Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

École Nationale Polytechnique d'Alger

Département d'Électrotechnique

Intelligence artificielle

Compte rendu de TP n° 1

Réalisation d'un perceptron multicouche et apprentissage par rétropropagation du gradient

---

Réalisé par :

LAGHA Khalil Ellah

MATALLAH Mohammed Achref

Proposé par :

Pr. A. Hellal

Année universitaire 2023–2024

Résumé

Ce travail pratique porte sur la réalisation complète d'un perceptron multicouche en MATLAB, sans recours à aucune boîte à outils : la propagation avant, la rétropropagation du gradient et la boucle d'apprentissage sont programmées à la main, sous forme matricielle. Le programme est conçu pour être général, c'est-à-dire capable de traiter n'importe quel jeu de données sans modification de son cœur de calcul, les dimensions du réseau, les poids initiaux, le taux d'apprentissage et la tolérance étant entièrement portés par les fichiers de données.

Le programme est validé sur quatre jeux de données : trois exemples analytiques à observation unique et une petite base d'apprentissage de trois observations présentées séquentiellement. Les trois premiers convergent respectivement en 3 709, 528 537 et 5 835 itérations, et la base d'apprentissage en 6 799, 6 875 et 6 875 itérations par observation, pour un temps total inférieur à une seconde.

L'analyse de ces résultats constitue l'essentiel du rapport, car les écarts entre les exemples s'expliquent tous par des phénomènes fondamentaux de l'apprentissage des réseaux de neurones. L'exemple 2 demande cent fois plus d'itérations que les autres pour trois raisons cumulées : biais désactivés, initialisation symétrique qui empêche les deux neurones cachés de se différencier, et cible fixée à 0,5, que la sigmoïde n'atteint qu'asymptotiquement. Sur la base d'apprentissage, les poids de la couche cachée ne bougent pas du tout : avec des entrées brutes de l'ordre de 230, les pré-activations atteignent plusieurs dizaines, la sigmoïde sature et sa dérivée devient numériquement nulle, si bien que le gradient ne traverse plus la couche cachée. Ces deux observations conduisent directement aux bonnes pratiques usuelles en apprentissage automatique : initialisation aléatoire, maintien des biais et normalisation des entrées.

Mots-clés : perceptron multicouche, rétropropagation du gradient, fonction sigmoïde, descente de gradient, saturation, rupture de symétrie, normalisation des entrées, MATLAB.

Table des matières

Table des figures

Liste des tableaux

Nomenclature

N, K, S	Nombres de neurones d'entrée, cachés et de sortie	–
x, x_0	Vecteur d'entrée et entrée de biais de la couche cachée	–
z_0	Entrée de biais de la couche de sortie	–
w_1, w_{01}	Poids et poids de biais de la couche cachée	–
w_2, w_{02}	Poids et poids de biais de la couche de sortie	–
U_{in}, U_{out}	Pré-activations et sorties de la couche cachée	–
Y_{in}, y	Pré-activations et sorties du réseau	–
y_d	Sortie désirée	–
e	Erreur en sortie, $e = y_d - y$	–
f	Fonction d'activation sigmoïde	–
δ_1, δ_2	Gradients locaux des couches cachée et de sortie	–
α	Taux d'apprentissage	–
tol	Tolérance du critère d'arrêt	–
E	Erreur quadratique	–

1 Introduction

1.1 Contexte

Le perceptron multicouche est l'un des modèles fondamentaux de l'apprentissage automatique. Un réseau de neurones comportant une seule couche cachée, entraîné par rétropropagation du gradient, est capable d'approcher une fonction non linéaire quelconque avec une précision arbitraire, résultat connu sous le nom de théorème d'approximation universelle. Cette propriété en fait l'outil de base des problèmes de classification et de régression, et le point de départ de toutes les architectures plus profondes développées depuis.

Sa mise en œuvre repose sur un algorithme unique, la rétropropagation, dont l'idée est de répondre à la question suivante : chaque poids du réseau contribue à l'erreur observée en sortie, mais dans quelle proportion ? La rétropropagation répond en appliquant la règle de dérivation des fonctions composées de la sortie vers l'entrée, ce qui permet de calculer le gradient de l'erreur par rapport à tous les poids en une seule passe arrière.

1.2 Travail demandé

L'énoncé demande de réaliser un programme MATLAB de perceptron multicouche applicable à n'importe quel système ou jeu de données. Les données de trois exemples différents sont fournies, ainsi qu'une petite base d'apprentissage de trois observations.

L'exigence centrale est celle de **généralité** : le programme doit résoudre tous les exemples sans modification de son code, grâce à une formulation matricielle. Cette contrainte a orienté toute l'architecture du programme, décrite à la section 3 : les dimensions du réseau, les poids initiaux, les biais, le taux d'apprentissage, la tolérance et la sortie désirée sont intégralement portés par les fichiers de données, et le cœur de calcul n'en connaît rien d'autre que leurs dimensions.

1.3 Organisation du rapport

La section 2 rappelle la théorie du perceptron multicouche et établit les équations de la rétropropagation. La section 3 décrit l'architecture du programme et ses deux implémentations. La section 4 présente les quatre jeux de données d'essai. La section 5 rassemble les résultats et la section 6 les analyse, en s'attachant particulièrement aux deux cas pathologiques qui se sont révélés les plus instructifs. La section 7 conclut.

2 Rappel théorique

2.1 Architecture du réseau

Le réseau comporte trois couches, représentées sur la figure 1 :

- une couche d'entrée de N variables $x = (x_1, \dots, x_N)^\top$, accompagnée d'une entrée de biais x_0 ;
- une couche cachée de K neurones, de poids $w_1 \in \mathbb{R}^{K \times N}$ et de poids de biais $w_{01} \in \mathbb{R}^K$;
- une couche de sortie de S neurones, de poids $w_2 \in \mathbb{R}^{S \times K}$ et de poids de biais $w_{02} \in \mathbb{R}^S$.

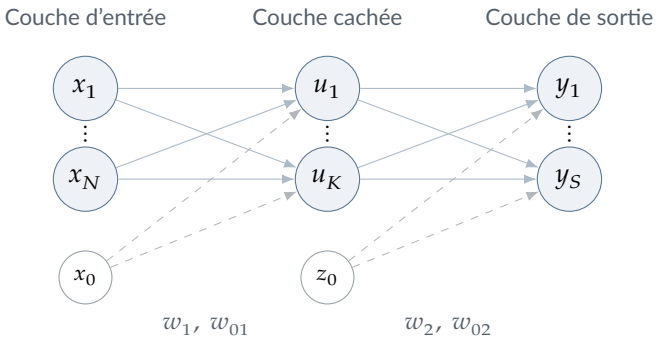

Fig. 1. Architecture du perceptron multicouche réalisé : N entrées, K neurones cachés, S sorties, avec une entrée de biais par couche.

Chaque neurone utilise la fonction d'activation sigmoïde

$$f(u) = \frac{1}{1+e^{-u}}, \qquad f'(u) = f(u)\,(1-f(u)). \tag{1}$$

La forme de la dérivée mérite d'être soulignée : elle s'exprime directement à partir de la sortie du neurone, sans qu'il soit nécessaire de conserver la pré-activation. Cela rend le calcul du gradient très économique, et c'est historiquement l'une des raisons du succès de la sigmoïde. La figure 2 représente la fonction et sa dérivée, et fait apparaître la propriété qui expliquera le comportement observé à la section 6.3 : la dérivée s'annule dès que $|u|$ dépasse quelques unités.

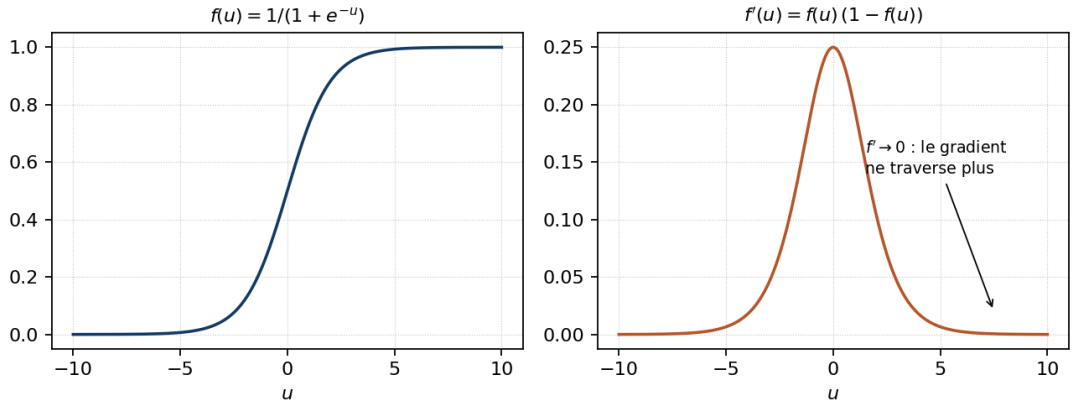

Fig. 2. Fonction sigmoïde et sa dérivée. Au-delà de $|u| \approx 6$, la dérivée est numériquement nulle : aucun gradient ne peut plus remonter à travers un neurone dans cet état.

2.2 Propagation avant

Pour une entrée x, la sortie du réseau se calcule en deux étapes, chacune consistant en un produit matriciel suivi d'une non-linéarité appliquée terme à terme :

$$U_{in} = w_1\, x + w_{01}\, x_0, \qquad U_{out} = f(U_{in}) \qquad \text{(couche cachée)}, \tag{2}$$

$$Y_{in} = w_2\, U_{out} + w_{02}\, z_0, \qquad y = f(Y_{in}) \qquad \text{(couche de sortie)}, \tag{3}$$

où x_0 et z_0 sont les entrées de biais, généralement égales à 1.

Le rôle du biais est souvent sous-estimé. Sans lui, la pré-activation d'un neurone s'annule

nécessairement lorsque toutes ses entrées sont nulles, et sa sortie vaut donc invariablement $f(0) = 0{,}5$. Le biais est le paramètre qui permet de déplacer le seuil de la fonction d'activation le long de l'axe des u ; le désactiver revient à contraindre toutes les frontières de décision à passer par l'origine. L'exemple 2 de la section 4 illustrera concrètement le coût de cette contrainte.

2.3 Rétropropagation du gradient

L'apprentissage consiste à minimiser l'erreur quadratique

$$E = \frac{1}{2} \sum_{j=1}^{S} (y_{d,j} - y_j)^2 \tag{4}$$

entre la sortie désirée y_d et la sortie calculée y.

Le problème central est l'attribution à chaque poids de sa contribution à l'erreur globale. La rétropropagation le résout par application systématique de la règle de dérivation en chaîne, en trois étapes :

1. propagation de l'entrée jusqu'à la sortie ;
2. évaluation de l'erreur en sortie, $e = y_d - y$;
3. rétropropagation de cette erreur vers les entrées et mise à jour des poids.

Les gradients locaux, traditionnellement notés δ, valent

$$\delta_2 = e \odot y \odot (1 - y) \qquad \text{(couche de sortie)}, \tag{5}$$

$$\delta_1 = (w_2^\top \, \delta_2) \odot U_{out} \odot (1 - U_{out}) \qquad \text{(couche cachée)}, \tag{6}$$

où $\odot$ désigne le produit terme à terme. Les poids sont alors corrigés proportionnellement au taux d'apprentissage α :

$$w_2 \leftarrow w_2 + \alpha \, \delta_2 \, U_{out}^\top, \qquad w_{02} \leftarrow w_{02} + \alpha \, \delta_2 \, z_0, \tag{7}$$

$$w_1 \leftarrow w_1 + \alpha \, \delta_1 \, x^\top, \qquad w_{01} \leftarrow w_{01} + \alpha \, \delta_1 \, x_0. \tag{8}$$

L'équation (6) contient toute la logique de l'algorithme et mérite qu'on s'y arrête. Le terme $w_2^\top \delta_2$ redistribue l'erreur de sortie vers les neurones cachés au prorata de leur influence, ce qui est la rétropropagation proprement dite. Le facteur $U_{out} \odot (1 - U_{out})$ est la dérivée de la sigmoïde de chaque neurone caché : il pondère l'erreur reçue par la sensibilité locale du neurone. C'est ce second facteur qui deviendra nul en cas de saturation, avec la conséquence analysée à la section 6.3.

2.4 Critère d'arrêt

L'apprentissage est itéré tant que l'erreur maximale en sortie dépasse une tolérance fixée :

$$\max_j |y_j - y_{d,j}| > \text{tol}. \tag{9}$$

Le choix de la norme infinie plutôt que de l'erreur quadratique moyenne est délibéré : il garantit que *chaque* sortie satisfait la tolérance, alors qu'une moyenne pourrait masquer une sortie très mauvaise compensée par une autre très bonne.

3 Architecture du programme

3.1 Organisation des fichiers

Le programme est découpé en scripts spécialisés, tous appelés depuis le programme principal.

Tab. 2. Organisation du programme.

Fichier	Rôle
`pmcmain.m`	programme principal, point d'entrée
`dataselect.m`	menu de choix des données et positionnement des drapeaux
`data1.m`, `data2.m`, `data3.m`	définition des exemples 1 à 3
`datass.m`	base d'apprentissage de trois observations
`datasymbol.m`	apprentissage observation par observation sur la base
`pmc.m`	boucle d'apprentissage et critère d'arrêt
`propagation.m` / `retropropagation.m`	version avec boucles explicites
`propagation2.m` / `retropropagation2.m`	version vectorisée, biais appris
`affichage.m`	affichage des résultats

Chaque fichier de données définit *entièrement* le réseau : dimensions N, K et S, poids initiaux, biais, taux d'apprentissage, tolérance et sortie désirée. Le cœur du programme est donc indépendant des données, comme l'exigeait l'énoncé. Les dimensions elles-mêmes ne sont jamais écrites en dur : elles sont déduites des tailles des vecteurs fournis, par `N = length(x)`, `K = length(w01)` et `S = length(yd)`.

3.2 Fonctionnement

Le programme principal `pmcmain.m` appelle `dataselect.m`, qui affiche un menu. Selon le choix, celui-ci charge l'exemple correspondant et positionne deux drapeaux : `CONDITIONS`, qui commande la sortie du programme, et `CONDITIONS2`, qui sélectionne le mode base de données.

Pour un exemple simple, le programme lance directement la boucle d'apprentissage `pmc.m`. Pour la base d'apprentissage, `datasymbol.m` présente les observations une par une au réseau, en conservant les poids appris d'une observation à la suivante, et appelle `pmc.m` pour chacune.

La boucle d'apprentissage constitue le cœur du programme.

```
e = 1000;       % erreur initiale, volontairement grande
niter = 0;      % compteur d'iterations

while max(abs(e)) > tol

    if CONDITIONS2 == 1
        propagation2;
        e = yd - y;         % erreur en sortie
        retropropagation2;
    else
        propagation;
        e = yd - y;         % erreur en sortie
        retropropagation;
    end

```

```
    niter = niter + 1;

end
```

Listing 1. `pmc.m` : boucle d'apprentissage et critère d'arrêt.

À chaque itération, une propagation avant calcule la sortie y, l'erreur $e = y_d - y$ est évaluée, puis la rétropropagation corrige les poids. La boucle s'arrête dès que $\max|e| \leq \text{tol}$. L'initialisation $e = 1000$ garantit l'exécution d'au moins une itération, MATLAB ne disposant pas de boucle *do–while*.

3.3 Les deux implémentations

Deux implémentations coexistent, et leur comparaison fait partie de l'objet du travail.

Version avec boucles. Les fichiers `propagation.m` et `retropropagation.m` suivent pas à pas les équations du cours, neurone par neurone, avec des boucles `for` imbriquées sur S, K et N. Cette version est utilisée pour les exemples 1 à 3 et sert de référence pédagogique : chaque ligne de code correspond exactement à une équation de la section 2, ce qui la rend vérifiable ligne à ligne.

```
% couche cachee
Uin  = w1*x + w01*x0;       % entrees des neurones caches
Uout = f(Uin);              % sorties des neurones caches

% couche de sortie
Yin  = w2*Uout + w02*z0;
Yout = f(Yin);
y = Yout;                   % sortie du reseau
```

Listing 2. `propagation.m` : propagation avant, transcription directe des équations (2) et (3).

Version vectorisée. Les fichiers `propagation2.m` et `retropropagation2.m` remplacent toutes les boucles par des produits matriciels, et apprennent en outre les poids de biais w_{01} et w_{02}. C'est la forme la plus efficace sous MATLAB, dont l'interpréteur est optimisé pour l'algèbre linéaire.

```
w2old = w2;     % poids avant mise a jour

% couche de sortie
delta2 = e .* y .* (1 - y);
w2  = w2  + alpha * delta2 * y1';
w02 = w02 + alpha * delta2 * z0;

% couche cachee
delta1 = (w2old' * delta2) .* y1 .* (1 - y1);
w1  = w1  + alpha * delta1 * x';
w01 = w01 + alpha * delta1 * x0;
```

Listing 3. `retropropagation2.m` : mise à jour vectorisée, correspondant aux équations (5) à (8).

Un détail du listing 3 est essentiel et facile à manquer : le calcul de δ_1 utilise `w2old`, c'est-à-dire les poids de la couche de sortie *avant* leur mise à jour. C'est conforme à la théorie, puisque le gradient est évalué en un point unique de l'espace des paramètres : les deux couches doivent être corrigées à partir du même état du réseau. Utiliser les poids déjà mis à jour introduirait

une inconsistance qui ne fait pas diverger l'apprentissage mais le ralentit et le rend dépendant de l'ordre d'écriture des lignes, faute difficile à détecter.

4 Données d'essai

Quatre jeux de données ont été utilisés, dont les caractéristiques sont résumées dans le tableau 3.

Tab. 3. Caractéristiques des quatre jeux de données.

Exemple	Fichier	$N \times K \times S$	α	tol	Particularité
1	`data1`	$2 \times 2 \times 1$	1	0,01	biais actifs, $x_0 = z_0 = 1$
2	`data2`	$2 \times 2 \times 2$	0,5	0,001	**biais désactivés**, w_1 symétrique
3	`data3`	$2 \times 2 \times 2$	1,5	0,001	biais actifs, α élevé
4	`datass`	$2 \times 2 \times 1$, 3 obs.	1	0,005	**entrées brutes** ≈ 230

Les sorties désirées sont respectivement $y_d = 1$ pour l'exemple 1, $y_d = (0{,}5\,;\,1)$ pour l'exemple 2 et $y_d = (0{,}01\,;\,0{,}99)$ pour l'exemple 3. L'exemple 4 est une petite base de classification binaire : trois observations à deux variables d'entrée brutes, par exemple $x = (227,\,138)$, et une étiquette valant 0 ou 1.

Deux particularités méritent d'être signalées dès à présent, car elles gouvernent l'essentiel des résultats.

L'exemple 2 est configuré avec $x_0 = z_0 = 0$, donc sans biais effectif, et avec une matrice de poids initiale $w_1 = \left(\begin{smallmatrix} 0{,}5 & 0{,}5 \\ 0{,}5 & 0{,}5 \end{smallmatrix}\right)$ dont les deux lignes sont identiques. Sa cible comporte en outre une composante à 0,5.

L'exemple 4 utilise des entrées non normalisées, de l'ordre de 230, alors que les poids initiaux sont de l'ordre de l'unité. Le produit $w_1 x$ atteint donc directement plusieurs dizaines.

5 Résultats

Les mesures ont été obtenues sous MATLAB R2024b. Les temps d'exécution sont mesurés par `tic/toc` sur la boucle d'apprentissage complète. Les courbes de convergence des figures 3 et 4 ont été obtenues en instrumentant la boucle pour enregistrer $\max_j |e_j|$ à chaque itération.

5.1 Exemples 1 à 3

Tab. 4. Convergence des exemples 1 à 3, version avec boucles.

Exemple	y final	y_d	Itérations	Temps
1	0,990	1	3 709	0,37 s
2	(0,500 ; 0,999)	(0,5 ; 1)	528 537	36,2 s
3	(0,011 ; 0,989)	(0,01 ; 0,99)	5 835	0,39 s

Les trois exemples convergent effectivement, c'est-à-dire que le critère d'arrêt (9) finit par être satisfait dans tous les cas. Les poids finaux obtenus valent, pour l'exemple 1,

$$w_1 = \begin{pmatrix} 0{,}700 & 0{,}763 \\ -0{,}200 & 1{,}155 \end{pmatrix}, \qquad w_2 = \begin{pmatrix} 3{,}065 & 3{,}291 \end{pmatrix},$$

et pour l'exemple 3,

$$w_1 = \begin{pmatrix} 0{,}217 & 1{,}546 \\ 0{,}315 & 1{,}607 \end{pmatrix}, \qquad w_2 = \begin{pmatrix} -2{,}905 & -2{,}939 \\ 2{,}188 & 2{,}271 \end{pmatrix}.$$

Une observation utile sur l'exemple 1 : la première colonne de w_1 est restée exactement à sa valeur initiale $(0{,}700\,;\,-0{,}200)^\top$. C'est attendu et non fautif, puisque la première composante de l'entrée vaut $x_1 = 0$: d'après la règle de mise à jour (8), la correction appliquée à $w_1(k,i)$ est proportionnelle à x_i, donc nulle pour $i = 1$. Un poids relié à une entrée identiquement nulle ne peut pas apprendre.

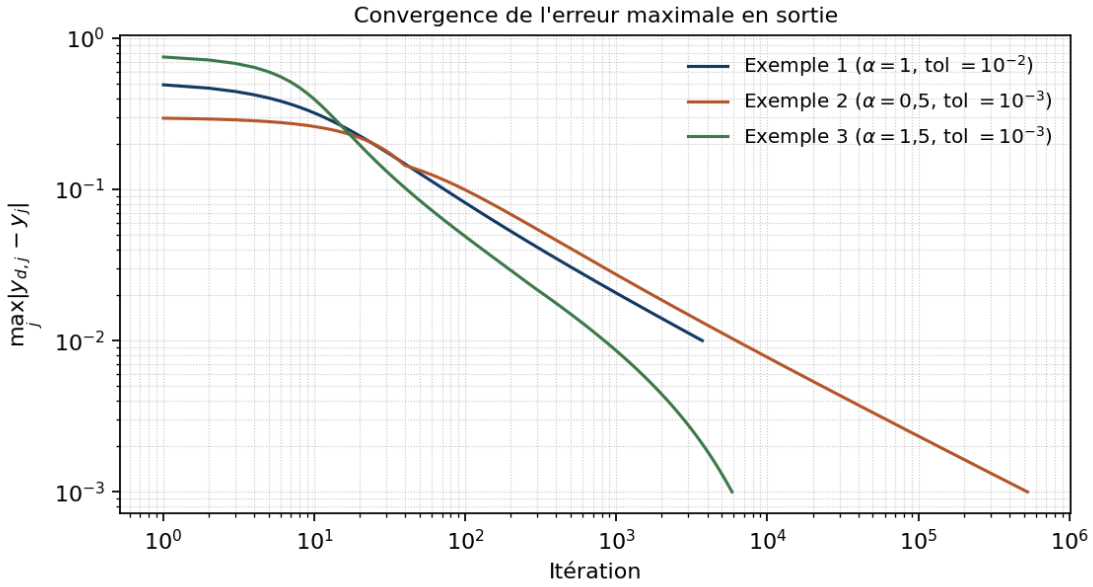

Fig. 3. Convergence de l'erreur maximale en sortie pour les trois exemples, échelles logarithmiques sur les deux axes. L'exemple 2 présente un long palier avant que l'erreur ne commence réellement à décroître.

La figure 3 met en évidence des allures très différentes. Les exemples 1 et 3 décroissent régulièrement dès les premières itérations, tandis que l'exemple 2 stagne pendant plusieurs centaines de milliers d'itérations avant de franchir sa tolérance. Cette différence est analysée à la section 6.2.

5.2 Base d'apprentissage

Pour la base d'apprentissage, traitée avec la version vectorisée, chaque observation est entraînée jusqu'à convergence, les poids étant conservés d'une observation à la suivante.

Tab. 5. Base d'apprentissage : itérations par observation. Temps total **0,99 s**.

Observation	Entrée x	y_d	Itérations
1	(227, 138)	1	6 799
2	(229, 110)	0	6 875
3	(235, 124)	1	6 875

Les poids finaux sont

$$w_2 = \begin{pmatrix} 2{,}074 & 1{,}664 \end{pmatrix}, \qquad w_{02} = 1{,}554,$$

tandis que w_1 et w_{01} restent *exactement* à leurs valeurs initiales

$$w_1 = \begin{pmatrix} 0{,}75 & -0{,}30 \\ 0{,}02 & 0{,}13 \end{pmatrix}, \qquad w_{01} = \begin{pmatrix} 0{,}2 \\ 0{,}1 \end{pmatrix}.$$

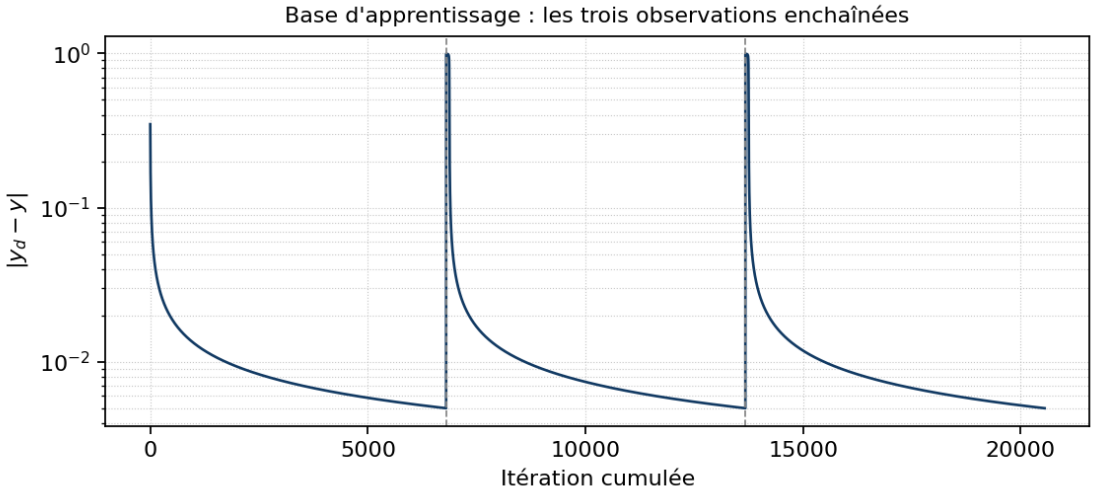

Fig. 4. Base d'apprentissage : erreur au cours des trois phases d'apprentissage enchaînées. Les traits verticaux marquent le passage d'une observation à la suivante.

Le fait que w_1 et w_{01} ne bougent pas d'un seul chiffre significatif, alors que la couche de sortie apprend normalement, n'est pas un défaut du programme mais un phénomène physique de l'algorithme, analysé à la section 6.3.

6 Discussion

6.1 Vitesse de convergence des exemples 1 et 3

Les exemples 1 et 3 convergent en quelques milliers d'itérations et en moins de 0,4 s. L'exemple 3, malgré une tolérance dix fois plus stricte que celle de l'exemple 1 (10^{-3} contre 10^{-2}) et une cible plus exigeante, converge en un temps comparable grâce à son taux d'apprentissage plus élevé ($\alpha = 1{,}5$ contre $\alpha = 1$).

C'est le comportement attendu d'une descente de gradient à pas fixe : la longueur du pas est proportionnelle à α, donc un α plus grand accélère la descente, tant qu'il ne provoque pas d'oscillations autour du minimum. Il existe une valeur critique au-delà de laquelle le pas dépasse systématiquement le minimum et l'apprentissage diverge ; ni l'un ni l'autre de ces exemples ne s'en approche, ce qui explique que le gain soit ici purement bénéfique.

6.2 Le cas pathologique de l'exemple 2

L'exemple 2 demande 528 537 itérations et 36 s, soit un facteur voisin de cent par rapport aux deux autres. Trois causes se conjuguent, et il est instructif de les séparer.

Biais désactivés. Avec $x_0 = z_0 = 0$, les termes $w_{01}x_0$ et $w_{02}z_0$ des équations (2) et (3) disparaissent. Le réseau perd un degré de liberté par neurone, celui qui permet d'ajuster le seuil de sa fonction d'activation. Toutes les frontières de décision sont contraintes à passer par l'origine, ce qui n'est pas toujours compatible avec la cible.

Initialisation symétrique. Les deux lignes de w_1 sont identiques, donc les deux neurones cachés reçoivent la même pré-activation et produisent la même sortie. D'après (6), ils reçoivent alors le même gradient et subissent la même correction : ils restent identiques à toutes les itérations. La symétrie n'est jamais brisée, et le réseau se comporte comme s'il n'avait qu'un seul neurone caché, avec la capacité de représentation correspondante. Les poids finaux confirment le diagnostic, les deux lignes de w_1 restant égales à $(1{,}066\,;\,1{,}631)$.

Cible à 0,5. La sortie désirée comporte une composante $y_{d,1} = 0{,}5$. Or la sigmoïde ne vaut exactement 0,5 que pour une pré-activation rigoureusement nulle. Le réseau doit donc annuler *asymptotiquement* l'activation du premier neurone de sortie, ce qui est intrinsèquement lent : plus l'on s'approche, plus le gradient diminue. Les poids finaux montrent que le réseau y parvient en annulant la première ligne de w_2, qui se stabilise à $(0\,;\,0)$.

Cet exemple, sans doute conçu pour cela, illustre l'importance pratique de trois règles aujourd'hui systématiques en apprentissage automatique : initialiser les poids de façon aléatoire et non symétrique, maintenir les biais actifs, et formuler les cibles dans une plage que la fonction d'activation atteint effectivement.

6.3 Saturation de la sigmoïde sur la base d'apprentissage

Sur la base d'apprentissage, les poids de la couche cachée w_1 et w_{01} ne bougent pas du tout. L'explication est entièrement contenue dans le facteur $U_{out} \odot (1 - U_{out})$ de l'équation (6).

Les entrées sont brutes et non normalisées, de l'ordre de $x \approx 230$. Avec des poids initiaux de l'ordre de l'unité, la pré-activation des neurones cachés vaut, pour la troisième observation par exemple,

$$U_{in} = w_1 x + w_{01}x_0 = \begin{pmatrix} 0{,}75 & -0{,}30 \\ 0{,}02 & 0{,}13 \end{pmatrix} \begin{pmatrix} 235 \\ 124 \end{pmatrix} + \begin{pmatrix} 0{,}2 \\ 0{,}1 \end{pmatrix} = \begin{pmatrix} 139{,}25 \\ 20{,}92 \end{pmatrix}. \tag{10}$$

À ces valeurs, la sigmoïde vaut 1 à la précision machine près, et sa dérivée $U_{out}(1-U_{out})$ est numériquement nulle : $f'(139) \approx 10^{-61}$ et $f'(20{,}9) \approx 8 \times 10^{-10}$. D'après (6), le gradient local δ_1 est donc nul, et d'après (8) les poids w_1 et w_{01} ne reçoivent aucune correction. Le gradient ne traverse plus la couche cachée : la figure 2 en donne la lecture graphique.

Tout l'apprentissage est alors porté par la seule couche de sortie, w_2 et w_{02}, qui suffit ici à faire converger chaque observation individuellement, puisque les deux sorties cachées sont saturées à 1 et que le réseau se réduit de fait à un neurone unique disposant de deux entrées constantes et d'un biais.

Il s'agit d'une limite classique des réseaux à sigmoïde, connue sous le nom de problème du gradient évanescent, et la parade est tout aussi classique : il faut **normaliser les entrées**, par exemple les ramener dans $[0, 1]$ ou les centrer-réduire, pour que les pré-activations restent dans la zone où la dérivée de la sigmoïde est significative. Avec les données de cet exemple, une simple division par 255 suffirait à ramener U_{in} dans un intervalle exploitable et à rendre la couche cachée effectivement entraînable.

6.4 Intérêt de la version vectorisée

La version vectorisée remplace trois boucles `for` imbriquées par quatre produits matriciels par itération. Sous MATLAB, dont l'interpréteur est optimisé pour l'algèbre linéaire et délègue les produits matriciels à des bibliothèques compilées, c'est à la fois la forme la plus rapide et la plus lisible.

Elle présente en outre un avantage direct au regard de l'énoncé : elle traite des réseaux de dimensions quelconques sans modification d'une seule ligne de code, puisque les dimensions n'apparaissent nulle part explicitement, alors que la version à boucles doit connaître S, K et N pour écrire ses bornes. C'est exactement l'exigence de généralité demandée. La version à boucles conserve néanmoins son intérêt pédagogique : elle rend visible ce que le produit matriciel condense, et c'est elle qui permet de vérifier les équations une à une.

7 Conclusion

Ce travail pratique a permis de construire un perceptron multicouche complet en MATLAB, de la propagation avant à la rétropropagation du gradient, sous une forme matricielle générale : le même programme résout les quatre jeux de données sans aucune modification de son cœur de calcul, ce qui répond à l'exigence de l'énoncé.

Au-delà de la programmation, l'analyse des résultats a mis en évidence plusieurs phénomènes fondamentaux de l'apprentissage des réseaux de neurones, que les jeux de données proposés semblent avoir été construits pour révéler.

L'influence du taux d'apprentissage sur la vitesse de convergence apparaît en comparant les exemples 1 et 3 : un α plus élevé compense une tolérance plus stricte. Le blocage de l'exemple 2 montre l'effet combiné d'une initialisation symétrique, qui empêche définitivement les deux neurones cachés de se différencier, de l'absence de biais, qui contraint les frontières de décision à passer par l'origine, et d'une cible que la sigmoïde n'atteint qu'asymptotiquement. La base d'apprentissage illustre enfin la saturation de la sigmoïde en présence d'entrées non normalisées : le calcul explicite des pré-activations, de l'ordre de 10^2, montre que la dérivée de l'activation tombe sous 10^{-9} et que la couche cachée cesse purement et simplement d'apprendre.

Ces trois observations rejoignent les bonnes pratiques employées aujourd'hui de façon systématique : initialisation aléatoire des poids pour briser la symétrie, maintien des biais, normalisation des données d'entrée et choix soigné du taux d'apprentissage. Les avoir rencontrées comme des difficultés concrètes, sur un programme écrit intégralement à la main, leur donne un contenu qu'un exposé théorique transmet difficilement.

Un prolongement naturel de ce travail consisterait à ajouter la normalisation des entrées et une initialisation aléatoire, puis à mesurer de nouveau le nombre d'itérations sur les quatre jeux de données : on s'attend à ce que l'exemple 2 retrouve un coût comparable aux autres et à ce que la couche cachée de l'exemple 4 participe enfin à l'apprentissage.

Références

[1] D. E. Rumelhart, G. E. Hinton et R. J. Williams, « Learning representations by back-propagating errors », *Nature*, vol. 323, p. 533–536, 1986.

[2] S. Haykin, *Neural Networks and Learning Machines*, 3e éd. Upper Saddle River, Pearson, 2009.

[3] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York, Springer, 2006.

[4] Y. LeCun, L. Bottou, G. B. Orr et K.-R. Müller, « Efficient BackProp », dans *Neural Networks : Tricks of the Trade*, Berlin, Springer, 1998, p. 9–50.

[5] I. Goodfellow, Y. Bengio et A. Courville, *Deep Learning*. Cambridge, MIT Press, 2016.

[6] X. Glorot et Y. Bengio, « Understanding the difficulty of training deep feedforward neural networks », *Proc. AISTATS*, p. 249–256, 2010.

A Jeux de données complets

Tab. 6. Valeurs initiales des quatre jeux de données.

	Exemple 1	Exemple 2	Exemple 3	Exemple 4
x	$(0\,;\,0{,}7)$	$(0{,}5\,;\,1)$	$(0{,}05\,;\,1)$	3 obs., cf. tab. 5
x_0	1	0	1	1
z_0	1	0	1	1
w_1	$\begin{pmatrix} 0{,}7 & -0{,}4 \\ -0{,}2 & 0{,}3 \end{pmatrix}$	$\begin{pmatrix} 0{,}5 & 0{,}5 \\ 0{,}5 & 0{,}5 \end{pmatrix}$	$\begin{pmatrix} 0{,}15 & 0{,}20 \\ 0{,}25 & 0{,}30 \end{pmatrix}$	$\begin{pmatrix} 0{,}75 & -0{,}30 \\ 0{,}02 & 0{,}13 \end{pmatrix}$
w_{01}	$(0{,}4\,;\,0{,}7)$	$(0\,;\,0)$	$(0{,}35\,;\,0{,}35)$	$(0{,}2\,;\,0{,}1)$
w_2	$(0{,}5\ \ 0{,}1)$	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0{,}40 & 0{,}45 \\ 0{,}50 & 0{,}55 \end{pmatrix}$	$(0{,}52\ \ 0{,}11)$
w_{02}	$-0{,}3$	$(0\,;\,0)$	$(0{,}60\,;\,0{,}60)$	0
y_d	1	$(0{,}5\,;\,1)$	$(0{,}01\,;\,0{,}99)$	$\{1,\,0,\,1\}$
α	1	0,5	1,5	1
tol	0,01	0,001	0,001	0,005

B Reproduction des résultats

Le code est disponible sous licence MIT à l'adresse `https://github.com/KhalilEllahLAGHA/mlp-backpropagation-matlab`. Pour reproduire les résultats de la section 5, exécuter `pmcmain.m` sous MATLAB R2024b et sélectionner l'un des quatre jeux de données dans le menu. Les courbes de convergence des figures 3 et 4 sont obtenues en enregistrant $\max_j |e_j|$ à chaque passage dans la boucle du listing 1.