



Introduction au Machine Learning

Séance 10 — Les réseaux de neurones : du neurone à la rétropropagation

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

<https://ebtoure.sn/cours/introduction-machine-learning/chapitres/10-reseaux-neurones.html>

Version du 11 octobre 2026



Version en ligne

Introduction au ML — Séance 10

Les réseaux de neurones : du neurone à la rétropropagation

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

Objectifs de la séance

Contenu

Le **réseau de neurones** est le modèle le plus emblématique de l'IA moderne. On le construit ici *brique par brique*, sans précipitation : un neurone seul, puis une couche, puis un réseau ; les poids, la matrice de poids, le biais. À chaque étape, on relie l'*image*, le *formalisme* et les *chiffres*.

- Le **neurone** : un objet déjà connu (la régression logistique de la Séance 6), formalisé et calculé sur des chiffres.
- La **largeur** (plusieurs neurones = une couche) et la **matrice de poids** W , présentée avec de vrais nombres — comme la matrice X de la Séance 3.
- La **profondeur** (empiler des couches) et le décompte des *paramètres*.
- L'**activation** non-linéaire et sa dérivée ; pourquoi elle est indispensable (XOR).
- L'**apprentissage** : la perte, la descente de gradient (Séance 5), et la **rétropropagation** — dérivée, puis déroulée à la main.
- **En pratique** (**MLPClassifier**) et le verdict honnête : réseau *contre* méthodes d'ensemble (Séance 8).

Le neurone

Le point de départ : un objet déjà connu

De la logistique au neurone

Le mot « neurone » intimide. Pourtant, la brique de base des réseaux est exactement la **régression logistique** de la Séance 6 : une somme pondérée des entrées, suivie d'une fonction qui produit la sortie. Rien de nouveau — seulement un vocabulaire et une façon de l'assembler. On part donc de ce socle, et on avance *doucement*.

Le plan de la séance suit cette montée graduelle : d'abord *un* neurone (cette section) ; puis plusieurs côte à côte, formant une *couche* (la largeur) ; puis des couches empilées (la profondeur). À chaque étape, on écrit le calcul avec des matrices et on le vérifie sur des chiffres concrets. L'objectif : qu'à la fin, un réseau ne soit plus une boîte noire, mais un empilement de calculs que vous savez refaire à la main.

L'idée : un neurone, deux gestes

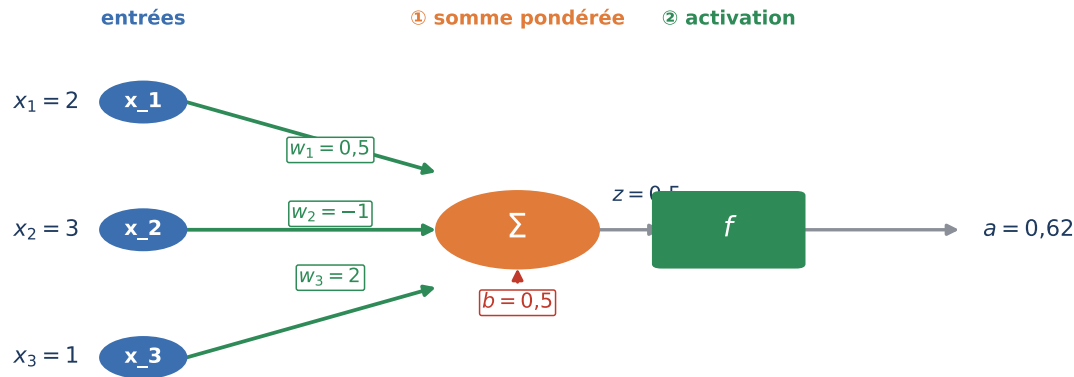
L'idée en une phrase

Un **neurone** prend des entrées, en calcule une *somme pondérée* (chaque entrée multipliée par un poids, plus un biais), puis applique une *fonction d'activation* qui produit la sortie.

$$z = w_1x_1 + w_2x_2 + \dots + w_dx_d + b = \mathbf{w}^\top \mathbf{x} + b \quad a = f(z)$$

Deux gestes seulement. **(1)** La somme pondérée z : les poids w_j disent l'importance de chaque entrée, le biais b décale le seuil. **(2)** L'activation f : elle transforme z en sortie a . Quand f est la sigmoïde σ , ce neurone est une régression logistique. Tout le reste de la séance consiste à répéter et empiler ce motif.

Le neurone, en image et en détail



$$z = w_1x_1 + w_2x_2 + w_3x_3 + b = (0,5)(2) + (-1)(3) + (2)(1) + 0,5 = 0,5 \rightarrow a = \sigma(0,5) = 0,62$$

Un neurone à trois entrées. Les valeurs x_j (bleu) sont pondérées par les poids w_j (vert, sur les flèches) ; on ajoute le biais b (rouge) ; la somme donne z ; l'activation f produit la sortie a . Ici $z = 0,5$ et $a = \sigma(0,5) = 0,62$. Tout le calcul d'un neurone tient dans cette image.

Le neurone à la main

Somme pondérée puis activation, sur des chiffres

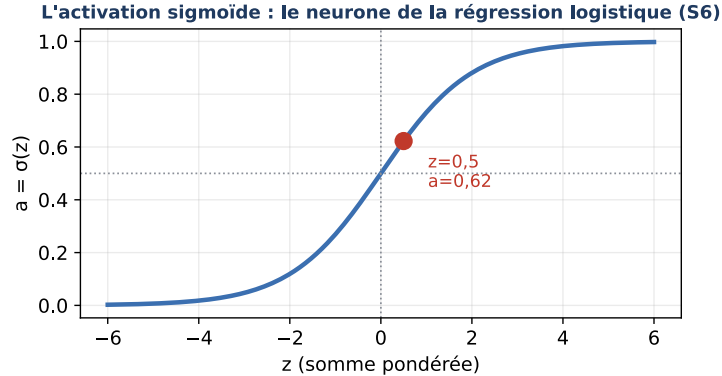
Entrées $\mathbf{x} = (2; 3; 1)$, poids $\mathbf{w} = (0,5; -1; 2)$, biais $b = 0,5$.

$$z = \mathbf{w}^\top \mathbf{x} + b = (0,5)(2) + (-1)(3) + (2)(1) + 0,5 = 1 - 3 + 2 + 0,5 = \mathbf{0,5}.$$

Puis l'activation. Avec la **sigmoïde** : $a = \sigma(0,5) = \frac{1}{1 + e^{-0,5}} = \mathbf{0,62}$. Avec **ReLU** : $a = \max(0, 0,5) = \mathbf{0,5}$.

Lecture : le poids $w_2 = -1$ *pénalise* l'entrée x_2 ; le poids $w_3 = 2$ l'amplifie. Le neurone résume les trois entrées en un seul nombre, puis le « presse » entre 0 et 1 (sigmoïde) — c'est, au chiffre près, le calcul d'une régression logistique de la Séance 6.

Le neurone est la régression logistique



La sigmoïde transforme la somme pondérée z en une valeur entre 0 et 1, interprétée comme une probabilité. Le point rouge marque notre exemple : $z = 0,5 \rightarrow a = 0,62$. Cette courbe est celle de la Séance 6 — un neurone à activation sigmoïde est une régression logistique.

Exercice — calculer la sortie d'un neurone

Énoncé

Un neurone à deux entrées, poids $\mathbf{w} = (1,5; -2)$, biais $b = 1$. (a) Calculer z pour $\mathbf{x} = (2; 1)$. (b) Donner la sortie avec ReLU. (c) Même question pour $\mathbf{x} = (0; 2)$.

Correction

(a) $z = (1,5)(2) + (-2)(1) + 1 = 3 - 2 + 1 = \mathbf{2}$. **(b)** ReLU : $\max(0, 2) = \mathbf{2}$. **(c)** $z = (1,5)(0) + (-2)(2) + 1 = -3$; ReLU : $\max(0, -3) = \mathbf{0}$ — le neurone est « éteint » pour cette entrée. C'est tout le rôle de ReLU : laisser passer le positif, bloquer le négatif.

La largeur : une couche de neurones

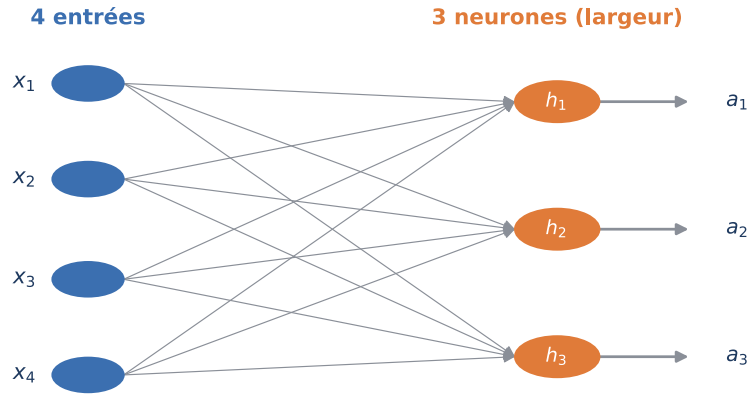
Un neurone ne voit qu'une chose

Pourquoi plusieurs neurones

Un seul neurone produit un seul nombre : il ne capte qu'un aspect des données. Pour en saisir plusieurs en parallèle, on place *plusieurs* neurones côte à côte, chacun avec ses propres poids — c'est la **largeur** d'une couche. Ils reçoivent tous les mêmes entrées, mais en tirent des réponses différentes.

Imaginons trois neurones sur les mêmes données patient : l'un pourrait réagir surtout à la fièvre, un autre à la glycémie, un troisième à l'âge — selon leurs poids. Ensemble, ils forment une **couche** qui décrit l'entrée sous trois angles. La question pratique : comment écrire le calcul de *tous* ces neurones d'un coup, proprement ? La réponse tient en un mot : une *matrice*.

Une couche, en image



Chaque neurone reçoit les 4 entrées : $4 \times 3 = 12$ poids + 3 biais. Une couche = un calcul $\mathbf{z} = W\mathbf{x} + \mathbf{b}$

Quatre entrées alimentent trois neurones (la *largeur* de la couche). Chaque neurone a ses propres poids vers les quatre entrées — d'où $4 \times 3 = 12$ poids, plus 3 biais. Chaque neurone produit sa sortie a_j . La couche transforme 4 nombres en 3 nombres.

L'idée : ranger les poids dans une matrice

L'idée en une phrase

Les poids de tous les neurones d'une couche se rangent dans une **matrice de poids** W : *une ligne par neurone, une colonne par entrée*. Le calcul de la couche entière s'écrit alors en une seule opération : $z = Wx + b$.

C'est exactement l'esprit de la **matrice** X de la Séance 3, où chaque ligne était un patient. Ici, chaque ligne de W est un neurone, et le produit matrice-vecteur Wx calcule *simultanément* les sommes pondérées de tous les neurones. Le vecteur de biais b ajoute un biais par neurone. Cette écriture compacte n'est pas qu'une élégance : c'est ce qui rend les réseaux calculables vite, et parallélisables sur une carte graphique.

La couche, formellement

Définition

Pour une couche de m neurones recevant d entrées : la matrice W est de taille $m \times d$, le biais $\mathbf{b}$ de taille m . La couche calcule un vecteur $\mathbf{z}$ de m sommes pondérées, puis applique l'activation composante par composante.

$$\mathbf{z} = W\mathbf{x} + \mathbf{b} \quad \mathbf{a} = f(\mathbf{z}) \quad W \in \mathbb{R}^{m \times d}, \mathbf{x} \in \mathbb{R}^d, \mathbf{b} \in \mathbb{R}^m$$

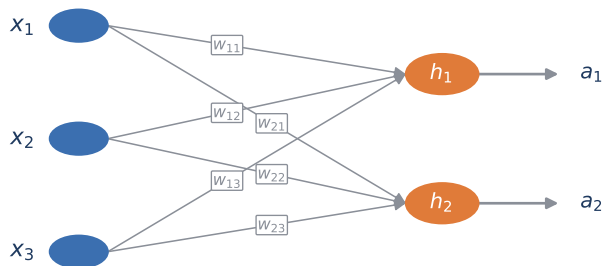
Les dimensions s'enchaînent : W ($m \times d$) fois $\mathbf{x}$ (d) donne $\mathbf{z}$ (m). Chaque *ligne* i de W contient les poids du neurone i ; le produit de cette ligne par $\mathbf{x}$ donne la somme pondérée z_i de ce neurone. Faire le produit matriciel, c'est faire les m sommes pondérées d'un seul geste. Voyons-le sur de vrais chiffres.

Du réseau à la matrice : la règle d'indexation

Lire la matrice sur le réseau

Chaque arête du réseau porte un poids. On les range dans W par une règle simple : le poids de l'**entrée** j vers le **neurone** i se place en *ligne* i , *colonne* j . Une ligne = un neurone ; une colonne = une entrée.

Réseau 3→2 : chaque arête porte un poids w_{ij}



w_{ij} : poids de l'entrée j vers le neurone i

$$W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

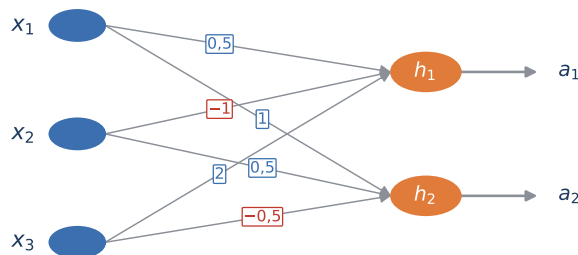
Ligne 1 : les poids du neurone h_1 (vers les 3 entrées). Ligne 2 : ceux de h_2 . Le poids w_{ij} relie l'entrée j au neurone i .

Du réseau à la matrice : les chiffres

On lit W directement sur les arêtes

Le même réseau $3 \rightarrow 2$, avec ses vrais poids. Entrée $\mathbf{x} = (2; 1; 0)$, biais $\mathbf{b} = (0,5; -1)$.

Le même réseau, avec les vrais poids



On lit directement la matrice W sur les arêtes

$$W = \begin{pmatrix} 0,5 & -1 & 2 \\ 1 & 0,5 & -0,5 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} 0,5 \\ -1 \end{pmatrix}$$

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}:$$

- $z_1 = (0,5)(2) + (-1)(1) + (2)(0) + 0,5 = 0,5$
- $z_2 = (1)(2) + (0,5)(1) + (-0,5)(0) - 1 = 1,5$

$$\text{puis } \mathbf{a} = \sigma(\mathbf{z}) = (0,62; 0,82).$$

La matrice de poids, avec de vrais chiffres

Comme la matrice X de la Séance 3

Une couche de 3 neurones sur 4 entrées. Entrée $\mathbf{x} = (1; 2; 0; 3)$, biais $\mathbf{b} = (0,5; -1; 1)$:

$$W = \begin{pmatrix} 0,5 & -1 & 2 & 0 \\ 1 & 0,5 & -0,5 & 1 \\ -1 & 2 & 0 & 0,5 \end{pmatrix} \begin{matrix} \leftarrow \text{neurone 1} \\ \leftarrow \text{neurone 2} \\ \leftarrow \text{neurone 3} \end{matrix} \quad \mathbf{x} = \begin{pmatrix} 1 \\ 2 \\ 0 \\ 3 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0,5 \\ -1 \\ 1 \end{pmatrix}$$

Chaque ligne est un neurone, chaque colonne une entrée. Le produit $W\mathbf{x} + \mathbf{b}$ donne les trois sommes pondérées en une fois.

La matrice de poids, en image

matrice de poids W (3 neurones $\times$ 4 entrées)

	x_1	x_2	x_3	x_4
neur. 1	0.5	-1	2	0
neur. 2	1	0.5	-0.5	1
neur. 3	-1	2	0	0.5

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}$$

$$= (-1 ; 4 ; 5,5)$$

$\mathbf{x}$

1
2
0
3

$\mathbf{b}$

0,5
-1
1

Une ligne de W = un neurone ; une colonne = une entrée. Le produit $W\mathbf{x}$ calcule les 3 sommes d'un coup.

La matrice W (3×4) : en bleu les poids positifs, en rouge les négatifs, en gris les nuls. À droite, le vecteur d'entrée $\mathbf{x}$ et le vecteur de biais $\mathbf{b}$. Le produit $W\mathbf{x} + \mathbf{b}$ calcule les trois sorties $\mathbf{z} = (-1 ; 4 ; 5,5)$ d'un seul coup. C'est le pendant, pour les poids, de la matrice X de la Séance 3.

Le calcul de la couche, terme à terme

$z = Wx + b$, ligne par ligne

Avec $x = (1; 2; 0; 3)$ et $b = (0,5; -1; 1)$:

- **neurone 1** : $(0,5)(1) + (-1)(2) + (2)(0) + (0)(3) + 0,5 = -1,5 + 0,5 = -1$.
- **neurone 2** : $(1)(1) + (0,5)(2) + (-0,5)(0) + (1)(3) - 1 = 5 - 1 = 4$.
- **neurone 3** : $(-1)(1) + (2)(2) + (0)(0) + (0,5)(3) + 1 = 4,5 + 1 = 5,5$.

Donc $z = (-1; 4; 5,5)$, puis $a = \sigma(z) = (0,27; 0,98; 1,00)$. La couche a transformé 4 entrées en 3 activations — chaque neurone répondant différemment selon ses poids.

Exercice — le calcul d'un neurone dans une couche

Énoncé

Reprenons la matrice W ci-dessus et $\mathbf{x} = (1; 2; 0; 3)$, $\mathbf{b} = (0,5; -1; 1)$. On ajoute un quatrième neurone, de poids $(2; 0; 1; -1)$ et de biais 0. Calculer sa somme pondérée z_4 , puis sa sortie sigmoïde.

Correction

$z_4 = (2)(1) + (0)(2) + (1)(0) + (-1)(3) + 0 = 2 - 3 = -1$. Sortie : $\sigma(-1) = \mathbf{0,27}$. Pour l'ajouter à la couche, il suffit d'ajouter une *ligne* $(2, 0, 1, -1)$ à W (qui devient 4×4) et une composante 0 à $\mathbf{b}$. Élargir une couche = ajouter des lignes à W .

La profondeur : empiler les couches

De la largeur à la profondeur

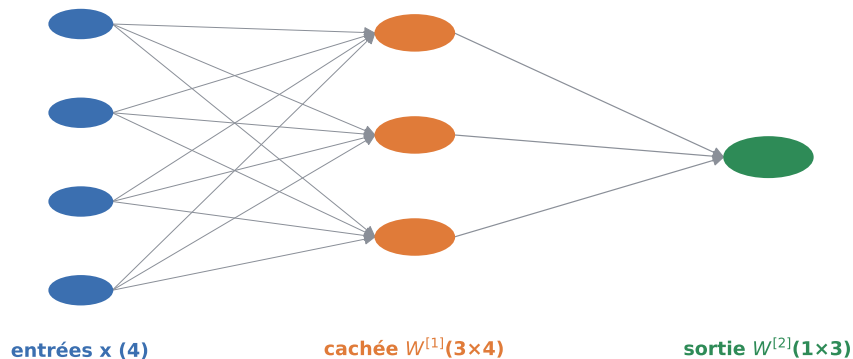
Empiler pour enrichir

Une couche transforme des entrées en activations. Et si l'on *réinjectait* ces activations dans une *autre* couche ? C'est la **profondeur** : empiler les couches, la sortie de l'une devenant l'entrée de la suivante. Chaque couche construit des représentations un peu plus élaborées que la précédente.

On distingue ainsi la **largeur** (le nombre de neurones *dans* une couche) et la **profondeur** (le nombre de couches *empilées*). Un réseau « profond » a beaucoup de couches — d'où le terme *apprentissage profond*. La structure typique : une couche d'entrée (les variables), une ou plusieurs couches *cachées* (au milieu), une couche de sortie (la prédiction). Calculer la prédiction, c'est traverser ces couches de gauche à droite : la *propagation avant*.

Un réseau, en image

Un réseau : empiler des couches (profondeur)



Un réseau 4—3—1 : quatre entrées, une couche cachée de trois neurones, une sortie. Chaque couche a sa matrice de poids : $W^{[1]} (3 \times 4)$ puis $W^{[2]} (1 \times 3)$. La *profondeur* est le nombre de couches (ici 2) ; la *largeur* de la couche cachée est 3.

La propagation avant, formellement

Définition

On note $\mathbf{a}^{[0]} = \mathbf{x}$ les entrées. Chaque couche ℓ a sa matrice $W^{[\ell]}$ et son biais $\mathbf{b}^{[\ell]}$, et calcule à partir de la couche précédente :

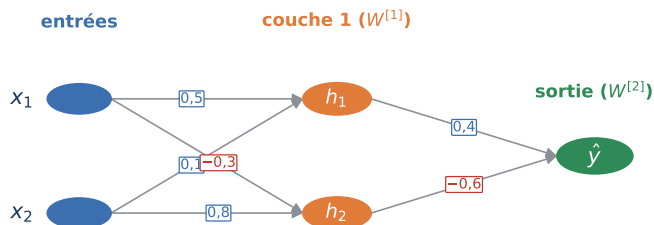
$$\mathbf{z}^{[\ell]} = W^{[\ell]} \mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]} \quad \mathbf{a}^{[\ell]} = f(\mathbf{z}^{[\ell]})$$

On enchaîne : $\mathbf{x} \rightarrow (\mathbf{z}^{[1]}, \mathbf{a}^{[1]}) \rightarrow (\mathbf{z}^{[2]}, \mathbf{a}^{[2]}) \rightarrow \dots \rightarrow \mathbf{a}^{[L]} = \hat{y}$. Chaque couche fait ses deux gestes — combinaison linéaire $W\mathbf{a} + \mathbf{b}$, puis activation f . La sortie de la dernière couche est la prédiction. Un réseau, en propagation avant, n'est donc qu'une **composition** de couches : linéaire, activation, linéaire, activation, et ainsi de suite.

Un réseau profond, ce sont des matrices (en chiffres)

Le même principe que la couche, empilé

Un réseau 2—2—1 avec ses vrais poids. Comme pour une couche, chaque jeu de poids est une *matrice* : $W^{[1]}$ pour la couche cachée, $W^{[2]}$ pour la sortie.



Un réseau profond = une suite de matrices de poids, chacune avec ses chiffres

$$W^{[1]} = \begin{pmatrix} 0,5 & 0,1 \\ -0,3 & 0,8 \end{pmatrix}, \quad W^{[2]} = \begin{pmatrix} 0,4 & -0,6 \end{pmatrix}$$

$W^{[1]}$ est 2×2 (2 neurones, 2 entrées) ;
 $W^{[2]}$ est 1×2 (1 sortie, 2 neurones cachés). Prédire = enchaîner $\mathbf{z}^{[1]}$
 $= W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}$ puis $\hat{y} = \sigma(W^{[2]}\mathbf{a}^{[1]} + b^{[2]})$. C'est ce réseau que l'on entraînera par rétropropagation.

Compter les paramètres d'un réseau

L'idée en une phrase

Les **paramètres** d'un réseau sont tous ses poids et tous ses biais — les nombres que l'apprentissage devra ajuster. On les compte couche par couche : pour une couche de m neurones sur d entrées, $m \times d$ poids et m biais.

Ce décompte importe : c'est le nombre de « boutons » que la descente de gradient devra régler. Plus il est grand, plus le réseau est expressif, mais plus il est lent à entraîner et enclin au sur-apprentissage (Séances 3 et 7). Sur notre petit réseau 4 — 3 — 1, comptons précisément.

Le décompte, sur le réseau 4—3—1

Compter les paramètres du réseau 4-3-1

	poids	biais	
couche cachée	$W^{[1]} : 3 \times 4 = 12$	$b^{[1]} : 3$	15
couche sortie	$W^{[2]} : 1 \times 3 = 3$	$b^{[2]} : 1$	4
TOTAL			19

Couche cachée : $W^{[1]}$ a $3 \times 4 = 12$ poids, plus 3 biais = 15. Couche de sortie : $W^{[2]}$ a $1 \times 3 = 3$ poids, plus 1 biais = 4. Total : **19** paramètres. Pour un si petit réseau, déjà 19 nombres à apprendre — d'où l'explosion dès qu'on agrandit, et le besoin d'une méthode automatique d'ajustement.

Exercice — dimensions et paramètres

Énoncé

Un réseau a 5 entrées, une couche cachée de 10 neurones, une seconde couche cachée de 4 neurones, et 1 sortie. (a) Quelle est la taille de chaque matrice de poids ? (b) Combien de paramètres au total (poids + biais) ?

Correction

(a) $W^{[1]} : 10 \times 5$; $W^{[2]} : 4 \times 10$; $W^{[3]} : 1 \times 4$. **(b)** Poids : $50 + 40 + 4 = 94$. Biais : $10 + 4 + 1 = 15$. Total : **109** paramètres. On voit l'explosion : un réseau modeste dépasse déjà la centaine de paramètres — impossible de les régler à la main, d'où la descente de gradient et la rétropropagation.

La non-linéarité

Pourquoi un seul neurone ne suffit pas

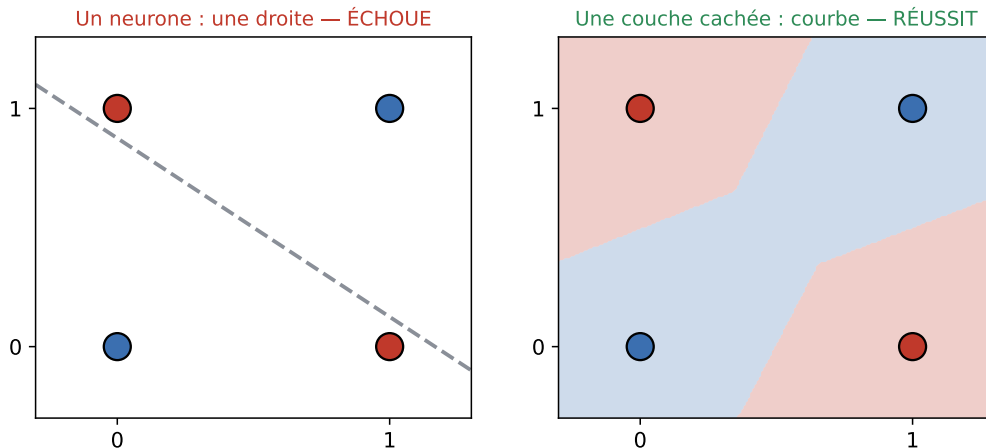
La limite de la frontière droite

Un seul neurone, comme la logistique, ne trace qu'une **frontière droite** (Séance 6). Dès que la séparation doit *courber*, il échoue. L'exemple canonique : le **XOR**, quatre points en damier qu'aucune droite ne sépare. Historiquement, c'est ce test qui a révélé la nécessité des couches cachées et de la non-linéarité.

Le XOR (« ou exclusif ») vaut 1 si *exactement une* entrée vaut 1. Les deux points de classe 1 sont en diagonale, comme les deux de classe 0 : aucune droite ne fonctionne. Une régression logistique y atteint 50 % d'accuracy (le hasard) ; un réseau à une couche cachée atteint 100 %. Mais attention : la couche cachée ne suffit pas seule — encore faut-il une *activation non-linéaire*, comme on va le voir.

XOR : un neurone échoue, une couche réussit

Le problème XOR : pourquoi il faut une couche cachée



À gauche, un neurone ne dispose que d'une droite : deux points seront toujours mal classés. À droite, une couche cachée combine plusieurs droites en une frontière *courbe* qui isole chaque classe. C'est l'apport conjoint de la couche cachée et de la non-linéarité.

L'idée : sans activation, tout s'effondre

L'idée en une phrase

La **fonction d'activation** doit être *non-linéaire*. Sans elle, empiler des couches ne sert à rien : la composition de transformations linéaires est *encore* linéaire — le réseau profond se réduit à un unique neurone.

C'est le point le plus contre-intuitif de la séance. On pourrait croire qu'empiler deux couches *linéaires* (somme pondérée sans activation) enrichit le modèle. Il n'en est rien. Démonstrons-le sur des matrices — c'est court, et cela éclaire *pourquoi* l'activation existe.

La preuve : deux couches linéaires = une seule

Composition linéaire

Deux couches *sans* activation : $\mathbf{a}^{[1]} = W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}$, puis $\hat{y} = W^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}$. En substituant :

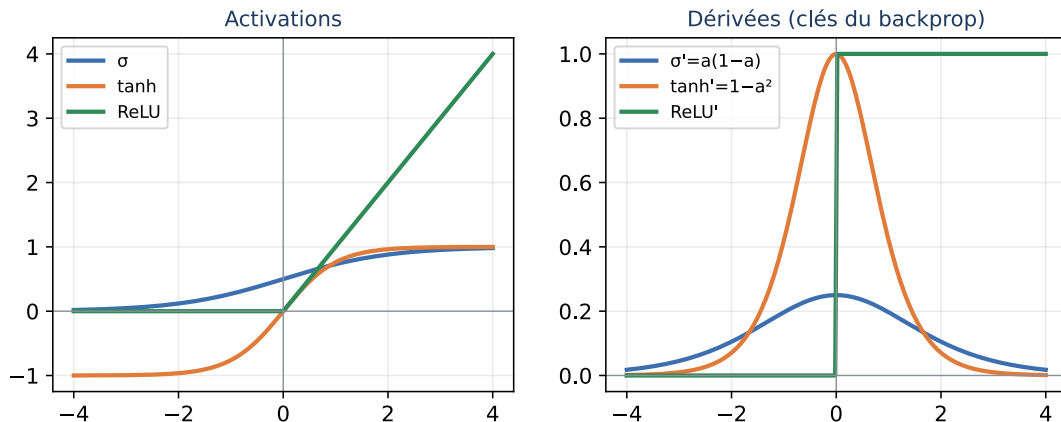
$$\hat{y} = W^{[2]}(W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]} = \underbrace{(W^{[2]}W^{[1]})}_{W_{\text{eq}}}\mathbf{x} + \underbrace{(W^{[2]}\mathbf{b}^{[1]} + \mathbf{b}^{[2]})}_{\mathbf{b}_{\text{eq}}}.$$

C'est une *unique* couche linéaire. Vérifié : avec $W^{[1]} = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$, $W^{[2]} = (1 \ -2)$, $\mathbf{x} = (1; 2)$, les deux couches donnent $-4, 5$, exactement comme la couche équivalente ($W_{\text{eq}} = \begin{pmatrix} 2 & -5 \end{pmatrix}$).

Empiler du linéaire n'apporte rien : il faut une activation non-linéaire entre les couches.

Les activations et leurs dérivées

Trois activations et leurs dérivées



À gauche, trois activations : **sigmoïde** (0 à 1), **tanh** (-1 à 1), **ReLU** ($\max(0, z)$). À droite, leurs *dérivées*, qui interviendront dans la rétropropagation : $\sigma' = a(1-a)$ (plafonne à 0,25), $\tanh' = 1-a^2$, $\text{ReLU}' = 1$ si $z > 0$ sinon 0. La faible dérivée de la sigmoïde explique qu'elle « écrase » les gradients en profondeur.

Pourquoi ReLU domine

La plus simple est la meilleure

- **Coût** : un simple test « $z > 0$? », bien plus rapide qu'une exponentielle — décisif avec des millions de neurones.
- **Gradients** : sa dérivée vaut 1 pour $z > 0$, ce qui laisse l'erreur se propager sur de nombreuses couches. La sigmoïde, dont la dérivée plafonne à 0,25, étouffe le gradient en profondeur.
- **Parcimonie** : elle éteint les neurones négatifs (sortie 0), rendant le réseau plus efficace.

En pratique : **ReLU** dans les couches cachées, **sigmoïde** en sortie pour une classification binaire. C'est le réglage par défaut, et il fonctionne presque toujours.

Piège — oublier la non-linéarité

Un réseau « linéaire » n'apprend rien de plus

Un réseau à plusieurs couches *sans* activation (ou avec une activation linéaire) donne l'illusion de la puissance, mais se réduit à une régression linéaire ordinaire — incapable de résoudre XOR ou tout motif courbe. La non-linéarité n'est pas un détail : c'est *la* raison d'être des couches.

Corollaire : la profondeur n'a de sens qu'avec des activations non-linéaires. Quand on dit qu'un réseau est « profond », ce qui compte n'est pas seulement le nombre de couches, mais le fait que chacune introduit une non-linéarité qui enrichit ce que le réseau peut représenter.

L'apprentissage : la perte et le gradient

Apprendre, c'est ajuster les poids

Le problème de l'apprentissage

Jusqu'ici, les poids étaient *donnés*. Mais d'où viennent-ils ? De l'**apprentissage** : on part de poids aléatoires, et on les ajuste pour que les prédictions $\hat{y}$ collent aux vraies étiquettes y . Il faut donc deux choses : une mesure de l'erreur (la *perte*), et une méthode pour la réduire (la *descente de gradient*, Séance 5).

La logique est exactement celle de la régression linéaire (S5) et logistique (S6) : définir une perte, puis descendre sa pente. La seule difficulté *propre* aux réseaux est calculatoire : comment obtenir la dérivée de la perte par rapport à *chacun* des nombreux poids, y compris ceux des couches cachées ? La réponse — la rétropropagation — viendra à la section suivante. D'abord, la perte.

La perte : l'entropie croisée

Définition

Pour une classification binaire, la sortie $\hat{y} \in [0, 1]$ est la probabilité prédite que $y = 1$. La **perte d'entropie croisée** (déjà vue en Séance 6, issue du maximum de vraisemblance) mesure l'écart entre $\hat{y}$ et y :

$$L(y, \hat{y}) = -[y \ln \hat{y} + (1 - y) \ln(1 - \hat{y})]$$

Lecture : si $y = 1$, la perte vaut $-\ln \hat{y}$ — nulle quand $\hat{y} \rightarrow 1$, énorme quand $\hat{y} \rightarrow 0$ (le modèle est puni d'être confiant et faux). Symétriquement si $y = 0$. Sur notre futur exemple ($y = 1, \hat{y} = 0,558$) : $L = -\ln(0,558) = 0,584$. Cette perte est différentiable — c'est ce qui permet la descente de gradient.

La descente de gradient, formellement

L'idée en une phrase

On ajuste chaque poids dans la direction *opposée* à sa dérivée partielle de la perte, d'un petit pas η (le taux d'apprentissage), et l'on répète. C'est la méthode de la Séance 5, appliquée à tous les poids du réseau.

$$w \leftarrow w - \eta \frac{\partial L}{\partial w} \quad (\text{pour } \textit{chaque} \text{ poids et biais})$$

Descendre la pente de la perte, exactement comme pour la régression linéaire. Le taux η règle la taille du pas : trop grand, on diverge ; trop petit, on traîne. La seule pièce manquante est le calcul de $\partial L / \partial w$ pour les *milliers* de poids du réseau — et c'est précisément ce que résout la rétropropagation, qu'on aborde maintenant.

La rétropropagation

Le défi : des milliers de dérivées

Le calcul naïf est impraticable

Il faut $\partial L / \partial w$ pour *chaque* poids. Les recalculer un à un, en refaisant tout le réseau à chaque fois, coûterait un temps prohibitif : le réseau serait inentraînable. Il faut une méthode qui obtient *tous* les gradients efficacement.

La clé : ces dérivées partagent d'énormes quantités de calcul. La **rétropropagation** les obtient toutes en une seule passe arrière, en réutilisant les résultats grâce à la *règle de dérivation en chaîne*. C'est cet algorithme — et lui seul — qui a rendu les réseaux profonds entraînaables, et donc possible toute l'IA moderne. On va le dériver, puis le dérouler à la main.

L'idée : renvoyer l'erreur en arrière

L'idée en une phrase

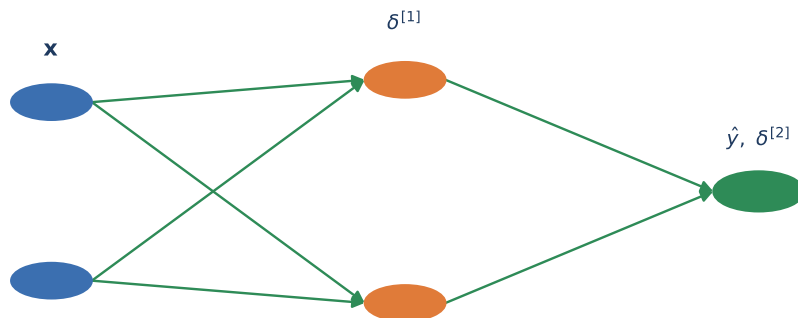
La **rétropropagation** calcule le gradient de la perte par rapport à chaque poids en partant de la *sortie* et en remontant couche par couche vers l'*entrée*, réutilisant à chaque étape l'erreur de la couche suivante.

L'image : une erreur constatée en sortie est « renvoyée » à travers le réseau ; chaque couche reçoit sa part de responsabilité — notée $\delta^{[\ell]}$, l'erreur de la couche ℓ — et en déduit comment corriger ses poids. À chaque itération, deux passes alternent : la propagation **avant** (calculer $\hat{y}$ et la perte), puis la propagation **arrière** (calculer tous les gradients). Établissons les quatre équations de cette passe arrière.

Les deux passes, en image

Les deux passes : avant pour prédire, arrière pour corriger

PROPAGATION AVANT → prédire $\hat{y}$



← RÉTROPROPAGATION : l'erreur δ remonte

En **vert**, la propagation avant : l'information va de l'entrée vers la sortie pour produire $\hat{y}$. En **rouge**, la rétropropagation : l'erreur δ remonte de la sortie vers l'entrée, chaque couche ajustant ses poids. Les deux passes s'enchaînent à chaque itération de la descente de gradient.

L'outil : la règle de dérivation en chaîne

Le seul ingrédient mathématique

Si une quantité L dépend de z , qui dépend de w , alors la dérivée de L par rapport à w est le *produit* des dérivées le long du chemin :

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Pour savoir comment un poids *profond* influence la perte, on enchaîne les sensibilités le long du chemin qui les relie. La rétropropagation n'est rien d'autre que l'application *organisée* de cette règle, de la sortie vers l'entrée, en stockant les résultats intermédiaires $\delta^{[\ell]}$ pour ne jamais recalculer deux fois la même chose. Établissons d'abord l'erreur de *sortie*, puis celle des couches *cachées*.

Dérivation 1 — l'erreur de sortie

Le résultat propre $\hat{y} - y$

On cherche $\delta^{[L]} = \partial L / \partial z^{[L]}$, avec $\hat{y} = \sigma(z^{[L]})$ et L l'entropie croisée. Par la règle en chaîne :

$$\delta^{[L]} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[L]}}.$$

Or $\frac{\partial L}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}$ et $\frac{\partial \hat{y}}{\partial z^{[L]}} = \sigma'(z^{[L]}) = \hat{y}(1 - \hat{y})$. En multipliant, le facteur $\hat{y}(1 - \hat{y})$ se simplifie :

$$\boxed{\delta^{[L]} = \hat{y} - y}$$

L'erreur de sortie est juste l'écart prédiction — vérité. Toute la complexité s'annule — c'est pourquoi le couple sigmoïde + entropie croisée s'est imposé.

Dérivation 2 — l'erreur d'une couche cachée

Remonter l'erreur, couche par couche

Pour une couche cachée ℓ , l'erreur $\delta^{[\ell]} = \partial L / \partial \mathbf{z}^{[\ell]}$ se déduit de celle de la couche *suivante*. Intuition : $\mathbf{z}^{[\ell]}$ influence la perte à *travers* la couche $\ell + 1$. La règle en chaîne donne, en propageant via les poids puis l'activation :

$$\delta^{[\ell]} = \left(W^{[\ell+1]}\right)^{\top} \delta^{[\ell+1]} \odot f'(\mathbf{z}^{[\ell]})$$

Deux gestes : **(1)** $\left(W^{[\ell+1]}\right)^{\top} \delta^{[\ell+1]}$ « renvoie » l'erreur de la couche suivante vers la couche ℓ (transposée des poids) ; **(2)** $\odot f'(\mathbf{z}^{[\ell]})$ la module par la pente locale de l'activation ($\odot$: produit composante par composante). C'est ici qu'intervient la *dérivée* de l'activation.

Les quatre équations de la rétropropagation

Tout le backprop tient en quatre lignes

(1) $\delta^{[L]} = \hat{y} - y$	erreur de sortie
(2) $\delta^{[\ell]} = \left(W^{[\ell+1]}\right)^{\top} \delta^{[\ell+1]} \odot f'(\mathbf{z}^{[\ell]})$	erreur rétropropagée
(3) $\frac{\partial L}{\partial W^{[\ell]}} = \delta^{[\ell]} \left(\mathbf{a}^{[\ell-1]}\right)^{\top}$	gradient des poids
(4) $\frac{\partial L}{\partial \mathbf{b}^{[\ell]}} = \delta^{[\ell]}$	gradient des biais

(1)–(2) : calculer les erreurs, de la sortie vers l'entrée. (3)–(4) : en déduire les gradients de tous les poids et biais. C'est *tout* l'algorithme. Appliquons-le maintenant, en chiffres.

Un pas de rétropropagation à la main

Le réseau et la propagation avant

Un réseau 2! – !2! – !1 en chiffres

Poids : $W^{[1]} = \begin{pmatrix} 0,5 & 0,1 \\ -0,3 & 0,8 \end{pmatrix}$, $b^{[1]} = (0,1; -0,2)$, $W^{[2]} = (0,4; -0,6)$, $b^{[2]} = 0,2$. Entrée $x = (1; 0)$, cible $y = 1$.

- $z^{[1]} = W^{[1]}x + b^{[1]} = (0,6; -0,5)$, $a^{[1]} = \sigma(z^{[1]}) = (0,646; 0,378)$.
- $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} = 0,232$, $\hat{y} = \sigma(0,232) = 0,558$.
- perte $L = -\ln(0,558) = 0,584$.

La prédiction 0,558 est loin de la cible 1 : la perte est élevée. La passe arrière va dire comment corriger chaque poids.

La passe arrière — erreurs et gradients

Application des quatre équations

Éq. 1 — erreur de sortie : $\delta^{[2]} = \hat{y} - y = 0,558 - 1 = -0,442$.

Éq. 3-4 — gradients de la sortie :

$$\frac{\partial L}{\partial W^{[2]}} = \delta^{[2]} \mathbf{a}^{[1]} = -0,442 (0,646; 0,378) = (-0,286; -0,167), \frac{\partial L}{\partial b^{[2]}} = -0,442.$$

Éq. 2 — erreur rétropropagée : $\delta^{[1]} = (W^{[2]})^\top \delta^{[2]} \odot \sigma'(z^{[1]}) = (-0,040; 0,062)$.

Éq. 3-4 — gradients de la couche cachée : $\frac{\partial L}{\partial W^{[1]}} = \delta^{[1]} \mathbf{x}^\top = \begin{pmatrix} -0,040 & 0 \\ 0,062 & 0 \end{pmatrix}, \frac{\partial L}{\partial \mathbf{b}^{[1]}} = (-0,040; 0,062)$.

Vérification : différences finies

La dérivation est-elle correcte ?

Pour être *sûr* de ces gradients, on les recalcule *indépendamment*, sans backprop, par la définition même de la dérivée — la **différence finie** : faire varier un poids de ε minuscule, mesurer la variation de perte, diviser.

$$\frac{\partial L}{\partial w} \approx \frac{L(w + \varepsilon) - L(w - \varepsilon)}{2\varepsilon} \quad (\varepsilon = 10^{-6})$$

Résultat (vérifié en code) : pour $\partial L / \partial W_1^{[2]}$, le backprop donne $-0,28559$ et la différence finie $-0,28559$ — écart de 6×10^{-11} . Idem pour *tous* les autres gradients (écarts $\sim 10^{-10}$). La dérivation des quatre équations est donc **exacte**. Ce « test du gradient » est le réflexe pour valider toute implémentation de backprop.

La mise à jour fait baisser la perte

Un pas de descente de gradient, $\eta = 0,5$

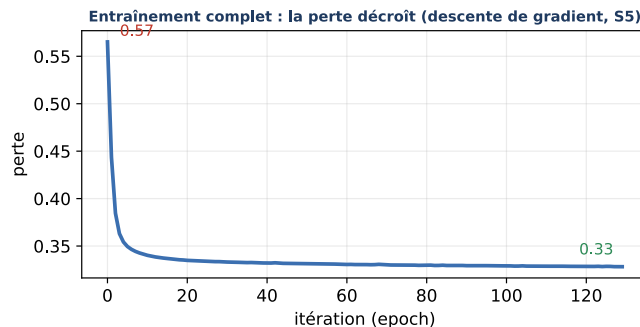
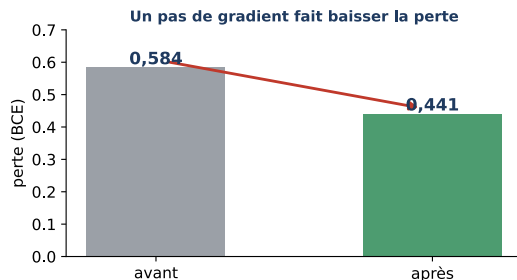
On applique $w \leftarrow w - \eta \partial L / \partial w$ à tous les poids. Par exemple :

$$W^{[2]} = (0,4; -0,6) \rightarrow (0,4; -0,6) - 0,5 (-0,286; -0,167) = (0,543; -0,517).$$

$$b^{[2]} = 0,2 \rightarrow 0,2 - 0,5(-0,442) = 0,421.$$

Après mise à jour de *tous* les poids, une nouvelle propagation avant donne une perte de **0,441** — contre 0,584 avant : une baisse de 0,143 en un seul pas. Le réseau a appris. Répété des milliers de fois sur tous les exemples, ce mécanisme entraîne le réseau.

La perte décroît, comme en Séance 5



À gauche, notre pas à la main : la perte chute de 0,584 à 0,441. À droite, l'entraînement complet d'un réseau sur DataSANTÉ : la perte décroît de 0,57 à 0,34 au fil des itérations, puis se stabilise. C'est, à plus grande échelle, la descente de gradient de la Séance 5.

Exercice — une étape de rétropropagation

Énoncé

Un neurone de sortie (sigmoïde) produit $\hat{y} = 0,8$ pour une cible $y = 0$. L'activation cachée précédente est $\mathbf{a}^{[1]} = (0,5; 1,0)$. (a) Calculer $\delta^{[2]} = \hat{y} - y$. (b) En déduire $\partial L / \partial W^{[2]} = \delta^{[2]} \mathbf{a}^{[1]}$. (c) Le poids $W_1^{[2]} = 0,3$: après un pas avec $\eta = 0,1$, que devient-il ?

Correction

(a) $\delta^{[2]} = 0,8 - 0 = 0,8$ (le modèle prédit fort alors que $y = 0$: grosse erreur). **(b)** $\partial L / \partial W^{[2]} = 0,8 (0,5; 1,0) = (0,40; 0,80)$. **(c)** $W_1^{[2]} \leftarrow 0,3 - 0,1(0,40) = 0,26$: le poids *diminue*, ce qui réduira la prédiction la prochaine fois — exactement l'effet voulu.

Piège — l'apprentissage est délicat

Plus de paramètres, plus de réglages

Beaucoup de poids = plus de façons de mal apprendre. La descente de gradient peut diverger si η est trop grand, stagner s'il est trop petit, ou rester coincée. Les réseaux sont très sensibles à la **standardisation** des entrées (Séance 9) et à l'*initialisation* des poids.

Conséquences pratiques : **toujours standardiser** ; surveiller la courbe de perte pour détecter une divergence ; accepter qu'un réseau demande plus de tâtonnement qu'un modèle d'ensemble. Le « gradient évanescent » (sigmoïde qui écrase les gradients en profondeur) est l'une de ces difficultés — et la raison du succès de ReLU.

En pratique avec scikit-learn

En pratique : MLPClassifier

```
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier

X = StandardScaler().fit_transform(df[colonnes])  # OBLIGATOIRE pour un reseau
reseau = MLPClassifier(
    hidden_layer_sizes=(16, 8),  # deux couches cachees : 16 puis 8 neurones
    activation="relu",           # ReLU dans les couches cachees
    max_iter=1000,               # iterations de descente de gradient
    random_state=0)
reseau.fit(X_train, y_train)
print(reseau.score(X_test, y_test))  # accuracy
print(reseau.loss_curve_[-1])        # perte finale
```

MLP signifie *Multi-Layer Perceptron* — le nom historique du réseau classique.

`hidden_layer_sizes=(16, 8)` décrit deux couches cachées. scikit-learn applique la rétropropagation et la descente de gradient en interne ; vous savez désormais ce qu'elles font. La standardisation préalable n'est pas optionnelle.

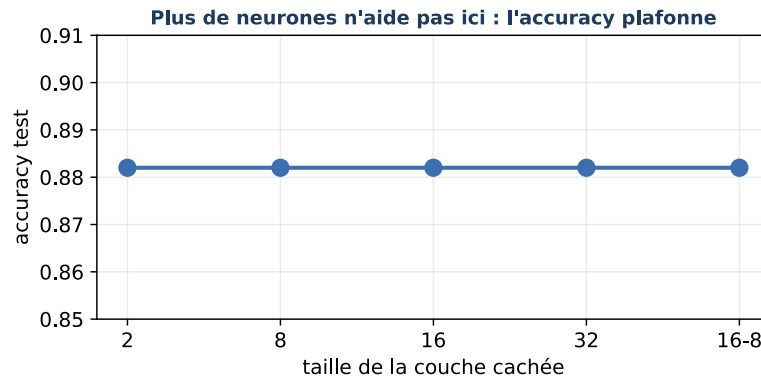
Les réglages qui comptent

Quatre leviers principaux

- `hidden_layer_sizes` : nombre de couches et de neurones (profondeur et largeur). Plus grand = plus expressif, mais plus lent et plus enclin au sur-apprentissage.
- `activation` : "relu" par défaut pour les couches cachées (le bon choix presque toujours).
- `alpha` : la régularisation L_2 (Séance 7) — augmenter contre le sur-apprentissage.
- `learning_rate_init` : le taux d'apprentissage η — trop grand, ça diverge ; trop petit, ça traîne.

À retenir : commencer simple (une ou deux petites couches), standardiser, surveiller la courbe de perte, n'agrandir que si le sous-apprentissage le justifie.

La taille de la couche : un plafond vite atteint



Sur DataSANTÉ, agrandir la couche cachée (2 $\rightarrow$ 8 $\rightarrow$ 16 $\rightarrow$ 32, puis deux couches) ne change *rien* à l'accuracy : elle plafonne à 0,882. La capacité utile dépend de la *complexité des données*, pas de la taille du modèle — et nos données tabulaires n'en demandent pas tant.

Le verdict : réseau contre ensembles

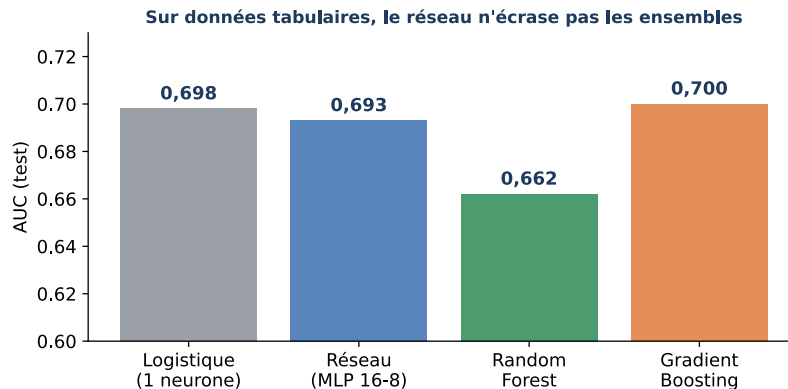
La question qui fâche

Faut-il un réseau sur nos données ?

Les réseaux dominent l'actualité — images, traduction, IA générative. La tentation est de les croire *toujours* supérieurs. Confrontons l'idée à nos données : un réseau bat-il les méthodes d'ensemble (Séance 8) sur le paludisme de DataSANTÉ ?

On compare, sur le même découpage train/test et avec les mêmes soins (standardisation pour le réseau), quatre modèles : la régression logistique (un neurone, S6), un réseau **MLP**, une forêt aléatoire et un gradient boosting (S8). Juge : l'AUC sur le test, comme en Séances 6–7.

Le comparatif honnête



Les quatre modèles se tiennent dans un mouchoir : gradient boosting 0,700, régression logistique 0,698, réseau 0,693, forêt 0,662. Le réseau ne fait *pas* mieux que la simple régression logistique — et reste derrière le boosting. Pour une complexité et un temps de réglage bien supérieurs, aucun gain.

La leçon : le réseau n'est pas magique

Sur données tabulaires, les ensembles règnent

Ce résultat n'est pas un accident : c'est un **fait empirique général**. Sur les données *tabulaires* (lignes et colonnes, comme un tableur médical), les méthodes d'ensemble — forêts, gradient boosting, XGBoost — sont régulièrement aussi bonnes ou meilleures que les réseaux, pour bien moins d'efforts.

Même leçon de maturité qu'en Séance 8 (« plus puissant n'est pas meilleur »), poussée plus loin : la *popularité* d'une famille de modèles ne dit rien de sa pertinence pour *votre* problème. Choisir un réseau par effet de mode, sur des données qui n'en ont pas besoin, c'est payer cher une complexité inutile.

Alors, quand le réseau gagne-t-il ?

Le domaine des réseaux

Les réseaux brillent là où la structure est *riche et non tabulaire* : **images** (chaque pixel lié à ses voisins), **texte** et **son** (séquences), et les problèmes à *très grand volume* de données.

Sur ces données, aucune méthode d'ensemble ne rivalise : c'est le territoire naturel de l'apprentissage profond, et la raison de son succès. Règle pratique : **tabulaire** de taille modeste → commencer par les ensembles ; **images, texte, son, très grand volume** → les réseaux prennent l'avantage. Et c'est précisément ce domaine — convolutions, séquences, transformeurs — que développera le cours de *Deep Learning* au prochain semestre, en s'appuyant sur la rétropropagation établie aujourd'hui.

Synthèse

Quel modèle pour quel problème

Une carte de tout le cours

situation	modèle de premier choix
relation simple, interprétable	régression linéaire / logistique (S5–S6)
données tabulaires, performance	forêt, gradient boosting (S8)
structure inconnue à explorer	clustering, PCA (S9)
images, texte, son, gros volume	réseaux de neurones (S10)
toujours, en premier	le modèle <i>simple</i> comme référence

Le protocole réseau

De bout en bout

1. Se demander si un réseau est *justifié* (tabulaire → essayer les ensembles d'abord) → 2. **standardiser**, sans exception → 3. commencer *petit* (une ou deux petites couches, **relu**) → 4. entraîner et **surveiller la courbe de perte** → 5. régler si besoin (**alpha**, taille, η) → 6. comparer *honnêtement* à une référence simple : si le réseau ne fait pas mieux, garder le modèle simple.

À retenir — Séance 10 (1/2)

Du neurone au réseau

- Un **neurone** = somme pondérée $z = \mathbf{w}^\top \mathbf{x} + b$ suivie d'une activation $f(z)$: c'est la régression logistique (S6) quand $f = \sigma$.
- La **largeur** (plusieurs neurones) se range dans une **matrice de poids** W ($m \times d$, une ligne par neurone) : la couche calcule $\mathbf{z} = W\mathbf{x} + \mathbf{b}$.
- La **profondeur** empile les couches ; prédire = *propagation avant* $\mathbf{z}^{[\ell]} = W^{[\ell]}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]}$.
- L'**activation non-linéaire** est indispensable (sans elle, empiler = une seule couche linéaire) ; **ReLU** domine dans les couches cachées.

À retenir — Séance 10 (2/2)

L'apprentissage et le verdict

- La **perte** (entropie croisée) mesure l'écart $\hat{y}$ vs y ; sigmoïde + BCE donne $\partial L / \partial z^{[L]} = \hat{y} - y$.
- Un réseau apprend par **descente de gradient** (S5) ; la **rétropropagation** calcule tous les gradients en une passe arrière (4 équations, règle de la chaîne).
- Le **test du gradient** (différences finies) valide toute implémentation ; toujours **standardiser**.
- **Verdict** : sur *tabulaire*, les ensembles (S8) égalent ou battent les réseaux ; le réseau gagne sur *images, texte, son, gros volumes*. Popularité $\neq$ pertinence.

Pièges fréquents

- Croire qu'un réseau est *toujours* supérieur : sur tabulaire, les ensembles font souvent aussi bien ou mieux.
- Oublier de **standardiser** : un réseau y est très sensible, bien plus qu'une forêt.
- Construire un réseau *sans* activation non-linéaire : il se réduit à une régression linéaire.
- Confondre largeur et profondeur, ou expressivité et apprentissage : un réseau *peut* tout représenter, encore faut-il *trouver* les poids.
- Ne pas surveiller la courbe de perte : on rate une divergence (η trop grand) ou une stagnation.
- Implémenter un backprop sans le **test du gradient** : une erreur de dérivation passe inaperçue.

Prochaine séance

Séance 11 — Le projet de bout en bout

La dernière séance n'introduit pas de nouveau modèle : elle *rassemble* tout le cours en un **projet complet**, du type des compétitions de science des données. Formuler le problème, préparer les données (S4), choisir et comparer des modèles (S3, S5–S10), évaluer honnêtement (S6–S7), interpréter et présenter. C'est le moment de devenir autonome — et l'évaluation se fera sur ce projet.

Références

- Goodfellow, Bengio, Courville — *Deep Learning*, MIT Press, 2016 (chap. 6 : réseaux et rétropropagation).
- Nielsen — *Neural Networks and Deep Learning*, 2015 (dérivation du backprop).
- Géron — *Hands-On Machine Learning*, 3^e éd., O'Reilly, 2022.
- Bishop — *Pattern Recognition and Machine Learning*, Springer, 2006 (chap. 5).
- Cours : Stanford CS231n (réseaux) ; University of Washington CSE 446.

Ressources de la séance

- **Notebook** · [TP 10 — Réseaux de neurones](#) (217 Ko)
- **TD** · [Fiche d'exercices 10](#) (218 Ko)