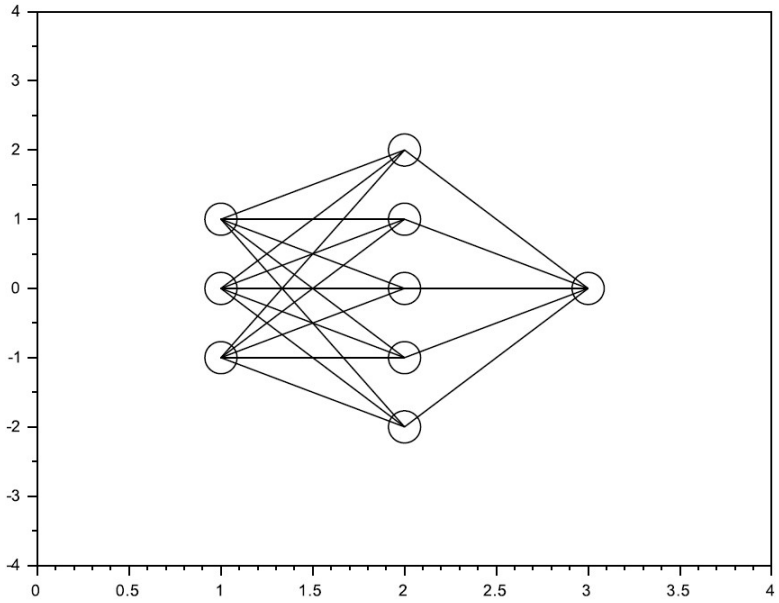
UV MT94
Cahier d'intégration:
Réseaux de Neurones Profonds

Rosalie Jardri

TC04



Semestre : P23

Remerciements

Je tiens à remercier le professeur Stéphane Mottelet de m'avoir guidée durant mes recherches ainsi que pour ses conseils et l'aide apportés lors de l'élaboration de mon cahier d'intégration, réalisé sur le semestre P23.

Je souhaite également remercier Antoine Thiameny, étudiant à l'UTC, avec qui j'ai pu échanger et discuter de mes résultats lors de l'implémentation des algorithmes sous SciLab.

Table des matières

Introduction

En l'espace de quelques années, l'intelligence artificielle est devenue omniprésente dans notre quotidien. Elle semble avoir atteint un sommet avec le dévellopement et la sortie de l'agent conversationnel ChatGPT en novembre 2022. Ce dernier constitue une véritable révolution technologique, aussi captivante qu'inquiétante. Si l'on s'interesse d'un peu plus près à son fonctionnement, ChatGPT est capable de comprendre et générer du langage naturel de façon autonome (algorithme de *Natural Language Processing*), en utilisant deux types d'apprentissage différents : (i) l'apprentissage supervisé, et (ii) l'apprentissage par renforcement. Ainsi, ChatGPT s'appuit sur l'architecture GPT (*Generative Pre-trained Transformer*), qui est un modèle de langage basé sur l'apprentissage profond (ou *Deep-Learning*). Cette architecture utilise des réseaux de neurones positionnels pour comprendre les relations entre les mots présents dans un texte. C'est parce qu'il a été entraîné au préalable sur de vastes corpus de texte, que le modèle GPT a acquis des connaissances générales sur le langage, qui lui permettent aujourd'hui de générer du texte de manière cohérente et naturelle.

Il est assez commun dans le langage courant de confondre deux des concepts principaux de la science des données : l'**apprentissage automatique**, aussi connu sous le nom de *Machine-Learning* et l'**apprentissage profond**, aussi connu sous le nom de *Deep-Learning*. En réalité, l'apprentissage automatique correspond à un sous-domaine de l'**intelligence artificielle** qui permet le développement d'algorithmes et de modèles statistiques, afin d'apprendre à des ordinateurs à prédire ou à prendre des décisions sans pour autant avoir été explicitement programmés à cet effet. En d'autres termes, il s'agit d'entraîner des algorithmes sur de grandes bases de données afin qu'ils puissent en extraire des règles ou motifs génériques traduisant certaines relations entre les données. Dans une seconde étape, ces modèles pourront être utilisés pour faire des prédictions ou prendre des décisions concernant de nouvelles données, indépendantes de celles de l'échantillon d'entraînement. L'apprentissage profond, quant à lui, est une sous-branche de l'apprentissage automatique. Comme pour ce dernier, il analyse des modèles et/ou des relations entre données, mais cette fois-ci en s'appuyant sur des réseaux de neurones à couches multiples (cf.Figure 1).

Ainsi, les modèles d'apprentissage profond sont générés à partir de grandes bases de données et sont capables de s'améliorer au fil du temps, devenant plus précis au fur et à mesure qu'ils traitent davantage de données. Cela les rend particulièrement bien adaptés aux problèmes complexes du monde réel et leur permet d'apprendre et de s'adapter à de nouvelles situations. Le Deep-Learning peut donc avoir de nombreuses applications et ne se limite pas à la recherche et au développement (R&D). Dans ce rapport, je me suis tout d'abord interessée au sujet par attrait pour la recherche et pour les réseaux de neurones, que je trouve passionnants. En faisant mes recherches, j'ai découvert que du Deep-Learning "se cachait" partout dans notre quotidien, du domaine de la maintenance préventive industrielle (afin d'anticiper des défaillances potentielles), à celui de la détection de fraudes pour les services publiques, grâce à l'analyse de l'écriture manuscrite.

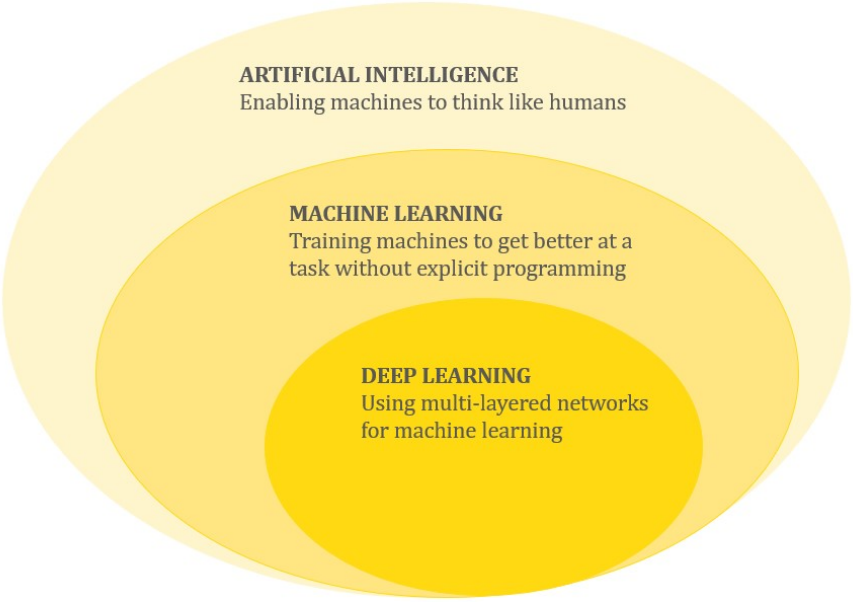


FIGURE 1 – Schéma illustrant l'imbrication des différents concepts évoqués : Intelligence Artificielle, Apprentissage Automatique et Apprentissage Profond.

Dans ce cahier d'intégration, je commencerai par étudier l'origine de l'apprentissage profond, ses concepts clefs, notamment le fonctionnement des réseaux de neurones, puis j'aborderai les différents algorithmes mathématiques utilisés en situation d'analyse de données. J'appliquerai ensuite ces concepts en implementant un neurone artificiel isolé, puis un réseau de neurones artificiels.

1 - Vision d'ensemble

Dans cette première partie, je vais tout d'abord aborder le fonctionnement d'un réseau de neurones artificiels, puis présenter les figures phares de l'histoire du Deep-Learning ainsi que leurs contributions respectives, enfin je présenterai les concepts mathématiques mis en jeu afin d'avoir une vision d'ensemble du domaine étudié dans ce rapport.

Pour rappel, l'objectif de l'apprentissage automatique est de déterminer la meilleure solution au problème posé en manipulant les paramètres d'un modèle sur la base de nombreuses données d'entraînement (c'est-à-dire trouver le modèle qui s'ajuste le mieux aux données fournies à la machine). Un modèle simple et bien connu est par exemple la fonction affine $f(x) = ax + b$.
Ici, il s'agit de minimiser les erreurs entre le modèle et les données sur la base d'un algorithme d'optimisation. Si l'on se réfère à notre exemple (cf. Figure 2), il faut tester les valeurs possibles de a et b (i.e., les paramètres du modèle), jusqu'à obtenir la combinaison minimisant la distance entre le modèle (i.e., la droite $y = ax + b$) et les données (i.e., les points).

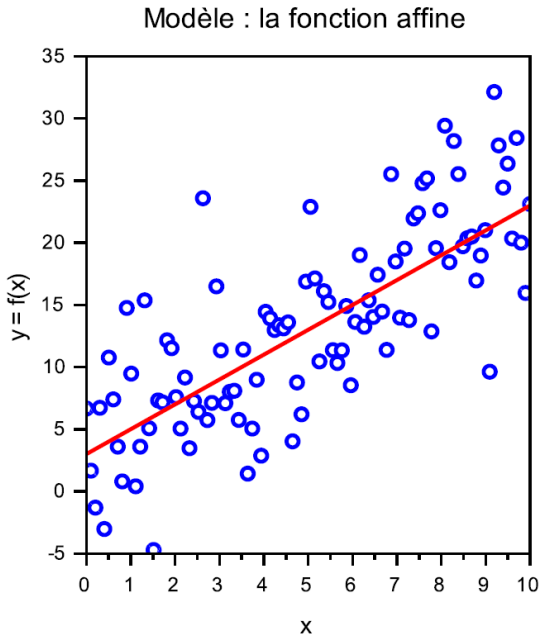


FIGURE 2 – Droite de la fonction $y = 2x + 6$ générée par Scilab (cf. Listing 1)

Il existe de nombreux modèles d'apprentissage automatique tels que les modèles linéaires, les arbres de décisions ou les machines à vecteurs de support (support-vector machine). Chacun de ces modèles possède son propre algorithme d'optimisation : (i) la *descente de gradient* pour les modèles linéaires, (ii) l'*algorithme CART* pour les arbres de décisions ou encore (iii) la *marge maximum* pour les machines à vecteurs de support.

```
//Modele choisi : droite affine
y=a*x+b
a=2; //coefficient directeur
b=3; //ordonne a l'origine

//Jeu de donnees
n=100; //nombre de points
xi=linspace(0,10,n); //valeurs xi entre 0 et 10
bruit=rand(1,n,'normal')*5;//bruit gaussien ajoute aux donnees
yi=a*xi+b+bruit;

//Representation graphique
clf
plot(xi,yi,'o',x,y,'r','thick',2)
xlabel("x")
ylabel("y = f(x)")
title("Modele : la fonction affine")
```

Listing 1 – Création du modèle (fonction affine) d'un jeu de données

J'ai succintement évoqué les modèles de Machine-Learning mais qu'en est-il du Deep-Learning ? Comme mentionné précédemment, en apprentissage profond, même si le principe général reste le même (i.e., données, modèle, optimisation), l'agorithme s'appuie sur un réseau de neurones artificiels. Le modèle n'est donc plus simplement une fonction du type $f(x) = ax+b$ mais plutôt un réseau de fonctions connectées les unes aux autres, d'où l'idée de *réseau de neurones*[1]. Mais comment ces réseaux sont-ils construits ? Et comment fonctionnent-ils ?

1.1 Les premiers neurones artificiels

Comment les réseaux de neurones ont-ils été inventés ? Quelle fut leur évolution au fil des années pour en arriver à la technologie que nous connaissons aujourd'hui ?
L'invention des premiers neurones artificiels date de 1943 par **Warren McCulloch** et **Walter Pitts**, deux mathématiciens et neuroscientifiques. Dans leur article intitulé *"A logical calculus of the ideas immanent in nervous activity"* (McCulloch and Pitts [1]), ils montrent comment ils ont pu programmer des neurones artificiels en s'inspirant du fonctionnement de neurones biologiques.

1. Il est intéressant de noter que plus ces réseaux sont profonds (c'est-à-dire plus ils contiennent de fonctions), plus la machine sera capable d'apprendre des tâches complexes. C'est pour cela que l'on parle d'apprentissage profond.

Avant d'aborder le contenu de leur article et les concepts y figurant, commençons par quelques rappels de neurophysiologie. Les neurones sont des cellules excitables connectées les unes aux autres, ayant pour rôle de transmettre de l'information dans notre système nerveux. Chaque neurone est composé respectivement : (i) de plusieurs dendrites, (ii) d'un corps cellulaire, et (iii) d'un axone (cf. Figure 3). Les dendrites peuvent être vues comme les portes d'entrées du neurone : ils reçoivent des signaux (excitateurs ou inhibiteurs) d'autres neurones le précédant dans la chaine d'information. Quand la somme de ces signaux dépasse un certain seuil, le neurone s'"active" et produit à son tour un signal éléctrique (e.g., le potentiel d'action) qui circulera le long de l'axone vers ses terminaisons pour être envoyé au neurone suivant.

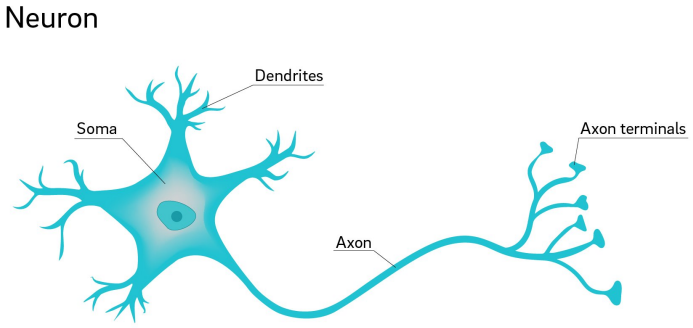


FIGURE 3 – Schéma simplifié d'un neurone et de ses composants

Dans leur article, McCulloch et Pitts ont donc essayé de modéliser ce fonctionnement en considérant qu'un neurone pouvait être représenté par une fonction de transfert, prenant des signaux x en entrée et retournant une sortie y.
Il existe donc deux grandes étapes à implémenter. La première est **une étape d'aggrégation**. Comme le montre l'équation 1, toutes les entrées du neurones sont multipliées par un coefficient w (représentant l'activité synaptique) et sommées :

$$f = w_1x_1 + w_2x_2 + w_3x_3 \quad avec \quad w = \begin{cases} +1 & \text{si le signal est excitateur} \\ -1 & \text{si le signal est inhibiteur} \end{cases} \tag{1}$$

La deuxième étape est **une étape d'activation**. Lorsque f dépasse un certain seuil (généralement 0), alors le neurone s'active et retourne une sortie y (Equation 2) :

$$y = \begin{cases} 1 & \text{si } f \geq 0 \\ 0 & \text{sinon} \end{cases} \tag{2}$$

En s'appuyant sur ces deux étapes, McCulloch et Pitts ont développé les premiers réseaux de neurones artificiels, plus tard renommés **threshold logic units**[2]. Ils ont d'ailleurs été capable de démontrer, avec ce modèle, qu'il était possible de reproduire certaines fonctions logiques, telles que la porte AND et la porte OR (cf. Listing 2). Ils ont également démontrés qu'en connectant plusieurs de ces fonctions les unes aux autres (à l'instar des neurones de notre cerveau), il était possible de résoudre n'importe quel problème de logique booléenne.

```
//Fonction de reproduction de la porte AND
function y = AND(x)
    w = [1, 1];// Poids des entrees
    b = -1.5;// Biais
    f = sum(w .* x) + b; //Calcul du potentiel d'activation
    if f >= 0
        y = 1;//Activation
    else
        y = 0;//Inhibition
    end
end

//Test de la porte logique AND
x1 = 0;
x2 = 0;
y = AND([x1, x2]);
disp(y);//Affiche 0

x1 = 0;
x2 = 1;
y = AND([x1, x2]);
disp(y);//Affiche 0

x1 = 1;
x2 = 0;
y = AND([x1, x2]);
disp(y);//Affiche 0

x1 = 1;
x2 = 1;
y = AND([x1, x2]);
disp(y);//Affiche 1
```

Listing 2 – Reproduction de la porte AND grâce au modèle de McCulloch et Pitts

A l'image de la sortie de ChatGPT, cette découverte de Warren McCulloch et Walter Pitts provoqua un engouement pour le domaine de l'intelligence artificielle. Cependant, même si ce modèle est à l'origine du Deep-Learning, il possède un certain nombre de

2. Ce nom vient du fait que ce modèle n'était à l'origine conçut que pour traiter des entrées logiques, valant soit 0, soit 1.

failles. En effet, il ne dispose pas de fonction d'apprentissage : il convient à l'utilisateur de trouver par lui même les valeurs des poids w, pour ensuite pouvoir les utiliser (nous avons par exemple fixé nous même w_1 et w_2 à 1, les poids des entrées de la porte AND lors de l'exemple vu précédemment).

1.2 L'invention du perceptron

Il faut attendre 1957, pour que **Frank Rosenblatt**, un psychologue américain, invente le perceptron (cf. Figure 4). En effet, ce chercheur parvient à améliorer le modèle de McCulloch et Pitts, en proposant **un algorithme d'apprentissage**. En vérité, il s'agira du tout premier algorithme d'apprentissage de l'histoire du Deep-Learning.
Le modèle de perceptron qu'il propose ressemble de très près au modèle étudié dans la partie 1.1 du rapport. Il s'agit donc d'un neurone artificiel, s'activant quand la somme pondérée de ses entrées dépasse un certain seuil. Cependant, l'algorithme d'apprentissage propre au perceptron lui permet de trouver par lui-même les valeurs de w, permettant ensuite d'obtenir les bonnes valeurs de y.

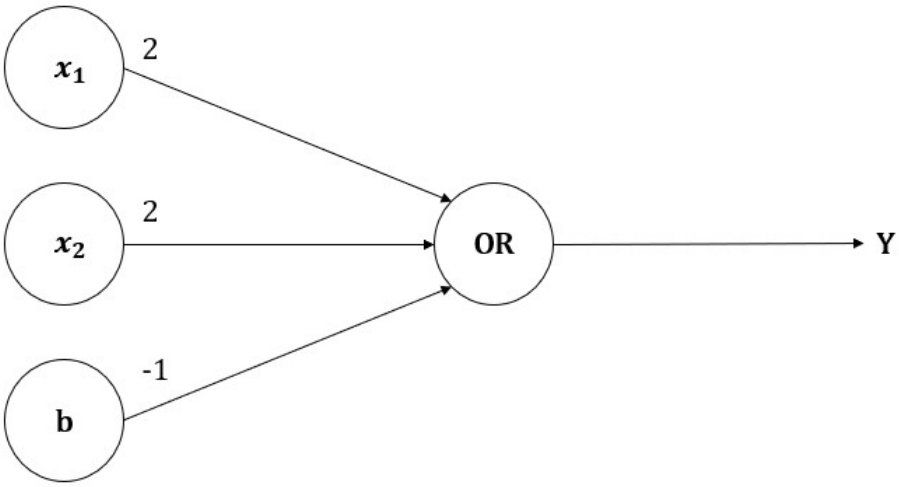


FIGURE 4 – Schéma d'un perceptron simple (à une couche) : exemple de la porte OR. Ici, le modèle utilisé est le suivant : $f = 2x_1 + 2x_2 - 1$.

Pour développer cet algorithme, Frank Rosenblatt s'est inspiré de la théorie de **Donald Hebb**, exposée dans son ouvrage de 1949 *"The Organization of Behavior : A Neuropsychological Theory"* (Hebb [2]). Ce dernier propose que lorsque deux neurones sont activés simultanément et de manière répétée, la connexion synaptique existant entre eux soit alors renforcée, facilitant par la suite la transmission de l'information entre ces neurones. Ce principe, appelé *plasticité synaptique* en neurosciences, est celui qui nous permet d'apprendre et de mémoriser.

Frank Rosenblatt a donc développé un algorithme d'apprentissage, à partir de ce que l'on appelle communément la "loi de Hebb". Son algorithme consiste à entrainer un neurone artificiel sur des données dites de références (X, y) pour que celui-ci renforce ses paramètres W à chaque fois qu'une entrée X est activée (en même temps que la sortie y présente dans ces données).Il a ainsi imaginé la formule suivante (Equation 3) :

$$W = W + \alpha(y_{true} - y)X \tag{3}$$

Les paramètres W sont ici mis à jour en calculant la différence entre la sortie dite de référence y_{true} et la sortie produite par le neurone notée y, le tout multiplié par la valeur de chaque entrée X et par un pas d'aprentissage positif α.

Plus précisement, tant que le neurone artificiel fournit une sortie différente de celle attendue, le coefficient W augmentera d'un petit α (Equation 4). En d'autres termes, W sera "renforcé", provoquant ainsi une augmentation de la fonction f (équation 1), rapprochant le neurone artificiel de son seuil d'activation. Le coefficient W continuera d'augmenter jusqu'à ce que $y_{true} = y$. Les paramètres cesseront alors d'évoluer (cf. Figure 5).

$$W = W + \alpha(y_{true} - y)X = \begin{cases} W + \alpha(1-0)X = W + X & \text{si } y = 0 \neq y_{true} = 1 \\ W + \alpha(1-1)X = W & \text{si } y = 1 = y_{true} \end{cases} \tag{4}$$

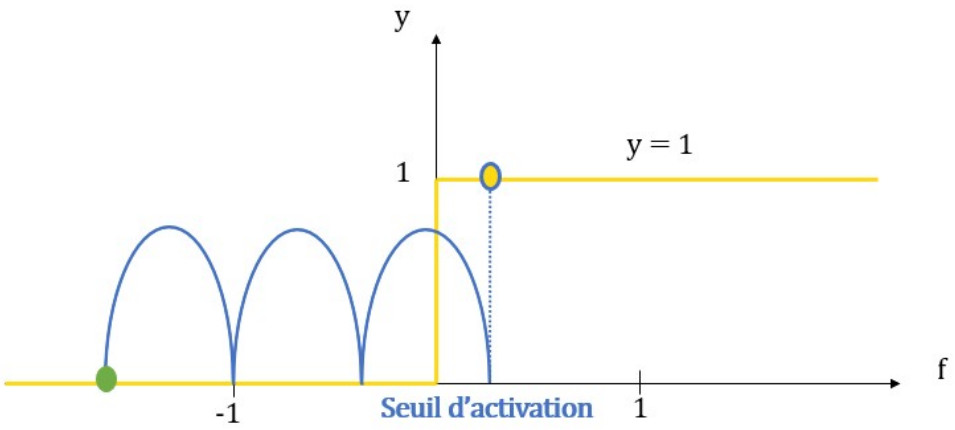


FIGURE 5 – Courbe représentant l'évolution y jusqu'à atteindre le seuil d'activation situé à y_{true} (fonction Heavy Side)

Comme pour la découverte de McCulloch et Pitts, cet algorithme d'apprentissage par perceptron provoqua un emballement dans le domaine de l'intelligence artificielle. Il

possède cependant, lui aussi, un certain nombre de limites. En effet, le perceptron est un modèle linéaire, c'est-à-dire que la représentation graphique de sa fonction d'aggrégation $f(x_1, x_2) = w_1x_1 + w_2x_2 + b$ est une droite dont l'inclinaison dépend des poids w. Le biais, noté b est un paramètre complémentaire permettant de modifier la position de la droite. Même si cette droite est capable de séparer deux classes de points, ce type de situation n'est pas très représentatif du monde réel : la majorité des phénomènes de notre univers ne sont pas linéaires et binaires. A lui seul, le perceptron n'est donc pas très utile dans ce type de situation.

1.3 L'invention du perceptron multicouches

Il fallut attendre 1986 pour que **Geoffrey Hinton**, un des pères de l'apprentissage profond, développe le perceptron multicouches, qui peut être considéré comme le premier véritable réseau de neurones artificiels (Minsky and Papert [3]).

En réalité, Geoffrey Hinton a utilisé une des idées évoquées par Warren McCulloch et Walter Pitts dans leur article et cité ci-après : "il est possible de résoudre des problèmes plus complexes en connectant plusieurs neurones ensemble". Ainsi, en connectant 3 perceptrons, les deux premiers vont chacun recevoir des entrées x_1 et x_2, faire des calculs en fonction de leurs paramètres et retourner une sortie y, qui pourra à son tour être envoyée vers un troisième perceptron, effectuant également des calculs avant d'obtenir un résultat final (cf. Figure 6). Cet exemple contenant trois neurones répartis en deux couches, une couche d'entrée et une couche de sortie, est appelé un **perceptron multicouches**. Le nombre de couches et de neurones ajoutés n'est pas limité d'un point de vue théorique et plus ils seront nombreux, plus le résultat sera complexe.

La représentation graphique de la sortie finale en fonction des entrées x_1 et x_2 n'est plus un modèle linéaire, mais un **modèle non-linéaire**. Parmi les modèles non-linéaires fréquents, nous pouvons citer la fonction tangeante hyperbolique : $f(x) = tanh(x)$ (cf. Figure 7 et Listing 3) [3].

```
// Generation de donnees
x = linspace(-5, 5, 100);
y = tanh(x);

// Affichage graphique
clf
plot(x, y, 'r');
xlabel("x");
```

3. Pour les réseaux de neurones profonds avec des couches cachées, les fonctions d'activation non-linéaires comme Tanh sont souvent utilisées pour approximer des relations non-linéaires complexes

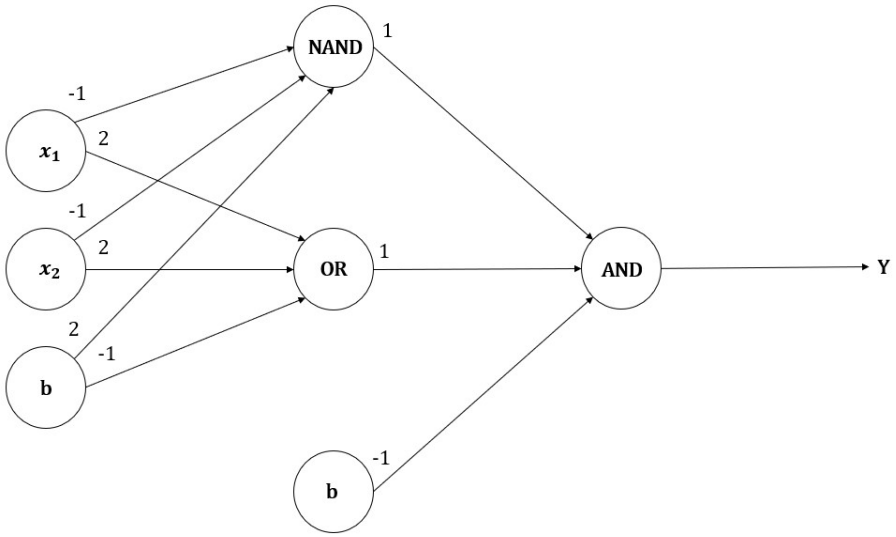


FIGURE 6 – Schéma d'un perceptron à deux couches : exemple de la fonction XOR. Ici, nous utilisons le modèle suivant : $f_{OR} = 2x_1 + 2x_2 - 1$, $f_{NAND} = -x_1 - x_2 + 2$ et $f_{AND} = x_1 + x_2 - 1$. Voir aussi (Dutta [4])

```
9  ylabel("y = tanh(x)");
10 title("Modele non-lineaire : la fonction tangeante
      hyperbolique");
```

Listing 3 – Représentation graphique de la fonction tanh, modèle non-linéaire célèbre

La solution proposée pour trouver les valeurs de paramètres w et b d'un perceptron multicouches (et tendre vers un modèle optimal) est une technique appelée **rétropropagation**. Cette dernière consiste à determiner comment la sortie du réseau varie en fonction des paramètres (W, b) présents dans chaque couche du modèle.

Il faut donc calculer une **chaîne de gradients**, indiquant comment la sortie varie en fonction de la dernière couche, alors notée $\frac{\partial f_5}{\partial f_4}$ (en supposant ici que nous ayons 5 couches), puis comment la dernière couche varie en fonction de l'avant-dernière $\frac{\partial f_4}{\partial f_3}$, puis comment l'avant-dernière varie en fonction de l'avant-avant dernière, etc. Nous répétons ce processus jusqu'à arriver à la première couche de notre réseau $\frac{\partial f_2}{\partial f_1}$. Voilà pourquoi nous parlons de "propagation arrière".

Les gradients obtenus permettent de mettre à jour les paramètres (poids et biais pour chaque couche) (W, b), avec pour objectif final de minimiser l'erreur entre la sortie du modèle et la sortie attendue. La formule utilisée ici, appelée **la formule de descente de gradient** (Equation 5), n'est pas très différente de celle proposée par Frank Rosenblatt (Equation 3) :

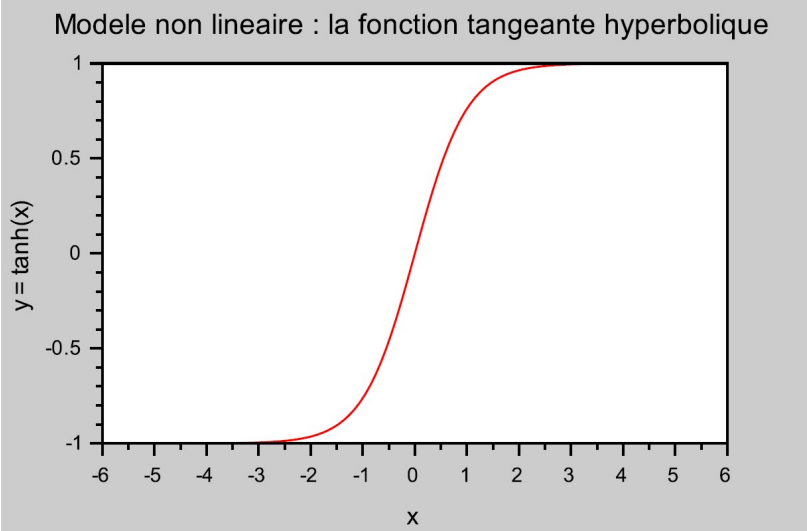


FIGURE 7 – Représentation de la fonction tangeante hyperbolique

$$W = W - \alpha(\frac{\partial Erreur}{\partial W}) \tag{5}$$

Cette équation, ainsi que les autres présentées dans cette partie seront étudiées plus en détail dans la partie 2.

Pour résumer, afin de construire un réseau de neurones, il faut répéter en boucle quatre étapes (détaillées dans le Tableau 1).

TABLEAU 1 – Les quatre grandes étapes du développement d'un réseau de neurones

Etape	Concept mathématique associé	Explication
1	Forward Propagation (ou propagation ascendante)	Circulation des données de la première à la dernière couche pour obtenir une sortie y
2	Cost function (ou fonction coût)	Calcul grâce à la fonction, de l'erreur entre la sortie actuelle et la sortie référence y_{TRUE}
3	Backward propagation (ou rétro-propagation)	Mesure des variations de la fonction coût par rapport à chacune des couches du modèle (en partant de la dernière et en remontant jusqu'à la toute première)
4	Gradient Descent (ou algorithme de descente de gardient)	Correction des paramètres du modèle grâce à l'algorithme (avant de reboucler vers la 1ère étape pour recommencer un cycle d'entraînement)

1.4 Le Deep-Learning moderne

Le modèle du perceptron multicouches a bien sûr continué d'évoluer au fil du temps, notamment avec l'apparition de nouvelles fonctions d'activations comme la *fonction logistique* (ou sigmoïde) étudiée plus tard dans la partie 7, la *fonction tangeante hyperbolique*, déjà évoquée plus haut, ou encore la fonction Rectified Linear Unit ou ReLU. Ces fonctions ont aujourd'hui totalement remplacé la **fonction Heavy Side**, considérée jusqu'à présent dans ce rapport, car elles offrent de bien meilleures performances.

Les premières variantes de perceptron multi-couches se développèrent dans les années 80. Le célèbre chercheur français Yann LeCun (Lauréat du prix Turing 2018, actuellement chez Méta) inventa les premiers **réseaux de neurones convolutifs** (réseaux capables de reconnaître et traiter des images) ou LeNET en 1990, en introduisant au début, de ces réseaux, des filtres mathématiques appelé "convolution & pooling". En 1997, les premiers **réseaux de neurones récurrents** ou LSTMs apparaissent. Ces derniers sont aussi une variante du perceptron multicouches et permettent de traiter efficacement les problèmes de séries temporelles, comme la lecture de texte ou encore la reconnaissance vocale.

Mais l'apprentissage profond ne prend réellement son envol qu'en 2012 lors d'une compétition de vision par ordinateur *ImageNet*, où une équipe de chercheurs menés par Geoffrey Hinton, réussit à développer un réseau de neurones capable de reconnaître n'importe quelle image avec une meilleure performance que tous les autres algorithmes disponibles à l'époque.

⋆ ⋆ ⋆

2 - Concepts mathématiques sous-jacents

2.1 Le cas du perceptron simple

Dans cette première sous-section, nous aborderons les concepts mathématiques sur lesquels s'appuie le perceptron simple. Le modèle du perceptron utilisé sera le suivant :

$$z(x_1, x_2) = w_1 x_1 + w_2 x_2 + b \tag{6}$$

avec w_1 et w_2 les poids associés respectivement aux entrées x_1 et x_2 et b, le biais.
Notre modèle est donc un modèle de classification binaire, capable de séparer linéairement deux classes de points. La droite générée par ce perceptron est appelée **frontière de décision** ($z = 0$).

2.1.1 Fonction sigmoïde ou logistique

Afin d'améliorer ce modèle, nous pouvons essayer de lier chaque prédiction à **une probabilité**. *"Plus un point sera éloigné de la frontière de décision, plus il sera évident ou probable que ce point appartienne à l'une ou l'autre classe."* Pour cela, nous pouvons utiliser une fonction d'activation (déjà évoquée dans la section 3) : plus la sortie retournée par cette dernière fonction sera proche de 0 ou de 1, plus elle s'éloignera de la frontière de décision ($z = 0$).

$$a(z) = \frac{1}{1 + e^{-z}} \tag{7}$$

La fonction sigmoïde (Figure 8 et Listing 4), également appelée fonction logistique permet de réaliser cette opération. Cette fonction (Equation 7) convertit la sortie z en une probabilité $a(Z)$, i.e., la probabilité qu'un élément appartienne à une classe en particulier.

```
function a = sigmoid(z)
    a = 1 ./ (1 + exp(-z));
endfunction

z = linspace(-10, 10, 100);
a = sigmoid(z);
clf
plot(z, a, 'r');
xlabel('z');
ylabel('a(z)');
title('Fonction sigmoide');
```

Listing 4 – Représentation graphique de la fonction tanh, modèle non-linéaire célèbre

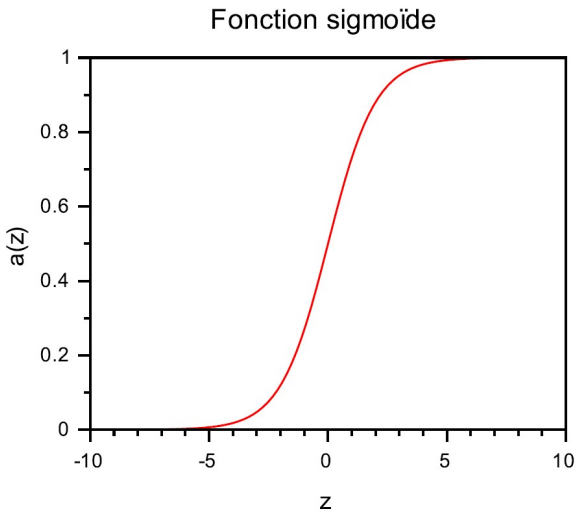


FIGURE 8 – Représentation de la fonction sigmoïde

Prenons un exemple :

Si un élément E, situé à droite de la frontière de décision, a en sortie $z = 5$, alors la probabilité $a(z)$ qu'il appartienne à la classe 1 est de $0,99$. En d'autres termes, il a 99% de chance d'appartenir à la catégorie 1 (probabilité élevée mais logique puisque E se trouve à droite de la frontière).

En réalité, ces valeurs suivent une loi de probabilité. Si nous revenons à notre exemple, la probabilité que l'élément E appartienne à la classe 1 est donnée par $a(z)$ et la probabilité qu'il appartienne à la classe 2 est donnée par $1 - a(z)$, ce qui peut être condensé via la **loi de Bernouilli** (Equation 8).

$$P(Y = y) = a(z)^y \times (1 - a(z))^{1-y} \tag{8}$$

Chaque neurone du perceptron utilise donc une fonction linéaire, suivie d'une fonction d'activation (ici la fonction sigmoïde), qui retourne une probabilité suivant la loi de Bernouilli. A ce stade, il faut encore définir une fonction coût permettant de mesurer les erreurs entre la valeur obtenue et la valeur attendue.

2.1.2 Fonction coût et maximum de vraisemblance

Dans le domaine de l'aprentissage automatique, une fonction coût (aussi connue sous le nom de *loss function* ou *cost function* en anglais) est une fonction, permettant de quantifier les erreurs effectives d'un modèle. Dans notre cas, il s'agit donc d'une fonction mesurant les distances (ou erreurs) entre les sorties $a(z)$ et les données y dont nous disposons. La fonction coût utilisée dans notre cas est appelée Log Loss (Equation 9).

$$\mathcal{L} = \frac{-1}{m} \sum_{i=0}^{m} y_i log(a_i) + (1 - y_i) log(1 - a_i) \tag{9}$$

Nous pouvons nous intéresser plus précisement à l'origine mathématique de cette fonction. En effet, une facon de calculer la performance du modèle est de calculer sa vraisemblance. En statistique, la **vraisemblance** indique la plausibilité d'un modèle vis à vis de données considérées comme vraies (i.e., données de référence), c'est-à-dire qu'elle permet de savoir si les prédictions du modèle sont en accord avec les données. Pour calculer la vraisemblance de notre modèle, il faut faire le produit de toutes ces probabilités en utilisant la loi de Bernouilli vue précédemment (Equation 10) [4].

$$L = \prod_{i=1}^{m} P(Y = y_i) = \prod_{i=1}^{m} a_i^{y_i} \times (1 - a_i)^{1-y_i} \tag{10}$$

Plus le résultat de ce calcul est proche de 100%, plus le modèle est vraisemblable (i.e., en accord parfait avec les données considérées comme vraies). A l'inverse, plus le résultat est proche de 0%, moins notre modèle est plausible. Il aura une chance d'exister mais si c'est le cas, cela signifierait que les données de référence dont nous disposons sont probablement fausses.

Cependant, un problème persiste avec l'utilisation du concept de vraisemblance puisque nous effectuons un produit de probabilités (c'est-à-dire un produit de nombres compris entre 0 et 1). Plus il y a de valeurs, plus le résultat va tendre vers 0. Ainsi dans la pratique lors du calcul de la vraisemblance sur des dizaines de milliers de points, le résultat risque d'être proche de 0 et la mémoire de l'ordinateur ne pourra stocker ce nombre. De plus si la valeur est égale à 0, le produit sera nul [5].

En réalité, il existe une "astuce" pour calculer cette vraisemblance sans pour autant converger vers 0 : utiliser une tranformation logarithmique [6]. En effet, l'une des règles dite de "base" des logarithmes est : $log(ab) = log(a) + log(b)$ [7]. Il suffit donc ici d'appliquer un logarithme sur L (Equation 11). Le calcul n'est alors plus un produit de nombres compris entre 0 et 1, mais devient une addition. De plus, avec la transformation logarithmique, les valeurs passent d'un espace [0,1] à un espace $]-\infty,+\infty[$.

4. Le L provient de *Likelihood*, signifiant vraisemblance en anglais.
5. Ce problème sera réglé par la transformation logarithmique qui tend vers l'infini.
6. La fonction log ne déformera pas nos résultats, puisqu'elle est monotone croissante. Ainsi, elle conservera l'ordre des termes et il suffira de chercher le maximum du log de la vraisemblance, pour trouver le maximum de cette dernière. Il est important de noter qu'il s'agit ici du logarithme népérien.
7. Une autre de ces règles est utilisée plus tard dans la démonstration : $log(a^y) = ylog(a)$

$$
\begin{aligned}
log(L) &= log\left(\prod_{i=1}^{m} a_i^{y_i} \times (1-a_i)^{1-y_i}\right) \\
&= \sum_{i=1}^{m} log\left(a_i^{y_i} \times (1-a_i)^{1-y_i}\right) \\
&= \sum_{i=1}^{m} log(a_i^{y_i}) + log((1-a_i)^{1-y_i})) \\
&= \sum_{i=1}^{m} y_i log(a_i) + (1-y_i) log(1-a_i)
\end{aligned}
\tag{11}
$$

La fonction obtenue ressemble alors à la fonction coût (Equation 9). Le seul élément manquant est le facteur $\frac{-1}{m}$ avant la somme. Ceci peut s'expliquer de plusieurs manières.

Tout d'abord, notre calcul consistait jusqu'ici à trouver le logarithme de la vraisemblance. L'objectif était de maximiser la vraisemblance afin d'obtenir le meilleur modèle possible. En mathématiques, les algorithmes de minimisation sont bien plus courants que ceux de maximisation. Pour maximiser, nous pouvons donc minimiser la fonction $-f(x)$, ce qui explique la présence d'un facteur -1 devant la somme (Equation 9). En outre, la présence du facteur $\frac{1}{m}$ peut s'expliquer par une volonté de normaliser le résultat. En résumé, il est donc possible d'utiliser la fonction coût (Equation 9) pour minimiser les erreurs du modèle.

2.1.3 Algorithme de descente de gradient

L'algorithme de **descente de gradient** est l'un des algorithmes d'optimisation les plus utilisés en apprentissage automatique et/ou profond. Il consiste à ajuster les paramètres W et b de façon à minimiser les erreurs du modèle, c'est-à-dire à **minimiser la fonction coût** (cf. Partie 7). Pour cela, il convient de déterminer la manière dont cette fonction varie en fonction des différents paramètres. Plus simplement, l'enjeu est de savoir si la fonction diminue lorsque W augmente (ou inversement).

Afin de répondre à cette question, il faut calculer le **gradient** (ou dérivée) de la fonction coût $\frac{\partial \mathcal{L}}{\partial W}$, puisqu'en mathématiques, la dérivée d'une fonction nous informe sur ses variations.
Ainsi, apppliqué à notre cas, une dérivée négative signifiera que la fonction diminue quand W augmente. Il faudra alors augmenter W pour réduire nos erreurs. A l'inverse si la dérivée est positive, cela indiquera que la fonction coût augmente quand W augmente et qu'il faudra diminuer W pour réduire nos erreurs.

Il faut donc ici utiliser la formule de descente de gradient (Equation 12), avec le paramètre W à l'instant $t+1$ et t, un pas d'apprentissage positif α et le gradient à

l'instant t (dérivée partielle de la fonction).

$$W_{t+1} = W_t - \alpha \frac{\partial \mathcal{L}}{\partial W_t} \tag{12}$$

Comme expliqué plus haut, un gradient négatif fera augmenter W. Le résultat obtenu sera de l'ordre de $W_{t+1} = W_t + P$, avec P positif. A l'inverse, si le gradient est positif, alors W va diminuer. En répétant cette formule de manière itérative, nous allons descendre progressivement le long de la courbe de la fonction coût et atteindre le minimum de la fonction [8].

Il est important de noter qu'il existe une condition nécessaire au bon fonctionnement de l'algorithme : la fonction coût doit être **convexe**, c'est-à-dire qu'il n'existe **pas de minimum local** sur lequel l'algorithme pourrait converger. Comme Logloss est une fonction convexe, nous pouvons bien utiliser l'algorithme de descente de gradient pour entraîner des neurones artificiels.

2.1.4 Calcul des gradients d'un neurone

Nous disposons à ce stade de presque toutes les formules nécessaires pour implémenter un réseau de neurones dans Scilab. Il ne manque plus que le gradient. Je vais donc détailler ci-dessous le calcul des différentes dérivées partielles nécessaires : $\frac{\partial \mathcal{L}}{\partial W_1}$, $\frac{\partial \mathcal{L}}{\partial W_2}$ et $\frac{\partial \mathcal{L}}{\partial b}$.

Commencons par $\frac{\partial \mathcal{L}}{\partial W_1}$. Nous pouvons remarquer que W_1 n'apparait pas dans $\mathcal{L}$. Il faudrait alors exprimer $\mathcal{L}$ avec $a(z)$ (Equation 7) et z (Equation 6), ce qui compliquerait énormément le calcul. En m'appuyant sur **la règle des chaînes** (Equation 13), je peux décomposer le calcul afin qu'il ne reste que trois termes, plus facilement dérivables :

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial a} \times \frac{\partial a}{\partial z} \times \frac{\partial z}{\partial W_1} \tag{13}$$

Commençons par la **première composante** (Equation 14), trouvée grâce aux dérivées suivantes $(log(a))' = \frac{1}{a}$ et $(log(1-a))' = -\frac{1}{1-a}$. Ici, le terme $\frac{-1}{m}$ et la somme $\sum_{i=1}^{m}$ ne changent pas puisque ce sont des facteurs (et ils ne dépendent pas de a).

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{-1}{m} \sum_{i=1}^{m} \left(\frac{y}{a} - \frac{1-y}{1-a} \right) \tag{14}$$

Etudions maintenant la **deuxième composante** (Equation 15). Afin de simplifier le calcul de la dérivée de a, j'utilise la fonction composée en posant : $a = g \circ f = g(f(z))$ avec $g(x) = \frac{1}{x}$ et $f(z) = 1 + exp(-z)$.

8. Le terme *Descente de Gradient* vient d'ailleurs de cette opération.

$$\frac{\partial a}{\partial z} = g'(f(z)) \times f'(z) = \frac{-1}{(1+exp(-z))^2} \times -exp(-z) = \frac{exp(-z)}{(1+exp(-z))^2} \tag{15}$$

Cependant, ce calcul peut encore être simplifié (Equation 16). En effet, je peux décomposer le résultat pour faire apparaître la fonction sigmoïde $a(z)$ en utilisant une astuce (ajoute $(+1-1)$ au numérateur).

$$\begin{aligned} \frac{\partial a}{\partial z} &= \frac{1}{1+exp(-z)} \times \frac{exp(-z)}{1+exp(-z)} = a \times \frac{exp(-z)+1-1}{1+exp(-z)} \\ &= a \times \left(\frac{exp(-z)+1}{1+exp(-z)} + \frac{-1}{1+exp(-z)} \right) = a \times (1-a) \end{aligned} \tag{16}$$

Etudions maintenant la **troisième composante** (Equation 17). Il ne reste que le facteur en face de w_1.

$$\frac{\partial z}{\partial w_1} = x_1 \tag{17}$$

Je peux désormais effectuer le calcul final de $\frac{\partial \mathcal{L}}{\partial W_1}$, en multipliant les trois expressions trouvées :

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W_1} &= \frac{-1}{m} \sum_{i=1}^{m} \left(\frac{y}{a} - \frac{1-y}{1-a} \right) \times a \times (1-a) \times x_1 \\ &= \frac{-1}{m} \sum_{i=1}^{m} (y(1-a) - (1-y)a) \times x_1 \\ &= \frac{-1}{m} \sum_{i=1}^{m} (y-a) \times x_1 \end{aligned} \tag{18}$$

Pour calculer les autres gradients, il faut également utiliser la règle des chaînes : seule la dernière composante change. Pour finir, le résultat obtenue est le suivant :

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} (a-y)\, x_1 \\ \frac{\partial \mathcal{L}}{\partial w_2} = \frac{1}{m} \sum_{i=1}^{m} (a-y)\, x_2 \\ \frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a-y) \end{cases} \tag{19}$$

2.1.5 Vectorisation des équations

Une **vectorisation** consiste à organiser les données sous la forme de vecteurs, matrices ou tableaux à N-dimensions, afin d'effectuer des opérations mathématiques sur l'ensemble des données. En programmation, la vectorisation permet d'obtenir un code plus simple et plus rapide à exécuter (e.g., une seule opération sur un vecteur versus une opération par éléments de la liste, nécessitant une boucle itérative for).

La vectorisation apparait indispensable pour réaliser de l'apprentissage automatique et/ou profond, en particulier lorsque de très grandes quantités de données doivent être manipulées. Au lieu de faire passer les données les unes après les autres dans le réseau de neurones, répétant ainsi en boucle les mêmes calculs des milliers de fois, nous pouvons vectoriser les équations du modèle afin de pouvoir traiter toutes les données en même temps. L'économie en temps de calcul est considérable. Notre but est donc de **ré-écrire au format matriciel l'ensemble des équations** vues précédemment.

Pour rappel, les matrices sont des tableaux à deux dimensions permettant de résoudre rapidement et facilement une grande quantité de problèmes mathématiques. En apprentissage profond, nous utilisons principalement trois opérations élémentaires : (i) les additions/ soustractions, (ii) les transposées, et (iii) les multiplications.

La première étape consiste à **vectoriser nos données** (ou *dataset*), notées (X, y). Nous obtenons une matrice X, de dimensions $m \times n$ et un vecteur y de dimensions $m \times 1$, avec m le nombre de données et n le nombre de variables du dataset. Pour cette démonstration, nous travaillerons avec $n = 2$ [9].

Je vais tout d'abord transformer notre matrice X de façon à **obtenir un vecteur** Z, de dimensions $m \times 1$, qui contient les valeurs z, associées à chaque données. Mais comment calculer ce vecteur Z ?

$$A = \begin{pmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(m)} \end{pmatrix} = \begin{bmatrix} w_1 x_1^{(1)} + w_2 x_2^{(1)} + b \\ w_1 x_1^{(2)} + w_2 x_2^{(2)} + b \\ \vdots \\ w_1 x_1^{(m)} + w_2 x_2^{(m)} + b \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + \begin{bmatrix} b \\ b \\ \vdots \\ b \end{bmatrix}$$

Nous pouvons rapidement remarquer que Z est égal au produit matriciel de X et W, auquel est sommé un vecteur b. En termes de dimensions, nous avons donc une matrice X de dimensions $(m, 2)$ multipliée à un vecteur de dimensions $(2, 1)$, ce qui nous donne un résultat de dimensions $(m, 1)$, résultat ensuite additionné au vecteur b, de dimensions $(m, 1)$. Ainsi, nous obtenons l'équation suivante (Equation 2.1.5) :

9. Cette démonstration peut facilement s'appliquer à des datasets contenant N variables (sans modifier les équations), en générant un vecteur W contenant N paramètres.

$$Z = X \cdot W + b \tag{20}$$

Intéressons nous maintenant à la vectorisation de A. Pour rappel, la fonction sigmoïde peut également s'écrire sous cette forme : $a^{(z)} = \sigma(z^{(i)}) = \frac{1}{1+e^{-z(i)}}$. Nous pouvons ré-écrire A tel que A est égal à une fonction σ de Z.

$$A = \begin{pmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{pmatrix} = \begin{bmatrix} \sigma(z^{(1)}) \\ \sigma(z^{(2)}) \\ \vdots \\ \sigma(z^{(m)}) \end{bmatrix} = \sigma \begin{bmatrix} (z^{(1)}) \\ (z^{(2)}) \\ \vdots \\ (z^{(m)}) \end{bmatrix} = \sigma(Z) \tag{21}$$

Nous allons maintenant vectoriser la fonction coût, qui permet de calculer l'erreur globale du modèle.

$$\mathcal{L} = \frac{-1}{m} \sum_{i=0}^{m} \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix} \times log \left(\begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix} \right) + \left(1 - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \right) \times log \left(1 - \begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix} \right) \tag{22}$$

$$= \frac{-1}{m} \sum_{i=0}^{m} \begin{pmatrix} y^{(1)} \times log(a^{(1)}) \\ y^{(2)} \times log(a^{(2)}) \\ \vdots \\ y^{(m)} \times log(a^{(m)}) \end{pmatrix} + \left(1 - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \right) \times log \left(1 - \begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix} \right)$$

$$= \frac{-1}{m} \sum_{i=0}^{m} \begin{pmatrix} y^{(1)} \times log(a^{(1)}) + (1 - y^{(1)}) \times log(1 - a^{(1)}) \\ y^{(2)} \times log(a^{(2)}) + (1 - y^{(2)}) \times log(1 - a^{(2)}) \\ \vdots \\ y^{(m)} \times log(a^{(m)}) + (1 - y^{(m)}) \times log(1 - a^{(m)}) \end{pmatrix}$$

$$= \frac{-1}{m} \sum_{i=0}^{m} y \times log(A) + (1 - y) \times log(1 - A)$$

Nous insérons d'abord les deux vecteurs a et y dans notre équation, puis nous effectuons un produit élément par élément. Pour ensuite nous donner un vecteur de dimensions $(m, 1)$, dont nous effectuons la somme élément par élément puisque dans cette fonction,

nous avons une somme pour i allant de 1 à m. Le résultat obtenu est un nombre réel, ce qui est logique puisque la fonction nous donne un coût, c'est-à-dire une mesure de l'erreur de notre modèle.

Intéressons nous maintenant à la vectorisation de la descente de gradients. Nous avons 3 formules pour mettre à jour les paramètres du modèle : w_1, w_2 et b.

$$\begin{cases} w_1 = w_1 - \alpha \frac{\partial \mathcal{L}}{w_1} \\ w_2 = w_2 - \alpha \frac{\partial \mathcal{L}}{w_2} \\ b = b - \alpha \frac{\partial \mathcal{L}}{b} \end{cases} \tag{23}$$

Nous avons déjà à notre disposition le vecteur W, de composantes w_1 et w_2, nous allons donc modifier le vecteur W en une fois. Nous allons donc mettre nos deux premières équations dans un vecteur et obtenir le résultat suivant :

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial \mathcal{L}}{w_1} \\ \frac{\partial \mathcal{L}}{w_2} \end{bmatrix} \tag{24}$$

Nous pouvons ensuite repérer une nouvelle fois le vecteur W et un facteur α multiplié par un autre vecteur, appelé le **jacobien**, contenant nos gradients. Nous pouvons donc résumer cela comme ceci :

$$W = W - \alpha \frac{\partial \mathcal{L}}{\partial W} \tag{25}$$

En ce qui concerne la paramètre b, il n'est pas nécéssaire de vectoriser l'équation puisque b peut être assimilé à un nombre réel. A ce stade, il faut encore vectoriser nos deux gradients : $\frac{\partial \mathcal{L}}{\partial W}$ (vecteur de dimensions $(2, 1)$) et $\frac{\partial \mathcal{L}}{\partial b}$.

$$\frac{\partial \mathcal{L}}{\partial W} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \end{pmatrix} = \begin{bmatrix} \frac{1}{m} \sum_{i=1}^{m} \left(a^{(i)} - y^{(i)}\right) x_1^{(i)} \\ \frac{1}{m} \sum_{i=1}^{m} \left(a^{(i)} - y^{(i)}\right) x_2^{(i)} \end{bmatrix} \tag{26}$$

$$= \frac{1}{m} \begin{pmatrix} \left(a^{(1)} - y^{(1)}\right) x_1^{(1)} + \left(a^{(2)} - y^{(2)}\right) x_1^{(2)} + \cdots + \left(a^{(m)} - y^{(m)}\right) x_1^{(m)} \\ \left(a^{(1)} - y^{(1)}\right) x_2^{(1)} + \left(a^{(2)} - y^{(2)}\right) x_2^{(2)} + \cdots + \left(a^{(m)} - y^{(m)}\right) x_2^{(m)} \end{pmatrix}$$

$$= \frac{1}{m} \begin{pmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \end{pmatrix} \cdot \left(\begin{pmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{pmatrix} - \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix} \right)$$

Nous obtenons ici un produit matriciel entre le vecteur X^T (de dimensions $(2, m)$) et le vecteur résultant de la soustraction de A et Y (de dimensions $(m, 1)$), ce qui nous donne un vecteur de dimensions $(2, 1)$ (ce qui correspond bien au jacobien). L'équation finale est la suivante :

$$\frac{\partial \mathcal{L}}{W} = \frac{1}{m} X^T \cdot (A - y) \tag{27}$$

Pour le gradient $\frac{\partial \mathcal{L}}{b}$ qui n'est pas un vecteur, nous obtenons :

$$\frac{\partial \mathcal{L}}{b} = \frac{1}{m} \sum_{i=0}^{m} (a^{(i)} - y^{(i)}) = \frac{1}{m} \sum_{i=0}^{m} \left(\begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \right) = \frac{1}{m} \sum_{i=0}^{m} (A - y) \tag{28}$$

Nous avons maintenant vectorisé toutes nos équations. Cette étape est essentielle puisqu'elle permet de traiter des problèmes avec beaucoup de variables.

2.1.6 Programmation d'un neurone artificiel

Nous pouvons à présent implémenter nos équations dans Scilab, afin de créer un neurone artificiel (script final complet en Annexe 2.2.3). Les grandes étapes de l'implémentation sont résumées dans un tableau (Tableau 2).

Commençons par la création d'un jeu de données (dataset) dans Scilab, contenant des valeurs aléatoires. Ici, nous allons travailler avec un dataset contenant 100 lignes et deux variables x_1 et x_2. Cela peut être représenter sous forme d'un tableau à 2 colonnes (une pour chaque x et une pour y), avec chacune 100 lignes. Nous pouvons ensuite implémenter notre fonction d'initialisation des paramètres. Lors de l'affichage, nous obtenons bien un vecteur W de dimensions $(2, 1)$ et un réel b.

Il faut ensuite implémenter les différentes fonctions présentes dans notre algorithme itératif. Tout d'abord, la fonction modèle, pour laquel le résultat est un vecteur A, de dimensions $(100, 1)$. Arrive ensuite la fonction coût, qui nous donne en sortie un réel.

Tableau 2 – Récapitulatif des étapes d'implémentation dans Scilab

Etape	Fonction/Algorithme	Description
1	Dataset	Choix du dataset : (X, y)
2	Fonction d'initialisation	Initialisation des paramètres W et b.
3	Algorithme itératif (répétition en boucle)	1) Fonction présentant notre modèle de neurones artificiels, contenant la fonction linéaire Z puis notre fonction d'activation A 2) Fonction d'évalution ou fonction coût évaluant la performance du modèle en comparant A aux données de référence y 2bis) Calcul des gradients de la fonction coût 3) Mise à jour des paramètres W et b de manière à réduire les erreurs de notre modèle

Puis, il faut implémenter la fonction des gradients, qui nous donne en sortie un vecteur de dimensions $(2, 1)$ et un réel. Enfin, pour finir l'implémentation des fonction, il faut créer une fonction permettant de mettre à jour les paramètres (poids et biais), donnant ainsi en sortie W de dimensions $(2, 1)$ et b un réel.

```
// Generer le dataset (X, y)
NbLignes = 100;
NbVar = 2;
rand("seed", 0);

X = rand(NbLignes, NbVar);
y = rand(NbLignes, 1);

// Representation graphique
clf
plot(X(:, 1), X(:, 2), 'bo', 'MarkerSize', 5);
xlabel('Variable 1');
ylabel('Variable 2');
title('Representation du dataset');

// Creation de la fonction d'initialisation
function [W, b] = initialisation(X)
    W = rand(NbVar, 1);
    b = rand();
endfunction

// Creation de la fonction modele
function A = model(X, W, b)
    Z = X * W + b;
    A = 1 ./ (1 + exp(-Z));
endfunction

// Creation de la fonction cout
m = NbLignes;
function Loss = logloss(A, y)
    Loss = (-1 / m) * sum((1 - y) .* log(A) + y .* log(1 - A));
```

```
endfunction

// Creation de la fonction des gradients
function [dW, db] = gradients(A, X, y)
    dW = (1 / m) * X' * (A - y);
    db = (1 / m) * sum(A - y);
endfunction

// Creation de la fonction de mise a jour
function [W, b] = update(dW, db, W, b, alpha)
    W = W - alpha * dW;
    b = b - alpha * db;
endfunction
```

Listing 5 – Initialisation des fonctions

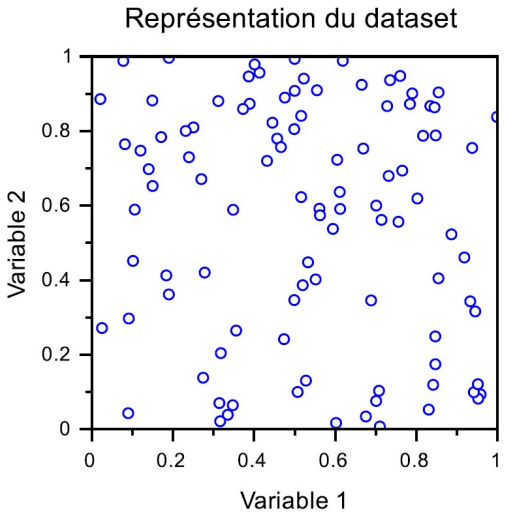


FIGURE 9 – Représentation de (X, y)

Une fois les initialisations terminées, nous pouvons créer notre algorithme itératif (sous forme de fonction), en imposant un taux d'apprentissage $\alpha = 0,1$ et un nombre de cycles $i = 100$:

```
// Creation de l'algorithme iteratif
alpha = 0.1; // taux d'apprentissage
NbIt = 100; // nombre d'iterations

function [W, b, ListeErreur] = neurone_artificiel(X, y, alpha,
   NbIt)
    // Initialisation de W et b
    [W, b] = initialisation(X);

    Iteration = 1:NbIt;
    ListeErreur = zeros(NbIt, 1);

```

```
    for i = 1:NbIt
        A = model(X, W, b);
        Loss = logloss(A, y);
        [dW, db] = gradients(A, X, y);
        [W, b] = update(dW, db, W, b, alpha);
        ListeErreur(i) = Loss;
    end
endfunction

[W, b, ListeErreur] = neurone_artificiel(X, y, alpha, NbIt);

clf();
plot(1:NbIt, ListeErreur, 'r');
xlabel('Cycles d apprentissage');
ylabel('Erreur');
```

Listing 6 – Création du neurone artificiel

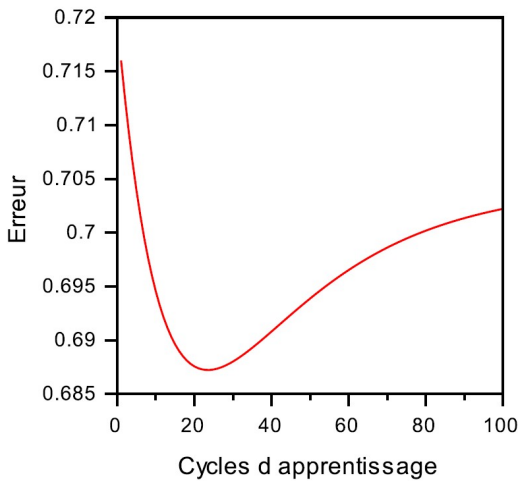


FIGURE 10 – Courbe d'apprentissage obtenue pour $\alpha = 0,1$ et 100 cycles

La courbe d'apprentissage obtenue (Figure 10) peut sembler étrange à première vue puisque l'erreur augmente de nouveau à partir du 30ème cycle environ (traduisant très probablement un problème de convergence ou d'instabilité de l'algorithme).

Nous pouvons ici émettre plusieurs hypothèses. La première est que le taux d'apprentissage serait trop élevé, ce qui peut entraîner des oscillations et des sauts autour de la solution optimale, rendant difficile la convergence vers un minimum global. En réduisant le taux d'apprentissage, notre courbe semble bien converger (Figure 11).

Nous pouvons également émettre une seconde hypothèse : le nombre d'itérations est peut-être insuffisant (i.e., le réseau de neurones ne peut alors pas converger vers une solution optimale). Cependant, l'augmentation du nombre d'itérations afin de permettre au réseau de neurones de s'ajuster davantage aux données, se traduit par un pic d'erreur minimal autour du 5ème cycle suivie d'une augmentation rapide (Figure 12).

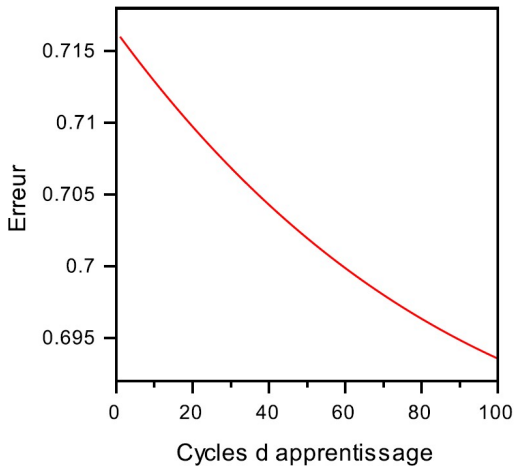


FIGURE 11 – Courbe d'apprentissage obtenue pour $\alpha = 0,01$ et 100 cycles

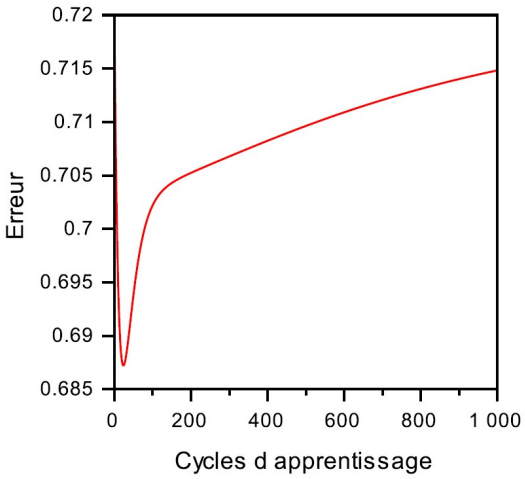


FIGURE 12 – Courbe d'apprentissage obtenue pour $\alpha = 0,1$ et 1000 cycles

Un problème de surapprentissage pourrait aussi être envisagé (troisième hypothèse) : si la courbe d'apprentissage monte seulement vers la fin, cela peut indiquer que le réseau de neurones a commencé à surapprendre les données d'entraînement.

Enfin, une dernière hypothèse concerne la dimensionnalité des données : en cas de dimensions élevées par rapport au nombre d'échantillons, l'apprentissage peut s'avérer plus difficile, voire impossible.

Une fois notre modèle finalisé, nous pouvons nous en servir pour faire des prédictions. C'est-à-dire que si je fournis des valeurs x_1 et x_2, spécifiques à un élément, au modèle, ce dernier pourra me renvoyer la probabilité que cet élément appartienne à l'une ou l'autre classe. Il faut donc créer une nouvelle fonction *predict*.

```
1
```

```
//Creation d'une fonction de prediction
function resultat=predict(X,W,b) //booleen : 1 (TRUE) ou 0 (FALSE)
    A = model(X, W, b);
    resultat = A >= 0.5;
endfunction

// Creation de l'algorithme iteratif
alpha = 0.01; // taux d'apprentissage
NbIt = 100; // nombre d'iterations

function [W, b, ListeErreur] = neurone_artificiel(X, y, alpha, NbIt)
    // Initialisation de W et b
    [W, b] = initialisation(X);

    Iteration = 1:NbIt;
    ListeErreur = zeros(NbIt, 1);

    for i = 1:NbIt
        A = model(X, W, b);
        Loss = logloss(A, y);
        [dW, db] = gradients(A, X, y);
        [W, b] = update(dW, db, W, b, alpha);
        ListeErreur(i) = Loss;
    end

    y_pred = predict(X,W,b);
    disp(y_pred);
endfunction
```

Listing 7 – Création d'une fonction de prédiction

La valeur de seuil de 0,5 a été choisie car elle est couramment utilisée pour la classification binaire, où les prédictions inférieures à 0,5 sont considérées comme propres à la classe "1" et les prédictions supérieures ou égales à 0,5 sont considérées comme propres à la classe "2".

Une autre supposition serait que le problème précédent provienne du jeu de données utilisé. En effet, en utilisant la fonction 'rand()' de Scilab, toutes les valeurs sont situées entre 0,5 et 0,6. Pour tester cette hypothèse, nous pouvons générer une seconde distribution, bimodale cette fois. Le dataset modifié (Listing 8) et sa représentation graphique (Figure 13) sont présentés ci-dessous :

```
// Generer le dataset (X, y)
NbLignes = 100;
NbVar = 2;
rand("seed", 0);

```

```
// Generer des valeurs aleatoires pour X avec distribution bimodale
X1 = rand(NbLignes/2, NbVar) + 0.4;  // Distribution 1 : Ajouter 0.4 a chaque valeur
X2 = rand(NbLignes/2, NbVar) - 0.4;  // Distribution 2 : Soustraire 0.4 a chaque valeur
X = [X1; X2];// Concatenation des deux distributions

//y = rand(NbLignes, 1) >= 0.5; // Generer des etiquettes binaires

y1 = ones(NbLignes/2, 1); // Etiquettes de classe 1 pour X1
y2 = zeros(NbLignes/2, 1); // Etiquettes de classe 2 pour X2
y = [y1; y2]; // Concatenation des etiquettes

// Representation graphique
clf
//plot(X(:, 1), X(:, 2), 'bo', 'MarkerSize', 5);
plot(X1(:,1),X1(:,2),'bo');
plot(X2(:,1),X2(:,2),'go');
xlabel('Variable 1');
ylabel('Variable 2');
title('Representation du dataset');
```

Listing 8 – Création d'un nouveau dataset avec une distribution bimodale

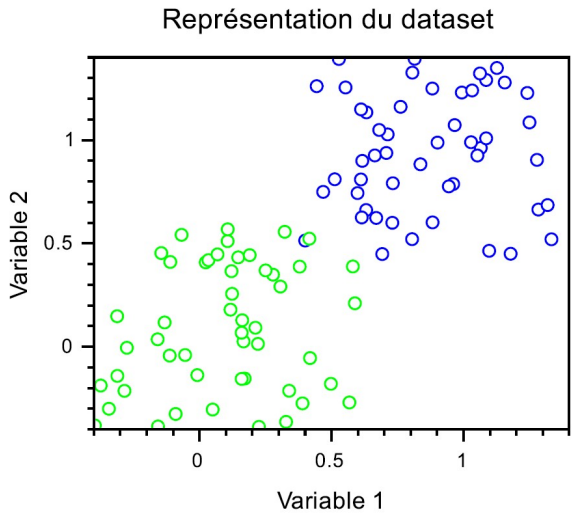


FIGURE 13 – Nouvelle représentation de (X, y)

La courbe d'apprentissage obtenue remonte encore une fois après une certaine valeur mais en jouant avec le taux d'aprentissage, la courbe s'améliore (Figure 14).

J'aimerais maintenant me servir de ce modèle préalablement entraîné pour faire de nouvelles prédictions. Nous pouvons donc tester la réponse du modèle pour quatre

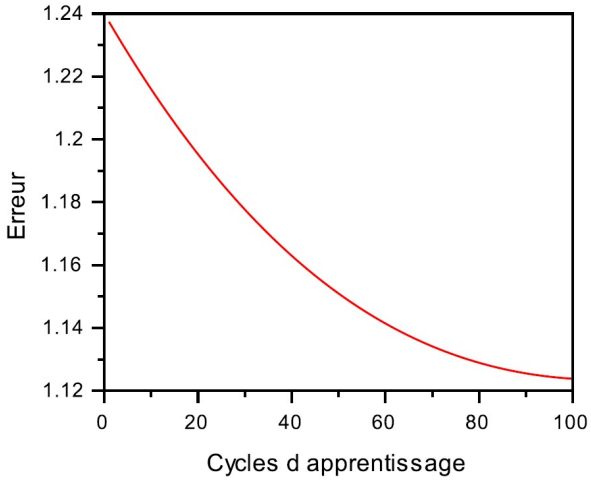


FIGURE 14 – Courbe d'apprentissage obtenue pour $alpha = 0,03$ et 100 cycles

points situés dans les classes "1" et "2" mais non utilisés lors de l'entraînement, et tracer la frontière de décision (l'ensemble des points (x_1, x_2) pour lesquels Z est égal à 0). En utilisant la fonction 'predict', nous obtenons 'F' (False) pour un de nos points et 'T' (True) pour les trois autres. Ces résultats coïncident avec la courbe obtenue (Figure 15). En outre, la frontière de décision sépare bien nos deux classes. Même si cette dernière n'est pas parfaite (ne sépare pas complétement les classes), cela peut s'expliquer par la nature du modèle. En effet, le perceptron simple est un **modèle linéaire**. Le perceptron multi-couche étant un modèle non-linéaire, il pourrait permettre de mieux apprendre à séparer les deux classes de points.

```
//Futures predictions
nouvElement=rand(4,2);
plot(nouvElement(:,1),nouvElement(:,2),'rx','MarkerSize',5);

y_predbis=predict(nouvElement, W, b);
disp(y_predbis);


//Frontiere de decision
x0 = linspace(max(X(:, 1))+0.5, min(X(:, 1))-0.5, 100);
x1 = (-W(1) * x0 - b) / W(2);
plot(x0, x1, 'r');
```

Listing 9 – Futures prédictions et frontière de décision

Dans cette section, nous avons donc vu comment construire un perceptron simple en Scilab, utilisable (et adaptable) pour des datasets test avec N variables (e.g. n=4 dans l'exemple ci-dessus). Il est important de noter qu'il est aussi possible de quantifier les performances du modèle sur des données tests non pas seulement visuellement mais en utilisant un métrique permettant de quantifier la performance de l'algorithme d'apprentissage (fonction appelée *'accuracy'* non implémentée dans ce rapport).

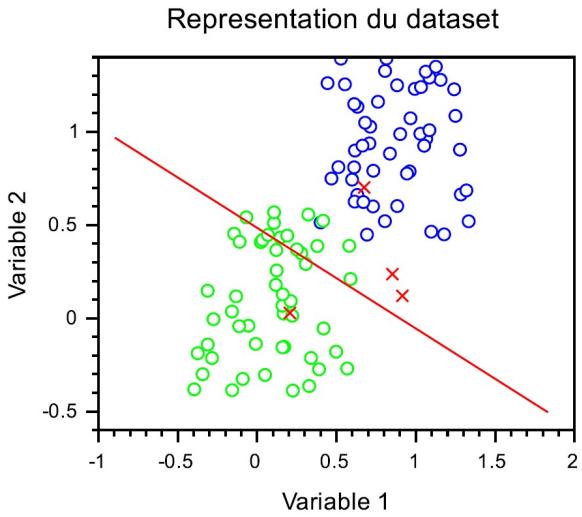


FIGURE 15 – Points de prédictions et frontière de décision

2.2 Le cas du perceptron à deux couches

Nous avons précédemment étudié les modèles de neurones artificiels, aussi connus sous le nom de **régression logistique** [10]. Ces modèles sont adaptés à des cas de figure très simples (e.g., pour séparer deux classes de points linéairement séparables). Cependant, si nous souhaitons résoudre des problèmes plus sophistiqués, nous ne pouvons pas utiliser ce type de modèle qui serait alors "biaisé".

Dans cette section, je vais donc améliorer les performances du modèle en lui ajoutant d'autres neurones : nous passons d'un perceptron mono-couche à un perceptron à deux couches. Autrement dit, en empilant plusieurs couches de neurones, j'obtiens un **modèle non linéaire**, capable de résoudre des problèmes plus complexes.

2.2.1 Propagation ascendante

Pour créer ce nouveau perceptron,je vais tout d'abord ajouter un neurone à côté de celui déjà construit (ces deux neurones seront situés dans la même couche du perceptron). Le premier neurone va alors produire des valeurs (z_1, a_1) et le deuxième va produire des valeurs (z_2, a_2). Puisque les deux neurones ne partagent pas les mêmes connections, ils n'ont pas les mêmes paramètres W et b (paramètres propres à chacun). Les poids W sont notés w_{ij} avec i correpondant au neurone et j correpondant à l'entrée. En suivant la même logique, les biais sont notés b_i avec i correspondant au neurone associé.

Nous pouvons alors écrire une fonction linéaire et une fonction d'activation associée

10. La régression logistique modélise la relation entre les variables explicatives et la variable cible en utilisant une fonction logistique ou sigmoïde. Le modèle de régression logistique est estimé à l'aide d'une méthode d'optimisation appelée la méthode du maximum de vraisemblance.

à chaque neurone (1er et 2ème neurone respectivement) :

$$\begin{cases} z_1 = w_{11}a_1 + w_{12}a_2 + b_1 \\ a_1 = \frac{1}{1+exp(-z_1)} \end{cases} \quad et \begin{cases} z_2 = w_{21}a_1 + w_{22}a_2 + b_2 \\ a_2 = \frac{1}{1+exp(-z_2)} \end{cases} \tag{29}$$

Voici donc un réseau de neurone à une couche (Figure 17), auquel nous pouvons en réalité ajouter autant de neurones que nous le souhaitons [11].

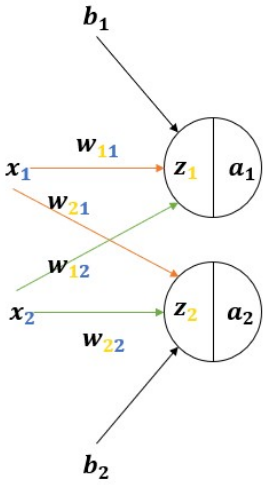


FIGURE 16 – Représentation schématique d'un réseau de neurones à une couche

Les résultats de la première couche peuvent ensuite être envoyés vers une seconde couche. Les éléments de la deuxième couche sont alors notés comme ceci $w_{11}^{[2]}$, tandis que ceux de la première couche seront maintenant notés comme cela $w_{11}^{[1]}$.

Comme précédemment, nous pouvons écrire une fonction linéaire et une fonction d'activation associée au neurone de la deuxième couche :

$$\begin{cases} z_1^{[2]} = w_{11}^{[2]}a_1^{[1]} + w_{12}^{[2]}a_2^{[1]} + b_1^{[2]} \\ a_1^{[2]} = \frac{1}{1+exp(-z_1^{[2]})} \end{cases} \tag{30}$$

Ici aussi, il est possible d'ajouter autant de neurones que nous le souhaitons au sein de cette deuxième couche (il est également possible de rajouter d'autres couches, en prenant en entrée les activations issues des couches précédentes).

11. Puisque chaque neurone possède ses propres paramètres, leur fonctionnement est indépendant les uns des autres, ce qui signifie que plus notre réseau contient de neurones, plus il sera puissant (mais aussi plus il sera lent à entraîner).

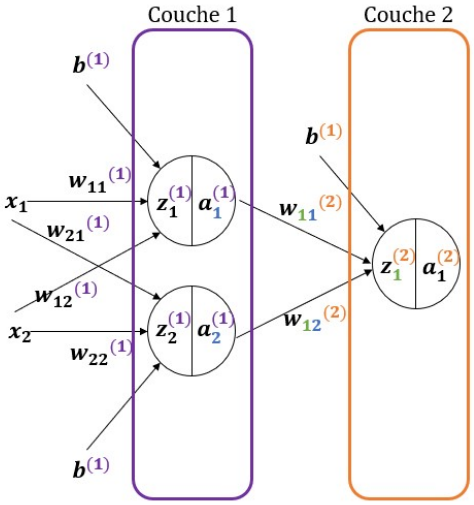


FIGURE 17 – Représentation schématique d'un réseau de neurones à deux couches

Grâce à la vectorisation, nous sommes capables de réunir les résultats de nos deux couches de neurones au sein d'une même matrice (Equation 31) :

$$Z^{[1]} = W^{[1]} \cdot X + b^{[1]} \tag{31}$$

Ici, $Z^{[1]}$ est de dimensions $(n^{[1]}, m)$, $W^{[1]}$ est de dimensions $(n^{[1]}, n^{[0]})$, X est de dimensions $(n^{[0]}, m)$ et $b^{[1]}$ est de dimensions $(n^{[1]}, 1)$, avec $n^{[1]}$ le nombre de neurones de de la première couche et $n^{[0]}$, le nombre de variables ou d'entrées.

Le principal avantage de cette technique est que si nous souhaitons ajouter un troisième neurone sur la première couche, il suffit d'ajouter une troisième colone à la matrice $W^{[1]}$, ainsi qu'un autre paramètre dans le vecteur $b^{[1]}$. Via le calcul matriciel, nous obtiendrons automatiquement une troisième colonne dans la matrice $Z^{[1]}$.

Nous pouvons ici transposer le vecteur X et échanger les vecteur W et X afin d'obtenir un vecteur ligne (parfois écrit comme cela dans la littérature afin de pouvoir aligner chaque ligne de la matrice avec un neurone).

Finalement, à partir de $Z^{[1]}$, nous pouvons calculer $A^{[1]}$ (Equation 32), de dimensions $(n^{[1]}, m)$, qui pourra ensuite être envoyé vers $Z^{[2]}$. Puis $A^{[2]}$ et $Z^{[3]}$ pourront être calculé, etc. Cette étape, permettant de faire passer les données de la première couche à la dernière, est appelée **forward propagation** ou propagation vers l'avant.

$$A^{[1]} = \frac{1}{1 + exp(-Z^{[1]})} \tag{32}$$

Après cette étape, l'entraînement du réseau de neurones peut commencer. Pour cela, nous allons utiliser la technique de **rétro-probagation**.

2.2.2 Rétro-propagation

Dans les sections précédentes, nous avons vu que pour entraîner un réseau de neurones, nous avions besoin d'une fonction coût, du calcul des gradients et d'une mise à jour des paramètres du modèle. Les gradients sont obtenus par la technique de rétro-propagation. Cette dernière consiste à retracer **l'évolution de la fonction coût de la dernière couche du réseau à la première couche**. Cette étape est proche du calcul des gradients effectué dans la partie 2.1.4 (utilisant la règle des chaînes). Nous obtenons les dérivées suivantes pour la deuxième couche (Equations 33 et 34) que nous pouvons simplifier en posant $dZ2$.

$$\frac{\partial \mathcal{L}}{W^{[2]}} = \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial W^{[2]}} = dZ2 \times \frac{\partial Z^{[2]}}{\partial W^{[2]}} \tag{33}$$

$$\frac{\partial \mathcal{L}}{b^{[2]}} = \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial b^{[2]}} = dZ2 \times \frac{\partial Z^{[2]}}{\partial b^{[2]}} \tag{34}$$

Nous pouvons ensuite faire de même pour la première couche (Equations 35 et 36), que nous pouvons simplifier en posant $dZ1$ (contenant $dZ2$).

$$\begin{aligned} \frac{\partial \mathcal{L}}{W^{[1]}} &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial W^{[1]}} \\ &= dZ2 \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial W^{[1]}} \\ &= dZ1 \times \frac{\partial Z^{[1]}}{\partial W^{[1]}} \end{aligned} \tag{35}$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{b^{[1]}} &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial b^{[1]}} \\ &= dZ2 \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial b^{[1]}} \\ &= dZ1 \times \frac{\partial Z^{[1]}}{\partial b^{[1]}} \end{aligned} \tag{36}$$

Nous pouvons maintenant calculer ces dérivées afin de les appliquer au modèle. Pour cela, il faut utiliser les différentes formules de la forward-propagation (Equations 31 et

32). Pour la fonction coût, il suffit d'injecter la sortie $A^{[1]}$ dans la formule logloss (à la place de A).

Commençons par calculer les **gradients de la deuxième couche** et par calculer $dZ2$ (Equation 37). Nous avons déjà calculé $\frac{\partial \mathcal{L}}{\partial A^{[2]}}$ et $\frac{\partial A^{[2]}}{\partial Z^{[2]}}$ dans la partie 2.1.4 (Equation 14 et 16).

$$\begin{aligned} dZ2 &= \frac{\partial \mathcal{L}}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{-y}{A^{[2]}} - \frac{1-y}{1-A^{[2]}}\right) \times A^{[2]}(1-A^{[2]}) \\ &= \frac{1}{m}\sum_{i=1}^{m}\left(-y(1-A^{[2]}) + (1-y)A^{[2]}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(A^{[2]} - y\right) \end{aligned} \tag{37}$$

Il est important de noter que $dZ2$ doit avoir les mêmes dimensions que $Z^{[2]}$ soit $(n^{[2]}, m)$. Ici, $A^{[2]}$ est lui aussi de dimensions $(n^{[2]}, m)$ et y est de dimensions $(1, m)$. Ce décalage peut être résolu par **broadcasting**, également appelée **diffusion** en français, une méthode permettant d'étendre les dimensions de y pour que celui-ci couvre les dimensions de A. Ainsi, $dZ2$ est bien de dimensions $(n^{[2]}, m)$ (nous ne prenons pas la somme en compte). Dans Scilab, le broadcasting n'est pas automatiquement appliqué. Il faut utiliser la fonction 'repmat()' qui permet de répliquer les lignes de notre matrice (Listing 10).

```
// Matrice de taille (4, 4)
a = [1, 1, 1, 1;
     2, 2, 2, 2;
     3, 3, 3, 3
     4, 4, 4, 4];

// Matrice de taille (4,1)
b = [5, 6, 7, 8];

// Broadcasting
nouvB= repmat(b, 4, 1);

// Addition des matrices avec des dimensions compatibles
Total = a + nouvB;

disp(Total);

//Resultat :
//6.    7.    8.    9.
//7.    8.    9.    10.
//8.    9.    10.   11.
//9.    10.   11.   12.
```

Listing 10 – Exemple de broadcasting sur Scilab

Nous pouvons maintenant calculer les dérivées $\frac{\partial \mathcal{L}}{W^{[2]}}$ (Equation 38) et $\frac{\partial \mathcal{L}}{b^{[2]}}$ (Equation 39) en injectant notre $dZ2$.

$$\frac{\partial \mathcal{L}}{W^{[2]}} = dZ2 \times \frac{\partial Z^{[2]}}{\partial W^{[2]}} = dZ2 \times A^{[1]} = dZ2 \cdot A^{[1]^T} \tag{38}$$

Lors de la vérification des dimensions, alors que nous souhaitons obtenir $\frac{\partial \mathcal{L}}{W^{[2]}}$ de dimensions $(n^{[2]}, n^{[1]})$, $dZ2$ est de dimensions $(n^{[2]}, m)$ et $A^{[1]}$ est de dimensions $(n^{[1]}, m)$. Nous allons donc transposer la matrice $A^{[1]}$, afin d'obtenir un produit matriciel (au lieu d'une multiplication élément par élément) et avoir les bonnes dimensions.

$$\frac{\partial \mathcal{L}}{b^{[2]}} = dZ2 \times \frac{\partial Z^{[2]}}{\partial b^{[2]}} = \frac{1}{m} \sum_{colonnes} dZ2 \tag{39}$$

Ici, la dérivée $\frac{\partial Z^{[2]}}{\partial b^{[2]}}$ est égale à 1. Lors de la vérification des dimensions, alors que nous souhaitons obtenir $\frac{\partial \mathcal{L}}{b^{[2]}}$ de dimensions $(n^{[2]}, 1)$, $dZ2$ est de dimensions $(n^{[2]}, m)$. Pour résoudre ce problème, nous pouvons réintroduire la somme (non prise en compte au préalable). m sera alors égale à 1 (somme des colonnes de la matrice).

Intéressons nous maintenant aux **gradients de la première couche** et commençons par $dZ1$ (Equation 40).

$$\begin{aligned} dZ1 &= dZ2 \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial b^{[1]}} \\ &= dZ2 \times W^{[2]} \times A^{[1]}(1 - A^{[1]}) \\ &= W^{[2]^T} \cdot dZ2 \times A^{[1]}(1 - A^{[1]}) \end{aligned} \tag{40}$$

Concernant les dimensions, $dZ1$ doit être de dimensions $(n^{[1]}, m)$. Or, $dZ2$ est de dimensions $(n^{[2]}, m)$, $W^{[2]}$ est de dimensions $(n^{[2]}, n^{[1]})$ et $A^{[A]}$ est de dimensions $(n^{[1]}, m)$. Ici encore, nous constatons des problèmes de dimensions. Une solution est de passer $W^{[2]}$ de l'autre côté de $dZ2$ et de le transposer (dimensions alors égales à $(n^{[1]}, n^{[2]})$). Le produit matriciel est alors de dimensions $(n^{[1]}, m)$, que l'on peut multiplier termes à termes avec $A^{[1]}(1 - A^{[1]})$, également de dimensions $(n^{[1]}, m)$ pour au final obtenir les dimensions espérées de $dZ1$.

Nous pouvons ensuite simplifier nos formules de gradients $\frac{\partial \mathcal{L}}{W^{[1]}}$ (Equation 41) et $\frac{\partial \mathcal{L}}{b^{[1]}}$ (Equation 42) en injectant notre $dZ1$.

$$\frac{\partial \mathcal{L}}{W^{[1]}} = dZ1 \times \frac{\partial Z^{[1]}}{\partial W^{[1]}} = dZ1 \times X = dZ1 \cdot X^T \tag{41}$$

$$\frac{\partial \mathcal{L}}{b^{[1]}} = dZ1 \times \frac{\partial Z^{[1]}}{\partial b^{[1]}} = dZ1 = \frac{1}{m} \sum_{colonnes} dZ1 \tag{42}$$

Vérifions maintenant nos dimensions. $\frac{\partial \mathcal{L}}{W^{[1]}}$ doit être de dimensions $(n^{[1]}, n^{[0]})$. Or, $dZ1$ est de dimensions $(n^{[1]}, m)$ et X est de dimensions $(n^{[0]}, m)$. Ici encore, il nous suffit de transposer X pour obtenir les bonnes dimensions. Pour $\frac{\partial \mathcal{L}}{b^{[1]}}$, nous pouvons appliquer la somme sur les colonnes, comme vu plus haut, afin d'avoir comme dimensions $(n^{[1]}, 1)$.

2.2.3 Programmation d'un réseau de neurones à 2 couches

Nous pouvons à présent implémenter nos équations dans Scilab, afin de créer un **réseau de neurones artificiels à 2 couches** (script final complet en Annexe 2.2.3). La structure de notre code sera très similaire à celle du neurone artificiel simple. En effet, au lieu d'avoir comme paramètres W et b, nous aurons $W^{[1]}$, $W^{[2]}$, $b^{[1]}$ et $b^{[2]}$. De même pour la fonction du modèle, au lieu d'avoir Z et A, nous aurons $Z^{[1]}$, $Z^{[2]}$, $A^{[1]}$ et $A^{[2]}$.

Comme précédemment, nous pouvons tout d'abord générer un jeu de données aléatoires, puis adapter nos différentes fonctions d'initialisation (Listing 17). Les fonctions 'model' et 'gradients' sont renommées respectivement : 'forward-propagation' et 'back-propagation'.

```
1
2  //Creation d'un dataset
3  function [X, y] = generate_dataset(NbEch)
4      X = rand(2, NbEch);
5      y = (X(1, :) + X(2, :)) >= 1;
6  endfunction
7
8  NbEch = 1000;
9  [X_train, y_train] = generate_dataset(NbEch);
10
11 //Initialisation variables
12 NbIt=100;
13 alpha = 0.1;//taux d'apprentissage
14 n0= size(X_train, "r");//nombre de variables dans X
15 n2= size(y_train, "r");
16 n1=2;//A nous de definir n1 car nombres de neurones dans notre
       premiere couche
17
18 // Creation de la fonction d'initialisation
19 function [W1, b1, W2, b2] = initialisation(n0, n1, n2)
20     //Parametres de la 1ere couche
21     W1 = rand(n1, n0);
22     b1 = rand(n1,1);
```

```
23      //Parametre de la 2eme couche
24      W2 = rand(n2, n1);
25      b2 = rand(n2, 1);
26  endfunction
27
28  // Creation de la fonction model renommee forward_propagation
29  function [A1, A2] = forward_propagation(X, W1, b1, W2, b2)
30      //Z1 = W1 * X + b1* ones(1, size(X, 2));;
31      Z1 = W1 * X + repmat(b1, 1, size(X, 1));
32      A1 = 1 ./ (1 + exp(-Z1));
33      Z2 = W2 * A1 + b2*ones(1, size(A1, 1));
34      A2 = 1 ./ (1 + exp(-Z2));
35  endfunction
36
37  // Creation de la fonction des gradients renommee backpropagation
38  y_size = size(y_train);
39  m = y_size(2);
40  function [dW1, db1, dW2, db2] = backpropagation(X, y, A1, A2, W1)
41
42      dZ2 = A2 - y;
43      dW2 = (1 / m) * dZ2 * (A1');
44      db2 = (1 / m) * sum(dZ2, "c");
45      db2 = db2 * ones(1, size(dZ2, 2)); // pour conserver un
           tableau a deux dimensions (important sinon effets de
           broadcasting)
46
47      dZ1 = (W2'*dZ2).*(A1.*(1-A1));
48      dW1 = (1 / m) * dZ1 * (X');
49      db1 = (1 / m) * sum(dZ1, "c");
50      db1 = db1 * ones(1, size(dZ1, 2)); // pour conserver un
           tableau a deux dimensions (important sinon effets de
           broadcasting)
51
52  endfunction
53
54  // Creation de la fonction de mise a jour
55  function [W1, b1, W2, b2] = update(dW1, dW2, db1, db2, W1, b1,
       W2, b2, alpha)
56      W1 = W1 - alpha * dW1;
57      //b1 = b1 - alpha * db1;
58      b1 = repmat(b1, 1, size(db1, 1 +i)) - alpha * db1;
59      W2 = W2 - alpha * dW2;
60      //b2 = b2 - alpha * db2;
61  endfunction
62
63  // Creation de la fonction cout
64  function Loss = logloss(A, y)
65      m = size(A,2);
66      Loss = (-1 / m) * sum((1 - y) .* log(A) + y .* log(1 - A));
```

```
endfunction

//Creation d'une fonction de prediction
function resultat = predict(X, W1, b1, W2, b2) //booleen : 1 (TRUE) ou 0 (FALSE)
    [A1, A2] = forward_propagation(X, W1, b1, W2, b2);
    resultat = A2 >= 0.5;
endfunction
```

Listing 11 – Initialisation des fonctions dites "de base"

Nous pouvons ensuite créer l'algorithme itératif en s'inspirant de celui du neurone artificiel simple (Listing 12). Une courbe d'apprentissage peut également être générée pour rechercher une convergence de l'erreur vers 0.

```
//Creation de l'algorithme iteratif
function [W1, W2, b1, b2] = reseau_neurones(X_train, y_train, alpha, NbIt, n1)
    // Initialisation des parametres
    [W1, b1, W2, b2] = initialisation(n0, n1, n2);

    train_loss=[];
    train_acc=[];
    for i = 1:NbIt
        disp(i)
        [A1, A2] = forward_propagation(X_train, W1, b1, W2, b2);
        [dW1, db1, dW2, db2] = backpropagation(X_train, y_train, A1, A2, W1);
        [W1, b1, W2, b2] = update(dW1, dW2, db1, db2, W1, b1, W2, b2, alpha);
        Loss = logloss(A2, y_train);
        if (modulo(i,10) == 0)
            train_loss = [train_loss, Loss];
            y_pred = predict(X_train, W1, b1, W2, b2);
        end
    end

    plot(train_loss, 'b', 'LineWidth', 1);
    xlabel('Iterations');
    ylabel('Train Loss');
    title('Train Loss');
    legend('Train Loss');

    //xtitle('Training Progress', 'fontsize', 14, 'fontweight', 'bold');

endfunction

```

```
[W1, W2, b1, b2] = reseau_neurones(X_train, y_train, alpha,
    NbIt, n1)
```

Listing 12 – Initialisation des fonctions dites "de base"

Une des premières remarques que l'on peut faire est qu'il y a un problème de dimensions lors de la mise à jour des biais $b1$ et $b2$ dans la fonction 'update()'. Cependant, en mettant les deux lignes en question en commentaire, l'algorithme tourne et nous obtenons une courbe (Figures 18). Nous pouvons voir que la courbe d'apprentissage converge ce qui est bon signe.

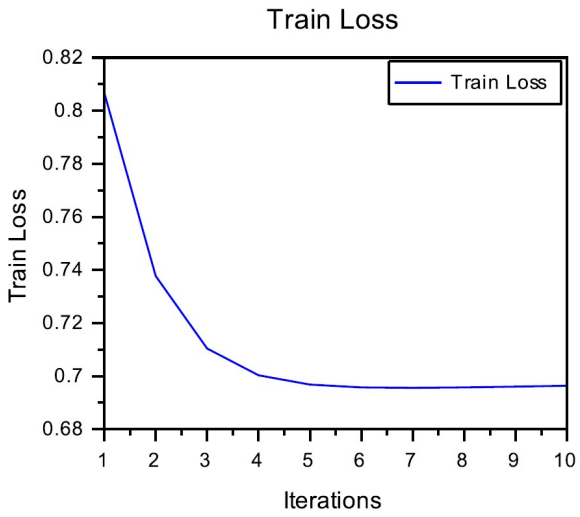


FIGURE 18 – Courbe d'apprentissage obtenue pour $alpha$=0.1

Après avoir finalement réussi à régler le problème du biais (Listing 13) et en jouant sur les dimensions des matrices de certaines fonctions (broadcasting), l'algorithme s'est arrêté de nouveau mais cette fois-ci sur une équation de 'forward propagation' lorsque i=2 (i.e., au deuxième passage dans la boucle 'for'). Ce problème pourrait être réglé par broadcasting, cependant, comme évoqué plus haut, cette fonction n'existe pas déjà dans Scilab (contrairement à Python par exemple). Je suis donc obligé de le faire manuellement avec 'repmat()'. Or, cela a été fait pour l'itération une, mais n'est donc pas adapté à l'itération deux. Après plusieurs tentatives, j'ai finalement décider d'itérer le nombre de colonne dans 'repmat()', afin de créer un ajustement automatique à chaque itération (Listing 14).

```
// Creation de la fonction de mise a jour
function [W1, b1, W2, b2] = update(dW1, dW2, db1, db2, W1, b1,
    W2, b2, alpha)
    W1 = W1 - alpha * dW1;
    //b1 = b1 - alpha * db1;
    b1 = repmat(b1, 1, size(db1, 2)) - alpha * db1;
    W2 = W2 - alpha * dW2;
    b2 = b2 - alpha * db2;
endfunction
```

Listing 13 – Résolution du problème de mise à jour de $b1$

```
// Creation de la fonction model renommee forward_propagation
function [A1, A2] = forward_propagation(X, W1, b1, W2, b2)
    //Z1 = W1 * X + b1* ones(1, size(X, 2));;
    Z1 = W1 * X + repmat(b1, 1, size(X, 1 + i));
    A1 = 1 ./ (1 + exp(-Z1));
    Z2 = W2 * A1 + b2*ones(1, size(A1, 1 + i));
    A2 = 1 ./ (1 + exp(-Z2));
endfunction
```

Listing 14 – Résolution du problème de mise à jour de $b1$

L'algorithme ainsi mis à jour fonctionne. En relançant plusieurs fois ce dernier, nous pouvons observer une amélioration progressive de l'apprentissage et une convergence vers 0.7 (Figure 19). Pour aller plus loin, nous pourrions (comme évoqué dans le cas du perceptron simple) évaluer la performance du modèle en mesurant son accuracy (non implémentée ici).

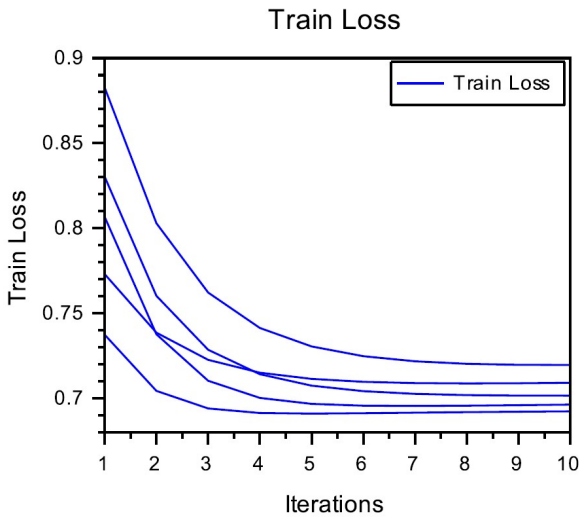


FIGURE 19 – Courbe d'apprentissage obtenue pour $alpha$=0.1

Pour conlure cette section, le passage d'un perceptron deux couches à un perceptron multi-couches (et donc à un réseau de neurones profond) nécessite quelques adaptations des formules mathématiques (avec plus de paramètres), puis de l'implémentation (comme vu précédemment lors du passage d'un perceptron simple au perceptron double). Cependant, l'utilisation des techniques de propagation vers l'avant et de rétro-propagation ne varie pas.

⋆ ⋆ ⋆

Conclusion

En terme de perspectives, il serait intéressant de pouvoir entrainer nos algorithmes d'apprentissage par perceptron multicouches sur des jeux de données réelles. Dans ce rapport, j'ai simulé les datasets d'entraînement et de test par soucis didactique, mais il est aussi possible d'utiliser des banques de données libres de droit, disponibles sur le web (e.g. ICDAR ou tutoriel sur www.scilab.org pour entraîner des algorithmes d'*Optical Character Recognition* (OCR), permettant la reconnaissance de lettres de l'alphabet manuscrites).

Pour conclure, ce cahier d'intégration a été pour moi l'occasion d'étudier ce qu'était l'apprentissage profond, de tester différents types de perceptrons (simples, doubles et multicouches), de comprendre les concepts mathématiques sous-jacents et de les implémenter dans Scilab. En complément de mes lectures, je remercie les auteurs de vidéos en ligne (Machine Learnia [5]), tutoriels (Hagan et al. [6], Malgouyres [7], Scilab [8], Mathworks [9]), blogs (Dutta [4]) et autres comptes Github (Tanchinluh [10]) pour les ressources en libre accès m'ayant permis de progresser en parallèle des cours et de mes premiers scripts. Ce cahier d'intégration m'a permis de mettre en pratique des concepts mathématiques et informatiques qui me paraissaient jusque-là très abstraits et d'apprendre à les utiliser dans Scilab. Je suis persuadée que les concepts abordés dans ce rapport et tout au long de l'UV me seront très utiles dans ma poursuite d'étude et ma future carrière, étant personnellement intéressée par la recherche et voulant m'orienter en filière IAD (Intelligence Artificielle et Sciences de Données). Je suis ravie d'avoir suivi cette UV.

Références

[1] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4) :115–133, December 1943. ISSN 1522-9602. doi : 10.1007/BF02478259. URL https://doi.org/10.1007/BF02478259. 4

[2] D.O. Hebb. The Organization of Behavior : A Neuropsychological Theory. Psychology Press, New York, May 2002. ISBN 978-1-4106-1240-3. doi : 10.4324/9781410612403. 7

[3] M. Minsky and S.A. Papert. Perceptrons : An Introduction to Computational Geometry. The MIT Press, September 2017. ISBN 978-0-262-34393-0. doi : 10.7551/mitpress/11301.001.0001. URL https://direct.mit.edu/books/book/3132/PerceptronsAn-Introduction-to-Computational. 9

[4] S. Dutta. Implementing the XOR Gate using Backpropagation in Neural Networks, May 2022. URL https://towardsdatascience.com/implementing-the-xor-gate-using-backpropagation-in-neural-networks-c1f255b4f20d. 10, 41, ix

[5] Machine Learnia. Formation Deep-Learning Complète (2021), April 2021. URL https://www.youtube.com/watch?v=XUFLq6dKQok. 41

[6] M.T. Hagan, H.B. Demuth, M.H. Beale, and O.D. Jesús. Neural Network Design. Martin Hagan, s.L, 2nd ed. edition edition, September 2014. ISBN 978-0-9717321-1-7. URL https://github.com/estamos/Neural-Network-Design-Solutions-Manual. 41

[7] F. Malgouyres. Francois Malgouyres : Enseignement (M2 MVA). URL https://www.math.univ-toulouse.fr/~fmalgouy/enseignement/mva.html. 41

[8] Scilab. Tutorials | Scilab. URL https://www.scilab.org/tutorials?field_tutorials_tid=17. 41

[9] Mathworks. Deep Learning : 3 choses à savoir - MATLAB & Simulink. URL https://fr.mathworks.com/discovery/deep-learning.html. 41

[10] Tanchinluh. Scilab Neural Network Module, May 2020. URL https://github.com/tanchinluh/neuralnetwork. original-date : 2019-10-15T01 :34 :17Z. 41

Annexe 1

```
function trace(x0, N, h)
    lx=list();
    ly=list();
    for j=1:length(x0)
        i=1:N(j);
        y=(i-1)*h-(N(j)-1)/2*h;
        x=x0(j)*ones(1,N(j));
        hg=plot(x,y,'o')
        hg.mark_size=20
        lx(j)=x;
        ly(j)=y;
    end
    for j=1:length(x0)-1
        for i=1:length(lx(j))
            for k=1:length(lx(j+1))
                X=[lx(j)(i) lx(j+1)(k)];
                Y=[ly(j)(i) ly(j+1)(k)];
                plot(X,Y)
            end
        end

    end
endfunction

h=1;
clf

trace([h 2*h 3*h],[4 3 1],h)

gca().data_bounds=[0 4 -4 4]
```

Listing 15 – Programme Scilab de schéma de réseau de neurone

Annexe 2

Table de vérité du OU LOGIQUE (ou OR) :

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	1

Table de vérité du ET LOGIQUE (ou AND) :

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Table de vérité du OU EXCLUSIF (ou XOR) :

A	B	$A \vee B$
0	1	1
0	0	0
1	1	0
1	0	1

Annexe 3

```
// Generer le dataset (X, y)
NbLignes = 100;
NbVar = 2;
rand("seed", 0);

// Generer des valeurs aleatoires pour X avec distribution bimodale
X1 = rand(NbLignes/2, NbVar) + 0.4;  // Distribution 1 : Ajouter 0.4 a chaque valeur
X2 = rand(NbLignes/2, NbVar) - 0.4;  // Distribution 2 : Soustraire 0.4 a chaque valeur
X = [X1; X2];// Concatenation des deux distributions

//y = rand(NbLignes, 1) >= 0.5; // Generer des etiquettes binaires

y1 = ones(NbLignes/2, 1); // Etiquettes de classe 1 pour X1
y2 = zeros(NbLignes/2, 1); // Etiquettes de classe 2 pour X2
y = [y1; y2]; // Concatenation des etiquettes

// Representation graphique
clf
//plot(X(:, 1), X(:, 2), 'bo', 'MarkerSize', 5);
plot(X1(:,1),X1(:,2),'bo');
plot(X2(:,1),X2(:,2),'go');
xlabel('Variable 1');
ylabel('Variable 2');
title('Representation du dataset');

// Creation de la fonction d'initialisation
function [W, b] = initialisation(X)
    W = rand(NbVar, 1);
    b = rand();
endfunction

// Creation de la fonction modele
function A = model(X, W, b)
    Z = X * W + b;
    A = 1 ./ (1 + exp(-Z));
endfunction

// Creation de la fonction cout
m = NbLignes;
function Loss = logloss(A, y)
    Loss = (-1 / m) * sum((1 - y) .* log(A) + y .* log(1 - A));
endfunction

```

```
// Creation de la fonction des gradients
function [dW, db] = gradients(A, X, y)
    dW = (1 / m) * X' * (A - y);
    db = (1 / m) * sum(A - y);
endfunction

// Creation de la fonction de mise a jour
function [W, b] = update(dW, db, W, b, alpha)
    W = W - alpha * dW;
    b = b - alpha * db;
endfunction

//Creation d'une fonction de prediction
function resultat = predict(X, W, b) //booleen : 1 (TRUE) ou 0 (FALSE)
    A = model(X, W, b);
    resultat = A >= 0.5;
endfunction


// Creation de l'algorithme iteratif
alpha = 0.1; // taux d'apprentissage
NbIt = 100; // nombre d'iterations

function [W, b, ListeErreur] = neurone_artificiel(X, y, alpha, NbIt)
    // Initialisation de W et b
    [W, b] = initialisation(X);

    Iteration = 1:NbIt;
    ListeErreur = zeros(NbIt, 1);

    for i = 1:NbIt
        A = model(X, W, b);
        Loss = logloss(A, y);
        [dW, db] = gradients(A, X, y);
        [W, b] = update(dW, db, W, b, alpha);
        ListeErreur(i) = Loss;
        //accuracy=1-Loss;
    end

endfunction

[W, b, ListeErreur] = neurone_artificiel(X, y, alpha, NbIt);

//clf();
//plot(1:NbIt, ListeErreur, 'r');
//xlabel('Cycles d apprentissage');
//ylabel('Erreur');
```

```
91
92 //Futures predictions
93 nouvElement=rand(4,2);
94 plot(nouvElement(:,1),nouvElement(:,2),'rx','MarkerSize',5);
95
96 y_predbis=predict(nouvElement, W, b);
97 disp(y_predbis);
98
99
100 //Frontiere de decision
101 x0 = linspace(max(X(:, 1))+0.5, min(X(:, 1))-0.5, 100);
102 x1 = (-W(1) * x0 - b) / W(2);
103 plot(x0, x1, 'r');
```

Listing 16 – Script Scilab d'un neurone artificiel

Annexe 4

```

//Creation d'un dataset
function [X, y] = generate_dataset(NbEch)
    X = rand(2, NbEch);
    y = (X(1, :) + X(2, :)) >= 1;
endfunction

NbEch = 1000;
[X_train, y_train] = generate_dataset(NbEch);

//Initialisation variables
NbIt=100;
alpha = 0.1;//taux d'apprentissage
n0= size(X_train, "r");//nombre de variables dans X
n2= size(y_train, "r");
n1=2;//A nous de definir n1 car nombres de neurones dans notre
   premiere couche

// Creation de la fonction d'initialisation
function [W1, b1, W2, b2] = initialisation(n0, n1, n2)
    //Parametres de la 1ere couche
    W1 = rand(n1, n0);
    b1 = rand(n1,1);
    //Parametre de la 2eme couche
    W2 = rand(n2, n1);
    b2 = rand(n2, 1);
endfunction



// Creation de la fonction model renommee forward_propagation
function [A1, A2] = forward_propagation(X, W1, b1, W2, b2)
    //Z1 = W1 * X + b1* ones(1, size(X, 2));;
    Z1 = W1 * X + repmat(b1, 1, size(X, 1 + i));
    A1 = 1 ./ (1 + exp(-Z1));
    Z2 = W2 * A1 + b2*ones(1, size(A1, 1 + i));
    A2 = 1 ./ (1 + exp(-Z2));
endfunction



// Creation de la fonction des gradients renommee backpropagation
y_size = size(y_train);
m = y_size(2);
function [dW1, db1, dW2, db2] = backpropagation(X, y, A1, A2, W1)

    dZ2 = A2 - y;
```

```
47      dW2 = (1 / m) * dZ2 * (A1');
48      db2 = (1 / m) * sum(dZ2, "c");
49      db2 = db2 * ones(1, size(dZ2, 2)); // pour conserver un
           tableau a deux dimensions (important sinon effets de
           broadcasting)
50
51      dZ1 = (W2'*dZ2).*(A1.*(1-A1));
52      dW1 = (1 / m) * dZ1 * (X');
53      db1 = (1 / m) * sum(dZ1, "c");
54      db1 = db1 * ones(1, size(dZ1, 2)); // pour conserver un
           tableau a deux dimensions (important sinon effets de
           broadcasting)
55
56  endfunction
57
58
59
60  // Creation de la fonction de mise a jour
61  function [W1, b1, W2, b2] = update(dW1, dW2, db1, db2, W1, b1,
       W2, b2, alpha)
62      W1 = W1 - alpha * dW1;
63      //b1 = b1 - alpha * db1;
64      b1 = repmat(b1, 1, size(db1, 1 +i)) - alpha * db1;
65      W2 = W2 - alpha * dW2;
66      b2 = b2 - alpha * db2;
67  endfunction
68
69  // Creation de la fonction cout
70  function Loss = logloss(A, y)
71      m = size(A,2);
72      Loss = (-1 / m) * sum((1 - y) .* log(A) + y .* log(1 - A));
73  endfunction
74
75  //Creation d'une fonction de prediction
76  function resultat = predict(X, W1, b1, W2, b2) //booleen : 1
       (TRUE) ou 0 (FALSE)
77      [A1, A2] = forward_propagation(X, W1, b1, W2, b2);
78      resultat = A2 >= 0.5;
79  endfunction
80
81
82
83  //Creation de l'algorithme iteratif
84  function [W1, W2, b1, b2] = reseau_neurones(X_train, y_train,
       alpha, NbIt, n1)
85      // Initialisation des parametres
86      [W1, b1, W2, b2] = initialisation(n0, n1, n2);
87
88      train_loss=[];
```

```
89      train_acc=[];
90      for i = 1:NbIt
91          disp(i)
92          [A1, A2] = forward_propagation(X_train, W1, b1, W2, b2);
93          [dW1, db1, dW2, db2] = backpropagation(X_train, y_train,
                A1, A2, W1);
94          [W1, b1, W2, b2] = update(dW1, dW2, db1, db2, W1, b1,
                W2, b2, alpha);
95          Loss = logloss(A2, y_train);
96          if (modulo(i,10) == 0)
97              train_loss = [train_loss, Loss];
98              y_pred = predict(X_train, W1, b1, W2, b2);
99          end
100     end
101
102     plot(train_loss, 'b', 'LineWidth', 1);
103     xlabel('Iterations');
104     ylabel('Train Loss');
105     title('Train Loss');
106     legend('Train Loss');
107
108     //xtitle('Training Progress', 'fontsize', 14, 'fontweight',
            'bold');
109
110 endfunction
111
112 [W1, W2, b1, b2] = reseau_neurones(X_train, y_train, alpha,
        NbIt, n1)
```

Listing 17 – Script Scilab d'un réseau de neurones 2 couches

Annexe 5

Liste des figures

Liste des tableaux

⋆ ⋆ ⋆