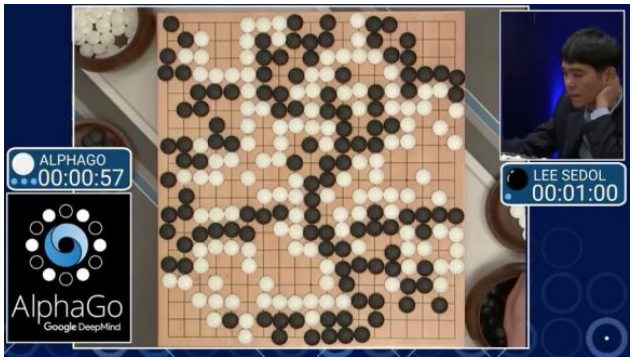
Introduction à l'apprentissage et aux réseaux de neurones

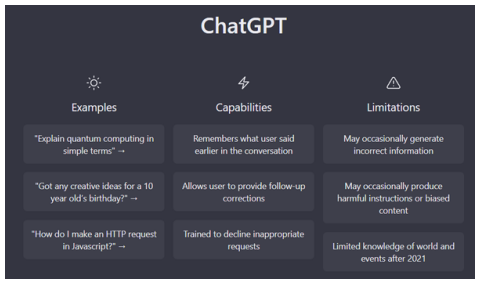
A. Carlier

2025

Intelligence Artificielle ?



DeepL



Aperçu du cours

- Réseaux de neurones artificiels
  - Régression linéaire et logistique
  - Neurones artificiels et perceptron monocouche
  - Perceptron multicouche
  - Apprentissage profond
- Réseaux de neurones convolutifs
  - Couches de convolution
  - Architectures convolutives pour la classification d'image
  - Transfert d'apprentissage
- Réseaux de neurones récurrents
  - Problèmes séquentiels
  - Neurone récurrent
  - Modèles à porte
- Transformers et LLM ?

Plan du cours

Une définition de l'apprentissage

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

Arthur Samuel (1959)

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

Tom Mitchell (1998)

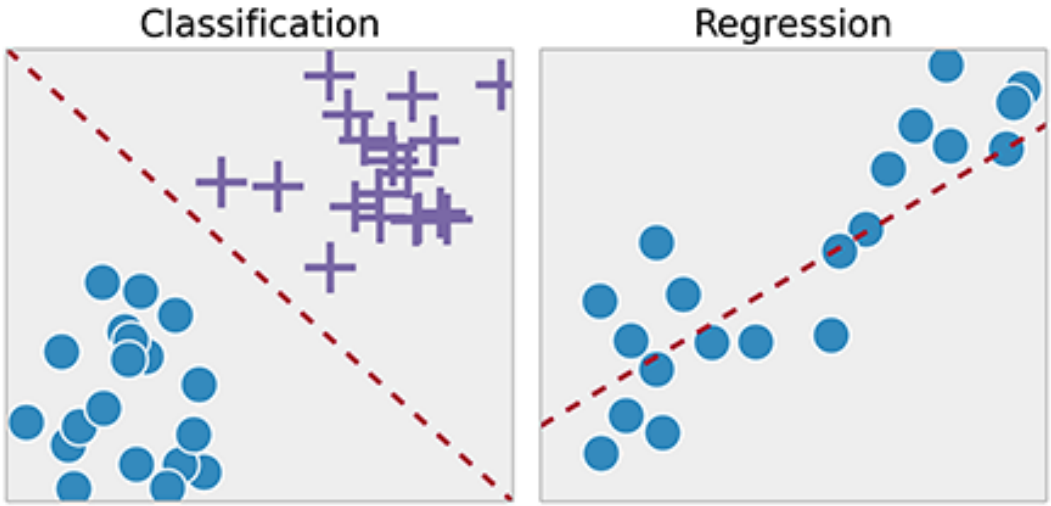
3 grands paradigmes d'apprentissage

Apprentissage supervisé

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

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

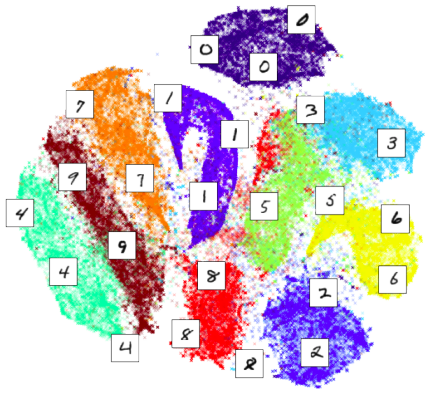
- Recherche d'un prédicteur qui minimise la différence entre labels réels et prédits
- Souvent coûteux car nécessite l'annotation de larges bases de données



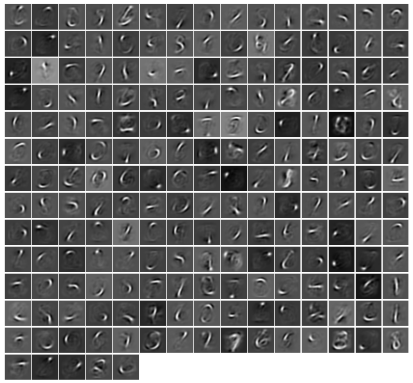
3 grands paradigmes d'apprentissage

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

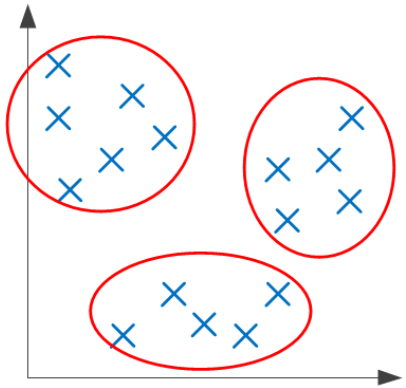
$$\mathcal{D} = \{\mathbf{x}^{(1)}, .., \mathbf{x}^{(m)}\}.$$



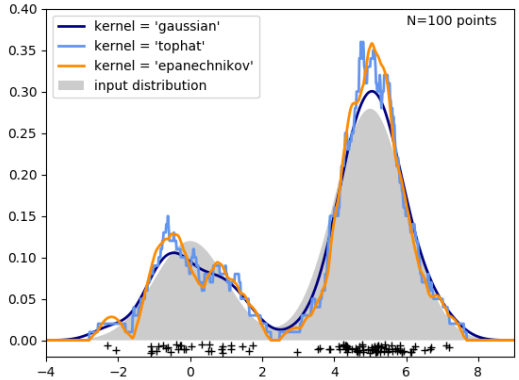
Réduction de dimension



Extraction de caractéristiques



Clustering

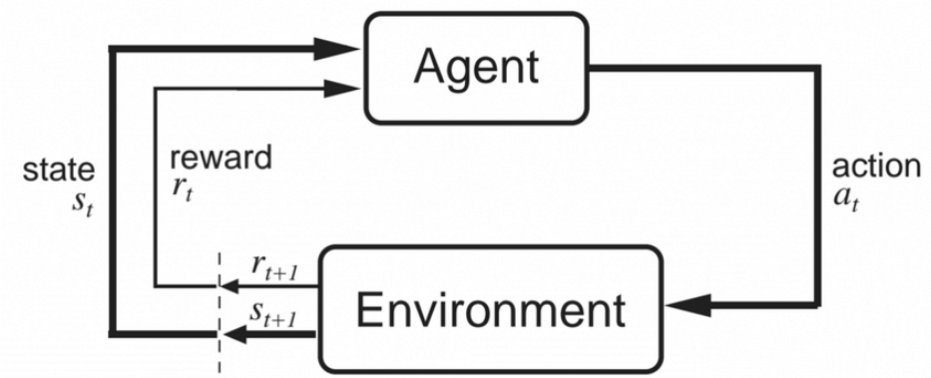


Estimation de densité

3 grands paradigmes d'apprentissage

Apprentissage par renforcement

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



Mais aussi...

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

Plan du cours

Apprentissage supervisé

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

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

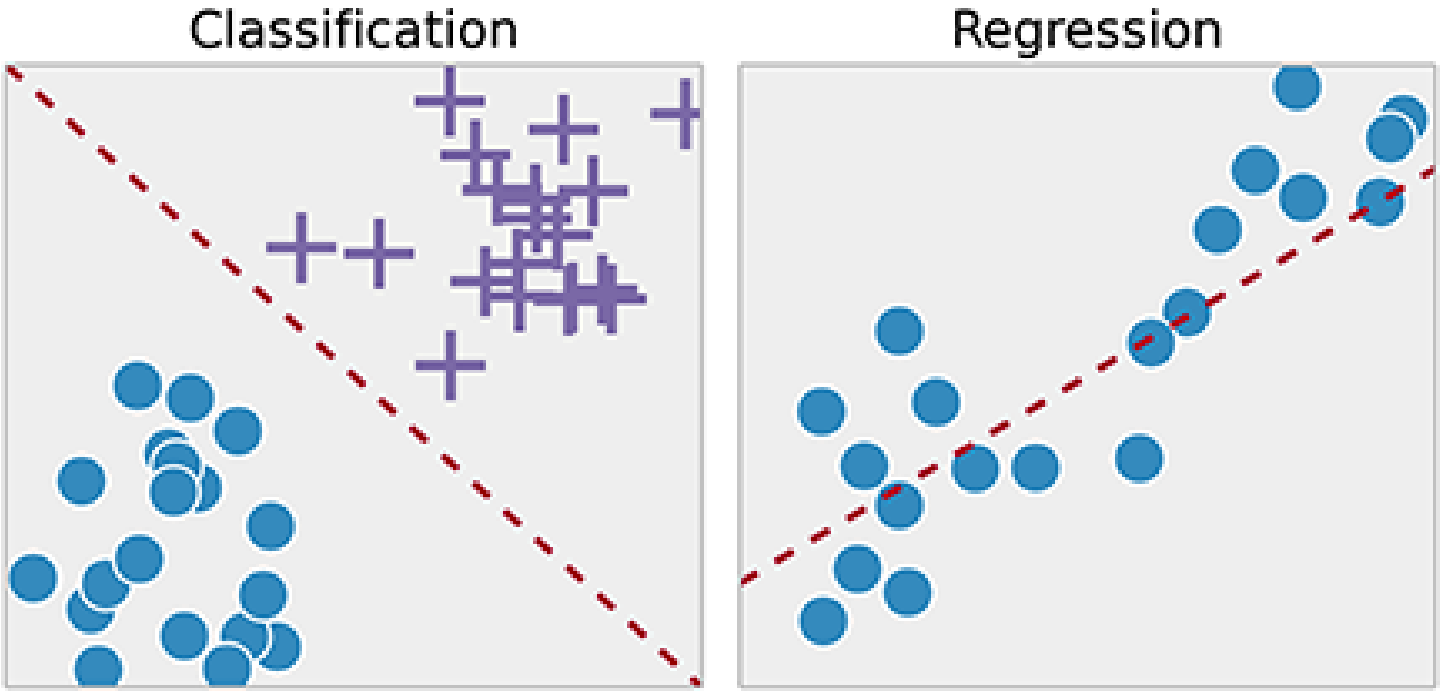
Les labels permettent d'enseigner à l'algorithme à établir des correspondances entre les observations et les labels.

Apprentissage supervisé

Il existe deux principaux types d'apprentissage supervisé :

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

Classification et Régression



$\Rightarrow$ Régression

Régression linéaire

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

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

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

et la fonction de coût quadratique s'écrit :

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

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

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

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

Régression linéaire

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

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

Régression aux moindres carrés

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

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

Régression linéaire

Résolution des problèmes aux moindres carrés :

Condition Nécessaire du premier ordre :

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

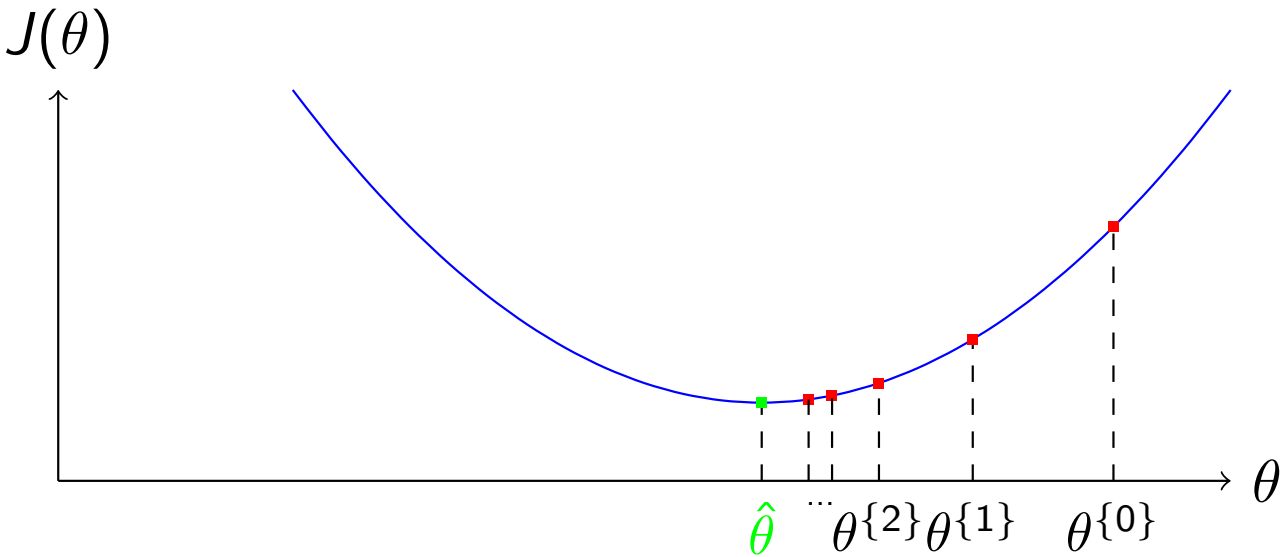
Si $det(\mathrm{X}^T\mathrm{X}) \neq 0$, la solution analytique est :

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

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

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

Régression linéaire



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

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

TANT QUE pas convergence **FAIRE**

  POUR j de 1 à d **FAIRE**

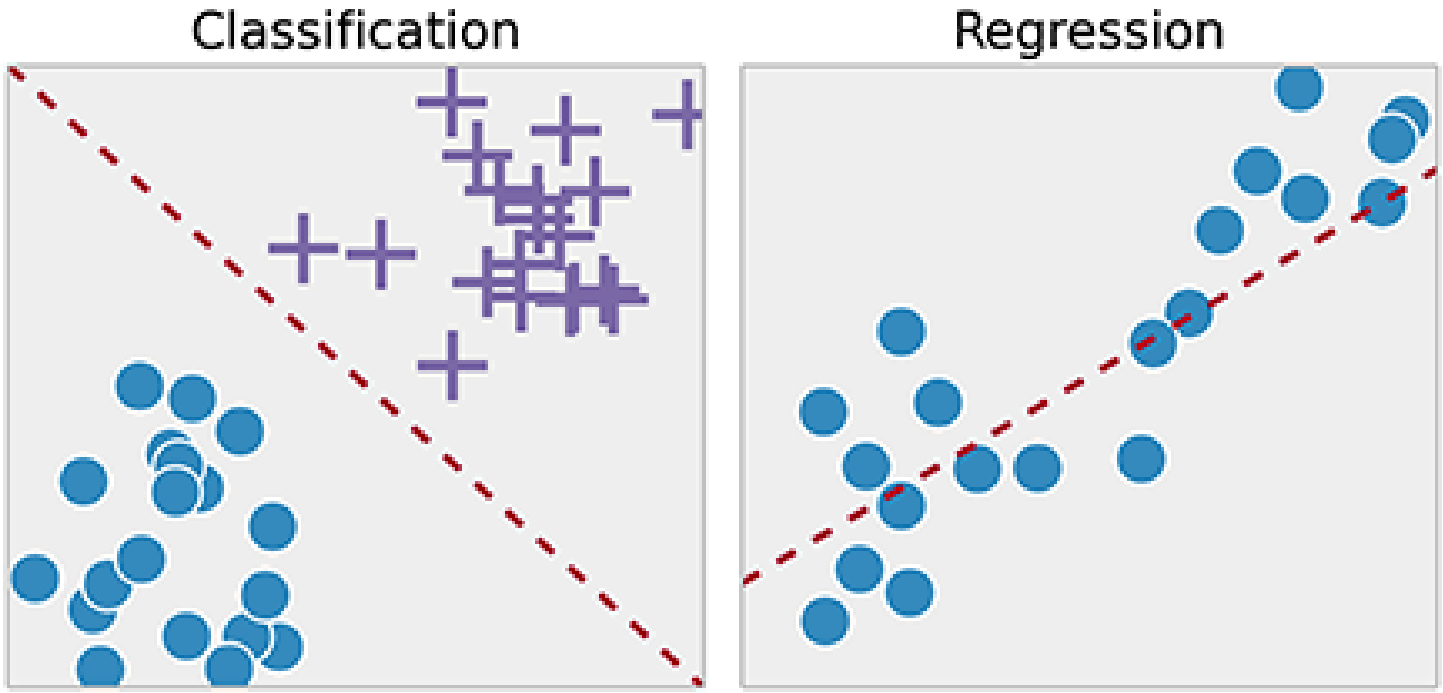
$$\theta_j^{\{k+1\}} \leftarrow \theta_j^{\{k\}} - \alpha \frac{\partial J(\theta^{\{k\}})}{\partial \theta_j}$$

  FIN POUR

  $k \leftarrow k + 1$

FIN TANT QUE

Classification et Régression



$\Rightarrow$ Classification

Régression logistique

En sortie, la sortie du modèle peut être :

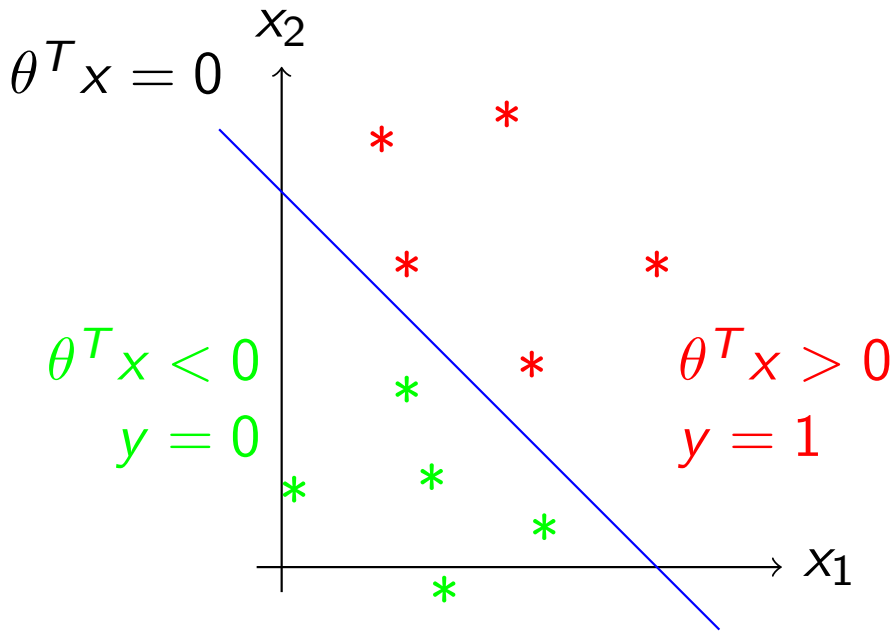
- **binaire** : échec/succès, 0/1, -/+
  $\Rightarrow$ fonction **sigmoïde ou logistique**
- **multinomiale** (multi-classes) : chiffres
  $\Rightarrow$ fonction **softmax**

Régression logistique : cas binaire

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

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

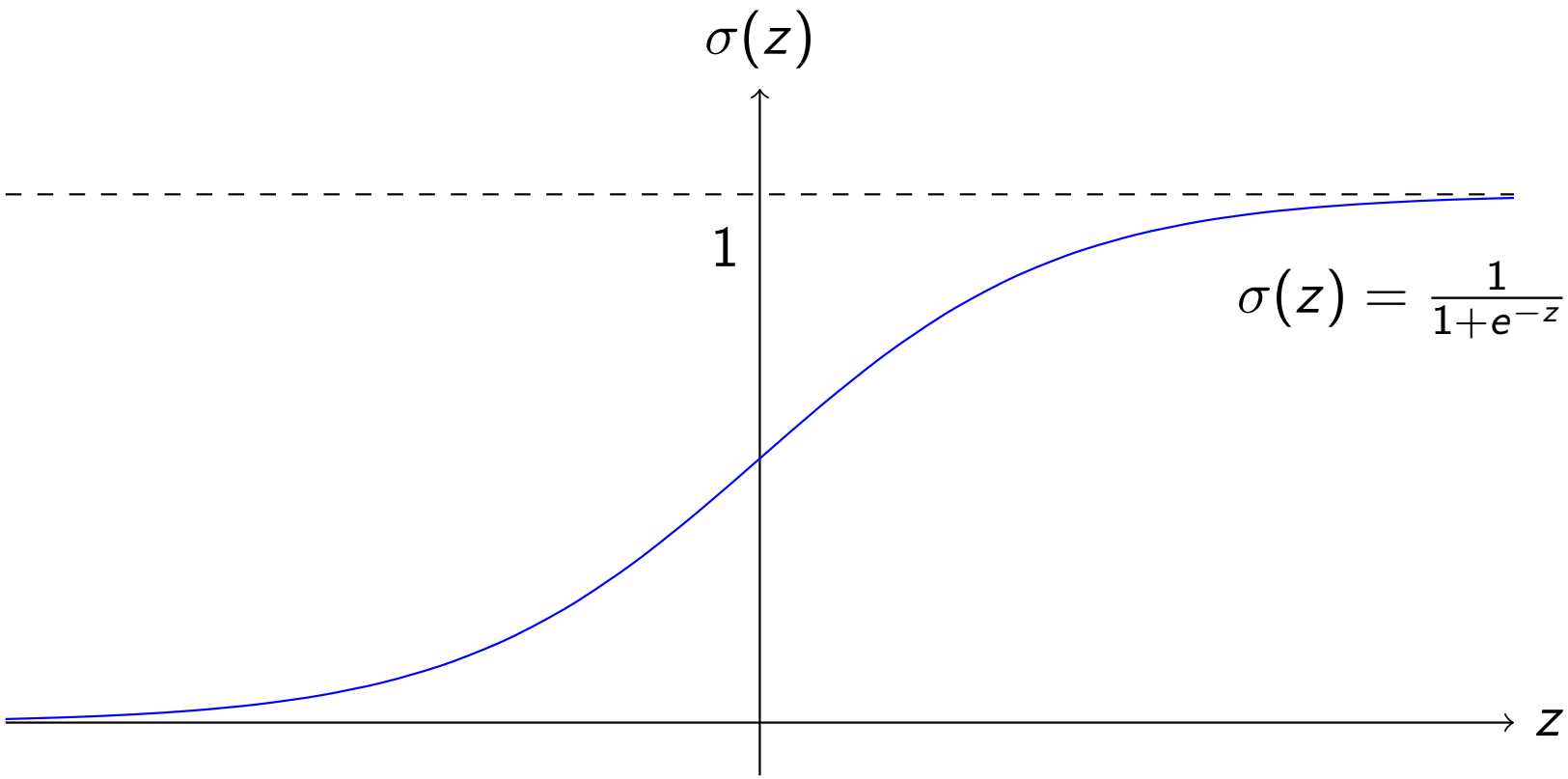
Ce modèle linéaire agit comme **séparateur** des 2 classes : $y \in \{0, 1\}$



Régression logistique : cas binaire

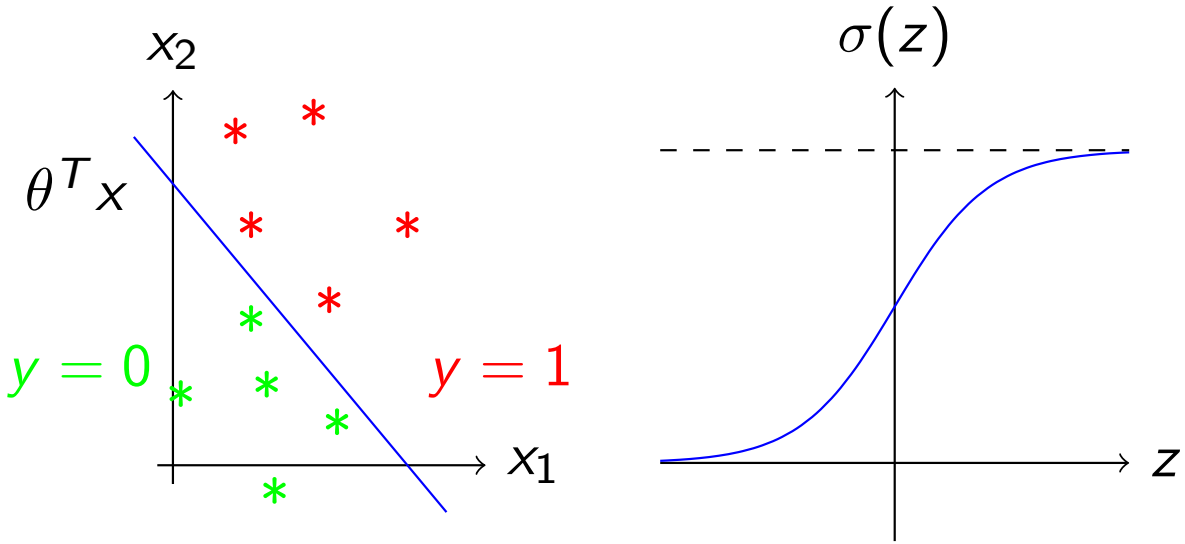
La **sigmoïde** ou **fonction logistique** définie par :

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



Régression logistique : cas binaire

Cas 2D où les données sont linéairement séparables



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

- on prédit $y = 1$ si $-3 + x_1 + x_2 > 0 \Leftrightarrow \sigma(\theta^T x) > 0.5$
- on prédit $y = 0$ si $x_1 + x_2 \leq 3 \Leftrightarrow \sigma(\theta^T x) < 0.5$

Le modèle h_θ fournit une prédiction homogène à une probabilité !

Régression logistique : cas binaire

Comment estimer automatiquement θ ?

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

$$x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_d^{(i)} \end{bmatrix} \in \mathbb{R}^d \text{ et } y^{(i)} \in \{0,1\}, \forall i \in \{1,..,m\}$$

et le **modèle** est défini par la fonction sigmoïde appliquée à l'équation de l'hyperplan séparateur :

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

Régression logistique : cas binaire

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

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

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

avec

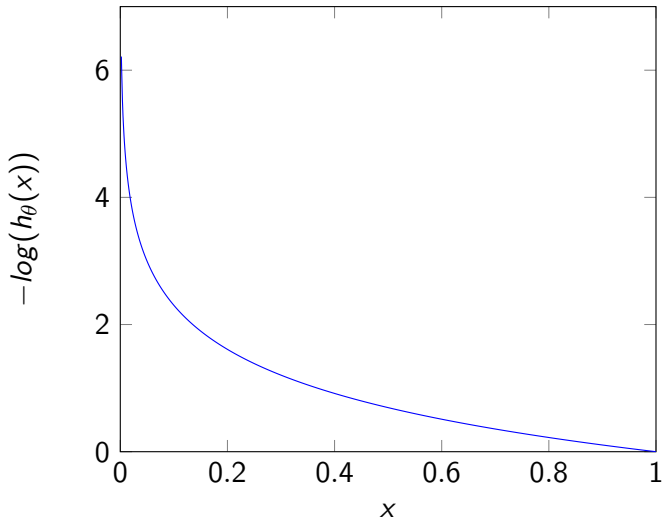
$$perte(h_\theta(x^{(i)}), y^{(i)}) = \frac{1}{2}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 = \frac{1}{2}\left(\frac{1}{1+e^{-\theta^T x^{(i)}}} - y^{(i)}\right)^2$$

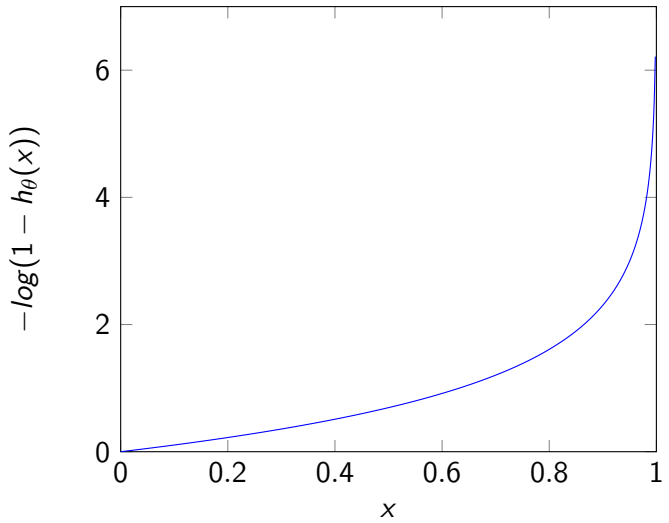
Cette fonction n'est **pas convexe** donc la descente de gradient ne garantit pas un minimum global !

Régression logistique : cas binaire

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

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





En faisant ce choix de fonction de coût, la fonction $J(\theta)$ **est convexe** !

Régression logistique : cas binaire

Sur les m exemples, la fonction de perte devient :

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

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

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

Précision sur le calcul précédent

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

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

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

Régression logistique : cas multiclasse

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

On utilise la **fonction softmax** :

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

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

Régression logistique : cas multiclasse

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

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

Régression logistique : cas multiclasse

La fonction objectif devient :

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

Le gradient s'écrit :

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

Plan du cours

Perceptron monocouche

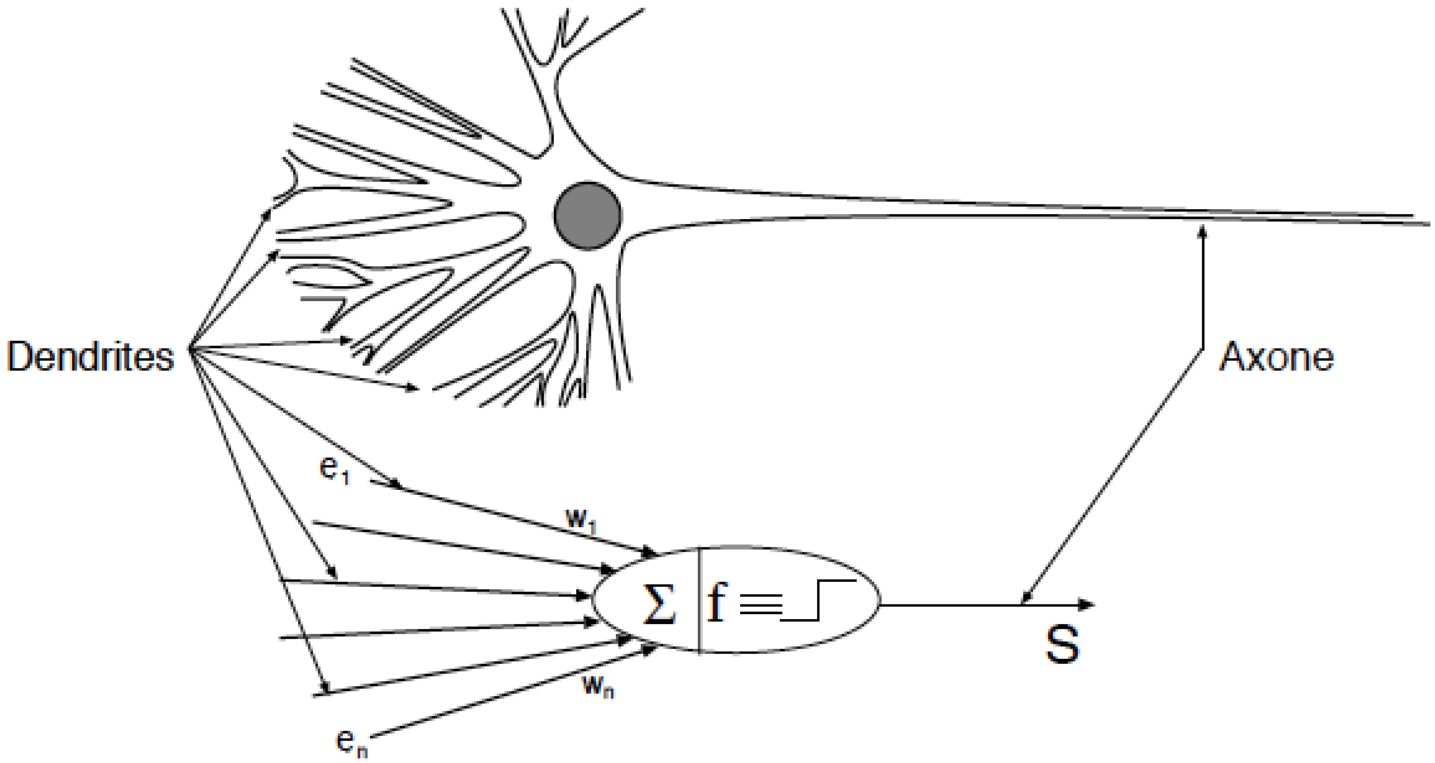
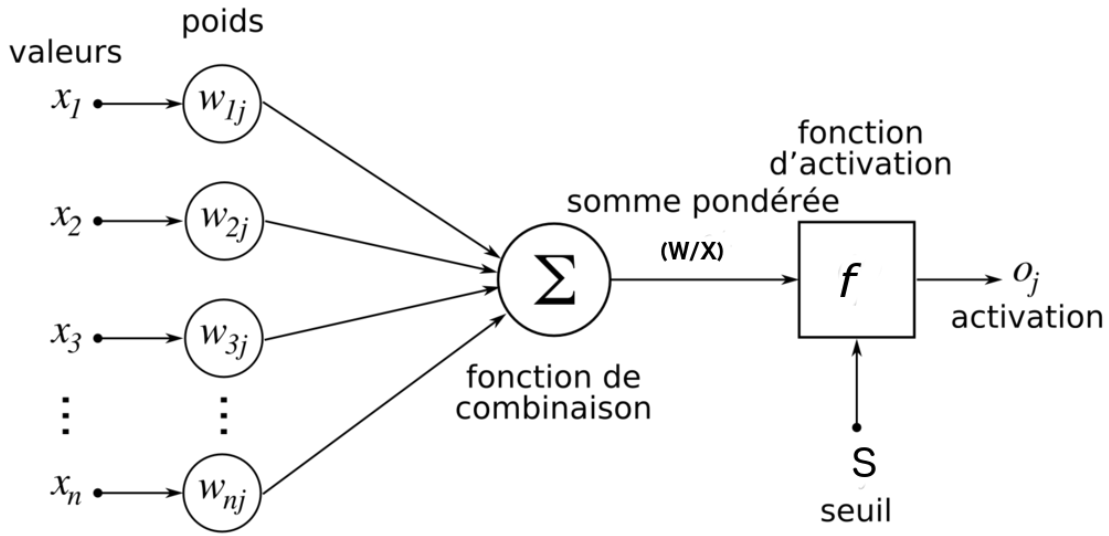


Figure – Structure d'un neurone biologique/artificiel

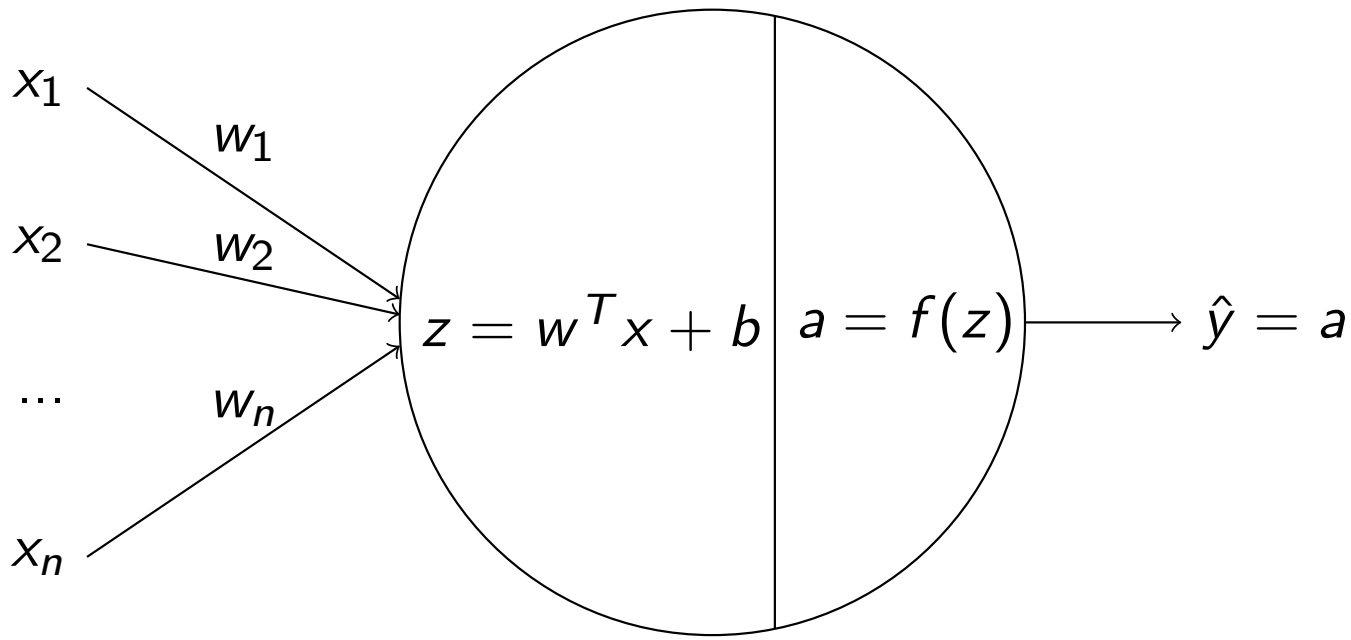
Perceptron monocouche : Fonctionnement



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

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

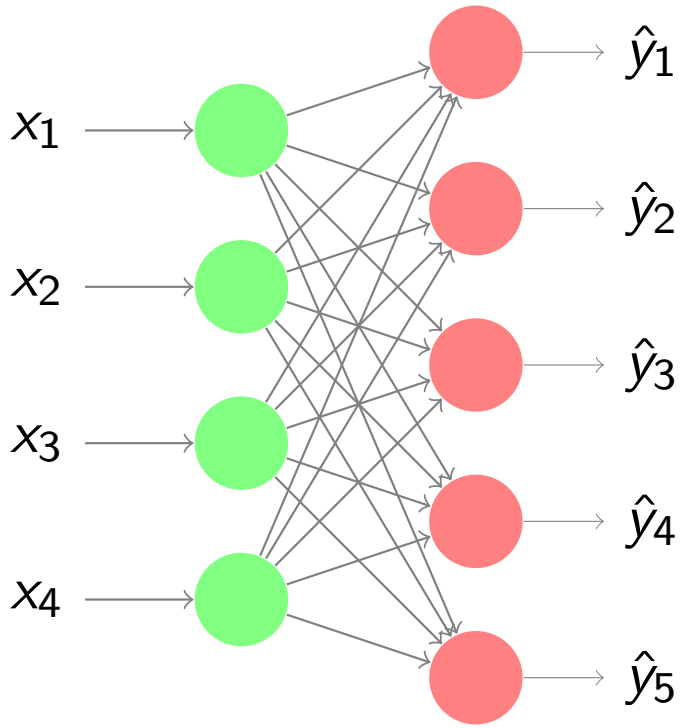
Perceptron monocouche : Fonctionnement



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

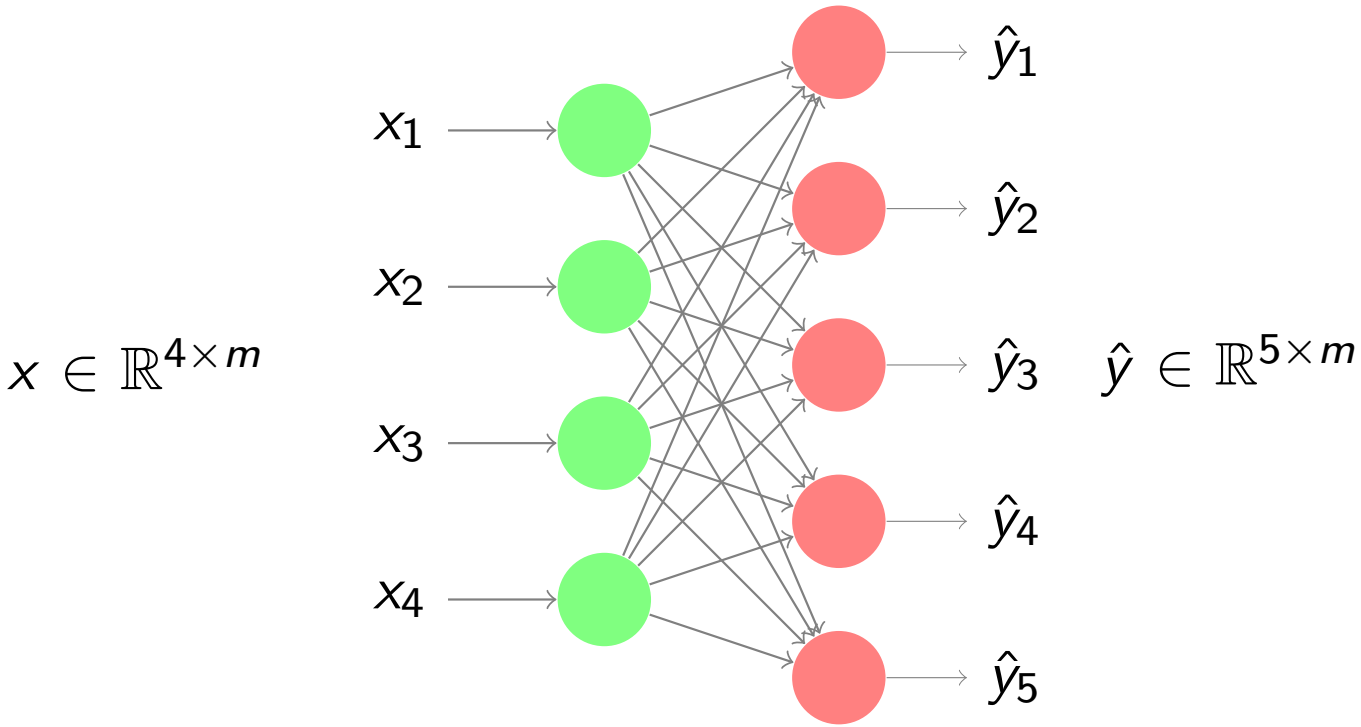
Représentation "neurone"

Réseau de neurones : Perceptron monocouche



Le vocable de perceptron monocouche est en fait un peu plus général et désigne également les cas où la sortie est définie par plusieurs neurones tels que décrits dans les diapositives précédentes.

Réseau de neurones : Perceptron monocouche

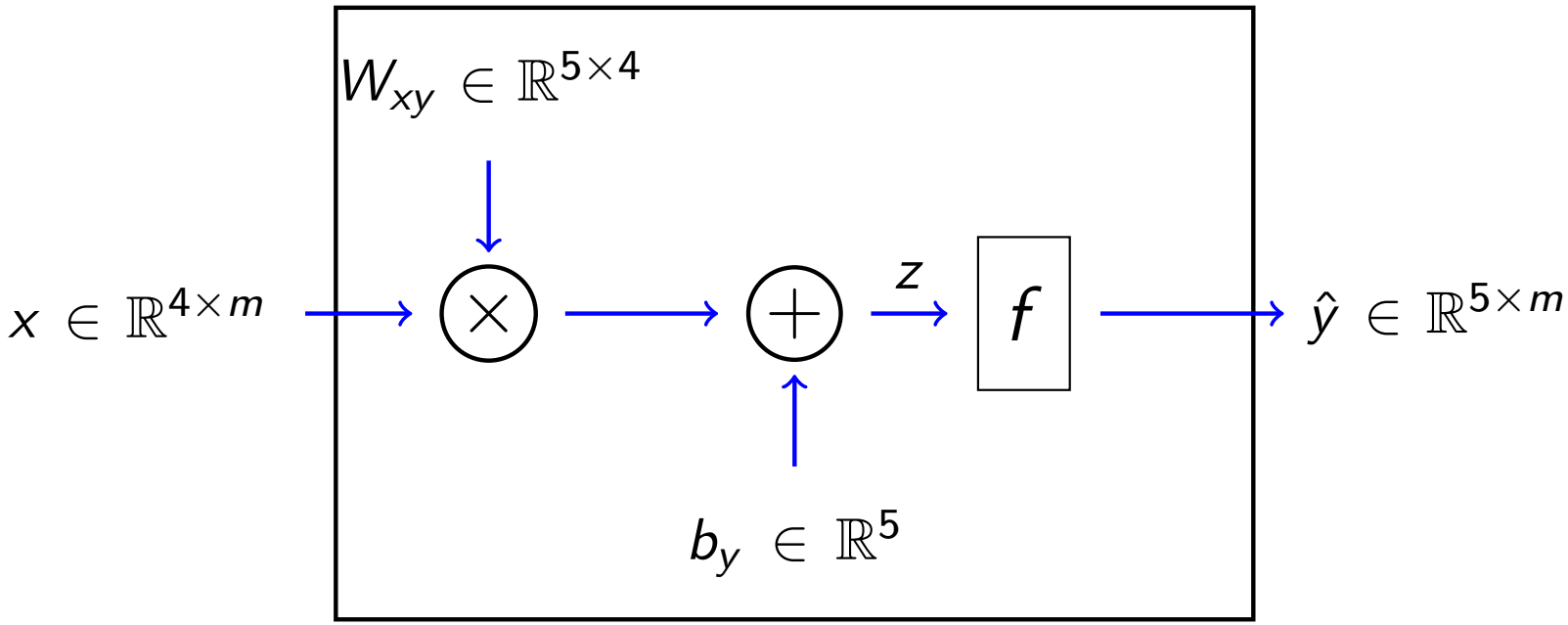


La prédiction du perceptron est implémentée par un calcul matriciel et permet la prédiction de plusieurs entrées simultanément :

$$\hat{y} = f(W_{xy}x + b_y)$$

où f est la fonction d'activation.

Réseau de neurones : Perceptron monocouche

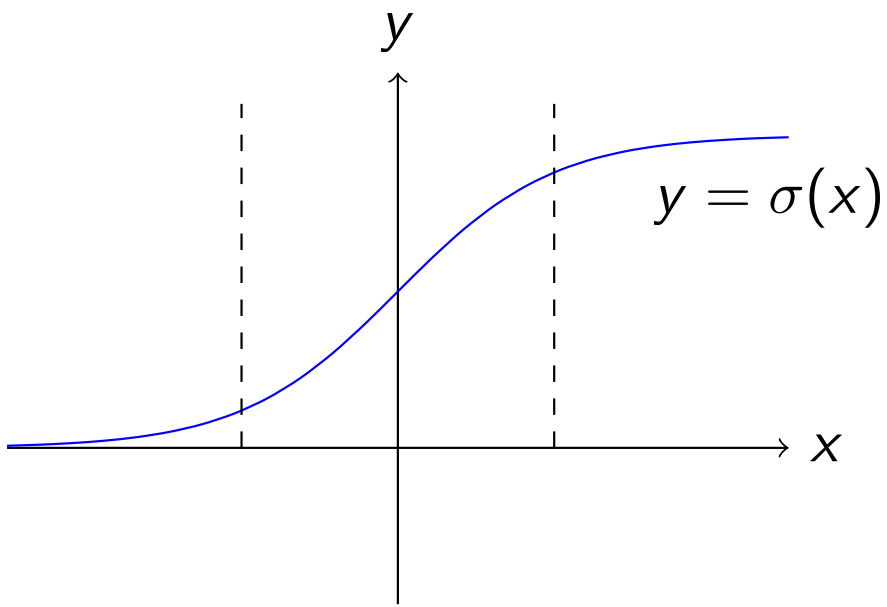


Une autre représentation, utile pour la suite.

Perceptron monocouche

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

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



Perceptron monocouche

Algorithm 1 Algorithme du perceptron monocouche

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

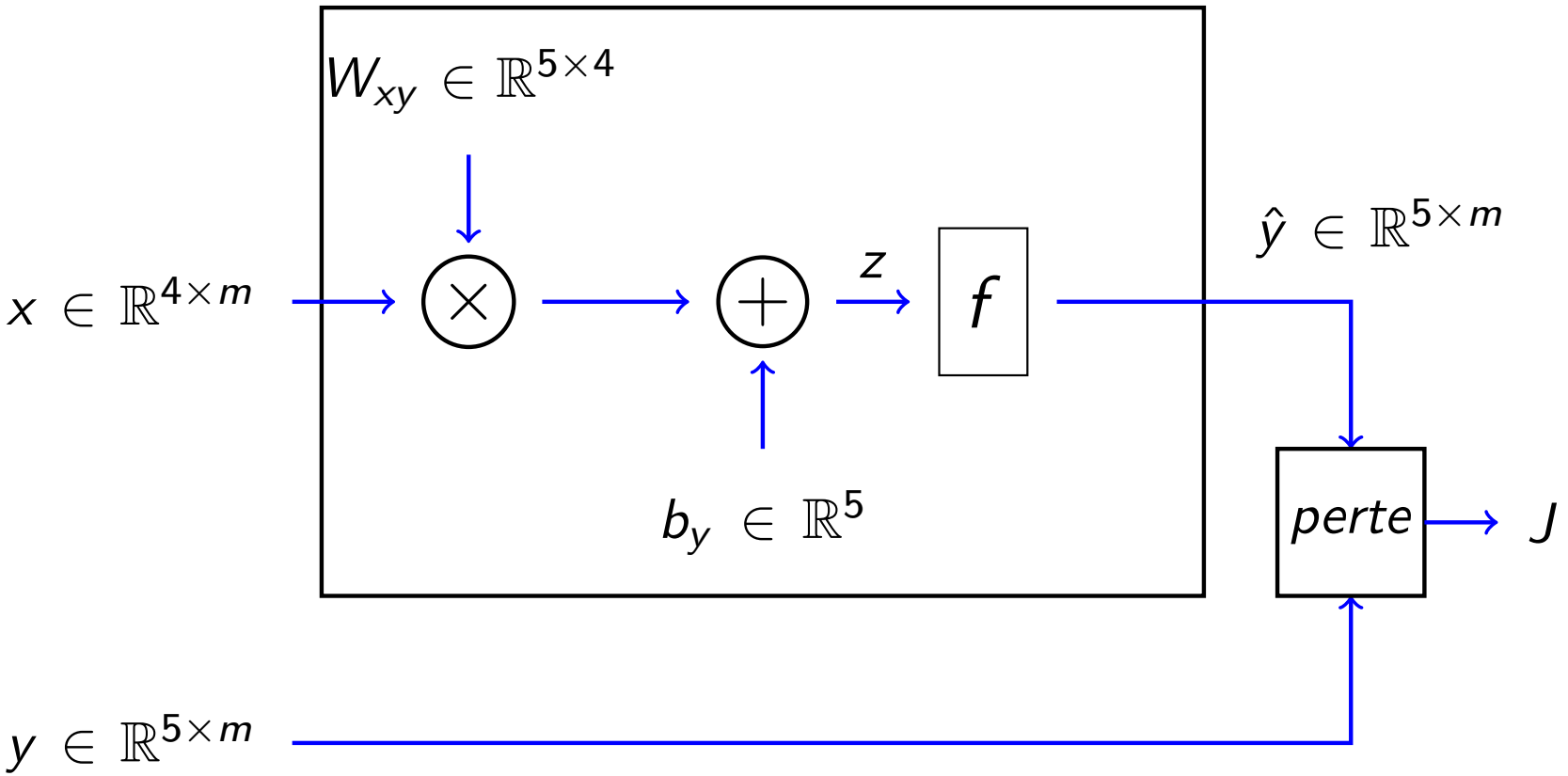
$$\hat{y}^{(i)\{k\}} = f(W_{xy}^{\{k\}} x^{(i)} + b_y^{\{k\}}) \text{ et } J = \sum_{i=1}^{m} \text{perte}(\hat{y}^{(i)\{k\}}, y^{(i)})$$

4. Mise à jour des paramètres

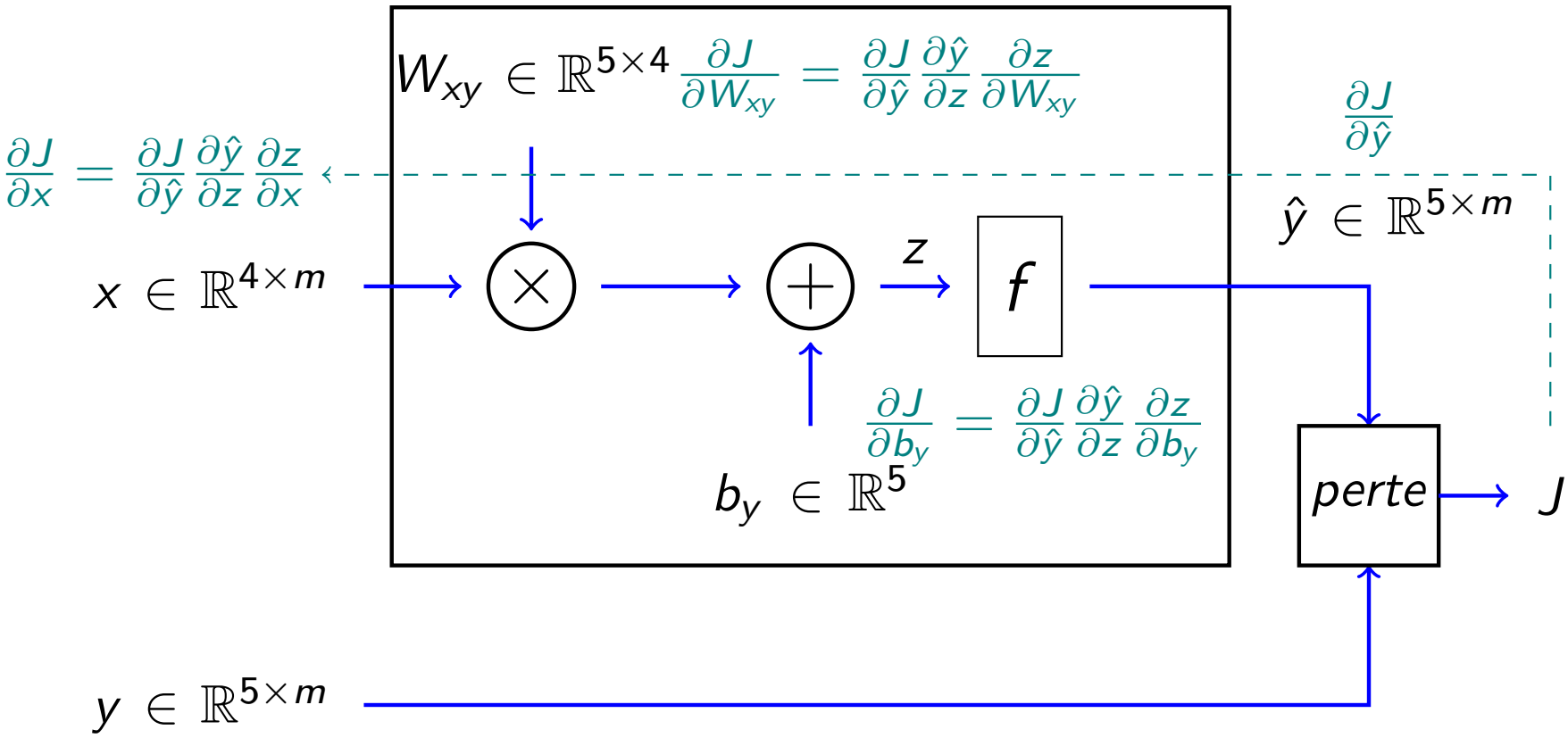
$$W_{xy}^{\{k+1\}} = W_{xy}^{\{k\}} - \alpha \frac{\partial J}{\partial W_{xy}}, \text{ et } b_y^{\{k+1\}} = b_y^{\{k\}} - \alpha \frac{\partial J}{\partial b_y}$$

   où α est le taux d'apprentissage ($0 \leq \alpha \leq 1$).
5. Retourner en 2 jusqu'à la convergence (c'est-à-dire $\hat{y}^{(i)\{k\}} \approx y^{(i)}$).

Perceptron monocouche : fonction objectif



Perceptron monocouche : calcul des gradients



Plus de détails en TP !

IMPORTANT

Pour résoudre un problème de **régression**, choisir :

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

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

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

Classification binaire

IMPORTANT

Pour résoudre un problème de **classification binaire**, choisir :

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

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

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

IMPORTANT

Pour résoudre un problème de **classification n-aire** (plus de deux classes), choisir :

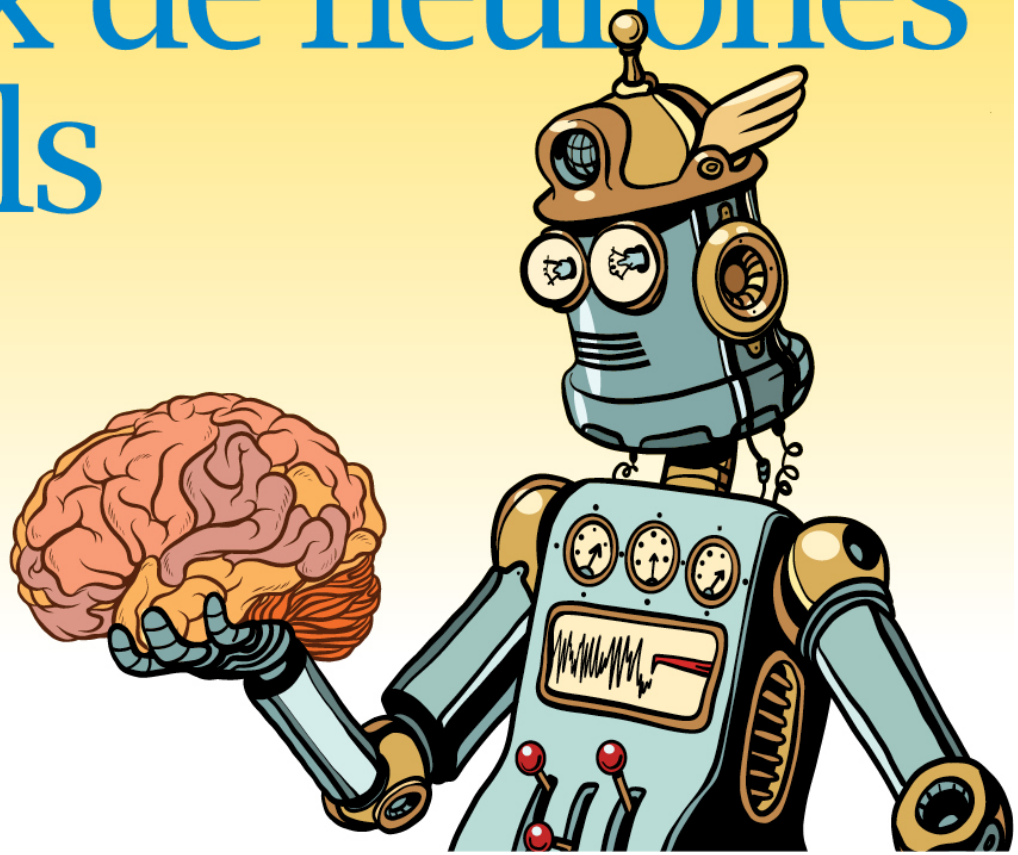
- Fonction d'activation **softmax** : $\text{softmax}(x) = \frac{1}{\sum_{j=1}^{n} e^{x_j}} \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_n} \end{bmatrix}$
- Fonction de coût **entropie croisée multinomiale** :

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

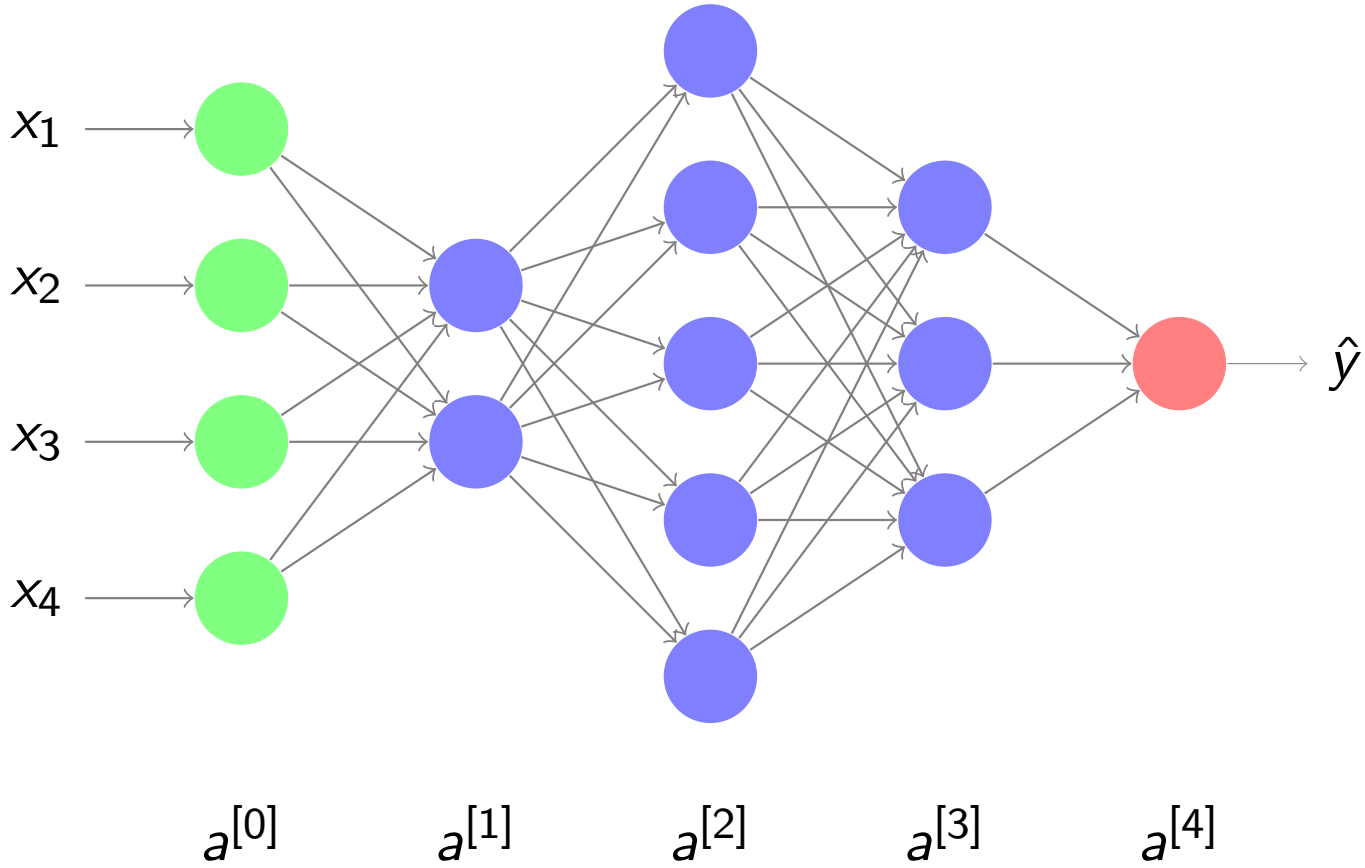
$\Rightarrow$ Perceptron monocouche **équivalent à la régression logistique** !

Plan du cours

Réseaux de neurones artificiels



Réseau de neurones : Perceptron multi-couches



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

Point sur les notations

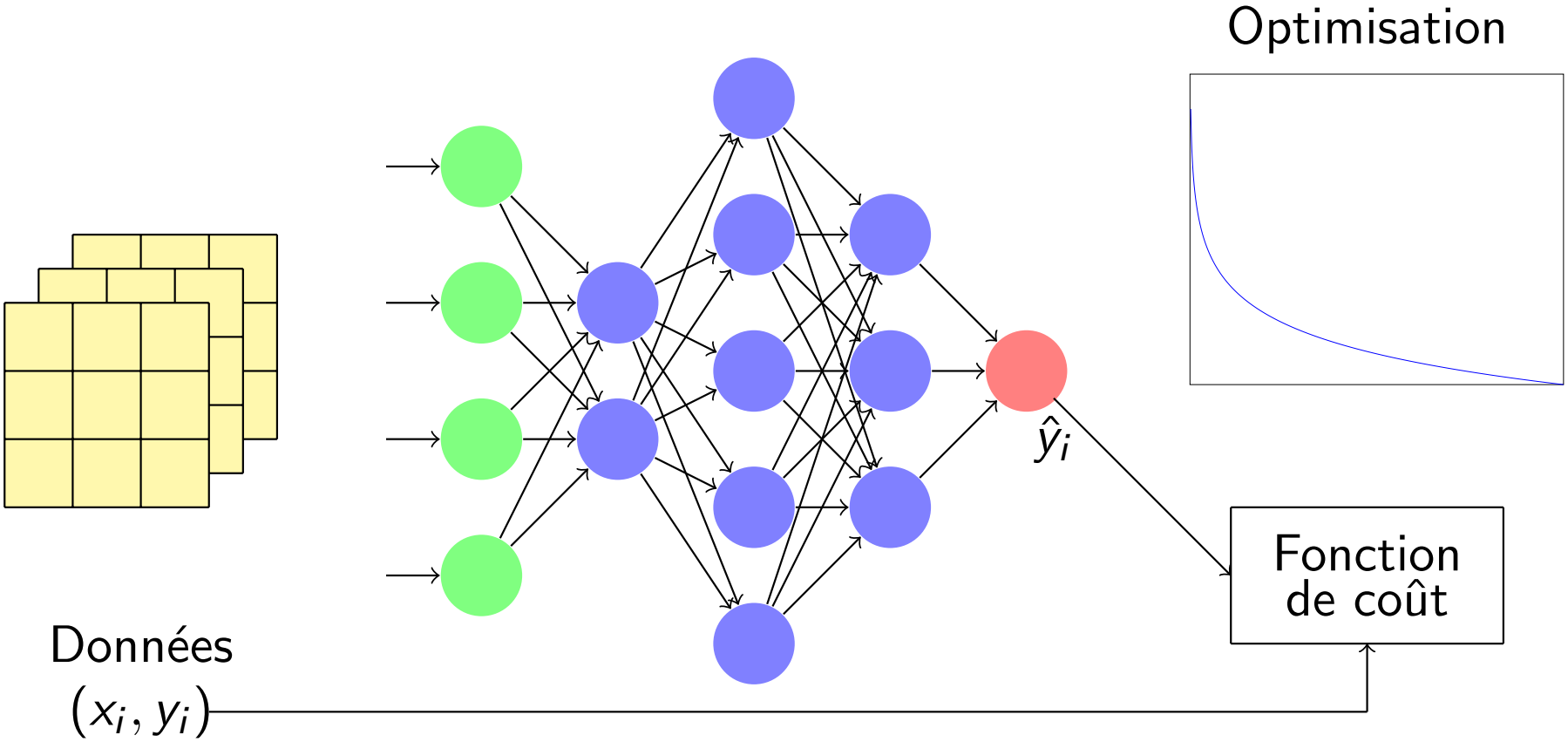
A ce stade, vous avez déjà remarqué la lourdeur des notations :

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

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

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

Vue d'ensemble

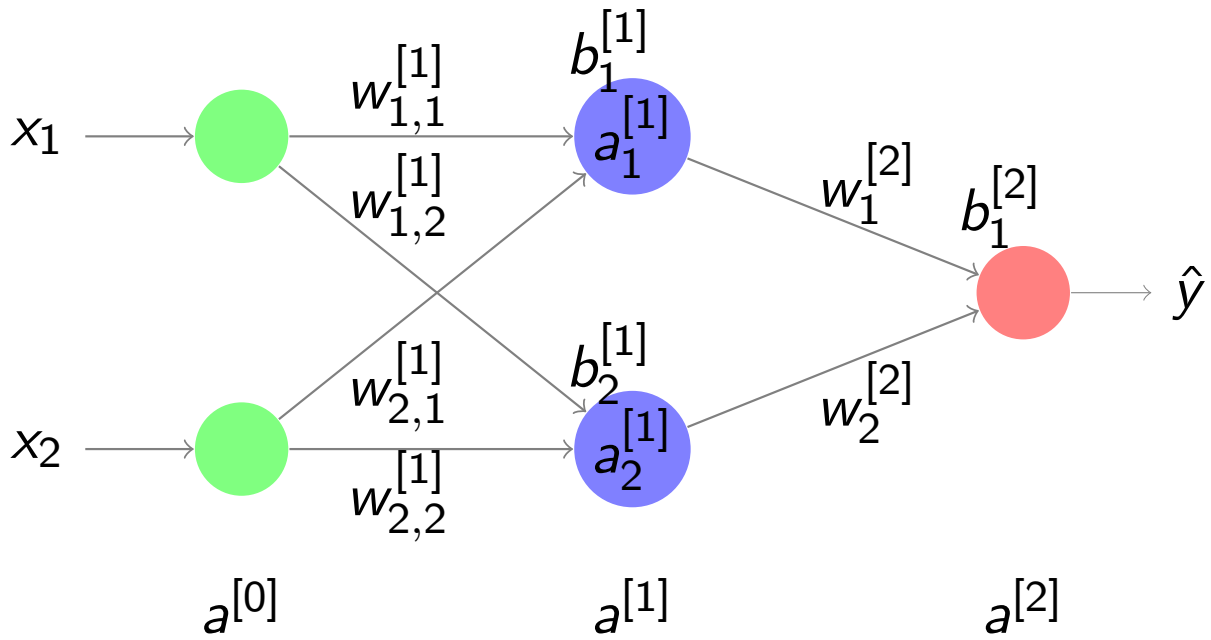


Réseau de neurones : perceptron multi-couches

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

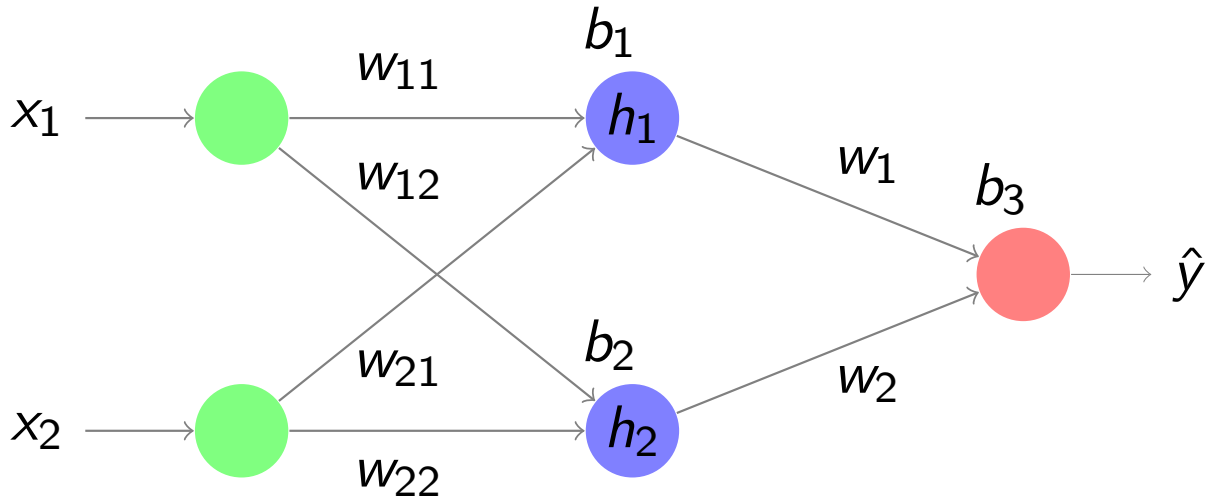
1. **Propagation des données** de la couche d'entrée à la couche de sortie ;
2. Calcul de l'**erreur de sortie** après la propagation des données ;
3. Calcul des **gradients d'erreurs** pour corriger les poids synaptiques des neurones de la **couche de sortie** ;
4. Calcul des gradients d'erreurs pour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

Illustration de l'entraînement d'un perceptron multi-couches



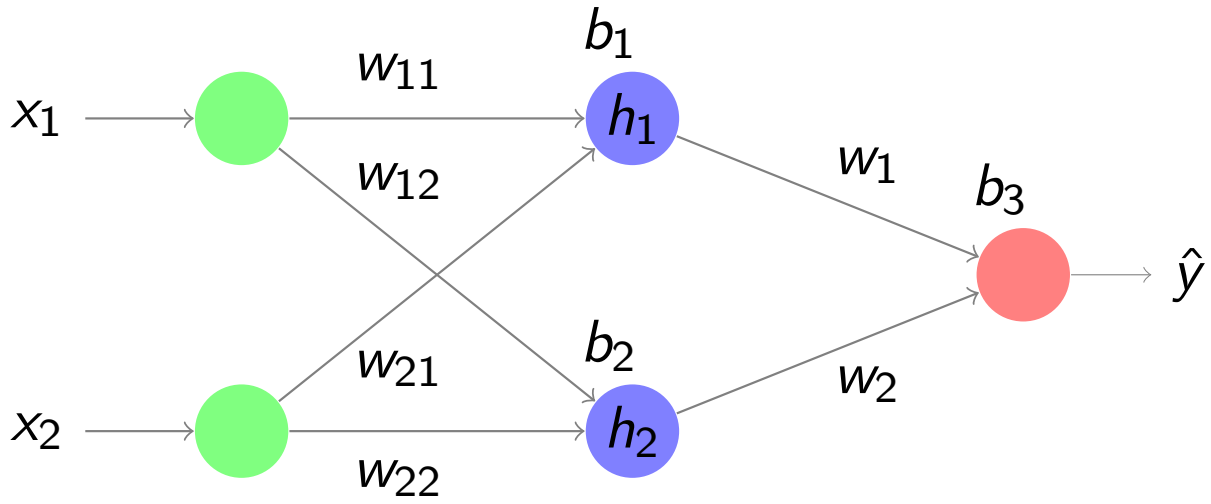
Simplifions un peu les notations (pour enlever l'indice de couche).

Illustration de l'entraînement d'un perceptron multi-couches



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

Perceptron multi-couches : entraînement



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

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

$$\hat{y} = \sigma\left(w_1 f(w_{11} x_1 + w_{21} x_2 + b_1) + w_2 f(w_{12} x_1 + w_{22} x_2 + b_2) - b_3\right)$$

Perceptron multi-couches : entraînement

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

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

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

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

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

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

Perceptron multi-couches : entraînement

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

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

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

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

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

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

et de la couche cachée :

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

Remarque : rétropropagation du gradient

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

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

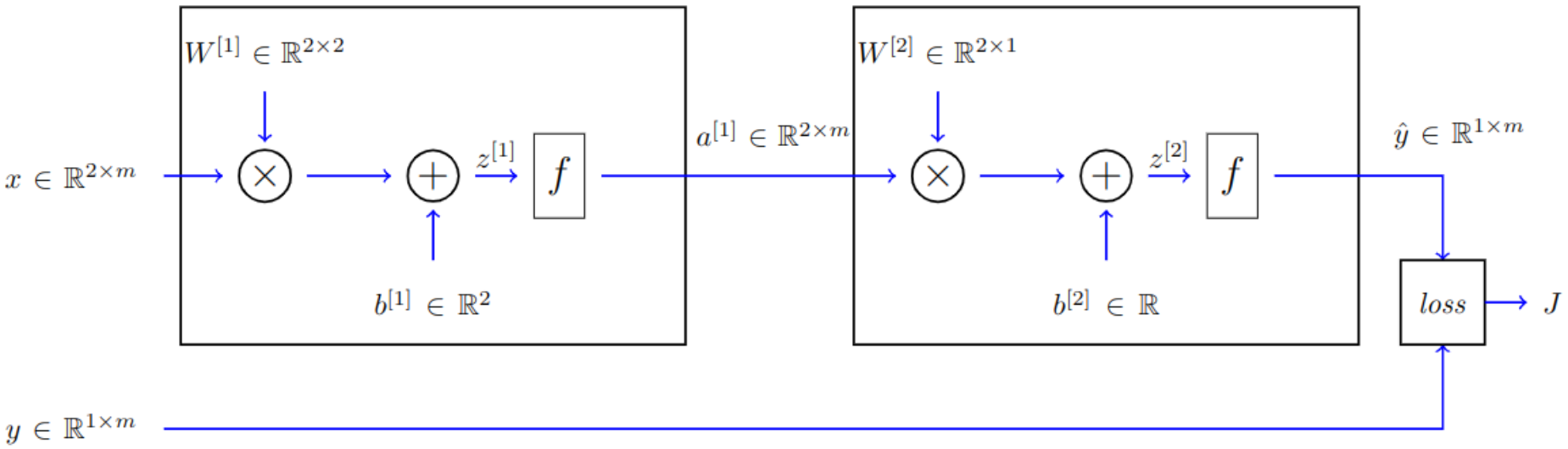
et

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

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

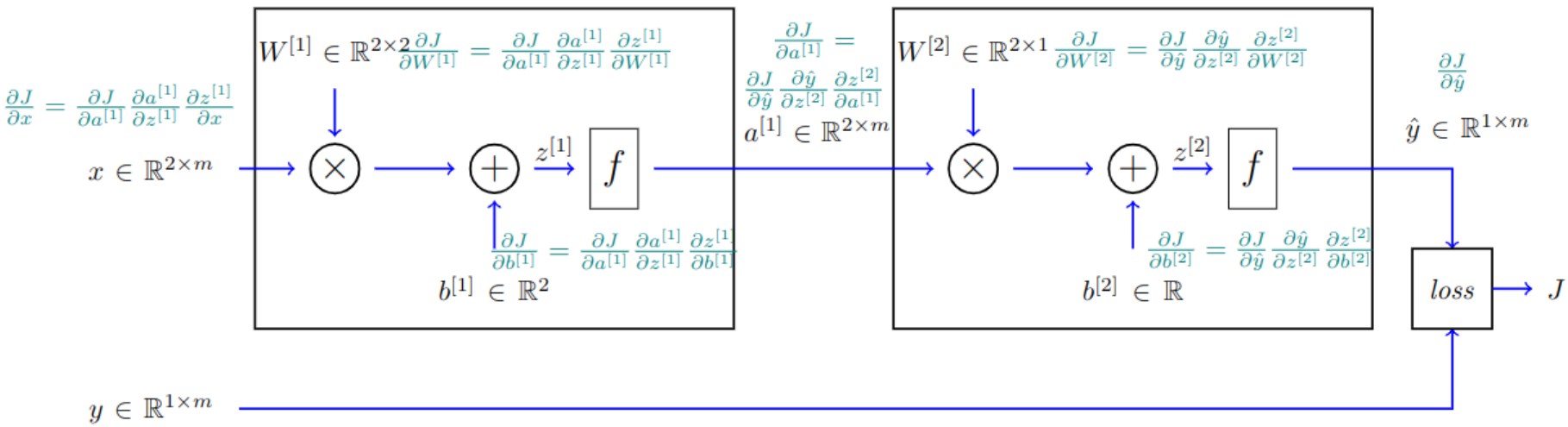
Remarque : rétropropagation du gradient

Une autre visualisation :



Remarque : rétropropagation du gradient

Une autre visualisation :



Perceptron multi-couches : interprétation

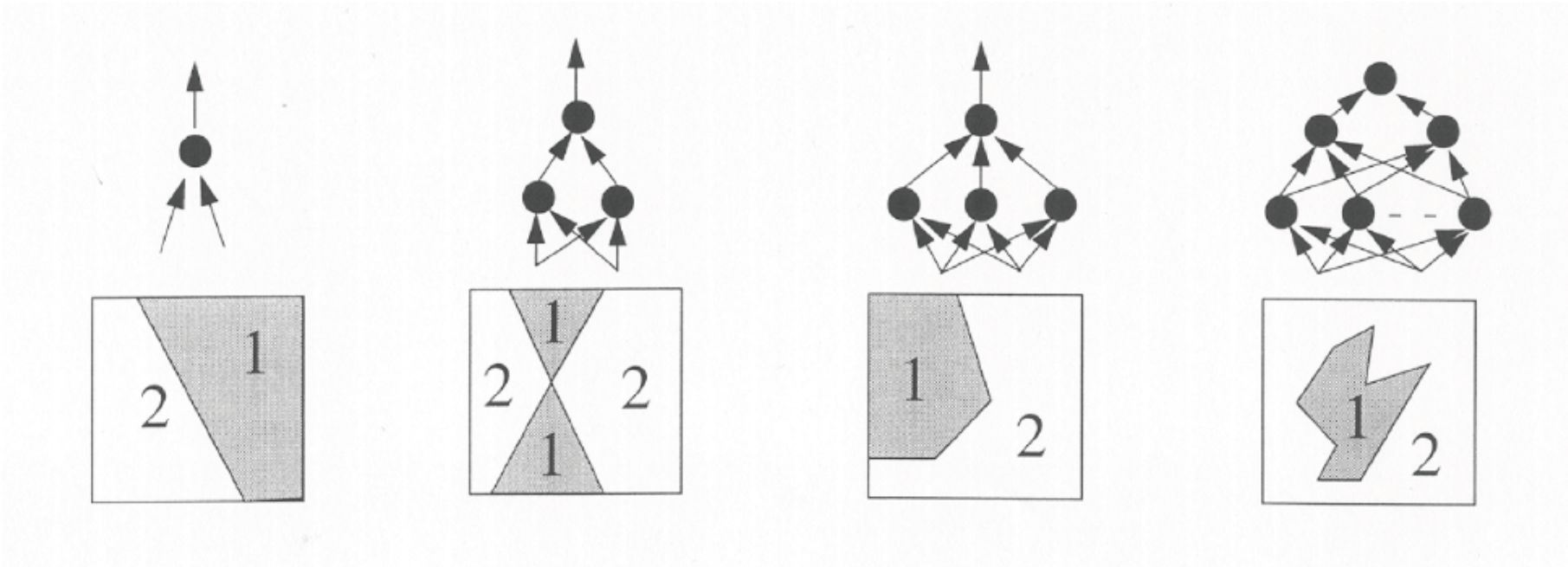


Figure – Intérêt du perceptron multi-couches : pouvoir séparateur

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

Perceptron multi-couches : interprétation

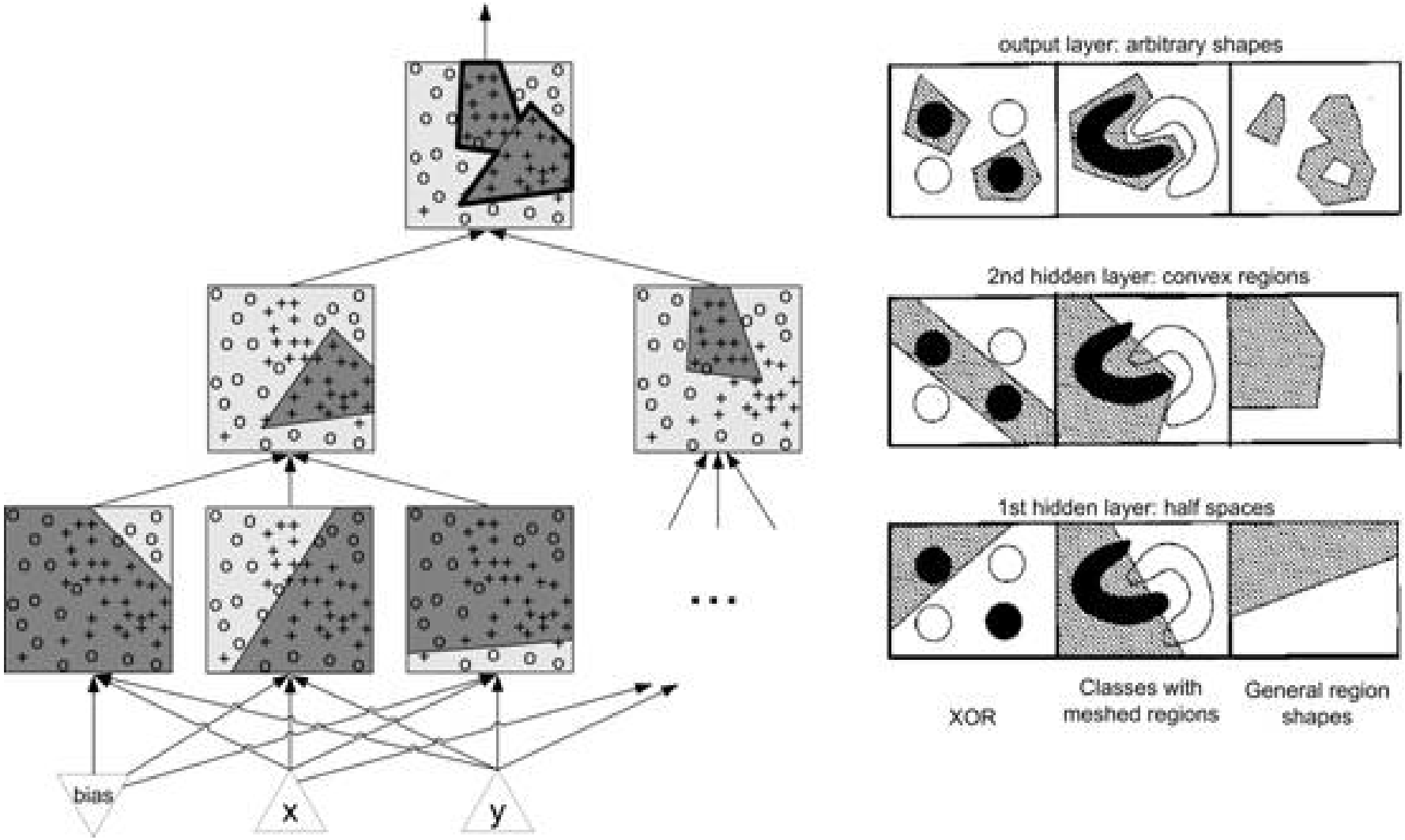


Figure – Intérêt du perceptron multi-couches

Théorème d'Approximation Universelle

Théorème d'approximation universelle (Cybenko 1989)

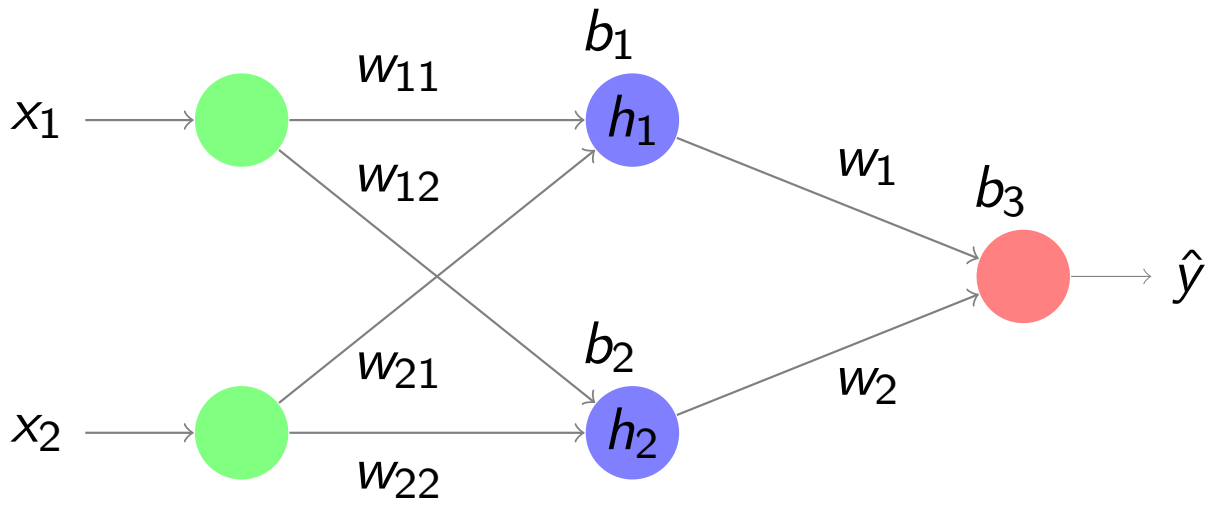
Toute fonction f, continue, de $[0,1]^m$ dans $\mathbb{R}$, peut être approximée par un perceptron multi-couches à une couche cachée comportant suffisamment de neurones (avec une fonction d'activation sigmoide).

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

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

Plan du cours

Retour sur le calcul des gradients



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

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

Retour sur le calcul des gradients

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

On a :

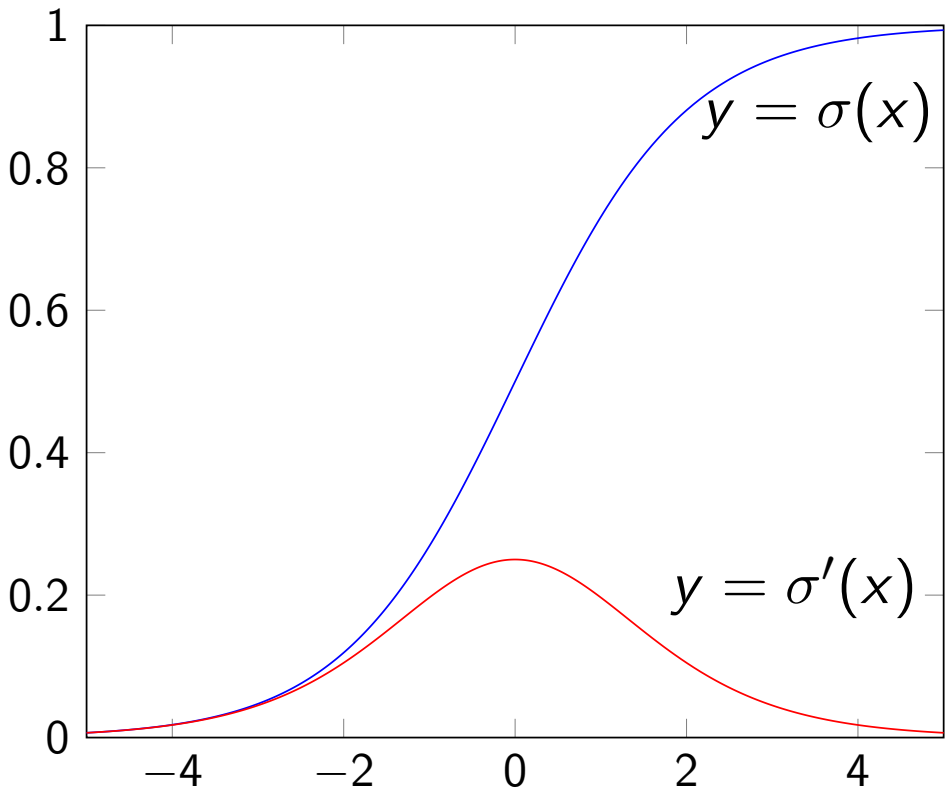
$$\frac{\partial z}{\partial h_j} = w_j$$

$$\frac{\partial h_j}{\partial z_j} = f'(z_j)$$

avec f' la dérivée de la fonction d'activation portée par le neurone.

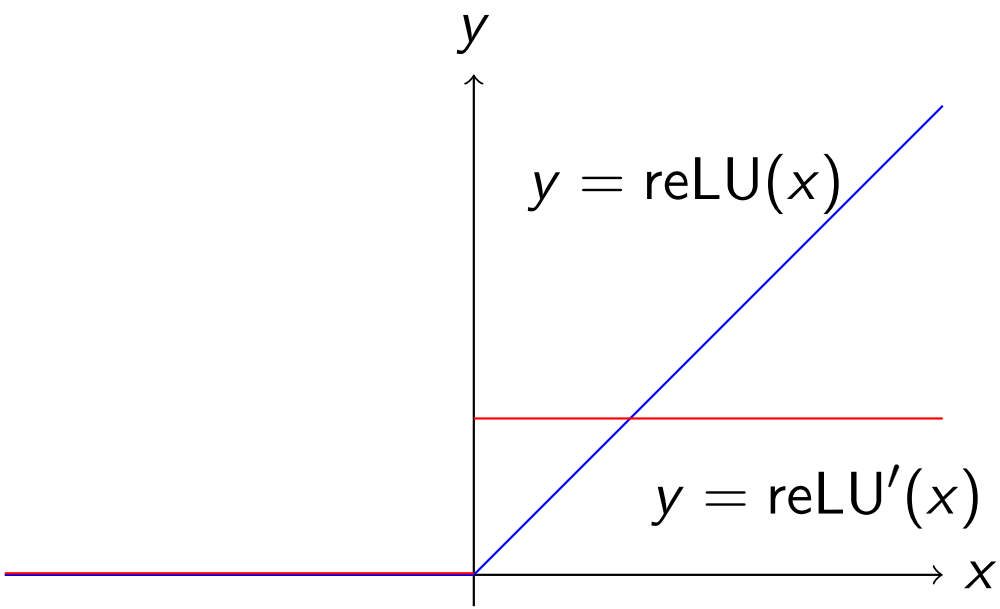
$$\frac{\partial z_j}{\partial w_{ij}} = x_i$$

Gros plan sur la fonction sigmoïde



La dérivée de la fonction sigmoïde est à valeurs dans $[0, \frac{1}{4}]$, ce qui diminue l'amplitude des gradients propagés à travers les couches du réseau de neurones. C'est le problème de l'**évanescence des gradients** (*vanishing gradients*).

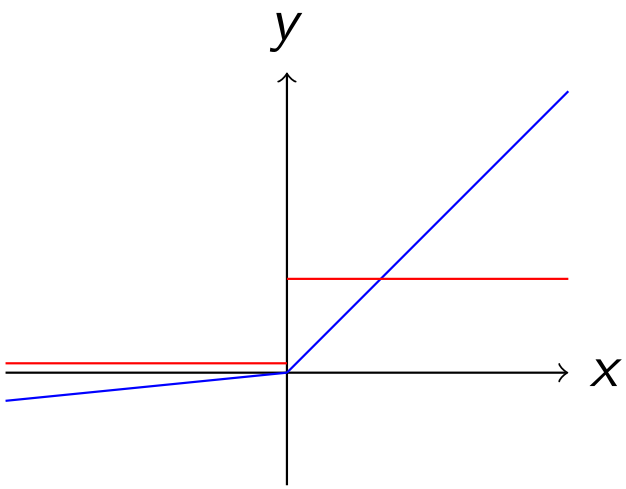
La fonction *rectified Linear Unit*

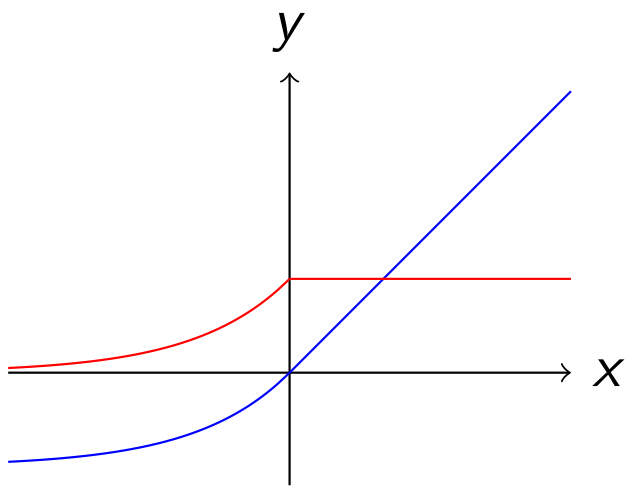


$$\text{reLU}(x) = \begin{cases} 0 & \text{si} \quad x < 0 \\ x & \text{sinon} \end{cases} \tag{4}$$

Le fait d'avoir un gradient constamment égal à 1 dans la zone activée améliore grandement la convergence.

D'autres variantes





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

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

Leaky Rectified Linear Unit

Exponential Linear Unit

Ces fonctions améliorent la convergence de la descente de gradient en limitant les occurences où le gradient est nul. De plus, en autorisant les activations négatives, le problème d'optimisation est mieux conditionné.

Plan du cours

Descente de gradient

Algorithme : Descente de gradient $(\mathcal{D}, \alpha)$
Initialiser $\theta^{\{0\}} \leftarrow 0$, $k \leftarrow 0$
TANT QUE pas convergence **FAIRE**
    POUR j de 1 à d **FAIRE**

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

    FIN POUR
    $k \leftarrow k + 1$
FIN TANT QUE

Calculer le gradient exact est **coûteux** parce qu'il faut évaluer le modèle sur tous les m exemples de l'ensemble de données à chaque itération.

Descente de gradient stochastique

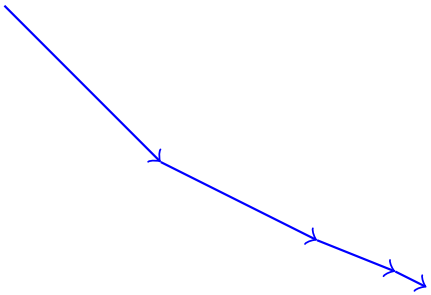
On peut en fait distinguer plusieurs alternatives pour l'algorithme de la descente de gradient :

- Algorithme **Batch** : utilise tous les m exemples de l'ensemble d'apprentissage pour calculer le gradient.
- Algorithme **Mini-batch** : approxime le gradient en le calculant en utilisant k exemples parmi les m échantillons de l'ensemble d'apprentissage, où $m >> k > 1$.
- Algorithme **Stochastique** : approche le gradient en le calculant sur un seul exemple ($k = 1$).

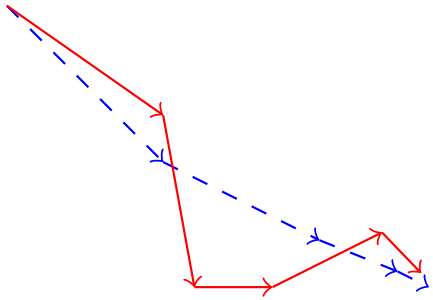
Par abus de langage, on utilise le terme de descente de gradient stochastique (SGD) y compris pour l'algorithme Mini-batch.

Descente de Gradient Stochastique

L'idée de la descente de gradient stochastique est que le gradient à calculer étant une espérance sur l'ensemble d'apprentissage, il est possible de l'estimer approximativement à l'aide d'un petit ensemble d'échantillons, appelé *mini-batch*. (C'est le même principe que pour les sondages d'opinion !)



Descente de gradient



Descente de gradient stochastique

Remarque : un mini-batch trop petit peut engendrer un bruit trop grand sur l'estimation du gradient et empêcher la convergence.

Variante 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 :

- Algorithme **Batch** : 1 itération par *epoch*.
- Algorithme **MiniBatch** : $\frac{m}{k}$ itérations par *epoch*.
- Algorithme **Stochastique** : m itérations par *epoch*.

Momentum

Pour éviter les problèmes liés à une stabilisation dans un minimum local, on ajoute un terme d'inertie (*momentum*).

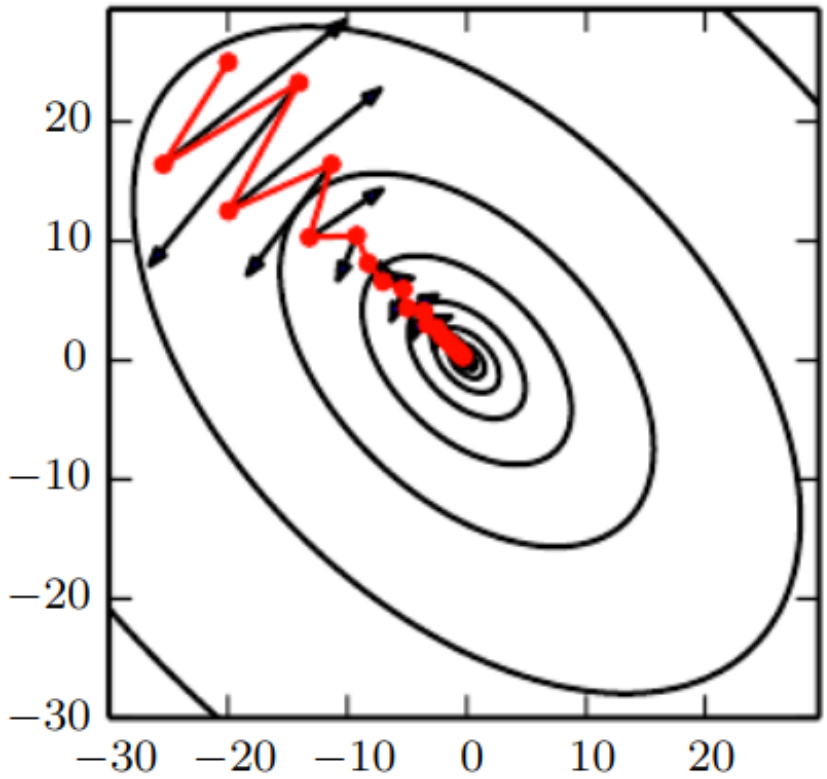


Image de [Goodfellow et al. 2015] Deep Learning

Momentum

En pratique, on adapte l'algorithme de descente du gradient, en remplaçant la mise à jour des paramètres

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

par deux étapes :

$$v = \eta v - \alpha \frac{\partial J}{\partial \theta}$$

$$\theta \leftarrow \theta + v$$

où v (pour vélocité) désigne la direction dans laquelle les paramètres vont être modifiés. v prend en compte les gradients précédents via le paramètre η ($0 < \eta < 1$), qui quantifie l'importance relative des gradients précédents par rapport au gradient courant.

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 parfois inefficace. On peut améliorer cette dernière en utilisant des optimiseurs adaptés.

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

1. Calcul du gradient : $g = \frac{\partial J}{\partial \theta}$
2. Accumulation des gradients : $r = r + ||g||_2$
3. Mise à jour des paramètres : $\theta \leftarrow \theta - \frac{\alpha}{\sqrt{r}} g$

Optimiseurs améliorant la descente de gradient stochastique

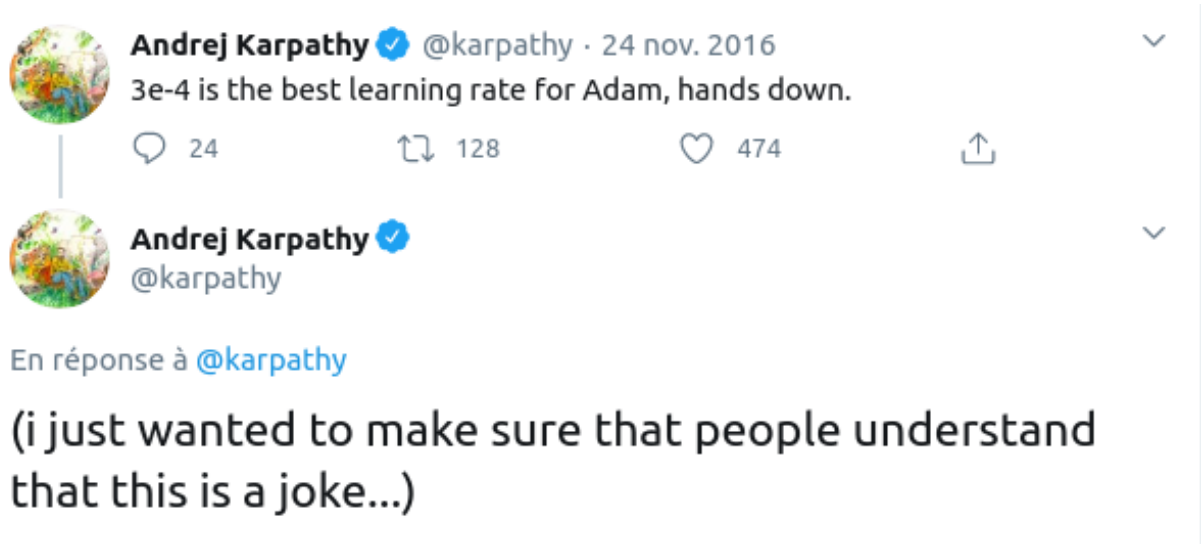
L'optimiseur **RMSProp** est presque identique à AdaGrad, mais l'impact des plus anciens gradients est altéré par un coefficient multiplicatif ρ inférieur à 1 (*weight decay*), ce qui améliore le comportement de l'algorithme dans le cas des bols allongés.

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

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

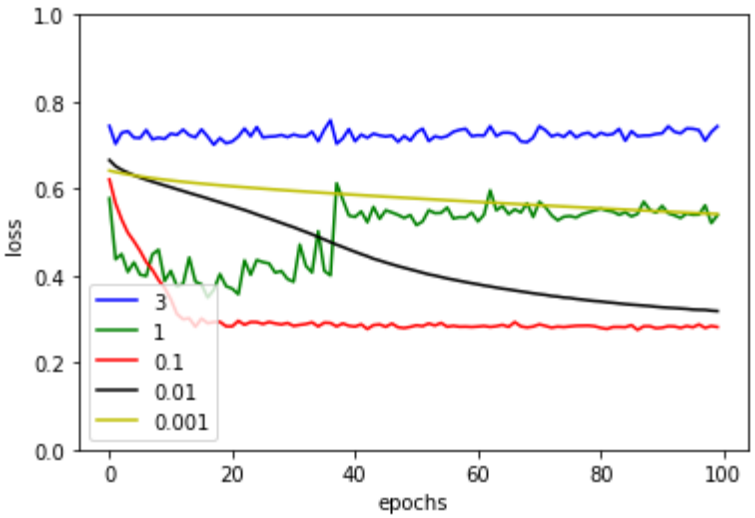
En pratique : choix de l'optimiseur

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

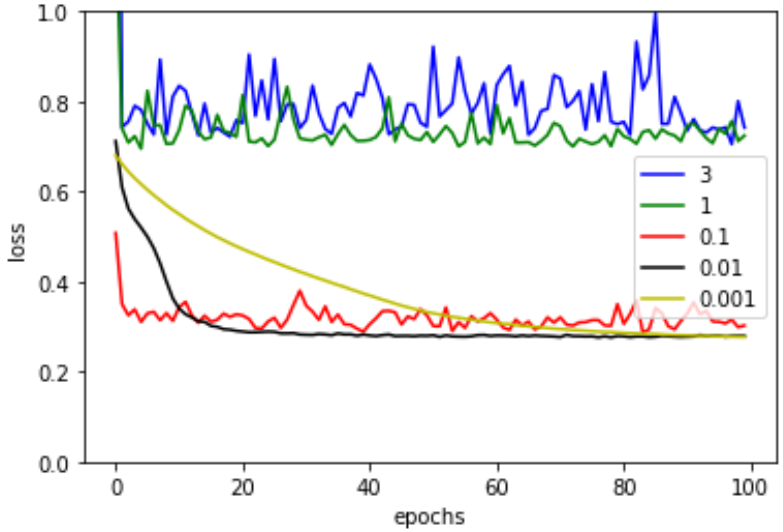


En pratique, les meilleurs résultats mis en avant dans les articles sont obtenus avec une simple descente de gradient stochastique, et une mise à jour programmée du taux d'apprentissage (cyclique, cosinus, etc.)

SGD vs. Adam sur un exemple simple



SGD

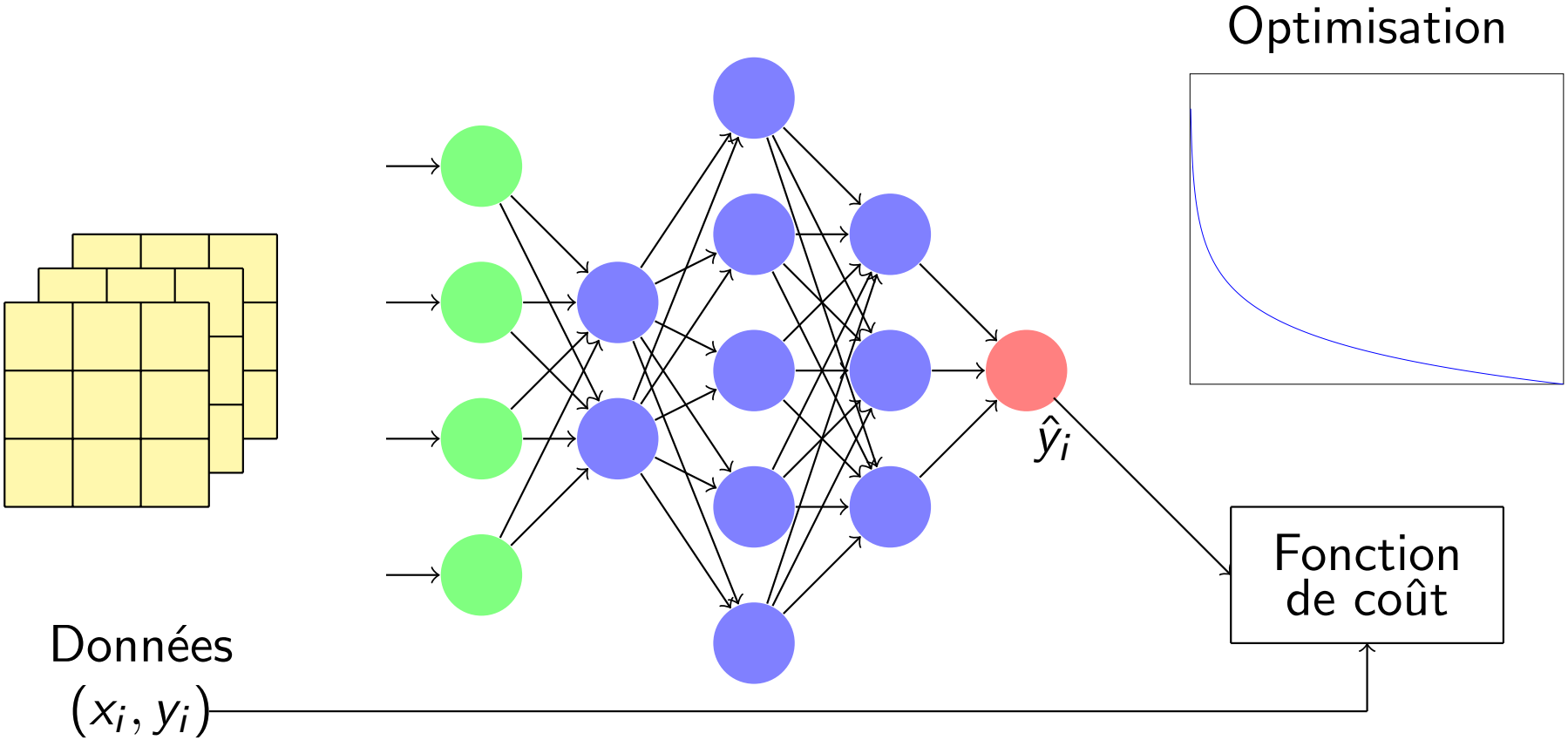


Adam

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

Plan du cours

Vue d'ensemble



profondeur, #neurones, activation α, momentum

$w_{i,j}, b_k$

hyperparamètres et paramètres

Risque empirique et Risque espéré

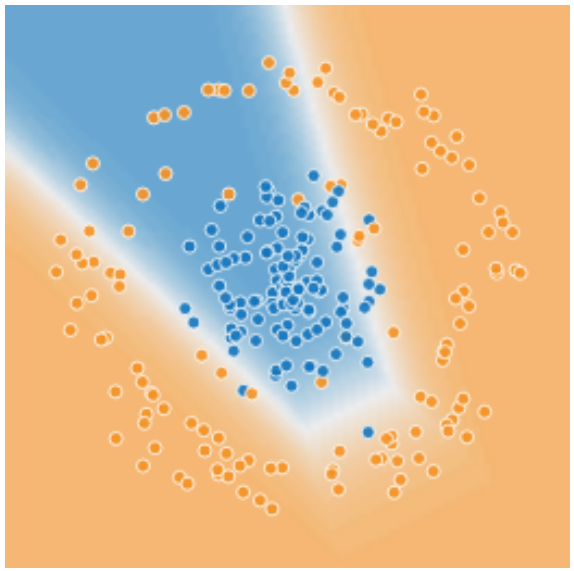
Risque empirique : erreur de prédiction moyenne sur l'ensemble d'apprentissage.
Risque espéré (ou risque de généralisation) : erreur de prédiction moyenne sur la population cible... **Inconnu !**

L'objectif principal d'un algorithme d'apprentissage est de minimiser le **risque espéré**, mais nous ne sommes capables d'évaluer que le **risque empirique**.

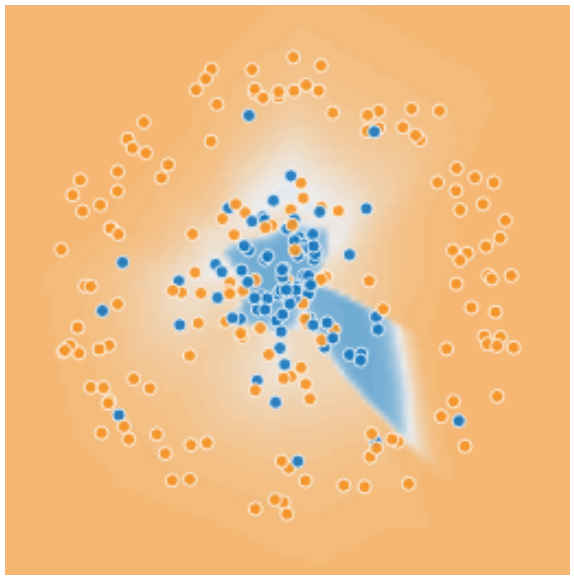
Sous/Sur-apprentissage

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



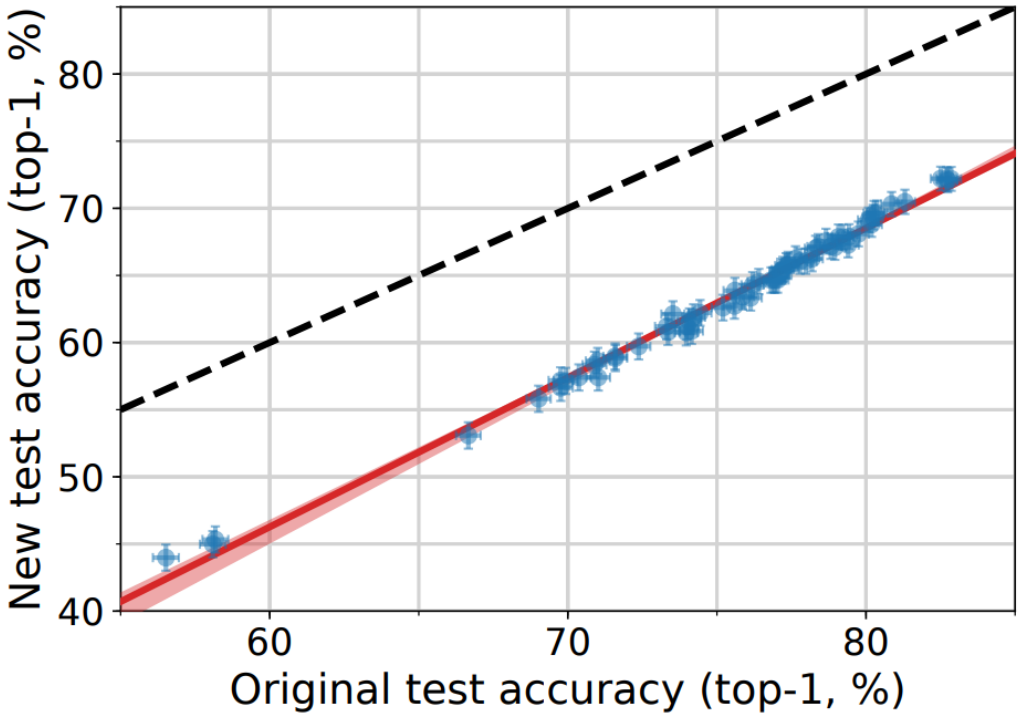
Sur-apprentissage

Sous/Sur-apprentissage

Solution : gérer 2 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 **test** : qui détermine la performance objective du réseau de neurones.

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.

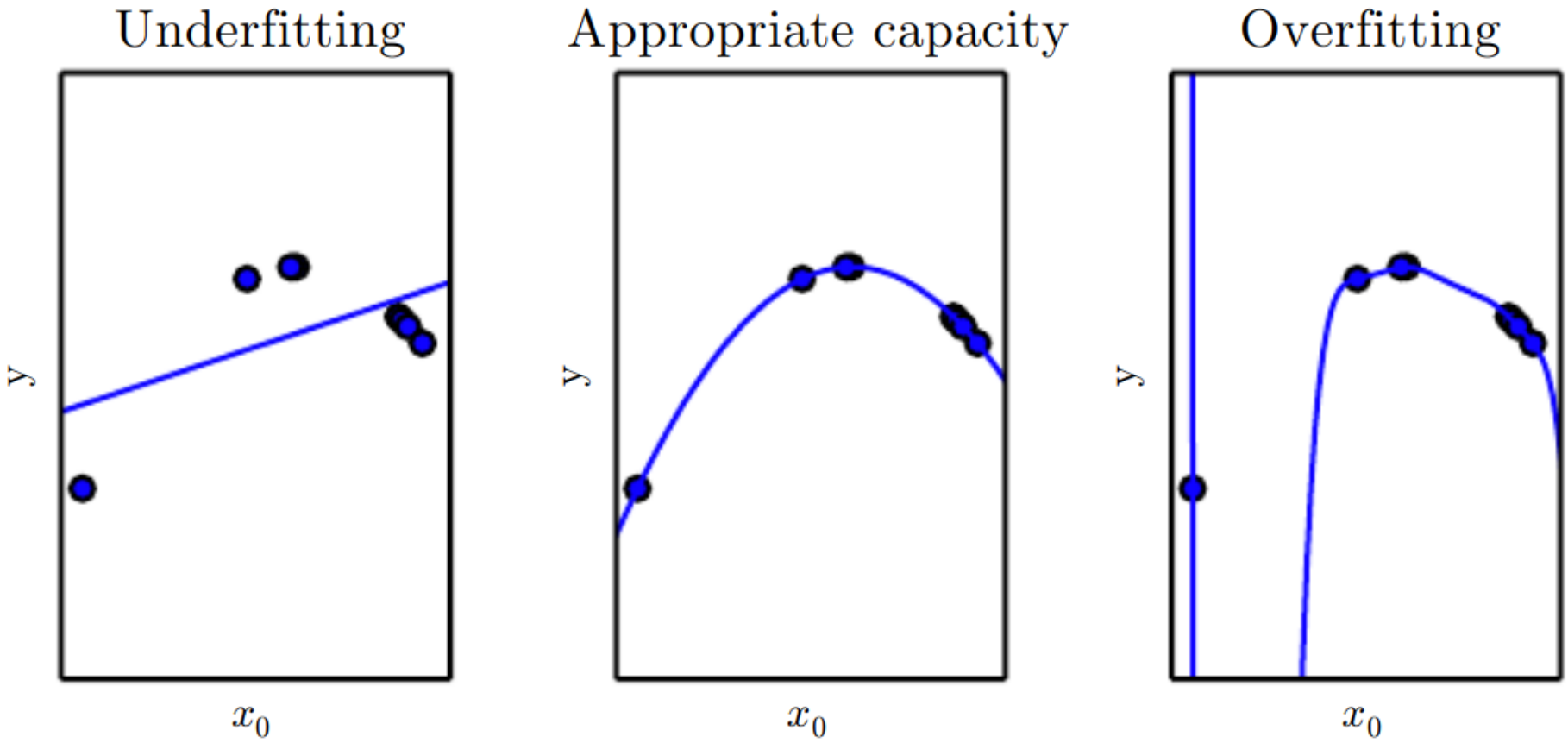
[Recht et al. 2019] Do ImageNet Classifiers Generalize to ImageNet ?

Sous/Sur-apprentissage

Méthodologiquement, il est donc plus juste de gérer 3 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.
- l'ensemble de **test** : qui détermine la performance objective du réseau de neurones.

Sous/Sur-apprentissage : le problème de la **capacité** du modèle

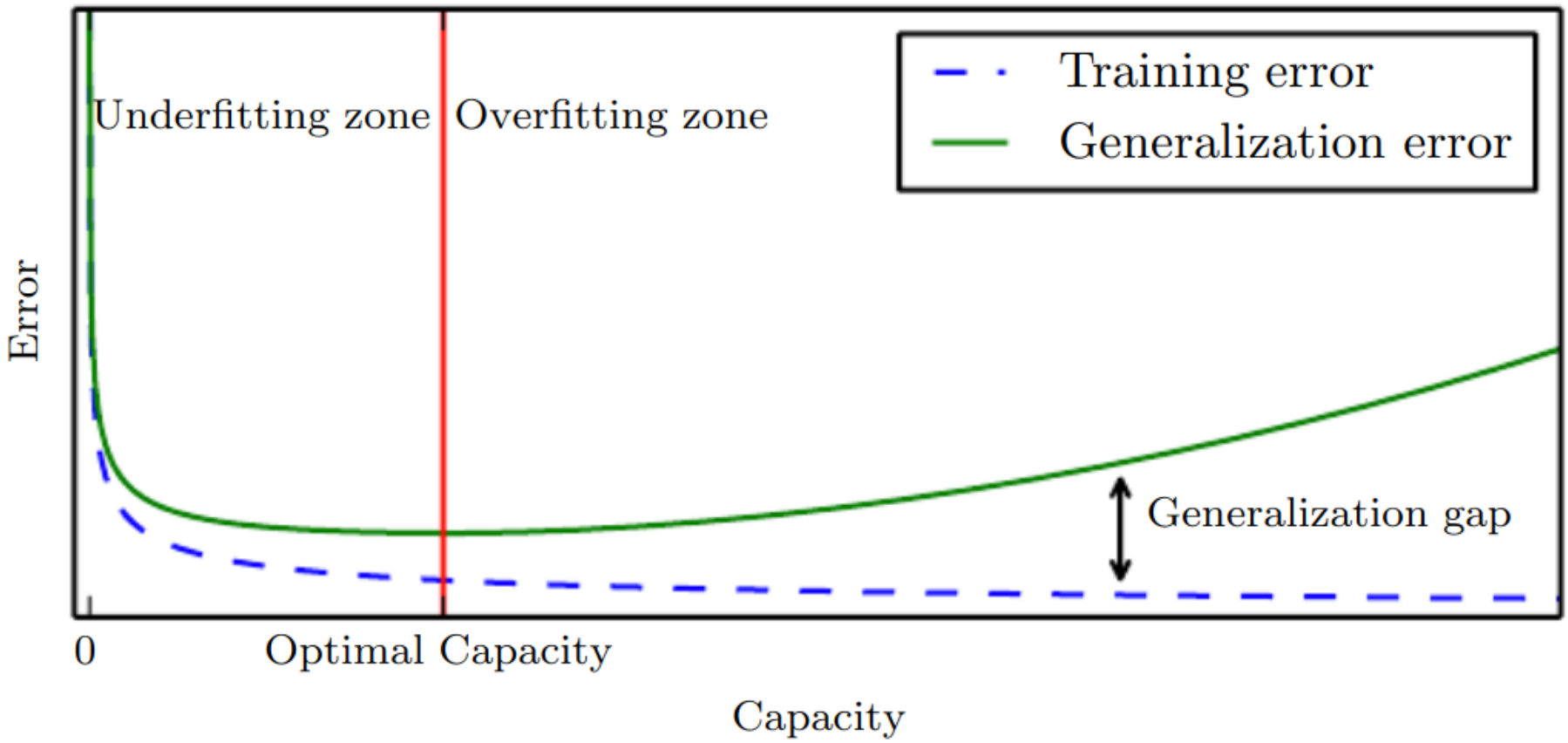


Un modèle de trop faible capacité engendre du sous-apprentissage.
Un modèle de trop large capacité engendre du sur-apprentissage.

Image de [Goodfellow et al. 2015] Deep Learning

Sous/Sur-apprentissage

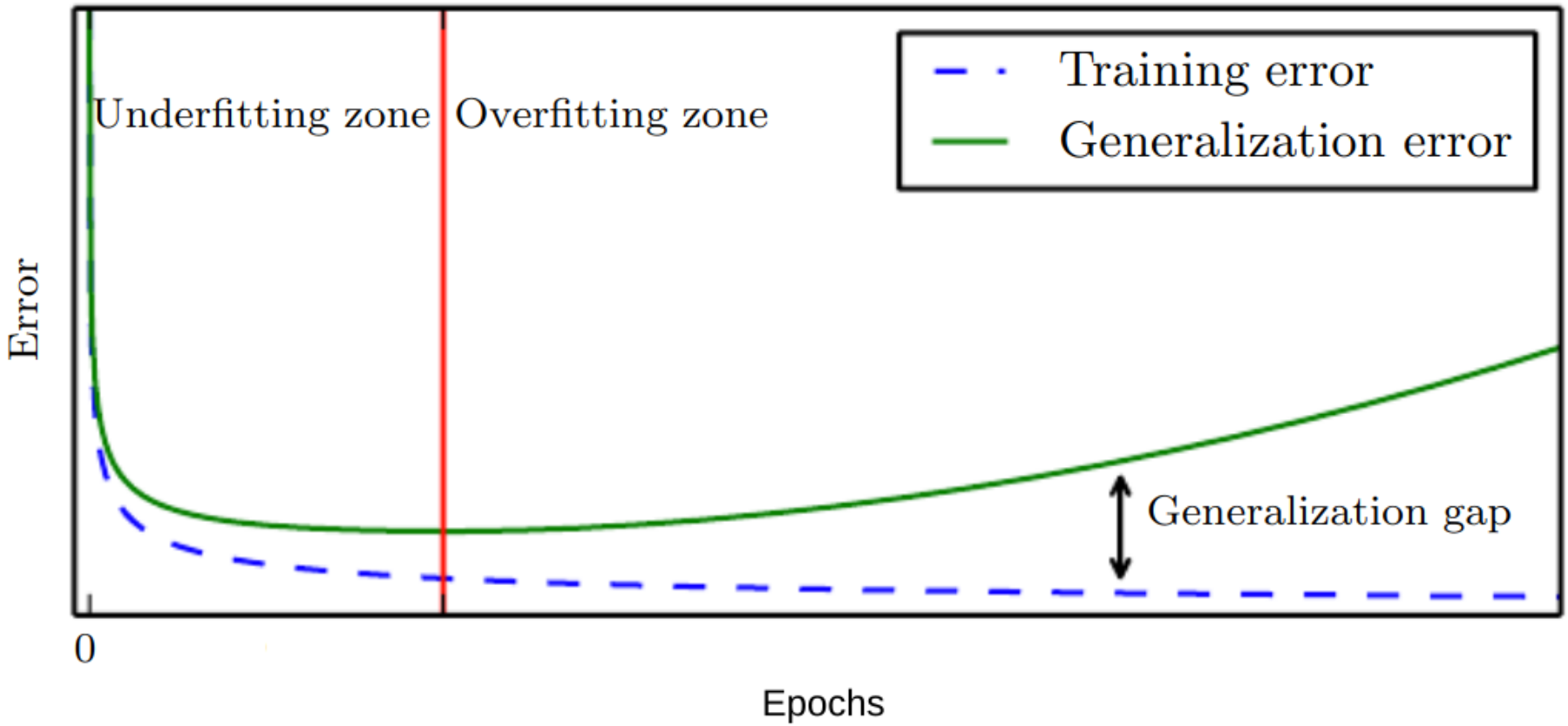
Illustration graphique :



Il est nécessaire d'identifier la capacité (profondeur, nombre de neurones) du réseau qui induit un apprentissage optimal.

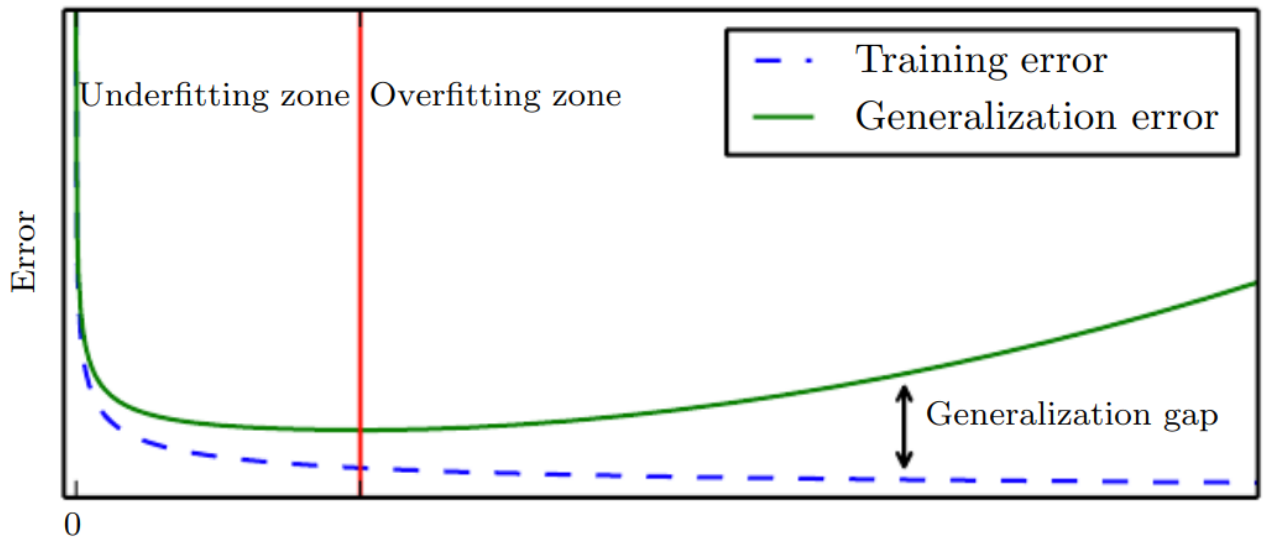
Image de [Goodfellow et al. 2015] Deep Learning

Sous/Sur-apprentissage



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

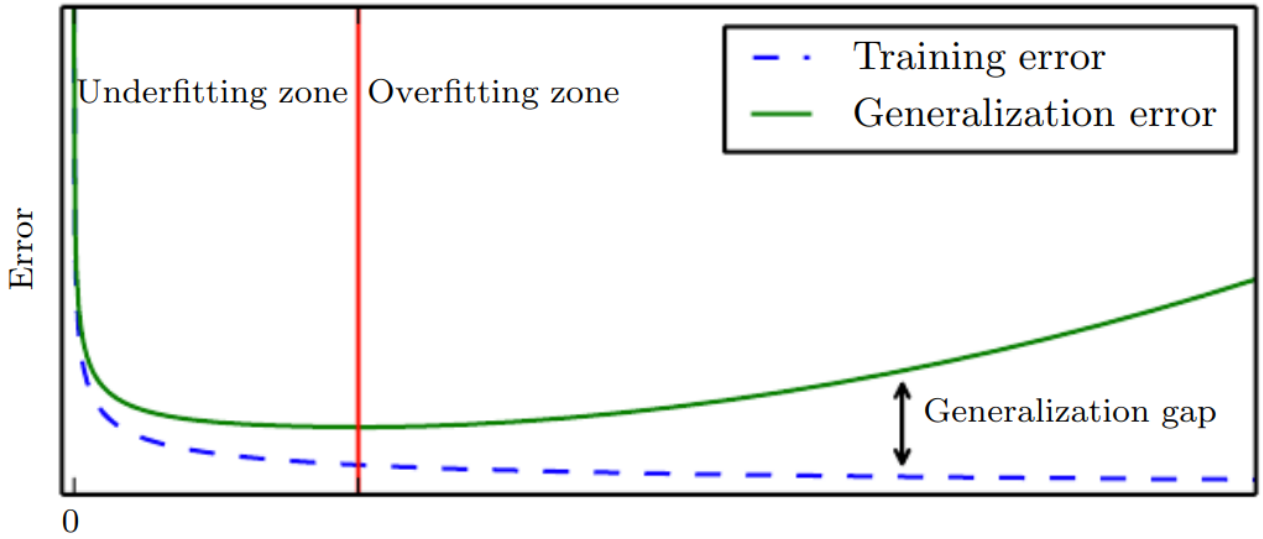
Que faire en cas de sous-apprentissage ?



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 ?



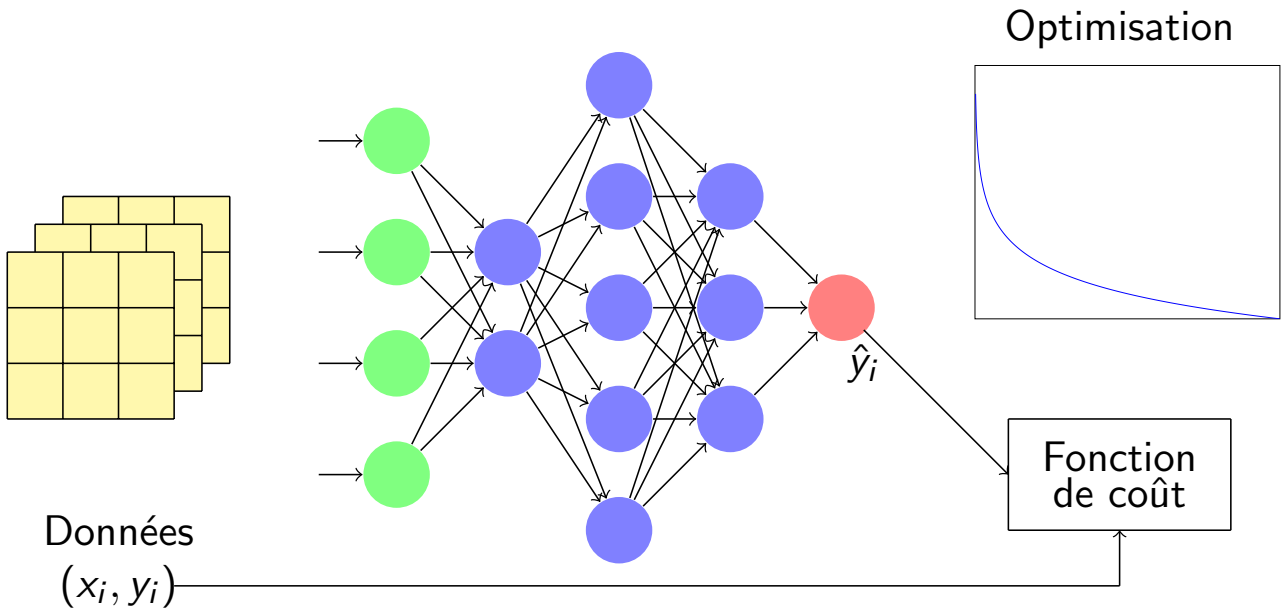
On peut :

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

Plan du cours

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.
En pratique : l'erreur sur l'ensemble de validation est bruitée, il faut attendre un peu avant de s'arrêter pour de bon.

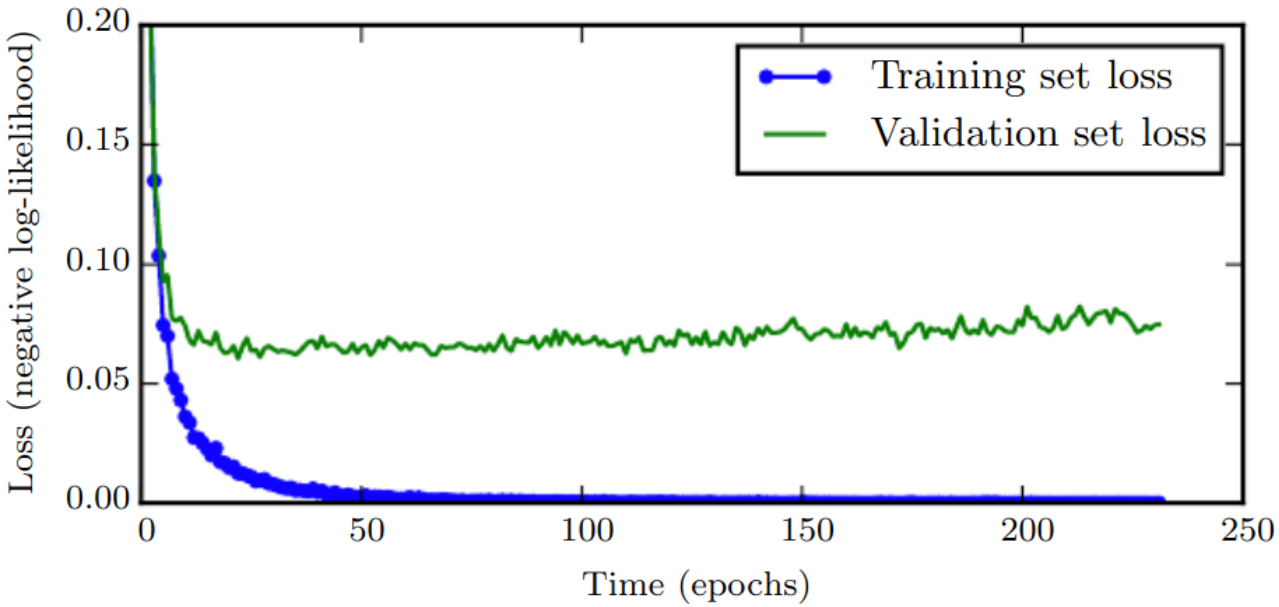


Image de [Goodfellow et al. 2015] Deep Learning

Weight decay

Ajout d'une contrainte sur les paramètres du réseau :

- Régularisation $\mathcal{L}^2$ ou **Ridge** maintient les coefficients du modèle aussi petits que possible :

$$J(\theta) = \text{RisqueEmpirique}(\theta) + \lambda \frac{1}{2} \sum_{i=1}^{m} \theta_i^2$$

  où λ contrôle la qualité de régularisation souhaitée

- Régularisation $\mathcal{L}^1$ ou **Lasso** : tend à éliminer complètement les poids des variables les moins importantes ($\Rightarrow$ produit un modèle creux) :

$$J(\theta) = \text{RisqueEmpirique}(\theta) + \lambda \sum_{i=1}^{m} |\theta_i|$$

[Krogh, Hertz 1992] A simple weight decay can improve generalization

Régularisation

- Régularisation **Elastic net** (filet élastique) : compromis entre Ridge et Lasso exprimé par le paramètre $r \in [0, 1]$:

$$J(\theta) = \text{RisqueEmpirique}(\theta) + r\lambda\frac{1}{2}\sum_{i=1}^{m}|\theta_i| + \frac{1-r}{2}\lambda\sum_{i=1}^{m}\theta_i^2$$

`https://playground.tensorflow.org/`

Augmentation de base de données

Utiliser des ensembles de données de taille réduite peut induire un surapprentissage.

$\rightarrow$ 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.

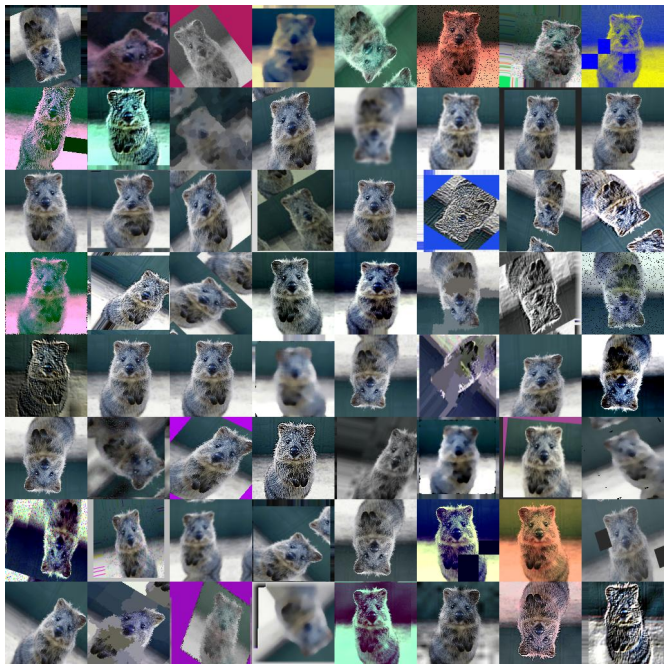


Image de `https://github.com/aleju/imgaug`

Bagging

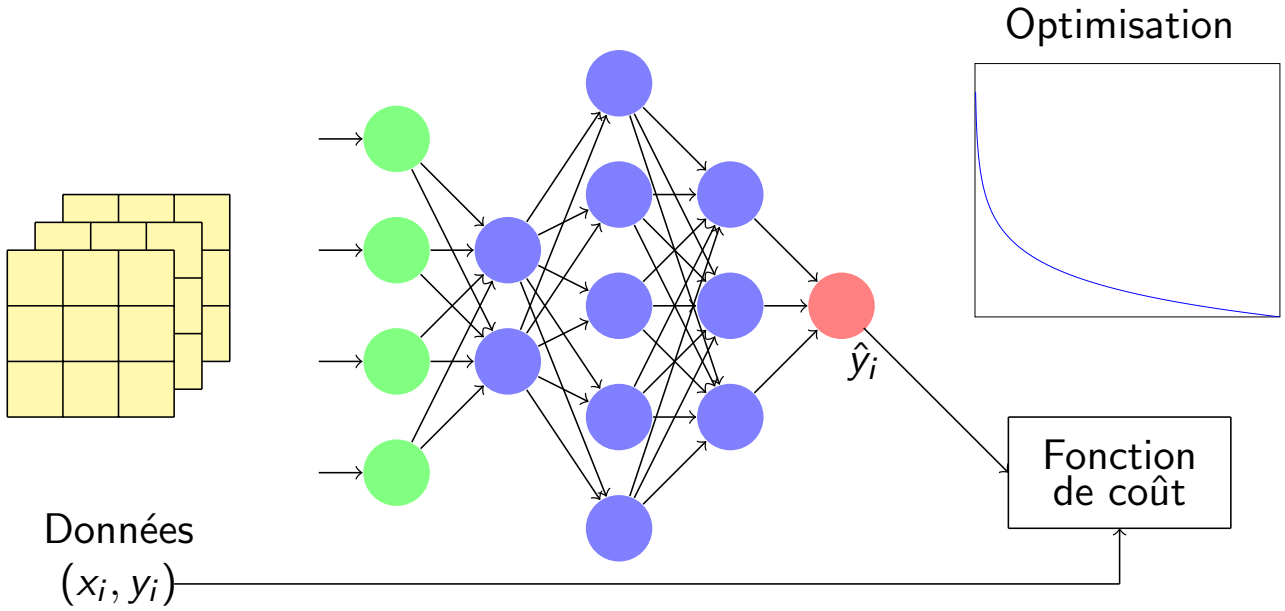
Pour réduire le risque espéré, on peut entraîner plusieurs modèles (formes de réseau) différents, et les faire voter pour dégager la prédiction la plus populaire.
L'intuition derrière cette méthode est que les différents modèles se tromperont à différentes reprises...

Plan du cours

Difficulté de l'apprentissage profond

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

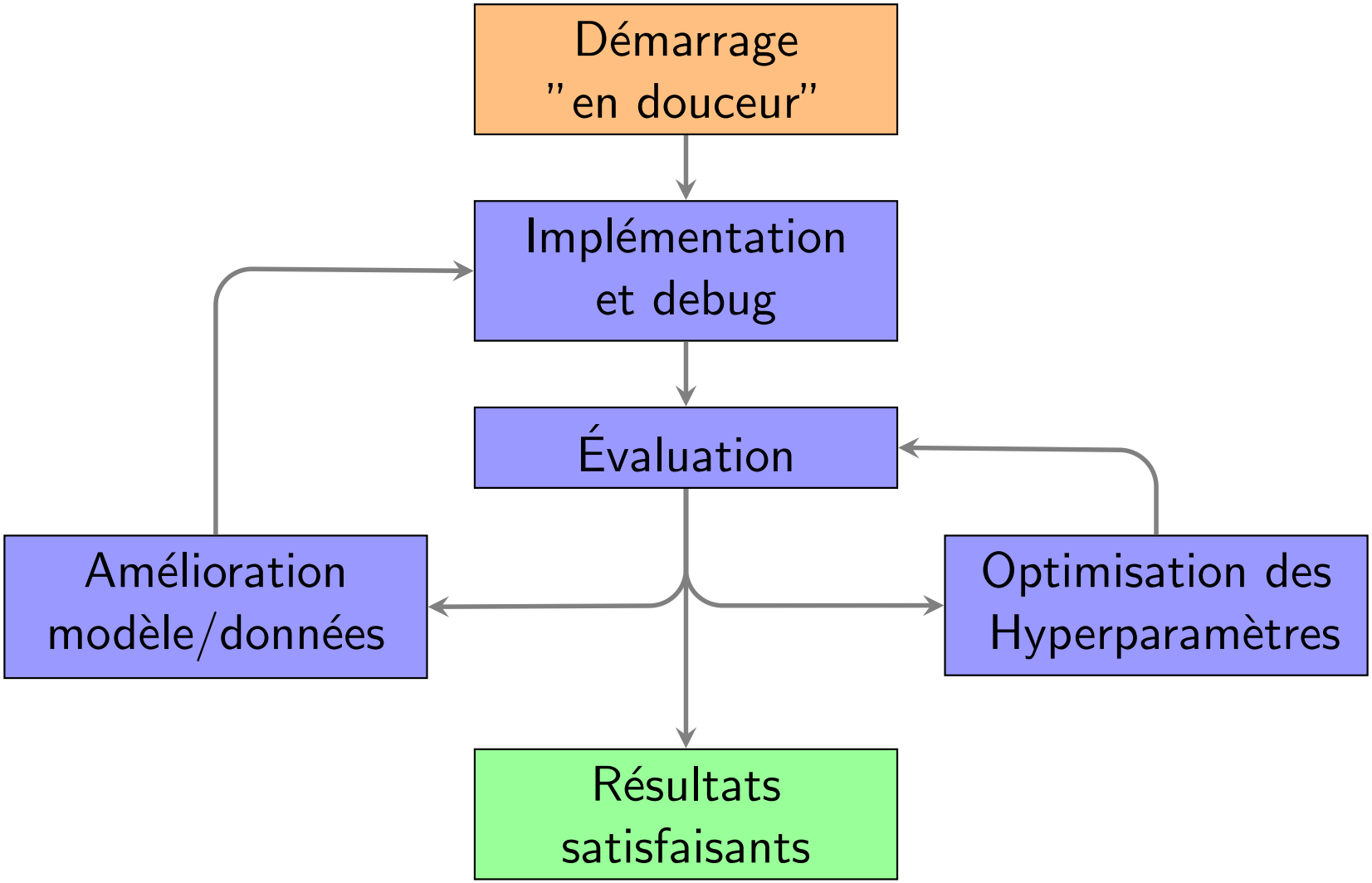
Comment s'y retrouver parmi toutes les combinaisons possibles d'architectures, d'hyperparamètres, de type de régularisation, etc. ? Tout n'est pas toujours nécessaire, ni désirable, ni aussi efficace en fonction des problèmes, des données, et du contexte.



profondeur, #neurones, activation α, momentum

$w_{i,j}, b_k$

Méthodologie générale



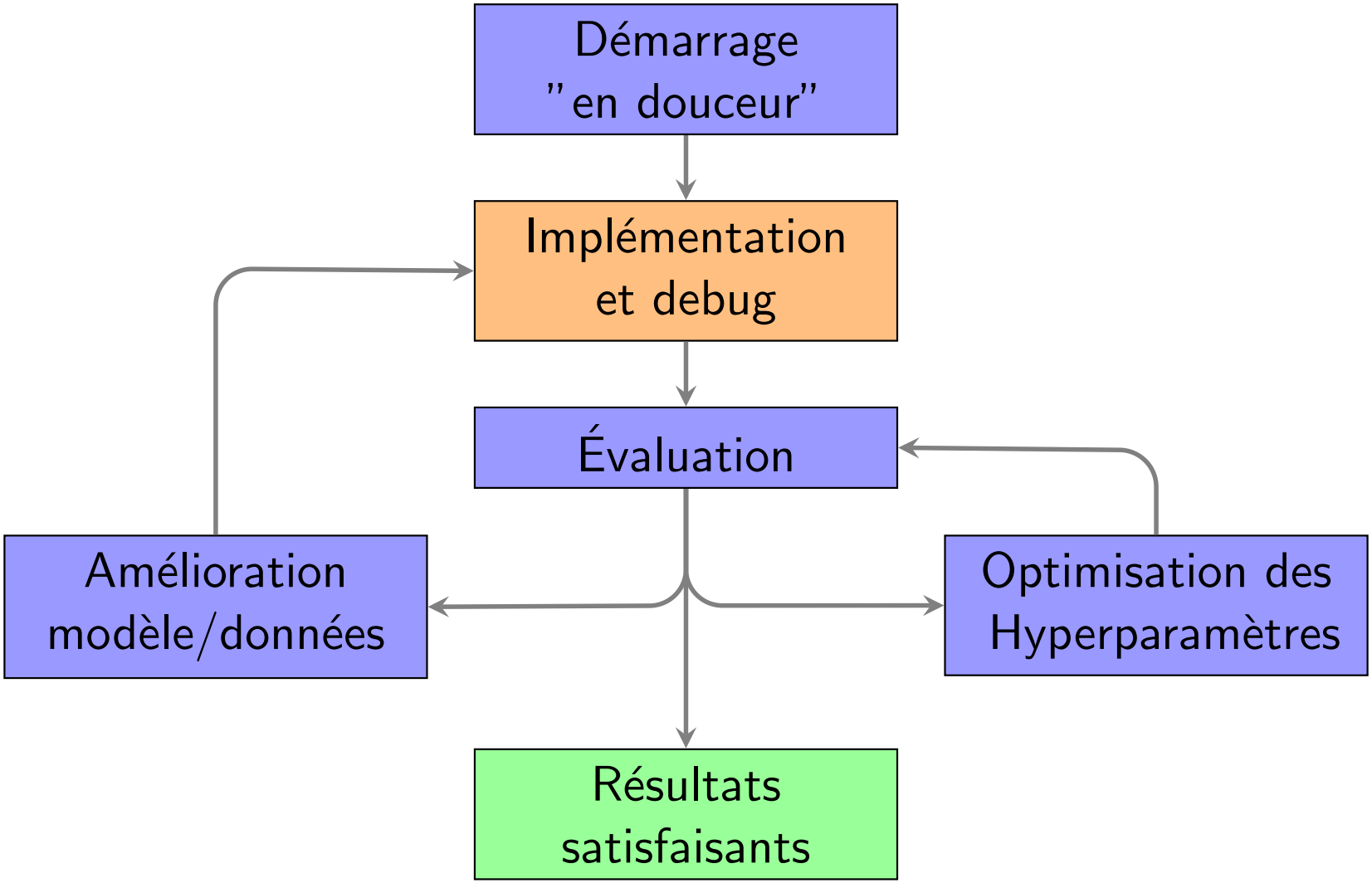
Adapté de *Troubleshooting Deep Neural Networks* de Josh Tobin

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 (avec taux d'apprentissage magique !), 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



Adapté de *Troubleshooting Deep Neural Networks* de Josh Tobin

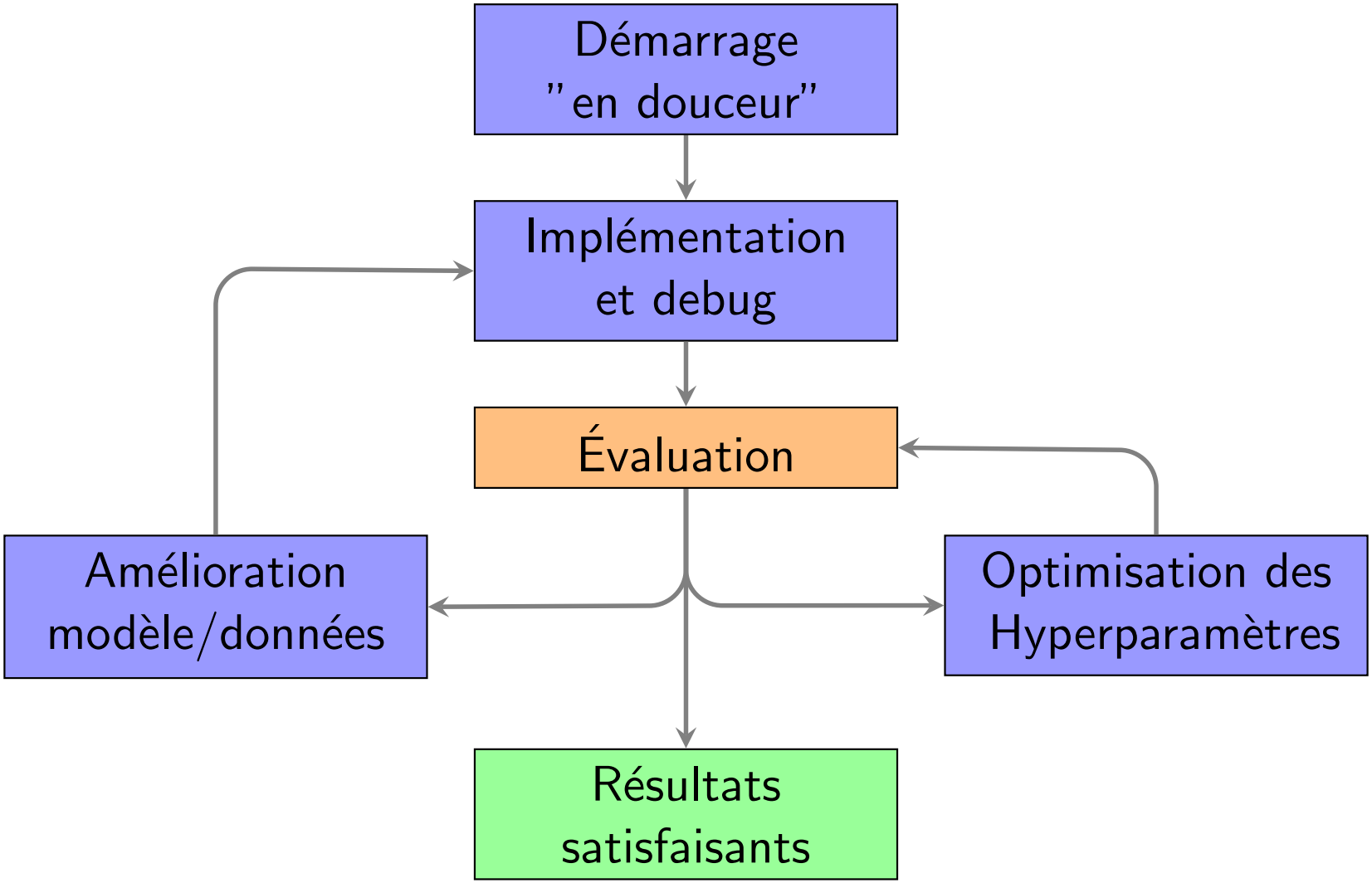
Débugger

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

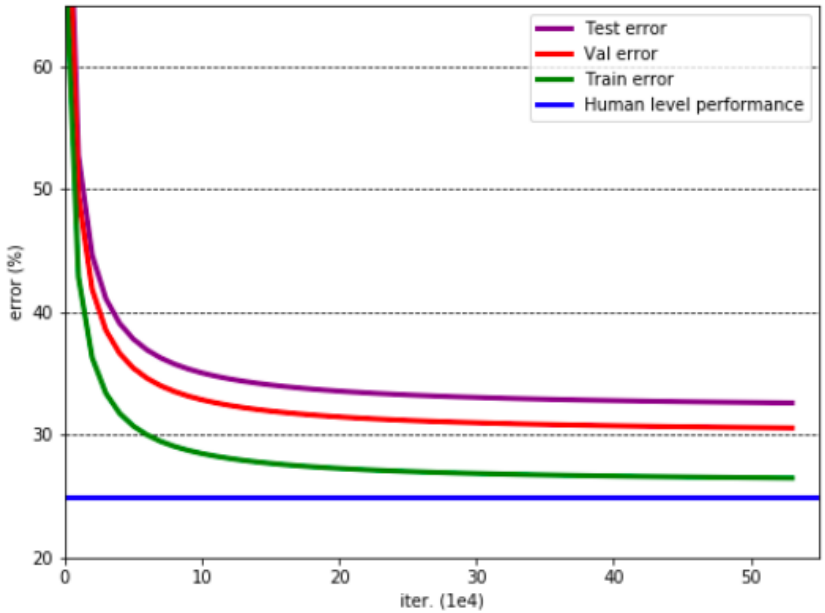
Conseil important : commencer par essayer de surapprendre un *batch* de données (voire une seule donnée)

Méthodologie générale



Adapté de *Troubleshooting Deep Neural Networks* de Josh Tobin

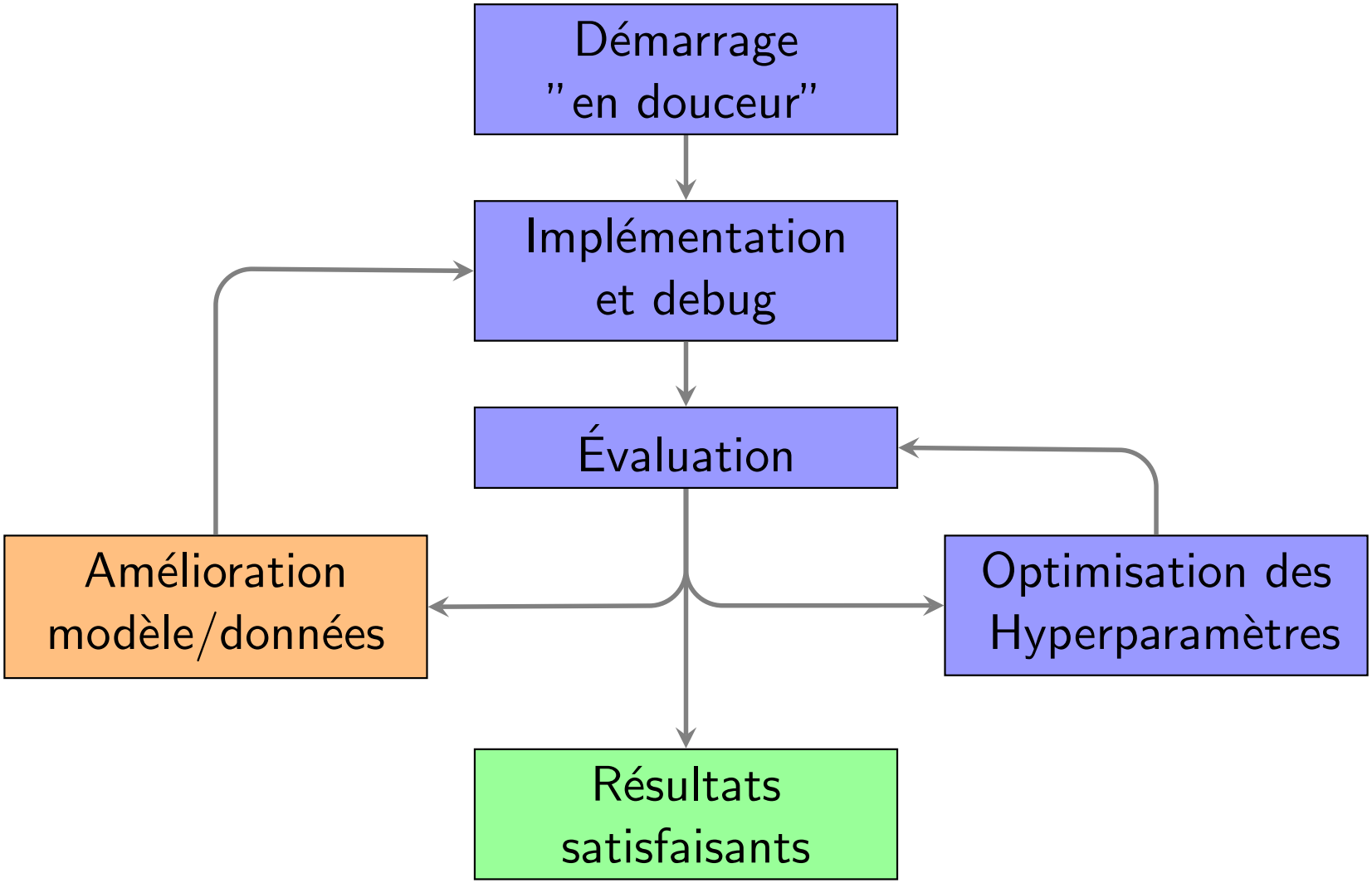
Évaluer la source des erreurs



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



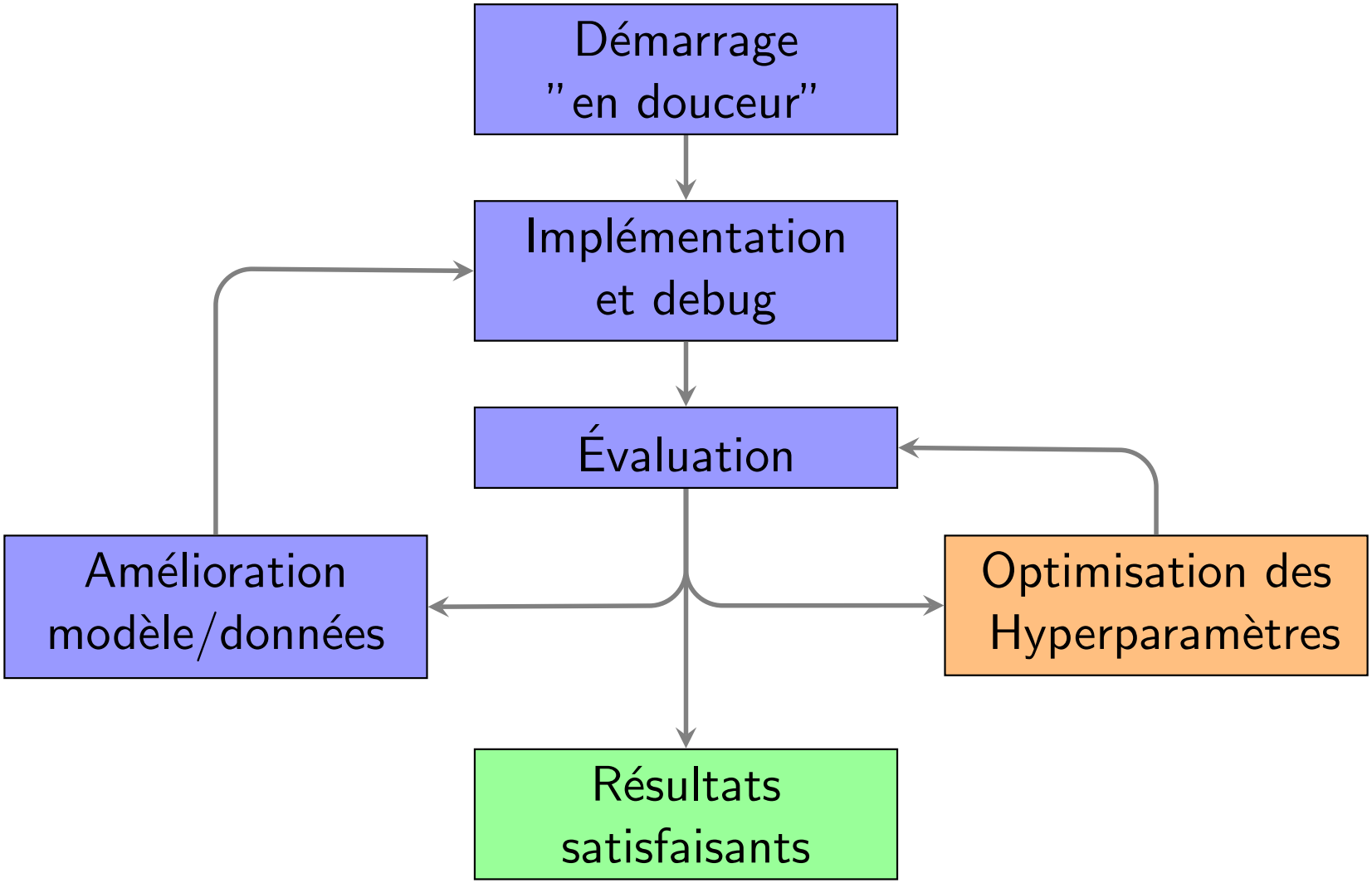
Adapté de *Troubleshooting Deep Neural Networks* de Josh Tobin

Améliorer le modèle et/ou les données

Pour corriger les problèmes mis à jour 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 !), ou augmentation des données
   Régularisation
3. Correction des problèmes de bases de donnée
   Déséquilibre de classes, distributions train/val/test différentes

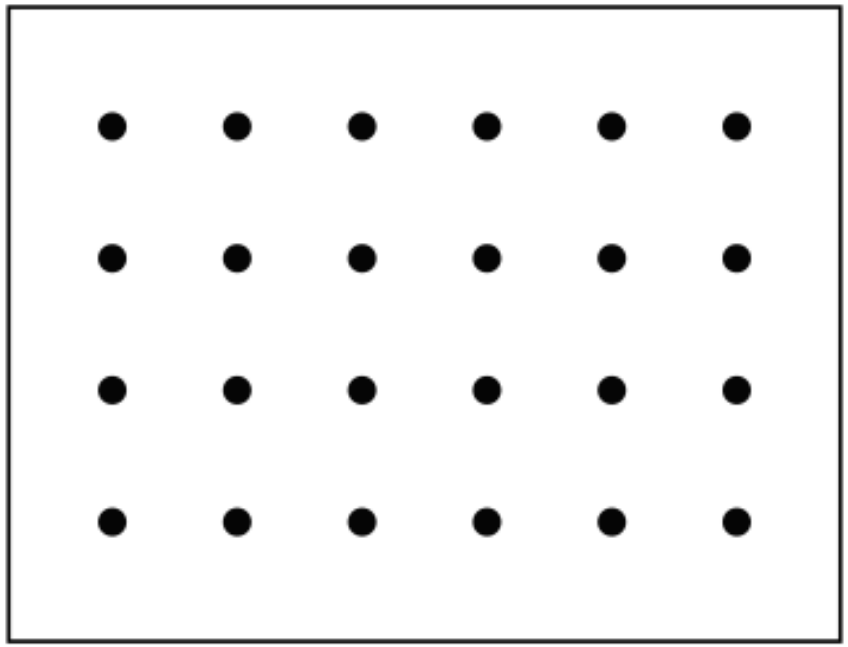
Méthodologie générale



Adapté de *Troubleshooting Deep Neural Networks* de Josh Tobin

Hyperparamètres

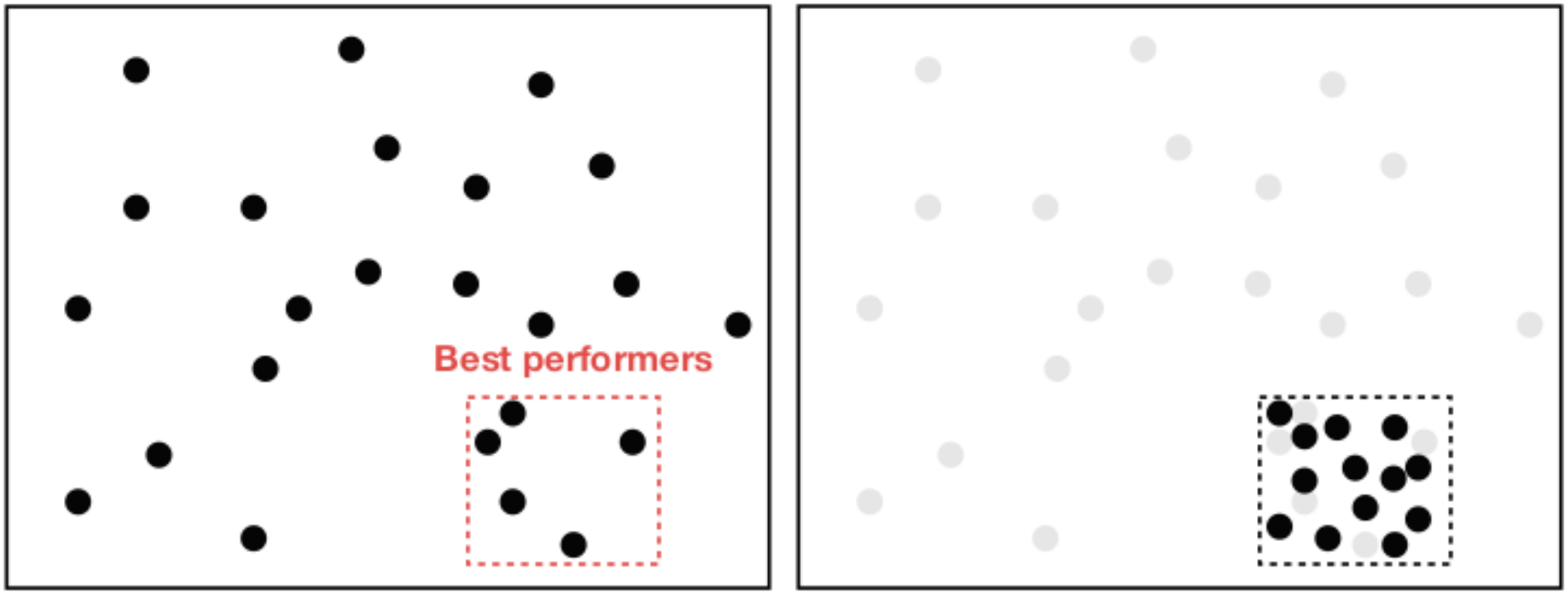
Pour mettre au point les valeurs optimales d'hyperparamètres (taux d'apprentissage, momentum, nombre de neurones par couche, etc.), une pratique commune est de déterminer un ensemble de valeurs possibles pour chaque hyperparamètre et de tester toutes les combinaisons. On conserve la combinaison qui minimise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

Visualisation

https://playground.tensorflow.org/