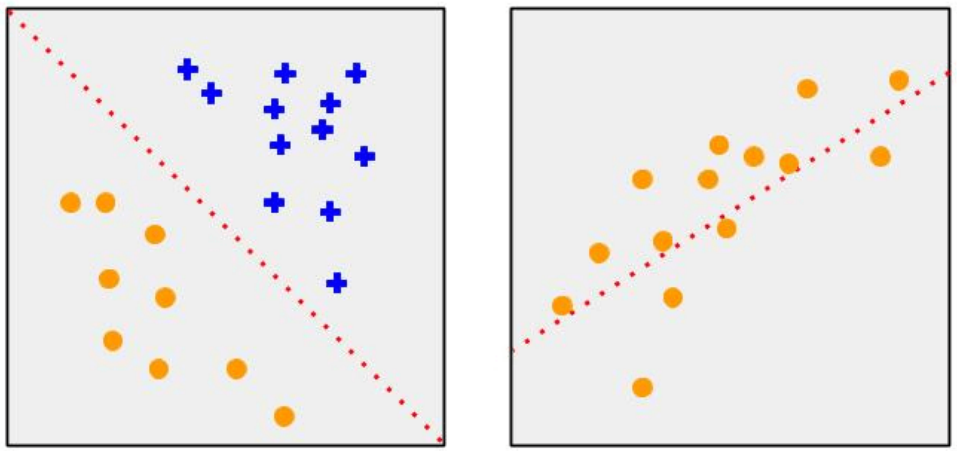
Apprentissage supervisé

Partie 2

Analyse de données et Classification 2
ENSEEIHT - 3ème année Sciences du Numérique

Contact :
Sandrine.Mouysset@irit.fr
sandrine.mouysset@toulouse-inp.fr

2 principaux types d'apprentissage supervisé :



Classification | Regression

Approches par modèles supervisés :

- Arbres de Décision
- Apprentissage d'ensemble : Forêts aléatoires
- Réseaux de neurones
- Ouverture sur le deep learning

Méthodes d'évaluation :

- Validation croisée
- Matrice de confusion
- Precision, Rappel, F-mesure
- Courbe ROC

Perceptron mono-couche :

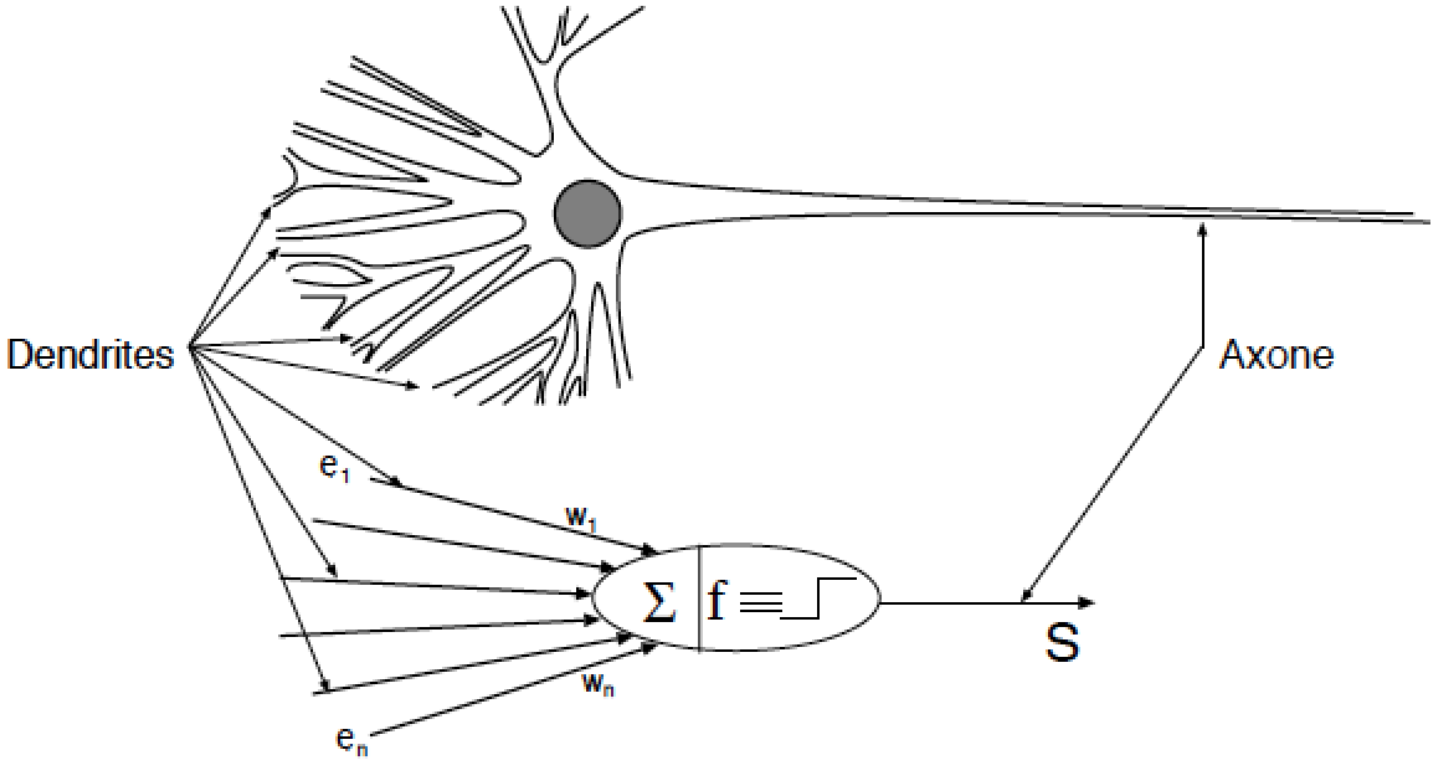
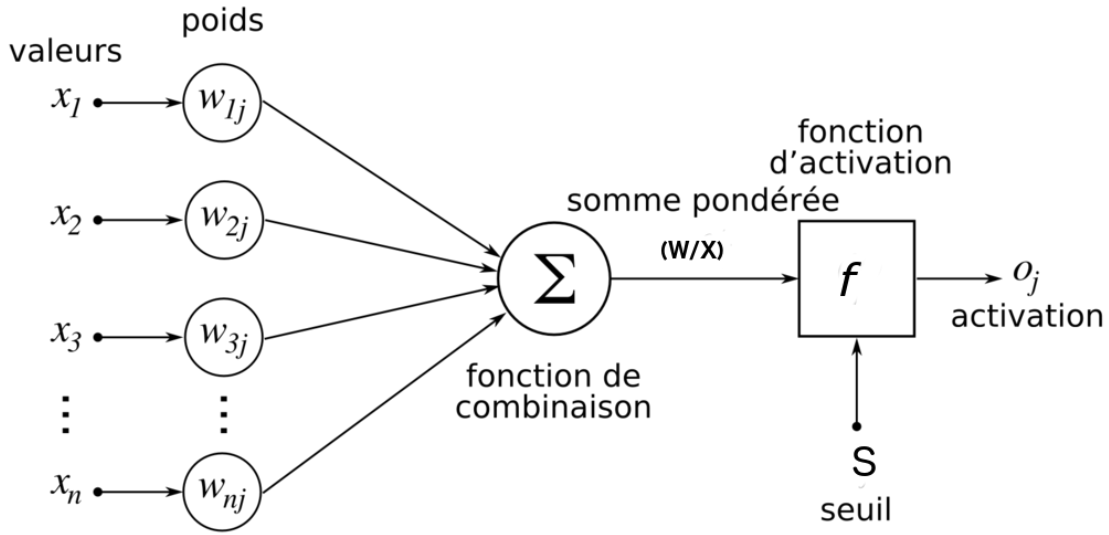


Figure: Structure d'un neurone biologique/artificiel

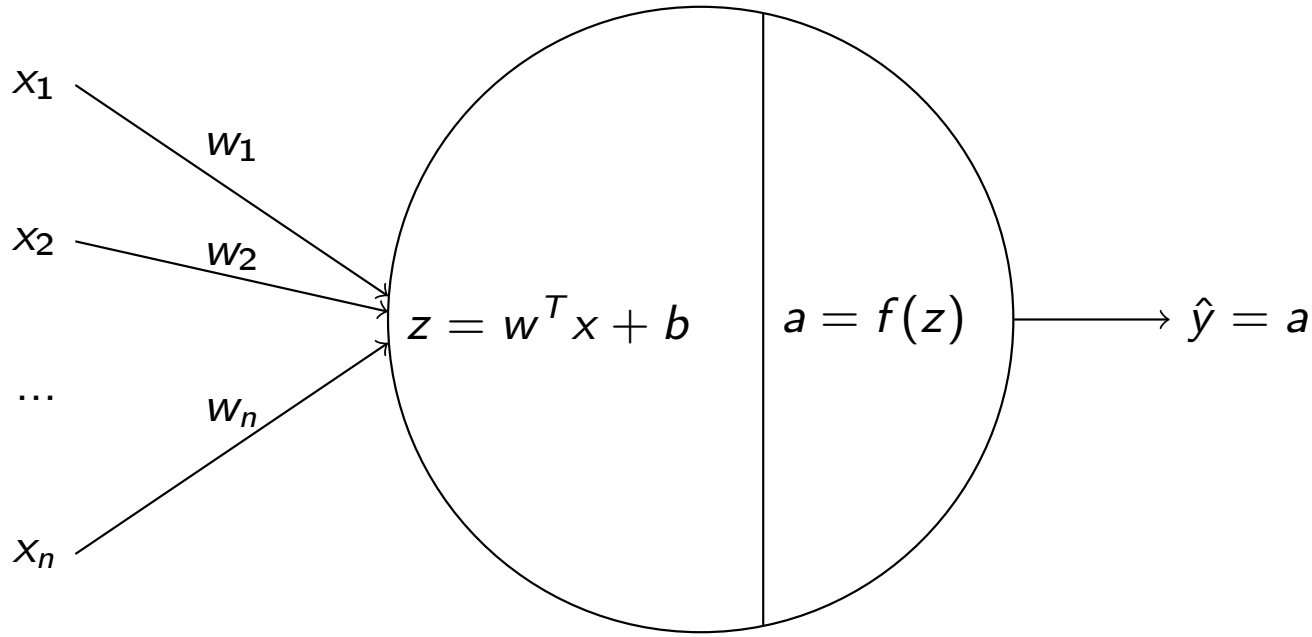
Fonctionnement :



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 : $a = f(z)$

Représentation d'un "neurone"



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 : $a = f(z)$

Fonction d'activations

Les **fonctions d'activation**, notées f, sont des fonctions de seuillage qui peuvent souvent se décomposer en trois parties:

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

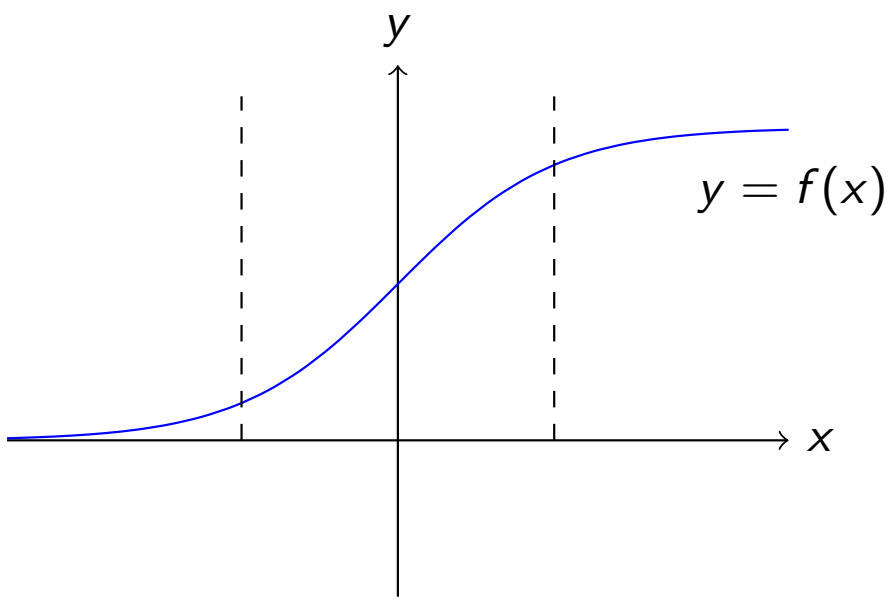
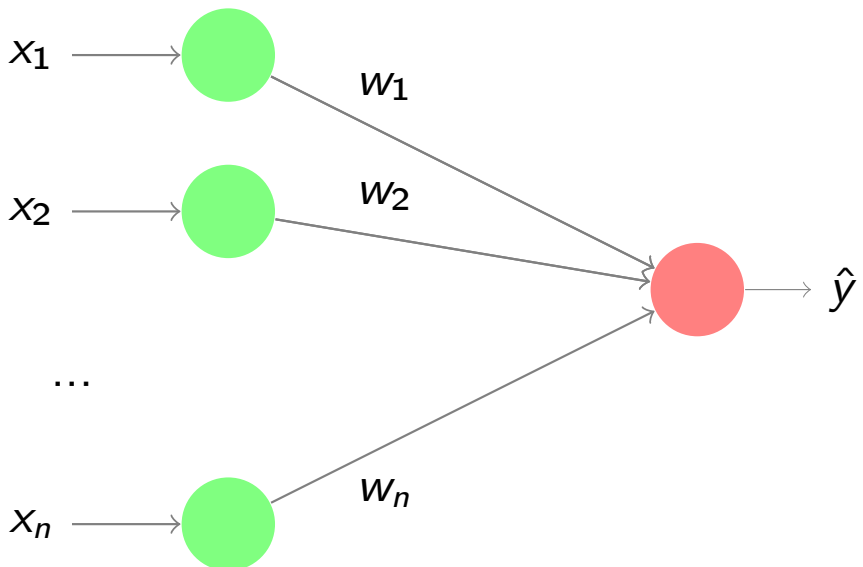


Figure: Exemple de fonction d'activation avec la fonction sigmoïde : $f(x) = \frac{1}{1+e^{-x}}$

Rappel des notations

Entrées $x^{[1]}, \ldots, x^{[n]}$



Algorithm 1 Algorithme du perceptron monocouche

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

$$\hat{y}^{[j](t)} = f\left(\sum_{i=1}^{n} w_i^{(t)} x_i^{[j]}\right) \text{ et } J = \sum_{j=1}^{m} \text{perte}(\hat{y}^{[j](t)}, y^{[j]})$$

4. Mise à jour des poids

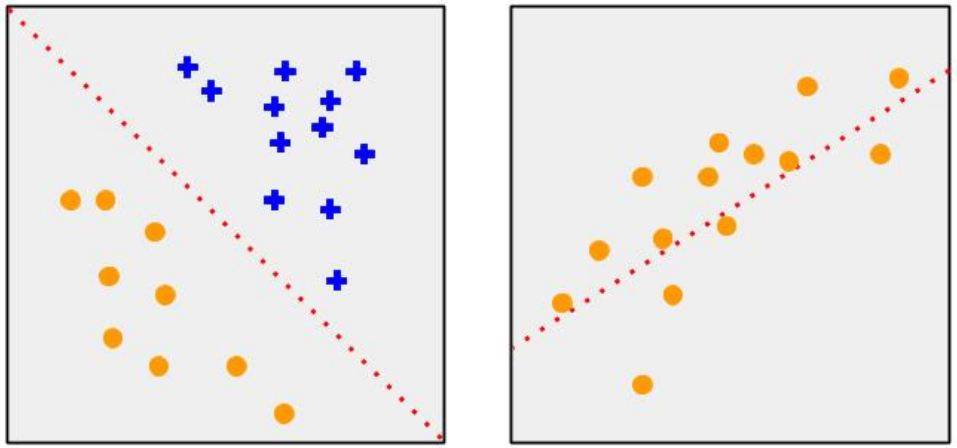
$$w_i^{(t+1)} = w_i^{(t)} - \alpha \frac{\partial J}{\partial w_i}$$

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

5. Retourner en 2 jusqu'à la convergence (c'est-à-dire $\hat{y_j}^{(t)} \approx y_j$).

$\Rightarrow$ Comment définir la **fonction de coût** ?

2 principaux types d'apprentissage supervisé :



Classification Regression

Classification ou Régression logistique: Assigner une catégorie à chaque observation

Cas binaire : échec/succès, 0/1

- fonction sigmoïde : $f(x_j) = (1 + e^{-x_j})^{-1}$
- Fonction de perte logistique ou entropie croisée (cross-entropy):

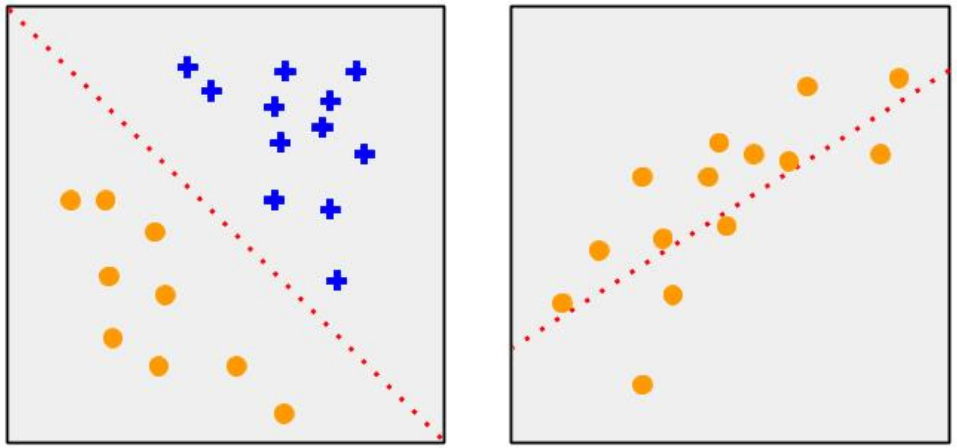
$$J = -y^{[j](t)} log(\hat{y}^{[j]}) - (1 - y^{[j](t)}) log(1 - \hat{y}^{[j]})$$

Cas multinomial (multi-classe) : identifier les chiffres

- fonction softmax : $f(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$
- fonction de perte logistique :

$$J = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{k} \mathbb{I}(y^{(i)} = j) log\left(\hat{y}^{[j]}\right)$$

2 principaux types d'apprentissage supervisé :



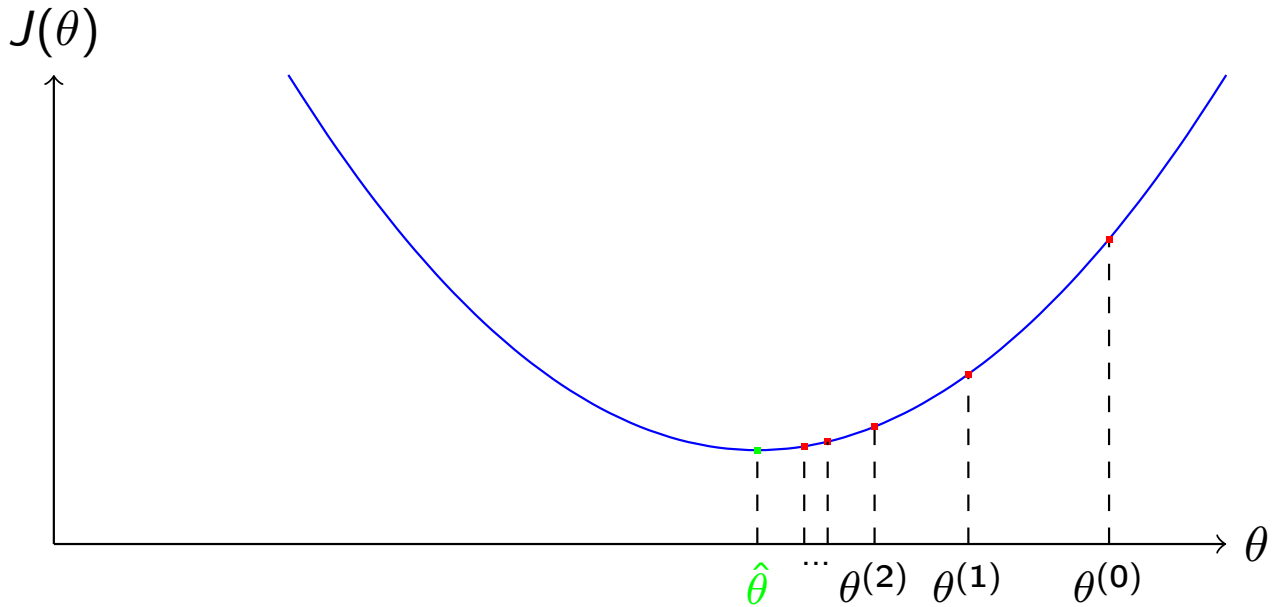
Régression : Prédire une valeur réelle à chaque observation :

- fonction linéaire : $f(x_j) = x_j$
- fonction de coût quadratique (MSE) :

$$J = \sum_{j=1}^{m} (\hat{y}^{[j](t)} - y^{[j]})^2$$

$\Rightarrow$ Comment **résoudre ce type de problème** ?

Descente de gradient:



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

1: Initialiser $\overrightarrow{\theta} \leftarrow \overrightarrow{0}$
2: TANT QUE pas convergence FAIRE
3: POUR k de 1 à d FAIRE
4: $\theta_k \leftarrow \theta_k - \alpha \frac{\partial J(\theta)}{\partial \theta_k}$

Version Régression :

- Fonction d'activation linéaire
- Fonction de coût quadratique :

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

- Descente de gradient :

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

$$\omega_j \leftarrow \omega_j - \alpha (\hat{y}_j - y_j) x_i$$

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

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

Version Classification :

- Fonction d'activation sigmoïde
- Fonction de coût entropie croisée :

$$J = -y_i log(\hat{y}_i) - (1 - y_i) log(1 - \hat{y}_i)$$

- Descente de gradient :

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

$$\omega_j \leftarrow \omega_j - \alpha(\hat{y}_j - y_j)x_i$$

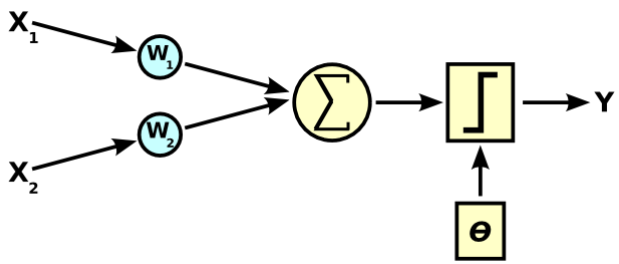
où α est le taux d'apprentissage (fixe ou variable).

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

Exercice On considère le jeu de données 2D suivant dont on associe une étiquette Y :

x_1	x_2	Y
2	1	1
0	-2	1
-2	1	-1
0	2	-1

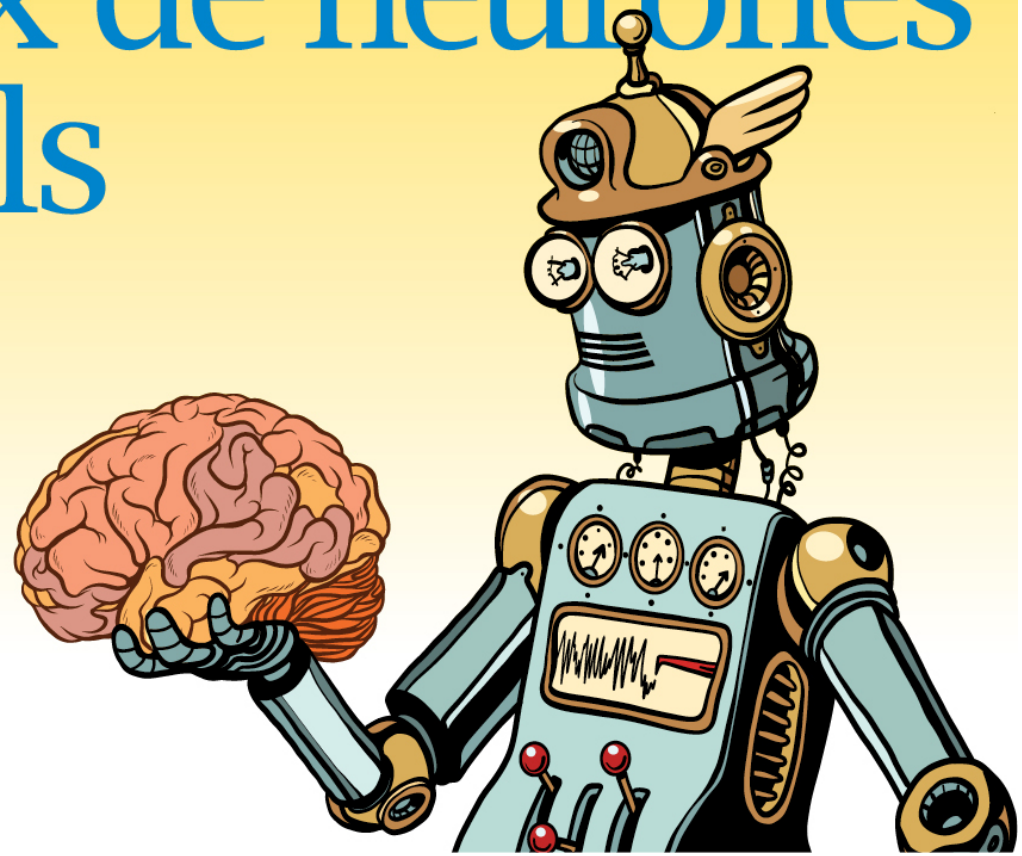
On étudie ces données à l'aide d'un perceptron monocouche :

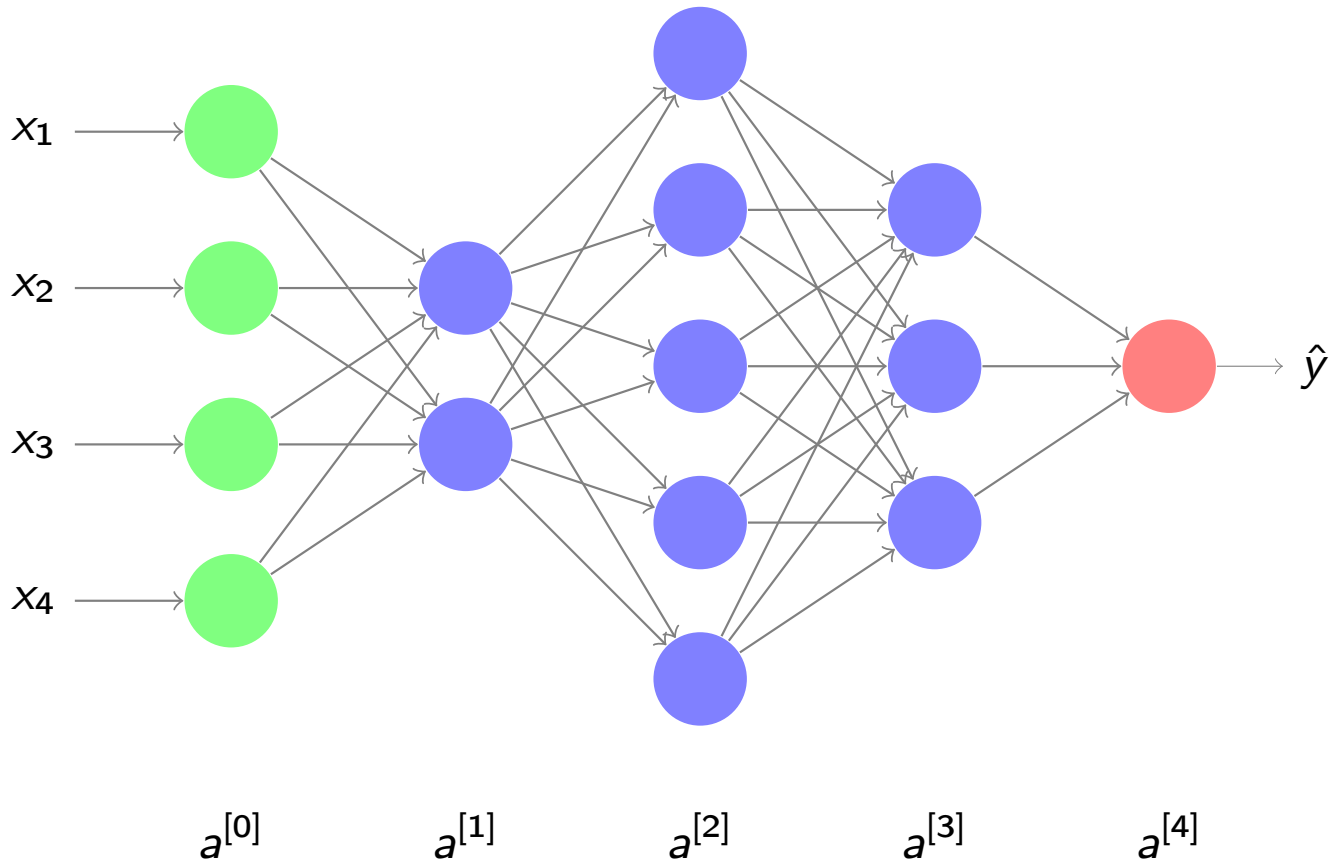


- la fonction d'activation est la fonction signe : $f(x) = -1$ si $x \leq 0, f(x) = 1$ sinon ;
- les paramètres de poids initiaux sont : $w_1 = -0.7,\ w_2 = 0.2$;
- le paramètre de mise à jour des poids est $\eta = 0.5$;
- le seuil (ou biais) est $\theta = 0.5$.

1. Calculer les poids w_1 et w_2 tels que le perceptron vérifie le base d'apprentissage.
2. Représenter les données et l'hyperplan séparateur.

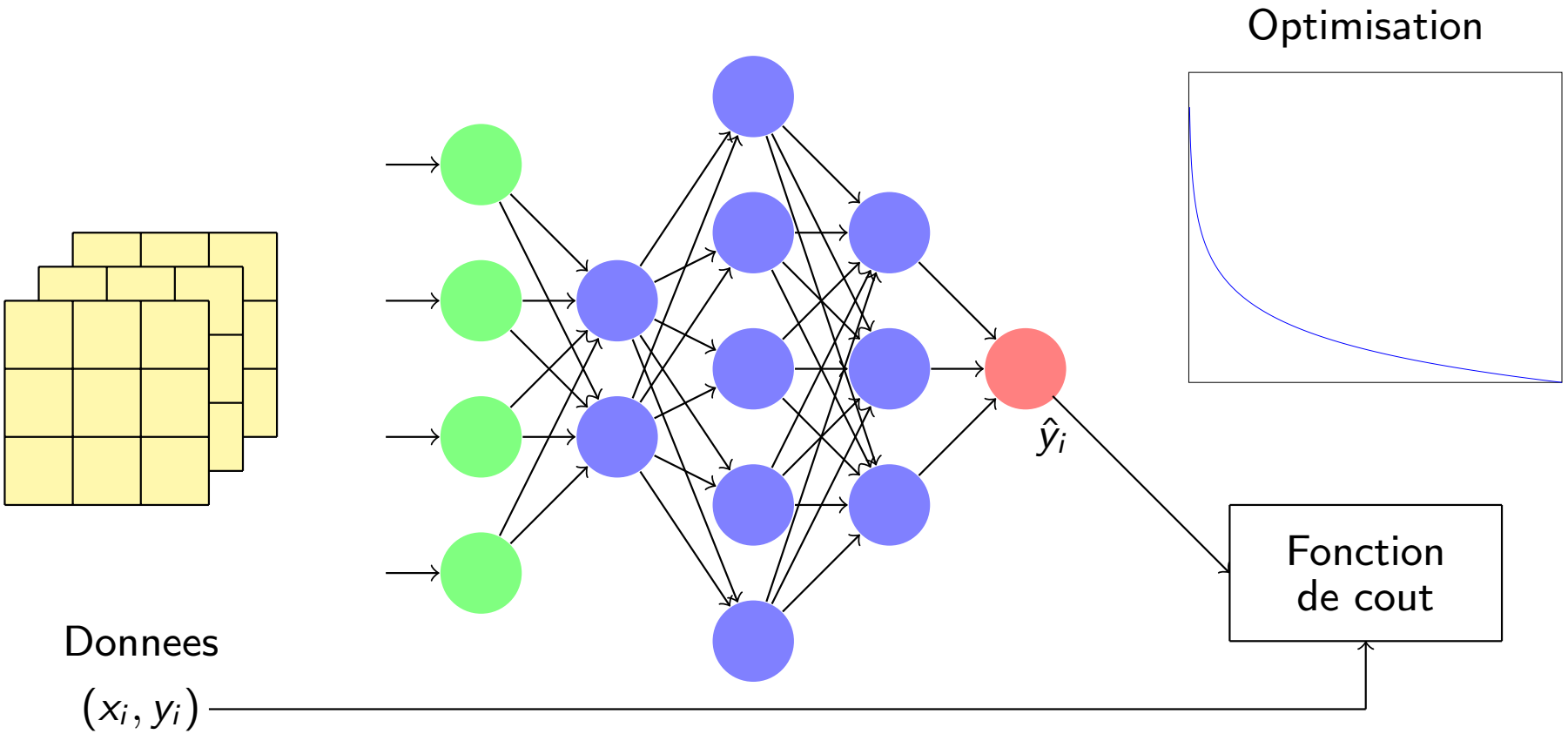
Réseaux de neurones artificiels





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.

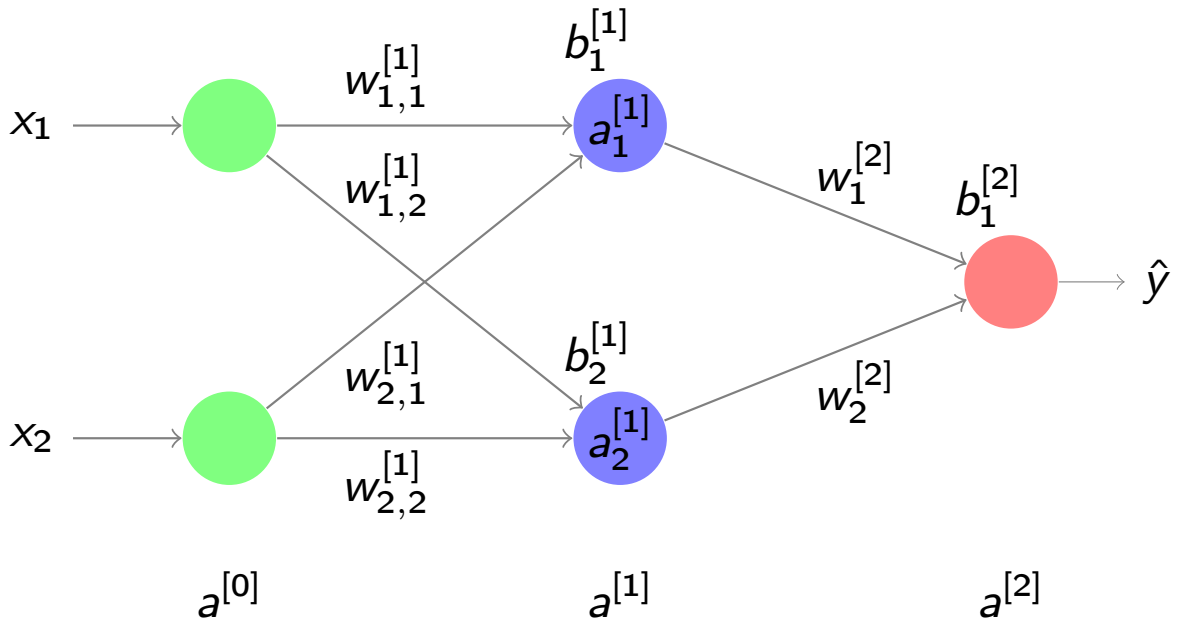
Vue d'ensemble



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

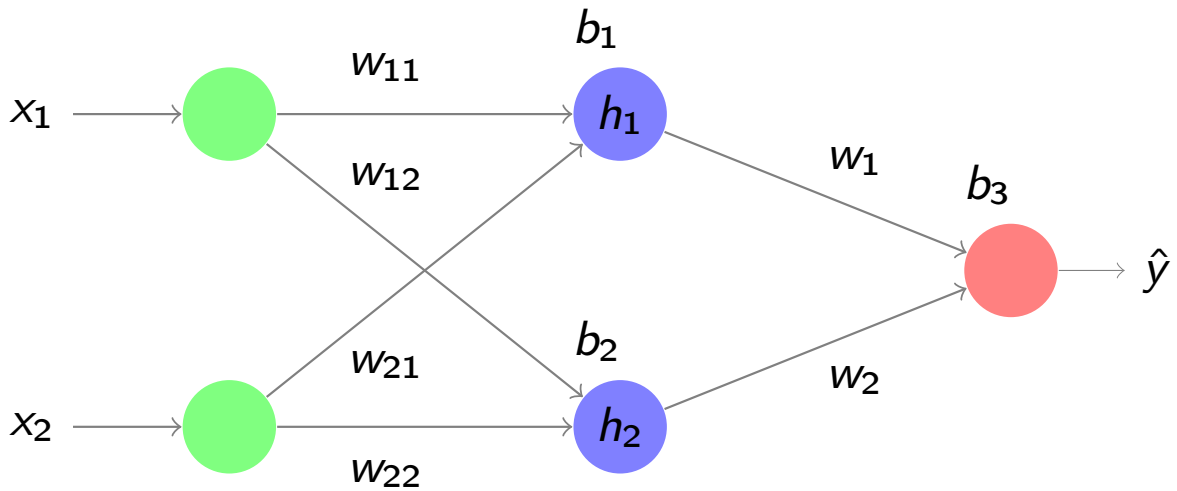
1. **Propagation des données** de la couche d'entrée à la couche de sortie;
2. Calcul de l'**erreur de sortie** après la propagation des données;
3. Calcul des **gradients d'erreurs** pour corriger les poids synaptiques des neurones de la **couche de sortie**;
4. Calcul des gradients d'erreurs pour corriger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

Illustration de l'entraînement d'un perceptron multicouche

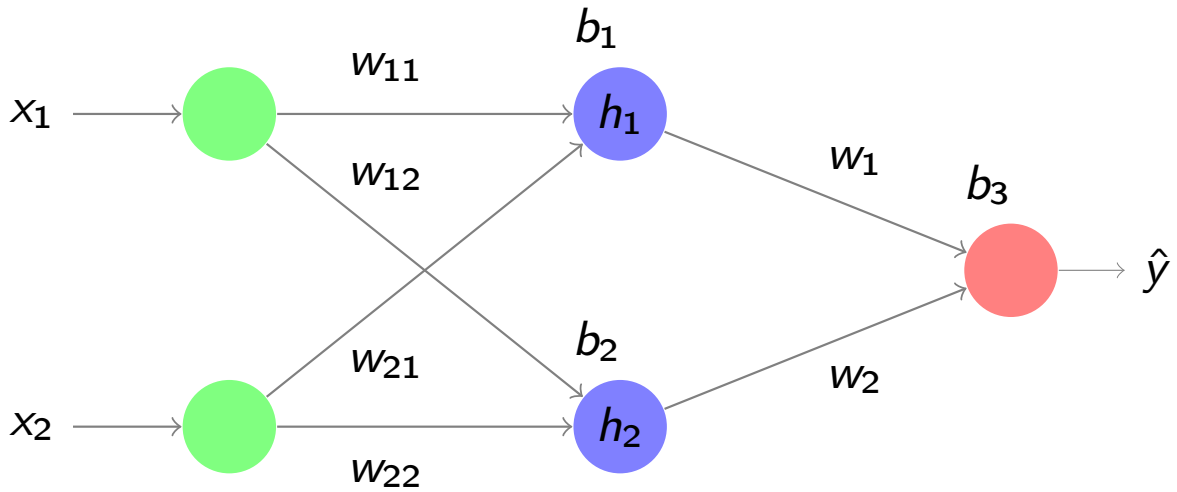


En simplifiant un peu les notations...

Illustration de l'entraînement d'un perceptron multicouche



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



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

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

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

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

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

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 :

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

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

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

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

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial h_j} \frac{\partial h_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}},$$
$$\frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial h_i} \frac{\partial h_i}{\partial z_i} \frac{\partial z_i}{\partial b_i},$$

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

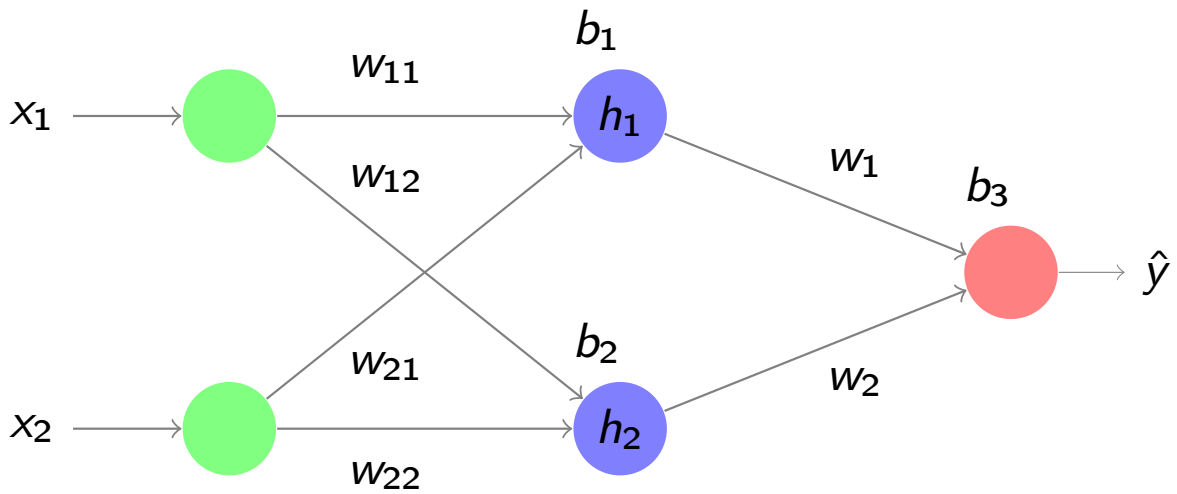
5) **Mise à jour** des poids synaptiques de la couche de sortie :

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

et de la couche cachée :

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

Retour sur le calcul des gradients



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

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

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

On a :

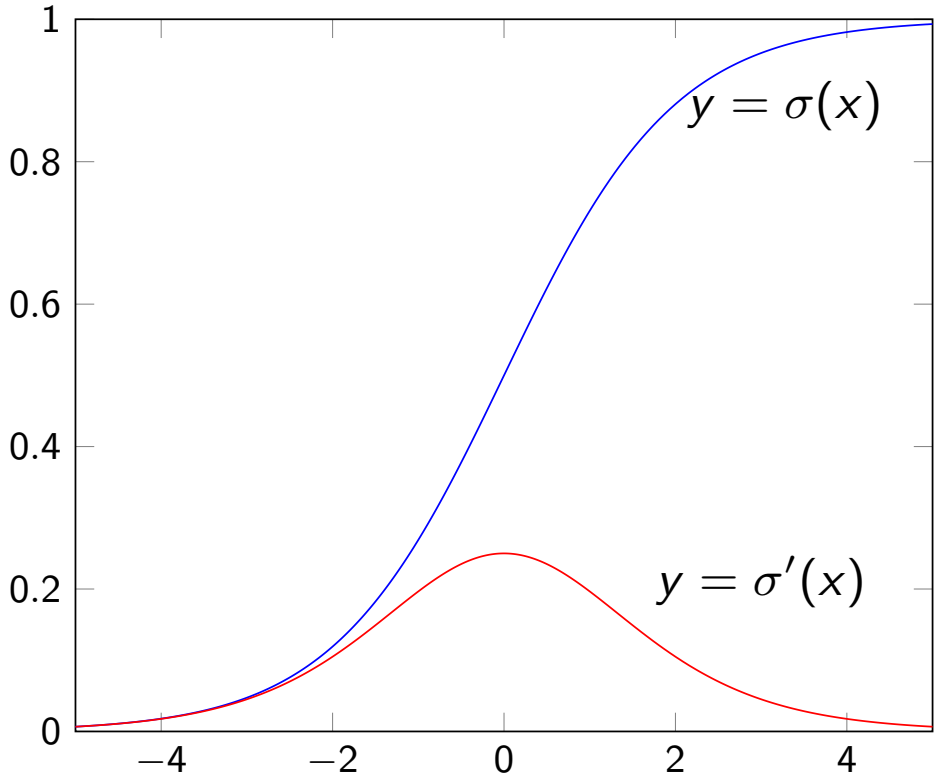
$$\frac{\partial z}{\partial h_j} = w_j$$

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

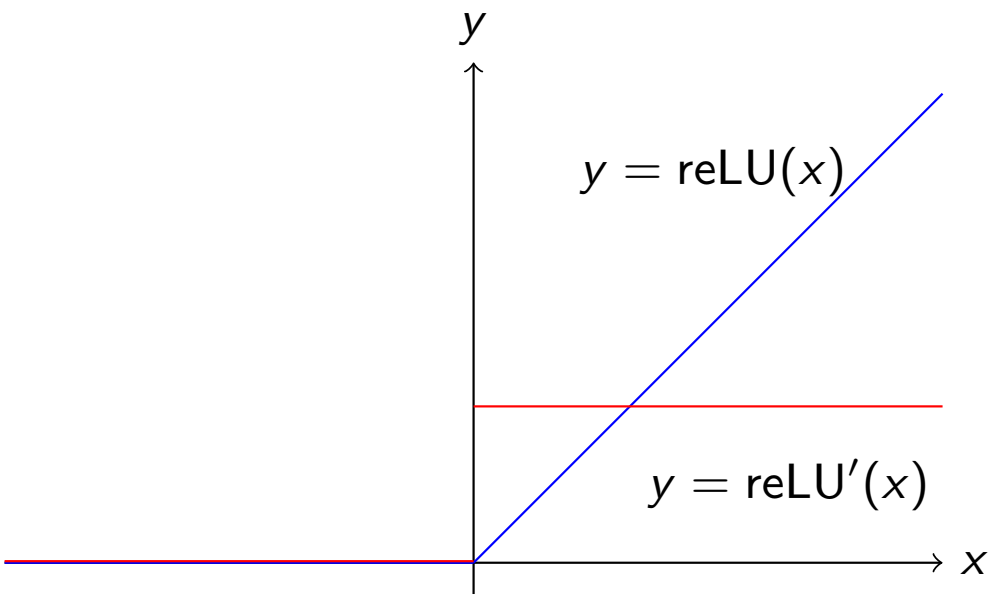
avec f' la dérivée de la fonction d'activation portée par le neurone.

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

Gros plan sur la fonction sigmoïde

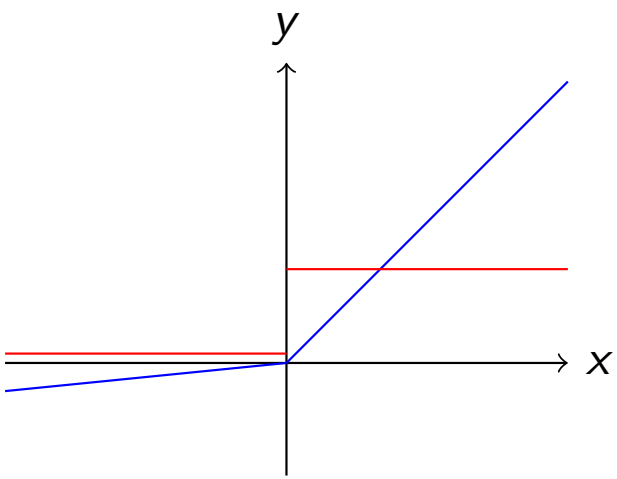


Le gradient de la fonction sigmoïde est très souvent proche de 0, ce qui empêche la propagation du gradient. C'est le problème de l'**évanescence des gradients** (*vanishing gradients*).



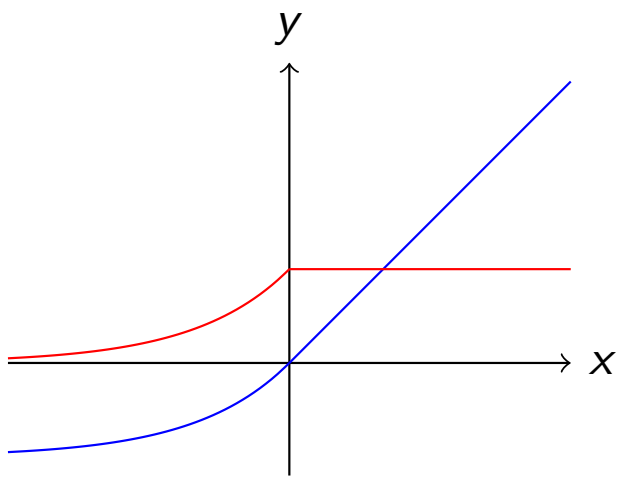
$$\text{reLU}(x) = \begin{cases} 0 & \text{si} \quad x < 0 \\ x & \text{sinon} \end{cases} \tag{1}$$

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



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

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

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.

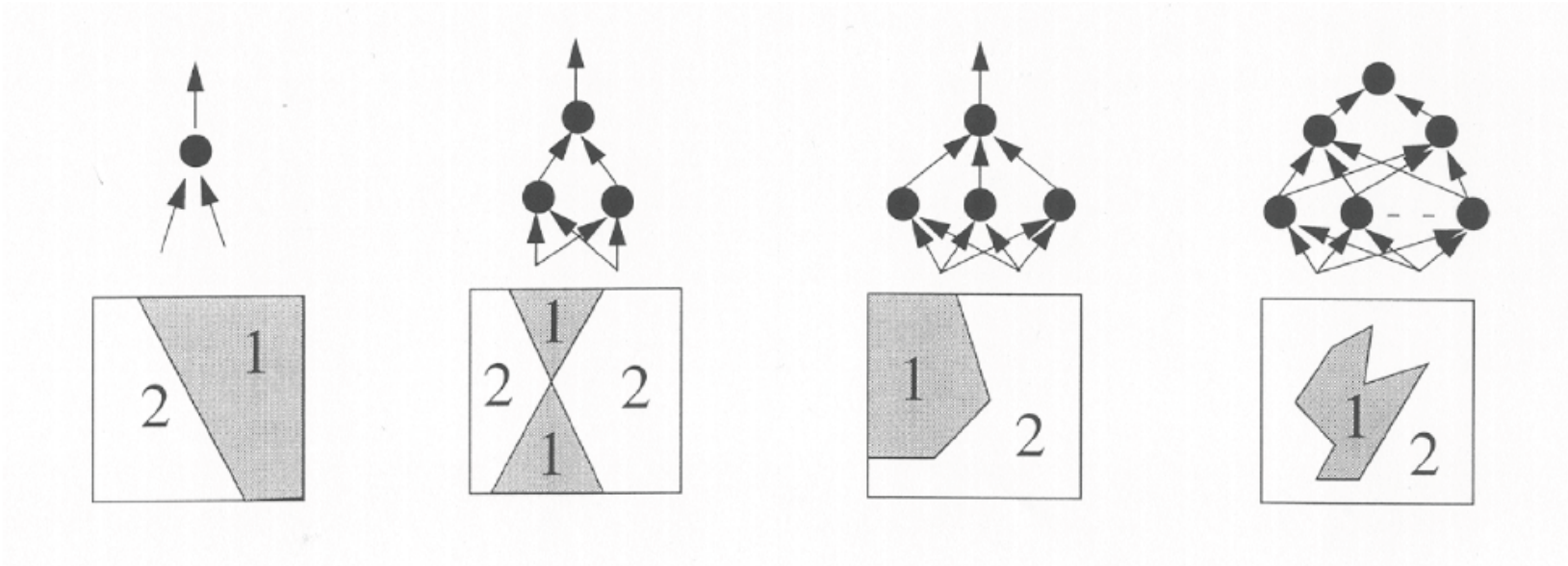


Figure: Intérêt du perceptron multicouche : pouvoir séparateur

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

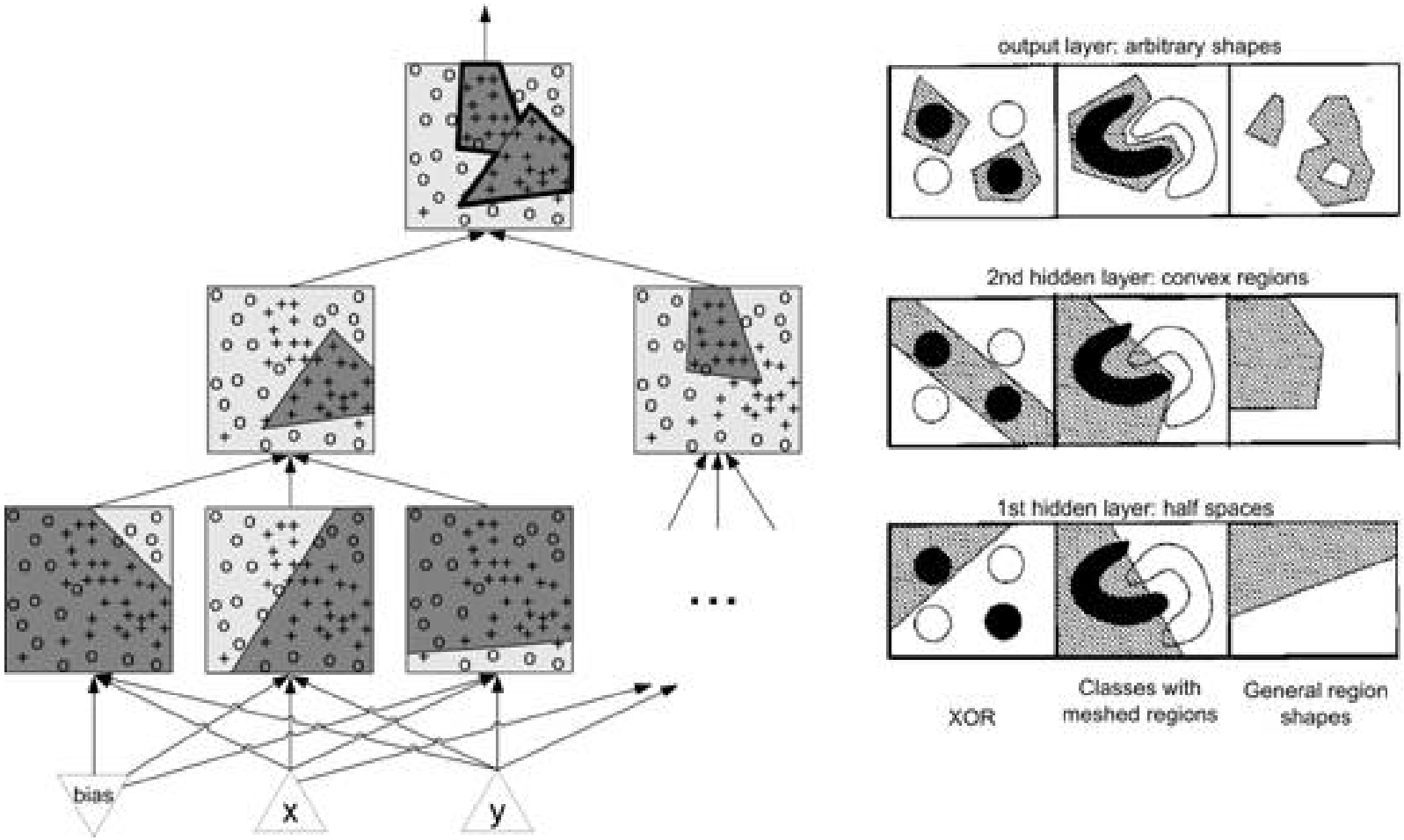
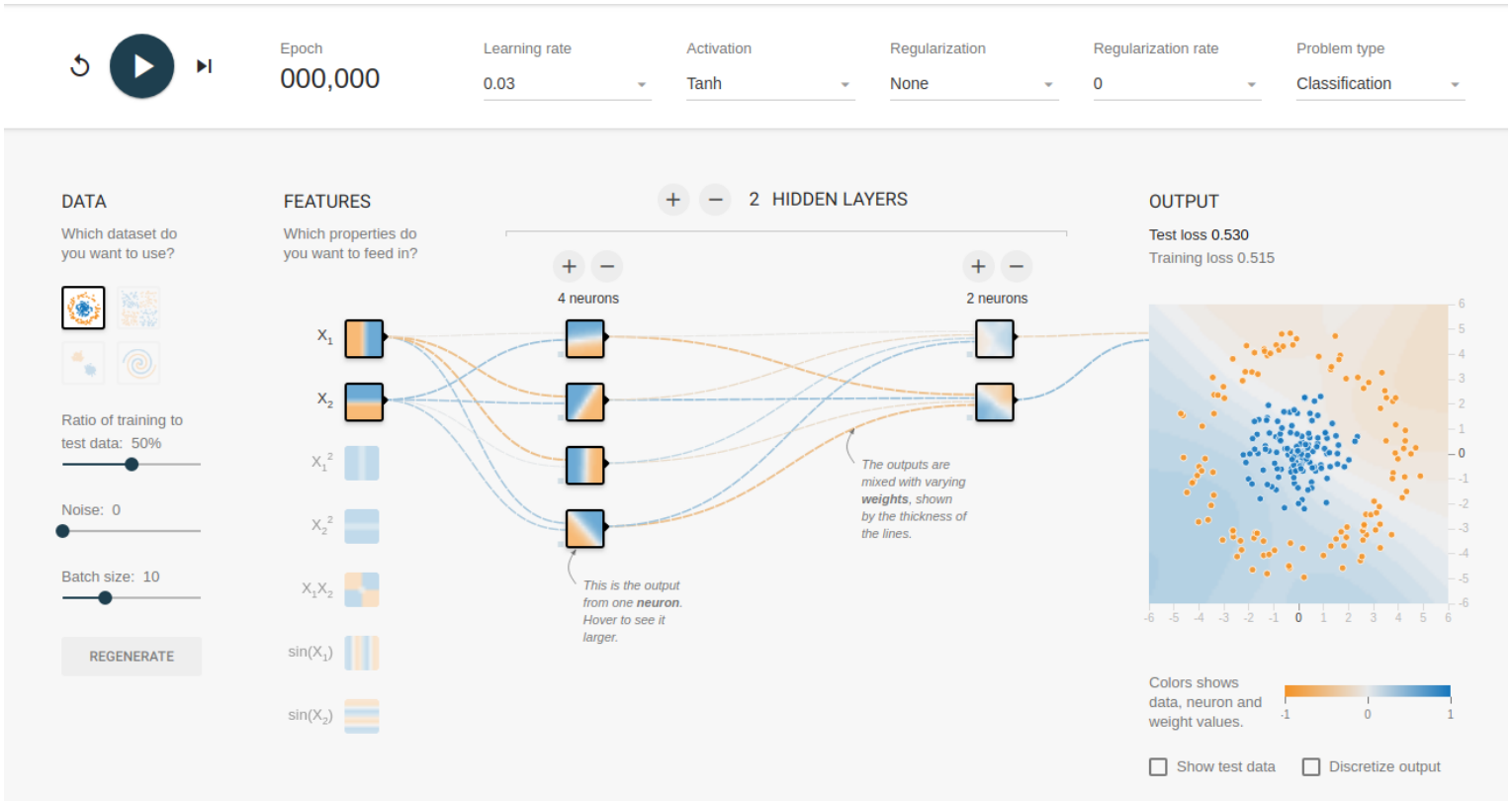


Figure: Intérêt du perceptron muticouche

https://playground.tensorflow.org/



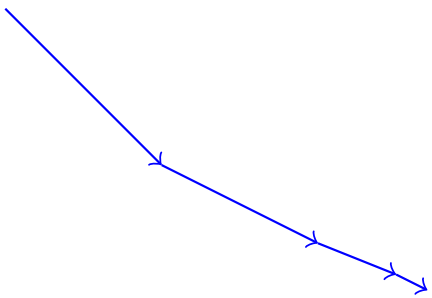
Calculer le gradient exact est coûteux parce qu'on doit évaluer le modèle sur tous les m exemples de l'ensemble de données.

Il existe des alternatives :

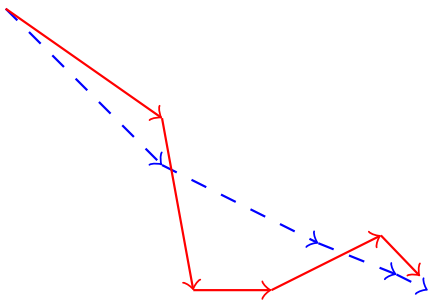
- Algorithme **Batch** : utilise tous les m exemples de l'ensemble d'apprentissage pour calculer le gradient.
- Algorithme **MiniBatch** : approxime le gradient en le calculant en utilisant k exemples parmi les m échantillons de l'ensemble d'apprentissage, où $m >> k > 1$.
- Algorithme **Stochastique** : 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 MiniBatch.

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*.



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.

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*.

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

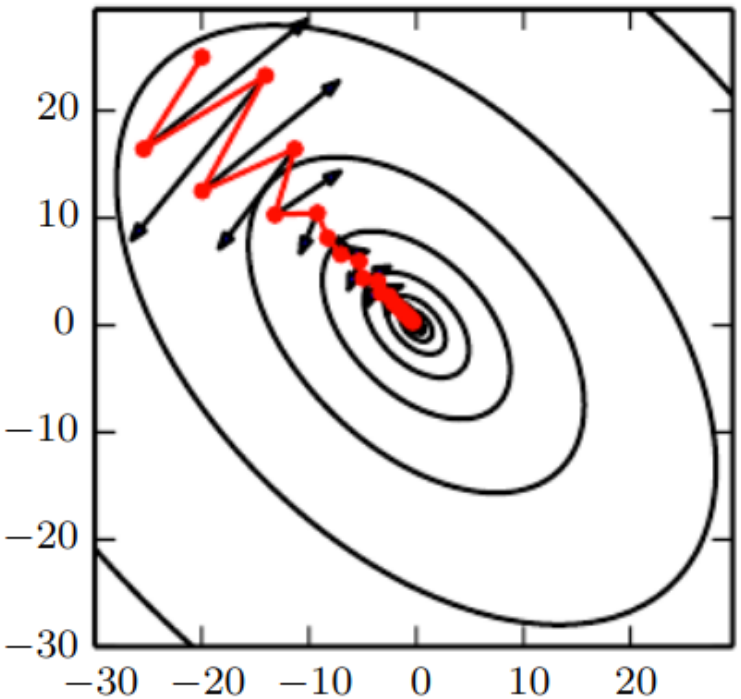


Image de [Goodfellow et al. 2015] Deep Learning

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.

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.

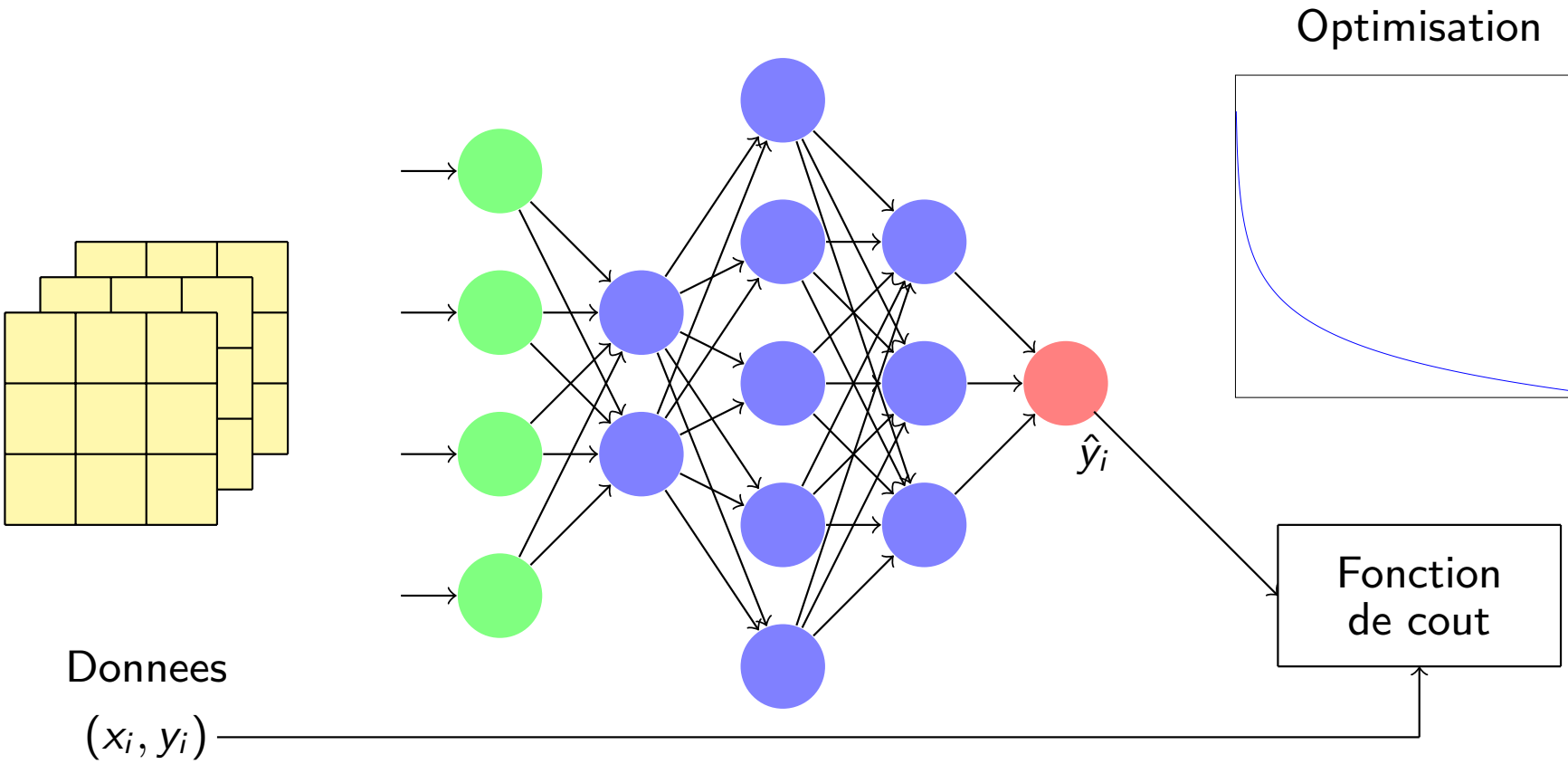
L'optimiseur **AdaGrad** introduit une forme d'adaptation du taux d'apprentissage en accumulant les carrés des gradients précédents.

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

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 améliorer le comportement de l'algorithme dans me 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.



profondeur, #neurones, activation α, momentum

$w_{i,j}$, b_k

hyperparamètres et paramètres