

Mémoire de fin d'études

Travaux d'Études et de Recherche

PRÉSENT PAR MING WEI ANG

DIRECTEUR DE RECHERCHE : PIERRE AILLIOT

UNIVERSITE DE BRETAGNE OCCIDENTALE – LICENCE MIASHS

Année universitaire 2022 - 2023

ÉTUDE D'UNE MÉTHODE DE MACHINE LEARNING, APPLICATION AVEC PYTHON

RÉSEAUX DES NEURONES ARTIFICIELLE

Présenté par Ming Wei ANG

Numéro d'étudiant : 22106585

Sous la direction Pierre AILLIOT

Mémoire présenté le 10/05/2023, devant un jury composé

Pierre AILLIOT

Catherine RAINER

Mémoire de L3 MIASHS Parcours math - éco

Table of Contents

1 L'INTRODUCTION AUX RÉSEAUX DES NEURONES.....	1
1.1 Introduction.....	1
2 RÉSEAUX DES NEURONES ARTIFICIELS.....	3
2.1 Neurone Formel.....	3
2.2 Perceptron.....	3
2.3 Fonction de Perte.....	5
2.3.1 Les Différentes Types de Fonctions de Perte.....	7
2.3.1.1 L'Erreur Quadratique Moyenne.....	7
2.3.1.2 L'Entropie Croisée Binaire.....	8
2.3.1.3 La Perte de la Machine à Vecteurs de Support Multi-classe (La Perte Hinge).....	9
2.4 L'Algorithme de Descente de Gradient.....	11
2.4.1 Les 3 Types de Descente de Gradient.....	12
2.4.1.1 La Descente de Gradient par Lots.....	12
2.4.1.2 La Descente de Gradient Stochastique.....	14
2.4.1.3 La descente de Gradient en Mini-Lot.....	16
2.5 L'Algorithme du Gradient dans Un Perceptron.....	17
2.6 La Fonction d'Activation (La Fonction de Transfert).....	20
2.7 Perceptron Multicouche.....	22
2.8 Rétropropagation.....	25
3 LES APPLICATIONS SIMPLE SANS BIBLIOTHEQUE DE DEEP LEARNING PAR MILO SPENCER-HARPER.....	30
3.1 Première Application Avec Un Perceptron Simple.....	30
3.1.1 Procédure d'Entraînement.....	30
3.1.2 L'Équation Pour Estimer la Sortie du Neurone.....	31
3.1.3 La Formule d'Ajustement des Poids.....	31
3.1.4 Résultat.....	32
3.2 Deuxième Application Avec Un Perceptron Multicouche.....	32
3.2.1 Résultat.....	33
4 CONCLUSION.....	34
5 BIBLIOGRAPHIE.....	35
6 ANNEXE.....	36

1 L'INTRODUCTION AUX RÉSEAUX DES NEURONES

1.1 Introduction

Les réseaux de neurones artificiels sont souvent mentionnés dans la même phrase que l'intelligence artificielle. Cependant, le terme "intelligence artificielle" est très large et couvre de nombreux domaines différents tels que l'apprentissage profond et l'apprentissage automatique. Beaucoup de gens ont du mal à les distinguer ou à comprendre leurs différences. Alors, comment pouvons-nous clarifier ces idées ? Qu'est-ce que des réseau de neurones ? Quelle fonction joue-t-ils dans le contexte de l'intelligence artificielle ?

Le désir de construire des machines intelligentes existe depuis la Grèce antique. Avant qu'Ada Lovelace ne construise le premier ordinateur en 1842, le développement des ordinateurs programmables a amené les gens à se demander si les machines pourraient un jour être intelligentes. L'intelligence artificielle prospère et se développe. C'est un domaine dynamique et illimité avec de multiples applications dans le monde réel et des projets de recherche en cours. Dans un avenir proche, les logiciels intelligents seront intégrés de manière transparente dans la société humaine, automatisant les tâches répétitives, comprenant avec précision les sons et les images, prenant des décisions plus précises, aidant les médecins à diagnostiquer les services de santé et alimentant la recherche scientifique critique. Cette prise de conscience a été déclenchée par le développement de ChatGPT.

En fait, les domaines de l'intelligence artificielle, de l'apprentissage en profondeur et même des réseaux de neurones artificiels dérivent tous de l'algèbre linéaire et de la théorie des probabilités. Ils sont les rouages de ce domaine, étroitement liés et formant une chaîne imbriquée. Voir la figure 1.1 pour mieux les distinguer. Ce rapport se concentre sur la discussion des réseaux de neurones artificiels, mais en raison de la grande pertinence du domaine, les parties autres que les réseaux de neurones artificiels peuvent être inclus sans explication.

Les réseaux de neurones artificiels (RNA) sont une technique d'apprentissage automatique populaire basée sur la régression logistique qui simule les mécanismes d'apprentissage biologique. Dans cette comparaison, les neurones résident dans le système nerveux humain. Les neurones sont connectés les uns aux autres à l'aide d'axones et de dendrites. La figure 1.2 (à gauche) montre ces liens. Les réseaux de neurones artificiels (RNA), qui impliquent la modélisation informatique de neurones biologiques appelés « neurones artificiels », imitent ce mécanisme biologique. Les poids ont le même

objectif que les forces de connexion synaptiques dans les entités biologiques et sont utilisés pour connecter ces unités de compte. Les RNAs sont généralement utilisés dans des domaines liés au traitement de l'information, ainsi que pour des tâches techniques telles que la reconnaissance de formes, la prédiction et la compression de données.

Dans les RNAs, chaque entrée de neurone est mise à l'échelle par un poids qui représente le flux d'informations, et l'activation de chaque neurone est déterminée par une formule mathématique. Cette structure est illustrée sur le côté droit de la figure 1.2. Un RNA utilise des poids comme paramètres intermédiaires pour générer des fonctions d'entrée et transmettre des valeurs spécifiques des neurones d'entrée aux neurones de sortie. L'apprentissage est réalisé en ajustant les poids entre les neurones connectés. Additionnez les entrées de tous les neurones d'un réseau basé sur une autre partie du réseau pour déterminer la sortie du neurone artificiel. La sortie d'un neurone d'entrée est générée en divisant l'entrée unique de chaque neurone d'entrée par son poids.

Tout comme les organismes vivants réels ont besoin de stimuli externes pour apprendre, les réseaux de neurones artificiels nécessitent des données d'entraînement constituées de paires d'entrée/sortie contenant les informations souhaitées. Par exemple, les représentations des largeurs de pétales et de sépales (entrée) et les étiquettes annotées de sortie (telles que les espèces d'iris) peuvent être incluses dans les données d'apprentissage. Un réseau neuronal est alimenté avec des données d'apprentissage et prédit des étiquettes de sortie sur la base de représentations d'entrée. Sur la base des correspondances entre les sorties prédites (telles que les probabilités d'iris versicolore) et les étiquettes de sortie annotées dans les données d'apprentissage, les pondérations du réseau neuronal reçoivent des informations sur leur précision. Les erreurs de jugement du réseau neuronal sont considérées comme des commentaires négatifs de la même manière que les organismes modifient la force de leurs synapses. Les poids entre les neurones sont également modifiés en fonction de l'erreur de prédiction. Les pondérations sont ajustées pour améliorer les performances de calcul et améliorer la précision des prévisions futures. Par conséquent, à mesure que le poids augmente, l'entrée est renforcée. Nous pouvons modifier les poids du neurone artificiel afin que certaines entrées produisent certaines sorties. Pour les grands réseaux de neurones avec de nombreux neurones, il est difficile de déterminer manuellement tous les poids corrects. Heureusement, il existe un moyen de modifier les poids d'un réseau de neurones artificiels pour obtenir le résultat souhaité. L'apprentissage ou l'entraînement est le processus de changement de poids. Le processus de formation vise à modifier les poids jusqu'à ce que l'erreur soit minimisée. En fin de compte, après les réseaux de neurones sont formés sur des grandes données d'iris, il sera capable de reconnaître avec précision des iris jamais vus auparavant. La

généralisation du modèle est le processus d'apprentissage à partir d'une petite collection de paires d'entrées/sorties et d'application de ces connaissances à d'autres cas. L'objectif principal est la capacité des modèles d'apprentissage automatique à généraliser l'apprentissage à partir de données d'apprentissage connues à de nouveaux cas.

2 RÉSEAUX DES NEURONES ARTIFICIELS

2.1 Neurone Formel

La structure, la complexité, le type de neurones utilisés (les fonctions de transition ou d'activation) , et le résultat désiré sont utilisés pour classer les principaux types de réseaux neuronaux, qui sont composés de neurones formels connectés dans un graphe. En fonction de leur conception architecturale, ces réseaux peuvent être classés en réseaux à une seule couche ou à plusieurs couches. Les réseaux à une seule couche effectuent une cartographie directe des entrées aux sorties en utilisant des fonctions linéaires. Ces systèmes sont également appelés perceptrons. En revanche, dans les réseaux à plusieurs couches, les neurones sont regroupés en couches avec des couches cachées séparant les couches d'entrée des couches de sortie. Cette architecture est également connue sous le nom de réseau de neurones à propagation avant (réseau feedforward).

2.2 Perceptron

La forme la plus fondamentale de réseau de neurones est le perceptron. Il ne contient qu'une seule couche d'entrée et un nœud de sortie. De manière similaire à un neurone biologique, un perceptron reçoit divers stimuli via des poids. Il analyse ces données et en déduit une conclusion. La figure 1.3 représente la construction fondamentale du perceptron.

Considérons le cas où chaque instance d'entraînement est de la forme (X, y) , avec chaque $X = [x_1, \dots, x_n]$ contenant n variables de caractéristiques et $y \in \{-1, 1\}$ contenant la valeur observée de la variable de classe binaire. Le terme "valeur observée" fait référence au fait qu'elle nous est fournie dans le cadre des données d'entraînement, et notre objectif est de prévoir la variable de classe dans des circonstances où elle n'est pas observée. Dans une application de détection de fraude par carte de crédit, par exemple, les caractéristiques peuvent représenter différents aspects d'un groupe de transactions par carte de crédit (comme le montant et la fréquence des transactions), et la variable de classe pourrait refléter si cet ensemble de transactions est frauduleux ou non. De toute évidence, dans ce type

d'application, il y aurait des exemples passés où la variable de classe est observée, ainsi que d'autres cas (présents) où la variable de classe n'a pas encore été observée mais doit être anticipée.

La couche d'entrée possède n nœuds qui envoient les n caractéristiques $X = [x_1, \dots, x_n]$ avec des bords pondérés $W = [w_{1j}, \dots, w_{nj}]$ à un nœud de sortie. Chaque poids a une valeur désignée w_{ij} et transfère des informations/stimuli de la source de neurone i , qui est désigné x_i . Le poids connectant les neurones i et j module ce stimulus (sa valeur), qui est lié aux données fournies par le neurone source i . C'est à dire $w_{ij} \cdot x_i$. En conséquence, le neurone j reçoit autant de stimuli qu'il y a d'entrées à ajouter. Une notation mathématique plus précise serait la suivante si nous connaissons le nombre de neurones sources connectés aux neurones j, n : Le nœud de sortie calcule $W \cdot X = \sum_{i=1}^n w_{ij} \cdot x_i$. La variable dépendante de X est ensuite prédite en utilisant la fonction d'activation h (section 2.6) de cette valeur réelle. La prévision \hat{y} est donc calculée comme $\hat{y} = h(W \cdot X) = h\left(\sum_{i=1}^n w_{ij} \cdot x_i\right)$.

La fonction d'activation convertit un nombre réel en +1 ou 1, ce qui est une catégorisation pour la classification binaire. La variable \hat{y} indique qu'il s'agit d'une valeur prévue plutôt qu'une valeur observée. Par conséquent, l'erreur de prédiction est $E(X) = y - \hat{y}$, qui est un nombre déterminé à partir de l'ensemble $\{-2, 0, 2\}$. Les poids dans le réseau neuronal doivent être ajustés dans la direction (inverse) du gradient d'erreur lorsque la valeur d'erreur $E(X)$ n'est pas nulle. Le biais est une composante invariante de la prédiction qui existe dans de nombreux contextes. Considérons un scénario où les variables caractéristiques sont centrées réduites mais la prédiction de classe binaire de la plage de $\{-1, 1\}$ a une variable centrée réduite qui n'est pas de 0. Cela est plus susceptible de se produire lorsqu'il y a un déséquilibre extrême dans la distribution de la classe binaire. En utilisant un neurone de biais, le biais peut être représenté comme le poids d'une arête. Pour cela, un neurone qui envoie toujours la valeur 1 au nœud de sortie est ajouté. La variable de biais est fournie par le poids de l'arête qui relie le neurone de biais au nœud de sortie. La figure 1.4 représente un neurone de biais à titre d'exemple. Cette partie invariante de la prédiction nécessite l'inclusion d'une variable de biais supplémentaire w_0 , c'est à dire $\hat{y} = h(W \cdot X + w_0) = h\left(w_0 + \sum_{i=1}^n w_{ij} \cdot x_i\right)$. Comme les neurones de biais peuvent intégrer des biais, les détails des algorithmes d'entraînement restent les mêmes en traitant simplement les neurones de biais comme n'importe quel autre neurone avec une valeur d'activation

constante de 1. Par conséquent, l'équation $\hat{y} = h(W \cdot X) = h(\sum_{i=1}^n w_i \cdot x_i)$, qui n'incorpore pas explicitement les biais, peut toujours être utilisée.

2.3 Fonction de Perte

Afin de réduire le nombre de mauvaises classifications, Rosenblatt a présenté l'algorithme du perceptron, et il y avait des preuves de convergence qui offraient des garanties de précision de l'algorithme dans des contextes plus simples. Dans un ensemble de données D avec des paires de caractéristiques et étiquettes, nous pouvons ainsi exprimer l'objectif de l'algorithme du perceptron par rapport à toutes les instances d'entraînement comme suit :

$$\underset{W}{\text{Min}} \ L(W) = \frac{1}{N} \sum_{i=1}^N L_i(\hat{y}_i, y_i) + \lambda R(W), \quad \forall (X_i, y_i) \in D$$

Dans la plupart des cas, la fonction de perte L est composée d'une fonction d'entrée, d'un ensemble de paramètres (appelés "poids") et de la vraie label. L_i est appelé la perte sur la classe i ème dans la famille multi-classe. R désigne une fonction de régularisation. Cette fonction peut être utilisée pour pénaliser la complexité des W . Par conséquent, le modèle est contraint de choisir des modèles plus simples que des modèles compliqués. Un autre paramètre optimisé par cette approche est λ , le coefficient R.

Cette forme de fonction d'objectif de minimisation est également connue sous le nom de fonction de perte. La fonction de perte est un composant clé de l'apprentissage en profondeur et de l'apprentissage automatique, et elle est utilisée dans pratiquement toutes les techniques d'apprentissage de réseau neuronal. Une fonction de perte est une fonction utilisée en optimisation mathématique et en théorie de la décision qui convertit un événement ou les valeurs de un ou plusieurs variables en un nombre réel qui reflète intuitivement un certain "coût" associé à l'événement.

La fonction de perte, qui compare les valeurs cibles et projetées de sortie et est généralement utilisée pour la régression ou la classification, est une façon d'évaluer la qualité de la simulation d'un modèle de réseau de neurones sur un ensemble de données. C'est une équation qui dépend de la configuration de l'algorithme d'apprentissage automatique. Nous cherchons à réduire cette différence de sortie entre la valeur prédictive et la cible pendant l'entraînement. Si la valeur de la fonction de perte est faible, le modèle est bon; sinon, nous devons ajuster les paramètres du modèle pour réduire la perte.

En fonction de la tâche que nous voulons que le réseau effectue, le vecteur de prédiction peut représenter différentes choses. Le vecteur de sortie \hat{y} est constitué de nombres continus pour les tâches de régression, qui sont essentiellement des prévisions de variables continues (comme le prix des actions, la demande anticipée de biens, etc.).

D'un autre côté, le vecteur de sortie \hat{y} peut représenter des notes de probabilité entre 0 et 1 pour des tâches de classification telles que la segmentation de clients ou la catégorisation d'images.

Le vrai étiquette ou le "ground truth" est la valeur cible que nous voulons que le réseau neuronal prédise, et il apparaît généralement sous la forme de y . Une valeur prédite \hat{y} qui est plus proche de la valeur cible y indique que le réseau neuronal fonctionne plus efficacement. Nous pouvons calculer la différence (ou l'erreur) entre la valeur \hat{y} et y en développant une fonction de perte dont la valeur dépend de la différence.

La fonction de perte est une fonction des poids du réseau neuronal car le vecteur de prédiction \hat{y} dépend d'eux. Étant donné que la fonction de perte dépend des poids, nous devons identifier un ensemble spécifique de poids pour lesquels la valeur de la fonction de perte est aussi faible que possible. Nous appliquons cela mathématiquement à l'aide d'une technique appelée descente de gradient (la section 2.4).

La différence entre \hat{y} et y détermine la valeur de cette fonction de perte. Une plus grande différence conduit à une valeur de fonction de perte plus élevée, tandis qu'une plus petite différence conduit à une valeur de fonction de perte plus faible. À mesure que l'écart entre la valeur prédite \hat{y} et la valeur cible y diminue, la minimisation de la fonction de perte incite directement le réseau neuronal à prédire des valeurs de manière plus précise.

En fait, le seul objectif du réseau neuronal est de minimiser la fonction de perte. Cela est dû au fait que la réduction de la fonction de perte conduit automatiquement le modèle de réseau neuronal à produire de meilleures prédictions, indépendamment des détails de la tâche à accomplir. Sans être explicitement codé avec une règle spécifique à la tâche, un réseau neuronal peut résoudre des problèmes car la réduction de la fonction de perte est un objectif qui peut être atteint dans toutes les situations et qui est indépendant de la tâche ou des facteurs externes.

2.3.1 Les Différentes Types de Fonctions de Perte

Les fonctions de perte fournissent plus qu'une simple illustration statique de la performance de votre modèle ; elles constituent également la base de la correspondance initiale de vos algorithmes avec les données. La plupart des algorithmes d'apprentissage automatique utilisent une fonction de perte de quelque sorte au cours de la phase d'optimisation, qui consiste à choisir les paramètres optimaux (poids) pour vos données.

Dans une illustration simple, prenons la régression linéaire. La ligne de meilleur ajustement dans la régression des "moindres carrés" classique est identifiée par l'erreur quadratique moyenne (EQM). Le EQM est calculé pour chaque ensemble de poids que le modèle tente sur l'ensemble des échantillons d'entrée. En utilisant une méthode d'optimisation telle que la descente de gradient, le modèle réduit ensuite les fonctions EQM au minimum absolu.

En deep learning, il existe plusieurs optimiseurs distincts, tout comme il existe de nombreuses variétés de fonctions de perte pour différentes situations. La fonction de perte et l'optimiseur travaillent ensemble pour fournir à l'algorithme l'ajustement le plus optimal possible à vos données. Par exemple, nous pouvons utiliser l'erreur quadratique moyenne, l'erreur absolue moyenne ou la perte de Huber dans les problèmes de régression ; nous avons entropie croisée binaire, entropie croisée catégorique, perte SVM multi-classe, etc. dans les problèmes de classification et dans les modèles de langage, il y a fonction de coût par triplet pour cartographier le "coût" dans les problèmes d'intégration de mots.

Dans ce rapport, nous présentons quelques-unes des principales fonctions de perte :

2.3.1.1 L'Erreur Quadratique Moyenne

L'erreur quadratique moyenne, l'une des fonctions de perte les plus largement utilisées, est utilisée pour les problèmes de régression. Dans l'ensemble du jeu de données, EQM calcule la moyenne des différences au carré entre les sorties cibles y et les sorties prédictes \hat{y} , comme suit :

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

Cette fonction est particulièrement bien adaptée pour calculer la perte en raison de certaines de ses caractéristiques. Comme la différence est au carré, il n'a pas d'importance si la valeur projetée est supérieure ou inférieure à la valeur désirée; néanmoins, les valeurs ayant une imprécision importante

sont punies. Le fait que EQM soit une fonction convexe avec un minimum global distinct facilite l'utilisation de l'optimisation de descente de gradient pour déterminer les valeurs de poids.

Nous avons un réseau neuronal, par exemple, qui utilise des informations sur les maisons (comme les pièces de l'appartement, la superficie et l'environnement) pour prédire les prix des maisons. Vous pouvez utiliser le EQM dans ce scénario. Cependant, un inconvénient de cette fonction de perte est qu'elle est particulièrement sensible aux valeurs aberrantes ; si une valeur prédictive est nettement plus grande ou moins grande que sa valeur cible, cela peut considérablement augmenter la perte.

2.3.1.2 L'Entropie Croisée Binaire

Cette fonction de perte est utilisée dans les modèles de classification binaire, où le modèle doit catégoriser une entrée dans l'une des deux catégories prédéfinies, telles que s'il pleuvra ou non.

Lorsqu'une entrée donnée est présentée à un réseau de neurones de classification, il produit un vecteur de probabilités qui représente la probabilité que l'entrée appartienne à chacune des catégories prédéfinies. Chacune des probabilités prédictives est comparée à la sortie de classe réelle en utilisant l'entropie croisée binaire. Les probabilités sont ensuite attribuées un score qui les pénalise en fonction de leur éloignement de la valeur cible (figure 1.5). En fonction de la proximité ou de l'éloignement de la valeur par rapport à la valeur cible, la catégorie ayant la probabilité la plus élevée sera choisie

comme résultat final, l'équation comme suit : $ECB = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(\hat{y}_i) + (1-y_i) \cdot \log(1-\hat{y}_i)$

Il n'y a que deux valeurs réelles de y qui peuvent être considérées comme valides dans une classification binaire : 0 ou 1. Par conséquent, pour calculer correctement la différence entre les valeurs cibles et prédictives, il faut comparer la valeur cible (0 ou 1) aux probabilités que l'entrée entre dans cette catégorie (\hat{y}_i = probabilité que la catégorie soit 1 ; $1-\hat{y}_i$ = probabilité que la catégorie soit 0).

Par exemple, le réseau neuronal peut déterminer s'il va pleuvoir ou non sur la base d'informations relatives à l'atmosphère (telles que la température, la couverture nuageuse et l'humidité). S'il pleut, la valeur cible fournie au réseau est de 1, sinon elle est de 0. Le réseau considère qu'il pleut si le résultat est supérieur à 0,5 et qu'il ne pleut pas si le résultat est inférieur à 0,5. La possibilité de pluie augmente à mesure que la valeur du score de probabilité augmente.

2.3.1.3 La Perte de la Machine à Vecteurs de Support Multi-classe (La Perte Hinge)

Pour identifier une fonction de perte lisse, dont le gradient est la mise à jour du perceptron, dans une tâche de classification binaire, le nombre d'erreurs de classification peut être exprimé sous la forme d'une fonction de perte 0/1 pour le point de données d'entraînement (X_i, y_i) comme suit:

$$L_i^{(0/1)} = \frac{1}{2} (y_i - h(W \cdot X_i))^2 = 1 - y_i \cdot h(W \cdot X_i)$$

Le fait de définir y_i^2 et signe $(W \cdot X_i)^2$ à 1 simplifiera l'objectif de fonction à droite car ils sont tous les deux calculés en élevant au carré des valeurs prises dans la plage de $\{-1,1\}$. Cependant, cette fonction d'objectif (fonction de perte 0/1 $L_i^{(0/1)}$) n'est pas lisse, elle ne peut donc pas être différentielle et son gradient est mal défini aux endroits où elle saute, surtout lorsqu'elle est accumulée sur une série de points. Le terme $-y_i \cdot h(W \cdot X_i)$ domine la fonction de perte 0/1, et la fonction de signe génère la plupart des problèmes liés à la non-différentiabilité. Comme l'optimisation basée sur le gradient définit les réseaux neuronaux, nous devons construire une fonction d'objectif lisse pour conduire les mises à jour du perceptron. Pour traiter toutes les prédictions correctes de manière uniforme et sans perte, la fonction de signe de la fonction de perte 0/1 précédente est abandonnée et les valeurs négatives sont assignées à 0: $L_i = \max\{-y_i(W \cdot X_i), 0\}$

La mise à jour du perceptron résulte du gradient de cette fonction objective lissée, et la mise à jour du perceptron est essentiellement $W \leftarrow W - \alpha \nabla_W L_i$ avec le paramètre α déterminant le taux d'apprentissage du réseau neuronal. Une fonction de perte de substitution lissée est la fonction de perte modifiée qui permet le calcul du gradient d'une fonction non différentiable. Une fonction de perte de substitution lissée est utilisée dans presque toutes les techniques d'apprentissage basées sur l'optimisation continue (comme les réseaux de neurones) qui fournissent des sorties discrètes, telles que des étiquettes de classe.

Cependant, le comportement pour les données qui ne peuvent pas être séparées linéairement en classes est très arbitraire et la solution résultante n'est parfois même pas une approximation décente. L'objectif de la séparation des classes peut être compromis par des mises à jour qui augmentent considérablement la fréquence des erreurs de classification tout en réduisant la perte en raison de la sensibilité directe de la perte à la magnitude du vecteur de poids. C'est un exemple de la façon dont les

fonctions de perte substitut peuvent parfois ne pas atteindre pleinement leurs objectifs initialement fixés. Cela rend la méthode instable et susceptible de produire des solutions de qualité très différente.

Par conséquent, plusieurs modifications de l'algorithme d'apprentissage pour les données indivisibles ont été proposées. L'une d'entre elles, très efficace, introduit le concept de marge dans la fonction de perte pour produire un algorithme équivalent à la machine à vecteurs de support linéaire. Pour cette raison, la machine à vecteurs de support linéaire est également appelée le perceptron de stabilité optimale. La perte MVS ou la perte hinge (voir figure 1.6) est une approximation lisse de la fonction de perte 0/1, et est définie comme suit: $L_i^{MVS}(y_i, (W \cdot X_i)) = \max\{0, 1 - y_i \cdot (W \cdot X_i)\}$

La fonction de perte hinge est une borne supérieure lisse sur la fonction de perte 0/1 et est couramment utilisée dans les algorithmes de machines à vecteurs de support (MVS). La mise à jour du perceptron peut être déduite du gradient de la fonction de perte hinge par rapport aux poids W , ce qui donne:

$$\nabla WL(y_i, (W \cdot X_i)) = -y_i \cdot X_i, \text{ si et seulement si } y_i \cdot (W \cdot X_i) < 1, \text{ et } \nabla WL(y_i, (W \cdot X_i)) = 0, \text{ sinon.}$$

Ce gradient peut être utilisé pour mettre à jour les poids W dans un algorithme de descente de gradient, ce qui est équivalent à la règle de mise à jour du perceptron.

Le critère du perceptron est une version décalée de la perte hinge utilisée dans les machines à vecteurs de support, et les mises à jour du perceptron peuvent être réécrites en utilisant la perte hinge comme suit :

$$W \leftarrow W + \alpha \sum_{(x,y) \in \hat{S}} y_i X_i \text{ et } W \leftarrow W, \text{ sinon.}$$

Comme vous pouvez le voir, les mises à jour du perceptron sont presque identiques aux mises à jour utilisées dans l'algorithme MVS, qui est basé sur la perte hinge. La seule différence est le signe de la mise à jour, qui est opposé dans les deux cas.

Ici, \hat{S} est décrit comme l'ensemble de tous les points d'entraînement mal classés $X \in S$ qui répondent tous à la condition que $y_i \cdot (W \cdot X_i) < 0$. Le perceptron utilise l'erreur $E(X_i)$ pour la mise à jour, tandis que y_i est utilisé dans la mise à jour ci-dessus, ce qui rend cette mise à jour considérablement

différente du perceptron. Le fait que la valeur d'erreur (entière) $E(X_i) = (y - h(W \cdot X_i)) \in \{-2, 2\}$ ne peut jamais être égale à 0 pour les points mal classés dans \hat{S} est un problème crucial. Par conséquent, pour les points mal classés, nous utilisons $E(X_i) = 2y_i$, et dans les mises à jour, $E(X_i)$ peut être changé en y_i après avoir enlevé le facteur 2 dans le taux d'apprentissage. La seule différence entre cette mise à jour et celle utilisée par la technique MVS primaire est que cette mise à jour est limitée à être appliquée aux points mal classés dans le perceptron, tandis que le MVS met également à jour les points marginalement précis près de la frontière de décision. L'une des principales différences entre les deux méthodes est que le MVS est sous la condition $y_i \cdot (W \cdot X_i) < 1$ [au lieu de la condition $y_i \cdot (W \cdot X_i) < 0$] pour définir \hat{S} .

2.4 L'Algorithme de Descente de Gradient

L'algorithme d'optimisation le plus couramment utilisé est la descente de gradient. Il est souvent utilisé pour entraîner des réseaux de neurones et des modèles d'apprentissage automatique. L'objectif de la plupart des algorithmes d'optimisation est de minimiser une fonction et il suffit de minimiser l'inverse d'une fonction pour la maximiser. Pour mieux comprendre comment l'expression "Descente de Gradient" s'applique aux algorithmes d'apprentissage automatique, examinons-la en détail.

Un gradient est une mesure qui exprime à quel point une ligne ou une courbe est raide. Il décrit la direction de la montée ou de la descente d'une ligne mathématiquement.

Une descente est un mouvement dans une direction descendante. Par conséquent, en utilisant les deux significations simples de ces mots, la méthode de descente de gradient mesure le mouvement vers le bas.

Ainsi, le but est de minimiser une fonction réelle différentiable définie sur un espace de Hilbert ou, de manière plus générale, sur un espace euclidien (par exemple, \mathbb{R}^n , l'espace des n -uplets de nombres réels avec un produit scalaire). Comme le processus est itératif, les avancements se font un après l'autre. La fonction est déplacée loin de l'emplacement actuel dans la direction opposée au gradient, ce qui la fait diminuer. La méthode numérique connue sous le nom de recherche linéaire contrôle le déplacement dans cette direction. Cette explication démontre que la méthode est un algorithme de direction descendante.

Lorsque nous entraînons un algorithme d'apprentissage automatique, nous essayons de trouver les poids et les biais dans le réseau qui nous aideront à résoudre le problème en question. Par exemple, si nous voulons construire une machine capable de résoudre un problème de catégorisation, elle devrait être capable de dire si une image est celle d'un chien ou d'un chat lorsque nous lui fournissons l'exemple. Nous utilisons des échantillons de données d'entraînement d'images correctement étiquetées de chats et de chiens pour entraîner l'algorithme afin de créer le modèle.

Bien que l'exemple précédent ait été une classification, le problème peut être celui de la localisation ou de la détection. Cependant, l'efficacité d'un réseau de neurones dans la résolution d'un problème est représentée par une fonction de perte, qui évalue l'erreur d'un modèle. Les poids et les biais utilisés pour le modèle final sont influencés par les dérivées partielles de la fonction de perte.

Ces modèles s'améliorent avec des données d'entraînement, et la fonction de perte dans la descente de gradient sert spécifiquement d'instrument de mesure, évaluant la précision du modèle à chaque itération de changements de paramètres. Le modèle continuera à modifier ses paramètres pour fournir l'erreur minimale jusqu'à ce que la fonction soit proche de zéro ou égale à zéro. Une fois que les modèles d'apprentissage automatique ont été améliorés pour leur précision, ils peuvent servir d'outils utiles pour les applications d'intelligence artificielle (IA) et d'informatique. En conclusion, la descente de gradient facilite la recherche de valeurs de paramètres qui réduisent la fonction de perte à un minimum local ou à la meilleure précision possible.

2.4.1 Les 3 Types de Descente de Gradient

2.4.1.1 La Descente de Gradient par Lots

Pour l'ensemble des données d'entraînement, la descente de gradient par lots calcule le gradient de la fonction de perte par rapport aux paramètres w : $w_{i+1} = w_i - \alpha \cdot \frac{\partial}{\partial w_i} L_i$

Après avoir évalué tous les exemples d'entraînement, la descente de gradient par lots met à jour le modèle en ajoutant les erreurs pour chaque point dans un ensemble d'entraînement. Un cycle ou une époque d'entraînement est un terme utilisé pour décrire l'ensemble du processus.

Même si la descente de gradient par lots accélère le calcul, elle prend encore beaucoup de temps pour traiter de vastes ensembles de données d'entraînement. La descente de gradient par lots peut être

extrêmement lente car nous devons calculer les gradients pour l'ensemble du jeu de données pour effectuer une seule mise à jour. De plus, cela nécessite de stocker toutes les données en mémoire, ce qui le rend impraticable pour les ensembles de données qui ne rentrent pas en mémoire. Parce que nous ne changeons pas nos paramètres de modèle aussi fréquemment que d'autres variantes de la descente de gradient, la descente de gradient par lots reste efficace sur le plan computationnel et conduit généralement à un gradient d'erreur stable et à une convergence. Cependant, parfois ce point de convergence n'est pas le meilleur, trouvant le minimum local plutôt que le minimum global.

- **Avantages**

- i. Cette version de la méthode de la descente la plus raide est plus efficace sur le plan computationnel que l'algorithme de descente de gradient stochastique car elle nécessite moins de changements de modèle.
- ii. Pour certaines questions, la réduction de la fréquence de mise à jour entraîne un gradient d'erreur plus stable et une convergence.
- iii. Une mise en œuvre de la méthode basée sur le traitement parallèle est rendue possible en séparant les calculs d'erreur de prédiction et les mises à jour de modèle.

- **Inconvénients**

- i. Le modèle pourrait converger prématûrement vers un ensemble de paramètres défavorable si le gradient d'erreur devient plus stable.
- ii. La difficulté supplémentaire de cumuler les erreurs de prédiction sur tous les échantillons d'entraînement est nécessaire pour les mises à jour de fin d'époque d'entraînement.
- iii. L'ensemble complet des données d'entraînement doit normalement être conservé en mémoire lors de l'utilisation de l'approche de descente de gradient par lots, c'est ainsi que l'algorithme est mis en œuvre.
- iv. Les grands ensembles de données peuvent ralentir considérablement les mises à jour du modèle ou l'entraînement.

v. Cela nécessite l'utilisation de plus de puissance de calcul.

2.4.1.2 La Descente de Gradient Stochastique

Au lieu de parcourir chaque élément de données dans notre ensemble d'entraînement puis de travailler vers un minimum local, la descente de gradient stochastique (DGS) fonctionne en prenant un seul point de données de l'ensemble d'entraînement et en calculant le gradient en fonction de ce point de données unique.

Chaque exemple d'entraînement x_i et l'étiquette y_i change le paramètre de poids w en fonction de la descente de gradient stochastique : $w_{i+1} = w_i - \alpha \cdot \frac{\partial}{\partial w_i} L(x_i, y_i; w_i)$

La descente de gradient présente divers inconvénients, cependant la descente de gradient stochastique réduit considérablement ces inconvénients. Contrairement à la descente de gradient par lots qui effectue des calculs redondants pour les ensembles de données volumineux en recalculant les gradients pour des exemples similaires avant chaque mise à jour des paramètres, la descente de gradient stochastique exécute une epoch d'entraînement pour chaque exemple dans l'ensemble de données et met à jour les paramètres de chaque exemple d'entraînement un à la fois. Par conséquent, elle est souvent beaucoup plus rapide. Ils sont plus simples à stocker en mémoire car nous n'avons besoin de conserver qu'un seul exemple d'entraînement. Comparées à la descente de gradient par lots, ces mises à jour fréquentes peuvent fournir une plus grande précision et une plus grande vitesse, mais elles peuvent également réduire l'efficacité de calcul.

De plus, un inconvénient de la descente de gradient stochastique est la possibilité de mises à jour bruyantes dans l'espace des paramètres. En raison de son caractère aléatoire, la descente de gradient stochastique est plus bruyante que la descente de gradient par lots lors du choix des points de données de l'ensemble d'entraînement pour calculer les gradients à chaque étape.

Nous devons boucler sur les données d'entraînement un certain nombre de fois et nous assurer que les données d'entraînement sont mélangées au début de la descente de gradient afin de tenir compte du bruit de la descente de gradient stochastique et de nous assurer d'arriver à une valeur de paramètre idéale.

Des valeurs de paramètre incertaines résultent du bruit lors de la tentative de calcul de la fonction de perte. Cependant, la descente de gradient stochastique atteindra un minimum local avec suffisamment de temps. Un autre avantage de la descente de gradient stochastique est son imprévisibilité et son bruit. Il est utile pour sortir d'un minimum local qui n'est pas le minimum global lorsque l'algorithme devient "bloqué".

Due à sa nature aléatoire et imprévisible lors de l'attribution des valeurs de paramètres à chaque étape, la descente de gradient stochastique a l'avantage d'échapper aux minimums locaux et de trouver le minimum global par rapport à la descente de gradient par lots. Chaque itération de la descente de gradient stochastique calcule le gradient sur la base d'une partition aléatoirement choisie de l'ensemble de données qui a été mélangé plutôt que d'utiliser l'ensemble complet des observations.

Bien que la descente de gradient stochastique surpassse la descente de gradient par lots en évitant les minimums locaux et en trouvant le minimum global grâce à son comportement aléatoire et imprévisible lors de l'attribution des valeurs de paramètres à chaque étape, les valeurs de paramètres de la descente de gradient par lots sont plus étroitement liées au minimum global et idéal. Lorsqu'il s'agit de choisir entre les deux variations d'algorithme de descente de gradient, il y a un compromis entre la vitesse et l'efficacité.

- **Avantages**

- i. Avec des mises à jour régulières, nous pouvons immédiatement voir l'efficacité du modèle et la vitesse d'amélioration.
- ii. La variation de l'approche de la descente de gradient la plus facile à comprendre et à appliquer est certainement celle-ci.
- iii. Nous pourrons collecter plus rapidement des informations sur certains problèmes en augmentant la fréquence des mises à jour du modèle.
- iv. Le modèle est capable d'éviter les minima locaux en raison du processus de mise à jour bruyant (comme la convergence prématuée).
- v. Plus rapide et nécessitant moins d'efforts de calcul.

vi. Conforme à l'ensemble de données plus large.

- **Inconvénients**

- i. Il faut beaucoup de temps pour entraîner le modèle avec de grands ensembles de données car des mises à jour fréquentes du modèle nécessitent plus de calculs que les autres paramètres de la descente de gradient.
- ii. Des signaux de gradient bruyants peuvent être produits par des mises à jour fréquentes. Cela peut entraîner des fluctuations importantes des paramètres du modèle et de la fonction de perte (plus de variance au cours de la session d'entraînement).
- iii. Il peut être difficile pour l'algorithme de s'engager sur l'erreur minimale du modèle s'il y a un apprentissage bruyant le long du gradient de l'erreur.

2.4.1.3 La descente de Gradient en Mini-Lot

Tant l'algorithme de descente de gradient par lot que l'algorithme de gradient stochastique sont combinés dans l'algorithme de gradient en mini-lot. Cette méthode divise aléatoirement l'ensemble de données d'entraînement en petits sous-ensembles (lots) et calcule les gradients pour chaque lot, plutôt que d'itérer sur l'ensemble de données complet ou une seule observation. Cette méthode crée un compromis entre la vitesse du gradient stochastique et l'efficacité de calcul du gradient de lot.

La formule de la descente de gradient en mini-lot, qui met à jour les poids w en utilisant un lot de données d'entraînement de taille n , est la suivante : $w_{i+1} = w_i - \alpha \cdot \frac{\partial}{\partial w_i} L(x_{i:i+n}, y_{i:i+n}; w_i)$

En utilisant des mini-lots, cela réduit la variance des mises à jour de paramètres, ce qui peut conduire à une convergence plus stable. Il peut également exploiter des optimisations matricielles hautement optimisées courantes dans les bibliothèques modernes de deep learning pour résoudre efficacement le gradient de chaque petit lot de données. Les tailles de mini-lot varient généralement de 50 à 256, mais elles peuvent changer en fonction de l'application.

La descente de gradient en mini-lot est plus lent que la descente de gradient stochastique mais utilise moins de données pour calculer les gradients par rapport à la descente de gradient par lots.

Le fait que le gradient de descente en mini-lot réduise le bruit dans l'espace de paramètres lui confère un avantage significatif par rapport à la descente de gradient stochastique. Par conséquent, par rapport à la descente de gradient stochastique, le gradient de descente en mini-lot permet d'obtenir des valeurs de paramètres optimaux plus réalistes.

- **Avantages**

- i. Par rapport à l'approche de descente de gradient par lots, le modèle est mis à jour plus fréquemment, ce qui permet une convergence plus fiable et empêche les minima locaux.
- ii. Comparé à la descente de gradient stochastique, les mises à jour par petits lots offrent une approche plus efficace sur le plan computationnel.
- iii. L'efficacité de ne pas avoir toutes les données d'entraînement en mémoire et la mise en œuvre de la méthode sont rendues possibles par le traitement par petits lots.

- **Inconvénients**

- i. Des hyperparamètres supplémentaires pour l'algorithme d'apprentissage appelés "taille de mini-lots" sont nécessaires lors de l'utilisation de la descente de gradient en mini-lots.
- ii. Les données d'erreur doivent être entassées sur un petit lot d'échantillons d'entraînement comme la descente de gradient par lots.
- iii. Cela produira des fonctions complexes.

2.5 L'Algorithme du Gradient dans Un Perceptron

Nous supposons que n observations $\{x_1, x_2, \dots, x_n\}$ ne sont pas observées simultanément, mais séquentiellement, l'une après l'autre, et nous essayons de minimiser l'erreur de prédiction sur l'ensemble d'entraînement lors de l'apprentissage d'un perceptron, c'est-à-dire l'apprentissage des poids de

connexion. Comme indiqué précédemment, l'objectif de l'algorithme du perceptron peut être exprimé sous forme de moindres carrés pour la régression linéaire, en ce qui concerne toutes les instances d'entraînement dans un ensemble de données D qui contient des paires de caractéristiques-et-étiquettes:

$$\underset{W}{\text{Min}} \ L = \sum_{(x, y) \in D} (y - \hat{y})^2 = \sum_{(x, y) \in D} (y - h(W \cdot X))^2$$

Nous pouvons supposer que l'algorithme du perceptron utilise l'algorithme du gradient de la fonction de perte par rapport à chaque échantillon, car nous avons déjà remarqué que la fonction de

perte est une fonction lisse et continue des variables : $\nabla_w L = \frac{2 \sum_{i=1}^d (y_i - \hat{y}_i) X_i}{d}, \forall (X_i, y_i) \in D$

Ainsi, l'entraînement d'un perceptron est une procédure itérative. La procédure d'entraînement des réseaux neuronaux consiste à fournir chaque instance de données d'entrée X dans le réseau une par une (ou en petits lots) pour produire la prédiction \hat{y} , même si la fonction d'objectif mentionnée ci-dessus est définie sur l'ensemble des données d'entraînement. Nous modifions les poids de connexion après chaque observation pour réduire l'erreur de prédiction actuelle du perceptron. Pour ce faire, nous utilisons l'algorithme du gradient : le gradient nous montre où une fonction (dans notre exemple, la fonction de perte) a le plus de variance, et nous dit d'aller dans la direction opposée du gradient pour trouver le minimum de la fonction. Le gradient de la fonction est égal à zéro lorsqu'elle est localement minimisée. Ensuite, en fonction de la valeur d'erreur $E(X) = (y - \hat{y})$, les poids sont modifiés. Le vecteur de poids W est modifié comme suit lorsque le point de données X est fourni au réseau :

$$W \leftarrow W - \alpha \cdot \frac{2(y - \hat{y})X}{d}$$

Le taux d'apprentissage est un hyperparamètre du réseau neuronal qui correspond au paramètre α . Si ce dernier est petit, l'algorithme prendra beaucoup de temps pour converger. Par conséquent, il est crucial de choisir judicieusement le taux d'apprentissage. Il existe des méthodes qui peuvent modifier ce taux de manière à ce qu'il soit plus élevé lorsque la réponse est éloignée et plus faible lorsqu'elle est proche.

L'algorithme du perceptron modifie de manière itérative les poids jusqu'à ce que la convergence soit atteinte en parcourant continuellement toutes les observations d'entraînement dans un ordre aléatoire. Le nombre de fois où un point de données d'entraînement est parcouru est à votre discréption.

Les cycles sont utilisés pour décrire chacune de ces itérations. La mise à jour de la descente de gradient peut également être exprimée comme suit en utilisant l'erreur $E(X) = (y - \hat{y})$:

$$W \leftarrow W - \alpha \cdot \frac{2E(X)X}{d}$$

L'algorithme du perceptron analyse chaque exemple d'entraînement X , détermine la prédiction \hat{y} et met à jour le vecteur de poids W conformément à la mise à jour de la descente de gradient tout au long de chaque époque. Jusqu'à ce que le vecteur de poids W trouve une solution qui classifie correctement les données d'entraînement, l'algorithme itère sur les exemples d'entraînement.

L'algorithme fondamental du perceptron peut être considéré comme une technique de descente de gradient stochastique qui, en effectuant des mises à jour de descente de gradient en ce qui concerne des points d'entraînement sélectionnés de manière aléatoire, minimise implicitement l'erreur quadratique de prédiction. On suppose que pendant l'entraînement, le réseau de neurones parcourt les points dans une séquence aléatoire et modifie les poids dans le but de diminuer l'erreur de prédiction sur ce point particulier. Par conséquent, nous commençons par sélectionner des valeurs de poids de connexion initiales $\{w_1, w_2, \dots, w_d\}$ au hasard. L'équation $W \leftarrow W + \alpha \cdot E(X)X$ montre clairement que les modifications de poids non nulles ne sont effectuées qu'après chaque observation (x_i, y_i) lorsque $y \neq \hat{y}$, ce qui n'arrive que lorsqu'une erreur se produit dans la prédiction. Les modifications mentionnées ci-dessus de l'équation $W \leftarrow W + \alpha \cdot E(X)X$ sont appliquées dans la descente de gradient stochastique en mini-lots sur un sous-ensemble de points d'entraînement S sélectionné de manière aléatoire :

$$W \leftarrow W - \alpha \cdot \frac{2 \sum_{i=1}^s E(X_i)X_i}{s} = W - \alpha \cdot \nabla_w L, \quad \forall X_i, y_i \in S$$

Un modèle linéaire, défini par l'équation $W \cdot X = 0$, est le type de modèle proposé par le perceptron. Le vecteur $W = (w_1, \dots, w_d)$ de dimension d est normal à l'hyperplan. De plus, $W \cdot X$ a une valeur positive pour les valeurs de X d'un côté de l'hyperplan et une valeur négative pour les valeurs de X de l'autre côté. Lorsque les données peuvent être séparées linéairement, ce type de modèle fonctionne particulièrement bien. Il était particulièrement important de démontrer que la méthode du perceptron converge vers des solutions appropriées dans certains cas spéciaux, car la technique du perceptron originale a été proposée comme une minimisation des erreurs de classification. Dans ce contexte, il a été démontré que l'algorithme du perceptron est une technique simple et efficace qui fonctionne bien pour les ensembles de données linéairement séparables. Lorsque les données sont linéairement séparables, l'algorithme du perceptron converge toujours pour produire une erreur nulle

sur les données d'entraînement. L'algorithme du perceptron peut ne pas fonctionner efficacement dans des situations où les données ne peuvent pas être séparées linéairement car sa convergence n'est pas garantie. Des algorithmes plus avancés, tels que les machines à vecteurs de support ou les réseaux neuronaux avec des couches cachées, seront plus appropriés dans certaines circonstances.

2.6 La Fonction d'Activation (La Fonction de Transfert)

Une fonction d'activation dans les réseaux de neurones artificiels détermine si un neurone doit être déclenché. Selon l'entrée, un circuit intégré conventionnel peut être considéré comme un réseau numérique de fonctions d'activation qui peuvent être "ALLUMER" (1) ou "ÉTEINDRE" (0). Cela implique qu'il déterminera, à l'aide de processus mathématiques plus simples, si l'entrée du neurone dans le réseau est significative tout au long du processus de prédiction. Par conséquent, l'objectif d'une fonction d'activation d'un nœud est de déterminer la sortie de ce nœud étant donné un ensemble de valeurs d'entrée fournies à ce nœud (ou couche).

Un aspect crucial de l'architecture des réseaux de neurones est la sélection de la fonction d'activation. La nécessité de prévoir une étiquette de classe binaire dans le cas du perceptron a conduit à la sélection de la fonction d'activation de Heaviside. Il peut y avoir d'autres situations, cependant, dans lesquelles différentes variables cibles peuvent être prévues. Par exemple, l'utilisation de la fonction d'activation identité est logique si la variable cible à prédire est réel. Le procédé résultant est identique à la régression des moindres carrés. Utilisez une fonction sigmoïde pour activer le nœud de sortie lors de la prédiction d'une probabilité de classe binaire, de sorte que la prédiction \hat{y} montre la probabilité que la valeur observée, y , de la variable dépendante soit 1. Étant donné que y est codé à partir de $\{-1,1\}$, la perte est le logarithme négatif de $|y/2 - 0.5 + \hat{y}|$. Si $|y/2 - 0.5 + \hat{y}|$ est la probabilité que la valeur prédite soit vraie, alors \hat{y} est la chance que y soit 1. L'examen des deux scénarios dans lesquels y est 0 ou 1 permettra de confirmer cette affirmation. Il peut être démontré que cette fonction de perte est une bonne représentation du négatif de la vraisemblance des données d'entraînement.

Passer du perceptron à une conception multicouche augmente l'importance des fonctions d'activation non linéaires. Il est possible d'utiliser de nombreuses fonctions non linéaires dans différentes couches, notamment la fonction de Heaviside, la sigmoïde et la tangente hyperbolique. La fonction d'activation est indiquée par la notation $\Phi : \hat{y} = \Phi(W \cdot X)$

Parce qu'un neurone calcule en réalité deux fonctions à l'intérieur d'un nœud, nous avons inclus à l'intérieur d'un neurone à la fois le symbole de sommation Σ et le symbole d'activation Φ . La figure 1.7 illustre la division des calculs du neurone en deux valeurs distinctes.

La valeur de pré-activation fait référence à la valeur calculée avant l'utilisation de la fonction d'activation $\Phi(\cdot)$, tandis que la valeur de post-activation fait référence à la valeur calculée après l'application de la fonction d'activation. Bien que les variables de pré-activation soient fréquemment utilisées dans divers types d'analyses, telles que les calculs de l'algorithme de rétropropagation, la sortie d'un neurone est toujours la valeur de post-activation.

L'activation identité ou linéaire, qui n'offre aucune non-linéarité, est la fonction d'activation la plus fondamentale $\Phi(v) = v$:

Lorsque l'objectif est une valeur réelle, le nœud de sortie utilise souvent la fonction d'activation linéaire. Même des sorties discrètes sont employées lors de la mise en place d'une fonction de perte de substitution lissée. Les fonctions d'activation traditionnelles qui ont été employées au début de la construction des réseaux neuronaux étaient le heaviside, la sigmoïde et la tangente hyperbolique :

$$\begin{aligned}\Phi(v) &= \mathbf{1}_{\mathbb{N}}(x) \text{ (Fonction Heaviside)} \\ \Phi(v) &= \frac{1}{1+e^{-v}} \text{ (Fonction Sigmoïde)} \\ \Phi(v) &= \frac{e^{2v}-1}{e^{2v}+1} \text{ (Fonction tanh)}\end{aligned}$$

Bien que la fonction d'activation Heaviside puisse être utilisée pour mapper des sorties binaires pendant la prédiction, elle ne peut pas être utilisée pour construire la fonction de perte pendant l'entraînement en raison de sa non-différentiabilité. Par exemple, les critères du perceptron en entraînement ne nécessitent qu'une activation linéaire, mais le perceptron utilise la fonction Heaviside pour la prédiction. Lors de calculs qui doivent être interprétés comme des probabilités, la fonction sigmoïde renvoie une valeur dans l'intervalle $(0,1)$, ce qui est utile. De plus, elle est utile pour la construction de fonctions de perte générées à partir de modèles de vraisemblance maximale et pour produire des sorties probabilistes. La fonction tanh est similaire en forme à la fonction sigmoïde, sauf qu'elle a été translatée et mise à l'échelle verticalement vers $[-1,1]$ et horizontalement vers $[-1,1]$. Voici quelques relations entre les fonctions tanh et sigmoïde : $\tanh(v) = 2 \cdot \text{sigmoïde}(2v) - 1$

Lorsque des résultats positifs et négatifs sont nécessaires dans les calculs, la fonction tanh est préférée à la fonction sigmoïde. De plus, il est plus facile de l'entraîner en raison de son centrage moyen et de son gradient plus grand (en raison de l'étirement) que la sigmoïde. Dans le passé, les fonctions sigmoïde et tanh étaient les méthodes privilégiées pour inclure la non-linéarité dans les réseaux de neurones. Plusieurs fonctions d'activation linéaires par morceaux ont récemment gagné en popularité, notamment :

$$\begin{aligned}\Phi(v) &= \max\{v, 0\} \text{ (ReLU)} \\ \Phi(v) &= \max\{\min\{v, 1\}, -1\} \text{ (hard tanh)}\end{aligned}$$

En raison de la simplicité de l'entraînement des réseaux de neurones multicouches avec les fonctions d'activation ReLU et hard tanh, les fonctions d'activation sigmoïde et soft tanh ont largement été remplacées dans les réseaux de neurones actuels.

La figure 1.8 montre les représentations visuelles de chacun des processus d'activation mentionnés ci-dessus. Il est important de remarquer que chaque fonction d'activation présentée ici est monotone. Mis à part la fonction d'activation linéaire, la plupart des autres fonctions d'activation se saturent à des valeurs absolues élevées de l'argument, à partir desquelles augmenter davantage l'argument a peu d'effet sur l'activation.

En outre, ces fonctions d'activation non linéaires sont très utiles dans les réseaux multicouches car elles permettent la construction de combinaisons plus puissantes de différents types de fonctions. Comme elles convertissent les sorties de n'importe quelle plage en sorties bornées, bon nombre de ces fonctions sont connues sous le nom de fonctions d'écrasement. L'adoption d'une activation non linéaire est essentielle pour améliorer la capacité d'un réseau à modéliser. Un réseau n'aurait pas plus de capacité de modélisation qu'un réseau linéaire à une seule couche s'il employait simplement des activations linéaires.

2.7 Perceptron Multicouche

De nombreuses couches de calcul sont présentes dans les réseaux de neurones multicouches. La couche de sortie est la seule couche qui effectue des calculs dans le perceptron, qui possède des couches d'entrée et de sortie. Tous les calculs sont entièrement transparents pour l'utilisateur car ils sont transmis de la couche d'entrée à la couche de sortie. Les couches intermédiaires supplémentaires (entre l'entrée et la sortie) dans les réseaux de neurones multicouches sont appelées couches cachées car les calculs qu'elles effectuent sont cachés à l'utilisateur. Étant donné que les couches successives se

nourrissent les unes des autres dans une direction avant de l'entrée à la sortie, l'architecture spécifique des réseaux de neurones multicouches est appelée "réseau à propagation avant". Dans les réseaux à propagation avant, il est entendu que chaque nœud dans une couche est relié à chaque autre nœud dans la couche supérieure. Une fois que le nombre de couches et le nombre/type de nœuds dans chaque couche ont été établis, l'architecture du réseau de neurones est presque entièrement définie. La fonction de perte, qui est optimisée à la couche de sortie, est le seul composant restant. Le critère de perceptron est utilisé par l'algorithme de perceptron, mais ce n'est pas le seul. Il est assez courant d'utiliser des sorties linéaires avec une perte quadratique pour une prédiction à valeur réelle et des sorties softmax avec une perte de croix-entropie pour une prédiction discrète.

Les neurones de biais peuvent être utilisés dans les couches cachées et les couches de sortie, tout comme dans les réseaux à une seule couche. La figure 1.9(a) et (b) montrent des exemples de réseaux multicouches avec ou sans neurones de biais, respectivement. Le réseau neuronal a trois couches dans chaque scénario. Comme il ne transmet que des données et n'effectue aucune computation, la couche d'entrée n'est souvent pas comptée. Les représentations vectorielles (en colonne) de ces sorties, indiquées par $\bar{h}_1 \dots \bar{h}_k$ et ayant des dimensions $p_1 \dots p_k$, sont présentes dans un réseau neuronal si chacune de ses k couches a $p_1 \dots p_k$ unités. Par conséquent, la dimensionnalité de chaque couche est définie comme le nombre total d'unités dans cette couche.

Alors que les poids entre la r -ème couche cachée et la $(r+1)$ -ème couche cachée sont notés par la matrice $p_r \cdot p_{r+1}$ notée W_r , les poids entre la couche d'entrée et la première couche cachée sont contenues dans une matrice W_1 de taille $d \cdot p_1$. La matrice finale W_{k+1} est de dimension $p_k \cdot o$ si la couche de sortie a o nœuds. Les équations récursives suivantes convertissent le vecteur d'entrée \bar{x} de

$$\bar{h}_1 = \Phi(W_1^T \bar{x}) \quad [\text{Entrée dans la couche cachée}]$$

dimension d en sorties : $\bar{h}_{p+1} = \Phi(W_{p+1}^T \bar{h}_p), \forall p \in [1 \dots k-1]$ $[\text{Cachée vers la couche cachée}]$

$$\bar{o} = \Phi(W_{k+1}^T \bar{h}_k) \quad [\text{Cachée vers la couche de sortie}]$$

Ici, l'application élément par élément de fonctions d'activation, comme la fonction sigmoïde, à leurs arguments de vecteur est utilisée. Cependant, certains algorithmes d'activation, comme le softmax (qui est fréquemment utilisé dans les couches de sortie), ont naturellement des arguments de vecteur. Bien que chaque unité de réseau neuronal ne contienne qu'une seule variable, de nombreux plans architecturaux regroupent les unités en une seule couche pour former une unité de vecteur unique, qui est représentée par un rectangle plutôt qu'un cercle. Par exemple, la figure 1.9(d) montre une transformation du modèle architectural basé sur les unités scalaires de la figure 1.9(c) en une

architecture neuronale basée sur les vecteurs. Des connexions matricielles sont utilisées pour connecter les unités de vecteurs. L'architecture neuronale basée sur les vecteurs suppose également que chaque unité de chaque couche utilise la même fonction d'activation, qui est appliquée à chaque couche de manière élémentaire. Comme la plupart des architectures neuronales utilisent la même fonction d'activation tout au long du pipeline de calcul, avec la seule variation apportée par les caractéristiques de la couche de sortie, cette contrainte ne pose généralement aucun problème. Dans le rapport suivant, des réseaux neuronaux avec des unités circulaires pour les variables scalaires et des unités rectangulaires pour les unités contenant des variables vectorielles seront présentés.

Les équations de récurrence et les topologies vectorielles discutées ci-dessus ne s'appliquent qu'aux réseaux de neurones à alimentation directe couche par couche, et ne peuvent donc pas toujours être appliquées à des conceptions architecturales non standard. La topologie peut permettre des connexions entre des niveaux non consécutifs ou des entrées peuvent être insérées dans des couches intermédiaires, permettant toutes sortes de conceptions non orthodoxes. De plus, les fonctions calculées au niveau d'un nœud ne peuvent pas nécessairement prendre la forme d'une union d'une fonction d'activation et d'une fonction linéaire. N'importe quel type de fonction de calcul arbitraire peut exister au niveau d'un nœud.

La figure 1.9 représente un type de conception relativement traditionnel, bien qu'il soit possible de le modifier de diverses manières, comme en permettant plusieurs nœuds de sortie. Les objectifs de l'application en question (comme la classification ou la réduction de dimensionnalité) dictent souvent ces décisions. L'auto-encodeur, qui produit les sorties à partir des entrées, est un exemple classique de configuration de réduction de dimensionnalité. Comme on le voit sur la figure 1.10, la quantité de sorties et d'entrées est donc égale. La représentation condensée de chaque instance est produite par la couche cachée constrictive centrale. Cette contrainte entraîne une petite perte de représentation, qui est généralement liée au bruit dans les données. Les sorties des couches cachées correspondent à la forme condensée des données. En fait, il peut être démontré qu'une version simplifiée de cette méthodologie est mathématiquement similaire à la méthode de décomposition en valeurs singulières, une technique populaire de réduction de dimensionnalité. Par conséquent, les réseaux plus profonds offrent des réductions naturellement plus puissantes.

Bien qu'une architecture entièrement connectée puisse fonctionner efficacement dans de nombreuses situations, de meilleures performances sont souvent obtenues en réduisant le nombre de connexions ou en les partageant de manière plus intelligente. Ces idées sont souvent découvertes en

utilisant une compréhension spécifique du domaine des données. L'architecture des réseaux de neurones convolutifs, dans laquelle l'architecture est délibérément construite pour correspondre aux caractéristiques habituelles des données d'image, est un exemple célèbre de ce type d'élagage et de partage de poids. En ajoutant une connaissance spécifique du domaine (ou un biais), une telle stratégie réduit le risque de sur-ajustement. Un problème courant dans la conception de réseaux de neurones est le sur-ajustement, qui amène le réseau à fonctionner souvent extrêmement bien sur les données d'entraînement, mais à mal généraliser aux données de test inconnues. Ce problème survient lorsqu'il y a trop de paramètres libres par rapport à la quantité de données d'entraînement, ce qui est généralement équivalent au nombre de connexions pondérées. Dans de telles situations, les nombreux paramètres rappellent les détails complexes des données d'entraînement mais sont incapables d'identifier les motifs statistiquement significatifs pour catégoriser les données de test inconnues. Il est évident que l'ajout de plus de nœuds au réseau de neurones a tendance à favoriser le sur-ajustement. Pour réduire le sur-ajustement, de nombreuses recherches récentes se sont concentrées à la fois sur la conception du réseau de neurones et sur les calculs effectués dans chaque nœud. De plus, la qualité de la réponse finale est également influencée par la façon dont le réseau de neurones est entraîné. La préformation est l'une des plusieurs stratégies innovantes pour éléver le niveau de la réponse apprise.

2.8 Rétropropagation

La procédure d'entraînement pour un réseau neuronal à couche unique est très simple car le gradient peut être calculé facilement et l'erreur (ou fonction de perte) peut être calculée directement en fonction des poids. Le problème avec les réseaux multicouches est que les poids dans les couches précédentes sont des fonctions de composition sophistiquées qui causent la perte. L'algorithme de rétropropagation est utilisé pour calculer le gradient d'une fonction de composition. La règle de la chaîne de calcul différentiel est utilisée par l'algorithme de rétropropagation pour calculer les gradients d'erreur en tant que sommes de produits de gradient locaux le long des nombreux chemins d'un nœud à la sortie. Bien que cette sommation implique un nombre exponentiel de parties (chemins), la programmation dynamique peut être utilisée pour la calculer rapidement. La programmation dynamique est directement appliquée dans l'algorithme de rétropropagation. Il se compose de deux phases principales, la phase directe et la phase inverse, respectivement. Il est nécessaire pour la phase directe de calculer les valeurs de sortie et les dérivées locales à différents nœuds, et il est nécessaire pour la phase inverse de collecter les produits de ces valeurs locales le long de tous les chemins menant à la sortie:

- Propagation avant : Pendant cette phase, les entrées pour une instance d'entraînement sont fournies au réseau neuronal. Cela provoque une cascade de calculs en avant à travers les couches en utilisant l'ensemble de poids actuel. La dérivée de la fonction de perte par rapport à la sortie est calculée, et la sortie finale prédite peut être comparée à celle de l'instance d'entraînement. Il est maintenant nécessaire de calculer la dérivée de cette perte par rapport aux poids de chaque couche dans la phase de retour en arrière.
- Rétropropagation : En utilisant la règle de la chaîne de calcul différentiel, l'objectif principal de la phase est de déterminer le gradient de la fonction de perte par rapport aux différents poids. Les poids sont mis à jour en utilisant ces gradients. Ce processus d'apprentissage est appelé la phase arrière parce que ces gradients sont appris dans la direction inverse, en commençant par le nœud de sortie. Considérez un ensemble de sorties o suivi d'une série d'unités cachées h_1, h_2, \dots, h_k , par rapport auxquelles la fonction de perte L est calculée. Supposons en outre que $w_{(h_r, h_{r+1})}$ est le poids du lien de l'unité cachée h_r à h_{r+1} . Le gradient de la fonction de perte par rapport à l'un quelconque de ces poids d'arête peut alors être déterminé en utilisant la règle de la chaîne dans le cas où un seul chemin mène de h_1 à o :

$$\frac{\partial L}{\partial w_{(h_{r-1}, h_r)}} = \frac{\partial L}{\partial o} \cdot \left[\frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{(h_{r-1}, h_r)}}, \quad \forall r \in 1 \dots k \quad (2.8.1)$$

L'équation susmentionnée suppose qu'il n'y a qu'un seul chemin de h_1 à o dans le réseau, même s'il peut y avoir en réalité un nombre exponentiel de chemins. La règle de chaîne multivariable est une version élargie de la règle de chaîne qui calcule le gradient dans un réseau computationnel où plusieurs chemins peuvent exister. La composition est ajoutée le long de chacun des chemins de h_1 à o pour y parvenir. La figure 1.13 illustre la règle de chaîne en action à l'aide d'un graphe computationnel avec deux chemins. Par conséquent, la formule susmentionnée est généralisée à la situation où un ensemble P de chemins existe de h_r à o :

$$\frac{\partial L}{\partial w_{(h_{r-1}, h_r)}} = \frac{\partial L}{\partial o} \cdot \left[\sum_{[h_r, h_{r+1}, \dots, h_k, o] \in P} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{(h_{r-1}, h_r)}} \quad (2.8.2)$$

où $\frac{\partial L}{\partial o} \cdot \left[\sum_{[h_r, h_{r+1}, \dots, h_k, o] \in P} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]$ est la rétropropagation calcule $\Delta(h_r, o) = \frac{\partial L}{\partial h_r}$

Le calcul de $\frac{\partial h_r}{\partial w_{(h_{r-1}, h_r)}}$ du côté droit sera couvert dans la section suivante (Équation 2.8.6). En revanche, le terme agrégé de chemin ci-dessus [annoté par $\Delta(h_r, o) = \frac{\partial L}{\partial h_r}$] est agrégé sur un nombre exponentiellement croissant de chemins (en ce qui concerne la longueur du chemin), ce qui semble à première vue être inextricable. Le fait que le graphe de calcul d'un réseau neuronal soit acyclique permet de calculer une telle agrégation de manière raisonnée dans le sens inverse en calculant d'abord $\Delta(h_r, o)$ pour les nœuds h_k les plus proches de o , puis en calculant de manière récursive ces valeurs pour les nœuds dans les couches antérieures en termes de nœuds dans les couches ultérieures. De plus, la valeur initiale de chaque nœud de sortie pour $\Delta(o, o)$ est la suivante: $\Delta(o, o) = \frac{\partial L}{\partial o}$ (2.8.3)

Lorsque l'on calcule toutes sortes de fonctions centrées sur les chemins dans des graphes acycliques dirigés, qui nécessiteraient autrement un nombre exponentiel d'opérations, cette technique de programmation dynamique est fréquemment utilisée. La règle de la chaîne multivariée peut être utilisée pour obtenir la récursion pour $\Delta(h_r, o)$:

$$\Delta(h_r, o) = \frac{\partial L}{\partial h_r} = \sum_{h: h_r \Rightarrow h} \frac{\partial L}{\partial h} \frac{\partial h}{\partial h_r} = \sum_{h: h_r \Rightarrow h} \frac{\partial h}{\partial h_r} \Delta(h, o) \quad (2.8.4)$$

Chaque h est dans une couche postérieure à h_r , donc lors de l'évaluation de $\Delta(h_r, o)$, $\Delta(h, o)$ a déjà été calculé. Cependant, pour calculer l'Équation 2.8.4, nous devons encore évaluer $\frac{\partial h}{\partial h_r}$. Soit a_h la valeur calculée dans l'unité cachée h juste avant d'appliquer la fonction d'activation $\Phi(\cdot)$ dans le scénario où le lien reliant h_r à h a un poids $w_{h_r, h}$. Ainsi, $h = \Phi(a_h)$, où a_h est une combinaison linéaire de ses entrées à partir d'unités de couches antérieures incidentes à h . La formule pour $\frac{\partial h}{\partial h_r}$ qui en résulte peut alors être obtenue en utilisant la règle de la chaîne univariée de la manière suivante :

$$\frac{\partial h}{\partial h_r} = \frac{\partial h}{\partial a_h} \cdot \frac{\partial a_h}{\partial h_r} = \frac{\partial \Phi(a_h)}{\partial a_h} \cdot w_{(h_r, h)} = \Phi'(a_h) \cdot w_{(h_r, h)}$$

L'équation 2.8.4 qui commence par le nœud de sortie et boucle de manière récursive dans la direction opposée, utilise cette valeur de $\frac{\partial h}{\partial h_r}$. Les modifications suivantes sont équivalentes dans la direction inverse : $\Delta(h_r, o) = \sum_{h: h_r \Rightarrow h} \Phi'(a_h) \cdot w_{(h_r, h)} \cdot \Delta(h, o)$ (2.8.5)

En conséquence, chaque nœud est traité précisément une fois pendant une passe arrière, et les gradients sont cumulativement ajoutés dans la direction arrière. De plus, afin de calculer le gradient par rapport à tous les poids de bord, l'équation 2.8.4 (qui appelle des opérations proportionnelles au nombre de bords sortants) doit être effectuée pour chaque bord entrant dans le nœud. Enfin, l'équation 2.8.2

appelle le calcul de $\frac{\partial h_r}{\partial w_{(h_{r-1}, h_r)}}$, ce qui peut être fait simplement en suivant ces étapes :

$$\frac{\partial h_r}{\partial w_{(h_{r-1}, h_r)}} = h_{r-1} \cdot \Phi'(a_{h_r}) \quad (2.8.6)$$

Le gradient par rapport aux poids est simple à calculer pour chaque bord d'incidence sur l'unité appropriée, et le gradient par rapport aux activations de couche est le gradient crucial qui est rétro-propagé dans ce cas.

De plus, selon les variables utilisées pour le chaînage intermédiaire, la récursion de programmation dynamique de l'Équation 2.8.5 peut être calculée de plusieurs manières. En ce qui concerne le résultat final de la rétropropagation, toutes ces récursions sont équivalentes. Dans les sections qui suivent, nous proposons une alternative à la récursion de programmation dynamique. En attendant, les variables "chaîne" pour la récursion de programmation dynamique dans l'Équation 2.8.2 sont les variables dans les couches cachées. Les valeurs de pré-activation des variables pour la règle de chaînage peuvent également être utilisées. Après avoir utilisé la transformation linéaire (mais avant d'utiliser les variables d'activation comme variables intermédiaires), les variables de pré-activation dans un neurone sont obtenues. La variable cachée $h = \Phi(a_h)$ a pour valeur de pré-activation a_h . La figure 1.7 illustre les variations entre les valeurs de pré- et post-activation dans un neurone. Par conséquent, la

règle de chaînage décrite ci-dessous peut être utilisée à la place de l'Équation 2.8.2 :

$$\frac{\partial L}{\partial w_{(h_{r-1}, h_r)}} = \frac{\partial L}{\partial o} \cdot \Phi'(a_o) \cdot \left[\sum_{[h_r, h_{r+1}, \dots, h_k, o] \in P} \frac{\partial a_o}{\partial a_{h_k}} \prod_{i=r}^{k-1} \frac{\partial a_{h_{i+1}}}{\partial a_{h_i}} \right] \frac{\partial a_{h_r}}{\partial w_{(h_{r-1}, h_r)}} \quad (2.8.7)$$

où $\frac{\partial L}{\partial o} \cdot \Phi'(a_o) \cdot \left[\sum_{[h_r, h_{r+1}, \dots, h_k, o] \in P} \frac{\partial a_o}{\partial a_{h_k}} \prod_{i=r}^{k-1} \frac{\partial a_{h_{i+1}}}{\partial a_{h_i}} \right]$ est la rétropropagation calculée $\delta(h_r, o) = \frac{\partial L}{\partial a_{h_r}}$

Dans ce cas, l'équation récursive a été construite en utilisant la notation $\delta(h_r, o) = \frac{\partial L}{\partial a_{h_r}}$ plutôt

que $\Delta(h_r, o) = \frac{\partial L}{\partial h_r}$. Voici comment la valeur de $\delta(o, o) = \frac{\partial L}{\partial a_o}$ est initialisée:

$$\delta(o, o) = \frac{\partial L}{\partial a_o} = \Phi'(a_o) \cdot \frac{\partial L}{\partial o} \quad (2.8.8)$$

Ensuite, on peut établir une récursion liée en utilisant la règle de la chaîne multivariable :

$$\Delta(h_r, o) = \frac{\partial L}{\partial a_{h_r}} = \sum_{h: h_r \Rightarrow h} \frac{\partial L}{\partial a_h} \frac{\partial a_h}{\partial a_{h_r}} = \Phi'(a_{h_r}) \sum_{h: h_r \Rightarrow h} w_{(h_r, h)} \delta(h, o) \quad (2.8.9)$$

$$\text{où } \frac{\partial L}{\partial a_h} = \delta(h, o) \text{ et } \frac{\partial a_h}{\partial a_{h_r}} = \Phi'(a_{h_r}) w_{(h_r, h)}$$

La rétropropagation est plus courante dans ce cas récurrent. Ensuite, en utilisant $\delta(h_r, o)$, la dérivée partielle de la perte par rapport au poids est calculée comme suit :

$$\frac{\partial L}{\partial w_{(h_{r-1}, h_r)}} = \delta(h_r, o) \cdot h_{r-1} \quad (2.8.10)$$

Le processus de mise à jour des nœuds se poursuit jusqu'à la convergence, tout comme avec le réseau à une seule couche, en parcourant itérativement les données d'entraînement en epochs. Pour apprendre les poids aux différents nœuds, un réseau neuronal peut parfois avoir besoin de parcourir des milliers d'epochs de données d'entraînement.

3 LES APPLICATIONS SIMPLE SANS BIBLIOTHEQUE DE DEEP LEARNING PAR MILO SPENCER-HARPER

3.1 Première Application Avec Un Perceptron Simple

Dans cette partie, je vais utiliser l'ensemble d'entraînement suivant pour entraîner un neurone à résoudre le problème suivant (figure 1.12):

- $x_1 = (0,0,1)$
- Entrée : $x_2 = (1,1,1)$
 $x_3 = (1,0,1)$
 $x_4 = (0,0,1)$
- $y_1 = 0$
- Sortie : $y_2 = 1$
 $y_3 = 1$
 $y_4 = 0$
- Nouvelle situation : $x_5 = (1,0,0)$
- Sortie : $\hat{y} = ?$

À partir de l'ensemble d'entraînement, la sortie est toujours égale à la valeur de la première colonne dans l'entrée. Par conséquent, la valeur cible y dans la nouvelle situation est égale à 1 et la valeur prédictive \hat{y} devrait également être 1 dans cette situation.

3.1.1 Procédure d'Entraînement

J'ai attribué à chaque entrée un poids, qui peut être une valeur positive ou négative, afin d'entraîner le neurone à fournir la réponse appropriée à la question. La sortie du neurone sera fortement influencée par une entrée avec un grand poids positif ou un grand poids négatif. J'ai donc choisi un poids aléatoire pour chacun. La procédure d'entraînement commencera alors :

- Pour déterminer la sortie d'un neurone, prendre les entrées d'un exemple d'ensemble d'entraînement, les peser correctement, puis les faire passer à travers une formule.
- Calculer l'erreur, qui est la différence entre la sortie du neurone et la sortie cible dans l'exemple de l'ensemble d'entraînement.
- Les poids doivent être légèrement modifiés en fonction de la direction de l'erreur.

- Exécuter cette procédure un total de 10 000 fois.

Les poids du neurone atteindront éventuellement leur niveau idéal pour l'ensemble d'entraînement. Le neurone devrait produire une prévision précise si on lui donne le temps de considérer une nouvelle situation qui correspond au même modèle (figure 1.13).

3.1.2 L'Équation Pour Estimer la Sortie du Neurone

La somme pondérée des entrées du neurone est la formule pour déterminer la sortie du neurone, qui est : $\sum poid_i \cdot entrée_i = poid_1 \cdot entrée_1 + poid_2 \cdot entrée_2 + poid_3 \cdot entrée_3$

Ensuite, il faut normaliser de sorte que le résultat soit compris entre 0 et 1. J'utilise la fonction Sigmoïde (figure 1.8) pour cela : $\frac{1}{1+e^{-x}}$

L'équation finale pour la sortie du neurone est donc obtenue en modifiant la première équation en la seconde : $Sortie\ du\ neurones = \frac{1}{1+e^{-(\sum poid_i \cdot entrée_i)}}$

3.1.3 La Formule d'Ajustement des Poids

Utilisez l'algorithme "Dérivée pondérée par l'erreur" pour changer les poids pendant le cycle d'entraînement (figure 1.13) :

Ajuster les poids par = erreur · entrée · gradient de la courbe sigmoïde (sortie)

Premièrement, je mettre l'échelle de correction à la magnitude de l'erreur. Ensuite, multipliez par l'entrée, qui est soit un 0, soit un 1. Si l'entrée est 0, le poids ne change pas. Enfin, multipliez par le gradient de la courbe Sigmoïde (Figure 1.8). Considérez ce qui suit pour comprendre ce dernier :

1. J'ai utilisé la courbe Sigmoïde pour déterminer la sortie du neurone.
2. Si la sortie est une valeur positive ou négative significative, le neurone était très certain de sa décision.
3. La Figure 1.8 montre la faible pente de la courbe Sigmoïde pour les nombres élevés.

- Le neurone ne veut pas changer le poids beaucoup s'il croit que le poids actuel est précis. Cela est réalisé en multipliant par la pente de la courbe Sigmoïde.

En prenant la dérivée, on peut obtenir la pente de la courbe Sigmoïde :

$$\text{la pente de la courbe Sigmoïde (sortie)} = \text{sortie} \cdot (1 - \text{sortie})$$

La formule finale pour modifier les poids est donc la suivante après avoir inséré la deuxième équation dans la première équation : *Ajuster les poids par = erreur · entrée · sortie · (1 - sortie)*

3.1.4 Résultat

Le réseau neuronal s'est entraîné en utilisant les données d'entraînement après s'être d'abord attribué des poids aléatoires. Ensuite, il a pris en compte un nouveau situation [1,0,0] et a prédit que 0,99993704 est très proche de la valeur cible de $y=1$ (figure 1.14).

3.2 Deuxième Application Avec Un Perceptron Multicouche

Dans cette deuxième partie, je vais entraîner un perceptron multicouche pour résoudre un problème similaire avec l'ensemble d'entraînement suivant (figure 1.15) :

- $x_1 = (0,0,1)$
- $x_2 = (0,1,1)$
- $x_3 = (1,0,1)$
- Entrées : $x_4 = (0,1,0)$
- $x_5 = (1,0,0)$
- $x_6 = (1,1,1)$
- $x_7 = (0,0,0)$

- $y_1 = 0$
- $y_2 = 1$
- $y_3 = 1$
- Sorties : $y_4 = 1$
- $y_5 = 1$
- $y_6 = 0$
- $y_7 = 0$
- Nouvelle situation : $x_8 = (1,1,0)$

- Sortie : $\hat{y}=?$

En effet, la troisième colonne de cet ensemble d'entraînement n'est pas important, cependant les deux premières colonnes démontrent le comportement d'une porte XOR (figure 1.16). La sortie est de 1 si la première colonne ou la deuxième colonne est de 1. Cependant, la sortie est de 0 si les deux colonnes sont 0 ou 1. Ainsi, 0 est la valeur cible.

Cependant, un seul neurone ne peut pas gérer autant d'informations. Il n'y a pas de lien direct un à un entre les entrées et la sortie, ce qui est ce que l'on entend par "modèle non-linéaire" dans ce cas. Je doit donc ajouter une deuxième couche cachée (Couche 1) composée de quatre neurones. Le réseau neuronal peut ainsi considérer les combinaisons d'entrée.

La sortie de la couche 1 dans ce perceptron multicouche passe dans la couche 2. Le réseau neuronal est maintenant capable d'identifier les corrélations entre la sortie de la couche 1 et la sortie de l'ensemble d'entraînement. En modifiant les poids dans chaque couche, le réseau neuronal renforcera ces associations à mesure qu'il acquiert plus de connaissances.

De plus, le problème de reconnaissance d'image est très similaire à cette situation. Par exemple, les pommes et les pixels n'ont pas de lien direct. Cependant, il existe une connexion directe entre les combinaisons de pixels et les pommes.

Le code a été modifié par rapport à le code précédent dans l'application 1. Il y a plus de couches cette fois-ci, ce qui est différent de l'exemple précédent. Les poids sont ajustés lorsque le réseau neuronal calcule l'erreur dans la couche 2 et la propage en arrière vers la couche 1. Ce processus est appelé "rétropropagation".

3.2.1 Résultat

Le réseau neuronal s'est entraîné en utilisant les données d'entraînement après avoir attribué des poids aléatoires à ses connexions synaptiques. Ensuite, il a réfléchi à une situation inconnue [1, 1, 0], où la valeur prédictive \hat{y} est 0,0078876 et la valeur cible $y=0$ sont toutes deux proches l'une de l'autre (figure 1.17).

4 CONCLUSION

Les réseaux de neurones artificiels, qui ont été développés pour reproduire numériquement le cerveau humain, peuvent apprendre, s'adapter et répondre à de nouvelles conditions, contrairement aux programmes informatiques traditionnels, qui sont souvent incapables de le faire. Mais est-ce qu'un perroquet acquerra finalement sa propre conscience s'il peut répondre à toutes les questions avec perfection ? Bien que nous ne puissions pas encore répondre à cette question maintenant, personne ne sait combien de temps durera le "maintenant".

Mais avant cela, les réseaux de neurones artificiels peuvent être utilisés pour construire la prochaine génération d'ordinateurs, sont déjà utilisés pour effectuer des analyses complexes dans divers secteurs, de l'ingénierie à la médecine. L'industrie du jeu vidéo s'appuie déjà fortement sur les réseaux de neurones artificiels. De plus, nous pouvons les utiliser pour lire l'écriture manuscrite, ce qui est utile dans des domaines tels que la banque. Dans le monde de la médecine, les réseaux de neurones artificiels sont également capables de nombreuses tâches cruciales. Ils peuvent être utilisés pour créer des modèles du corps humain qui aideront les médecins à diagnostiquer correctement les maladies de leurs patients.

De plus, des images médicales complexes telles que les scanners CT peuvent maintenant être analysées plus rapidement et précisément grâce aux réseaux de neurones artificiels. De nombreux problèmes abstraits seront résolus par des ordinateurs basés sur des réseaux de neurones. À partir de leurs erreurs, ils se développeront. Peut-être qu'à l'avenir, un dispositif appelé interface cerveau-machine nous permettra de relier les gens aux machines. Cela convertira les impulsions mentales des personnes en signaux que les ordinateurs peuvent utiliser pour fonctionner. Peut-être que toutes nos interactions avec l'environnement seront mentales à l'avenir.

À une époque où tout change rapidement, nous devons continuer à rivaliser avec le temps. En tant que personnes ordinaires, nous ne pouvons pas prévoir l'avenir, nous ne devrions pas avoir peur ou nous inquiéter que l'IA remplace notre travail, car l'avenir est irrésistible, donc nous devrions nous mettre à jour avec notre temps, en apprendre davantage sur l'IA et laisser l'IA nous aider à effectuer autant de tâches triviales que possible. Enfin, ce n'est pas l'IA qui nous remplacera, mais les personnes qui l'utilisent.

5 BIBLIOGRAPHIE

https://d1wqtxts1xzle7.cloudfront.net/31377814/Artificial_Neural_Network-libre.pdf?1392407140=&response-content-disposition=inline%3B+filename%3DIISTE_May_30th_Edition_Peer_reviewed_art.pdf&Expires=1674826173&Signature=XL-4esQxs9V8FCUISNUk1TsLEyt~dWXTaXDGh7Ui7hmsZPQsXLCZZS8gX03kehb0b9uW5BEJyR9mqfPP8pp5IRADPvl-bRgGSig7fqX2VSklncUwiQTpj1cWfZLp15OJDXmlfbRsHsnV-HamOgOl9-deWb8q5uvK81KuYzE-hiieV8IDekj7~MXYH1t8YVseRGRhc0z~iZi0G5CMKJZaR3c1F27DpD~p3VmzmzE7KmakbMhVhdWszKxtcmv1w1wD4IHvlznaOWoOAceNZ2WRfDAJrXwdeLRlrvJFodzUXrVwlA1dI152gl5m6HMODle-crY5y9YhxmBrSA_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA

L'apprentissage profond (Deep Learning) - Livre d'Aaron Courville, Ian Goodfellow et Yoshua Bengio (<https://www.deeplearningbook.org/>)

Neural Networks and Deep Learning: A Textbook - Livre du Charu C. Aggarwal (http://ndl.ethernet.edu.et/bitstream/123456789/88552/1/2018_Book_NeuralNetworksAndDeepLearning.pdf)

<https://statquest.org/video-index/>

<http://wikistat.fr/>

<https://scilogis.fr/intelligence-mecanique/repronons-bases-neurone-artificiel-neurone-biologique/>

<https://www.datarobot.com/blog/introduction-to-loss-functions/#:~:text=There%20are%20several%20different%20common,just%20to%20name%20a%20few.%E2%80%9D>

<https://www.analyticsvidhya.com/blog/2022/06/understanding-loss-function-in-deep-learning/>

<https://builtin.com/machine-learning/loss-functions>

<https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9#:~:text=A%20loss%20function%20is%20a,the%20predicted%20and%20target%20outputs.>

<http://iamtrask.github.io/2015/07/12/basic-python-network/>

<https://shiva-verma.medium.com/understanding-different-loss-functions-for-neural-networks-dd1ed0274718>

<https://kids.frontiersin.org/articles/10.3389/frym.2021.560631>

<https://medium.com/technology-invention-and-more/how-to-build-a-simple-neural-network-in-9-lines-of-python-code-cc8f23647ca1>

<https://medium.com/technology-invention-and-more/how-to-build-a-multi-layered-neural-network-in-python-53ec3d1d326a>

<https://towardsdatascience.com/a-definitive-explanation-to-hinge-loss-for-support-vector-machines-ab6d8d3178f1>

<https://towardsdatascience.com/an-introduction-to-gradient-descent-and-backpropagation-in-machine-learning-algorithms-a14727be70e9>

https://www.wikiwand.com/fr/Algorithme_du_gradient

<https://www.ibm.com/topics/gradient-descent#:~:text=There%20are%20three%20types%20of,and%20mini%2Dbatch%20gradient%20descent.>

<https://openclassrooms.com/fr/courses/4470406-utilisez-des-modeles-supervises-non-lineaires/4730716-entrainez-un-reseau-de-neurones-simple>

[https://www.ruder.io/optimizing-gradient-descent/#:~:text=Batch%20gradient%20descent,-Vanilla%20gradient%20descent&text=%CE%B8%3D%CE%B8%E2%88%92%CE%B7%E2%8B%85%E2%88%87,%E2%88%87%20%CE%B8%20J%20\(%20%CE%B8%20\)%20.](https://www.ruder.io/optimizing-gradient-descent/#:~:text=Batch%20gradient%20descent,-Vanilla%20gradient%20descent&text=%CE%B8%3D%CE%B8%E2%88%92%CE%B7%E2%8B%85%E2%88%87,%E2%88%87%20%CE%B8%20J%20(%20%CE%B8%20)%20.)

<https://www.ibm.com/topics/gradient-descent#:~:text=There%20are%20three%20types%20of,and%20mini%2Dbatch%20gradient%20descent.>

<https://towardsdatascience.com/an-introduction-to-gradient-descent-and-backpropagation-in-machine-learning-algorithms-a14727be70e9>

<https://www.analyticsvidhya.com/blog/2022/07/gradient-descent-and-its-types/>

<https://www.baeldung.com/cs/gradient-stochastic-and-mini-batch>

<https://www.v7labs.com/blog/neural-networks-activation-functions>

https://en.wikipedia.org/wiki/Activation_function

<https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141>

<https://www.simplilearn.com/tutorials/deep-learning-tutorial/multilayer-perceptron>

<https://www.baeldung.com/cs/neural-networks-backprop-vs-feedforward>

<https://towardsdatascience.com/an-introduction-to-gradient-descent-and-backpropagation-81648bdb19b2>

<https://www.techtarget.com/searchenterpriseai/definition/backpropagation-algorithm>

6 ANNEXE

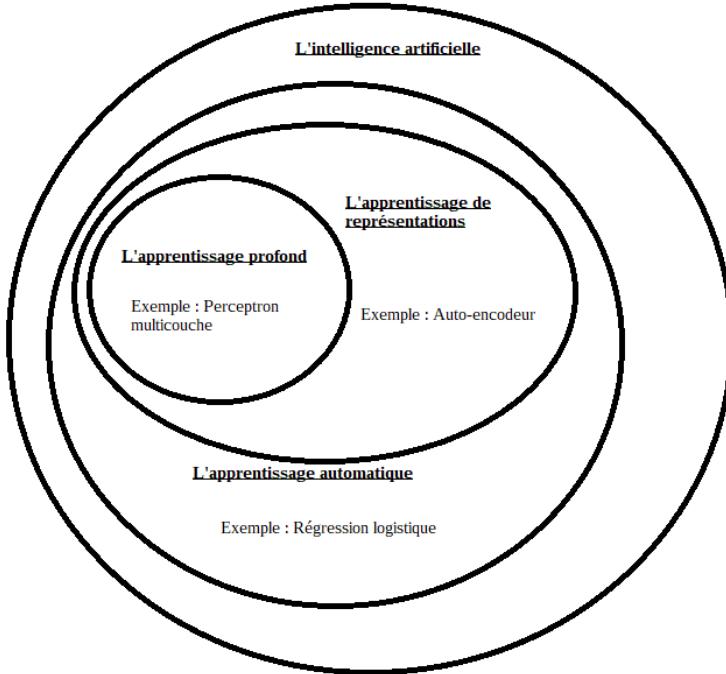


Figure 1.1 : La connexion entre ces différents domaines de l'IA est représentée dans ce diagramme de Venn. Une illustration de la technologie de l'IA peut être trouvée dans chaque partie du diagramme de Venn. (src : L'apprentissage profond (Deep Learning) - Livre d'Aaron Courville, Ian Goodfellow et Yoshua Bengio)

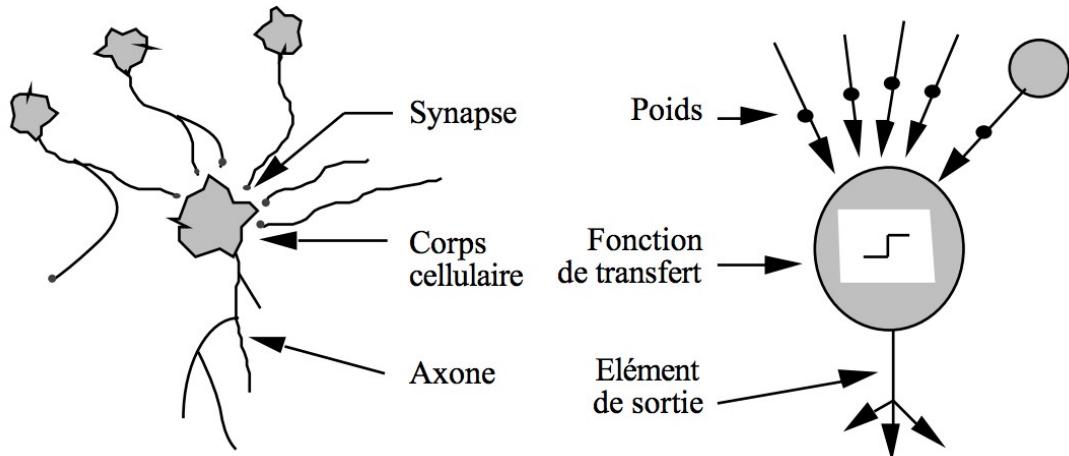


Figure 1.2 : Appariement des neurones biologiques et artificiels par [Claude Touzet](#)

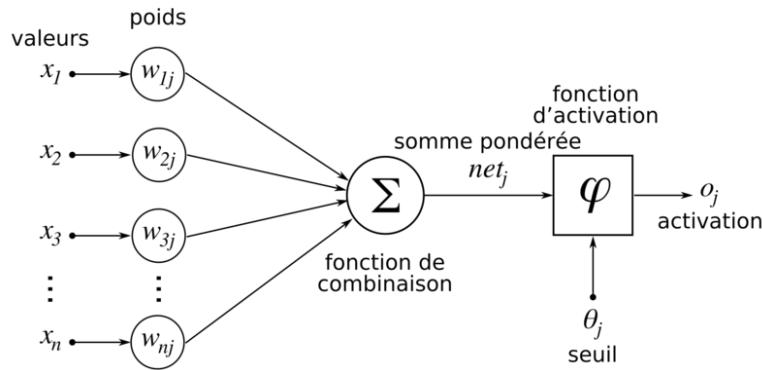


Figure 1.3 : Conception d'un neurone artificiel (src : [wikipedia](#)), dans le rapport dessus, on remplace la signe de fonction d'activation par h et l'activation \hat{y} au lieu de o_j .

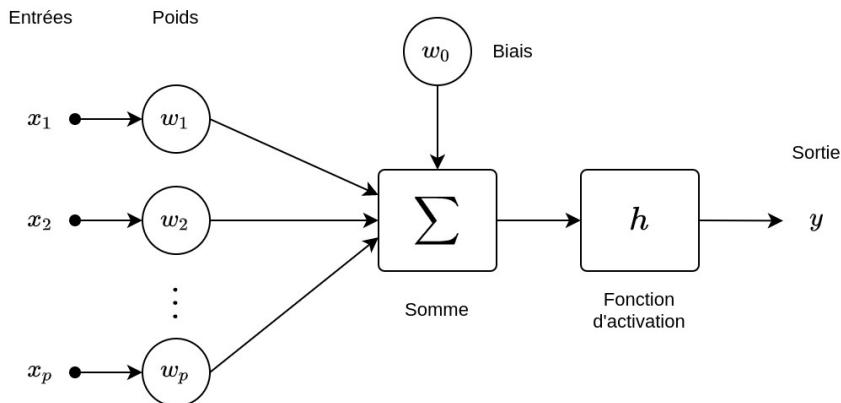


Figure 1.4 : Conception d'un neurone artificiel avec biais (src : <https://blent.ai/blog/a/reseaux-de-neurones-tout-comprendre>), dans le rapport dessus, on a remplacé la sortie y par \hat{y} .

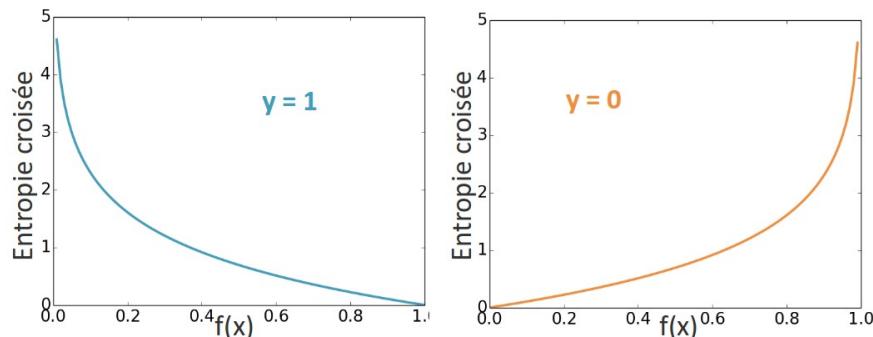


Figure 1.5 : Lorsque $y=1$, plus l'entropie croisée est grande, plus $f(x)/\hat{y}$ est proche de 0. En revanche, plus la prévision est proche de 1, plus l'entropie croisée est élevée lorsque $y=0$.

(<https://openclassrooms.com/fr/courses/4470406-utilisez-des-modeles-supervisés-non-linéaires/4730716-entraînez-un-réseau-de-neurones-simple>)

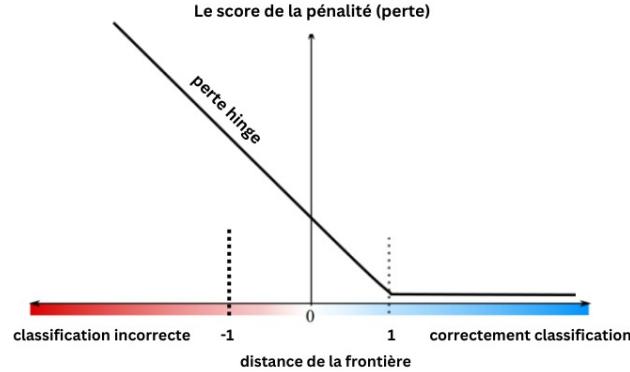


Figure 1.6 : La perte hinge, le score de perte est nul lorsque une instance est supérieure ou égale à la limite de 1.

Nous recevons un score de perte de 1 si la distance à la frontière est de 0 (c'est-à-dire que l'instance se trouve réellement sur la frontière).

Les occurrences qui ont été mal classées entraîneront un score de perte élevé tandis que les points correctement classés auront un faible (ou aucun) score de perte.

(src : <https://math.stackexchange.com/questions/782586/how-do-you-minimize-hinge-loss>)

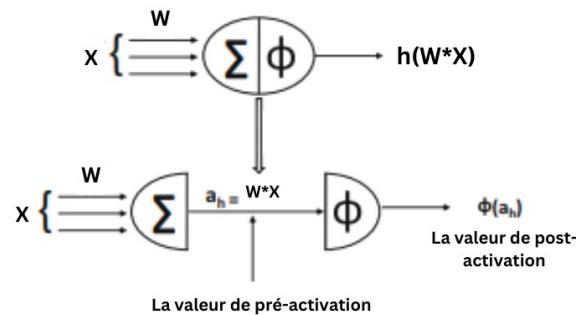


Figure 1.7 : les valeurs de pré- et post-activation d'un neurone (src : Neural Networks and Deep Learning: A Textbook - Livre du Charu C. Aggarwal)

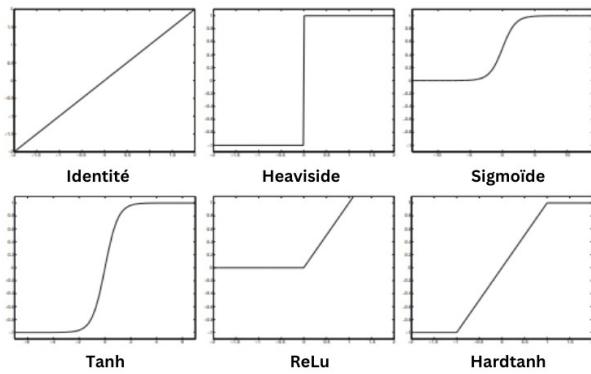


Figure 1.8 : Les différentes types de fonction d'activation (src : Neural Networks and Deep Learning: A Textbook - Livre du Charu C. Aggarwal)

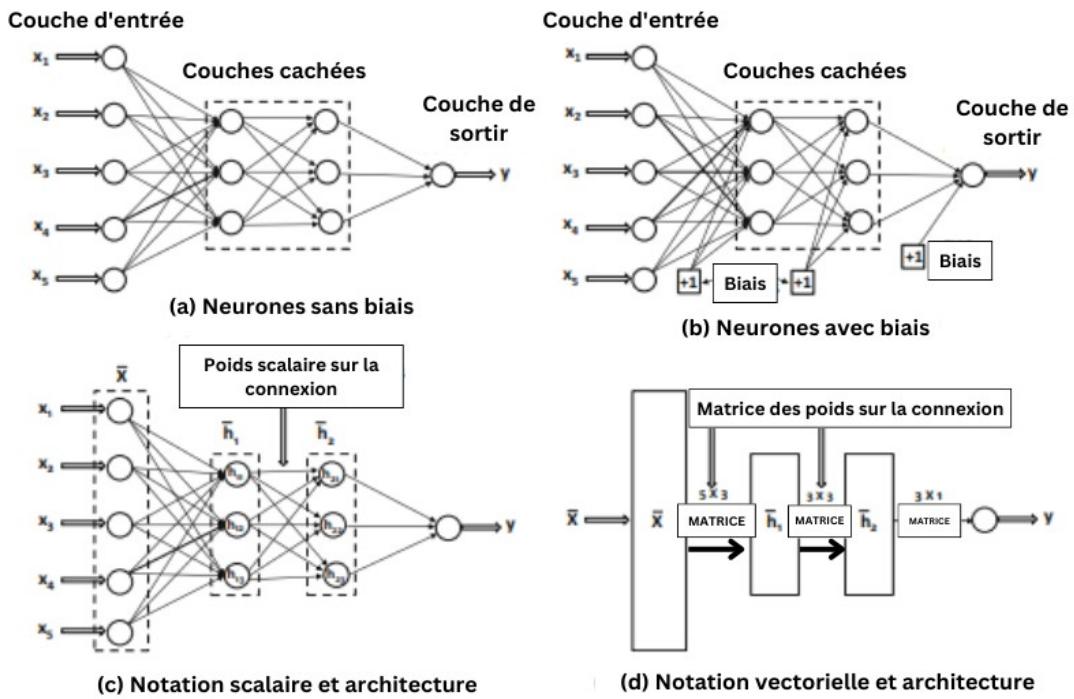


Figure 1.9 : La structure fondamentale d'un réseau feed-forward se compose d'une couche de sortie et de deux couches cachées. Bien que chaque unité ne contienne qu'une seule variable scalaire, on fait souvent référence à l'ensemble de la couche comme à une seule unité vectorielle. Les unités vectorielles ont souvent des matrices de connexion entre elles et sont représentées sous forme de rectangles. (src : Neural Networks and Deep Learning: A Textbook - Livre du Charu C. Aggarwal)

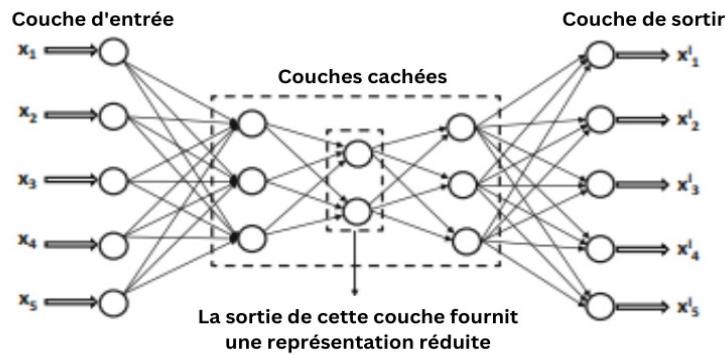
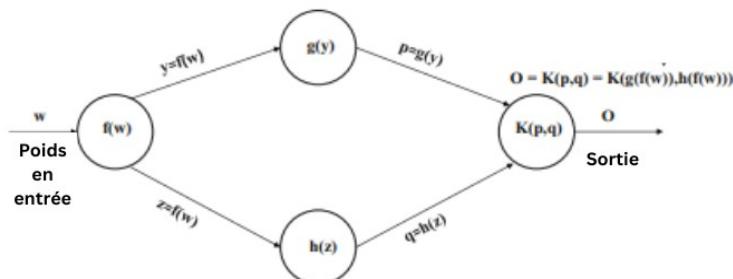


Figure 1.10 : Une illustration d'un auto-encodeur à sorties multiples.
 (src : Neural Networks and Deep Learning: A Textbook - Livre du Charu C. Aggarwal)



$$\begin{aligned}
 \frac{\partial o}{\partial w} &= \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial w} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial w} \quad [\text{La règle de la chaîne multivariable}] \\
 &= \underbrace{\frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial y} \cdot \frac{\partial y}{\partial w}}_{\text{Premier chemin}} + \underbrace{\frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial w}}_{\text{Deuxième chemin}} \quad [\text{La règle de la chaîne univariées}] \\
 &= \underbrace{\frac{\partial K(p, q)}{\partial p} \cdot g'(y) \cdot f'(w)}_{\text{Premier chemin}} + \underbrace{\frac{\partial K(p, q)}{\partial q} \cdot h'(z) \cdot f'(w)}_{\text{Deuxième chemin}}
 \end{aligned}$$

Figure 1.11 : Un exemple de la règle de la chaîne dans les graphes computationnels : Les produits de dérivées partielles le long des chemins du poids w à la sortie o sont agrégés. La valeur obtenue donne la dérivée de la sortie o par rapport au poids w . Dans cet exemple condensé, il n'y a que deux chemins possibles de l'entrée à la sortie. (src : Neural Networks and Deep Learning: A Textbook - Livre du Charu C. Aggarwal)

	Entrée			Sortie
	0	0	1	0
x_1	0	0	1	0
x_2	1	1	1	1
x_3	1	0	1	1
x_4	0	1	1	0
Nouvelle situation	1	0	0	?

Figure 1.12 : Ensemble d'entraînement pour un perceptron.

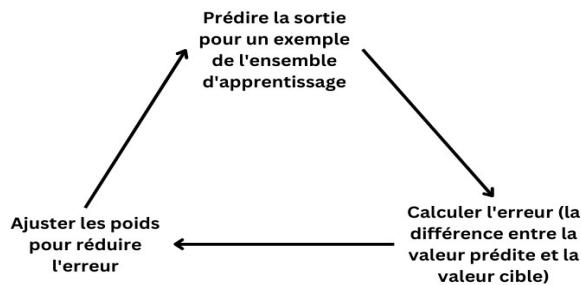


Figure 1.13 : Cycle d'entraînement du réseau de neurones.

```

PS C:\Users\mingw> & C:/Users/mingw/AppData/Local/Programs/Py
Poids synaptiques initiaux aléatoires :
[[ -0.16595599]
 [ 0.44064899]
 [ -0.99977125]]
Nouveaux poids synaptiques après l'entraînement :
[[ 9.67299303]
 [-0.2078435 ]
 [-4.62963669]]
La valeur prédictive pour la nouvelle situation [1, 0, 0] -> ?
[0.99993704]
PS C:\Users\mingw>
  
```

Figure 1.14 : Le résultat de la première application.

	Entrée			Sortie
x_1	0	0	1	0
x_2	0	1	1	1
x_3	1	0	1	1
x_4	0	1	0	1
x_5	1	0	0	1
x_6	1	1	1	0
x_7	0	0	0	0
Nouvelle situation	1	1	0	?

Figure 1.15 : Ensemble d'entraînement pour perceptron multicouche.

Entrée		Sortie
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 1.16 : la table de la porte XOR

```
PS C:\Users\mingw> & C:/Users/mingw/AppData/Local/Programs/Python/Python37-32/python.exe C:/Users/mingw/Desktop/Projet Python/Perceptron/multicouche.py
Etage 1) Poids synaptiques initiaux aléatoires :
    Couche 1 (4 neurones, chacun avec 3 entrées) :
[[ -0.16595599  0.44064899 -0.99977125 -0.39533485]
 [ -0.70648822 -0.81532281 -0.62747958 -0.30887855]
 [ -0.20646505  0.07763347 -0.16161097  0.370439 ]]
    Couche 2 (1 neurone, avec 4 entrées) :
[[ -0.5910955 ]
 [  0.75623487]
 [ -0.94522481]
 [  0.34093502]]
Etage 2) Nouveaux poids synaptiques après l'entraînement :
    Couche 1 (4 neurones, chacun avec 3 entrées) :
[[ 0.3122465   4.57704063 -6.15329916 -8.75834924]
 [ 0.19676933 -8.74975548 -6.1638187   4.40720501]
 [-0.03327074 -0.58272995  0.08319184 -0.39787635]]
    Couche 2 (1 neurone, avec 4 entrées) :
[[ -8.18850925]
 [ 10.13210706]
 [-21.33532796]
 [  9.90935111]]
Etage 3) La valeur prédite pour la nouvelle situation [1, 1, 0] -> ?:
[0.0078876]
PS C:\Users\mingw>
```

Figure 1.17 : Le résultat de la deuxième application.