

Neural Network Basics

High-Dimensional Statistics and Deep Learning

Juliette Chevallier

2023 – 2024

INSA Toulouse, juliette.chevallier@insa-toulouse.fr

What is machine learning ?

Machine learning is a “Field of study that gives computers the ability to learn without being explicitly programmed.”

→ How to construct computer programs that automatically improve with the experience ?

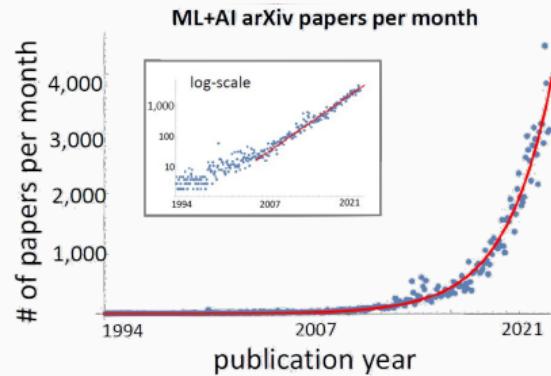
Arthur Samuel (1959)

“A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.”

Tom Mitchell (1998)

Course Outline

- Une brève histoire de l'Intelligence Artificielle
- Apprentissage statistique
- Régression linéaire et régression logistique
- Perceptron monocouche
- Réseaux de neurones (perceptron multicouche)
- Fonctions d'activation
- Optimiseurs
- Sous-apprentissage et sur-apprentissage
- Régularisation
- Méthodologie en apprentissage profond



Course Outline

Une brève histoire de l'Intelligence Artificielle

Apprentissage statistique

Régression linéaire et régression logistique

Perceptron monocouche

Réseaux de neurones (perceptron multicouche)

Fonctions d'activation

Optimiseurs

Sous-apprentissage et sur-apprentissage

Régularisation

Méthodologie en apprentissage profond

Une brève histoire de l'Intelligence Artificielle 1/3

1. Depuis l'antiquité : Attrait pour les hommes artificiels/machines pensantes
Pygmalion et Galatée (~ 210 av. J.C.), *Golem*, *Frankenstein*, etc.

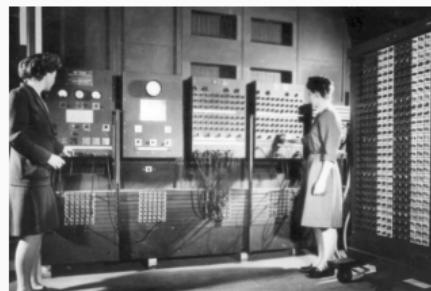
Automates anthropomorphes (XVIII^e siècle) : *Canard de Vaucanson*, *Jaquet-Droz*

~~> **Le processus de pensée humaine peut être mécanisé.**

“La raison [...] n'est rien d'autre que le fait de calculer” *Hobbes*. *Descartes*, *Leibniz*.



(a) Automates de Jaquet-Droz



(b) ENIAC – Jean Bartik et Frances Spence

2. XXe siècle : **Naissance de logique mathématique**. *Boole*, *Russell*, *Gödel*, *Turing*.

“Le raisonnement mathématique peut-il être entièrement formalisé ?” *Hilbert*.

Une brève histoire de l'Intelligence Artificielle 2/3

3. 1930-1950 : **Naissance de la cybernétique.** Le cerveau est un *réseau électrique* de **neurones** qui envoient des *impulsions* de type tout-ou-rien. *Wiener*
 - ~~> 1943 : Premier modèle de neurone formel de *McCulloch* et *Pitts. Shannon*.
 - ~~> 1958 : Perceptrons. *Rosenblatt*.
4. 1956 : **Conférence de Dartmouth.** Naissance de l'intelligence artificielle.
Organisée par *Minsky, McCarthy, Shannon, Rochester*.
 - Première mention de l'appellation “**Machine Learning**” (Intelligence Artificielle),
 - Définitions des objectifs, Concrétisation, Scission d'avec l'informatique, etc.
5. 1950-75 : **Premier âge d'or.**
 - Premier chatbot (ELIZA, 1966),
 - *Programme de dames d'Arthur Samuel (1952)* : Premier programme d'**auto-apprentissage**, jouant aux dames et s'améliorant en jouant.

Une brève histoire de l'Intelligence Artificielle 3/3

6. 1975-80 : **Hibernation.**

- *Limites de la puissance de calcul* : Programme NLP sur un dictionnaire de... 20 mots,
- *Pas de base de données massive*,
- On délaisse le connexionnisme (perceptrons) au profit du calcul des prédicts (langage Prolog, *Colmerauer, Roussel*)

7. À partir de 1980 : **Le boom.**

- *Naissance des systèmes experts*,
Reproduire les mécanismes cognitifs d'un expert, ds un domaine de connaissance particulier.
- *Ingénierie de la connaissance*.
Intelligence basée sur la capacité à utiliser une **large quantité de savoirs**.
- *Renaissance du connexionnisme*.
- *Loi de Moore*.
Vitesse et la capacité de mémoire des ordinateurs doublent tous les deux ans.

Une brève histoire de l'IA : Exemples marquants

- 1997 : **Deep Blue**. Premier système informatique de jeu d'échecs à battre le champion du monde en titre, *Garry Kasparov*,

- 1999 : **AiBo**. Chien robot de compagnie.

- 2002 : **Roomba**. Premier robot aspirateur.
- 2005 : Premières **voitures autonomes** à réussir le DARPA challenge.
- 2011 : “Naissance” de **Siri**, **Watson** gagne un tournois de *Jeopardy*.

- 2014 : **Eugene Goostman**, agent conversationnel, passe le **test de Turing**,
DeepFace : Détection automatique des visages sur Facebook.
- 2015 : **AlphaGo** gagne contre un des meilleurs joueurs au jeu de Go.
- 2019 : **AlphaStar** a atteint le statut de Grandmaster sur StarCraft II.
- 2022 : Agent conversationnel **ChatGPT**, etc.

Course Outline

Une brève histoire de l'Intelligence Artificielle

Apprentissage statistique

Régression linéaire et régression logistique

Perceptron monocouche

Réseaux de neurones (perceptron multicouche)

Fonctions d'activation

Optimiseurs

Sous-apprentissage et sur-apprentissage

Régularisation

Méthodologie en apprentissage profond

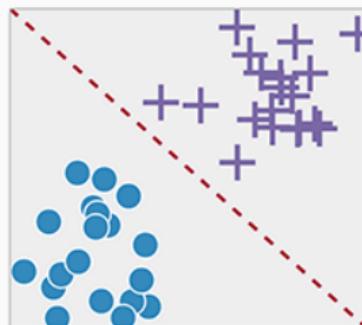
Trois grands paradigmes d'apprentissage : Apprentissage supervisé

Apprentissage supervisé :

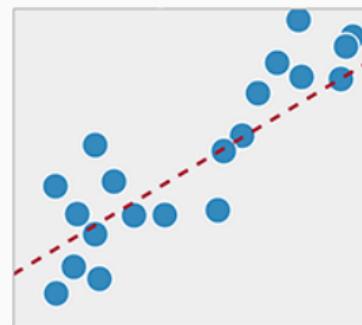
- Un **oracle** (expert) fournit un ensemble d'apprentissage (données, **labels**) :

$$\mathcal{D} = \{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\};$$

- Recherche d'un prédicteur *minimisant la différence entre **labels** réels et prédis* ;



(a) Classification



(b) Régression

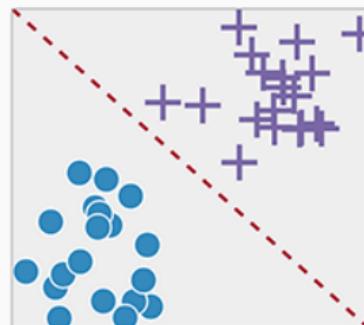
Trois grands paradigmes d'apprentissage : Apprentissage supervisé

Apprentissage supervisé :

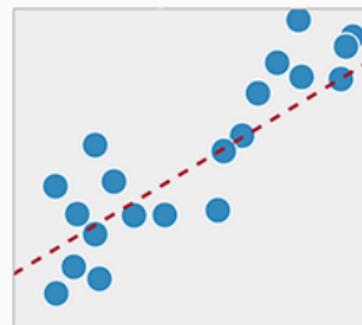
- Un **oracle** (expert) fournit un ensemble d'apprentissage (données, **labels**) :

$$\mathcal{D} = \{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\};$$

- Recherche d'un prédicteur *minimisant la différence entre **labels** réels et prédicts* ;
- **Pbm** : Souvent *coûteux* car nécessite l'**annotation** de larges bases de données.



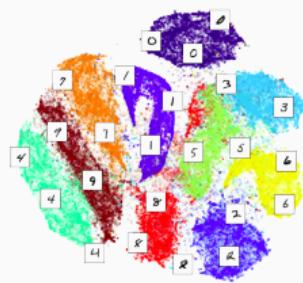
(a) Classification



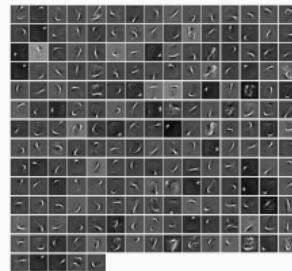
(b) Régression

Trois grands paradigmes d'apprentissage : Apprentissage non-supervisé

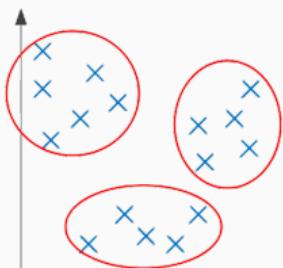
Dans le cadre de l'**apprentissage non supervisé**, on cherche à inférer de l'information à partir d'**observations uniquement** : $\mathcal{D} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$.



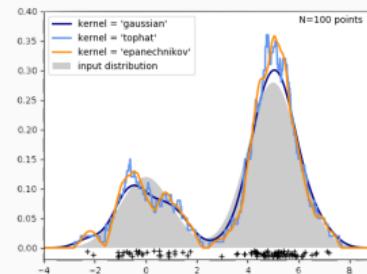
(a) Réduction de dimension



(b) Extraction de caractéristiques



(c) Clustering

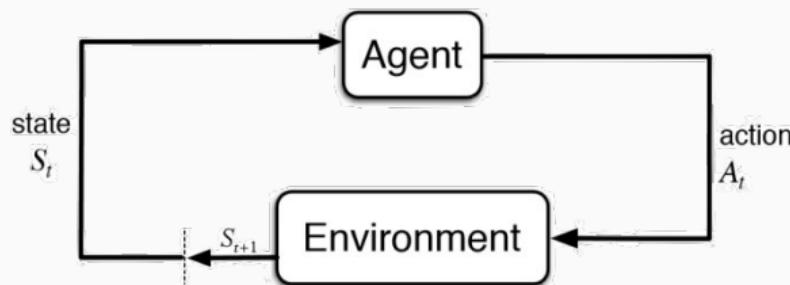


(d) Éstimation de densité

Trois grands paradigmes d'apprentissage : Apprentissage par renforcement

Apprentissage par renforcement :

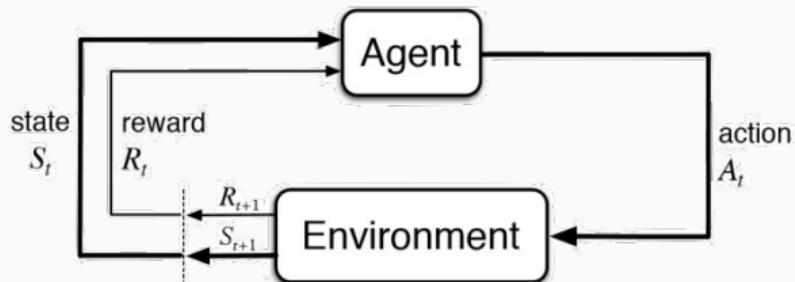
- Définition d'un **agent** comportemental, qui peut prendre un ensemble de décisions (**actions**) en fonction de l'**état** d'un certain système ;



Trois grands paradigmes d'apprentissage : Apprentissage par renforcement

Apprentissage par renforcement :

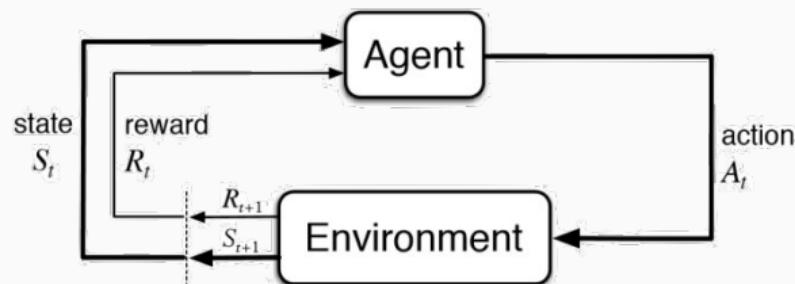
- Définition d'un **agent** comportemental, qui peut prendre un ensemble de décisions (**actions**) en fonction de l'**état** d'un certain système ;
- L'agent obtient des **récompenses** pour chacune de ses actions ;



Trois grands paradigmes d'apprentissage : Apprentissage par renforcement

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 être due à :
 - Un trop **petit nombre** d'annotations (*few-shots/one-shot learning*), ou
 - Les annotations sont trop **bruitées**, ou **imprécises**.

Mais aussi...

- **Apprentissage faiblement supervisé** : La faible supervision peut être due à :
 - Un trop petit nombre d'annotations (*few-shots/one-shot learning*), ou
 - Les annotations sont trop bruitées, ou imprécises.
- **Apprentissage semi-supervisé** :
 - On dispose à la fois d'un **faible** nombre de **données annotées** & d'un **grand** nombre de **données non-annotées**.

Mais aussi...

- **Apprentissage faiblement supervisé :** La faible supervision peut être due à :
 - Un trop petit nombre d'annotations (*few-shots/one-shot learning*), ou
 - Les annotations sont trop bruitées, ou imprécises.
- **Apprentissage semi-supervisé :**
 - On dispose à la fois d'un faible nombre de données annotées & 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.

Course Outline

Une brève histoire de l'Intelligence Artificielle

Apprentissage statistique

Régression linéaire et régression logistique

Perceptron monocouche

Réseaux de neurones (perceptron multicouche)

Fonctions d'activation

Optimiseurs

Sous-apprentissage et sur-apprentissage

Régularisation

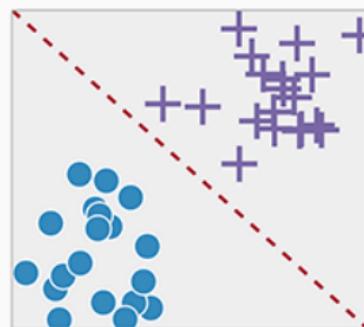
Méthodologie en apprentissage profond

Apprentissage supervisé

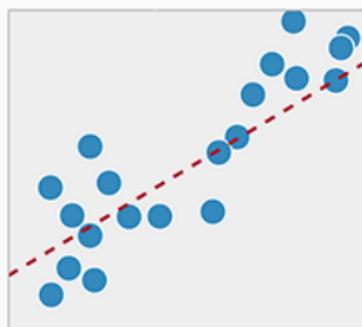
Dans le cadre de l'**apprentissage supervisé**, on dispose d'*observations* et de leurs étiquettes (appelées encore cibles (**targets**), catégories ou **labels**) qui constituent un ensemble d'apprentissage. On le note :

$$\mathcal{D} = \left\{ (\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), \dots, (\mathbf{x}^{(m)}, \mathbf{y}^{(m)}) \right\}, \quad \text{où} \quad \begin{cases} \mathbf{x}^{(i)} = (\mathbf{x}_0^{(i)}, \dots, \mathbf{x}_d^{(i)}) \in \mathcal{X} \subset \mathbb{R}^{d+1}, \\ y^{(i)} \in \mathcal{Y} \subset \mathbb{R}. \end{cases}$$

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



(a) Classification



(b) Régression

Apprentissage supervisé

Idée générale :

1. On observe m **réalisations** d'un couple de variables aléatoires (X, Y) définis sur $\mathcal{X} \times \mathcal{Y}$ de loi \mathbb{P} *inconnue* ;
2. On définit une **règle de prédiction** $f: \mathcal{X} \rightarrow \mathcal{Y}$, mesurable, permettant de “prédire” le **label** $\hat{y} = f(x) \in \mathcal{Y}$ associé à chacune des observations $x \in \mathcal{X}$;
3. On définit une **fonction de perte** $\ell: \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}^+$ qui mesure la **distance** entre les vrais labels et leur prédiction : $\forall y, y' \in \mathcal{Y}$ telle que $y \neq y'$,

$$\ell(y, y) = 0 \quad \text{et} \quad \ell(y, y') > 0.$$

Apprentissage supervisé

Deux principaux types d'apprentissage *supervisé* :

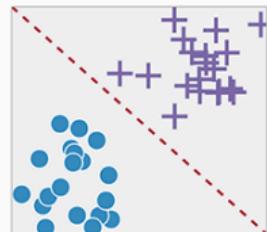
- **Classification**

- **Régression :**

Apprentissage supervisé

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.
 - x : vecteur ou matrice des intensités des pixels de l'image,
 - y : identité du chiffre.
- **Régression** :

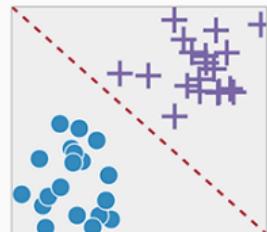


Apprentissage supervisé

Deux principaux types d'apprentissage *supervisé* :

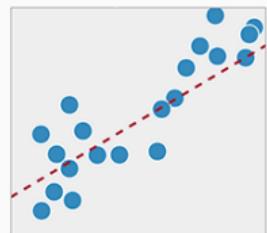
- **Classification** : Assigner une *catégorie* à chaque observation :

- Les catégories sont *discrètes*,
- La cible est un indice de *classe* : $y \in [0, K - 1]$.
- *Exemple* : reconnaissance de chiffres manuscrits.
 - 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.
 - x : vecteur contenant l'information sur l'activité économique,
 - y : valeur de l'action le lendemain.

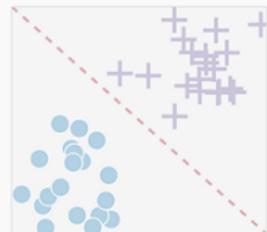


Apprentissage supervisé

Deux principaux types d'apprentissage *supervisé* :

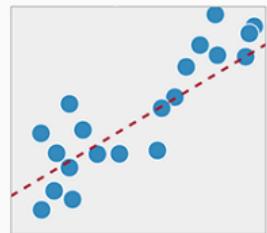
- **Classification** : Assigner une *catégorie* à chaque observation :

- Les catégories sont *discrètes*,
- La cible est un indice de *classe* : $y \in [0, K - 1]$.
- Exemple : reconnaissance de chiffres manuscrits.
 - 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.
 - x : vecteur contenant l'information sur l'activité économique,
 - y : valeur de l'action le lendemain.



Régression linéaire

Soit l'ensemble \mathcal{D} contenant m **réalisations** : $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$.

- **Prédicteur** : Pour le $i^{\text{ème}}$ exemple $x^{(i)}$, le modèle **linéaire** s'écrit $\hat{y}^{(i)} = {}^t \theta x^{(i)}$.
- **Fonction de perte** : on utilise la fonction de coût quadratique $(\hat{y}^{(i)} - y^{(i)})^2$.

Régression linéaire

Soit l'ensemble \mathcal{D} contenant m **réalisations** : $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$.

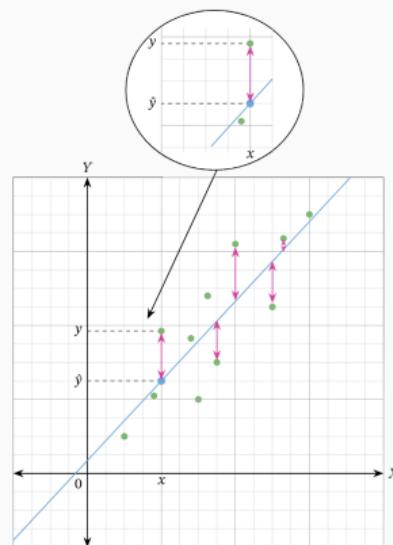
- **Prédicteur** : Pour le $i^{\text{ème}}$ exemple $x^{(i)}$, le modèle **linéaire** s'écrit $\hat{y}^{(i)} = {}^t\theta x^{(i)}$.
- **Fonction de perte** : on utilise la fonction de coût quadratique $(\hat{y}^{(i)} - y^{(i)})^2$.

~~ Formulation par **Moindres Carrés** !

Trouver θ qui minimise la perte sur tous les exemples d'apprentissage.

~~ Fonction objectif $J(\theta)$:

$$\theta^* \in \arg \min_{\theta} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2 := J(\theta)$$



Régression linéaire

Soit l'ensemble \mathcal{D} contenant m exemples d'apprentissage en dimension d :

$$\mathcal{D} = \left\{ (\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(m)}, y^{(m)}) \right\}, \quad \text{avec } \mathbf{x}^{(i)} = (\mathbf{x}_0^{(i)}, \dots, \mathbf{x}_d^{(i)}) \in \mathbb{R}^{d+1}.$$

On définit :

- $\mathbf{X} \in \mathbb{R}^{m \times (d+1)}$ matrice des données,
- $\mathbf{y} \in \mathbb{R}^m$: vecteur de cibles,
- $\hat{\mathbf{y}} \in \mathbb{R}^m$: vecteur de prédictions avec $\hat{\mathbf{y}} = \mathbf{X}\theta$,
- $\theta \in \mathbb{R}^{d+1}$ vecteur des paramètres du modèle à estimer.

Régression linéaire par moindres carrés

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

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

Régression linéaire – En pratique

Résolution des moindres carrés : Minimiser $J(\theta) := \min_{\theta} \|X\theta - y\|^2$.

Régression linéaire – En pratique

Résolution des moindres carrés : Minimiser $J(\theta) := \min_{\theta} \|X\theta - y\|^2$.

- Condition Nécessaire du premier ordre : $\frac{dJ(\theta)}{d\theta} = 2^t X(X\theta - y) = 0$;
- Si $\text{Det}(^t X X) \neq 0$, la solution analytique est $\hat{\theta} = (^t X X)^{-1} {}^t X y$.

Régression linéaire – En pratique

Résolution des moindres carrés : Minimiser $J(\theta) := \min_{\theta} \|X\theta - y\|^2$.

- Condition Nécessaire du premier ordre : $\frac{dJ(\theta)}{d\theta} = 2^t X(X\theta - y) = 0$;
- Si $\text{Det}({}^t X X) \neq 0$, la solution analytique est $\hat{\theta} = ({}^t X X)^{-1} {}^t X y$.

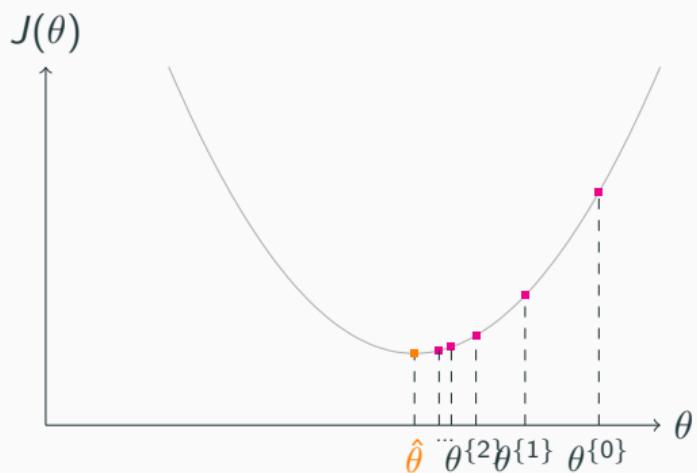
Problème : Calculs coûteux (inversion de matrice)

~~> Solution itérative : **Descente de gradient**

Remarque : Les problèmes aux moindres carrés sont **convexes**.

~~> Minimum local est global !

Descente de gradient – Cas convexe



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

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

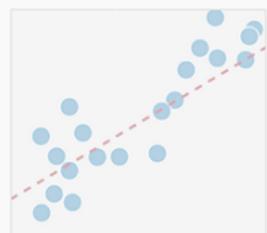
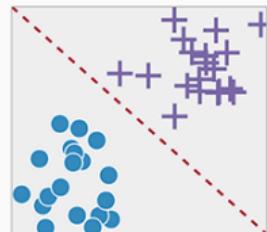
$k \leftarrow k + 1$

FIN TANT QUE

Apprentissage supervisé

Deux principaux types d'apprentissage *supervisé* :

- **Classification** Assigner une *catégorie* à chaque observation :
 - Les catégories sont *discrètes*,
 - La cible est un indice de *classe* : $y \in \{0, \dots, K - 1\}$.
 - Exemple : reconnaissance de chiffres manuscrits.
 - 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.
 - x : vecteur contenant l'information sur l'activité économique,
 - y : valeur de l'action le lendemain.



Régression logistique

Il s'agit d'un algorithme de **classification** malgré son nom.

Il est très populaire et très utilisé car il est simple et efficace en général.

Sortie du modèle peut être :

- **binaire** : échec/succès, 0/1, -/+
~~~ Fonction **sigmoïde ou logistique**
  
- **multinomiale** (multi-classes) : chiffres  
~~~ Fonction **softmax**

Régression logistique : Cas binaire

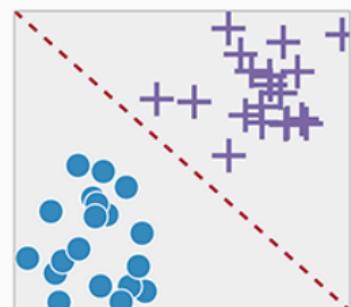
Comme avec la régression linéaire, on prend un **modèle linéaire** type :

$$z = \theta_0 + \theta_1 x_1 + \dots + \theta_d x_d = {}^t \theta \mathbf{x}$$

Ce modèle linéaire agit comme **séparateur** des 2 classes.

Puis on veut une probabilité : $0 \leq h_\theta(x) = \pi(z) \leq 1$ 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 ($y = 1$).



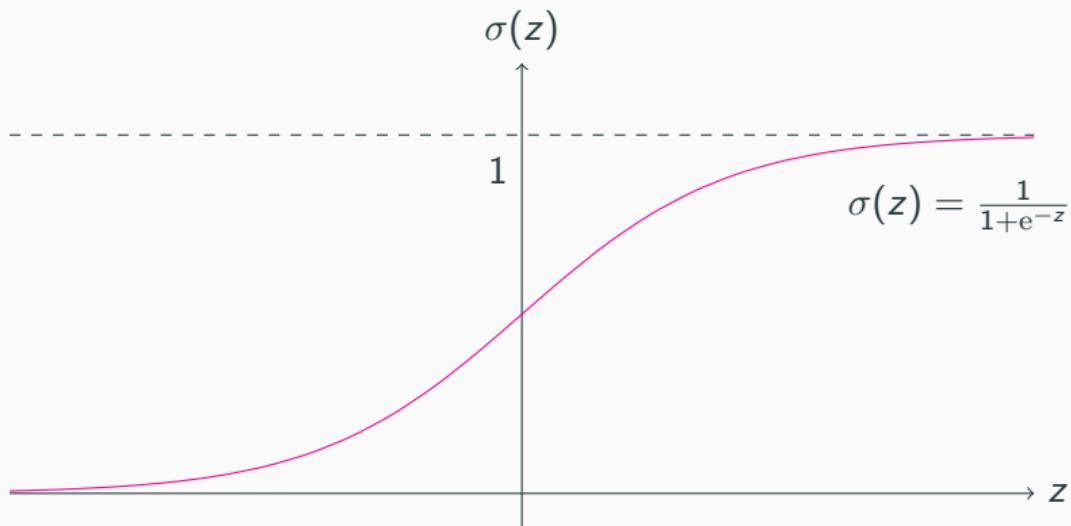
Quelle est cette fonction π telle que :

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

Régression logistique : Cas binaire

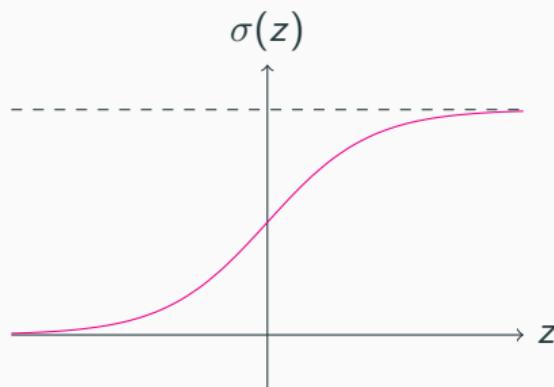
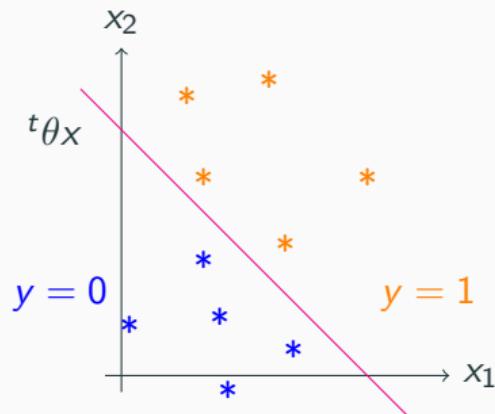
La **sigmoïde** ou **fonction logistique** définie par :

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



Régression logistique : Cas binaire

Cas 2D où les données sont linéairement séparables :



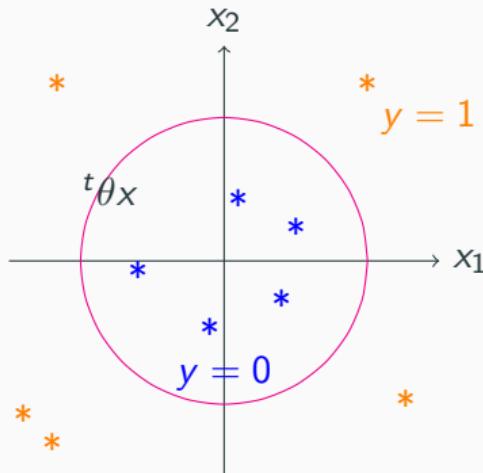
$$h_{\theta}(x) = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

- on prédit $\hat{y} = 1$ si $x_1 + x_2 > 3$,
- on prédit $\hat{y} = 0$ si $x_1 + x_2 \leq 3$.

$$y = 3 - x_1 - x_2$$

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 $x_1^2 + x_2^2 \leq 1$.

$$x_1^2 + x_2^2 = 1$$

Régression logistique : Cas binaire

Comment estimer automatiquement θ ?

- On a un **corpus d'apprentissage \mathcal{D}** , contenant m exemples avec : $\forall i \in \llbracket 1, m \rrbracket$,
$$x^{(i)} = (x_0^{(i)}, x_1^{(i)}, \dots, x_d^{(i)}) \in \mathbb{R}^{d+1} \quad \text{et} \quad y^{(i)} \in \{0, 1\},$$
- et le **modèle** est défini par la *fonction sigmoïde* : $\mathbb{P}(y = 1|x; \theta) = \frac{1}{1 + e^{-t_{\theta x}}}.$

~~~ **Idée** : Minimiser **fonction objectif** par **Descente de gradient**.

# Régression logistique : Cas binaire

*Comment estimer automatiquement  $\theta$  ?*

- On a un **corpus d'apprentissage  $\mathcal{D}$** , contenant  $m$  exemples avec :  $\forall i \in \llbracket 1, m \rrbracket$ ,

$$x^{(i)} = (x_0^{(i)}, x_1^{(i)}, \dots, x_d^{(i)}) \in \mathbb{R}^{d+1} \quad \text{et} \quad y^{(i)} \in \{0, 1\},$$

- et le **modèle** est défini par la *fonction sigmoïde* :  $\mathbb{P}(y = 1|x; \theta) = \frac{1}{1 + e^{-t_{\theta x}}}.$

~~~ **Idée** : Minimiser **fonction objectif** par **Descente de gradient**.

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

$$\begin{cases} J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{perte}\left(h_{\theta}(x^{(i)}), y^{(i)}\right) \\ \text{perte}\left(h_{\theta}(x^{(i)}), y^{(i)}\right) = \frac{1}{2} \left(h_{\theta}(x^{(i)}) - y^{(i)}\right)^2 = \frac{1}{2} \left(\frac{1}{1 + e^{-t_{\theta x^{(i)}}}} - y^{(i)}\right)^2 \end{cases}$$

Régression logistique : Cas binaire

Comment estimer automatiquement θ ?

- On a un **corpus d'apprentissage \mathcal{D}** , contenant m exemples avec : $\forall i \in \llbracket 1, m \rrbracket$,

$$x^{(i)} = (x_0^{(i)}, x_1^{(i)}, \dots, x_d^{(i)}) \in \mathbb{R}^{d+1} \quad \text{et} \quad y^{(i)} \in \{0, 1\},$$

- et le **modèle** est défini par la *fonction sigmoïde* : $\mathbb{P}(y = 1|x; \theta) = \frac{1}{1 + e^{-t_{\theta x}}}.$

~~~ **Idée** : Minimiser **fonction objectif** par **Descente de gradient**.

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

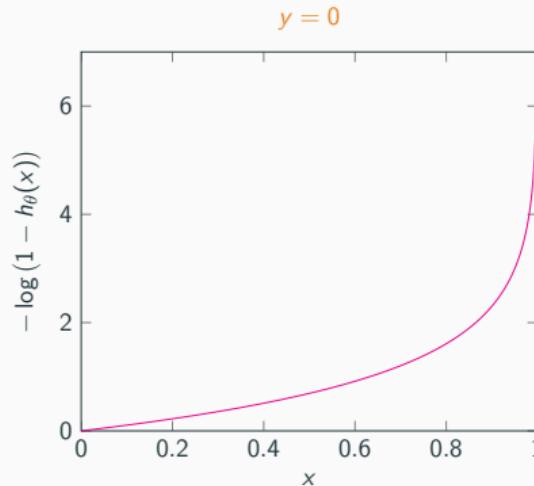
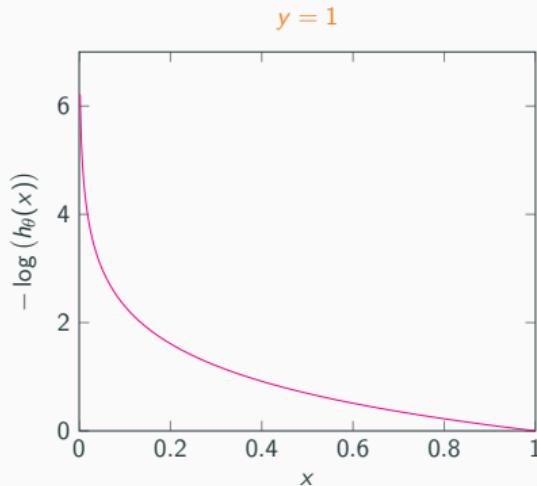
$$\begin{cases} J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{perte}\left(h_{\theta}(x^{(i)}), y^{(i)}\right) \\ \text{perte}\left(h_{\theta}(x^{(i)}), y^{(i)}\right) = \frac{1}{2} \left(h_{\theta}(x^{(i)}) - y^{(i)}\right)^2 = \frac{1}{2} \left(\frac{1}{1 + e^{-t_{\theta x^{(i)}}}} - y^{(i)}\right)^2 \end{cases}$$

**Problème** : 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 :

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



## 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.

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

## Précision sur le calcul précédent

Pour le calcul de  $\frac{\partial}{\partial \theta_j} [\log(h_\theta(x))]$ , on pose  $z = {}^t \theta x$ ,  $u = \sigma(z)$  et  $v = \log(u)$ .

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

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  $k > 2$  ?

1. On utilise la **fonction softmax** :

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

avec  $y^{(i)} = {}^t(0 \dots 0 1 0 \dots 0)$  où 1 à la  $i$ ème coordonnée ;

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

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

## Régression logistique : Cas multiclasse

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

3. La fonction objectif devient :

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

4. Le gradient s'écrit :

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

# Course Outline

Une brève histoire de l'Intelligence Artificielle

Apprentissage statistique

Régression linéaire et régression logistique

**Perceptron monocouche**

Réseaux de neurones (perceptron multicouche)

Fonctions d'activation

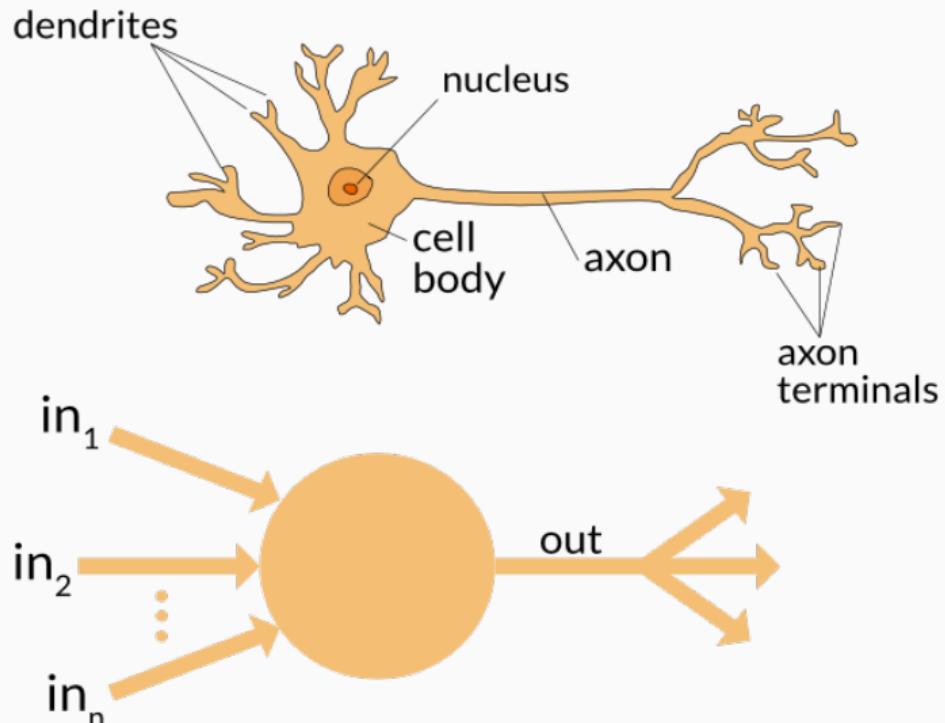
Optimiseurs

Sous-apprentissage et sur-apprentissage

Régularisation

Méthodologie en apprentissage profond

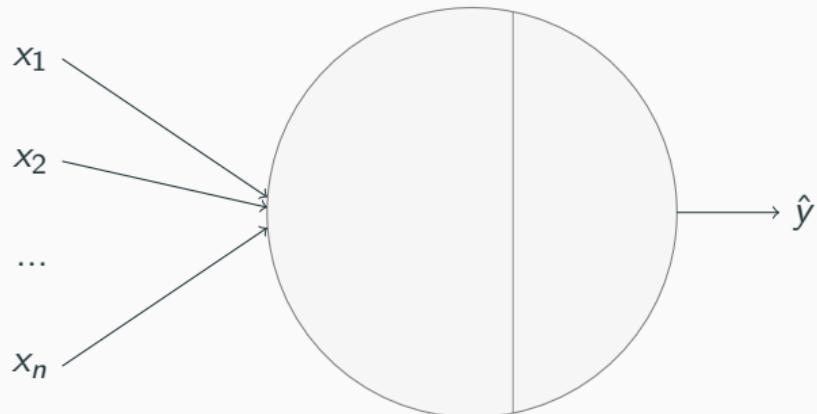
## Perceptron monocouche



**Figure 5 – Structure d'un neurone biologique/artificiel**

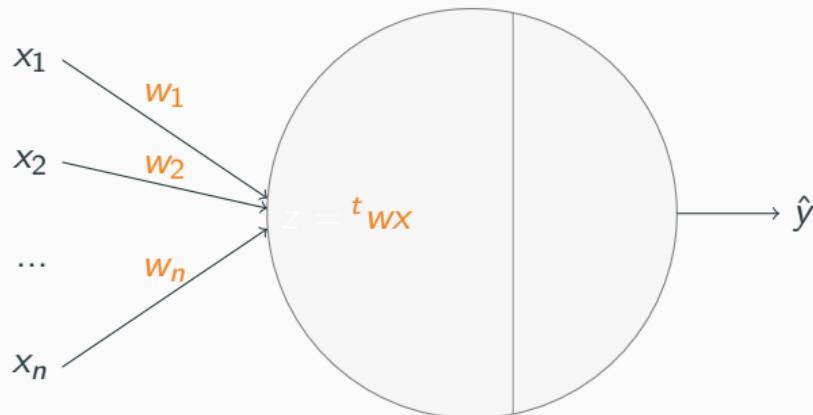
## Perceptron monocouche : Fonctionnement

**Neurone artificiel**  $\longleftrightarrow$  triplet : poids synaptique  $w$ , biais  $b$ , fonction d'activation  $f$ .



## Perceptron monocouche : Fonctionnement

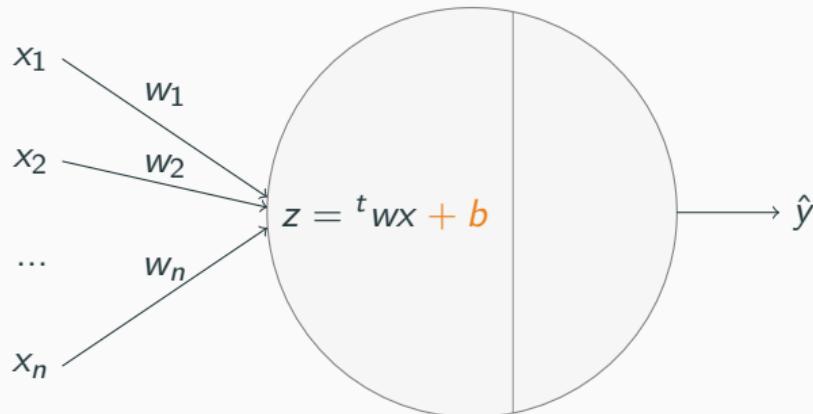
**Neurone artificiel**  $\longleftrightarrow$  triplet : **poids synaptique  $w$** , biais  $b$ , fonction d'activation  $f$ .



1. Produit scalaire entre les entrées  $x$  et les **poids synaptique  $w$**  :  ${}^t w x$  ;

## Perceptron monocouche : Fonctionnement

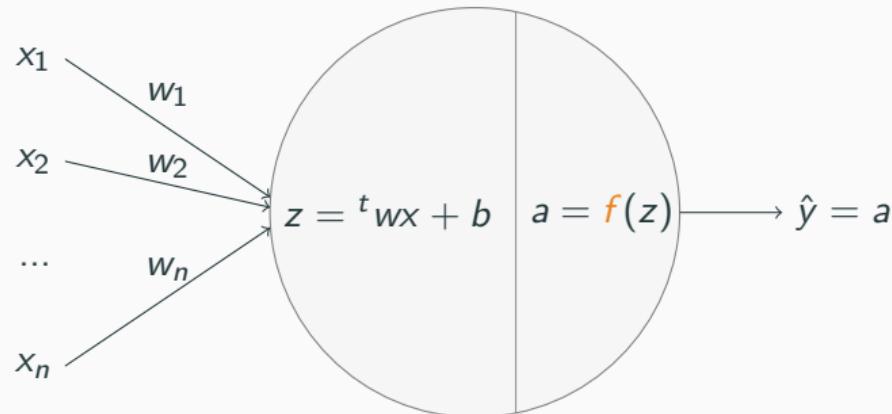
**Neurone artificiel**  $\longleftrightarrow$  triplet : poids synaptique  $w$ , biais  $b$ , fonction d'activation  $f$ .



1. Produit scalaire entre les entrées  $x$  et les **poids synaptique**  $w$  :  ${}^t w x$  ;
2. Ajout d'une valeur de référence (**biais**  $b$ ) :  $z = {}^t w x + b$  ;

## Perceptron monocouche : Fonctionnement

**Neurone artificiel**  $\longleftrightarrow$  triplet : poids synaptique  $w$ , biais  $b$ , fonction d'activation  $f$ .

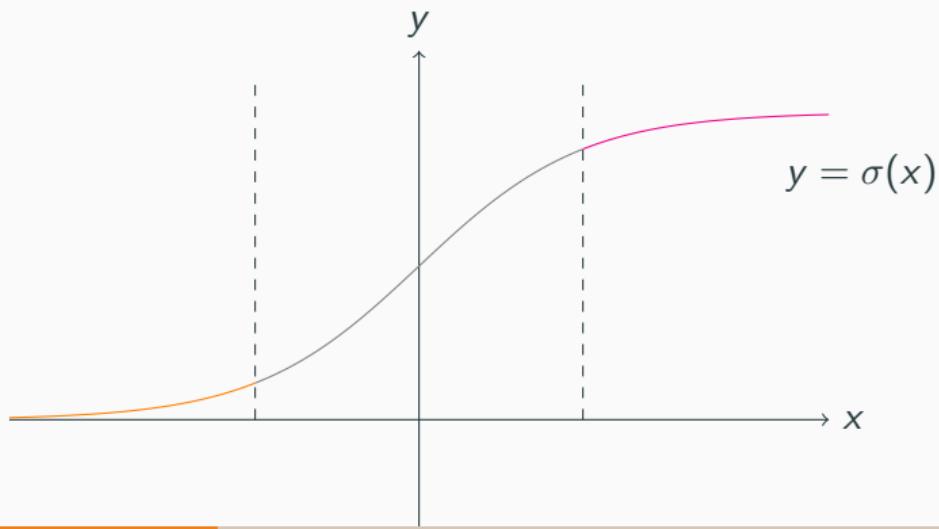


1. Produit scalaire entre les entrées  $x$  et les **poids synaptique**  $w$  :  ${}^t w x$  ;
2. Ajout d'une valeur de référence (**biais**  $b$ ) :  $z = {}^t w x + b$  ;
3. Application de la **fonction d'activation** à la valeur obtenue  $z$  :  $a = f(z)$  .

## 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 :

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



---

### Algorithm 1 Algorithme du perceptron monocouche

---

1. **Initialisation** des poids  $w_j^{\{0\}}$  ;
2. Présentation d'un vecteur d'entrées  $x^{(1)}, \dots, x^{(m)}$ ,  
et du vecteur de sortie correspondantes  $y^{(1)}, \dots, y^{(m)}$  ;

## Perceptron monocouche

---

### Algorithm 2 Algorithme du perceptron monocouche

---

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

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

## Perceptron monocouche

---

### Algorithm 3 Algorithme du perceptron monocouche

---

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

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

4. Mise à jour des **poids** : Soit un taux d'apprentissage  $0 \leq \alpha \leq 1$ ,

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

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

## Réseau de neurones : Perceptron monocouche

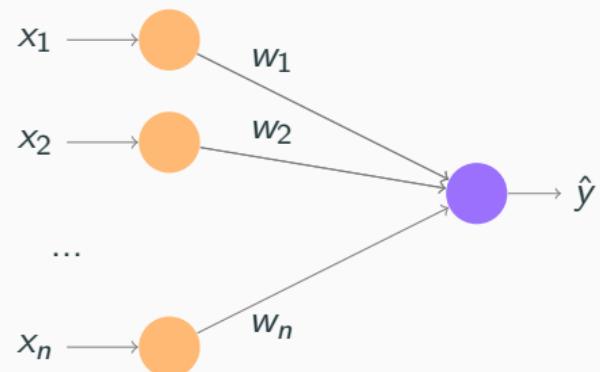
- Fonction d'activation  $f$ ,

- Fonction de coût  $J$ ,

- Descente de gradient :

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

Entrées  $x^{(1)}, \dots, x^{(m)}$



## Réseau de neurones : Perceptron monocouche

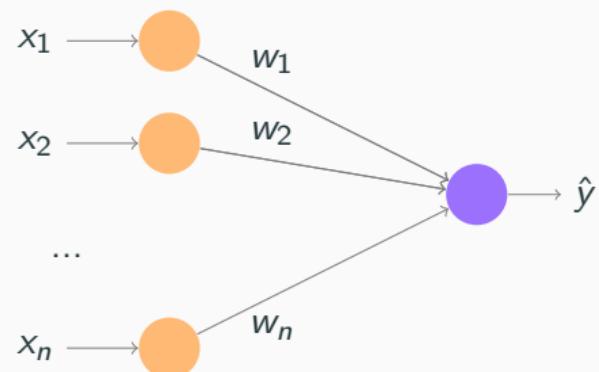
- Fonction d'activation linéaire  $f(x) = x$ ,

- Fonction de coût  $J$ ,

- Descente de gradient :

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

Entrées  $x^{(1)}, \dots, x^{(m)}$



## Réseau de neurones : Perceptron monocouche

- Fonction d'activation linéaire  $f(x) = x$ ,

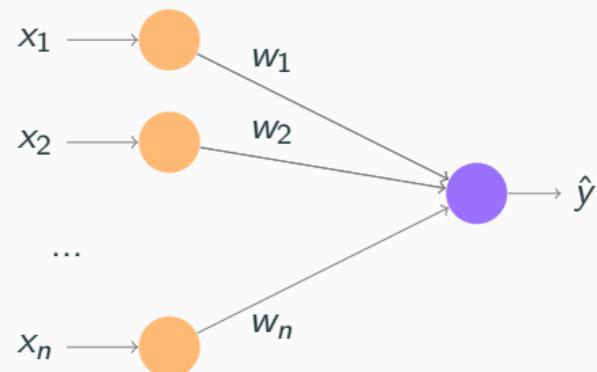
- Fonction de coût quadratique :

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

Entrées  $x^{(1)}, \dots, x^{(m)}$

- Descente de gradient :

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



## Réseau de neurones : Perceptron monocouche

- Fonction d'activation **linéaire**  $f(x) = x$ ,

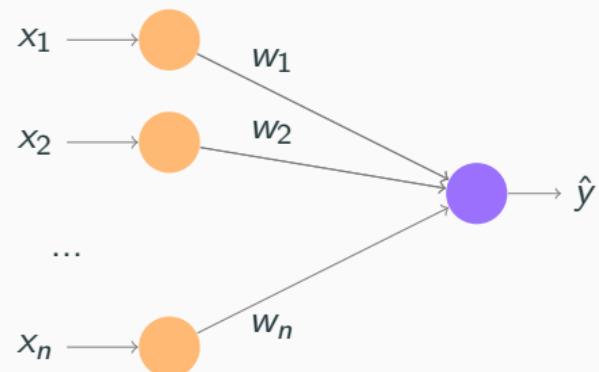
- Fonction de coût **quadratique** :

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

- Descente de gradient :

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

Entrées  $x^{(1)}, \dots, x^{(m)}$



## Réseau de neurones : Perceptron monocouche

- Fonction d'activation **linéaire**  $f(x) = x$ ,

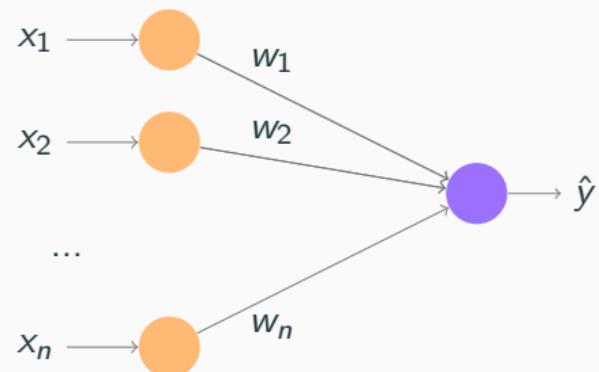
- Fonction de coût **quadratique** :

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

- Descente de gradient :

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

Entrées  $x^{(1)}, \dots, x^{(m)}$



↝ Perceptron monocouche équivalent à la **régression linéaire** !

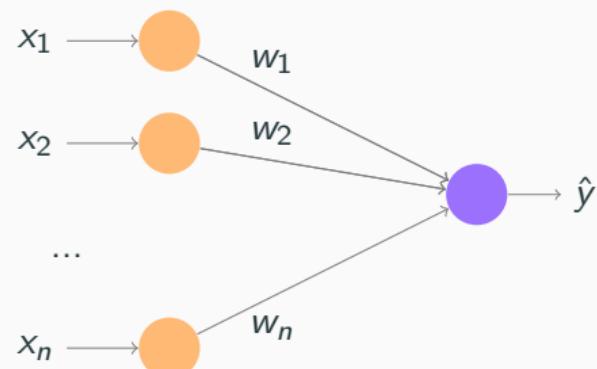
## Réseau de neurones : Perceptron monocouche

- Fonction d'activation **sigmoïde**  $\sigma(x) = \frac{1}{1+e^{-x}}$ ,

- Fonction de coût **entropie croisée** :

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

Entrées  $x^{(1)}, \dots, x^{(m)}$



- Descente de gradient :

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

↗ Perceptron monocouche équivalent à la **régression logistique** !

# Course Outline

Une brève histoire de l'Intelligence Artificielle

Apprentissage statistique

Régression linéaire et régression logistique

Perceptron monocouche

Réseaux de neurones (perceptron multicouche)

Fonctions d'activation

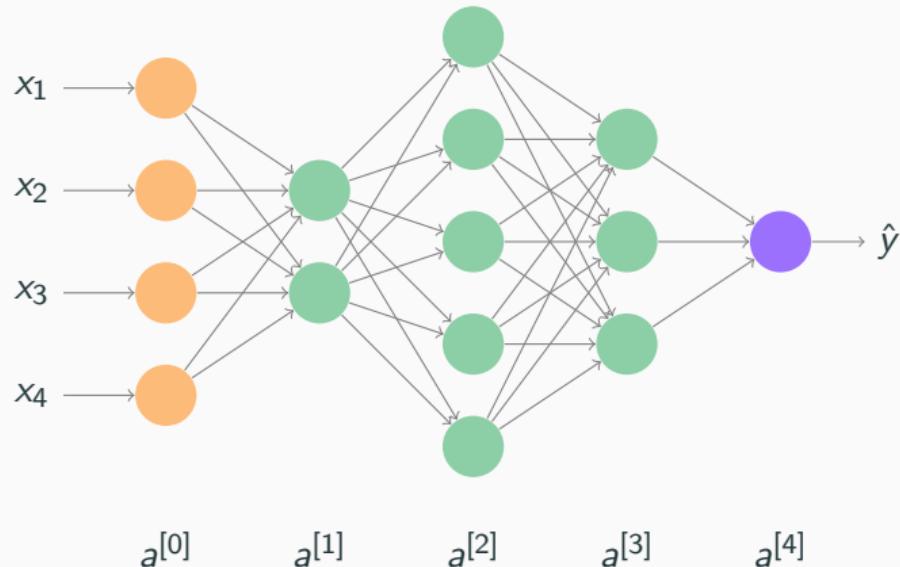
Optimiseurs

Sous-apprentissage et sur-apprentissage

Régularisation

Méthodologie en apprentissage profond

## 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

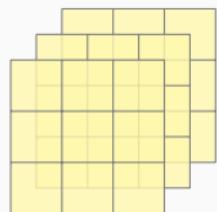
A ce stade, vous avez déjà remarqué la lourdeur des notations :

- 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 :  $a^{[c]}$ .

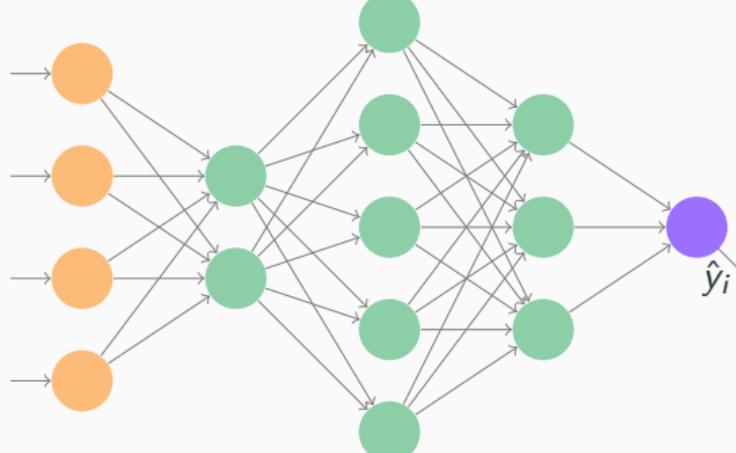
Ainsi, par exemple,  $a_j^{(i)[c]\{k\}}$  désigne l’activation du  $j$ -ième neurone de la couche  $c$ , calculée depuis la  $i$ -ème donnée, lors de la  $k$ -ème itération de la descente de gradient.

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

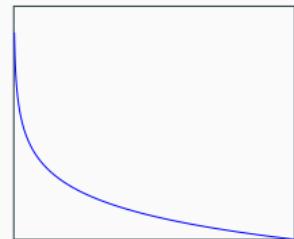
# Vue d'ensemble



Données  
 $(x_i, y_i)$



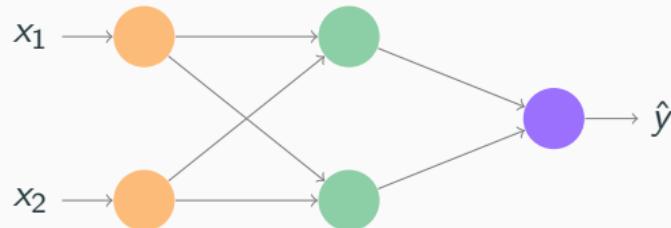
Optimisation



Fonction  
de coût

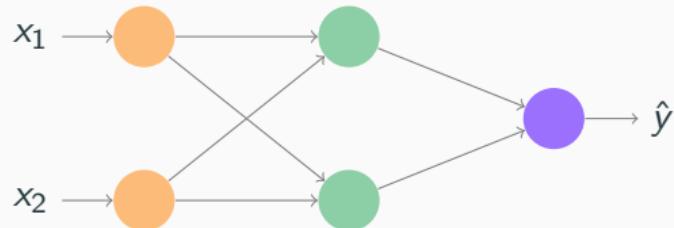
## Réseau de neurones : perceptrons multicouche

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 :



## Réseau de neurones : perceptrons multicouche

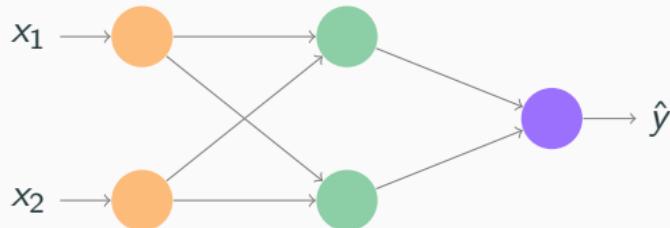
Après la première phase d'initialisation, l'algorithme d'apprentissage (basé sur la descente de gradient) comporte **5 étapes** qui se répètent jusqu'à convergence :



1. Propagation des données de la couche d'entrée à la couche de sortie ;

## Réseau de neurones : perceptrons multicouche

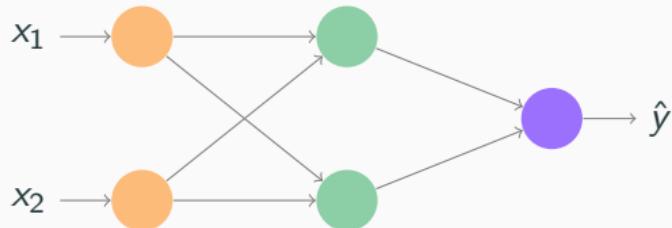
Après la première phase d'initialisation, l'algorithme d'apprentissage (basé sur la descente de gradient) comporte **5 étapes** qui se répètent jusqu'à convergence :



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 ;

## Réseau de neurones : perceptrons multicouche

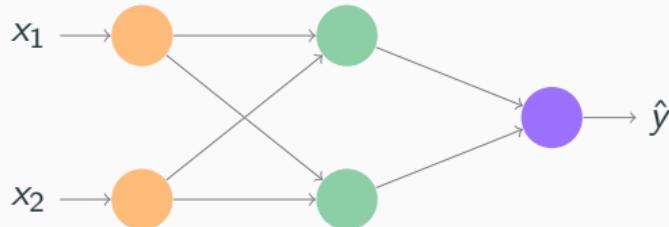
Après la première phase d'initialisation, l'algorithme d'apprentissage (basé sur la descente de gradient) comporte **5 étapes** qui se répètent jusqu'à convergence :



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** ;

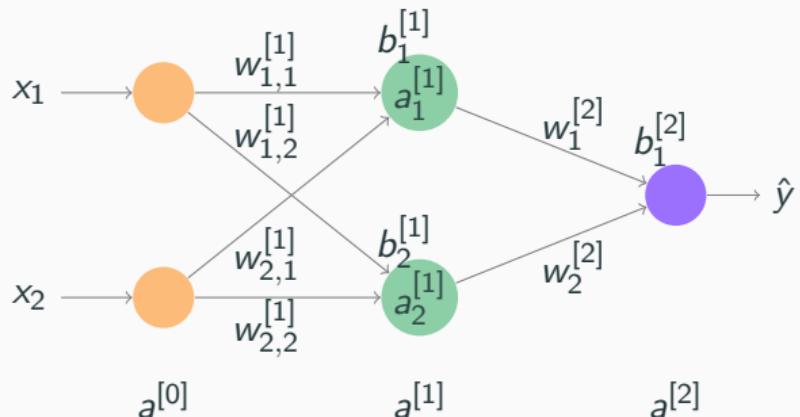
## Réseau de neurones : perceptrons multicouche

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 :



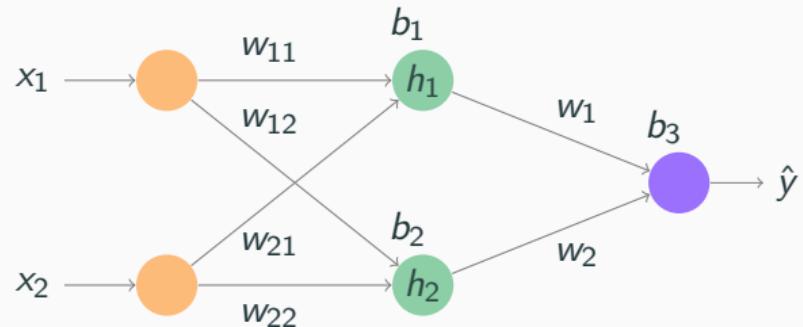
1. **Propagation des données** de la couche d'entrée à la couche de sortie ;
2. Calcul de l'**erreur de sortie** après la propagation des données ;
3. Calcul des **gradients d'erreurs** pour corriger les poids synaptiques des neurones de la **couche de sortie** ;
4. Calcul des **gradients d'erreurs**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

## Perceptrons multicouche : Un exemple



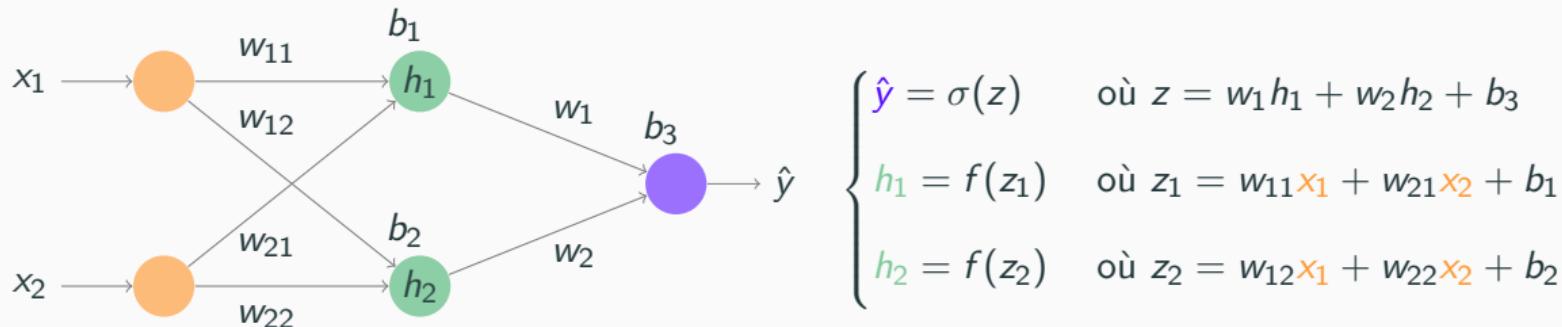
Simplifions un peu les notations (pour enlever l'indice de couche).

## Illustration de l'entraînement d'un perceptron multicouche



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

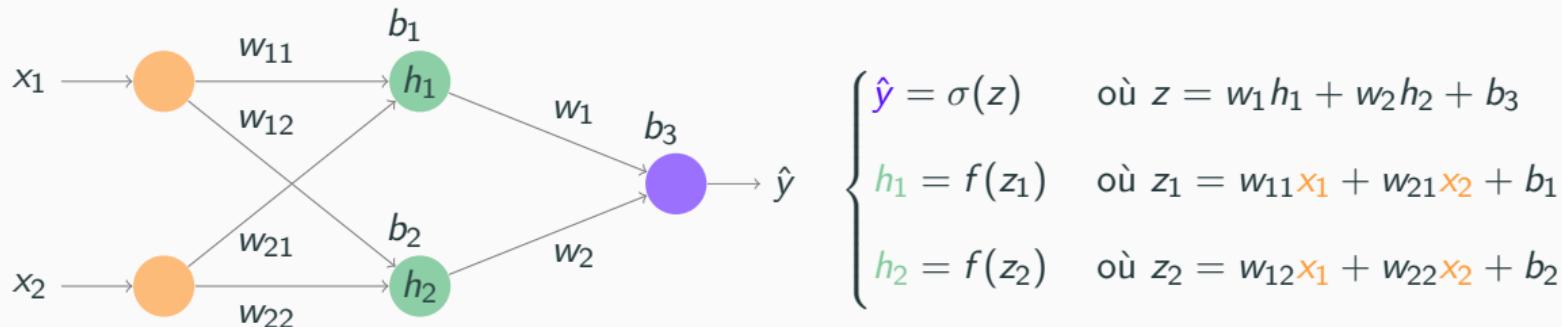
## Illustration de l'entraînement d'un perceptron multicouche



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

$$\hat{y} = \sigma \left( w_1 \times \underbrace{\mathbf{f}(w_{11}x_1 + w_{12}x_2 + b_1)}_{h_1} + w_2 \times \underbrace{\mathbf{f}(w_{12}x_1 + w_{22}x_2 + b_2)}_{h_2} + b_3 \right);$$

## Illustration de l'entraînement d'un perceptron multicouche



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

$$\hat{y} = \sigma \left( w_1 \times \underbrace{f(w_{11}x_1 + w_{12}x_2 + b_1)}_{h_1} + w_2 \times \underbrace{f(w_{12}x_1 + w_{22}x_2 + b_2)}_{h_2} + b_3 \right);$$

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

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

## Illustration de l'entraînement d'un perceptron multicouche

3. Calcul des **gradients d'erreurs** pour corriger les poids synaptiques des neurones de la **couche de sortie** :

En utilisant la *règle de la chaîne*, i.e.

$$\frac{\partial}{\partial x} f(y(x)) = \frac{\partial}{\partial y} f(y) \cdot \frac{\partial}{\partial x} y(x)$$

## Illustration de l'entraînement d'un perceptron multicouche

3. Calcul des **gradients d'erreurs** pour corriger les poids synaptiques des neurones de la **couche de sortie** :

En utilisant la *règle de la chaîne*, i.e.

$$\frac{\partial}{\partial x} f(y(x)) = \frac{\partial}{\partial y} f(y) \cdot \frac{\partial}{\partial x} y(x)$$

$$\forall j \in \{1, 2\}, \quad \frac{\partial J}{\partial w_j} = ?$$

$$\left\{ \begin{array}{l} J(w) = \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log(\hat{y}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \\ \hat{y} = \sigma(w_1 \times \mathbf{f}(w_{11}x_1 + w_{12}x_2 + b_1) + w_2 \times \mathbf{f}(w_{21}x_1 + w_{22}x_2 + b_2) + b_3) \end{array} \right.$$

## Illustration de l'entraînement d'un perceptron multicouche

3. Calcul des **gradients d'erreurs** pour corriger les poids synaptiques des neurones de la **couche de sortie** :

En utilisant la *règle de la chaîne*, i.e.

$$\frac{\partial}{\partial x} f(y(x)) = \frac{\partial}{\partial y} f(y) \cdot \frac{\partial}{\partial x} y(x)$$

$$\forall j \in \{1, 2\}, \quad \frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_j}$$

$$\left\{ \begin{array}{l} J(w) = \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log(\hat{y}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \\ \hat{y} = \sigma(w_1 \times \mathbf{f}(w_{11}x_1 + w_{12}x_2 + b_1) + w_2 \times \mathbf{f}(w_{21}x_1 + w_{22}x_2 + b_2) + b_3) \end{array} \right.$$

## Illustration de l'entraînement d'un perceptron multicouche

3. Calcul des **gradients d'erreurs** pour corriger les poids synaptiques des neurones de la **couche de sortie** :

En utilisant la *règle de la chaîne*, i.e.  $\frac{\partial}{\partial x} f(y(x)) = \frac{\partial}{\partial y} f(y) \cdot \frac{\partial}{\partial x} y(x)$

$$\forall j \in \{1, 2\}, \quad \frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_j} \quad \text{et} \quad \frac{\partial J}{\partial b_3} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial b_3};$$

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

$$\forall j, j' \in \{1, 2\}, \quad \frac{\partial J}{\partial w_{jj'}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_{j'}} \cdot \frac{\partial h_{j'}}{\partial z_{j'}} \cdot \frac{\partial z_{j'}}{\partial w_{jj'}} \quad \text{et} \quad \frac{\partial J}{\partial b_j} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_j} \cdot \frac{\partial h_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial b_j};$$

$$\left\{ \begin{array}{l} J(w) = \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log(\hat{y}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \\ \hat{y} = \sigma(w_1 \times \mathbf{f}(w_{11}x_1 + w_{12}x_2 + b_1) + w_2 \times \mathbf{f}(w_{21}x_1 + w_{22}x_2 + b_2) + b_3) \end{array} \right.$$

## Illustration de l'entraînement d'un perceptron multicouche

3. Calcul des **gradients d'erreurs** pour corriger les poids synaptiques des neurones de la **couche de sortie** :

En utilisant la *règle de la chaîne*, i.e.  $\frac{\partial}{\partial x} f(y(x)) = \frac{\partial}{\partial y} f(y) \cdot \frac{\partial}{\partial x} y(x)$

$$\forall j \in \{1, 2\}, \quad \frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_j} \quad \text{et} \quad \frac{\partial J}{\partial b_3} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial b_3};$$

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

$$\forall j, j' \in \{1, 2\}, \quad \frac{\partial J}{\partial w_{jj'}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_{j'}} \cdot \frac{\partial h_{j'}}{\partial z_{j'}} \cdot \frac{\partial z_{j'}}{\partial w_{jj'}} \quad \text{et} \quad \frac{\partial J}{\partial b_j} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_j} \cdot \frac{\partial h_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial b_j};$$

5. **Mise à jour** des poids synaptiques de la couche de sortie, et de la couche cachée :

$$w_j \leftarrow w_j - \alpha \frac{\partial J}{\partial w_j} \quad \text{et} \quad w_{jj'} \leftarrow w_{jj'} - \alpha \frac{\partial J}{\partial w_{jj'}}$$

## Remarque : Rétropropagation du gradient

**Remarque** : Calcule des gradients des poids synaptiques :

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_j} \quad \text{et} \quad \frac{\partial J}{\partial w_{jj'}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_{j'}} \cdot \frac{\partial h_{j'}}{\partial z_{j'}} \cdot \frac{\partial z_{j'}}{\partial w_{jj'}} ,$$

## Remarque : Rétropropagation du gradient

**Remarque :** Calcule des gradients des poids synaptiques :

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_j} \quad \text{et} \quad \frac{\partial J}{\partial w_{jj'}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_{j'}} \cdot \frac{\partial h_{j'}}{\partial z_{j'}} \cdot \frac{\partial z_{j'}}{\partial w_{jj'}} ,$$

## Remarque : Rétropropagation du gradient

**Remarque :** Calcule des gradients des poids synaptiques :

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_j} \quad \text{et} \quad \frac{\partial J}{\partial w_{jj'}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_{j'}} \cdot \frac{\partial h_{j'}}{\partial z_{j'}} \cdot \frac{\partial z_{j'}}{\partial w_{jj'}} ,$$

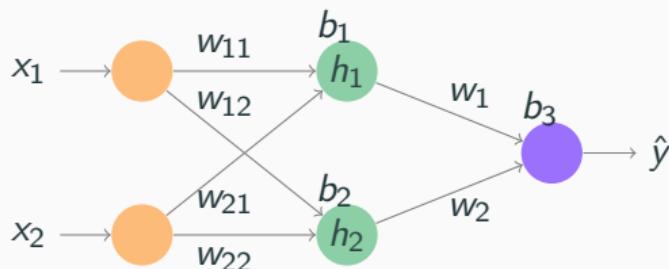
↔ Il est intéressant de calculer d'abord  $\frac{\partial J}{\partial w_j}$ ,  
puis de réutiliser les calculs intermédiaires pour ensuite calculer  $\frac{\partial J}{\partial w_{jj'}}$ .

## Remarque : Rétropropagation du gradient

Remarque : Calcule des gradients des poids synaptiques :

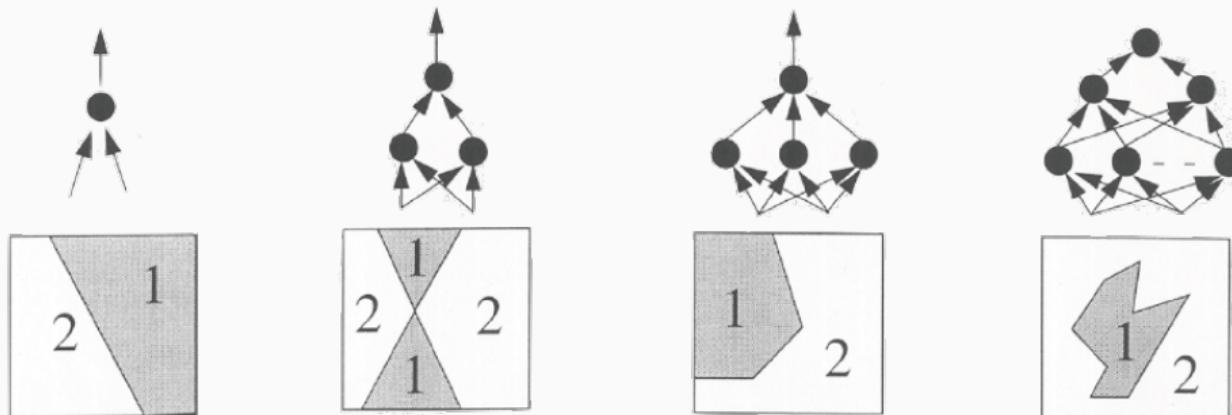
$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_j} \quad \text{et} \quad \frac{\partial J}{\partial w_{jj'}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_{j'}} \cdot \frac{\partial h_{j'}}{\partial z_{j'}} \cdot \frac{\partial z_{j'}}{\partial w_{jj'}} ,$$

↔ 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 6 – 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

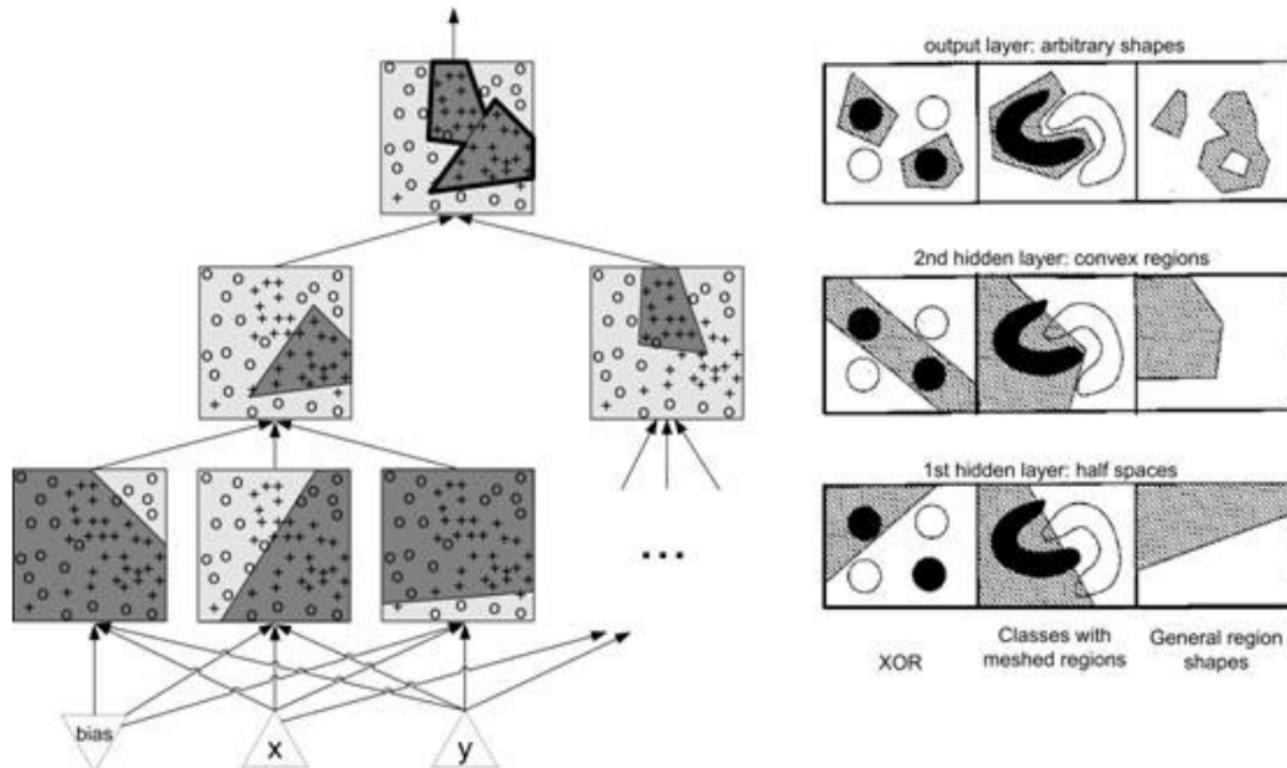


Figure 7 – Intérêt du perceptron multicouche

# 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 sigmoïde).

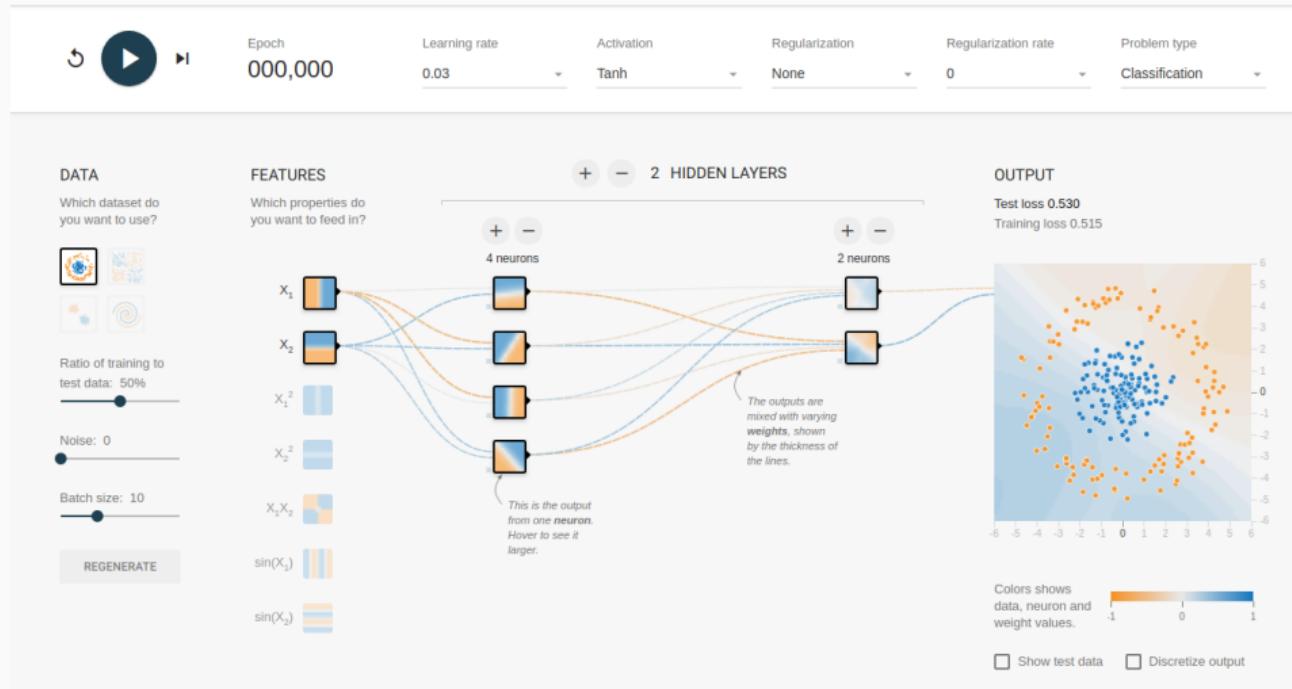
Note : le théorème a été également prouvé avec la fonction reLU.

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



# Visualisation

<https://playground.tensorflow.org/>



# Course Outline

Une brève histoire de l'Intelligence Artificielle

Apprentissage statistique

Régression linéaire et régression logistique

Perceptron monocouche

Réseaux de neurones (perceptron multicouche)

## Fonctions d'activation

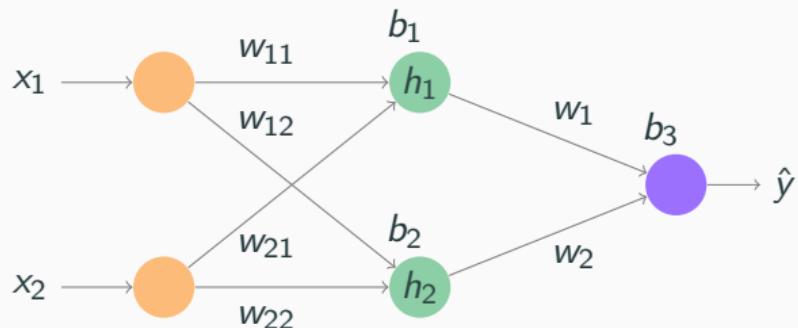
Optimiseurs

Sous-apprentissage et sur-apprentissage

Régularisation

Méthodologie en apprentissage profond

## Retour sur le calcul des gradients

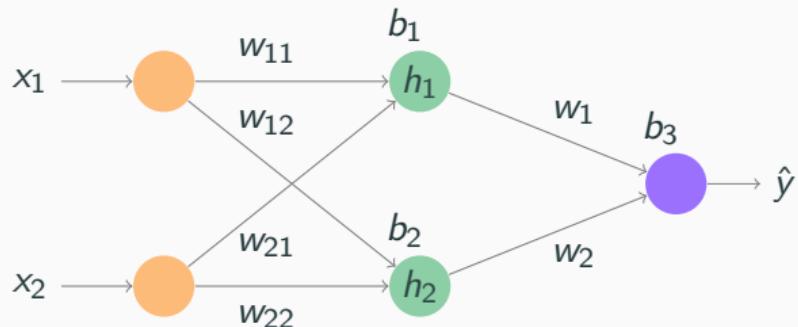


$$\begin{cases} \hat{y} = \sigma(z) & \text{où } z = w_1 h_1 + w_2 h_2 + b_3 \\ h_1 = f(z_1) & \text{où } z_1 = w_{11} x_1 + w_{21} x_2 + b_1 \\ h_2 = f(z_2) & \text{où } z_2 = w_{12} x_1 + w_{22} x_2 + b_2 \end{cases}$$

D'après la règle de la chaîne,  $\forall i, j \in \{1, 2\}$ ,

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_j} \cdot \frac{\partial h_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}.$$

## Retour sur le calcul des gradients



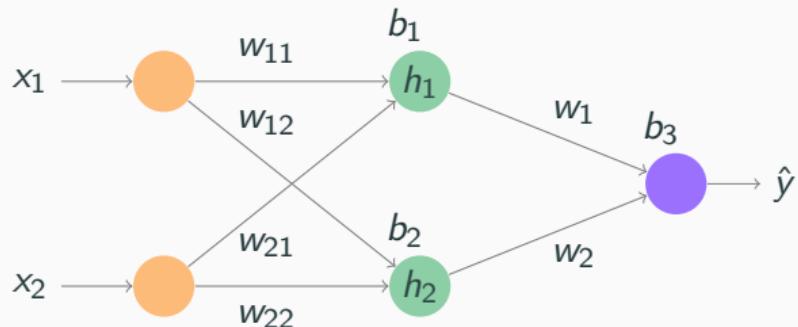
$$\begin{cases} \hat{y} = \sigma(z) & \text{où } z = w_1h_1 + w_2h_2 + b_3 \\ h_1 = f(z_1) & \text{où } z_1 = w_{11}x_1 + w_{21}x_2 + b_1 \\ h_2 = f(z_2) & \text{où } z_2 = w_{12}x_1 + w_{22}x_2 + b_2 \end{cases}$$

D'après la règle de la chaîne,  $\forall i, j \in \{1, 2\}$ ,  $\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_j} \cdot \frac{\partial h_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}$ .

Soit  $f'$  la dérivée de la fonction d'activation portée par le neurone. On a :

$$\frac{\partial z_j}{\partial w_{ij}} = x_i$$

## Retour sur le calcul des gradients



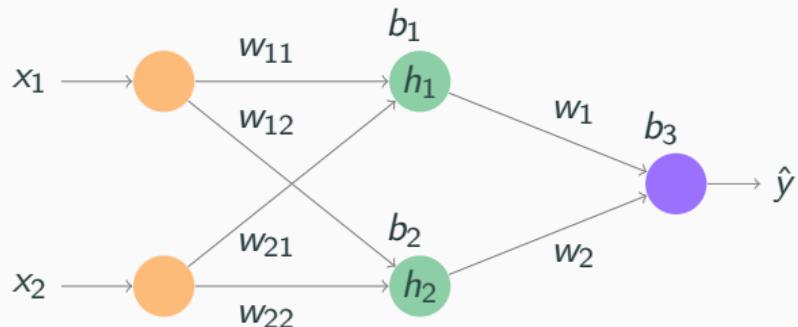
$$\begin{cases} \hat{y} = \sigma(z) & \text{où } z = w_1 h_1 + w_2 h_2 + b_3 \\ h_1 = f(z_1) & \text{où } z_1 = w_{11} x_1 + w_{21} x_2 + b_1 \\ h_2 = f(z_2) & \text{où } z_2 = w_{12} x_1 + w_{22} x_2 + b_2 \end{cases}$$

D'après la règle de la chaîne,  $\forall i, j \in \{1, 2\}$ ,  $\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_j} \cdot \frac{\partial h_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}$ .

Soit  $f'$  la dérivée de la fonction d'activation portée par le neurone. On a :

$$\frac{\partial z_j}{\partial w_{ij}} = x_i \quad , \quad \frac{\partial h_j}{\partial z_j} = f'(z_j)$$

## Retour sur le calcul des gradients



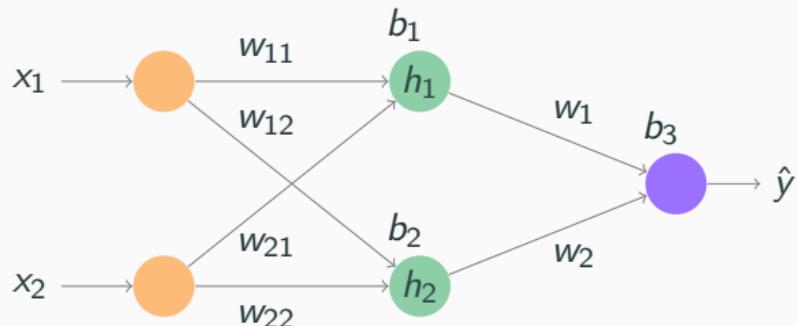
$$\begin{cases} \hat{y} = \sigma(z) & \text{où } z = w_1 h_1 + w_2 h_2 + b_3 \\ h_1 = f(z_1) & \text{où } z_1 = w_{11} x_1 + w_{12} x_2 + b_1 \\ h_2 = f(z_2) & \text{où } z_2 = w_{21} x_1 + w_{22} x_2 + b_2 \end{cases}$$

D'après la règle de la chaîne,  $\forall i, j \in \{1, 2\}$ ,  $\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_j} \cdot \frac{\partial h_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}$ .

Soit  $f'$  la dérivée de la fonction d'activation portée par le neurone. On a :

$$\frac{\partial z_j}{\partial w_{ij}} = x_i \quad , \quad \frac{\partial h_j}{\partial z_j} = f'(z_j) \quad \text{et} \quad \frac{\partial z}{\partial h_j} = w_j .$$

## Retour sur le calcul des gradients



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

D'après la règle de la chaîne,  $\forall i, j \in \{1, 2\}$ ,

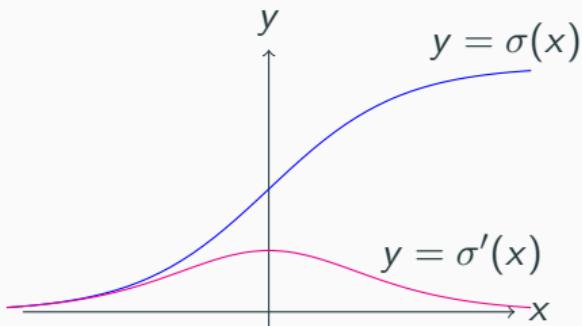
$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial h_j} \cdot \frac{\partial h_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}.$$

Soit  $f'$  la **dérivée de la fonction d'activation** portée par le neurone. On a :

$$\frac{\partial z_j}{\partial w_{ij}} = x_i \quad , \quad \frac{\partial h_j}{\partial z_j} = f'(z_j) \quad \text{et} \quad \frac{\partial z}{\partial h_j} = w_j .$$

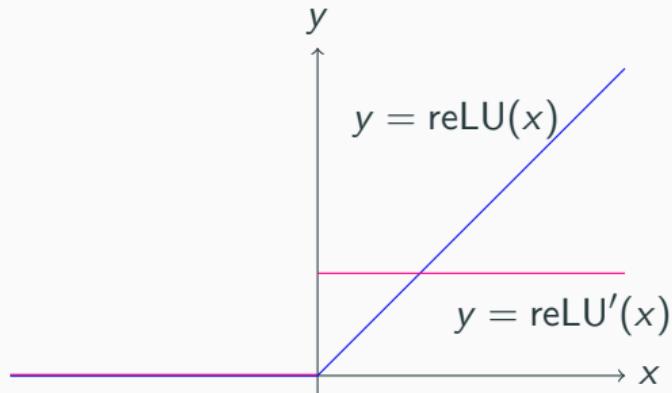
~~> Importance du choix de la fonction d'activation !

## Gros plan sur la fonction sigmoïde



- Dérivée de la fonction sigmoïde est à valeurs dans  $[0, \frac{1}{4}]$ ,
    - ~~ Diminue l'amplitude des gradients propagés à travers les couches du réseau de neurones.
- ⇒ Problème de l'**évanescence des gradients** (**vanishing gradients**).

## La fonction *rectified Linear Unit*



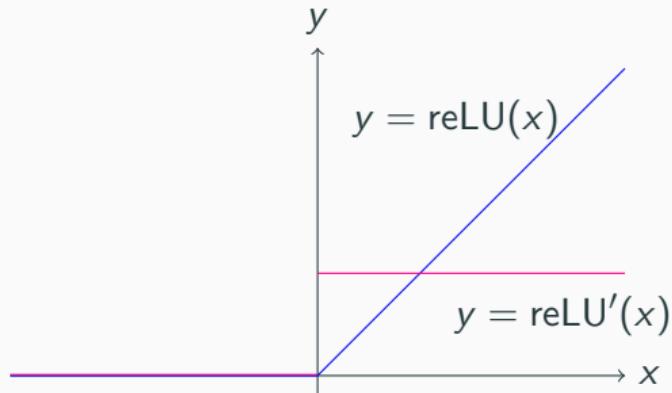
$$\text{ReLU}(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{sinon} \end{cases}$$

Gradient constamment égal à

1 dans la zone activée

↝ Améliore la convergence.

## La fonction *rectified Linear Unit*



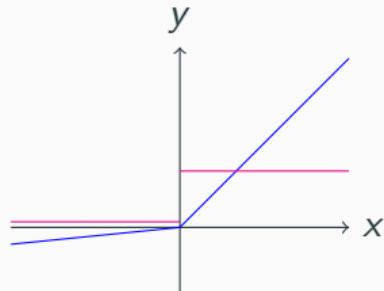
$$\text{ReLU}(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{sinon} \end{cases}$$

Gradient constamment égal à 1 dans la zone activée  
~~> Améliore la convergence.

**Mais !**

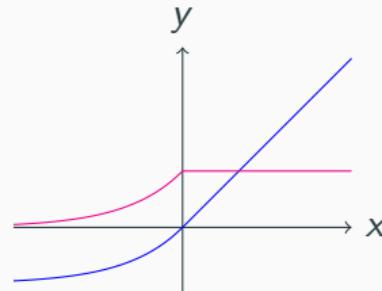
- Pas dérivable en 0,
- Dérivée nulle sur  $\mathbb{R}^-$ 
  - ~~> Pas de propagation si activation négative,
- À valeur dans  $\mathbb{R}^+$ 
  - ~~> Optimisation plus compliquée si beaucoup de couches cachées.

## D'autres variantes



$$\text{leakyReLU}(x) = \begin{cases} \alpha x & \text{si } x < 0 \\ x & \text{sinon} \end{cases}$$

*Leaky Rectified Linear Unit*

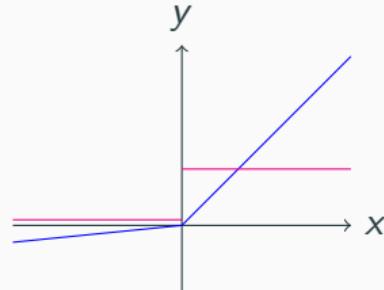


$$\text{eLU}(x) = \begin{cases} \alpha(e^x - 1) & \text{si } x < 0 \\ x & \text{sinon} \end{cases}$$

*Exponential Linear Unit*

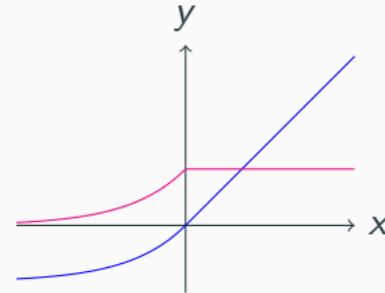
- Moins d'occurrences où le gradient est nul
  - ~~> Meilleure convergence de la descente de gradient.
- Activations négatives possibles
  - ~~> Problème d'optimisation mieux conditionné.

## D'autres variantes



$$\text{leakyReLU}(x) = \begin{cases} \alpha x & \text{si } x < 0 \\ x & \text{sinon} \end{cases}$$

*Leaky Rectified Linear Unit*



$$\text{eLU}(x) = \begin{cases} \alpha(e^x - 1) & \text{si } x < 0 \\ x & \text{sinon} \end{cases}$$

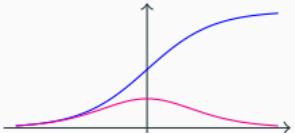
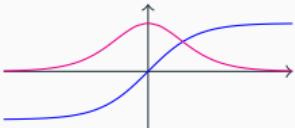
*Exponential Linear Unit*

- Moins d'occurrences où le gradient est nul
  - ~~> Meilleure convergence de la descente de gradient.
- Activations négatives possibles
  - ~~> Problème d'optimisation mieux conditionné.



Meilleur entraînement  
≠ Meilleure performance !

# Fonctions d'activation usuelles

| Fonction d'activation | Image                                                                                      | Centré en 0           | Saturation                                   | Gradient évanescent               | Coût numérique |
|-----------------------|--------------------------------------------------------------------------------------------|-----------------------|----------------------------------------------|-----------------------------------|----------------|
| <b>Sigmoïde</b>       |  [0, 1]   | Non                   | Valeurs <i>positives</i> et <i>négatives</i> | Oui                               | Lourd          |
| <b>Tanh</b>           |  [-1, 1]  | Oui                   | Valeurs <i>positives</i> et <i>négatives</i> | Oui                               | Lourd          |
| <b>ReLu</b>           |  [0, +∞[  | Non                   | Valeurs <i>négatives</i>                     | Oui<br>(mais moins que σ et tanh) | Facile         |
| <b>Leaky ReLu</b>     |  ]-∞, +∞[ | Non<br>(mais presque) | Non                                          | Non                               | Facile         |

# Course Outline

Une brève histoire de l'Intelligence Artificielle

Apprentissage statistique

Régression linéaire et régression logistique

Perceptron monocouche

Réseaux de neurones (perceptron multicouche)

Fonctions d'activation

## Optimiseurs

Sous-apprentissage et sur-apprentissage

Régularisation

Méthodologie en apprentissage profond

# Apprentissage supervisé et Risque empirique

En apprentissage supervisé, on va minimiser un *risque empirique* !

**Apprentissage supervisé :**

1. On observe  $m$  réalisations d'un couple  $(X, Y) \sim \mathbb{P}(\mathcal{X} \times \mathcal{Y})$ , de loi  $\mathbb{P}$  inconnue;
2. On définit une **règle de prédiction**  $f: \mathcal{X} \rightarrow \mathcal{Y}$  mesurable  $\rightsquigarrow \hat{y} = f(x) \in \mathcal{Y}$  ;
3. On définit une **fonction de perte**  $\ell: \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}^+$  :  
$$\forall y, y' \in \mathcal{Y} \quad \text{tque} \quad y \neq y', \quad \ell(y, y') = 0 \quad \text{et} \quad \ell(y, y') > 0;$$
4. On définit le **risque**, ou *erreur généralisée*, associé à la fonction de perte  $\ell$  et à la règle de prédiction  $f : \mathcal{R}_{\mathbb{P}}(f) = \mathbb{E}_{(X, Y) \sim \mathbb{P}} [\ell(Y, f(X))]$  ;

5. On approche ce risque par son *estimateur empirique*

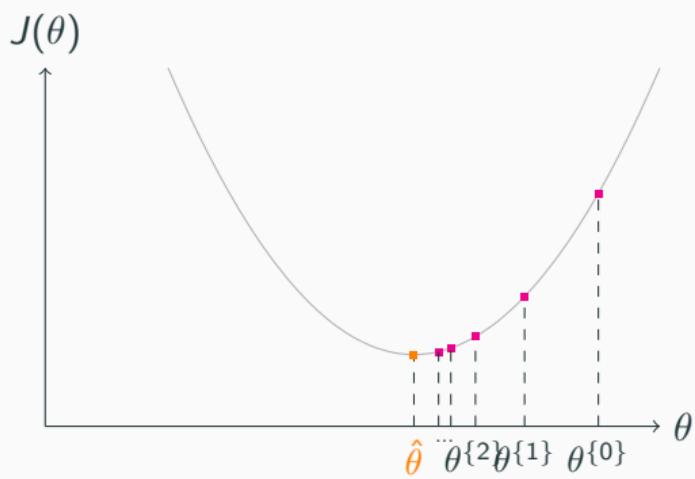
$\rightsquigarrow$  Risque empirique :

$$\mathcal{R}_n(f) = \frac{1}{n} \sum_{i=1}^n \ell(y^{(i)}, f(x^{(i)}))$$

*Espérance approchée  
par somme de  
Monte-Carlo !*

## Rappel : Descente de gradient

Modèle paramétrique :  $f = f_{\theta} \implies \begin{cases} J^*(\theta) := \mathcal{R}_{\mathbb{P}}(f_{\theta}) = \mathbb{E}_{(X,Y) \sim \mathbb{P}} [\ell(Y, f_{\theta}(X))] \\ J(\theta) := \mathcal{R}_n(f_{\theta}) = \frac{1}{n} \sum_{i=1}^n \ell(y^{(i)}, f_{\theta}(x^{(i)})) \end{cases}$



**Algorithme** : Descente de gradient

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

**TANT QUE** pas convergence

**POUR**  $j$  de 1 à  $d$

$$\theta_j^{\{k+1\}} = \theta_j^{\{k\}} - \eta \frac{\partial J^*(\theta^{\{k\}})}{\partial \theta_j}$$

**FIN POUR**

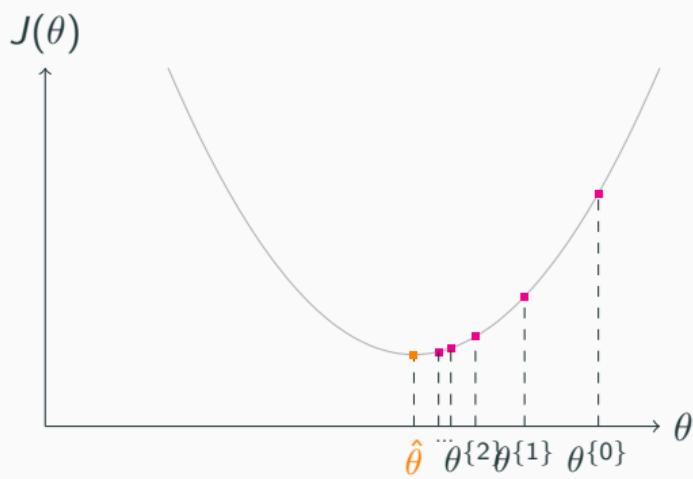
$k \leftarrow k + 1$

**FIN TANT QUE**

- Convergence vers un minimum local de  $J^*$ ,
- Un seul hyperparamètre  $\eta$ .

## Rappel : Descente de gradient

Modèle paramétrique :  $f = f_{\theta} \implies \begin{cases} J^*(\theta) := \mathcal{R}_{\mathbb{P}}(f_{\theta}) = \mathbb{E}_{(X,Y) \sim \mathbb{P}} [\ell(Y, f_{\theta}(X))] \\ J(\theta) := \mathcal{R}_n(f_{\theta}) = \frac{1}{n} \sum_{i=1}^n \ell(y^{(i)}, f_{\theta}(x^{(i)})) \end{cases}$



**Algorithme** : Descente de gradient

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

**TANT QUE** pas convergence

**POUR**  $j$  de 1 à  $d$

$$\theta_j^{(k+1)} = \theta_j^{(k)} - \eta \frac{\partial J^*(\theta^{(k)})}{\partial \theta_j}$$

**FIN POUR**

$k \leftarrow k + 1$

**FIN TANT QUE**

- Convergence vers un minimum local de  $J^*$ ,
- Un seul hyperparamètre  $\eta$ .

**Problème** : On a pas accès à  $J^*$  !

~~> Adapter la descente de gradient à  $J_{62}$

## Descente de gradient stochastique

Étant donné le risque empirique  $J(\theta) = \frac{1}{m} \sum_{i=1}^m \ell(y^{(i)}, f_\theta(x^{(i)}))$ , on a :

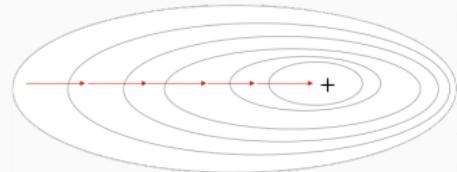
$$\nabla J(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla \ell(y^{(i)}, f_\theta(x^{(i)}))$$

↔ Espérance de  $\nabla \ell$  par somme de Monte-Carlo

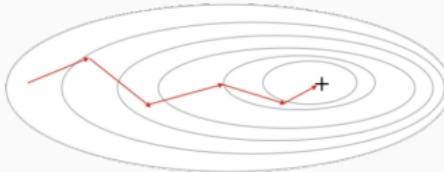
Plusieurs manière d'approcher cette espérance, *i.e.* plusieurs alternatives pour l'algorithme de la descente de gradient :

- Par somme de Monte-Carlo ↽ **Descente de gradient batch**, ou classique ;  
On calcule le gradient en utilisant **tous** les  $n$  échantillon de l'ensemble d'apprentissage.
- Par un seul élément de la somme ↽ **Descente de gradient stochastique** (SGD) ;  
On calcule le gradient en utilisant **1** seul échantillon de l'ensemble d'apprentissage.
- Entre les deux ↽ **Descente de gradient mini-batch**.  
On calcule le gradient en utilisant  $1 < m \ll n$  échantillons parmi les  $n$  de l'ensemble d'apprentissage.

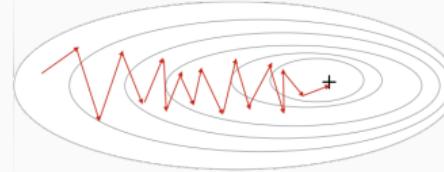
# Descente de gradient stochastique



(a) Descente de gradient **batch**

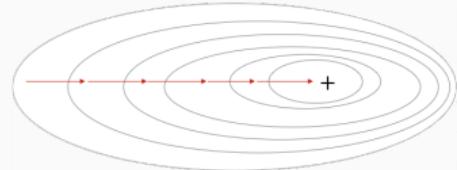


(b) Descente de gradient  
mini-batch



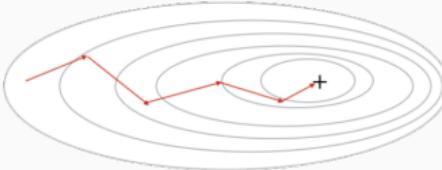
(c) Descente de gradient  
stochastique

# Descente de gradient stochastique



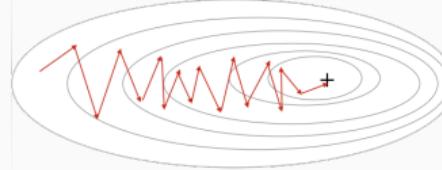
(a) Descente de gradient **batch**

$$\theta_j^{\{k+1\}} = \theta_j^{\{k\}} - \frac{\eta}{n} \sum_{i=1}^n \frac{\partial \ell_i(\theta^{\{k\}})}{\partial \theta_j}$$



(b) Descente de gradient  
mini-batch

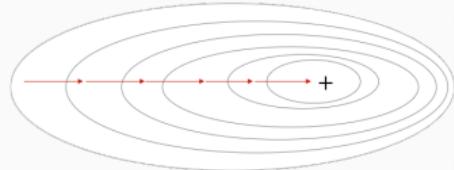
$$\theta_j^{\{k+1\}} = \theta_j^{\{k\}} - \frac{\eta}{m} \sum_{i=1}^m \frac{\partial \ell_i(\theta^{\{k\}})}{\partial \theta_j}$$



(c) Descente de gradient  
stochastique

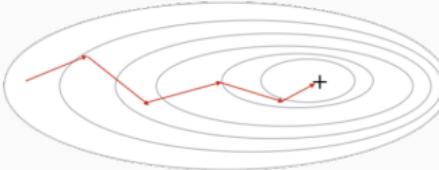
$$\theta_j^{\{k+1\}} = \theta_j^{\{k\}} - \eta_k \frac{\partial \ell_i(\theta^{\{k\}})}{\partial \theta_j}$$

# Descente de gradient stochastique



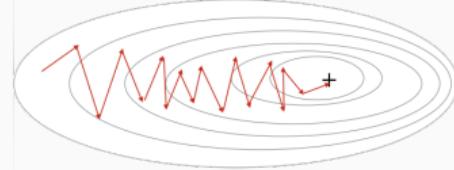
(a) Descente de gradient **batch**

$$\theta_j^{\{k+1\}} = \theta_j^{\{k\}} - \frac{\eta}{n} \sum_{i=1}^n \frac{\partial \ell_i(\theta^{\{k\}})}{\partial \theta_j}$$



(b) Descente de gradient  
mini-batch

$$\theta_j^{\{k+1\}} = \theta_j^{\{k\}} - \frac{\eta}{m} \sum_{i=1}^m \frac{\partial \ell_i(\theta^{\{k\}})}{\partial \theta_j}$$



(c) Descente de gradient  
stochastique

$$\theta_j^{\{k+1\}} = \theta_j^{\{k\}} - \eta_k \frac{\partial \ell_i(\theta^{\{k\}})}{\partial \theta_j}$$

## Convergence du SGD vers un minimum local

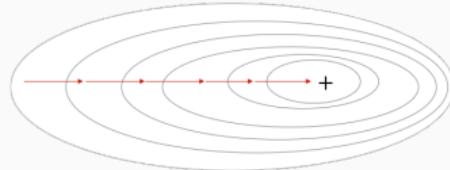
Supposons :

1.  $\sum_{k=1}^{\infty} \eta_k < \infty$  et  $\sum_{k=1}^{\infty} \eta_k^2 = \infty$

2. Des hypothèses de régularité sur  $J \longleftrightarrow \ell$ .

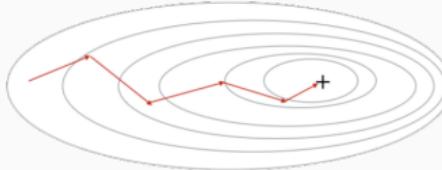
Alors la suite  $\theta^{\{k\}}$  converge vers un minimum local de  $J$ .

# Descente de gradient stochastique



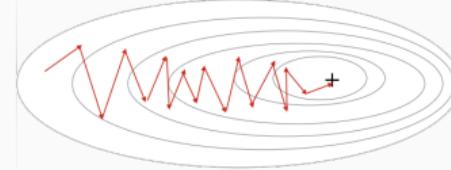
(a) Descente de gradient **batch**

$$\theta_j^{\{k+1\}} = \theta_j^{\{k\}} - \frac{\eta}{n} \sum_{i=1}^n \frac{\partial \ell_i(\theta^{\{k\}})}{\partial \theta_j}$$



(b) Descente de gradient **mini-batch**

$$\theta_j^{\{k+1\}} = \theta_j^{\{k\}} - \frac{\eta}{m} \sum_{i=1}^m \frac{\partial \ell_i(\theta^{\{k\}})}{\partial \theta_j}$$



(c) Descente de gradient **stochastique**

$$\theta_j^{\{k+1\}} = \theta_j^{\{k\}} - \eta_k \frac{\partial \ell_i(\theta^{\{k\}})}{\partial \theta_j}$$

## Remarques :

- Par abus de langage, on utilise le terme SGD y compris pour l'algorithme Mini-batch,
- Un mini-batch trop petit peut engendrer un bruit trop grand sur l'estimation du gradient et empêcher la convergence.

## Convergence du SGD vers un minimum local

Supposons :

$$1. \sum_{k=1}^{\infty} \eta_k < \infty \text{ et } \sum_{k=1}^{\infty} \eta_k^2 = \infty$$

2. Des hypothèses de régularité sur  $J \longleftrightarrow \ell$ .

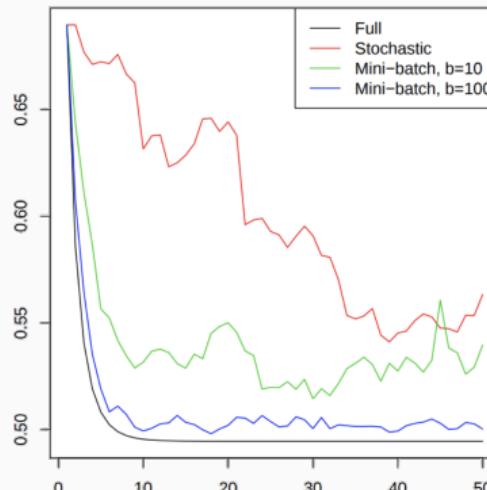
Alors la suite  $\theta^{\{k\}}$  converge vers un minimum local de  $J$ .

## Variantes de la descente de gradient

On parle d'**epoch** lorsque l'ensemble d'apprentissage a été visité **entièrement** pour le calcul des gradients.

Au final on a donc :

- Descente de gradient **batch** :  
1 itération par *epoch*.
- Descente de gradient **mini-batch** :  
 $\frac{n}{m}$  itérations par *epoch*.
- Descente de gradient **stochastique** :  
 $n$  itérations par *epoch*.

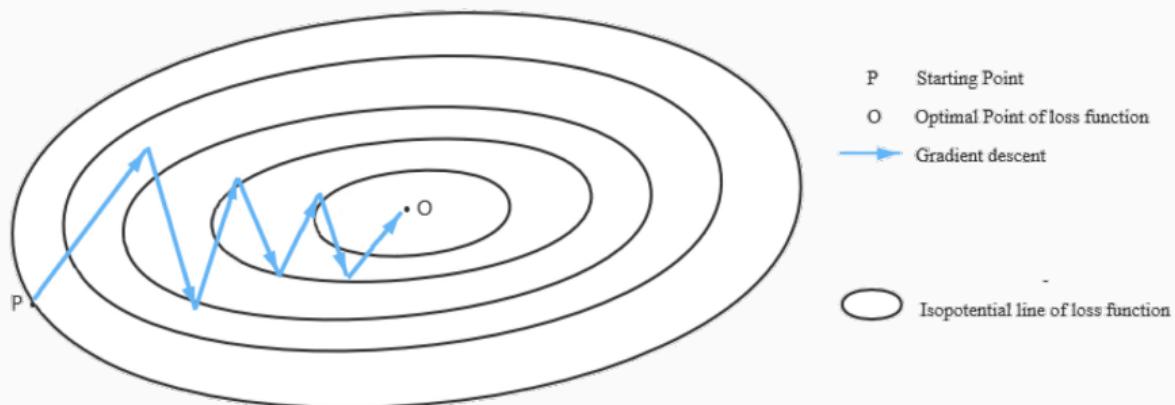


---

**“Meilleur choix”** : SGD mini-batch, avec petite taille de mini-batch  $m$ .

# Momentum

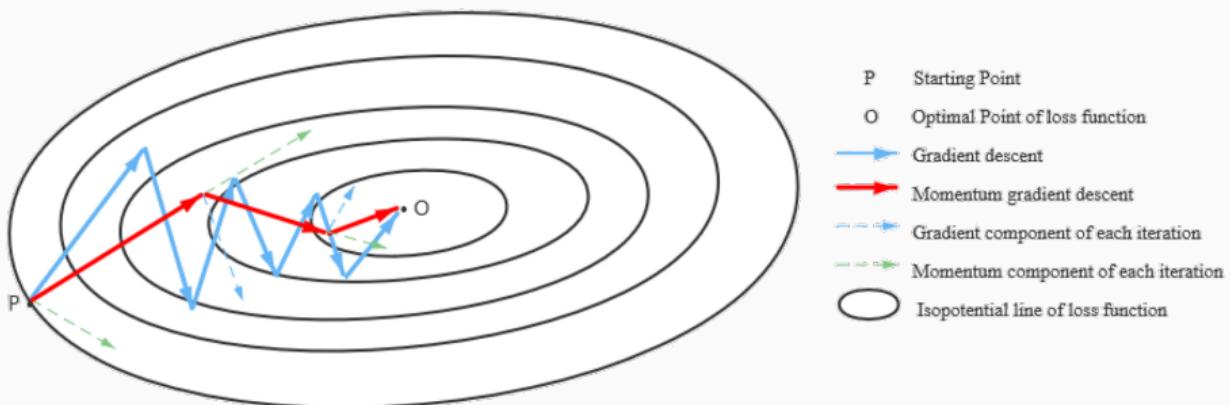
Le SGD mini-batch effectue la mise à jour des paramètres après avoir vu seulement un sous-ensemble de l'ensemble d'apprentissage.



# Momentum

Le SGD mini-batch effectue la mise à jour des paramètres après avoir vu seulement un sous-ensemble de l'ensemble d'apprentissage.

→ Pour réduire la variance, limiter les oscillation le long de la trajectoire de convergence et éviter de rester trop facilement bloqué dans un minimum local, on ajoute un terme d'**inertie** ou **momentum**.



## Momentum

**En pratique** : on adapte l'algorithme de descente du gradient pour tenir compte des gradients précédents et lisser la mise à jour :

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

**En pratique** : on adapte l'algorithme de descente du gradient pour tenir compte des gradients précédents et lisser la mise à jour :

$$\theta_j^{k+1} = \theta_j^k - \eta \frac{\partial J(\theta^k)}{\partial \theta_j} \quad \rightsquigarrow \quad \begin{cases} v^{k+1} &= \beta v^k - \eta \frac{\partial J(\theta^k)}{\partial \theta_j} \\ \theta_j^{k+1} &= \theta_j^k + v^{k+1} \end{cases}$$

Avec :

- **Vélocité  $v$**  : direction dans laquelle les paramètres vont être modifiés.  
Prend en compte les gradients précédents via le paramètre
- $\beta \in ]0, 1[$  : quantifie l'importance relative des gradients précédents par rapport au gradient courant.

**Remarque** :  $\beta \simeq 0.9$  en général

## Optimiseurs améliorant la descente de gradient stochastique

Dans les espaces paramétriques de grande dimension, la topologie de la fonction objectif rend la descente de gradient difficile voire inefficace. On peut améliorer cette dernière en utilisant des optimiseurs adaptés.

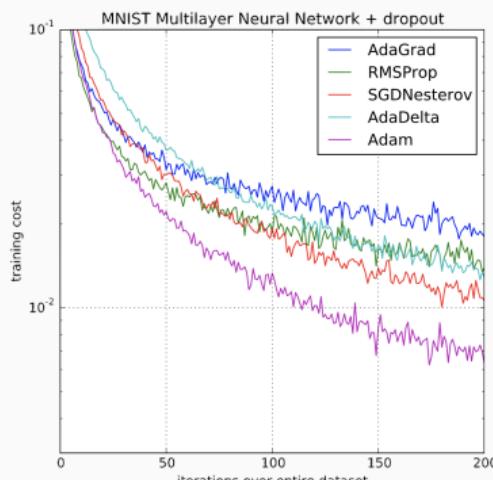
# Optimiseurs améliorant la descente de gradient stochastique

Dans les espaces paramétriques de grande dimension, la topologie de la fonction objectif rend la descente de gradient difficile voire inefficace. On peut améliorer cette dernière en utilisant des optimiseurs adaptés.

↝ NAG, AdaGrad, Adadelta, RMSProp, Adam, AdaMax, Nadam, AMSGrad, AdamW, QHAdam, YellowFin, AggMo, QHM, Demon Adam, etc.

## Deux postes de blog intéressants à ce sujet :

- [ruder.io/optimizing-gradient-descent](http://ruder.io/optimizing-gradient-descent/),
- [johnchenresearch.github.io/demon](http://johnchenresearch.github.io/demon).



## Optimiseurs améliorant la descente de gradient stochastique

L'optimiseur **AdaGrad** introduit une forme d'**adaptation** du taux d'apprentissage en accumulant les carrés des gradients précédents.

### Idée générale :

- Effectuer de plus *petites mises à jour*, c'est-à-dire des taux d'apprentissage faibles, pour les paramètres associés aux *caractéristiques fréquentes*,
- Et des mises à jour *plus importantes*, c'est-à-dire des taux d'apprentissage élevés, pour les paramètres associés aux caractéristiques *peu fréquentes*.

### Mise en œuvre :

1. Calcul du gradient :  $g^{\{k\}} = \frac{\partial J(\theta^{\{k\}})}{\partial \theta}$ ,
2. Accumulation des gradients :  $r^{\{k+1\}} = r^{\{k\}} + \|g^{\{k\}}\|_2$ ,
3. Mise à jour des paramètres :  $\theta^{\{k+1\}} = \theta^{\{k\}} - \frac{\eta}{\sqrt{r^{\{k\}}}} g^{\{k\}}$ .

## Optimiseurs améliorant la descente de gradient stochastique

L'optimiseur **RMSProp** est presque identique à AdaGrad, mais l'impact des plus anciens gradients est altéré par un coefficient multiplicatif  $\rho \in ]0, 1[$  (*weight decay*).

Cela améliore le comportement de l'algorithme dans le cas des borts allongés.

**Mise en œuvre :**

1. Calcul du gradient :  $g^{\{k\}} = \frac{\partial J(\theta^{\{k\}})}{\partial \theta}$ ,
2. Accumulation des gradients :  $r^{\{k+1\}} = \rho r^{\{k\}} + (1 - \rho) \|g^{\{k\}}\|_2$ ,
3. Mise à jour des paramètres :  $\theta^{\{k+1\}} = \theta^{\{k\}} - \frac{\eta}{\sqrt{r^{\{k\}}}} g^{\{k\}}$ .

Enfin, l'optimiseur **Adam** est similaire à RMSProp, mais **adapte** également le **momentum**.

## En pratique : Choix de l'optimiseur

Adam est souvent un bon choix pour débuter (avec le taux d'apprentissage "magique")

The screenshot shows a Twitter conversation between user @karpathy. The first tweet is from Andrej Karpathy (@karpathy) on November 24, 2016, stating: "3e-4 is the best learning rate for Adam, hands down." It has 24 replies, 128 retweets, and 474 likes. The second tweet is a reply from Andrej Karpathy (@karpathy) saying: "En réponse à @karpathy (i just wanted to make sure that people understand that this is a joke...)"

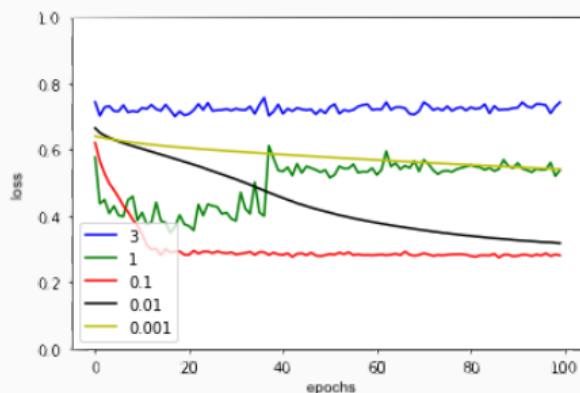
Andrej Karpathy ✅ @karpathy · 24 nov. 2016  
3e-4 is the best learning rate for Adam, hands down.  
24 128 474

Andrej Karpathy ✅ @karpathy  
En réponse à @karpathy  
(i just wanted to make sure that people understand that this is a joke...)

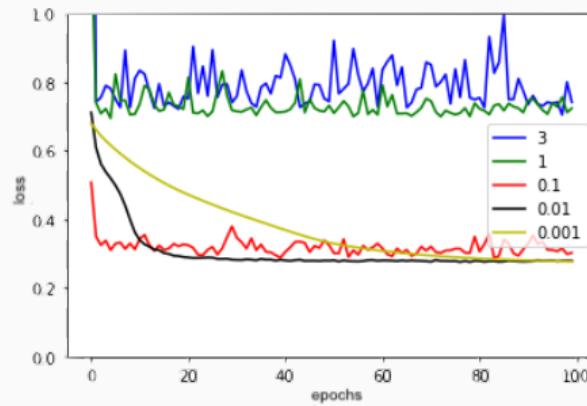
En pratique, les meilleurs résultats mis en avant dans les articles sont obtenus avec une simple descente de gradient stochastique, et une mise à jour programmée du taux d'apprentissage (cyclique, cosinus, etc.)

# Influence du taux d'apprentissage

Évolution de la perte d'apprentissage au cours de l'entraînement, pour différents taux d'apprentissage, avec SGD et Adam (classification binaire).

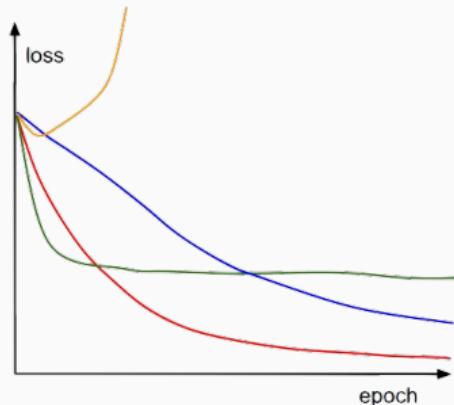


SGD



Adam

# Influence du taux d'apprentissage



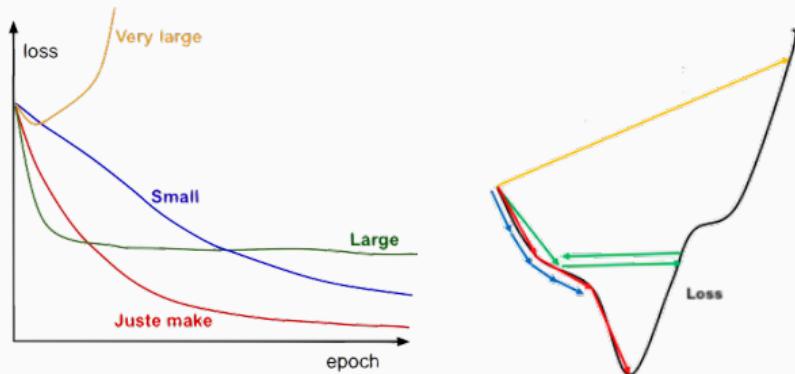
**Q1 :** La courbe jaune diverge car :

1. Le taux d'apprentissage est trop fort,
2. Le taux d'apprentissage est trop faible,
3. Cela n'a aucun rapport avec le taux d'apprentissage.

**Q2 :** Classez les courbes verte, rouge et bleue par ordre croissant de taux d'apprentissage :

1. verte - rouge - bleue
2. verte - bleue - rouge
3. rouge - bleue - verte
4. rouge - verte - bleue
5. bleue - verte - rouge
6. bleue - rouge - verte

# Influence du taux d'apprentissage



**Q1 :** La courbe jaune diverge car :

1. Le taux d'apprentissage est trop fort,
2. Le taux d'apprentissage est trop faible,
3. Cela n'a aucun rapport avec le taux d'apprentissage.

**Q2 :** Classez les courbes verte, rouge et bleue par ordre croissant de taux d'apprentissage :

1. verte - rouge - bleue
2. verte - bleue - rouge
3. rouge - bleue - verte
4. rouge - verte - bleue
5. bleue - verte - rouge
6. bleue - rouge - verte

# Course Outline

Une brève histoire de l'Intelligence Artificielle

Apprentissage statistique

Régression linéaire et régression logistique

Perceptron monocouche

Réseaux de neurones (perceptron multicouche)

Fonctions d'activation

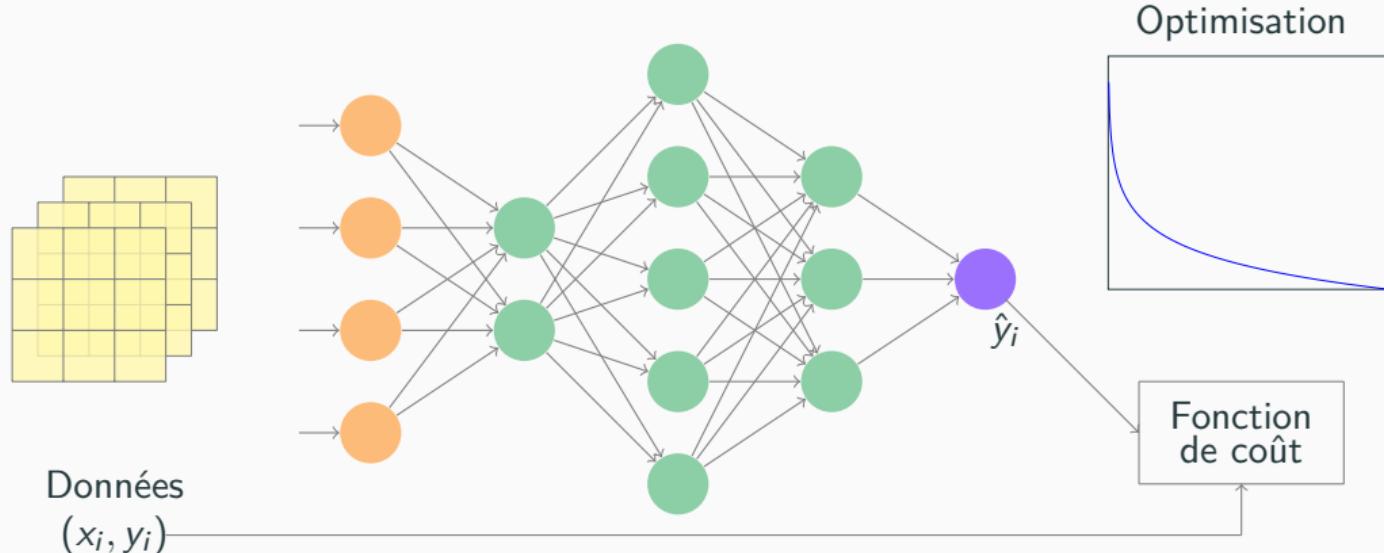
Optimiseurs

**Sous-apprentissage et sur-apprentissage**

Régularisation

Méthodologie en apprentissage profond

# Vue d'ensemble



---

## Paramètres

 $w_{i,j}$  $b_k$ 

## Hyper-paramètres

profondeur, nb neurones,  
activation,  
 $\eta$ , momentum

## L'apprentissage profond en pratique : Des limitations

**Risque empirique** : Lorsque l'on entraîne un réseau de neurones, notre but est de minimiser le *risque* ou erreur généralisée. Mais, nous ne sommes capables d'évaluer que le *risque empirique*...

**De plus** : Avec l'augmentation de la profondeur des réseaux de neurones, un certain nombre de problèmes se posent

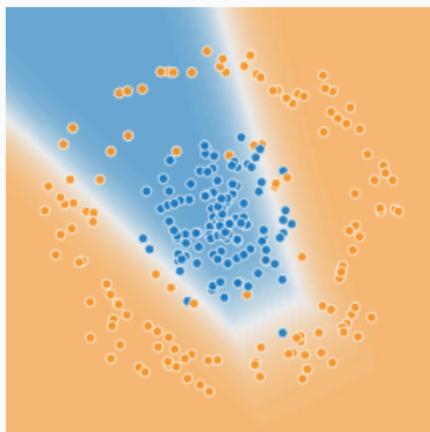
- Choix des hyper-paramètres,
- *Vanishing* et *exploding* gradients,
- Sur-apprentissage ou sous-apprentissage, etc.

## Sous/Sur-apprentissage

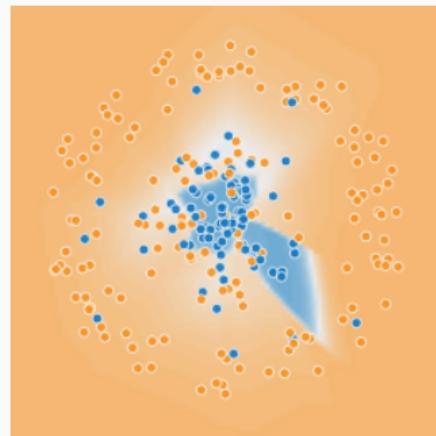
On parle de **sous-apprentissage** (*underfitting*) lorsque le modèle appris explique trop **mal** l'ensemble d'apprentissage.

On parle de **sur-apprentissage** (*overfitting*) lorsque le modèle appris explique à l'inverse trop bien l'ensemble d'apprentissage.

~~> Ce modèle se **généralise** alors **mal** à la population cible.



Sous-apprentissage



Sur-apprentissage

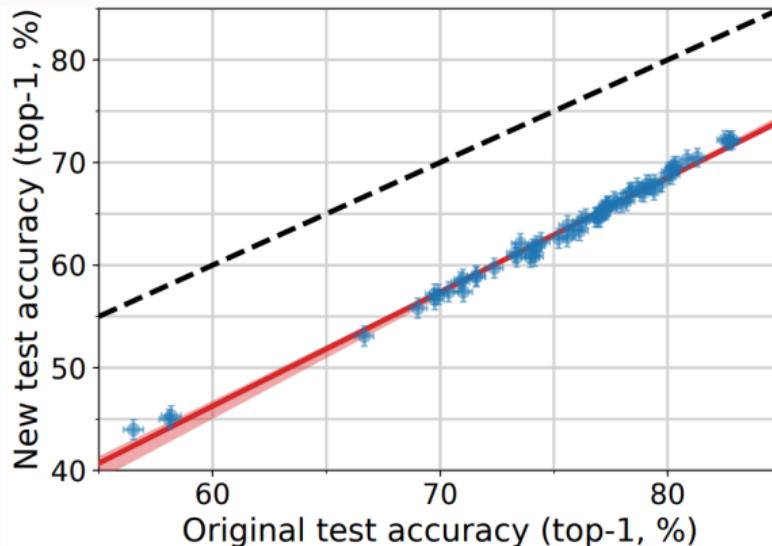
# Sous/Sur-apprentissage

**Solution :** Gérer **trois** ensembles de données distincts :

- **L'ensemble d'apprentissage**, sur lequel on va effectuer la descente de gradient et donc optimiser les **paramètres** du modèle.
- **L'ensemble de validation** : qui va nous fournir une estimation de l'erreur de généralisation, et nous permettre d'optimiser les **hyperparamètres** du modèle (par exemple par *validation-croisée*).
- **L'ensemble de test** : qui détermine la performance objective du réseau de neurones. *On n'utilise JAMAIS cet ensemble lors de la phase d'entraînement !*

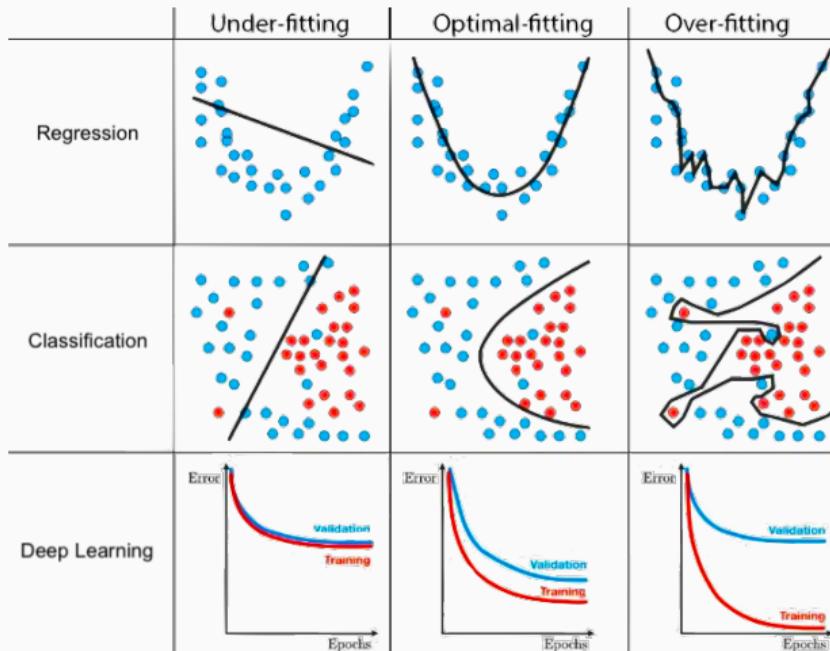


## L'importance de l'ensemble de validation



Les classifieurs sont en moyenne 10 à 15% plus performants sur l'ensemble de test original d'ImageNet que sur un nouvel ensemble généré par la même procédure.

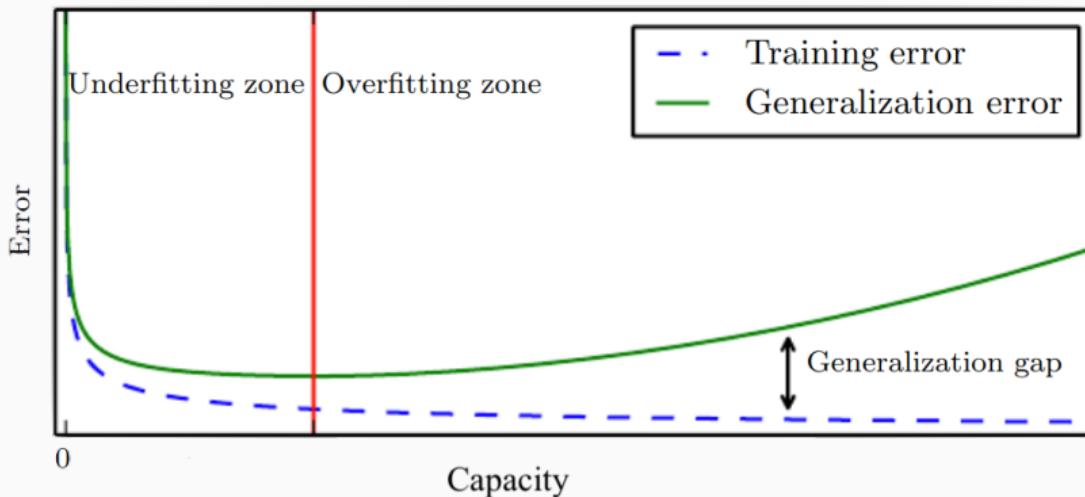
# Sous/Sur-apprentissage : Le problème de la capacité du modèle



## *En pratique :*

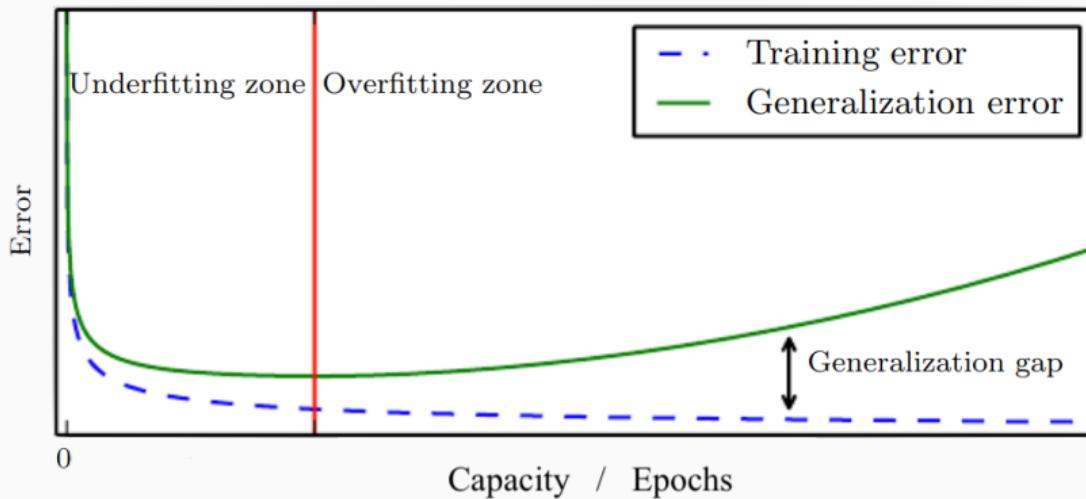
Pour déceler le **sur-apprentissage**, on vérifie que l'erreur sur l'ensemble de validation est comparable à l'erreur sur l'ensemble d'apprentissage.

## Sous/Sur-apprentissage



Un modèle de trop grande capacité (profondeur, nombre de neurones) engendre du sur-apprentissage.

## Sous/Sur-apprentissage



Un modèle de trop large capacité (profondeur, nombre de neurones) engendre du sur-apprentissage.

La courbe ci-dessus est valable pour la capacité du réseau, mais aussi pour la durée d'entraînement !

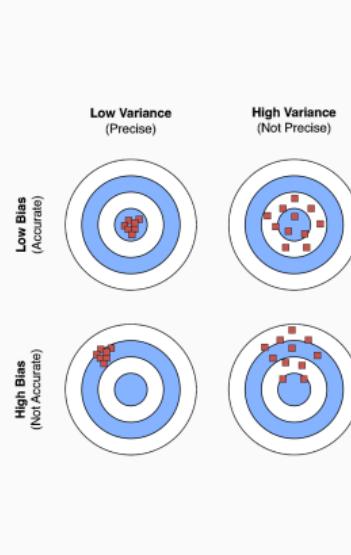
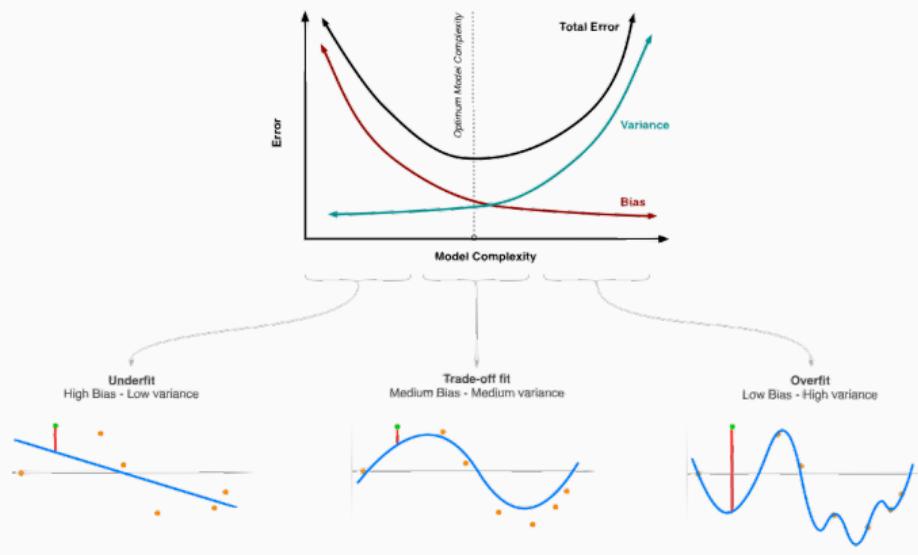
Image de [Goodfellow et al. 2015] Deep Learning

# Compromis Biais-Variance

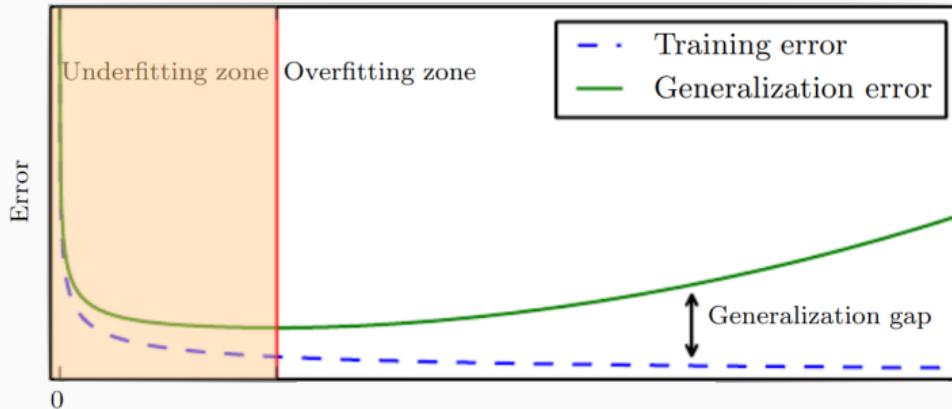
Pour tout estimateur  $\hat{\theta}$  of  $\theta$  : 
$$\mathcal{R}(\hat{\theta}) = \text{Bias}(\hat{\theta}) + \text{Var}(\hat{\theta}) + \text{"noise"}$$

Stratégie pour améliorer la prédiction :

Augmenter légèrement le biais pour diminuer la variance.



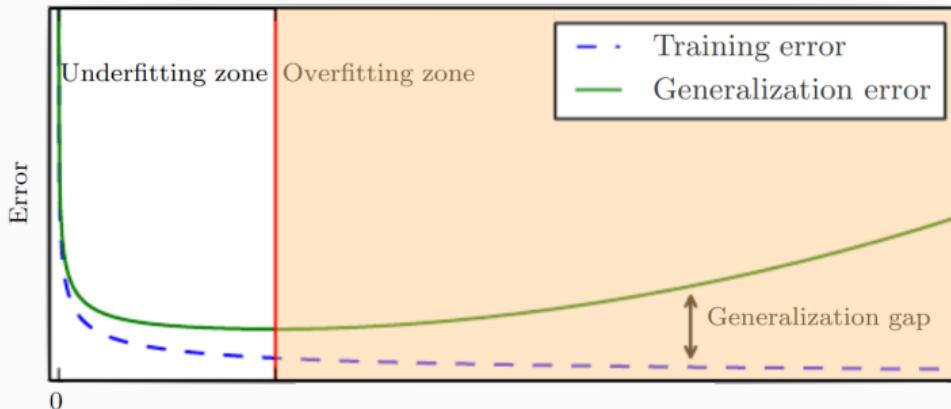
## Que faire en cas de sous-apprentissage ?



*On peut :*

- **Augmenter la capacité du réseau** : augmenter la profondeur, ajouter des neurones dans les couches cachées, changer d'architecture.
- **Améliorer l'entraînement** : augmenter le nombre d'*epochs*, changer le taux d'apprentissage, ajouter du momentum, éventuellement changer d'optimiseur.

## Que faire en cas de sur-apprentissage ?



*On peut :*

- **Diminuer la capacité du réseau** : en pratique cela est déconseillé.
- **Stopper l'entraînement plus tôt**, i.e. diminuer le nombre d'epochs.
- **Régulariser**

# Course Outline

Une brève histoire de l'Intelligence Artificielle

Apprentissage statistique

Régression linéaire et régression logistique

Perceptron monocouche

Réseaux de neurones (perceptron multicouche)

Fonctions d'activation

Optimiseurs

Sous-apprentissage et sur-apprentissage

**Régularisation**

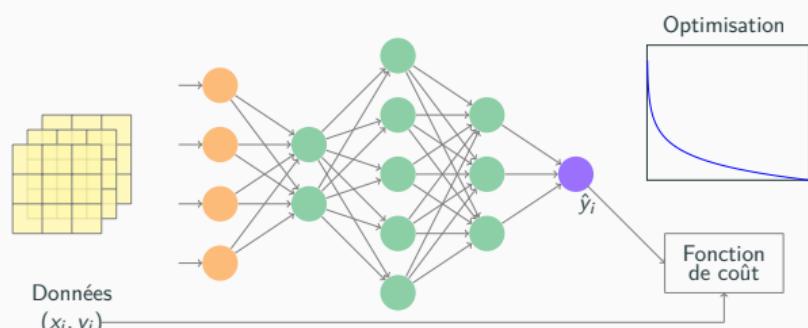
Méthodologie en apprentissage profond

# Régularisation

La régularisation consiste à imposer des **contraintes** sur le **processus d'apprentissage** afin de limiter le sur-apprentissage.

Ces contraintes peuvent s'impliquer à tous les niveaux :

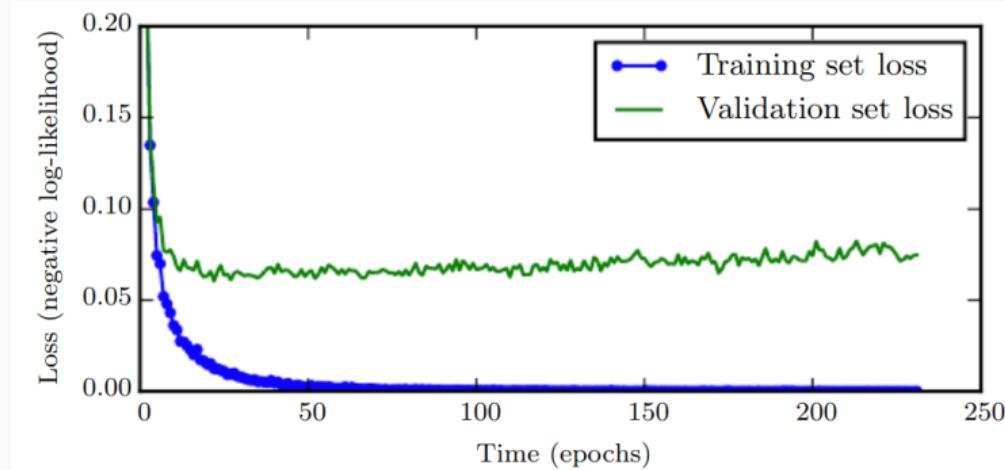
- sur les *données*,
- sur les *paramètres* du réseau,
- dans la *fonction de coût*,
- ou encore dans l'*optimiseur*.



## Arrêt anticipé – *Early stopping*

L'arrêt anticipé est une stratégie de régularisation qui consiste à observer l'erreur commise sur l'ensemble de validation et mettre un terme à l'apprentissage quand cette erreur commence à remonter.

**Remarque :** En pratique, l'erreur sur l'ensemble de validation est bruitée, il faut attendre un peu avant de s'arrêter pour de bon. Dans keras, “early stopping”.



# Régularisation ou Weight decay

**Une solution :** Ajout d'une contrainte sur les paramètres du réseau :

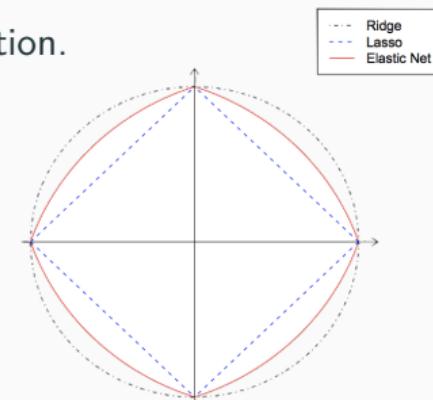
$$\hat{\theta}_{\text{reg}} \in \operatorname{argmin}_{\theta} \mathcal{R}_n(\theta) + \lambda \operatorname{pen}(\theta),$$

avec  $\lambda \in \mathbb{R}^+$  paramètre de réglage, tuning parameter, &  $\operatorname{pen}(\theta) \equiv \|\theta\|_q^q$ .

- $q = 2$  : Régularisation **Ridge** ou  $\mathbb{L}^2$ ,
- $q = 1$  : Régularisation **Lasso** ou  $\mathbb{L}^1$ .

**Choix du paramètre**  $\lambda \in \mathbb{R}^+ \longleftrightarrow$  Force de la pénalisation.

- $\lambda = 0$  : Pas de pénalité,
- $\lim_{\lambda \rightarrow +\infty} \hat{\theta}_{\text{reg}}(\lambda) \rightarrow 0$ ,
- Entre les deux : Trouver un équilibre
  1. Ajuster le vrai modèle,
  2. Réduire/Sélectionner les coefficients.



## Régularisation Ridge ( $q = 2$ )

- Reformulation 1 :

$$\hat{\theta}_{\text{ridge}} \in \operatorname{argmin}_{\theta} \mathcal{R}_n(\theta) + \lambda \|\theta\|_2^2$$

Soit  $\lambda > 0$ .

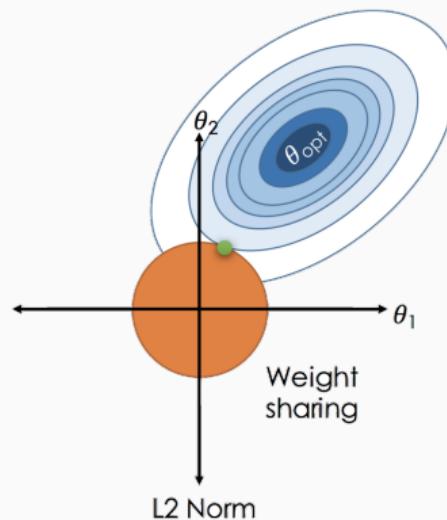
- Reformulation 2 :

$$\hat{\theta}_{\text{ridge}} \in \operatorname{argmin}_{\theta} \mathcal{R}_n(\theta)$$

sous la contrainte

$$\|\theta\|_2^2 \sum_{i=1}^m \theta_i^2 \leq r(\lambda),$$

avec  $r$  bijective.



**Remarque :** L'estimateur ridge est biaisé et variance plus petite que  $\hat{\theta}$ . De plus, ceci est d'autant plus vrai que  $\lambda$  est grand.

## Régularisation Lasso ( $q = 1$ )

- Reformulation 1 :

$$\hat{\theta}_{\text{lasso}} \in \operatorname{argmin}_{\theta} \mathcal{R}_n(\theta) + \lambda \|\theta\|_1$$

Soit  $\lambda > 0$ .

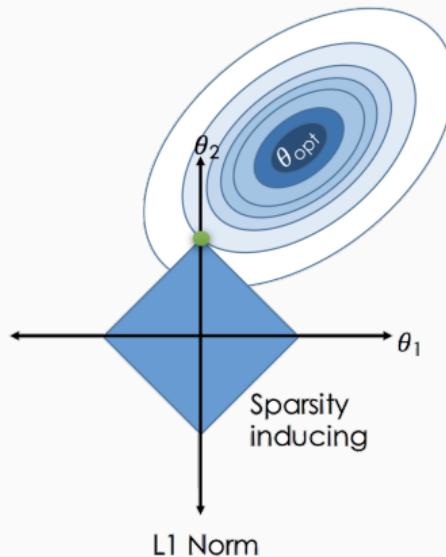
- Reformulation 2 :

$$\hat{\theta}_{\text{lasso}} \in \operatorname{argmin}_{\theta} \mathcal{R}_n(\theta)$$

sous la contrainte

$$\|\theta\|_1 = \sum_{i=1}^m |\theta_i| \leq r(\lambda),$$

avec  $r$  bijective.



**Remarque :** L'estimateur Lasso permet l'annulation de certains coefficients de  $\theta$ .

~~~ Sélection de variable / Estimateur parcimonieux, ou sparse.

Régularisation Ridge vs. Lasso

Aucune méthode n'est inconditionnellement meilleure que l'autre :

- **Lasso** : Petit nombre de paramètres significatifs et les autres sont proches de zéro,
i.e. seuls quelques prédicteurs influencent la réponse.
- **Ridge** : Plusieurs grands paramètres ayant à peu près la même valeur,
i.e. la plupart des prédicteurs ont un impact sur la réponse.

Régularisation Ridge vs. Lasso \rightsquigarrow Elastic-Net

Aucune méthode n'est inconditionnellement meilleure que l'autre :

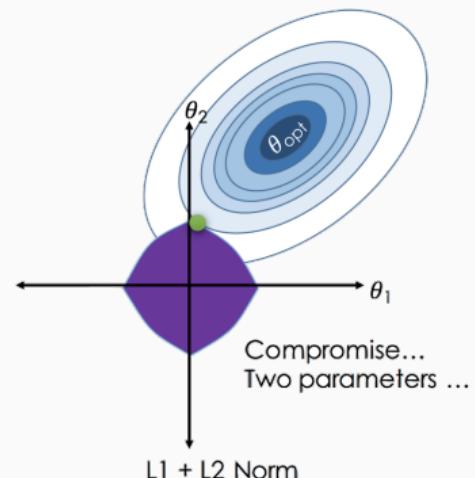
- **Lasso** : Petit nombre de paramètres significatifs et les autres sont proches de zéro,
i.e. seuls quelques prédicteurs influencent la réponse.
- **Ridge** : Plusieurs grands paramètres ayant à peu près la même valeur,
i.e. la plupart des prédicteurs ont un impact sur la réponse.

\rightsquigarrow Combiner Ridge et Lasso :

Régularisation **Elastic-Net** (*filet élastique*).

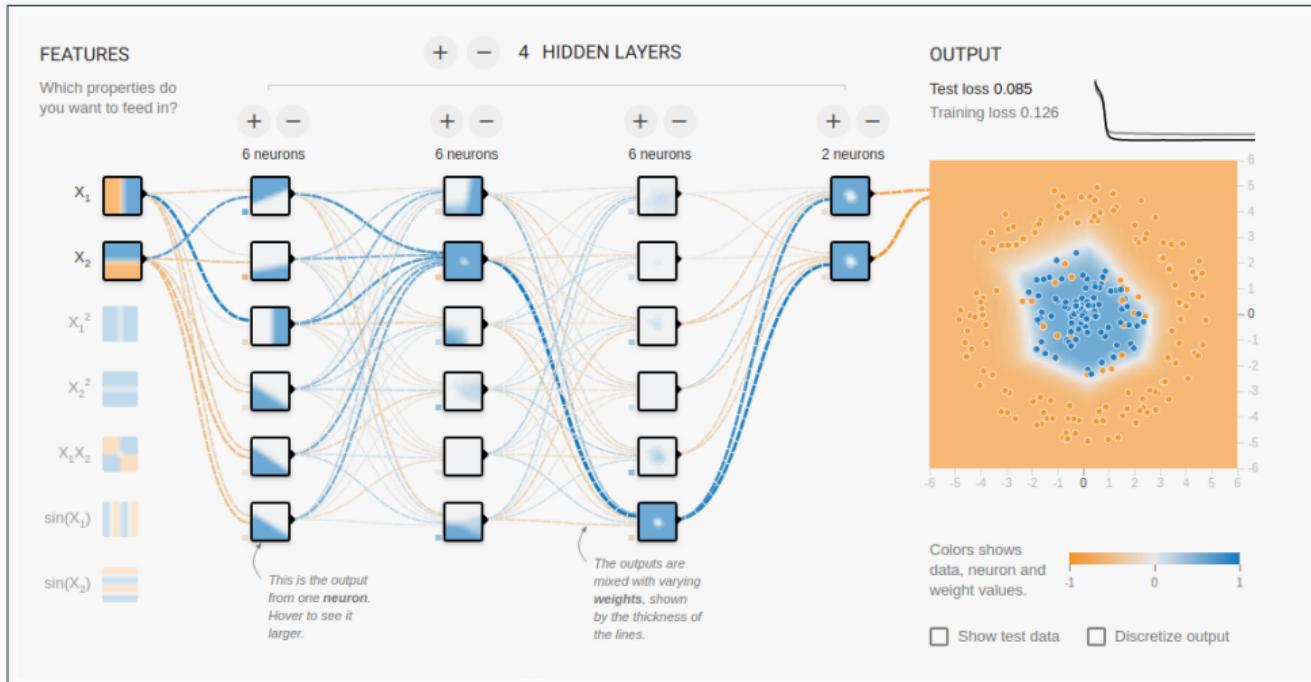
Soit $\lambda > 0$ et $\alpha \in]0, 1[$.

$$\hat{\theta}_{\text{Elastic-Net}} \in \operatorname{argmin}_{\theta} \mathcal{R}_n + \lambda (\alpha \|\theta\|_1 + (1 - \alpha) \|\theta\|_2^2)$$



Dropout

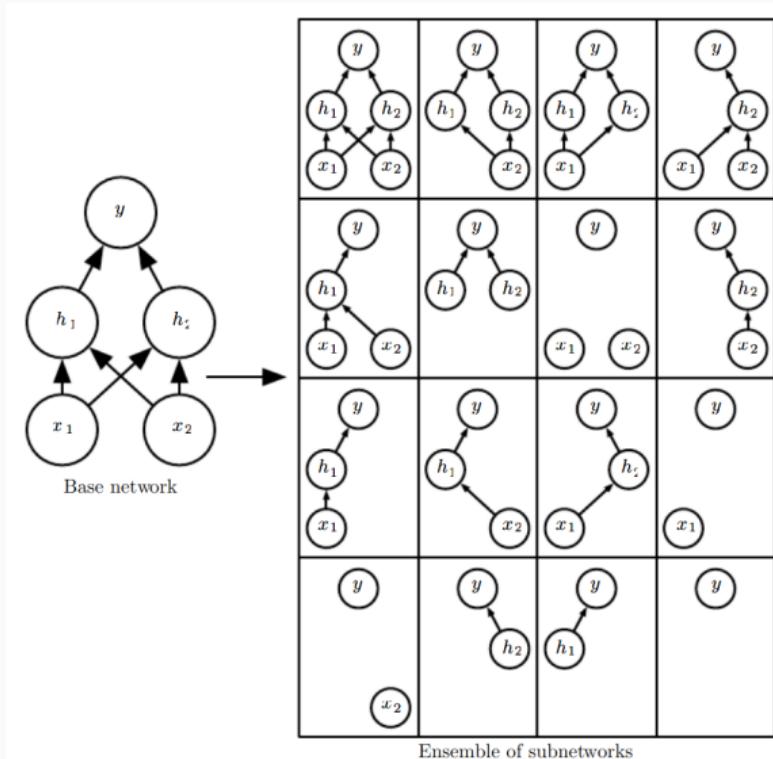
Une des causes de sur-apprentissage est l'émergence, pendant l'entraînement, de chemins préférentiels dans le réseau de neurones.



Dropout

Pour éviter ce phénomène, on peut aléatoirement “éteindre” (déconnecter) des neurones du réseau pendant l’étape de prédiction. Cela permet de favoriser la **redondance** et l'**exhaustivité** des prédictions.

Si un réseau apprend à détecter un visage parce qu'un neurone particulier détecte très bien les nez, il faut enseigner au réseau à rechercher d'autres composants caractéristiques du visage au cas où l'information du nez serait moins présente dans certaines images.



Augmentation de base de données

Utiliser des ensembles de données de taille réduite peut induire un sur-apprentissage.

→ *Augmentation artificielle de la taille de la base de données* en les altérant avec des **transformations contrôlées**.

Cette pratique est particulièrement utile en *traitement d'images* : un réseau de neurones ne sait pas reconnaître des formes dans une image qui sont transformées par translation, rotation ou changement d'échelle.



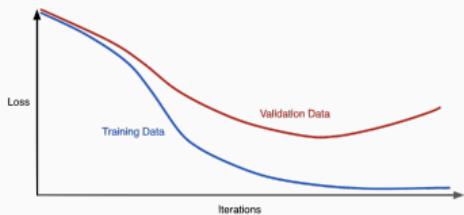
Image de <https://github.com/aleju/imgaug>

Bagging

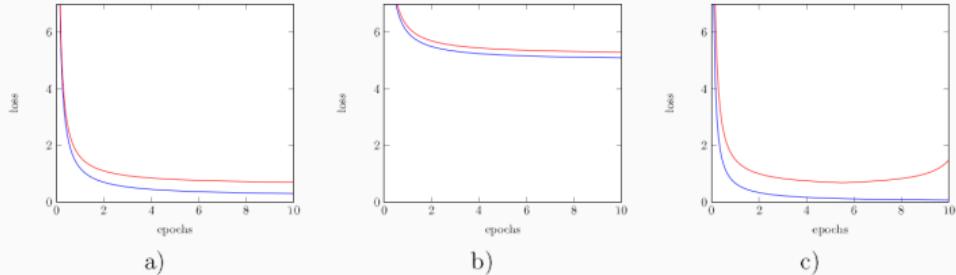
Pour réduire le risque espéré, on peut entraîner plusieurs modèles (formes de réseau) différents, et les faire voter pour dégager la prédiction la plus populaire.

L'intuition derrière cette méthode est que les différents modèles se tromperont à différentes reprises...

Régularisation



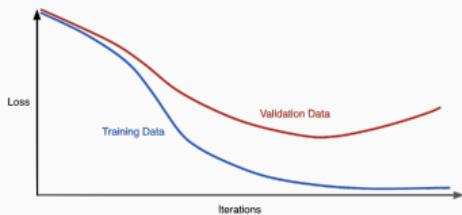
Q3 : D'après vous, pour améliorer les résultats, vaudrait-il mieux augmenter ou diminuer la régularisation ?



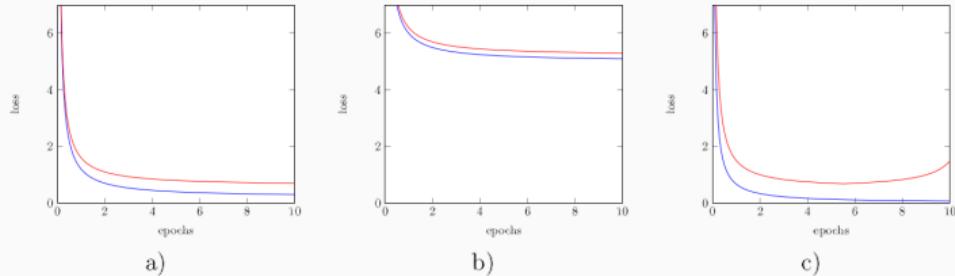
Q4 : Classez ces courbes par ordre croissant du taux de régularisation utilisé :

1. a - b - c
2. a - c - b
3. b - a - c
4. b - c - a
5. c - a - b
6. c - b - a

Régularisation



Q3 : D'après vous, pour améliorer les résultats, vaudrait-il mieux **augmenter** ou diminuer la régularisation ?



Q4 : Classez ces courbes par ordre croissant du taux de régularisation utilisé :

1. a - b - c
2. a - c - b
3. **b - a - c**
4. b - c - a
5. c - a - b
6. c - b - a

Course Outline

Une brève histoire de l'Intelligence Artificielle

Apprentissage statistique

Régression linéaire et régression logistique

Perceptron monocouche

Réseaux de neurones (perceptron multicouche)

Fonctions d'activation

Optimiseurs

Sous-apprentissage et sur-apprentissage

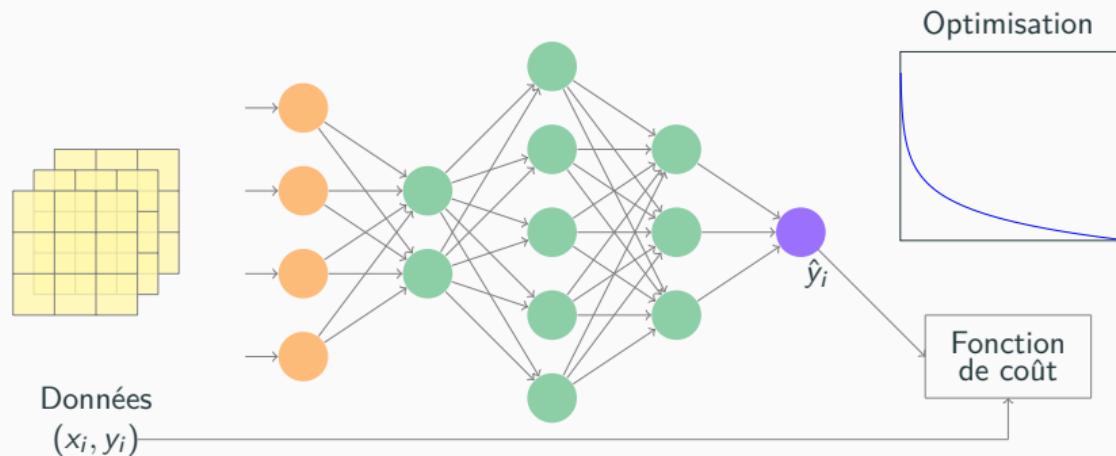
Régularisation

Méthodologie en apprentissage profond

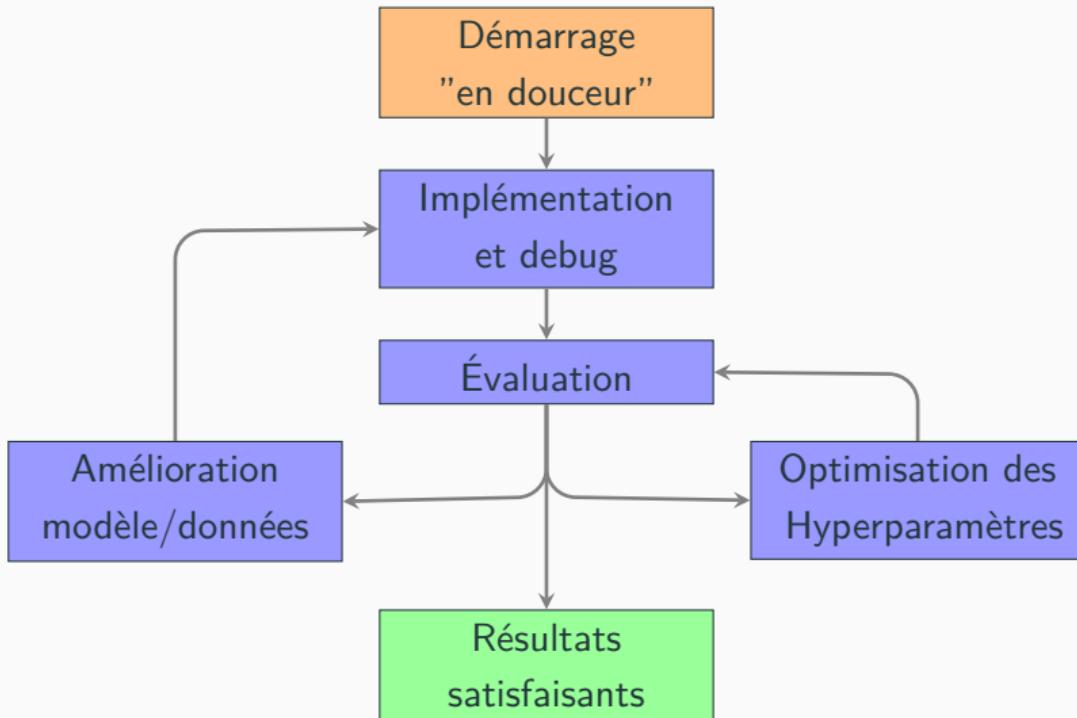
Difficulté de l'apprentissage profond

Vous connaissez maintenant tous les éléments constitutifs d'un algorithme basé sur l'apprentissage profond.

Comment s'y retrouver parmi toutes les combinaisons possibles d'architectures, d'hyperparamètres, de type de régularisation, etc.? Tout n'est pas toujours nécessaire, ni désirable, ni aussi efficace en fonction des problèmes, des données, et du contexte.



Méthodologie générale



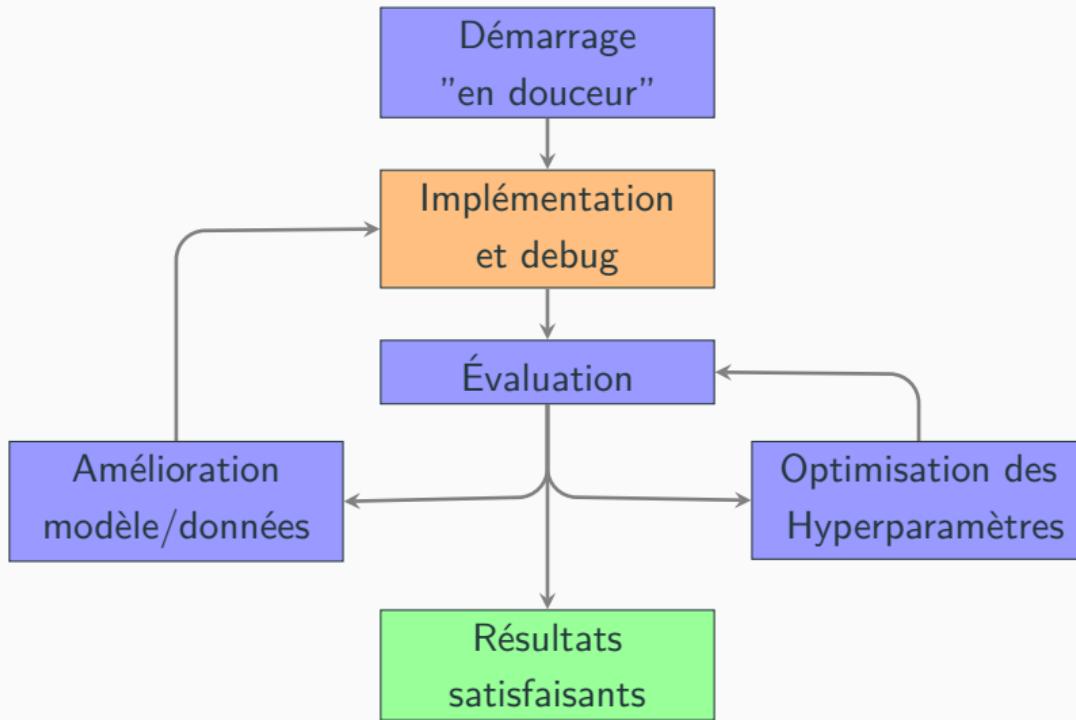
Démarrer simplement

Pour commencer sur un nouveau problème, l'objectif est de mettre au point une **première version** simpliste, mais qui fonctionne, à partir de laquelle on va itérer.

Pour cela :

- Choisir une **architecture simple**,
Perceptron multi-couche, LeNet, LSTM,
- Utiliser des **paramètres par défauts** efficaces,
Adam, activation `ReLU`, pas de régularisation,
- **Normalisation** des données,
Centrer-réduire, ramener entre 0 et 1,
- Éventuellement **simplifier** le problème,
Ensemble d'apprentissage réduit, diminuer le nombre de classes, la dimension des images.

Méthodologie générale



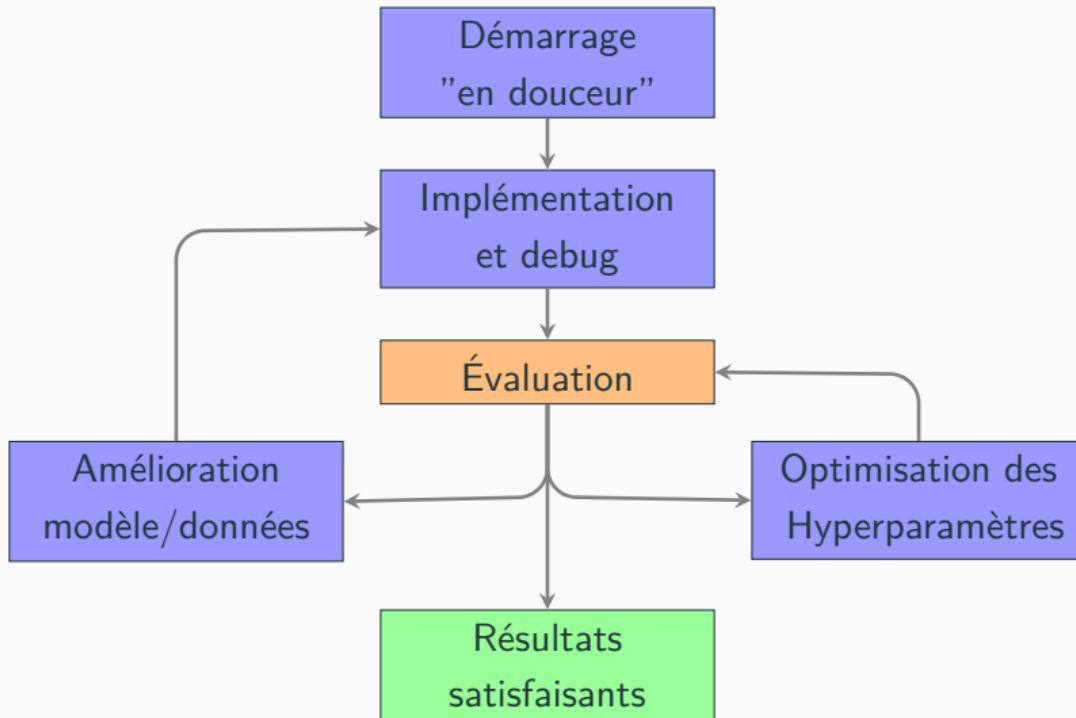
Débugger

Quelques erreurs et problèmes courants :

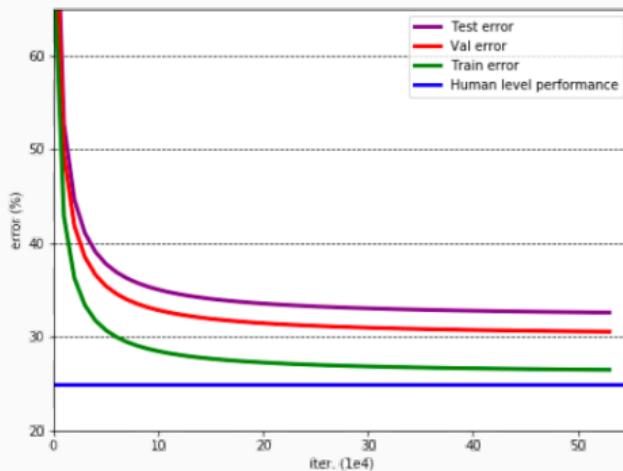
- Taille ou type des tenseurs incorrects,
- Pré-traitement des données oublié ou fait plusieurs fois,
- Sortie non adaptée à la fonction de perte,
- Instabilité numérique (NaN, Inf) liée aux exponentielles, logarithmes, divisions, etc.,
- Déficit de RAM GPU.

Conseil important : commencer par essayer de surapprendre un *batch* de données (voire une seule donnée)

Méthodologie générale



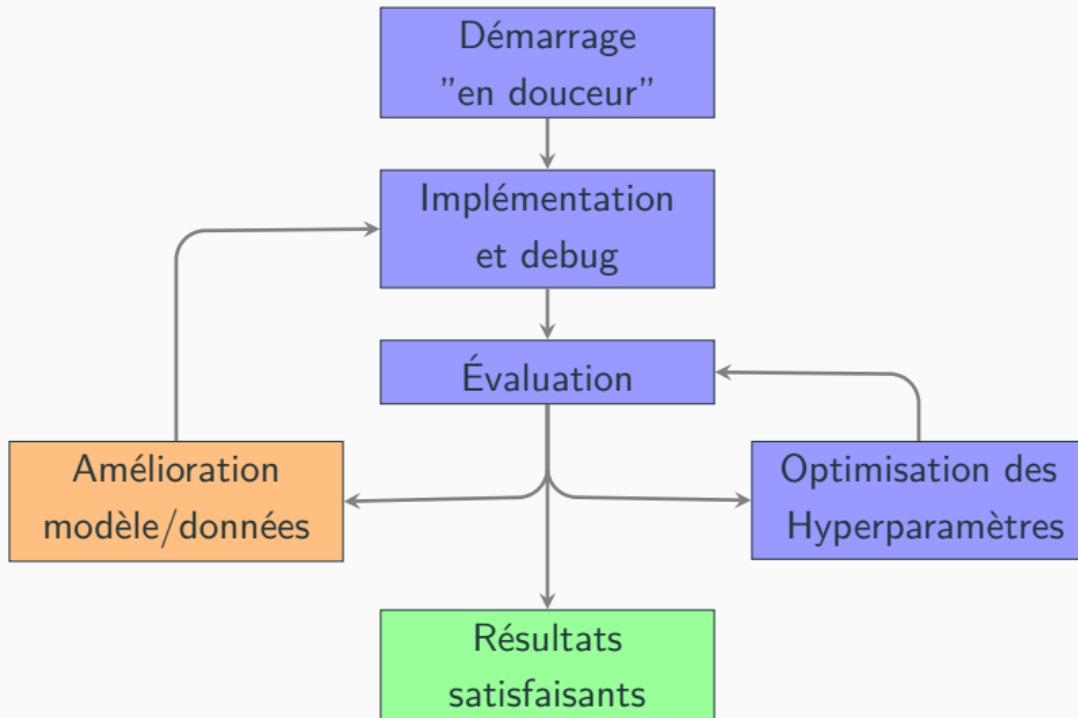
Évaluer la source des erreurs



L'erreur sur l'ensemble de test peut provenir de différentes sources :

- **Erreur irréductible** : meilleure performance objectivement atteignable
- **Sous-apprentissage**
- **Sur-apprentissage**
- **Sur-apprentissage de l'ensemble de validation**

Méthodologie générale



Améliorer le modèle et/ou les données

Pour corriger les problèmes révélés lors de l'**évaluation** on va privilégier l'ordre suivant :

1. Correction du **sous-apprentissage**

- *Augmentation de la taille du modèle,*
- *Changement d'architecture,*
- *Réduction de la régularisation,*

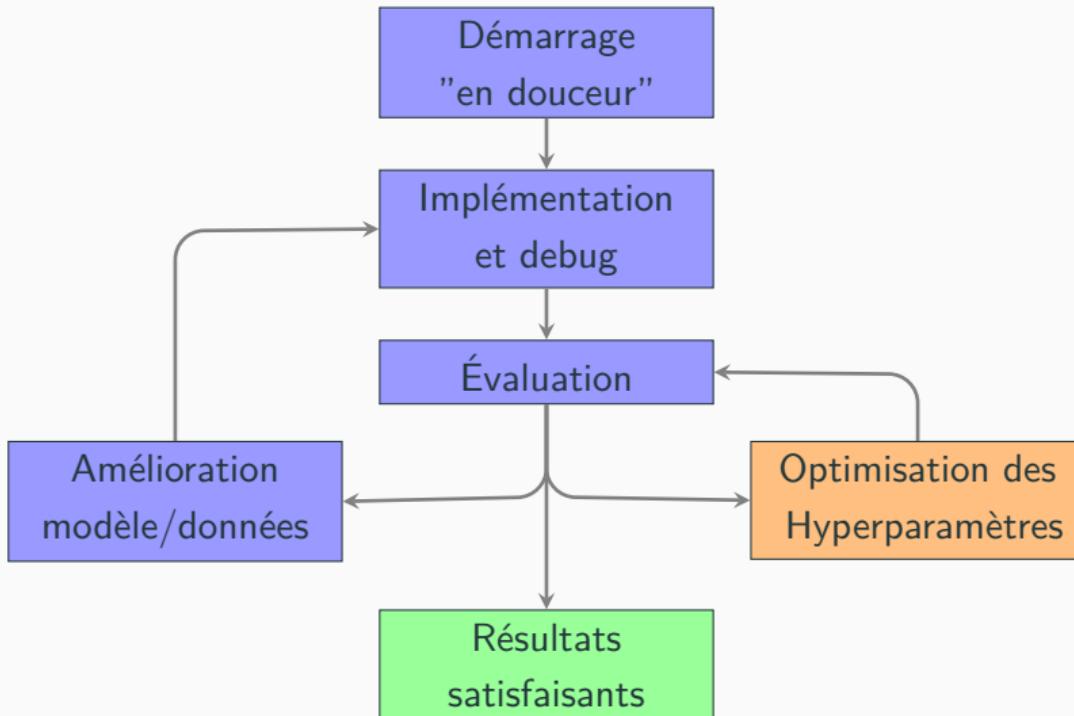
2. Correction du **sur-apprentissage**

- *Ajout de données (quand c'est possible !),*
- *Augmentation des données,*
- *Régularisation,*

3. Correction des problèmes de **bases de donnée**

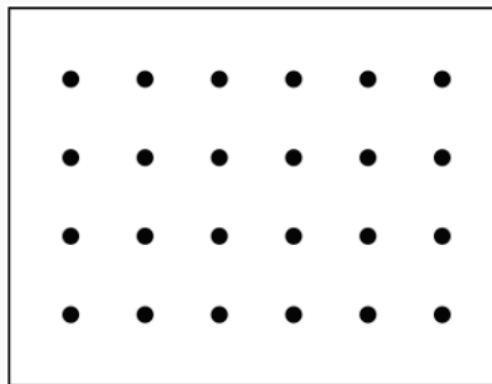
- *Déséquilibre de classes,*
- *Distributions train/val/test différentes.*

Méthodologie générale



Hyperparamètres

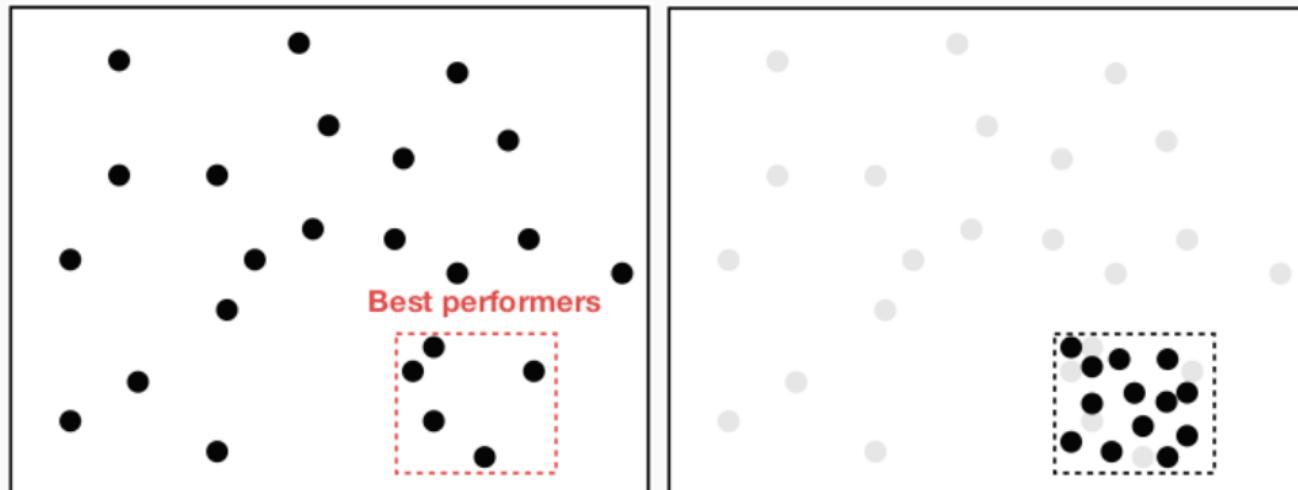
Pour mettre au point les valeurs optimales d'**hyperparamètres** (taux d'apprentissage, momentum, nombre de neurones par couche, etc.), une pratique commune est de déterminer un ensemble de valeurs possibles pour chaque hyperparamètre et de tester toutes les combinaisons. On conserve la combinaison qui minimise l'erreur sur l'ensemble de validation.



Chaque point correspond à l'entraînement complet d'un réseau de neurones
(peut prendre plusieurs heures/jours/semaines...)

Hyperparamètres

Une variante plus couramment utilisée est d'implémenter une recherche aléatoire *coarse-to-fine*.



Chaque point correspond à l'entraînement complet d'un réseau de neurones
(peut prendre plusieurs heures/jours/semaines...)