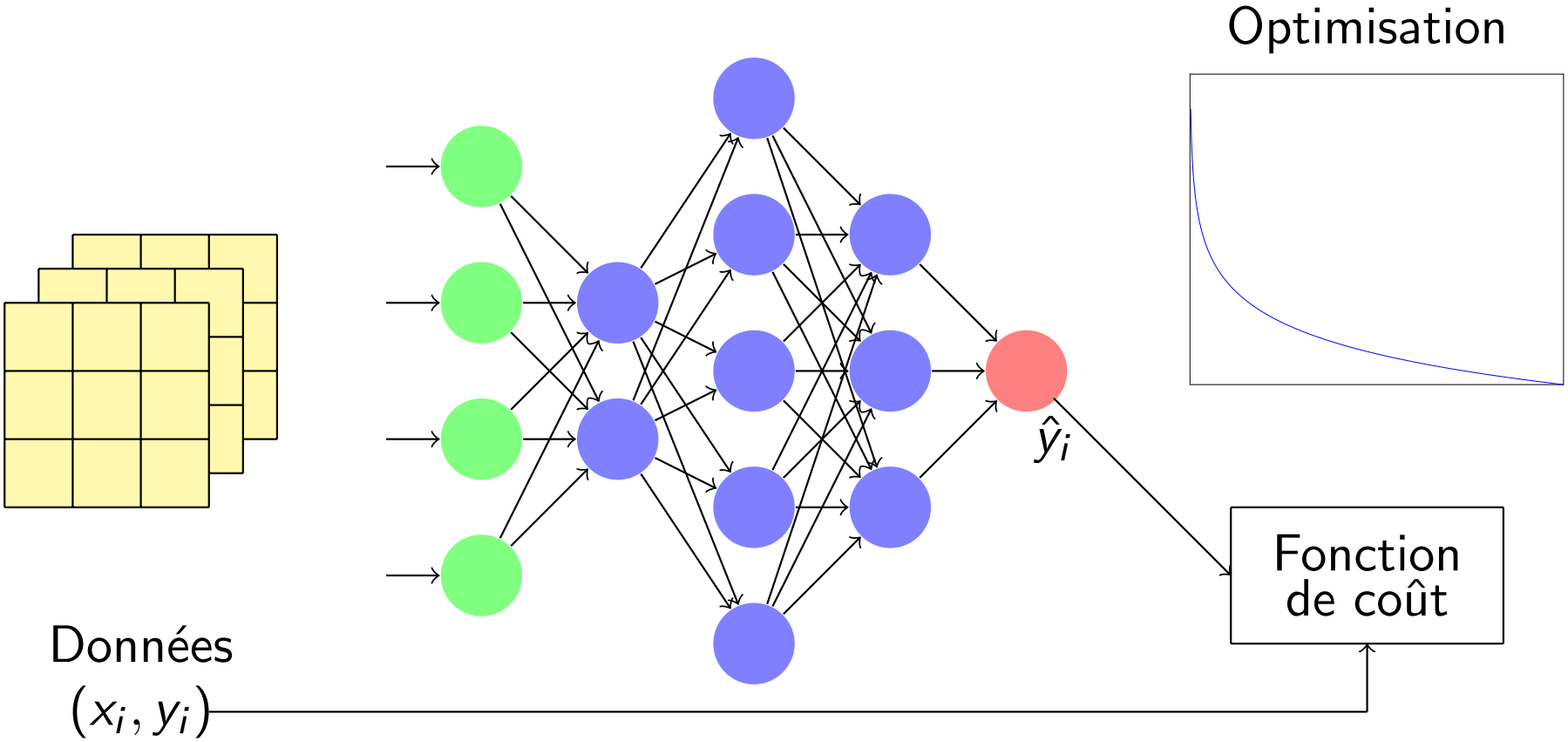
A ce stade, vous avez déjà remarqué la lourdeur des notations :

- les "()" désignent l'indice d'une donnée parmi l'ensemble des données $(x^{(i)})$
- les "$\{\}$" désignent l'itération courante de la descente de gradient $(w^{\{k\}})$
- les "[]" désignent l'indice de la couche $(a^{[c]})$

Ainsi, par exemple, $a_j^{(i)[c]\{k\}}$ désigne l'activation du j-ième neurone de la couche c, calculée depuis la i-ème donnée, lors de la k-ème itération de la descente de gradient.

Dans la suite on essaiera de simplifier les notations à chaque fois que cela sera possible...

Vue d'ensemble

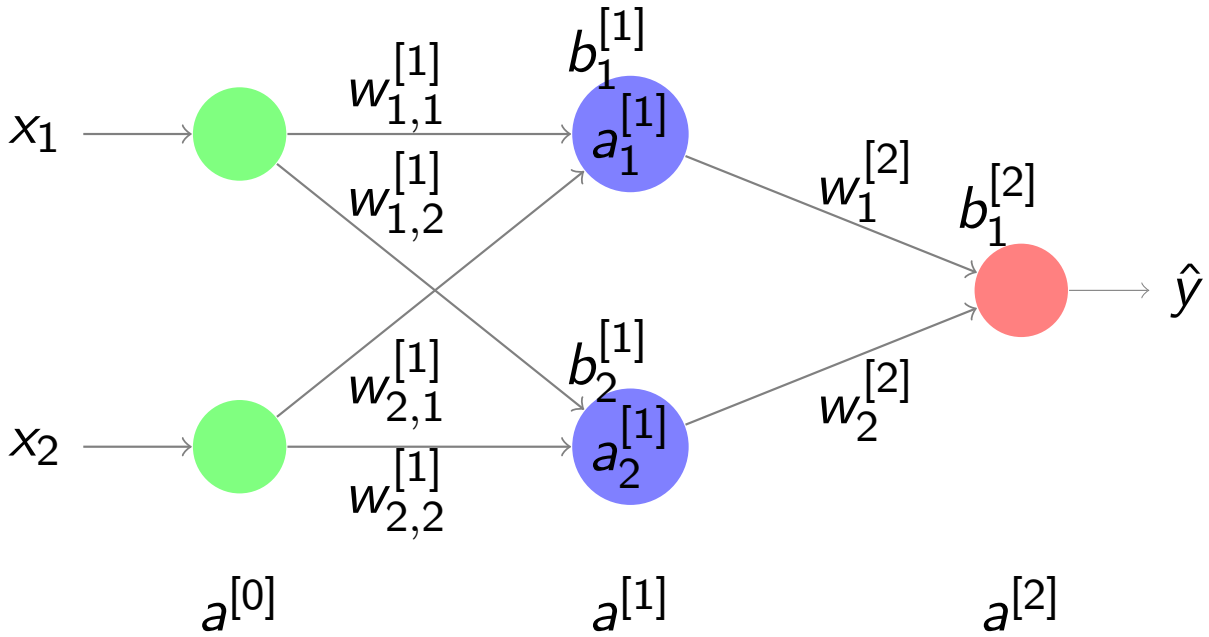


Réseau de neurones : perceptrons multicouche

Après la première phase d'initialisation, l'algorithme d'apprentissage (basé sur la descente de gradient) comporte 5 étapes qui se répètent jusqu'à convergence :

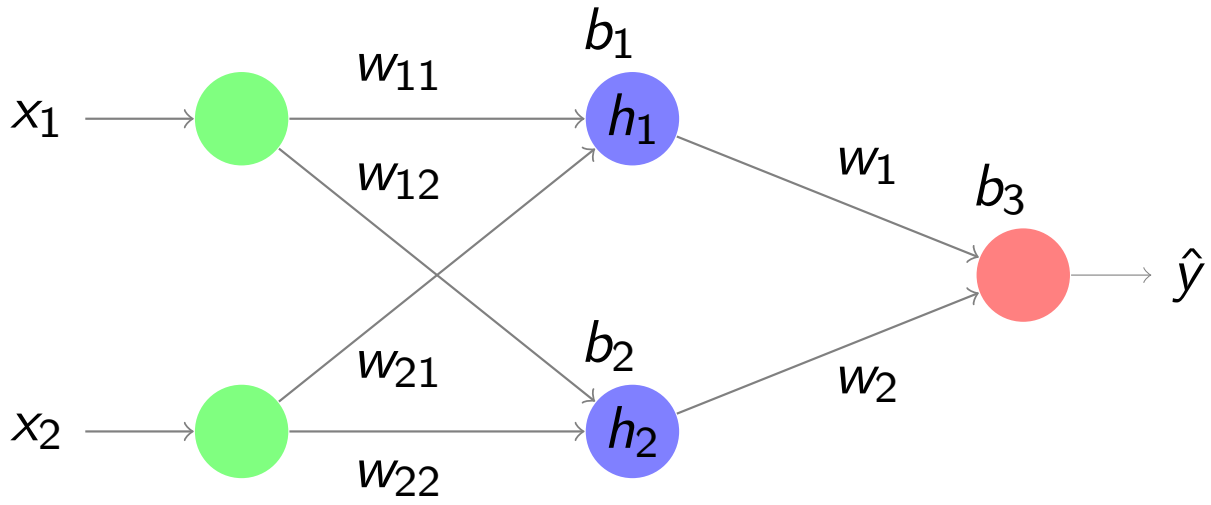
1. **Propagation des données** de la couche d'entrée à la couche de sortie ;
2. Calcul de l'**erreur de sortie** après la propagation des données ;
3. Calcul des **gradients d'erreurs** pour corriger les poids synaptiques des neurones de la **couche de sortie** ;
4. Calcul des gradients d'erreurs pour corriger les poids synaptiques des neurones **des couches cachées** ;
5. **Mise à jour** des poids synaptiques de la couche de sortie et de la couche cachée.

Illustration de l'entraînement d'un perceptron multicouche



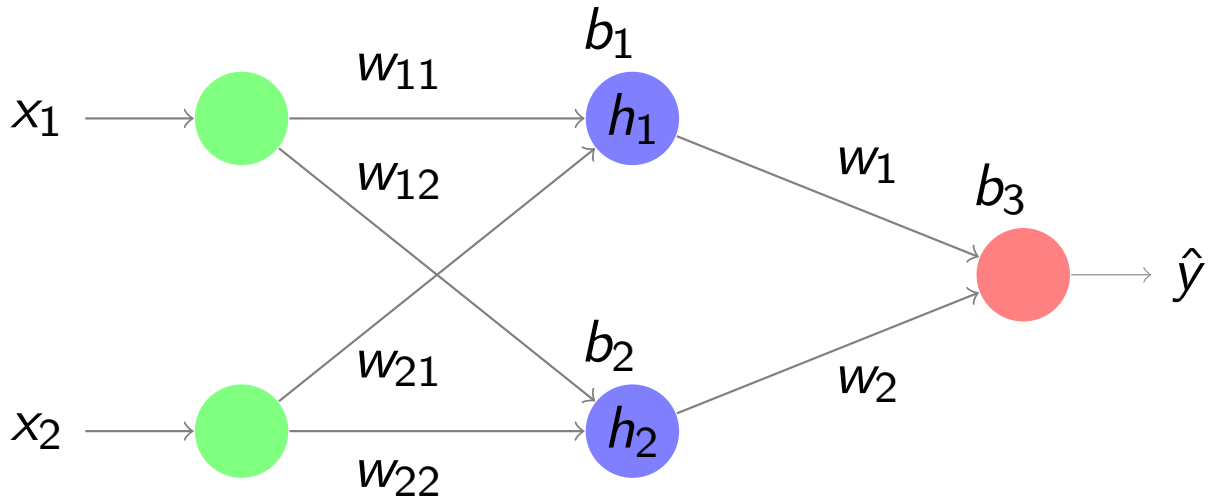
Simplifions un peu les notations (pour enlever l'indice de couche).

Illustration de l'entraînement d'un perceptron multicouche



$$\begin{cases} \hat{y} = \sigma(z) \text{ où } z = w_1 h_1 + w_2 h_2 + b_3 \\ h_1 = f(z_1) \text{ où } z_1 = w_{11} x_1 + w_{21} x_2 + b_1 \\ h_2 = f(z_2) \text{ où } z_2 = w_{12} x_1 + w_{22} x_2 + b_2 \end{cases}$$

Perceptrons multicouche : un exemple



$$\begin{cases} \hat{y} = \sigma(z) \text{ où } z = w_1 h_1 + w_2 h_2 + b_3 \\ h_1 = f(z_1) \text{ où } z_1 = w_{11} x_1 + w_{21} x_2 + b_1 \\ h_2 = f(z_2) \text{ où } z_2 = w_{12} x_1 + w_{22} x_2 + b_2 \end{cases}$$

1) **Propagation des données** de la couche d'entrée à la couche de sortie :

$$y = \sigma\left(w_1 f(w_{11} x_1 + w_{12} x_2 + b_1) + w_2 f(w_{12} x_1 + w_{22} x_2 + b_2) - b_3\right)$$

Réseau de neurones : perceptrons multicouche

2) Calcul de l'**erreur de sortie** (fonction objectif) après la propagation des données :

$$J = \frac{1}{m}\sum_{i=1}^{m} -y^{(i)} log(\hat{y}^{(i)}) - (1 - y^{(i)}) log(1 - \hat{y}^{(i)})$$

3) Calcul des **gradients d'erreurs** pour corriger les poids synaptiques des neurones de la **couche de sortie** ;
En utilisant le principe du *chaînage des dérivées partielles* ($\frac{\partial f(y)}{\partial x} = \frac{\partial f(y)}{\partial y} . \frac{\partial y}{\partial x}$), on obtient :

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_j},$$

$$\frac{\partial J}{\partial b_3} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial b_3},$$

pour $j = \{1, 2\}$

Réseau de neurones : perceptrons multicouche

4) Calcul des gradients d'erreurs pour corriger les poids synaptiques des neurones **des couches cachées** ;

$$\frac{\partial J}{\partial w_{jj'}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial h_{j'}} \frac{\partial h_{j'}}{\partial z_{j'}} \frac{\partial z_{j'}}{\partial w_{jj'}},$$

$$\frac{\partial J}{\partial b_j} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial h_j} \frac{\partial h_j}{\partial z_j} \frac{\partial z_j}{\partial b_j},$$

pour $j, j' = \{1, 2\}$.

5) **Mise à jour** des poids synaptiques de la couche de sortie :

$$w_j \leftarrow w_j - \alpha \frac{\partial J}{\partial w_j}$$

et de la couche cachée :

$$w_{jj'} \leftarrow w_{jj'} - \alpha \frac{\partial J}{\partial w_{jj'}}.$$

Remarque : rétropropagation du gradient

Pour calculer les gradients des poids synaptiques présentés plus tôt, pour rappel :

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_j},$$

et

$$\frac{\partial J}{\partial w_{jj'}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial h_{j'}} \frac{\partial h_{j'}}{\partial z_{j'}} \frac{\partial z_{j'}}{\partial w_{jj'}},$$

il est intéressant de calculer d'abord $\frac{\partial J}{\partial w_j}$, puis de réutiliser les calculs intermédiaires pour ensuite calculer $\frac{\partial J}{\partial w_{jj'}}$.
C'est l'algorithme, efficace, de la **rétropropagation du gradient** qui procède ainsi des dernières couches jusqu'aux premières couches pour le calcul des gradients.

Perceptrons multi-couches : interprétation

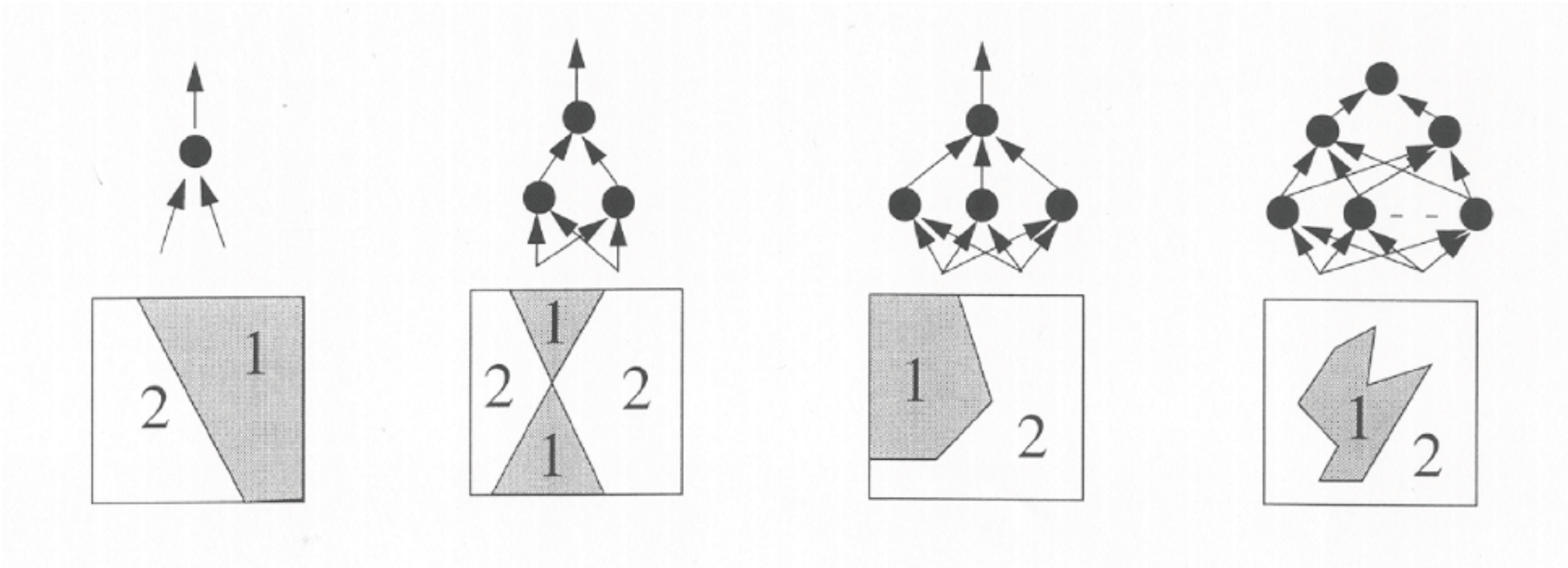
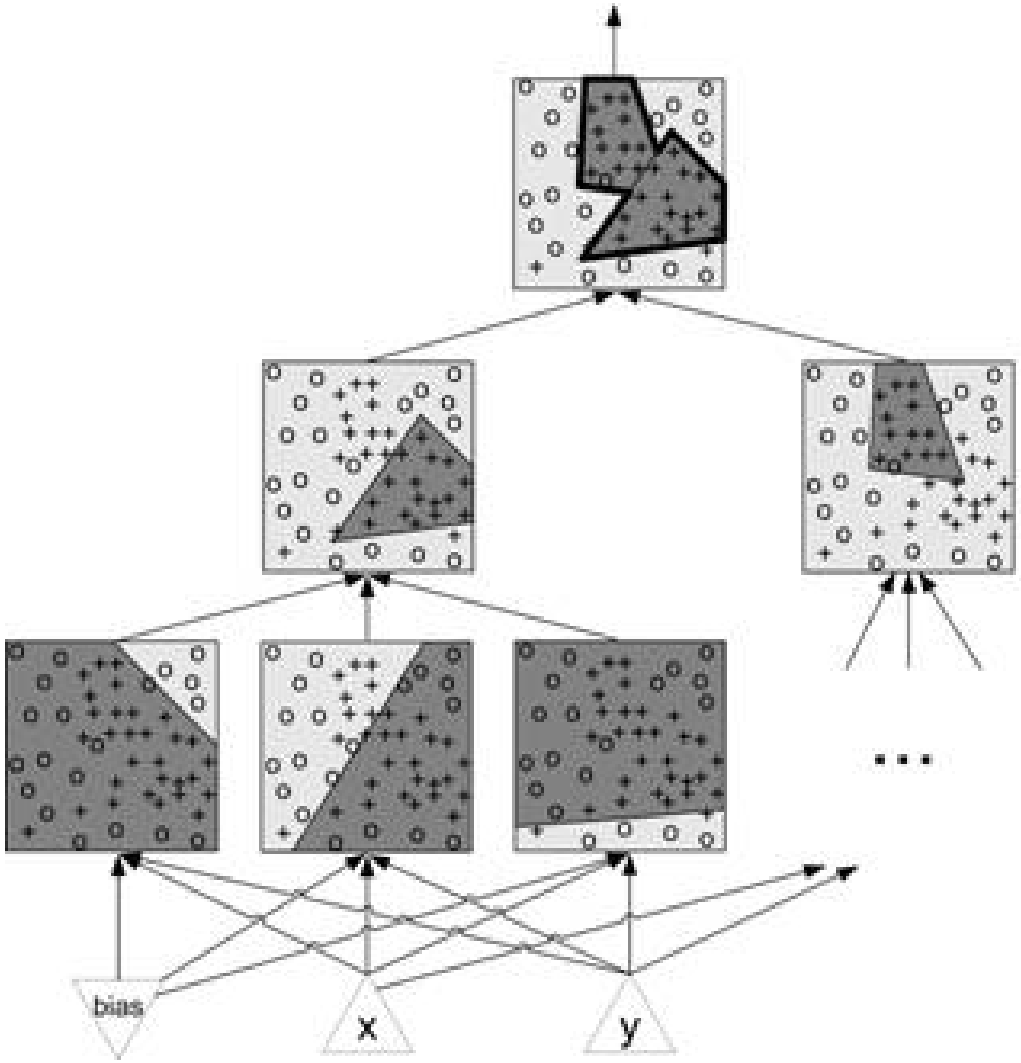


FIGURE – Intérêt du perceptron multicouche : pouvoir séparateur

L'augmentation du nombre de couches et du nombre de neurones accroît le pouvoir de séparation

Perceptrons multi-couches : interprétation



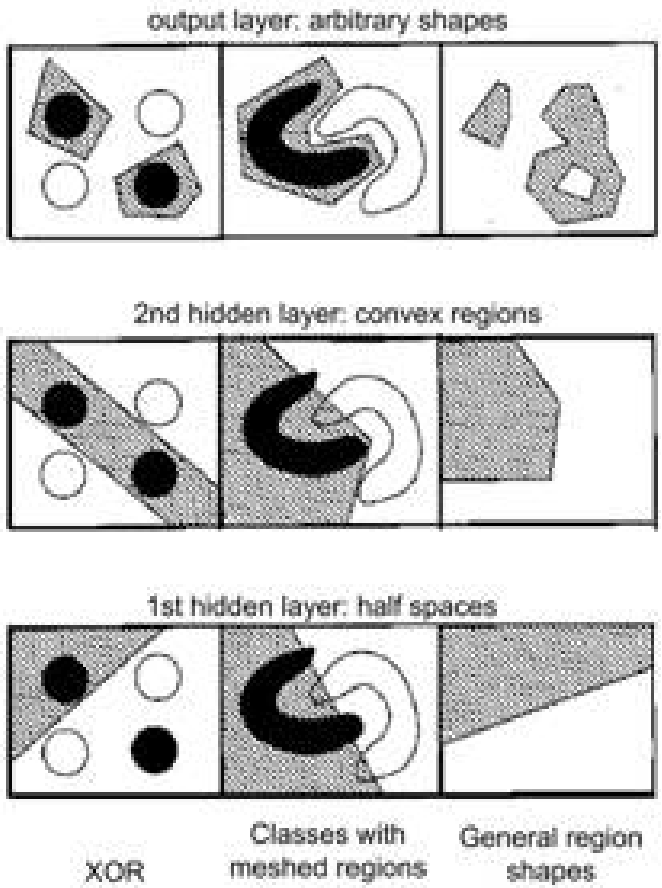


FIGURE – Intérêt du perceptron muticouche

Théorème d'Approximation Universelle

Théorème d'approximation universelle (Cybenko 1989)

Toute fonction f, continue, de $[0,1]^m$ dans $\mathbb{R}$, peut être approximée par un perceptron multi-couche à une couche cachée comportant suffisamment de neurones (avec une fonction d'activation sigmoide).

Note : le théorème a été également prouvé avec la fonction reLU.

Le théorème **ne dit pas comment** déterminer ce réseau de neurones !

Plan du cours

Visualisation

https://playground.tensorflow.org/