République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITÉ MOHAMED KHIDER, BISKRA

FACULTÉ des SCIENCES EXACTES et des SCIENCES de la NATURE et de la VIE

DÉPARTEMENT DE MATHÉMATIQUES

Mémoire présenté en vue de l'obtention du Diplôme **:**

MASTER en Mathématiques

Option : **Analyse**

Par

GHODBANE RADHIA

Titre :

Résolutions des systèmes linéaires par des réseaux de neurones artificiels

Membres du Comité d'Examen :

Dr.	Laadjal Baya	UMKB	Président
Pr.	Khelil Naceur	UMKB	Encadreur
Dr.	Adouane Saida	UMKB	Examinateur

19 Juin 2023

Dédicace

Je dédie ce modeste travail à :

mes chèrs parents.

Houcine & Darmouna

mes sœurs

Wassila, Wafa, Wissam, Hamida, Souad

mes frères

Ahmed, Ammar

mes oncles, leurs femmes et leurs enfants

mes tantes et leurs enfants

mes âme sœur

Wanassa ,Fatima, Sabrina, Hayat

tous mes amis et mes camarades

Zoulikha, Samia, Amira,Salim, Ramdhane, Halim, Amine

Et à toute ma famille

GHODBANE & DHAHOUA

REMERCIEMENTS

Je tiens tout d'abord à remercier Pr. Khelil Naceur à l'Université de Biskra, qui m'a donné ce sujet et m'a encadré pendant mon Master. Il a toujours été à mon écoute et son point de vue complémentaire est souvent été trés utile.

Je suis trés reconnaissante envers Dr. Laadjal Baya, et Dr. Adoune Saida d'avoir manisfesté de l'interêt pour mon travail en acceptant de participer à ce jury. Leurs remarques et commentaires constructifs m'ont permis d'en améliorer le manuscrit.

Enfin, que toutes les personnes qui ont partcipé de prés ou de loin à la réalisation de ce travail trouvent ici l'expression de mes sincères remerciements.

Mes derniers remerciements vont à mes parents, mes frères et soeurs et tous les amies.

Table des matières

Table des figures

Introduction

Motivation

Plusieurs méthodes peuvent être utilisées pour résoudre des systèmes linéaires. Une des options consiste à utiliser des solveurs linéaires préexistants. Dans ce mémoire, nous proposons une deuxième option, qui construit des réseaux de neurones. Ces réseaux de neurones sont créés en raison de la capacité à travailler avec des fonctions de perte personnalisées et calculer des estimations de x pour des problèmes à grande échelle.
Ainsi, le but de ce mémoire est de résoudre un système linéaire $Ax = b$ avec un réseau de neurones et discuter des avantages de l'utilisation des réseaux de neurones par rapport aux solveurs linéaires.

Ce mémoire commence par une brève introduction aux systèmes d'équations linéaires. Nous présenterons alors différents solveurs linéaires pour résoudre ces systèmes linéaires. Pour la mise en place des réseaux de neurones nous utilisons Matlab. Par conséquent. Nous approfondissons les connaissances des fonctions d'algèbre qui nous aident à trouver une solution x pour le problème $Ax = b$.
Dans Matlab, vous pouvez résoudre des systèmes d'équations linéaires à l'aide de l'opérateur antislash ou de la fonction linsolve.

Nous présentons ensuite les caractéristiques des réseaux non linéaires en profondeur car notre réseau linéaire adopte certaines des principales caractéristiques des réseaux non li-

néaires. La raison pour laquelle nous nous référons au réseau de neurones spécifiquement comme linéaire est que les réseaux de neurones représentent généralement des solutions aux problèmes non linéaires. Cependant, notre réseau représente une solution pour un système linéaire. Notre approche de la construction de réseaux de neurones est principalement basée sur l'article "Neural réseaux pour résoudre des systèmes d'équations linéaires et des problèmes connexes » 1 qui a été écrit en 1992 par Cichocki et Unbehauen. La théorie derrière ce travail est de résoudre itérativement un système d'équations linéaire $Ax = b$, afin de converger vers une solution x la plus proche possible de la solution exacte. Normalement, nous attendons d'un réseau de neurones qu'il s'entraîne avec un ensemble de données qui a de nombreuses informations. La raison en est d'acquérir un réseau aussi précis que possible pour résoudre tout autre problème connexe. Dans notre implémentation, nous travaillons avec une méthode différente. Plutôt que d'alimenter le réseau par différentes matrices A et b, on se contente d'une matrice fixe A, et d'un vecteur fixe b comme échantillons d'apprentissage.

Après avoir expliqué l'approche de Cichocki et Unbehauen, nous démontrons la réimplémentation de la méthode de résolution itérative.
L'un des avantages les plus cruciaux de l'utilisation de la méthode de résolution itérative est que nous pouvons bénéficier des fonctions de perte personnalisées. Dans ce mémoire, quatre fonctions énergétiques différentes, seront présentés, à savoir ; moindre carrés ordinaire, moindre carrés itérativement repondéré, la moindre valeur absolue et le problème de chebyshev, seront expliqués en détail. Ces fonctions énergétiques utilisées dans les travaux de Cichocki et Unbehauen servent le devoir d'une fonction de perte, et nous nous implémentons que celle des moindres carré, et enfin nous donerons une conclusion

Chapitre 1

Systèmes linéaires

1.1 Systèmes linéaires

Dans cette section, nous décrivons d'abord ce qu'est-une équation linéaire. Ensuite, nous passons aux systèmes d'équations linéaires. Nous fournissons également la forme matricielle d'un système linéaire.

– Équations linéaires

Une équation linéaire à variables $x_1, x_2, ..., x_n$ a la forme : $a_1x_1 + a_2x_2 + +a_nx_n = b$; où $a_i \in R$ représente les coefficients de x_i et $b \in R$ représente le terme constant 5. On l'appelle une équation linéaire parce que l'ensemble des solutions forme une ligne droite dans l'hyperplan.

– Système d'équations linéaires

Un ensemble finie d'équations linéaires avec les mêmes variables est un système d'équations linéaires. Un exemple de système linéaire avec n variables différentes et n équations différentes peut être défini comme :

$$a_{11}x_1 + a_{12}x_2 + +a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + +a_{2n}x_n = b_2$$

.

.

.

$$a_{n1}x_1 + a_{n2}x_2 + + a_{nn}x_n = b_n$$

On peut démontrer que ce système linéaire peut se mettre sous la forme

$$\begin{bmatrix} a_{11} & a_{12} & .. & a_{1n} \\ a_{21} & a_{22} & .. & a_{2n} \\ .. & .. & .. & .. \\ a_{n1} & a_{n2} & .. & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ . \\ b_n \end{bmatrix} \tag{1.1}$$

ou en notation matricielle $Ax = b$.

Un ensemble de solutions $x = (x_1, x_2, ..., x_n)$ est un tuple qui fait de chaque équation une affirmation vraie 5 .

1.2 Solveurs de Matlab

Dans Matlab, vous pouvez résoudre des systèmes d'équations linéaires à l'aide de l'opérateur *antislash* ou de la fonction *linsolve*.

Voici un exemple d'utilisation de l'opérateur barre oblique inverse :

```
1  % Define the coefficient matrix A and the right-hand side vector b
2  A=[1 2 3; 4 5 6; 7 8 9];
3  b=[10; 11; 12];
4 % Solve the system of equations Ax = b
5 x=linsolve(A,b);
6 % Display the solution
7  disp(x);
```

Voici un exemple d'utilisation de l'opérateur antislash :

```
1   % Define the coefficient matrix A and the right-hand side vector b
2 A=[1 2 3; 4 5 6; 7 8 9];
3  b=[10; 11; 12];
4 % Solve the system of equations Ax = b
5 x=A\b;
6 % Display the solution
7  disp(x);
```

1.2.1 Decomposition LU

Le processus de décomposition LU calcule U, qui est une matrice triangulaire supérieure, et L, qui est une matrice triangulaire inférieure, de sorte que $A = LU$ 5. Le calcul de la solution peut être fournie par la décomposition LU . Ainsi, A est factorisé avec la décomposition LU, puis x est résolu avec des substitutions avant et arrière.

Dans Matlab, vous pouvez utiliser la fonction intégrée lu pour calculer la décomposition LU d'une matrice.

Voici un exemple de code qui montre comment utiliser la fonction lu :

```
1   A = [4 3; 6 3];
2  [L, U, P] = lu(A);
```

Dans cet exemple, A est la matrice que nous voulons décomposer, et L, U et P sont les matrices représentant respectivement la matrice triangulaire inférieure, la matrice triangulaire supérieure et la matrice de permutation.
Notez que la fonction lu renvoie la matrice de permutation P car l'algorithme utilisé pour calculer la décomposition peut impliquer des échanges de lignes.

Après avoir exécuté le code ci-dessus, vous pouvez accéder aux matrices L, U et P par leurs noms de variables. Par exemple, pour afficher L et U, vous pouvez utiliser la fonction disp :

```
1  affichage('L = ');
2  affichage(L);
3 affichage('U = ');
4  affichage(U);
```

1.2.2 Méthode des moindres carrés

Si la matrice A n'est pas carrée et n'a pas de rang complet, on utilise la méthode des moindres carrés
Pour effectuer une régression des moindres carrés dans MATLAB, vous pouvez utiliser la fonction $polyfit$. Cette fonction peut ajuster un polynôme d'un degré spécifié à vos données et renvoyer les coefficients du polynôme qui minimisent la somme des erreurs au carré.
Voici un exemple d'utilisation de polyfit :

```
% Sample data
x = [1 2 3 4 5];
y = [1.5 3.5 5.5 7.5 9.5];
% Fit a polynomial of degree 1 (i.e., a straight line)
p = polyfit(x, y, 1);
% Evaluate the fitted line at some points
x_eval = linspace(0, 6, 100);
y_eval = polyval(p, x_eval);
% Plot the data and fitted line
plot(x, y, 'o', x_eval, y_eval)
xlabel('x')
ylabel('y')
legend('Data', 'Fitted line')
```

Dans cet exemple, nous créons deux vecteurs x et y qui représentent nos exemples de données. Nous appelons ensuite $polyfit$ avec les arguments x, y et 1 pour ajuster une ligne droite aux données. Les coefficients résultants sont stockés dans p. Nous utilisons ensuite *polyval* pour évaluer la droite d'ajustement en 100 points équidistants entre 0 et 6. Enfin, nous traçons les points de données et la droite d'ajustement.

Vous pouvez ajuster le degré du polynôme en changeant le troisième argument en $polyfit$. Par exemple, pour ajuster une courbe quadratique aux données, vous utiliseriez plutôt $polyfit(x, y, 2)$.

La méthode des moindres carrés est une technique de résolution d'un ensemble d'équations surdéterminé $Ax = b$, où $A \in R^{m \times n}$ est le modèle matriciel, $b \in R^m$ est le vecteur de mesure, et $x \in R^n$ est le vecteur inconnu 1. A et b sont donnés à l'avance et x est inconnu. Nous pouvons définir le modèle d'estimation linéaire comme suit :

$$Ax = b + r = b_{true} \tag{1.2}$$

où $r \in R^m$ est le vecteur inconnu des erreurs de mesure que l'on veut minimiser et $b_{true} \in R^m$ est le vecteur des valeurs exactes 3.

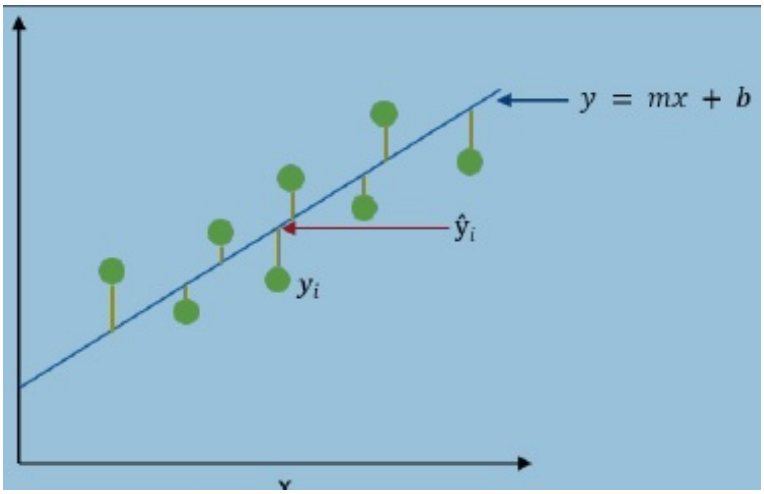

FIG. 1.1 – Régression des moindres carrés

La figure 1.1 montre comment la méthode des moindres carrés est calculée pour un problème. Supposons que notre ensemble de données se compose de n points de données $(x_i, y_i), i = 1, ..., n$. Les coefficients de x représente A et y représente b_{exacte} en 1.2. Le jeu de données est représenté par les points verts. x_i est une variable indépendante et $y_i$est une variable dépendante calculée par observation. L'objectif est d'obtenir une ligne de meilleur ajustement à un ensemble de données donné qui minimise les résidus r. La différence entre la sortie réelle y_i et la sortie prédite par le modèle $\hat{y}_i$, démontré avec des lignes orange, nous donne le r_i. Selon le modèle d'estimation 1.2, $\hat{y}$ correspond à b. La valeur prédite par le modèle peut également être définie comme $f(x_i, \beta)$. Pour un problème bidimensionnel, l'ordonnée à l'origine peut être notée 1 et la pente 0. Ainsi, $f(x_i, \beta) = \beta_0 x + \beta_1$. Afin de trouver les paramètres optimaux, par exemple 0 et 1 pour le problème donné, la somme des carrés des résidus est calculée :

$$s = \sum_{i=1}^{n} r_i^2$$

et la méthode essaie de minimiser S. La méthode des moindres carrés est sensible aux valeurs aberrantes 1. S'il y a des valeurs aberrantes dans les données, la ligne sera décalée dans la direction de l'erreur.

Chapitre 2

Les réseaux de neurones

Dans cette section, nous examinons plus en détail les réseaux de neurones non linéaires. La méthode itérative que nous avons proposé dans ce mémoire pour résoudre des systèmes linéaires adopte les caractéristiques générales des réseaux de neurones non linéaires. Par conséquent, il est crucial de discuter des concepts concernant réseaux de neurones non linéaires dans un premier temps.

2.1 Introduction aux réseaux de neurones

Les réseaux de neurones artificiels attirent l'attention en raison de leurs performances exceptionnelles en matière de reconnaissance de formes et de modélisation de relations non linéaires impliquant un grand nombre de variables. Le cerveau humain est l'inspiration pour créer des réseaux de neurones artificiels. Ainsi, un réseau de neurones est un modèle informatique avec une architecture qui imite essentiellement les compétences organisationnelles du cerveau humain et l'acquisition de connaissances. Les couches d'un réseau neuronal artificiel représentent le réseau de neurones dans le cerveau 6. Le principe de fonctionnement d'un neurone biologique est de recevoir des signaux électriques d'autres neurones de poids différents. Le neurone est déclenché si la valeur de tous les signaux

électriques est suffisamment grande, sinon, le neurone est dans un état inactif. Basé sur le principe de la façon dont le biologique les neurones fonctionnent, les neurones artificiels sont conçus pour avoir des états dynamiques qui peuvent contenir des valeurs.

Il existe trois composants fondamentaux d'un réseau de neurones artificiels, à savoir : la couche d'entrée, la couche cachée et la couche de sortie. La couche d'entrée est l'endroit où le réseau rencontre l'entrée présentée et la couche de sortie conserve la réponse du réseau à l'entrée. Les couches intermédiaires, également appelées couches cachées, nous permettent de calculer des *mappages* compliqués entre les motifs. Ces couches sont des éléments de traitement interconnectés, appelés neurones. Les neurones d'un réseau de neurones artificiels interagissent les uns avec les autres via des connexions pondérées et ils forment ensemble des couches. Chaque neurone de la couche précédente est connecté à tous les neurones de la couche suivante.

Le nombre de neurones et de couches dépend de l'utilisateur et de l'application spécifique. Par exemple, si la couche cachée possède un nombre réduit de nœuds, le réseau ne sera pas en mesure d'effectuer correctement la tâche pour laquelle il est formé. En revanche, si la couche cachée possède un nombre élevé de neurones, le réseau ne pourra pas généraliser les données données et le temps d'apprentissage sera plus long 3. C'est parce que les modèles seront mémorisés par le réseau, ce qui est un problème d'apprentissage en profondeur 3. Nous voulons que notre réseau estime les sorties généralement à partir de l'ensemble de données, plutôt que de les mémoriser.

2.1.1 Présentation générale : Entraînement Tests

Afin de former un réseau, un ensemble de données est donné avec l'entrée réelle et la sortie correspondante. Pendant la formation, les informations sont transmises de la couche d'entrée aux couches cachées et la sortie est donnée par la couche la plus externe 3. Le but du réseau est de faire une estimation aussi précise que possible pour une entrée

donnée. Ainsi, un nombre élevé de données est fourni au modèle en s'aventurant dans l'augmentation du temps. Le nombre de données d'entraînement doit être inférieur à cinq à dix fois le nombre de connexions dans le réseau 3. Un autre aspect pour obtenir une solution optimale consiste à alimenter les données de manière aléatoire à chaque itération pendant la formation. Le réseau généralisera l'information et ne sera pas biaisé par l'ordre de celle-ci 3. L'ensemble de données complet est transmis au réseau pendant la formation en fonction du nombre d'époques. Une époque, également appelée cycle, est un passage à travers l'ensemble de données d'apprentissage avec la mise à jour des poids. Le nombre d'époques dépend de l'application spécifique. A la fin de chaque cycle, les poids sont mis à jour. Une fois la formation terminée, le réseau devrait pour produire des sorties précises avec un minimum d'erreurs concernant l'ensemble de données donné. Pour utiliser le modèle pour différents ensembles de données, les poids sont stockés dans le réseau . Le réseau est testé avec un ensemble de données de test, également appelé ensemble de tests, pour évaluer les performances du réseau formé avec des pondérations formées. Pendant la phase de test, aucun apprentissage et l'ajustement du poids a lieu. Afin d'évaluer les résultats, au moins 10 à 20 % de l'ensemble de données 3 doivent être conservés. Les prédictions du modèle sont comparées aux valeurs de sortie cibles. Ils doivent être fiables étant donné que les valeurs d'entrée se situent dans la plage de l'ensemble de données d'apprentissage. Si le modèle prédit la sortie avec précision, il est établi que le modèle généralise l'information avec succès et on peut lui faire confiance. Après avoir acquis des résultats pratiques, le modèle peut être utilisé dans des applications pratiques .

2.1.2 Le Perceptron : Propagation vers l'avant

Le traitement de l'information dans un réseau neuronal commence par l'envoi des données à la couche d'entrée. En raison de la structure des couches interconnectées, les données d'entrée sont traitées avec les poids associés. La sortie de chaque neurone est produite en multipliant l'entrée du neurone et le vecteur de poids correspondant. Ensuite, le ré-

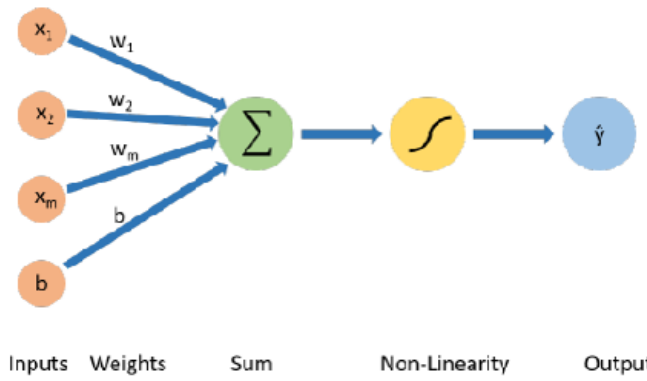

FIG. 2.1 – Perceptron

sultat est passé à travers une fonction d'activation et il est fourni à la couche suivante en tant qu'entrée. Par conséquent, le nombre de neurones dans la sortie de la couche précédente est égal à l'entrée dans la couche actuelle. À la fin, les données que nous voulons obtenir sont les valeurs des neurones de la dernière couche. Nous pouvons démontrer le processus mentionné avec un modèle, appelé le perceptron. Perceptron est le bloc de construction structurel de l'apprentissage en profondeur qui démontre le modèle d'un neurone biologique. La figure 2.1 montre un exemple de perceptron. L'entrée est notée par $x = [x_1, x_2, \cdots, x_m] \in R^m$ avec m entrées et il y a aussi un biais b. Le biais est une forme spéciale de poids et il est optimisé pendant le processus d'entraînement avec les poids. Les poids définies comme $w = [w_1, w_2, \cdots, w_m] \in R^m$, connectent l'entrée correspondante à la prochaine couche. La force des relations entre les neurones interconnectés est déterminée par les poids. Dans les réseaux de neurones multicouches, il existe deux opérations fondamentales pour la modélisation et le process us de formation, à savoir la propagation vers l'avant et la rétropropagation 3.

Ici, nous élaborons la propagation vers l'avant. La logique derrière la propagation vers l'avant est pour additionner la multiplication des entrées avec les poids correspondants et le biais. On peut noter la somme avec z :

$$z = b + \sum_{i=1}^{m} x_i w_i$$

Le biais est une constante qui aide à décaler la fonction d'activation. La valeur z est passée à une fonction d'activation notée g, où l'état de réponse du neurone est simulé. Ainsi, nous obtenons la sortie y :

$$y = g(z)$$

Globalement, la sortie d'un seul perceptron est calculée avec :

$$y = g(b + \sum_{i=1}^{m} x_i w_i) \tag{2.1}$$

Le but de la propagation vers l'avant est de générer une prédiction et de calculer la perte entre-temps. Le calcul de la perte est décrit en détail en 2.1.4.

2.1.3 Fonctions d'activation

Les fonctions d'activation ont un rôle crucial pour les réseaux de neurones artificiels. Ils permettent au réseau d'apprendre en fournissant des fonctions non linéaires entre l'entrée et la sortie correspondante. Outre les fonctions d'activation non linéaires, il existe également des fonctions d'activation linéaires. Les fonctions linéaires ne sont généralement pas préférées car le réseau ne peut s'adapter qu'aux changements linéaires dans l'entrée. Cependant, dans le monde réel, la plupart des relations ne sont pas linéaires. Les fonctions d'activation non linéaires permettent au réseau d'apprendre les dépendances non linéaires dans les données. En d'autres termes, les données qui ne peuvent pas être classées linéairement peuvent être différenciées à l'aide de données fonctions d'activation non linéaires

.

Dans ce mémoire, nous nous concentrerons uniquement sur les fonctions d'activation linéaires, car nous nous concentrons sur la solution aux problèmes linéaires. Néanmoins, les fonctions de perte dont nous discutons pour ces systèmes linéaires peuvent être non linéaires et, en principe, peuvent également être traitées comme une fonction d'activation

non linéaire. Par conséquent, il est utile de discuter également des activations non linéaires à ce stade. Dans la rétropropagation, les dérivées des fonctions d'activation dans chaque couche doivent être calculées. Par conséquent, il est crucial que les fonctions d'activation soient différentiables presque partout avec $h : R \rightarrow R$.

Il existe plusieurs fonctions d'activation qui sont utilisées dans différents scénarios et chacune d'elles a des caractéristiques différentes. Pour presque tous les paramètres, choisir la bonne fonction pour un réseau spécifique nécessite des essais et des erreurs, car il n'existe aucune règle sur la fonction d'activation qui fonctionnera le mieux dans différents scénarios. Nous élaborons ici les fonctions d'activation les plus courantes et examinons les avantages et les inconvénients 3 .

1. Fonction d'activation linéaire

   La fonction d'activation est appelée linéaire car elle est proportionnelle à l'entrée donnée.

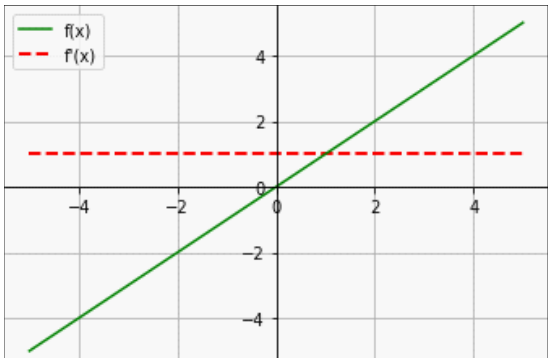

FIG. 2.2 – Fonction d'activation linéaire et la dérivée

– avantage : Une fonction d'activation linéaire a une dérivée m, en considérant qu'une ligne linéaire a la fonction $y = mx + c$. La valeur constante m peut être choisie arbitrairement. La figure 2.2 est un exemple de fonction linéaire de pente $m = 1$ et $c = 0$.

– Désavantage : La dérivée de la fonction sera toujours la même. Il ne suffit pas d'ajuster les poids et les biais lors de la rétropropagation avec une constante valeur indépendante de l'entrée. Par conséquent, les fonctions d'activation linéaires fonctionneront mal pour les tâches non linéaires, car l'erreur du modèle ne sera pas améliorée à chaque itération.

2. Fonctions d'activation non linéaires

a) La fonction d'activation sigmoïde, également connue sous le nom de courbe logistique, est une fonction en forme de s non linéaire . C'est l'une des fonctions d'activation les plus courantes. Comme le montre la figure 2.3 , la sortie est mise à l'échelle entre 0 et 1 comme une distribution de probabilité .

La fonction sigmoïde est définie comme :

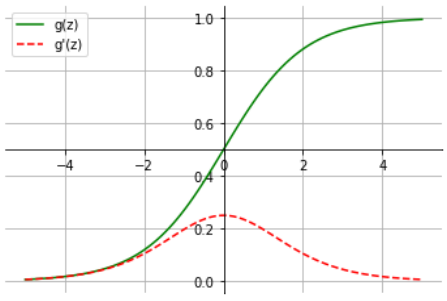

FIG. 2.3 – Sigmoïde et la dérivée

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

et la dérivée de la fonction est : $\frac{d}{dz}g(z) = g(z)(1 - g(z))$

Les valeurs de sortie ont le même signe. Le résultat le plus probable pour le modèle d'entrée donné est la sortie avec la plus grande valeur .

Avantage :

– Il est bien adapté aux problèmes, où nous devons prédire la probabilité comme une sortie.

Disadvantage :

– Il existe un problème de gradient de fuite car il en résulte un gradient nul dans la limite :

$lim_{z\to\infty}g(z) = 0, lim_{z\to\infty}g'(z) = 0$

Pendant la rétropropagation, les sorties sont chaînées par rapport à l'algorithme de descente de gradient les unes des autres dans la direction des couches internes. Par conséquent, les couches internes ont des gradients décroissants vers 0, ce qui les amène à moins contribuer au processus d'apprentissage. À la fin, le réseau peut avoir des résultats inexacts et le calcul des sorties peut prendre plus de temps. Par conséquent, la fonction d'activation sigmoïde est utilisée au niveau de la sortie plutôt que dans les couches cachées.

b) tanh La fonction d'activation de tanh ou de tangente hyperbolique présente des similitudes avec le sigmoïde, comme être en forme de s.

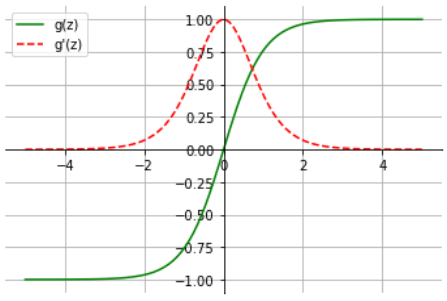

FIG. 2.4 – tanh et sa dérivée

comme étant en forme de S. Cependant, comme le montre la figure 2.4, cette fonction est symétrique à l'origine, ce qui signifie que les valeurs de sortie des neurones peuvent avoir des signes différents. La sortie est mise à l'échelle entre -1 et 1.

La fonction Tanh est définie comme : $g(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$

et la dérivée de la fonction est : $g\prime(z) = 1 - \tanh^2(z)$

Avantage :

– La moyenne des données d'entrée est approximativement nulle. Autrement dit, l'entrée les données sont normalisées. Par conséquent, la convergence est plus ra-

pide que sigmoïde , ce qui rend la fonction d'activation de tanh plus préférable. De plus, tanh réalise une erreur de classification inférieure .

Désavantage :

– Comme la fonction d'activation sigmoïde, il existe un problème de gradient de fuite. Par conséquent, en utilisant les fonctions d'activation sigmoïde et tanh sur les couches cachées n'est pas favorisé.

c) ReLU Selon des recherches dans le domaine des neurosciences, seuls un à quatre pour cent des neurones sont dans un état actif en même temps. En revanche, les fonctions d'activation telles que sigmoïde et tanh activent près de la moitié des neurones du réseau en même temps. Étant donné que chaque neurone dans un état actif passe par des opérations telles que la propagation vers l'avant et la rétropropagation, il devient difficile pour le réseau de s'entraîner. Rectified linear unit est introduite pour améliorer l'efficacité concernant le problème mentionné.

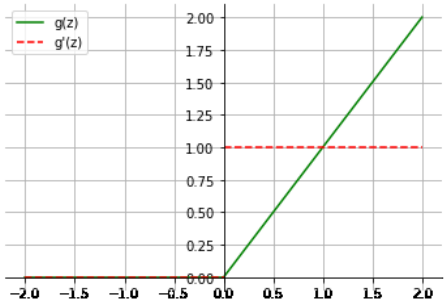

FIG. 2.5 – ReLU et sa dérivée

ReLU est l'une des fonctions d'activation les plus largement utilisées et elle est définie comme : $g(z) = \max(0, z) = \begin{cases} z \text{ si } z \geq 0 \\ 0 \text{ si } z < 0 \end{cases}$

La dérivée de ReLU est :$g\prime(z) = \begin{cases} 1 \text{ si } z \geq 0 \\ 0 \text{ si } z < 0 \end{cases}$

Avantages :

- Les neurones sont partiellement activés . Par conséquent, l'efficacité est plus élevée, puisqu'à chaque itération le réseau n'exploite pas tous les nœuds. Si la sortie de la fonction d'activation est 0, le neurone est désactivé comme le montre la figure 2.5 , ce qui signifie que la sortie du neurone ne sera pas transmise comme entrée à la couche suivante. La fonction d'activation $ReLU$ ne peut être utilisée que dans les couches cachées du réseau.
- Puisque la dérivée de la fonction est 1, lorsque l'entrée $x > 0$, le problème du gradient de fuite est résolu . Le processus de formation est plus rapide et c'est pourquoi $ReLU$ est préféré la plupart du temps. Dans les réseaux de neurones profonds, les fonctions d'activation sigmoïde et tanh sont remplacées par $ReLU$ et ELU pour résoudre le problème du gradient de fuite et accélérer la vitesse de convergence .
- Contrairement à sigmoïde et tanh, il est moins coûteux de calculer ReLU car la fonction d'activation ne contient pas de fonction exponentielle.

Désavantages :

- Un neurone est désactivé lorsque l'entrée x est inférieure à 0 empêchant les mises à jour sur les poids relatifs .
- Étant donné que la fonction a une nature linéaire, elle n'est pas bien adaptée aux problèmes de classification comme tanh et sigmoïde.

d) ELU est une variante de $ReLU$, qui signifie unité linéaire exponentielle. Le but est de diminuer l'effet de décalage de polarisation du $ReLU$ en poussant les moyens d'activation plus proche de zéro.

Des sorties négatives peuvent être produites avec $\alpha(e^z - 1)$. Un exemple est illustré à la Figure 2.6 avec $\alpha = 1$. ELU est défini comme :$g(z) = \begin{cases} z \text{ si } z > 0 \\ \alpha(e^z - 1) \text{ si } z \leq 0 \end{cases}$

La dérivée de ReLU est : $g'(z) = \begin{cases} 1 \text{ si } z > 0 \\ \alpha e^z \text{ si } z \leq 0 \end{cases}$

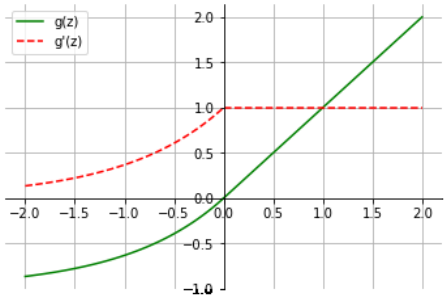

FIG. 2.6 – ELU et sa dérivée, $\alpha = 1$

Avantages :

- Comme *ReLU*, le problème de gradient de fuite est évité en raison de la dérivée constante dans la partie positive.
- La moyenne de la sortie se rapproche de zéro. En conséquence, ELU converge plus rapidement.
- Les résultats expérimentaux montrent que lorsqu'il y a plus de cinq couches dans réseau, ELU généralise mieux que *ReLU*. De plus, ELU permet un apprentissage plus rapide.

Désavantage :

- La recherche d'un A optimal par le réseau prend du temps.

2.1.4 Fonctions de perte

Dans ce mémoire, nous élaborerons différentes fonctions énergétiques en 2.2.2 que nous avons utilisées comme fonctions de perte dans nos réseaux. La logique des fonctions d'énergie est la même que celle des fonctions de perte qui sont utilisés dans les réseaux de neurones non linéaires. Par conséquent, nous discutons de différentes fonctions de perte afin de démontrer leur rôle dans le processus d'apprentissage. Une fonction de perte peut également être appelée fonction coût, risque empirique ou fonction objectif. Le coût engendré par des prédictions incorrectes est mesuré par la perte de notre réseau. Nous désignons

la perte par L et elle est calculée par $L(f(x_i, W), y_i)$, où $f(x^{(i)}; W)$ est la sortie prédite du modèle par rapport aux poids. La sortie réelle que nous voulons acquérir est $y^{(i)}$ et l'erreur est la différence entre la sortie réelle et la valeur prédite sortie du modèle. Selon l'erreur, les poids des neurones sont modifiés. La perte est réalisée grâce à des algorithmes d'optimisation, qui sont expliqués en 2.1.5 .

Pour mesurer la perte totale sur l'ensemble de données, nous calculons la perte empirique :

$$J(W) = \frac{1}{n}\left(\sum_{i=1}^{n} L(f(x_i, W), y_i)\right)$$

Pour chaque entrée x_i, la perte est calculée et additionnée. Pour trouver la perte moyenne de réseau la somme est divisée par le nombre d'échantillons d'apprentissage n. Il existe différents types de fonctions de perte ayant des objectifs différents. Ici, on discute certaines des fonctions de perte les plus courantes et quelle fonction de perte opter pour différentes scénarios.

2.1.5 Optimisation des pertes

Pour obtenir la perte la plus faible, nous devons trouver les poids W qui minimisent la fonction de perte indiquée ci-dessous :

$$W^* = \arg\min_W \frac{1}{n}\left(\sum_{i=1}^{n} L(f(x_i, W), y_i)\right)$$

$$W^* = \arg\min_W J(W)$$

$w = w_1, w_2, \cdots, w_m$ est la collection de poids à travers le réseau de neurones de toutes les couches. Dans un problème d'optimisation, nous voulons optimiser tous les poids pour minimiser la perte empirique. En 2.1.4 nous avons décrit la perte en fonction des poids du réseau avec $J(W)$.

2.1.6 Rétropropagation

Actuellement, la rétropropagation est la technique la plus populaire pour effectuer des tâches d'apprentissage supervisé telles que la reconnaissance de formes. L'algorithme peut être appliqué aux réseaux de neurones qui représentent des solutions à des problèmes non linéaires. Pour rétropropager, au moins une couche cachée est nécessaire et les recherches ont montré que les réseaux avec une seule couche cachée sont capable de fournir des approximations précises de toute fonction continue. Néanmoins, nous appliquons également une version très simplifiée de la rétropropagation dans la méthode de résolution itérative car nous devons calculer les gradients pour mettre à jour la solution x. Le but de l'algorithme est d'obtenir un réseau précis qui peut bien fonctionner sur des données en adaptant les paramètres. Par conséquent, nous devons ajuster les pondérations de manière à ce que, si elles diminuent, l'erreur soit plus élevée qu'elle ne l'augmente et vice versa. Cette méthode est effectuée sur tous les poids du réseau puis on recommence. Pour un passage à travers le réseau, les dérivées de tous les poids sont calculées à l'aide de la règle de la chaîne. Nous continuons à minimiser l'erreur en ajustant les poids jusqu'à ce que l'erreur et les poids soient stabilisés.Pour visualiser de quelle rétropropagation nous bénéficions, la figure 2.7. C'est une démonstration

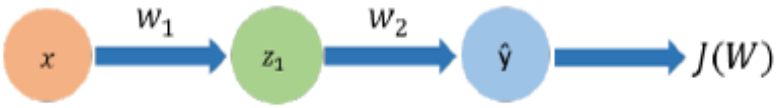

FIG. 2.7 – Rétropropagation

d'un réseau de neurones hautement simplifié qui contient un neurone d'entrée x, une couche cachée existant uniquement à partir d'un neurone z_1 et d'un neurone de sortie $\hat{y}$. Pour déterminer comment un petit changement d'un poids w_2 affecte la perte finale $J(W)$, nous calculons :

$$\frac{\partial J(W)}{\partial(w_2)} = \frac{\partial J(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w_2}$$

Nous appliquons la règle de la chaîne pour calculer le gradient selon w_2. Si nous voulons calculons le gradient selon w_1, puis nous appliquons la règle de chaîne récursivement :

$$\frac{\partial J(W)}{\partial(w_1)} = \frac{\partial J(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_1} \times \frac{\partial z_1}{\partial w_1}$$

où z_1 est la fonction d'activation de la première unité. Comme le nom rétropropagation l'indique, lorsque nous calculons le gradient par rapport à w_1, nous prenons du recul et nous incluons la dérivée de la sortie $\hat{y}$ par rapport à z_1 et la dérivée de z_1 par rapport à w_1. Pour compléter la rétropropagation, l'algorithme est répété pour chaque poids du réseau en utilisant les gradients des couches externes.

2.2 Réseaux de neurones pour résoudre des systèmes d'équations linéaires

À ce stade, nous passons de la théorie générale des réseaux de neurones non linéaires à la résolution de problèmes linéaires. La raison pour laquelle nous avons déjà discuté des réseaux de neurones est qu'ils nous permettent de représenter solutions aux problèmes (généralement : non linéaires), et nous utiliserons cette propriété maintenant pour représenter les solutions aux systèmes linéaires à la place. De plus, nous discutons de la régularisation spécifique méthodes pour les systèmes linéaires.

2.2.1 Formulation du problème

On propose de construire des réseaux de neurones pour résoudre un système linéaire de manière itérative.Le modèle d'estimation linéaire est :

$$Ax = b + r = b_{true}$$

$A = [a_{ij}] \in R^{n \times n}$ est la matrice modèle, $b \in R^n$ est un vecteur d'observations ou de mesures, $b_{true} \in R^n$ est un vecteur de valeurs vraies et enfin, $r \in R$ est un vecteur inconnu d'erreurs de mesure. Ce que nous recherchons est une estimation de $x = [x_1, x_2, \cdots, x_n] \in R^n$.

Une fonction énergétique hypothétique est construite afin de trouver un vecteur x qui minimise la fonction énergétique. Les fonctions énergétiques sont définies comme suit :

$$E(x) = \sum_{i=1}^{n} \sigma_i (r_i(x))$$

σ_i représente la fonction convexe, pour $\sigma_i(r_i) = \frac{r_i^2}{2}$

Un autre aspect que nous devons connaître pour comprendre la fonction énergétique est le vecteur résiduel r défini comme :

$$E(x) = [r_1(x), r_2(x), ..., r_m(x)]^T = Ax - b \tag{2.2}$$

Chaque membre du vecteur r en 2.2 peut être calculé avec :

$$r_i(x) = a_i^T x - b_i = \sum_{j=1}^{n} a_{ij} x_j - b_i$$

2.2.2 Fonctions d'énergies

La fonction énergétique du problème est définie comme suit :

$$E(x) = \frac{1}{2} \sum_{i=1}^{m} r_i^2 = \frac{1}{2} r^T(x) r(x) = \frac{1}{2} (Ax - b)^T (Ax - b) = \frac{1}{2} \|Ax - b\|_2^2 \tag{2.3}$$

Comme mentionné précédemment, notre objectif est de trouver un x qui minimise la fonction énergétique. Par conséquent, nous devons calculer le gradient de la fonction énergétique $\nabla E(x)$. Pour calculer le gradient, les dérivées de la fonction énergétique sont calculées par rapport à chaque élément de x comme indiqué ci-dessous :

$$\nabla E(x) = A^T (Ax - b) = \left[\frac{\partial E(x)}{\partial x_1}, \frac{\partial E(x)}{\partial x_2}, ..., \frac{\partial E(x)}{\partial x_n} \right] \tag{2.4}$$

Où $x(0) = x^0$. Après avoir calculé le gradient par rapport à la fonction d'énergie spécifiée, nous mettons à l'échelle le gradient en le multipliant par $-\mu(t)$

$$\frac{dx}{dt} = -\mu(t) \nabla E(x) \tag{2.5}$$

$\mu(t) = [\mu_{ij}(t)]$ est une matrice $n \times n$ définie positive souvent diagonale. Les entrées de la matrice dépendent du temps t.

La procédure d'apprentissage de l'ensemble du système par rapport aux moindres carrés ordinaires est :

$$\frac{dx_j}{dt} = -\sum_{p=1}^{n} \mu_{jp} \left(\sum_{p=1}^{n} a_{ip} \left(\sum_{k=1}^{n} a_{ik} x_k - b_i \right) \right) \tag{2.6}$$

Avec x_j^0, pour $j = 1, 2, ..., n$. x est calculé comme un point limite à partir de x^0, qui peut être choisi aléatoirement à l'état initial.

L'objectif lors de la construction d'une matrice $\mu(t)$ est de fournir la vitesse de convergence et la stabilité des équations différentielles. D'une part, si μ_{ij} est choisi petit, la convergence la vitesse à la solution désirée x sera plutôt lente.D'autre part, si μ_{ij} est choisi grand, il peut provoquer un comportement instable du réseau de neurones. Par conséquent, $\mu(t)$ sert ici le devoir du taux d'apprentissage dont nous avons discuté en 2.1.5.

Il y en a deux possibilités de décider des entrées μ_{ij} . Premièrement, désigner des entrées constantes pour les poids. Deuxièmement, sélectionner les entrées de manière adaptative.

La sélection adaptative peut augmenter le taux de convergence sans causer de problèmes de stabilité, par opposition au choix direct de grandes constantes.
Les valeurs absolues des résidus sont d'abord calculées. Ensuite, le maximum le long des résidus est sélectionné.

Toute l'approche avec les équations différentielles peut être effectuée en trois étapes essentielles étapes et chaque étape représente une couche dans le réseau illustré à la Figure 2.8 . L'architecture de la méthode de résolution itérative ressemble au perceptron en 2.1.2. La différence entre les modèles est que le réseau construit pour la méthode de résolution itérative stocke x, la solution que nous voulons estimer, sous forme de poids. De plus, le réseau ne traite aucune entrée contrairement aux réseaux de neurones habituels. Nous avons un A fixe et un b fixe.

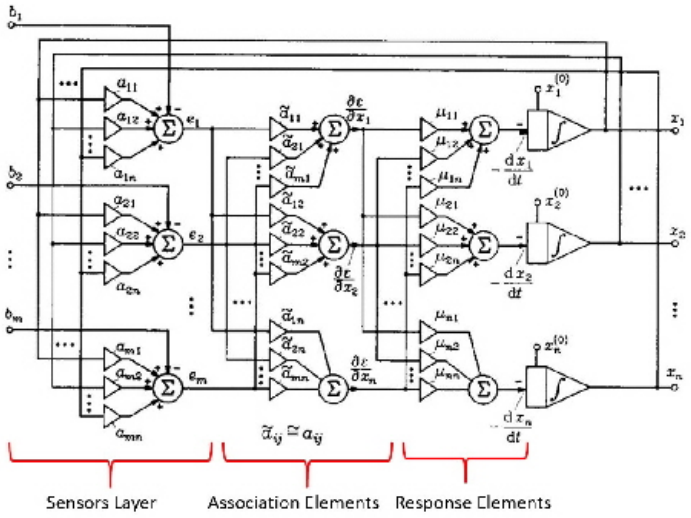

FIG. 2.8 – Architecture d'un réseau de neurones pour résoudre un système linéaire avec le critère des moindres carrés ordinaires. Adapté de [5].

1. Calcul de l'erreur ; la première étape consiste à calculer l'erreur :

$$e_i(x) = \sum_{k=1}^{n} a_{ik}x_k - b_i, i = 1, 2, .., n$$

Exceptionnellement, une non-linéarité avec g est ajoutée à la fonction énergétique du problème des moindres carrés itérativement repondéré :

$e_i(x) = g\left[\sum_{k=1}^{n} a_{ik}x_k - b_i\right], i = 1, 2, .., n$

Dans la première couche, à savoir la couche capteurs, les signaux d'erreur sont produits comme le montre la figure 2.8.

2. Calcul du gradient : La deuxième couche est constituée d'éléments d'association qui calculent les composantes du gradient de la fonction avec la contribution de la première couche, pour les problèmes des moindres carrés et des moindres carrés itérativement repondérés, les gradients sont calculés de la même manière, ce qui correspond à :

$\frac{\partial\epsilon(x)}{\partial x_p} = \sum_{i=1}^{n} a_{ip}e_i(x), p = 1, 2, .., n$ pour le problème de moindre valeur absolue, le gradient est :

$$\frac{d\epsilon(x)}{dx_p} = \sum_{p=1}^{n} a_{ip}\left(-b_i + \sum_{j=1}^{n} a_{ij}x_j\right), p = 1, ..., n \tag{2.7}$$

3. Constitution du système d'apprentissage approprié ; la troisième et dernière couche est formée d'éléments de réponse. Avec cette étape, l'ensemble du processus d'apprentissage est terminé pour une itération. La déscente est obtenue grace à la mise à échelle avec $-\mu$:

$$\frac{dx_j}{dt} = -\sum_{p=1}^{n} - \mu_{jp}\frac{\partial\epsilon(x)}{\partial x_p}, x_j(0), j = 1, 2, .., n \tag{2.8}$$

Chapitre 3

Réseau de neurones à descente de gradient

Dans ce chapitre, la mise en œuvre de réseaux de neurones capables de résoudre des systèmes linéaires est expliquée en profondeur. Le travail de ce mémoire est basé principalement sur l'article "Réseaux de neurones pour la résolution de systèmes d'équations linéaires et de problèmes connexes" 1 et l'article "Solving Linear Equations with Gradient Neural Networks" 2, où leur approche pour la construction de réseaux de neurones est discutée en détail dans la section 2.2 . Nous commençons par la réimplémentation de la méthode de résolution itérative adoptée à partir de l'article. Dans un premier temps, nous ne bénéficions d'aucun framework logiciel pour le deep learning.

3.1 Structure du réseau de neurones

Avant d'expliquer la mise en œuvre de la méthode de résolution itérative, nous donnons un aperçu général de la structure du réseau. Dans cette section, nous examinons la superposition des réseau et l'utilisation de fonctions d'activation à l'avance en tenant compte de la structure des réseaux de neurones non linéaires.

3.1.1 Couches

Contrairement aux réseaux de neurones non linéaires, notre réseau ne représente pas de cartographies non linéaires. Dans la figure 2.8 de la dernière section 2.2 , le modèle linéaire pour la méthode de résolution itérative est démontré. Selon Cichocki et Unbehauen, le modèle consiste de trois couches et dans chaque couche différents calculs pour construire le système d'apprentissage ont lieu. Ainsi, la logique de la stratification est différente d'aujourd'hui comme nous l'avons vu en 2.1. Les trois couches mentionnées correspondent à une couche linéaire à l'heure actuelle parce que dans notre implémentation, nous n'utilisons aucune autre couche cachée et nous n'avons pas non plus de couches d'entrée et de sortie séparées. Notre réseau ne traite aucune entrée mais stocke la solution x dans une seule couche linéaire en tant que poids. La raison pour laquelle ils divisent le réseau en trois couches différentes est de distinguer les différentes étapes de calcul les unes des autres.

3.1.2 Fonction d'activation

Dans notre réseau, nous n'utilisons aucune fonction d'activation comme les réseaux de neurones non linéaires habituels discuté en 2.1.3 . Néanmoins, cette fonction d'activation n'est pas appliquée au réseau directement comme les réseaux de neurones non linéaires, où à chaque couche la multiplication des poids et des entrées lui sont transmises comme décrit en 2.1.2. Cette fonction d'activation fait partie de la fonction de perte. Nous n'avons donc pas besoin d'implémenter la fonction d'activation explicitement pour le processus de formation.

3.2 Simulation par ordinateur

Afin d'étudier et de vérifier le modèle réseau de neurones à descente de gradient proposé pour résoudre les équations linéaires 1.1, nous pourrions effectuer des simulations informatiques via la formule de mise à jour des poids 2.8 . Un certain nombre de coefficients

de test A et b et de poids initial $w(0)$ sont générés aléatoirement pour les simulations. Les observations générales sont qu'il est à la fois efficace d'employer les formule de mise à jour des poids 2.8 qui obtient finalement la solution de 1.1. De plus, en utilisant la mise à jour du poids la formule 2.8 est beaucoup plus rapide et plus précise que l'utilisation des solveurs MATLAB.

Exemple 3.2.1 *Application*

Une matrice de coefficients non singulière $A \in R^{4\times4}$et le vecteur $b \in R^{4\times1}$ de 1.1 sont générés aléatoirement comme ci-dessous :

$$A = \begin{bmatrix} 0.8214 & 0.9218 & 0.9355 & 0.0579 \\ 0.4447 & 0.7382 & 0.9169 & 0.3529 \\ 0.6154 & 0.1763 & 0.4103 & 0.8132 \\ 0.7919 & 0.4057 & 0.8936 & 0.0099 \end{bmatrix}, b = \begin{bmatrix} 0.9501 \\ 0.2311 \\ 0.6068 \\ 0.4860 \end{bmatrix}$$

pour lequel nous calculons la solution en se servant du code MATLAB suivant.

```
1   A=rand(4);
2 A=rand(4);
3 b=rand(4,1);
4 Initial=rand(4,1);
5 eta=1.2/trace(A'*A);
6 w=Initial;
7 for i=1 :3000
8 e=b-A*w;
9 w=w+eta*A'*(b-A*w);
10 error(i)=0.5*(e'*e);
11  end
```

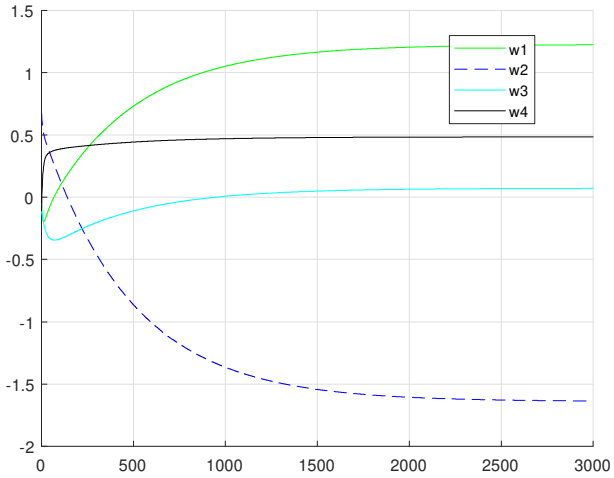

FIG. 3.1 – Réseau via une formule de mise à jour du poids [3]

Comme le montre la figure 3.1, chaque élément de poids du réseau de neuronne artificiel $x(k)$ converge. La figure 3.2 montre la convergence de l'erreur de calcul $E(x)$ vers zéro, qui est synthétisée par la formule de mise à jour du poids 2.8 du réseau de neuronne artificiel . Évidemment, après 3000 itérations, l'erreur d'entraînement résiduelle E [noté erreur (3000) dans le code MATLAB ci-dessus] pourrait converger à $4,0656 \times 10^{-22}$ près, et $x(3000)$ est calculé comme

$$w(3000) = \begin{bmatrix} w1(3000) \\ w2(3000) \\ w3(3000) \\ w4(3000) \end{bmatrix} = \begin{bmatrix} 1.5321 \\ 0.9105 \\ -1.2273 \\ 0.0086 \end{bmatrix}$$

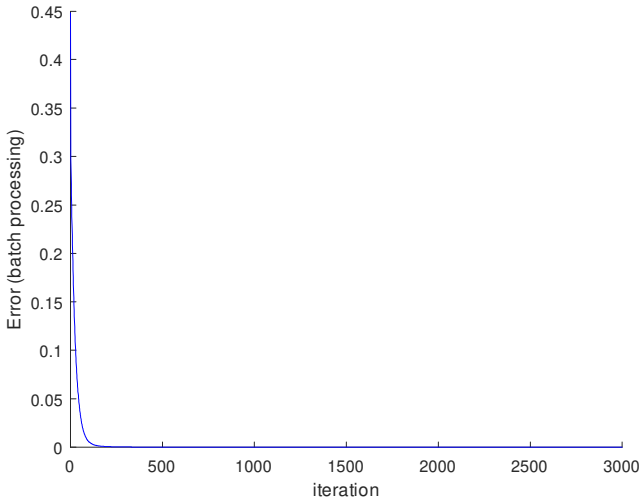

FIG. 3.2 – Erreur d'entraînement en utilisant la formule de mise à jour du poids [3]

Conclusion

Résumé

Les réseaux de neurones ont des fonctionnalités compatibles pour travailler avec des opérations matricielles. Dans ce mémoire nous avons proposé une méthode itérative pour résoudre un système linéaire. Nous avons d'abord examiné la non-linéarité réseaux de neurones en profondeur, puis nous avons adopté certaines des caractéristiques de ces réseaux de neurones dans les modèles que nous avons construits. La raison pour laquelle nous avons implémenté une méthode de résolution itérative au lieu d'utiliser des solveurs linéaires préexistants est que nous voulions obtenir un système qui peut fonctionner avec de grandes matrices. Les solveurs linéaires sont faciles à mettre en œuvre et ils renvoient des résultats plus précis pour les problèmes à petite échelle $Ax = b$. Pour montrer le fonctionnement de notre réseau, nous avons utilisé un problème à petite échelle avec une matrice $A \in R^{4\times4}$et le vecteur $b \in R^{4\times1}$ en 3.2.1 . Pour le problème démontré, nous avons obtenu une solution x identique à la solution réelle avec la méthode solve(). Par conséquent, nous avons obtenu un meilleur résultat avec un solveur linéaire que les réseaux que nous avons construits pour résoudre un système linéaire. Néanmoins, dans la section 3.2.1, nous avons montré un problème à grande échelle, où nous avons utilisé un $A \in R^{10000\times10000}$ et un $b \in R^{10000}$, et nous avons obtenu une estimation de x que nous ne pouvions pas obtenir avec des solveurs linéaires.

Pour conclure, nous avons implémenté une méthode pour résoudre itérativement un sys-

tème linéaire qui présente des avantages sur les solveurs linéaires.

Bibliographie

[1] A. Cichocki and R. Unbehauen. Neural networks for solving systems of linear equations and related problems. IEEE Transactions and Circuits and Systems : Fundemental Theory and Applications, 1992.

[2] H. Khelil, N. Khelil and L. Djerou, Solving Linear Equations with Gradient Neural Networks, AMS, Biskra, May 14-15, 2023.

[3] I. Kidil, Neural Networks Solving Linear Systems, TECHNISCHE UNIVERSITAT MUNCHEN, 2021

[4] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong, Mathematics for Machine Learning 2020.

[5] N. Khelil, L. Djerou and H. Khelil, Analyse numérique II, Éditions universitaires européennes , 2021.

[6] O. Sayah, Une approche pratique des réseaux de neurones artificiels, Mémoire Master, Université de Biskra, 2022.

[7] Phil Kim, MATLAB Deep Learning : With Machine Learning, Neural Networks and Artificial Intelligence, 2017

[8] S. Haykin Neural Networks : A Comprehensive Foundation. Macmillan Publishing, New York, 1994.

ملخص

يعد حل الأنظمة الخطية جزءًا أساسيًا من الهندسة وعلوم الكمبيوتر كما تستخدم في العديد من المجالات مثل المحاكاة العددية ومعالجة الصور والإشارات وديناميكيات السوائل. يمكن تمثيل المعادلات الخطية $Ax = b$، وهو نظام يتكون من المصفوفة A، والحل x، والبيانات b. في هذا العمل، نستكشف الحساب لحل الأنظمة الخطية باستخدام شبكة عصبية. نبني شبكات عصبية تمثل ناقل الحل x. تم إصلاح A و b والهدف من الشبكة هو إيجاد تقدير لـ x يقلل من الخطأ r في $r = Ax - b$. كميزة، يمكن للشبكات العصبية تقدير الحلول للمشاكل واسعة النطاق.

Abstract

Solving linear systems is a fundamental part of engineering and computer science as it is used in many fields such as numerical simulations, image and signal processing and fluid dynamics. Linear equations can be collectively represented as Ax=b, a system consisting of matrix A, solution x, and data b. In this work, we explore computation to solve linear systems, using a neural network. We build neural networks that represent the solution vector x. A and b are fixed and the goal of the network is to find an estimate of x that minimizes the error r in r=Ax-b. As an advantage, neural networks can estimate solutions for large-scale problems.

Résumé

La résolution de systèmes linéaires est un élément fondamental de l'ingénierie et de la science informatique car elle est utilisée dans de nombreux domaines tels que les simulations numériques, l'image et traitement du signal et dynamique des fluides. Les équations linéaires peuvent être représentées collectivement comme Ax=b, un système composé d'une matrice A, d'une solution x et de données b. Dans ce travail, nous explorons le calcul pour résoudre des systèmes linéaires, à l'aide d'un réseau de neurones. Nous construisons des réseaux de neurones qui représentent le vecteur solution x. A et b sont fixes et le but du réseau est de trouver une estimation de x qui minimise l'erreur r dans r=Ax-b. Comme avantage, les réseaux de neurones peuvent estimer des solutions pour des problèmes à grande échelle.