Mémoire de recherche

valant évaluation dans le cadre de :

Diplôme : M1 SEP mention Mathémtiques
Année universitaire : 2021-2022
Module d'enseignement : SEP0851
Responsable : Philippe Regnault
Comptant pour : 50 %

Finalisé le : 17 mai 2022
Page(s) : 56
Références(s) : 0
Figure(s) – Table(s) : 42 – 0
Théorème(s) : 0

Théorie et implémentation des réseaux de neurones de type RNN et LSTM

Léo GABET leo.gabet@etudiant.univ-reims.fr

Mots-clés : Machine Learning, Deep Learning, IA, Perceptron, Gradient.

Résumé : Dans ce mémoire de recherche, l'objectif va d'être d'approcher et d'étudier mathématiquement la théorie des réseaux de neurones à c couche(s) et n neurone(s) dans le but de comprendre le fonctionnement des réseaux de neurones de type RNN et LSTM.

Table des matières

1. Introduction

De nos jours, la recherche en intelligence artificielle se développe de plus en plus, ce qui nous amène à créer des algorithmes plus complexe et plus dense. L'outil mathématiques se retrouve être au cœur du sujet de l'IA (Intelligence Artificiel), c'est pour cela que nous détaillerons plus concrètement l'importance de ce domaine dans ce sujet. Avant tout, nous allons expliciter les différentes définitions de certains termes que nous serons amenés à utiliser pour l'explication de plusieurs mécanismes et fonctionnement de l'IA. De plus, nous exposerons le côté historique de l'évolution dans l'IA à travers les nombreux chercheurs et découvertes qui ont permis de faire avancer le domaine de l'IA. Suite à cela, nous commencerons par aborder le thème principal de notre sujet de mémoire qui sera basé sur le fonctionnement et l'intérêt des réseaux de neurones pour le développement de l'IA, ainsi que son exploitation dans différents domaines de la vie courante à travers les RNN (Recurrent Neural Network) et LSTM (Long Short Time Memory).

Voici quelques définitions qui nous serons utiles pour la suite :

1.1. Machine Learning

Le machine Learning repose sur le fait de pouvoir développer un modèle, en se servant d'un algorithme d'optimisation pour minimiser les erreurs entre le modèle et les données acquises. Ce concept se base donc sur de multiples modèles comme les modèles linéaires, les arbres de décision et les supports Vector machines. Par exemple, pour chaque modèle, nous aurons un algorithme d'optimisation, la descente de gradient pour le modèle linéaire, l'algorithme CART pour les arbres de décision, ainsi que le marge maximum pour les supports Vector machines.

1.2. Perceptron

Le perceptron est un modèle linéaire qui dépend de plusieurs paramètres dont le biais, nous exploiterons énormément ce terme, car les réseaux de neurones récurrents ne sont autres que des perceptrons. Le perceptron s'inspire d'origine de la théorie de Hebb, sur le fait que lorsque deux neurones biologiques sont excités conjointement, alors ils renforcent leur lien synaptique. Cela s'avère très intéressant d'exploiter ce lien dans la mise en application des outils mathématiques que nous verrons par la suite. En effet, le but sera d'entraîner un neurone artificiel sur des données de références, par exemple un couple (X, Y) pour que celui-ci renforce ses paramètres que nous nommerons ici W à chaque fois qu'une entrée X est activée en même temps que la sortie Y présente dans ces données. Nous reverrons cela plus tard.

1.3. Perceptron multicouche

Le perceptron multicouche, comme son nom l'indique, à pour intérêt de connecter plusieurs perceptrons entre eux. Ce procédé permettra de surélever les problèmes des modèles non linéaires qui ont bloqué l'évolution de l'IA durant un certain temps, comme nous le verrons d'ici quelques instants.

1.4. Neurones artificiels

Les neurones artificiels que nous nommerons réseau de neurones artificiels, sont basés à l'origine sur le système de fonctionnement et de conception des neurones biologiques qui ont pour but aujourd'hui de se rapprocher des méthodes statistiques. Attention, les neurones artificiels ne fonctionnent pas comme ceux du cerveau humain. Par exemple, dans le domaine de l'aviation, nous nous sommes inspirés des oiseaux, ce n'est pas pour autant que les avions en aéronautique volent comme eux.

1.5. Deep Learning

Le Deep Learning permet de développer un réseau de neurones artificiels à partir des données, des modèles et des algorithmes d'optimisations, ainsi que sur la minimisation des erreurs (CHARNIAK, E 2021).

2. Côté Historique

Nous allons ici détailler les moments clés de l'évolution dans l'IA à travers les nombreux chercheurs et découvertes qui ont permis de faire avancer ce domaine scientifique.

Nous commençons en **1943** par l'invention des premiers neurones artificiels nommé les *TRESHOLD LOGIC UNIT* (origine pour des entrées logiques comme 0 et 1) par Warren McCulloch [1] et Walter Pitts [2] qui étaient tous les deux des neurologistes américains. Le principe de ces neurones repose sur l'inspiration des neurones biologiques du cerveau humain. En effet, à leur époque c'était une toute nouvelle science de travailler sur des neurones artificiels. Durant les années de **1950**, Alan Turing [3] publia un article nommé le *Computing Machinery and Inteligence* sur le thème de l'IA dans lequel fut introduit le test de Turing. Ce test représente le fait qu'une machine puisse imiter une conversation humaine. En **1957**, Frank Rosenblatt [4] inventa le perceptron que nous avons défini plus haut. Malheureusement, en **1969**, les chercheurs Marvin Lee Minsky [5] et Seymour Papert [6] ont publié un article nommé *XOR Problem* mettant en avant les limites théoriques du perceptron, par exemple l'impossibilité de traiter des problèmes non linéaires. Puis, ils propagèrent implicitement toutes ces limitations à tous les modèles de réseaux de neurones artificiels. Ce qui entraîna une grande impasse pour la recherche en IA. Entre **1970** et **1985**, il n'y a plus eu aucun investisseur en IA ce qui freina considérablement les recherches qui furent à l'époque dites "gelés". Puis, en **1986**, Geoffrey Hinton [7] inventa le concept du Perceptron Multicouche, qui tout comme la définition du perceptron, nous l'avons défini précédemment. En **1990**, Yann LeCun [8] mettra en place les réseaux de neurones convolutifs qui reposent essentiellement sur le traitement des images. A l'époque, les premières images commençant à être numérisées, ce réseau de neurones convolutifs remis un coup d'accélération dans le développement de l'IA. C'est en **1997** que les réseaux de neurones récurrents furent élaborés qui sont à la base une variante au perceptron multicouche, dans le cadre de traiter des problèmes de séries temporelles. A nouveau, nous rentrons dans une phase de "gèle" pour la recherche en IA, car les dispositifs technologiques, à savoir les machines, les algorithmes n'étaient pas encore assez développés pour permettre à l'IA d'avancer.
C'est en **2012** que Geoffrey Hinton décida d'organiser une compétition nommée *ImageNet* qui avait pour intention de relancer la recherche en IA par les réseaux de neurones récurrents. Cette compétition a symbolisé l'envol du Deep Learning que nous avions défini au début de ce mémoire. De nos jours, nous continuons d'avancer en IA par la construction d'algorithmes de plus en plus complexe et par l'acquisition d'une base de données gigantesques. En effet, la plus grosse différence par rapport au début de l'IA, c'est le nombre de données. Aujourd'hui, l'ensemble du traitement de toutes ces données ont permis aux différents algorithmes de l'IA de progresser et nous ne sommes qu'au commencement, tellement l'ampleur des données que nous avons à traiter est colossale.

1. Warren Sturgis McCulloch, né en 1898 et mort en 1969 est un chercheur en neurologie américain.
2. Walter Pitts, né en 1923 et mort en 1969 est un scientifique américain étudiant la psychologie cognitive.
3. Alan Mathison Turing, né en 1912 et mort en 1954 est un mathématicien et cryptologue britannique.
4. Frank Rosenblatt né en 1928 et mort en 1971 était un psychologue américain qui travailla sur l'IA.
5. Marvin Lee Minsky, né en 1927 et mort en 2016 est un scientifique américain ayant travaillé sur le domaine des sciences cognitives et l'IA.
6. Seymour Papert, né en 1928 et mort en 2016 était un mathématicien et informaticien.
7. Geoffrey Hinton né en 1947, est un chercheur canadien spécialiste de l'IA et des réseaux de neurones artificiels.
8. Yann Le Cun, né en 1960 est un chercheur en IA et vision artificielle.

3. Fonctionnement des réseaux de neurones artificiels

3.1. Une première approche théorique

Il s'agit d'aborder le fonctionnement des réseaux de neurones artificiels pour en comprendre l'intérêt et l'usage en IA de part une première approche synthétique. Tout d'abord, nous partons d'une base de données que nous allons initier dans un algorithme, puis ces données vont être traitées par des fonctions que nous nommerons *cost functions*, qui ensuite nous sortirons un résultat, qui est donc la sortie de l'algorithme. Pour en revenir sur ces fonctions, il s'agit du principe d'agrégation (qui correspond en biologie à la somme de toutes les entrées du neurone avec les coefficients représentant l'activité synaptique), par exemple nous pouvons utiliser la fonction :

$$f(x) = w_1 \times x_1 + w_2 \times x_2$$

où x_1 et x_2 sont les données, et w_1, w_2 sont appelés les poids. A partir de cette équation, nous pouvons définir un biais nommé b qui a pour intérêt dans une étude statistique d'être considéré comme une caractéristique d'une question ou d'une démarche, qui a pour conséquence de générer des erreurs dans les résultats ou l'interprétation des résultats de l'étude en cours. Notre fonction peut s'écrire de la manière suivante :

$$f(x) = w_1 \times x_1 + w_2 \times x_2 + b$$

Ensuite, notre intérêt va d'être d'aborder une nouvelle phase lors de la programmation, il s'agit de la phase d'**activation**. Par exemple, on test si f est positive ou pas, pour affecter une valeur Y qui voudra 0 ou 1, on parle de la fonction *Heaviside*.

Depuis les années **1960**, la fonction *Heaviside* a été remplacée par des fonctions d'activitions plus intéressantes et performantes que nous allons décrire dès maintenant (voir figure 1).

— La fonction *Logisitique*, aussi appelé la *Sigmoïde* permet de retourner des résultats entre 0 et 1 de manière plus étendue, on ne passe pas de 0 à 1 de manière brutale, comme avec la fonction *Heaviside* qui elle, est une fonction de type escalier.

— La fonction *tanh* appelé tangente hyperbolique, permet de retourner des résultats entre -1 et 1 de manière étendue.

— La fonction *Softmax* peut être utilisée en machine learning pour convertir un score en probabilité dans un contexte de classification multi-classe. Nous ne l'aborderons pas plus dans cette étude.

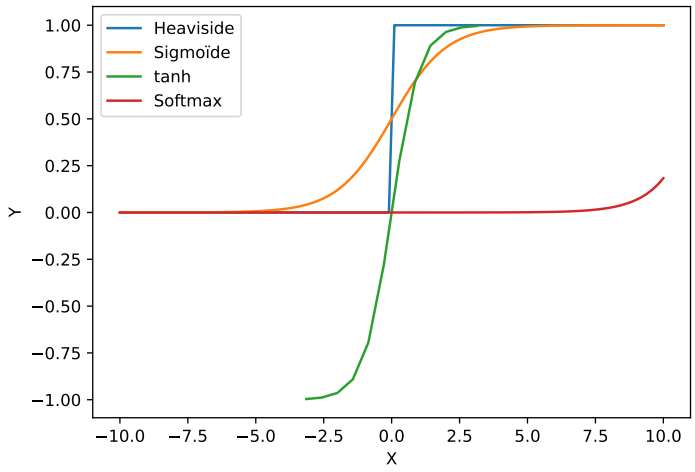

FIGURE 1 – Représentation des fonctions d'acitivations heaviside / Sigmoïde / tanh / Softmax

L'intérêt à l'IA d'utiliser ces algorithmes de traitement a pour but de les faire apprendre. C'est-à-dire que si nous connaissons de base la valeur finale Y, le but est de relancer l'algorithme en repartant de la sortie Y prédite pour revenir aux *cost function*, puis aux entrées.
Il s'agit en fait d'analyser les *cost functions* pour minimiser l'erreur que nous avons eue, entre notre valeur de sortie Y prédite et celle que nous connaissons de base. Cette minimisation est en fait la recherche en mathématiques du minimum des *cost functions* par **l'algorithme de la descente du gradient** (que nous écrirons par la suite directement la méthode du gradient).

Pour faire le lien avec les différents vocabulaire du Deep Learning, nous nommerons la **Forward Propagation** la phase qui symbolise le chemin des entrées des données jusqu'à leurs sorties et la **Back Propagation** symbolisant le chemin de la sortie des données jusqu'à leurs entrées.

3.2. Evaluation du modèle

En faisant le lien avec les stastitiques, nos fameuses *cost functions* vont pouvoir nous permettre d'évaluer notre modèle que nous avons mis en place. Il va nous falloir calculer son **maximum de vraisemblance** ce qui nous permettra de quantifier les erreurs effectuées par notre modèle. On rappel qu'il s'agit en faite de mesurer les distances entre les sorties de notre fonction d'activation et nos valeurs Y dont nous disposons. Nous nommerons cette vraisemblance par la fonction *Log Loss* défini par

$$LL = -\frac{1}{m}\sum_{i=0}^{m} y_i Log(a_i) + (1-y_i)Log(1-a_i). \tag{1}$$

avec

$$\begin{cases} m : \text{Nombre de données} \\ y_i : \text{Donnée n°i} \\ a_i : \text{Sortie n°i} \end{cases}$$

En effet, la vraisemblance va nous permettre de voir la plausibilité de notre étude vis à vis des vraies données dont nous disposons. Mais avant tout, il est temps de montrer d'où provient cette fonction qui nous permettra par la suite d'évaluer la perfomance de nos modèles.

Démonstration. En statistique, nous savons de base que la formule de vraisemblance est donnée par la fonction L :

$$L = \prod_{i=0}^{m} P(Y = y_i) = \prod_{i=0}^{m} a_i^{y_i}(1-a_i)^{1-y_i}.$$

Or, un problème va survenir à partir de cette formule, c'est le fait de multiplier plusieurs nombres qui ont une valeur entre 0 et 1 (exemple : $0,5 \times 0,5 = 0,25$) ce qui nous donne un résultat très proche de 0. Pour remédier à ce problème, nous allons introduire la fonction Log pour notre fonction L. Nous écrirons $Log(L) = LL$ par la suite.

$$Log(L) = Log(\prod_{i=0}^{m} a_i^{y_i}(1-a_i)^{1-y_i})$$

$$\iff LL = \sum_{i=0}^{m} Log(a_i^{y_i}) + Log((1-a_i)^{1-y_i})$$

$$\iff LL = \sum_{i=0}^{m} y_i Log(a_i) + (1-y_i)Log(1-a_i).$$

Ce qui nous donnera un résultat négatif, nous verrons d'ici quelques instants comment être sur d'avoir un résultat positif. Nous avons utilisé la fonction Log d'une part pour nous permettre d'avoir un résultat cochérent et d'autre part, car dorénavant nous allons chercher à maximiser notre fonction LL et la recherche du maximum de notre fonction LL correspond au maximum de notre vraisemblance.

Nous avons la formule suivante :

$$Argmax(L) = Argmax(LL).$$

Il est très simple de le constater par la superposition des graphes des fonctions L et LL. Malheureusement, il est très rare de chercher à maximiser une fonction en mathématiques, au contraire, on cherche à minimiser ! ce qui nous donne comme formule :

$$Maximer(f(x)) = Minimiser - f(x).$$

Le facteur $\frac{1}{m}$ à pour but de normaliser notre fonction LL et le facteur négatif "$-$" de nous permettre d'avoir un résultat positif à la fin.

La preuve de l'origine de notre fonction LL est donc achevée.

$$LL = -\frac{1}{m}\sum_{i=0}^{m} y_i Log(a_i) + (1-y_i)Log(1-a_i). \tag{1}$$

□

3.3. Descente de Gradient

Cette étape consite à ajuster les paramètres w et b de façon à minimiser les erreurs du modèle, c'est à dire à minimiser notre fonction LL.

Application de la méthode de l'algorithme de la descente de gradient donnée par les formules suivantes.

$$w_{t+1} = w_t - \alpha\frac{\partial L}{\partial w_t}. \tag{2}$$

$$b_{t+1} = b_t - \alpha\frac{\partial L}{\partial b_t}. \tag{3}$$

avec

$$\begin{cases} w_{t+1} : \text{Paramètre w à l'instant t+1} \\ w_t : \text{Paramètre w à l'instant t} \\ b_{t+1} : \text{Biais b à l'instant t+1} \\ b_t : \text{Biais b à l'instant t} \\ \alpha : \text{Pas d'apprentissage positif} \\ \frac{\partial L}{\partial w_t} : \text{Gradient à l'instant t} \end{cases}$$

Cependant, il est capital d'avoir une fonction **convexe** pour notre étude, ce qui est bien le cas de notre Log Loss (LL) qui est bien une fonction **convexe**. En effet, nous verrons plus tard que la dérivéee de la fonction Log Loss est croissante, ce qui prouve sa convexité.

A partir des formules précédentes, ce qui va nous interpeller n'est autre que le calcul des dérivéees de $\frac{\partial L}{\partial w_1}$, $\frac{\partial L}{\partial w_2}$ et $\frac{\partial L}{\partial b}$.

Démonstration. Pour commencer, voici toutes les formules que nous avons abordées et qui vont être utilisées dans les calculs suivants:

Fonction :

$$Z^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + b. \tag{4}$$

Fonction sigmoïde (activation) :

$$a(Z) = \frac{1}{1 + e^{-Z}}. \tag{5}$$

Fonction coût (LL) :

$$LL(a, y) = -\frac{1}{m} \sum_{i=0}^{m} y_i^{(i)} Log(a_i^{(i)}) + (1 - y_i^{(i)}) Log(1 - a_i^{(i)}). \tag{6}$$

Descente de gradient pour les paramètres w et b :

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t}. \tag{7}$$

$$b_{t+1} = b_t - \alpha \frac{\partial L}{\partial b_t}. \tag{8}$$

Nous allons développer notre dérivéee $\frac{\partial L}{\partial w_1}$ de la manière suivante :

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a} \times \frac{\partial a}{\partial Z} \times \frac{\partial Z}{\partial w_1}. \tag{9}$$

On constate que par élimination des facteurs communs, on retrouve bien notre expression de départ. Calculons séparement chaque terme de ce calcul.

Pour le premier terme, on dérive en fonction des coefficients a.

$$\frac{\partial L}{\partial a} = -\frac{1}{m}\sum_{i=1}^{m}\frac{y^{(i)}}{a^{(i)}} - \frac{1-y^{(i)}}{1-a^{(i)}}$$

Pour le second terme, on dérive notre sigmoïde en fonction de Z.

$$\frac{\partial a}{\partial Z} = \frac{e^{-Z}}{(1+e^{-Z})^2}$$

$$\iff \frac{\partial a}{\partial Z} = \frac{1}{(1+e^{-Z})} \times \frac{e^{-Z}}{(1+e^{-Z})}$$

$$\iff \frac{\partial a}{\partial Z} = a(Z) \times \left(\frac{1+e^{-Z}}{1+e^{-Z}} - \frac{1}{1+e^{-Z}}\right)$$

$$\iff \frac{\partial a}{\partial Z} = a(Z) \times (1-a(Z)).$$

Pour le dernier terme, on dérive notre fonction Z en fonction du poids $w1$.

$$\frac{\partial Z}{\partial w_1} = x_1.$$

Ce qui nous donne en rassemblant ces trois dérivéees partielles, la dérivéee de $\frac{\partial L}{\partial w_1}$ qui vaut :

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a} \times \frac{\partial a}{\partial Z} \times \frac{\partial Z}{\partial w_1}$$

$$\iff \frac{\partial L}{\partial w_1} = \left(-\frac{1}{m}\sum_{i=1}^{m}\frac{y^{(i)}}{a^{(i)}} - \frac{1-y^{(i)}}{1-a^{(i)}}\right) \times \Big(a(Z) \times (1-a(Z))\Big) \times x_1^{(i)}$$

$$\iff \frac{\partial L}{\partial w_1} = -\frac{1}{m}\sum_{i=1}^{m}\Big(y^{(i)}(1-a^{(i)}) - a^{(i)}(1-y^{(i)})\Big) \times x_1^{(i)}$$

$$\iff \frac{\partial L}{\partial w_1} = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - y^{(i)}a^{(i)} - a^{(i)} + a^{(i)}y^{(i)}) \times x_1^{(i)}$$

$$\iff \frac{\partial L}{\partial w_1} = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - a^{(i)}) \times x_1^{(i)}.$$

On trouve par les mêmes méthodes le calcul de la dérivéee de $\frac{\partial L}{\partial w_2}$:

$$\frac{\partial L}{\partial w_2} = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - a^{(i)}) \times x_2^{(i)}. \tag{10}$$

Puis,

$$\frac{\partial L}{\partial b} = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - a^{(i)}) = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)}). \tag{11}$$

Pour information, on voit bien que les dérivées partielles de notre fonction Log Loss sont croissantes ce qui prouve bien la convexité que nous avons évoquée quelques temps auparavant.

□

Il ne reste plus qu'à déterminer la valeur pour laquelle notre fonction *sigmoïde* atteint son extremum. Nous avons vu que sa dérivée était de la forme :

$$a'(Z) = \frac{e^{-Z}}{(1+e^{-Z})^2}$$

ce qui nous informe que $a'(Z) > 0$ comme le confirme le graphique suivant (figure 2).

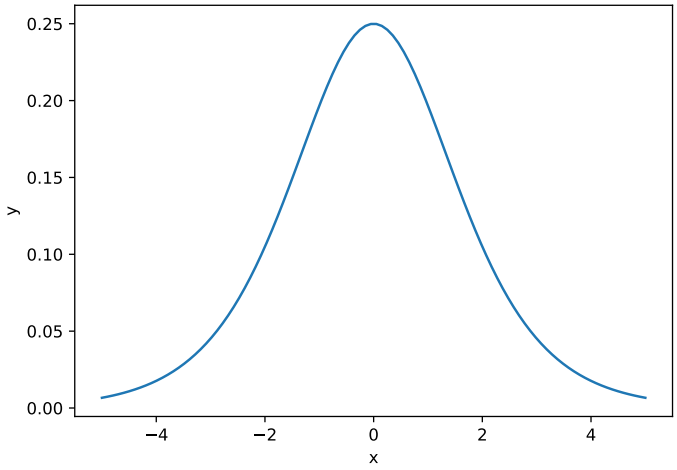

FIGURE 2 – Représentation de la dérivée de la fonction d'activation sigmoïde

Cependant, on peut voir une valeur maximale atteinte en $x = 0$. On calcule l'image de 0 par la fonction sigmoïde.

$$a'(0) = \frac{1}{1+e^0} \iff a'(0) = \frac{1}{2}$$

C'est à partir de la valeur $0,5$ que nous pourrons effectuer nos prédictions sur les jeux de données que nous exploiterons par la suite.

3.4. Vectorisation

Nous serons amenés à manipuler nos données et fonction à l'aide des outils de programmation, mais avant cela, une étape que l'on ne peut pas contourner, n'est autre que de vectoriser nos données et fonctions. Il s'agit en faite de mettre nos données dans des vecteurs, des matrices ou des tableaux à n-dimension afin que l'on puisse appliquer des opérations mathématiques sur l'ensemble de nos données de manière plus rapide. De par ce fait, de pouvoir manipuler de même nos fonctions et nos dérivées que nous avons présentées précédement.

On cherche d'abord à vectoriser notre équation :

$$Z^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + b. \qquad (4)$$

On commence par mettre nos données dans une matrice X de dimension m lignes et 2 colonnes que l'on notre par (m, 2) :

$$X = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ | & | \\ x_1^{(m)} & x_2^{(m)} \end{pmatrix}.$$

Ensuite, on range nos paramètres w_1 et w_2 dans une matrice W de dimension (2, 1) :

$$W = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}.$$

Puis, nos biais dans un vecteur b de dimension (m, 1) :

$$b = \begin{pmatrix} b \\ b \\ | \\ b \end{pmatrix}.$$

L'équation que l'on cherche à vectoriser peut donc s'écrire :

$$Z = X.W + b. \qquad (12)$$

Avec " . " la mutliplication matricielle.

On cherche à obtenir un vecteur Z de dimension (m, 1). On va vérifier que c'est bien le cas avec l'équation vectorisée d'au dessus.

Démonstration.

$$Z = X.W + b.$$

$$\iff Z = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ | & | \\ x_1^{(m)} & x_2^{(m)} \end{pmatrix} . \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} + \begin{pmatrix} b \\ b \\ | \\ b \end{pmatrix} .$$

On peut ici utiliser le produit matriciel, car la dimension de X est (m, 2) et celle de W est (2, 1), donc on obtiendra une matrice de dimension (m, 1).

$$\iff Z = \begin{pmatrix} w_1 x_1^{(1)} + w_2 x_2^{(1)} \\ w_1 x_1^{(2)} + w_2 x_2^{(2)} \\ | \\ w_1 x_1^{(m)} + w_2 x_2^{(m)} \end{pmatrix} + \begin{pmatrix} b \\ b \\ | \\ b \end{pmatrix} .$$

Maintenant, nous pouvons utiliser l'addition entre deux matrices qui sont de dimensions (m, 1) chacune, donc on obtiendra une matrice Z de dimension (m, 1).

$$\iff Z = \begin{pmatrix} w_1 x_1^{(1)} + w_2 x_2^{(1)} + b \\ w_1 x_1^{(2)} + w_2 x_2^{(2)} + b \\ | \\ w_1 x_1^{(m)} + w_2 x_2^{(m)} + b \end{pmatrix}$$

$$\iff Z = \begin{pmatrix} Z^{(1)} \\ Z^{(2)} \\ | \\ Z^{(m)} \end{pmatrix} .$$

On obtient bien un vecteur Z de dimension (m, 1).

□

De même, nous pouvons ranger nos données de sorties y_i dans un vecteur Y de dimension (m, 1) :

$$Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ | \\ y^{(m)} \end{pmatrix} .$$

Vectorisation de la fonction d'activation sigmoïde :

Rappel :

$$a^{(i)} = \frac{1}{1 + e^{-Z^{(i)}}} = \sigma(Z^{(i)}). \qquad (5)$$

Soit A la matrice d'activation de dimension (m, 1) qui représente la vectorisation des fonctions d'activations :

$$A = \begin{pmatrix} a^{(1)} \\ a^{(2)} \\ | \\ a^{(m)} \end{pmatrix} = \begin{pmatrix} \sigma(Z^{(1)}) \\ \sigma(Z^{(2)}) \\ | \\ \sigma(Z^{(m)}) \end{pmatrix} = \sigma \begin{pmatrix} Z^{(1)} \\ Z^{(2)} \\ | \\ Z^{(m)} \end{pmatrix} = \sigma(Z^{(i)}). \qquad (13)$$

Vectorisons notre fonction Log Loss (LL)

Rappel :

$$LL(a, y) = -\frac{1}{m} \sum_{i=0}^{m} y_i^{(i)} Log(a_i^{(i)}) + (1 - y_i^{(i)}) Log(1 - a_i^{(i)}). \qquad (6)$$

La vectorisaiton de LL nous donne :

$$LL = -\frac{1}{m} \sum_{i=0}^{m} Y Log(A) + (1 - Y) Log(1 - A). \qquad (14)$$

Nous rappellons que l'objectif est de trouver une valeur réelle après le calcul de notre fonction LL, c'est ce que nous allons montrer dès à présent.

Démonstration.

$$LL = -\frac{1}{m} \sum_{i=0}^{m} \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ | \\ y^{(m)} \end{pmatrix} Log \begin{pmatrix} a^{(1)} \\ a^{(2)} \\ | \\ a^{(m)} \end{pmatrix} + \left(1 - \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ | \\ y^{(m)} \end{pmatrix}\right) Log\left(1 - \begin{pmatrix} a^{(1)} \\ a^{(2)} \\ | \\ a^{(m)} \end{pmatrix}\right)$$

$$\iff LL = -\frac{1}{m} \sum_{i=0}^{m} \begin{pmatrix} y^{(1)} Log(a^{(1)}) + (1 - y^{(1)}) Log(1 - a^{(1)}) \\ y^{(2)} Log(a^{(2)}) + (1 - y^{(2)}) Log(1 - a^{(2)}) \\ | \\ y^{(m)} Log(a^{(m)}) + (1 - y^{(m)}) Log(1 - a^{(m)}) \end{pmatrix}$$

$$\iff LL = -\frac{1}{m} \sum_{i=0}^{m} \Big(y^{(1)} Log(a^{(1)}) + (1 - y^{(1)}) Log(1 - a^{(1)}) + ... + y^{(m)} Log(a^{(m)}) + (1 - y^{(m)}) Log(1 - a^{(m)}) \Big)$$

$\iff LL =$ Une valeur réel.

On trouve bien une valeur réelle à la fin du calcul de LL.

□

Il ne reste plus qu'à vectoriser nos gradients.

Rappel :

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t}. \qquad (7)$$

En terme de vecteur, nous avons :

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} - \alpha \begin{pmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \end{pmatrix}$$

$$W = W - \alpha \frac{\partial L}{\partial W}. \qquad (15)$$

Où le terme $\frac{\partial L}{\partial W}$ s'appelle le Jacobien.

Puis pour le biais, nous avons la formule suivante :

$$b = b - \alpha \frac{\partial L}{\partial b}. \qquad (16)$$

On vectorise notre Jacobien :

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \end{pmatrix}$$

$$\iff \frac{\partial L}{\partial W} = \begin{pmatrix} \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)}) \times x_1^{(i)} \\ \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)}) \times x_2^{(i)} \end{pmatrix}$$

$$\iff \frac{\partial L}{\partial W} = \frac{1}{m}\begin{pmatrix} \sum_{i=1}^{m}(a^{(i)} - y^{(i)}) \times x_1^{(i)} \\ \sum_{i=1}^{m}(a^{(i)} - y^{(i)}) \times x_2^{(i)} \end{pmatrix}$$

$$\iff \frac{\partial L}{\partial W} = \frac{1}{m}\begin{pmatrix} (a^{(1)} - y^{(1)})x_1^{(1)} + ... + (a^{(m)} - y^{(m)})x_1^{(m)} \\ (a^{(1)} - y^{(1)})x_2^{(1)} + ... + (a^{(m)} - y^{(m)})x_2^{(m)} \end{pmatrix}$$

$$\iff \frac{\partial L}{\partial W} = \frac{1}{m}\begin{pmatrix} x_1^{(1)} & x_1^{(2)} & ... & x_1^{(m)} \\ x_1^{(1)} & x_2^{(2)} & ... & x_2^{(m)} \end{pmatrix}\begin{pmatrix} (a^{(1)} - y^{(1)}) \\ (a^{(2)} - y^{(2)}) \\ | \\ (a^{(m)} - y^{(m)}) \end{pmatrix}.$$

On rappelle que :

$$X = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ | & | \\ x_1^{(m)} & x_2^{(m)} \end{pmatrix}.$$

Or, on remarque :

$$X^T = \begin{pmatrix} x_1^{(1)} & x_1^{(1)} & ... & x_1^{(m)} \\ x_2^{(2)} & x_2^{(2)} & ... & x_2^{(m)} \end{pmatrix}.$$

Donc

$$\frac{\partial L}{\partial W} = \frac{1}{m} X^T \begin{pmatrix} (a^{(1)} - y^{(1)}) \\ (a^{(2)} - y^{(2)}) \\ | \\ (a^{(m)} - y^{(m)}) \end{pmatrix}$$

$$\iff \frac{\partial L}{\partial W} = \frac{1}{m} X^T \left(\begin{pmatrix} a^{(1)} \\ a^{(2)} \\ | \\ a^{(m)} \end{pmatrix} - \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ | \\ y^{(m)} \end{pmatrix} \right)$$

$$\iff \frac{\partial L}{\partial W} = \frac{1}{m} X^T (A - Y).$$

De même, nous avons que :

$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} \left(\begin{pmatrix} a^{(1)} \\ a^{(2)} \\ | \\ a^{(m)} \end{pmatrix} - \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ | \\ y^{(m)} \end{pmatrix} \right)$$

$$\iff \frac{\partial L}{\partial b} = \frac{1}{m} \sum (A - Y).$$

Nous avons donc les formules suivantes :

$$\frac{\partial L}{\partial W} = \frac{1}{m} X^T (A - Y). \tag{17}$$

$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum (A - Y). \tag{18}$$

Ce qui achève nos vectorisations, dont nous allons exploiter par la suite à l'aide de nos programmes en langage python.

3.5. Représentation d'un réseau de neurone

En premier, de manière imagée la structure d'un neurone a une couche dans lequel nous avons nos entrées X_i, nos poids associés W_{ij}, d'une fonction modèle, puis d'une fonction d'activation et enfin une sortie Y_j.

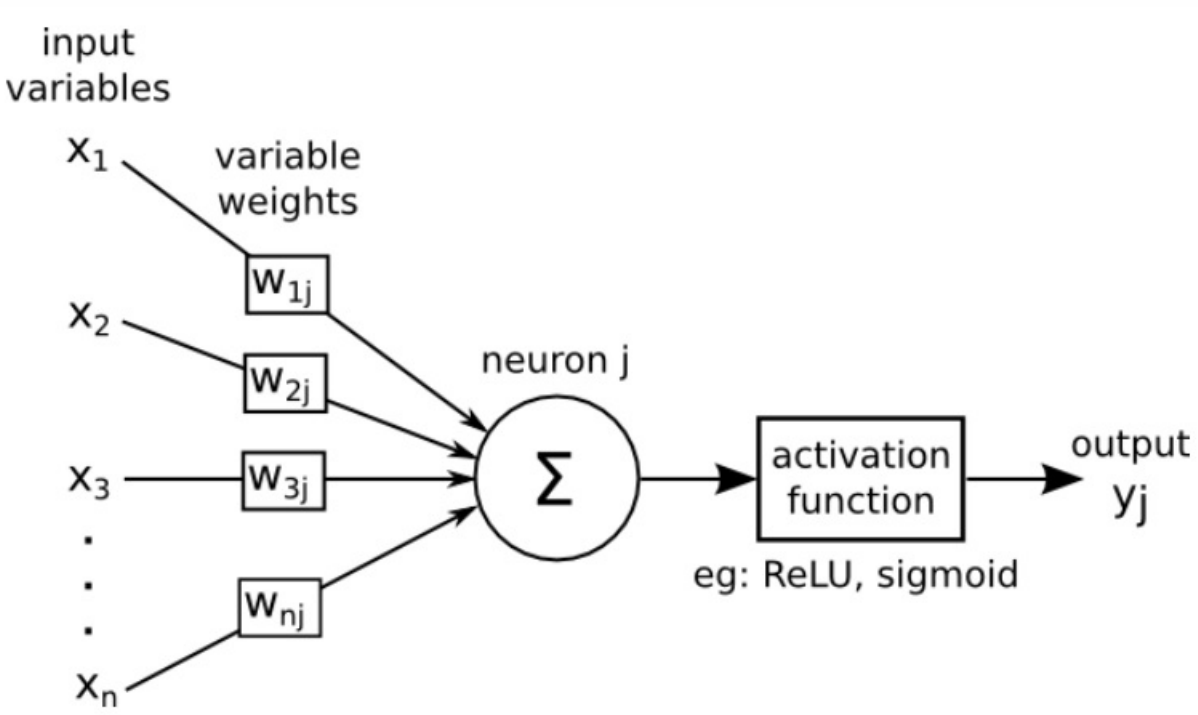

FIGURE 3 – Schéma d'un neurone à une couche composé de plusieurs entrées, d'une fonction d'activation et d'une sortie ; source: andrewjames turner.co.uk

On retrouve facilement notre fonction Z dans le cas où nous n'avons que deux entrées x_1 et x_2 associée respectivement aux poids w_1 et w_2 avec de plus un biais b ce qui nous donne $Z = w_1x_1 + w_2x_2 + b\$$.

Voici un autre exemple de réseau de neurone à deux couches cachées qui sont composées de quatre neurones chacun, de trois entrées et d'une sortie.

— Input layer : Représente les entrées de données.
— Hidden layer : Symbolise les couches du réseau de neurone dites "cachées" où se passe l'ensemble de nos calculs.
— Output layer : Couche de sortie des données.

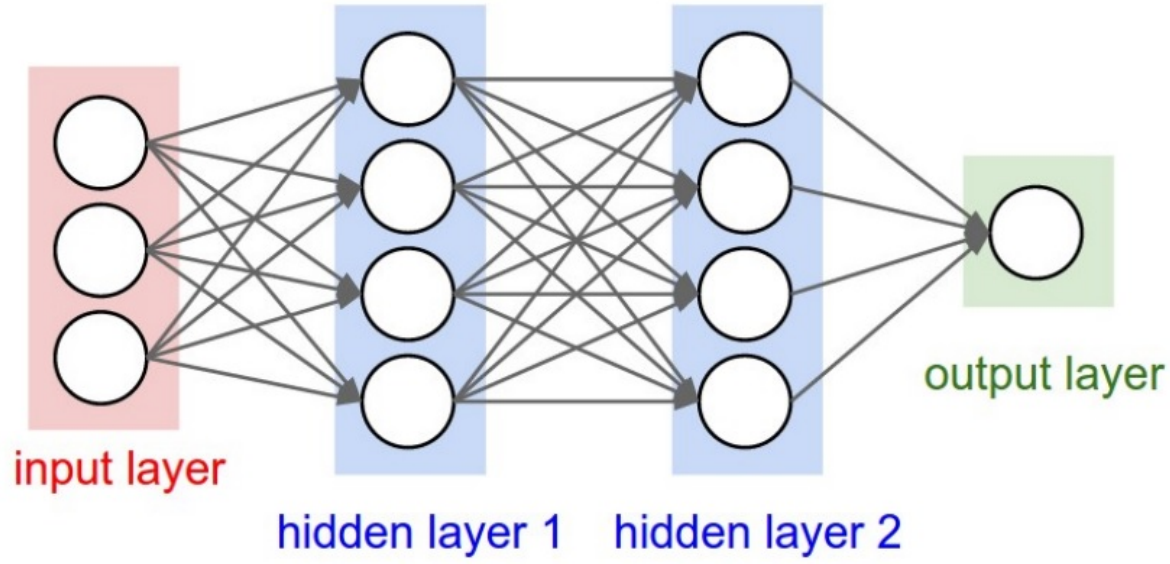

FIGURE 4 – Représentation réseau de neurone tiré de la revue scientifique "Neural Networks and Introduction to Deep Learning"

4. Réseau de neurone à une couche en langage python

Afin de mieux comprendre la théorie de façon mathématiques que nous venons d'expliquer, nous allons mettre en pratique nos vectorisations à l'aide d'un premier programme autour d'un réseau de neurone à une couche composé d'un neurone. Notons que l'ensemble des codes *pythons* que nous utiliserons dans ce mémoire ont été inspirés par les programmes *python* du Data Scientist Guillaume Saint-Cirgue.

4.1. Première base de données

Nous allons partir d'une base de donnée du package `make_blobs` de la librairie `sklearn.datasests`. Commençons par représenter nos données à l'aide d'un nuage de points. Notons qu'avec 'random_state=0', nous fixons le générateur de nombre aléatoire.

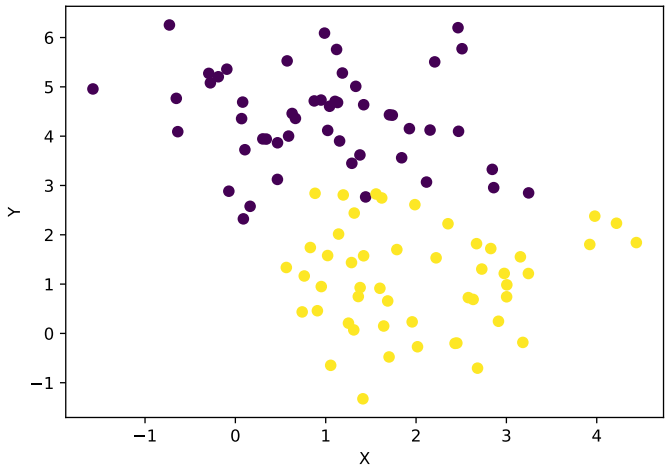

FIGURE 5 – Représentation de la première base de données à partir d'un nuage de points

Ici, nous allons chercher à séparer nos données en 2 groupes, à l'aide d'une régression linéaire telle que la fonction $Z = w_1x_1 + w_2x_2 + b$. Pour se faire, nous allons mettre en oeuvre toutes nos formules depuis le début. Le but étant de chercher quels sont les valeurs des coefficients w_1, w_2 et b qui permettent d'obtenir le meilleur modèle, à savoir celui qui minimise le plus nos erreurs. ## Initialisation

Commençons par initialiser notre paramètre w et notre biais b : On vérifie que nous travaillons avec les dimensions attendues, à savoir une dimension de W qui vaudra (2, 1) et biais de dimension 1.

```
# Initialiser les parametres W et b
def initialisation(X):
    W = np.random.randn(X.shape[1], 1) #Dim(W)=(2,1)
    b = np.random.randn(1) #Dim(b)=(1,) -> nb réel
    return (W, b)
W, b = initialisation(X)
```

```
## Dimension de W = (2, 1)
```

```
## Dimension de b = (1,)
```

4.2. Fonction d'activation

Nous continuons avec notre fonction modèle Z vu plus haut et notre fonction d'activition qui n'est autre que la *sigmoïde* qui doit avoir ici 100 réponses au total, car il y a 100 données (on utilise ici les formules (12) et (13)):

```
# Fonction sigmoïde
def sigmoide(X, W, b):
    Z = X.dot(W) + b
    A = 1 / (1 + np.exp(-Z))
    return A
A = sigmoide(X,W,b)
```

```
## Dimension de A = (100, 1)
```

4.3. Foncion coût

Il est temps de s'attaquer à notre fonction coût qui n'est autre que notre fonction Log Loss, pour rappel, nous attendons une valeur réelle comme résultat (on utilise la formule (14)) :

```
# Fonction coût
def log_loss(A, Y):
    return 1 / len(Y) * np.sum(-Y * np.log(A) - (1 - Y) * np.log(1 - A))
LL = log_loss(A, Y)
```

```
## La valeur de Log Loss = 2.6572510657259683
```

$1/len(Y)$ représente notre facteur $1/m$ avec $len(Y) = m$ donc le nombre total de données.

4.4. Calcul des gradients

On définit nos gradients, comme nous l'avions fait lors de l'étape de vectorisation, on cherche à vérifier que la dimension de dW soit la même que celle de W et que db soit bien une valeur réelle.

On implémente ici nos formules (17) et (18) :

```
# Gradient
def gradients(A, X, Y):
    dW = 1 / len(Y) * np.dot(X.T, A - Y) #Jacobien
    db = 1 / len(Y) * np.sum(A - Y)
    return (dW, db)
dW, db = gradients(A, X, Y)
```

```
## Dimension de dW = (2, 1)
```

```
## Valeur de db = 0.3999064280036948
```

4.5. Méthode de la descente de gradient

Nous pouvons mettre en place notre méthode de la descente de gradient qui va nous permettre de mettre à jour nos paramètres W et b où nous vérifions que les dimensions soient bien les mêmes qu'au début (formule utilisées (15) et (16) et alpha un petit pas positif) :

```
# Mise à jour de nos paramètres
def update(dW, db, W, b, alpha):
    W = W - alpha * dW
    b = b - alpha * db
    return (W, b)
W, b = update(dW, db, W, b, 0.01)
```

```
## Dimension de W = (2, 1)
```

```
## Dimension de b = (1,)
```

4.6. Prédiction

Une dernière étape symbolisant l'intérêt de notre programme, celui de prédire. Cela servira dans le programme principal quand il faudra prédire à quel moment notre fonction d'activation nous renvoie un résultat, à savoir ici quand la valeur de la fonction est plus grande que 0,5. La valeur 0,5 représente la valeur pour laquelle, nous avons que la dérivée de notre fonction *sigmoïde* admet son maximum.

En attendant, vérifions que la dimension de A nous renvoies bien notre nombre de données (ici 100) :

```
# Prédire la répartition des données à partir du modèle choisi (ici la Sigmoïde)
def predict(X, W, b):
    A = sigmoide(X, W, b)
    return A >= 0.5
A = predict(X, W, b)
```

```
## Dimension de A = (100, 1)
```

4.7. Fonction principal

Nous sommes arrivés au stade de notre fonction principale, qui va nous permettre de faire appel à toutes les fonctions que nous venons de définir, à savoir, notre réseau de neurone à une couche. Nous rajouter l'option n_iter pour le nombre de fois où nous lançons le programme.

Testons ce programme avec nos données et un pas alpha de 0.1.

```
## La precision du modele est de 90.0 %
## Valeurs des poids W1, W2 et du biais b :
```

```
## W1 =  [1.36922046]
## W2 =  [-1.28217318]
## b =  [0.97098196]
```

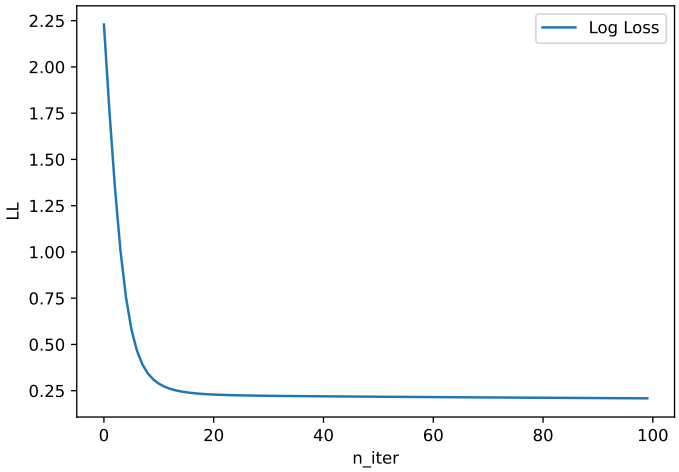

FIGURE 6 – Evolution de la fonction Log Loss sur l'apprentissage du réseau de neurone

On voit bien que notre fonction coût diminiue au fur et à mesure des itérations, elle se stabilise même, ce qui nous montre que l'apprentissage du réseau est terminé.

4.8. Frontière de décision

4.8.1 A partir des prédictions de notre réseau

On trace alors la frontrière de décision qui est en faite la régression du modèle à partir des paramètres que nous avons trouvé dans notre réseau de neurone. En effet, notre réseau de neurone nous a permis de choisir les meilleurs coefficients w_1, w_2 et b tel que la perfomance du modèle soit le plus proche de 100%. Voici donc la régression linéaire de cet exemple de jeux de données avec une perfomance de 90 % à partir de notre réseau de neurone.

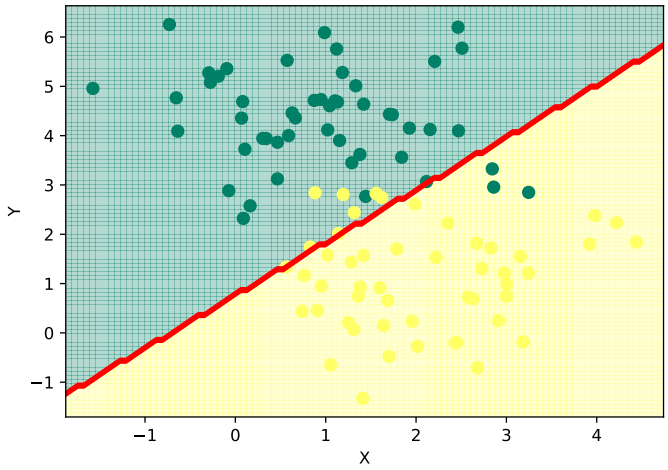

FIGURE 7 – Résultat de la prédiction de notre réseau de neurone pour notre première base de données permettant de séparer nos données en deux classes différentes

Notre fonction modèle Z s'écrit :

```
## Z = [1.36922046] * x1 + [-1.28217318] * x2 + [0.97098196]
```

4.8.2 Sans passer par la prédiction

Il s'agit d'un exemple très simple, nous pouvions de base déterminer notre droite de régression à l'aide d'une simple formule mathématiques. Il suffit de poser

$$x_2 = \frac{-w_1 \times x_1 - b}{w_2}$$

En effet, pour résoudre cette catégorisation de ce jeu de données en 2 classes, il suffisait de résoudre $Z = 0$.

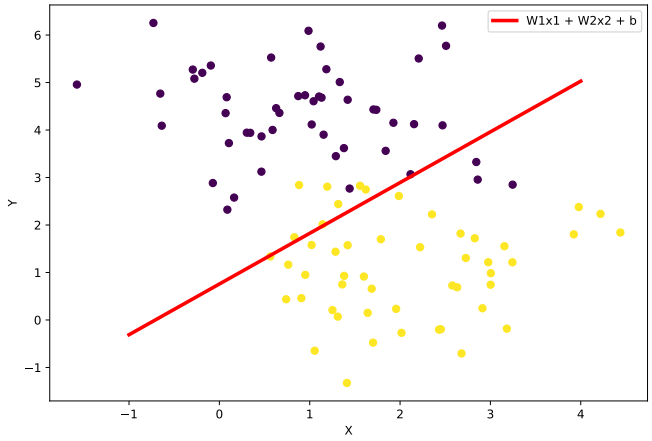

FIGURE 8 – Résultat de la régression linéaire standard sans passer par notre réseau de neurone

On constate que nous avons la même droite que par notre réseau.

5. Réseau de neurone à deux couches

Nous venons de voir qu'il était très simple de prévoir un modèle à un neurone pour une simple régression linéaire, mais dans notre vie de tous les jours, tout n'est pas aussi simple. C'est pour cela que la régression polynomiale existe comme la fonction $w_1x_1^2 + w_2x_2^2 + w_3x_1 + w_4x_2 + b$, dans le but d'améliorer nos modèles et pouvoir prédire avec de meilleurs perfomances. Dans cette partie, nous allons mettre en oeuvre la théorie d'un réseau de neurone à deux entrées, composées de deux couches, mais avec cette fois-ci deux neurones sur la première couche et un seul neurone sur la deuxième couche symbolisant la sortie.

On pose :

$$\begin{cases} W_{i,j} : \text{Paramètre assoicié au neurone i et provenant de l'entrée j} \\ b_i : \text{Biais associé au neurone i} \\ n^{[0]} : \text{Nombre d'entrée dans la couche [0]} \\ n^{[1]} : \text{Nombre de neurones dans la couche [1]} \\ n^{[2]} : \text{Nombre de neurones dans la couche [2]} \end{cases}$$

On note que plus le réseau est profond (beaucoup de couches et de neurones), plus il est capable d'apprendre des choses de plus en plus complexe, mais cela rend l'apprentissage très long. Comme nous l'avons vu avec notre première expérience en *python* durant la partie précédente, pour implémenter nos modèles, il n'est pas pratique d'écrire les équations de chaque neurone. C'est pour cela que nous choisissons de vectoriser nos équations avant de les programmer.

5.1. Forward Propagation à deux couches

On pose :

$$X = \begin{pmatrix} x_1^{(1)} & \dots & x_1^{(m)} \\ x_2^{(1)} & \dots & x_2^{(m)} \end{pmatrix}.$$

$$Y = \begin{pmatrix} y^{(1)} & \dots & y^{(m)} \end{pmatrix}.$$

Où X est de dimension ($n^{[0]}$, m) et Y de dimension (1, m)

5.1.1 Vectorisation des équations de la première couche

$$Z^{[1]} = W^{[1]}.X + b^{[1]}. \tag{19}$$

Avec

$$b^{[1]} = \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \end{pmatrix}.$$

et

$$W^{[1]} = \begin{pmatrix} w_{11}^{[1]} & w_{12}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} \end{pmatrix}.$$

$$A^{[1]} = \frac{1}{1 + e^{-Z[1]}}. \tag{20}$$

De plus,

$$\begin{cases} W^{[1]} : \text{Dimension } (n^{[1]}, n^{[0]}) \\ b^{[1]} : \text{Dimension } (n^{[1]}, 1) \\ Z^{[1]}, A^{[1]} : \text{Dimension } (n^{[1]}, m) \end{cases}$$

5.1.2 Vectorisation des équations de la deuxième couche

$$Z^{[2]} = W^{[2]}.A^{[1]} + b^{[2]}. \tag{21}$$

Avec

$$b^{[2]} = \begin{pmatrix} b_1^{[2]} \\ b_2^{[2]} \end{pmatrix}.$$

et

$$W^{[2]} = \begin{pmatrix} w_{11}^{[2]} & w_{12}^{[2]} \\ w_{21}^{[2]} & w_{22}^{[2]} \end{pmatrix}.$$

$$A^{[2]} = \frac{1}{1 + e^{-Z[2]}}. \tag{22}$$

De plus,

$$\begin{cases} W^{[2]} : \text{Dimension } (n^{[2]}, n^{[1]}) \\ b^{[2]} : \text{Dimension } (n^{[2]}, 1) \\ Z^{[2]}, A^{[2]} : \text{Dimension } (n^{[2]}, m) \end{cases}$$

5.1.3 Vectorisation Log Loss

$$LL = -\frac{1}{m}\sum_{i=0}^{m} YLog(A^{[2]}) + (1 - Y)Log(1 - A^{[2]}). \tag{23}$$

5.2. Back Propagation à deux couches

On rappel, qu'il s'agit de revenir sur nos pas, donc on refait le chemin inverse de la *Forward Propagation*. L'intérêt ici va d'être de calculer les dérivéees suivantes : $\frac{\partial L}{\partial W^{[2]}}$, $\frac{\partial L}{\partial b^{[2]}}$ et $\frac{\partial L}{\partial W^{[1]}}$, $\frac{\partial L}{\partial b^{[1]}}$.

On reprend le même principe que nous avions vu lors des calculs de dérivées de la back propagation, c'est à dire que pour chaque dérivée, nous allons les développer de manière à trouver leurs formules plus aisement. Nous commençons par les dérivées de la deuxième couche, puis celle de la première couche.

5.2.1 Deuxième couche

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial W^{[2]}}.$$

On pose dZ_2 qui sera de dimension $(n^{[2]}, m)$:

$$dZ_2 = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}}$$

$$\iff dZ_2 = -\frac{1}{m}\Big(\sum_{i=1}^{m} \frac{Y}{A^{[2]}} - \frac{1-Y}{1-A^{[2]}}\Big)\Big(A^{[2]}(1-A^{[2]})\Big)$$

$$\iff dZ_2 = -\frac{1}{m}\Big(\sum_{i=1}^{m} Y(1-A^{[2]}) - (1-Y)A^{[2]}\Big)$$

$$\iff dZ_2 = \frac{1}{m}\Big(\sum_{i=1}^{m} A^{[2]} - Y\Big)$$

Donc

$$dZ_2 = \frac{1}{m}\Big(\sum_{i=1}^{m} A^{[2]} - Y\Big) \tag{24}$$

Or $A^{[2]}$ est de dimension $(n^{[2]}, m)$, mais Y est de dimension (1, m). Pas d'inquiétude, nous pouvons utiliser le **Broadcasting** sur Y qui va donc étendre sa dimension, c'est à dire avoir une dimension du type $(n^{[2]}, m)$. Les dimensions étant vérifiées, nous pouvons valider la formule de dérivée pour dZ_2.

Nous avons donc que :

$$\frac{\partial L}{\partial W^{[2]}} = dZ_2 \times \frac{\partial Z^{[2]}}{\partial W^{[2]}}$$

$$\iff \frac{\partial L}{\partial W^{[2]}} = dZ_2 \times A^{[1]}$$

Comme $\frac{\partial L}{\partial W^{[2]}}$ est de dimension $(n^{[2]}, n^{[1]})$ tant dis que ∂Z_2 et $A^{[1]}$ sont de dimension respectivement $(n^{[2]}, m)$ et $(n^{[1]}, m)$. Il n'est donc pas possible d'utiliser le produit matriciel, car les dimensions ne sont pas bonnes. Par contre, on remarque que si on prend la transposée de la matrice $A^{[1]}$, alors nous obtiendrons une matrice $(A^{[1]})^T$ qui sera de dimension $(m, n^{[1]})$. Grâce à cela, nous pouvons utiliser le produit matriciel entre dZ_2 et $(A^{[1]})^T$ ce qui donne comme résultat que :

$$\frac{\partial L}{\partial W^{[2]}} = dZ_2 \times (A^{[1]})^T. \tag{25}$$

Il ne nous reste que le calcul de $\frac{\partial L}{\partial b^{[2]}}$ qui sera de dimension $(n^{[2]}, 1)$:

$$\frac{\partial L}{\partial b^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial b^{[2]}}.$$

$$\iff \frac{\partial L}{\partial b^{[2]}} = dZ_2 \times \frac{\partial Z^{[2]}}{\partial b^{[2]}}$$

$$\iff \frac{\partial L}{\partial b^{[2]}} = dZ_2 \times 1$$

Nous avons bien un résultat de dimension $(n^{[2]}, 1)$ donc :

$$\frac{\partial L}{\partial b^{[2]}} = dZ_2 = \frac{1}{m}\Big(\sum_{i=1}^{m} A^{[2]} - Y\Big). \tag{26}$$

5.2.2 Première couche

Dqns cette première couche, les méthodes de calculs des dérivées vont être très similaire à ceux de la deuxième couche.

$$\frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial W^{[1]}}.$$

On pose dZ_1 qui sera de dimension $(n^{[1]}, m)$:

$$dZ_1 = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}}$$

$$\iff dZ_1 = dZ_2 \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}}$$

$$\iff dZ_1 = dZ_2 \times W^{[2]} \times A^{[1]}(1 - A^{[1]}).$$

Avec

$$\begin{cases} dZ_2 : \text{Dimension } (n^{[2]}, m) \\ W^{[2]} : \text{Dimension } (n^{[2]}, n^{[1]}) \\ A^{[1]}(1 - A^{[1]}) : \text{Dimension } (n^{[1]}, m) \end{cases}$$

Nous avons le même problème qu'avant au niveau des dimensions pour le produit matriciel entre dZ_2 et $W^{[2]}$. On recommence la même astuce que toute à l'heure, on reprend la transposée de $W^{[2]}$ et on interchange dZ_2 et $W^{[2]}$ ce qui nous donne une mutliplication matricielle de dimension $(n^{[1]}, m)$. Il ne reste que la multiplication terme à terme entre $(W^{[2]})^T.dZ_2$ et $A^{[1]}(1 - A^{[1]})$ qui donnera une matrice de dimension $(n^{[1]}, m)$, c'est ce que nous attendions du résultat de dZ_1.

Donc :

$$dZ_1 = (W^{[2]})^T.dZ_2 \times A^{[1]}(1 - A^{[1]}). \tag{27}$$

Remarque : Le point entre $(W^{[2]})^T$ et dZ_2 symbolise le produit matriciel et le fois entre $(W^{[2]})^T.dZ_2$ et $A^{[1]}(1 - A^{[1]})$ symbolise un produit de terme à terme.

Il ne reste plus qu'à calculer $\frac{\partial L}{\partial W^{[1]}}$ et $\frac{\partial L}{\partial b^{[1]}}$ qui seront de dimension respectivement $(n^{[1]}, n^{[0]})$ et $(n^{[1]}, 1)$.

$$\frac{\partial L}{\partial W^{[1]}} = dZ_1 \times \frac{\partial Z^{[1]}}{\partial W^{[1]}}$$

$$\iff \frac{\partial L}{\partial W^{[1]}} = dZ_1 \times X.$$

Toujours avec nos dimensions, le même soucis persiste, à savoir que nous avons dZ_1 qui est de dimension $(n^{[1]}, m)$ et X de dimension $(n^{[0]}, m)$. L'astuce reste la même, on va prendre la transposée de X qui sera donc de dimension $(m, n^{[0]})$, donc le produit matricel nous donnera un résultat de dimension $(n^{[1]}, n^{[0]})$.

Alors :

$$\frac{\partial L}{\partial W^{[1]}} = dZ_1 \times X^T. \tag{28}$$

Puis,

$$\frac{\partial L}{\partial b^{[1]}} = dZ_1 \times \frac{\partial Z^{[1]}}{\partial b^{[1]}}.$$

Donc,

$$\frac{\partial L}{\partial b^{[1]}} = dZ_1. \tag{29}$$

Avec dZ_1 qui est bien de dimension $(n^{[1]}, 1)$.

5.2.3 Méthode de gradient pour les deux couches

On pose $dW^{[1]} = \frac{\partial L}{\partial W^{[1]}}$ et $dW^{[2]} = \frac{\partial L}{\partial W^{[2]}}$, puis $db^{[1]} = \frac{\partial L}{\partial b^{[1]}}$ et $db^{[2]} = \frac{\partial L}{\partial b^{[2]}}$.

$$W^{[1]} = W^{[1]} - \alpha \times dW^{[1]}. \tag{30}$$

$$b^{[1]} = b^{[1]} - \alpha \times db^{[1]}. \tag{31}$$

$$W^{[2]} = W^{[2]} - \alpha \times dW^{[2]}. \tag{32}$$

$$b^{[2]} = b^{[2]} - \alpha \times db^{[2]}. \tag{33}$$

5.3. Deuxième base de données en langage python

Nous allons partir d'une base de donnée du package `make_circles` de la librairie `sklearn.datasests`. On commence par importer nos données et les représenter par un nuage de points (figure 9).

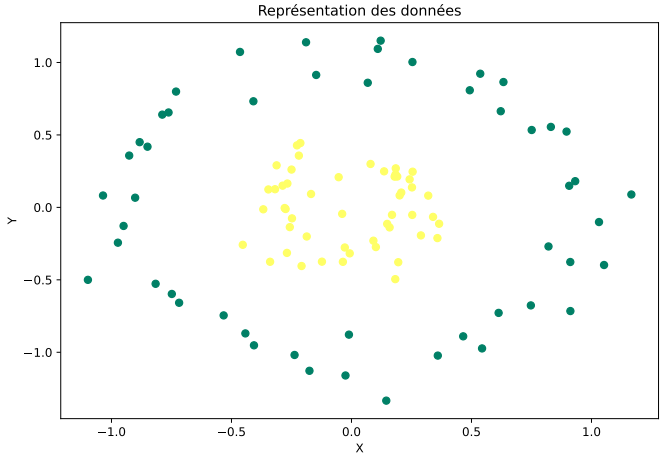

FIGURE 9 – Représentation de la deuxième base de données à partir d'un nuage de points

Pour la suite, nous reprenons exactement les mêmes étapes que lors du premier programme, cependant nous modifions au fur et à mesure nos fonctions pour qu'elles soient adaptées à notre réseau de neurone à deux couches.

On commence par initialiser nos paramètres, ici nous rappelons que la couche $n^{[0]}$ représente nos entrées, notre couche $n^{[1]}$ représente la couche cachée et $n^{[2]}$ la couche de sortie. On parle de deux couches, car les calculs vont être pratiqués sur les couches $n^{[1]}$ et $n^{[2]}$. Nous allons avoir donc 2 matrices de poids $W^{[1]}$ et $W^{[2]}$, ainsi que deux biais $b^{[1]}$ et $b^{[2]}$ à optimiser pour obtenir un modèle ayant une perfomance proche des 100% permettant de séparer au mieux nos deux classes de points.

Après avoir paramétré nos poids et biais, on enchaine avec l'implémentation de nos fonctions modèles $Z^{[1]}$ et $Z^{[2]}$, puis d'activations $A^{[1]}$ et $A^{[2]}$. On utilise nos formules (19), (20), (21) et (22) qui sont respectivement la fonction modèle et d'activation de la première couche et ceux de la deuxième couche.

On utilise ensuite notre fonction Log Loss (formule (23)) qui nous servira encore de fonction coût permettant d'évaluer notre modèle. On rappel qu'on attend de la fonction LL une valeur réelle.

On continue par l'introduction de nos dérivées par l'intermédiaire des formules (24), (25) et (26) pour la deuxième couche, puis les formules (27), (28) et (29) pour la première couche.

On utilise la méthode de gradient à l'aide des formules (30) et (31) pour la mise à jour des paramètres $W1$ et $b1$ de la première couche, ainsi que les formules (32) et (33) pour ceux de la deuxième couche. De plus, on incorpore encore un pas alpha réel positif.

Il est maintenant temps d'utiliser l'ensemble de nos fonctions à l'aide d'un programme principal. Nous allons par la suite executer ce programme avec diverses paramètres, notamment en faisant varier le nombre de neurones sur la couche cachée $n^{[1]}$ pour nous permettre de voir qu'elle sera le meilleur modèle ayant la meilleure perfomance d'apprentissage. On restera avec un pas **alpha de 0.1** et **1000 itérations**.

```
## 2 neurones sur la couche caché n1
## La precision du modèle est de 87.0 %
```

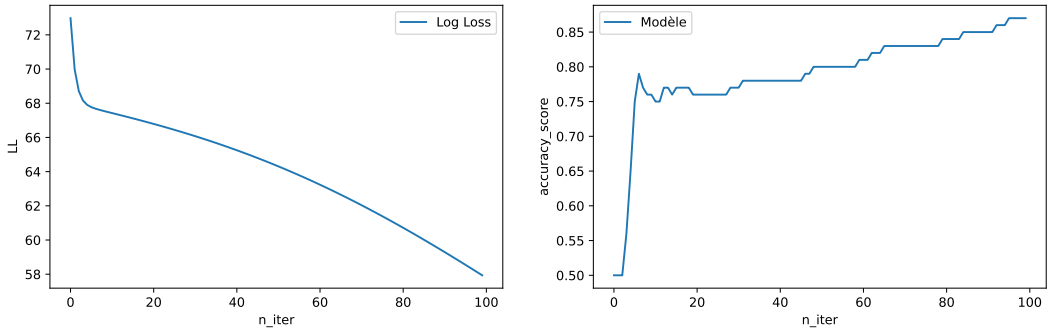

FIGURE 10 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

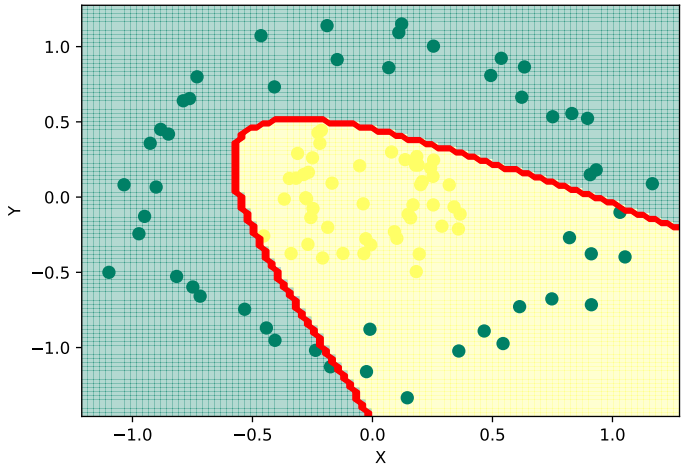

FIGURE 11 – Régression obtenue pour 2 neurones sur la couche cachée n1 pour notre deuxième base de données

On remarque que notre fonction LL (figure 10 coté gauche) ne se stabilise pas, ce qui nous donne un apprentissage incomplet (figure 10 coté droit).

```
## 8 neurones sur la couche caché n1
## La precision du modèle est de 99.0 %
```

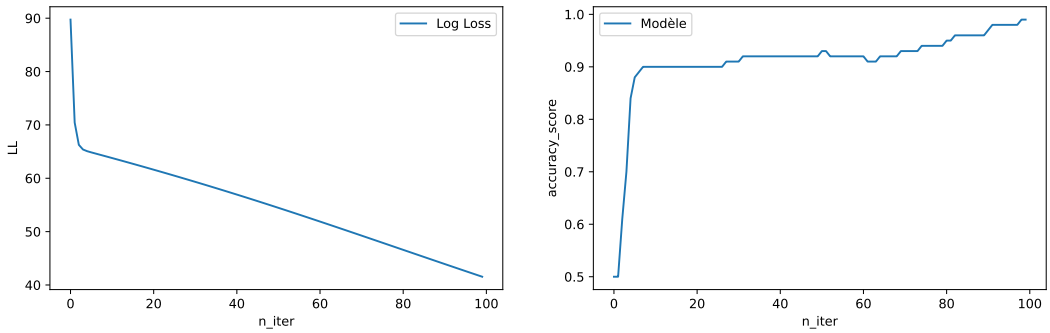

FIGURE 12 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

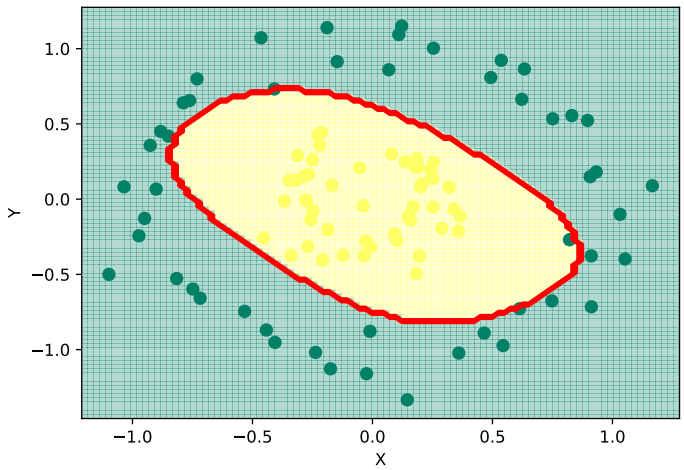

FIGURE 13 – Régression obtenue pour 8 neurones sur la couche cachée n1 pour notre deuxième base de données

La fonction LL n'est toujours pas stablisé (figure 12 coté gauche), mais on remarque que l'apprentissage est meilleur (figure 12 coté droit).

```
## 16 neurones sur la couche caché n1
## La precision du modèle est de 99.0 %
```

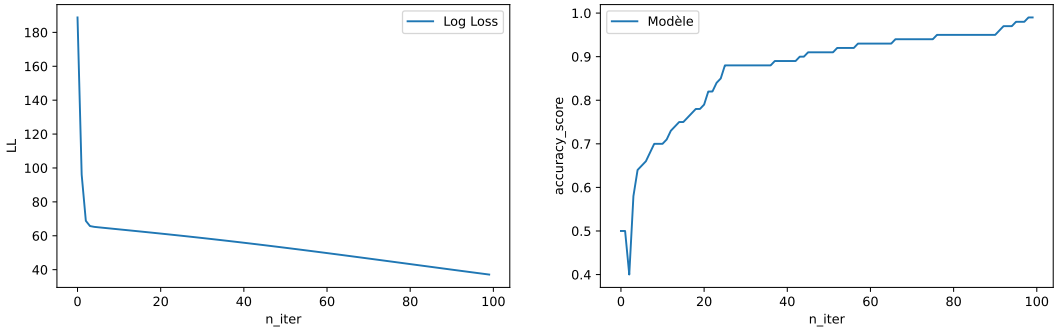

FIGURE 14 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

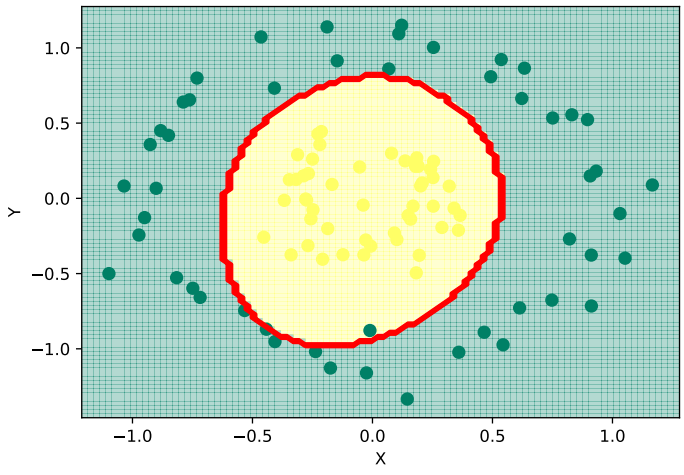

FIGURE 15 – Régression obtenue pour 16 neurones sur la couche cachée n1 pour notre deuxième base de données

A partir de 16 neurones sur la couche $n^{[1]}$, notre fonction LL se stabilise comme nous pouvons le voir sur la figure 14 partie gauche, ce qui se voit aussi sur la perfomance de l'apprentissage (figure 14 coté droit) qui elle, approche les 100%.

```
## 32 neurones sur la couche caché n1
## La precision du modèle est de 100.0 %
```

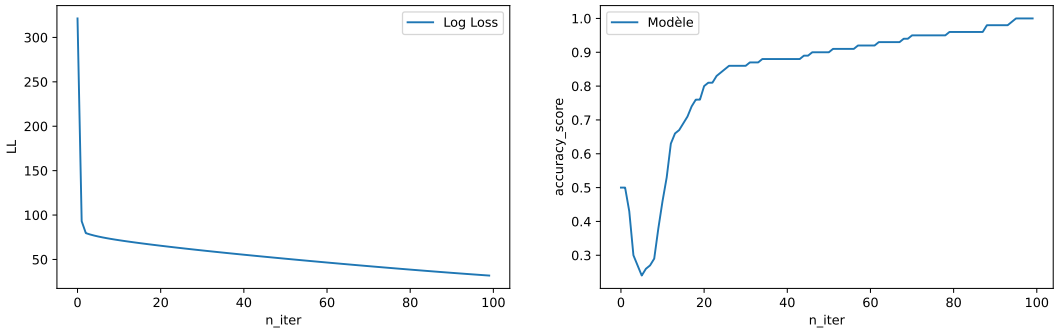

FIGURE 16 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

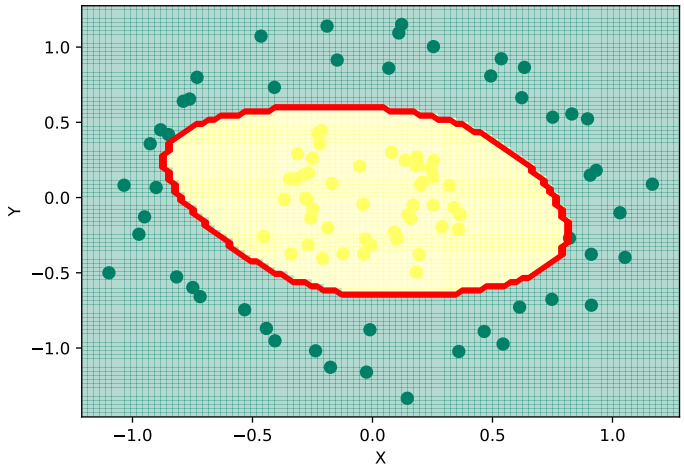

FIGURE 17 – Régression obtenue pour 32 neurones sur la couche cachée n1 pour notre deuxième base de données

On constate que plus le nombre de neurones augementent sur notre couche $n^{[1]}$, plus la peformance de notre modèle devient meilleur. En effet, les neurones réseaux de neurones semblent plus performant dès qu'ils sont composés de plusieurs neurones sur les couches cachées.

5.4. Retours sur la première base de données en python

Il s'agit ici de montrer que la perfomance de notre modèle de notre première base de données que nous avons exploitées toute à l'heure est meilleur avec un modèle qui est non-linéaire.

```
## La precision du modele est de 92.0 %
```

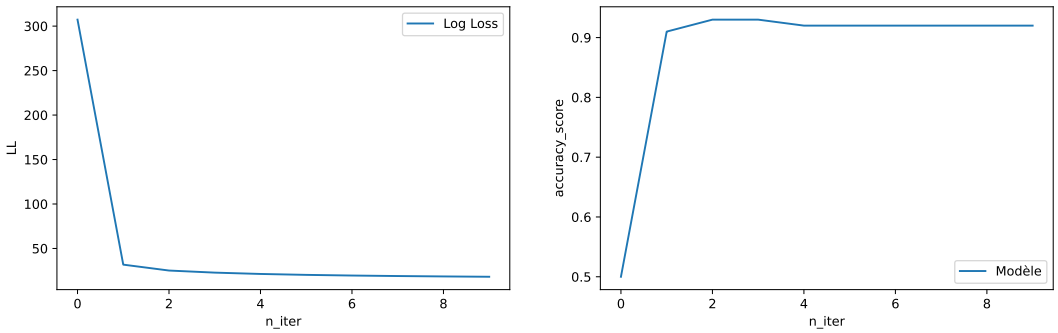

FIGURE 18 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

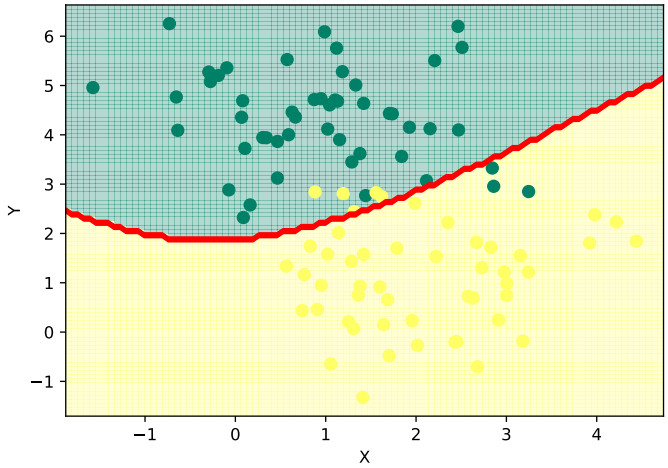

FIGURE 19 – Régression bbtenue pour 28 neurones sur la couche n1 pour notre première base de donnée

Notre perfomance s'améliore comme nous pouvons le constater (figure 18 partie gauche), ce qui correspond à nos attentes. On passe ici à une perfomance de 92% au lieu de 90% auparavant. Bien évidement, on peut toujours améliorer l'apprentissage, mais mieux vaut-il rajouter des neurones ou des couches dans notre réseau de neurones ? c'est ce que nous allons aborder dès à présent.

6. Réseau de neurone à c couche(s) et à n neurone(s)

Après plusieurs approches et mise en pratique par l'intermédiaire du logiciel *Python*, nous pouvons dorénavant généraliser nos différentes formules, dérivées et vectorisations sur n'importe quel type de réseau de neurone. C'est avec *python* que nous allons créer un programme nous permettant de choisir le nombre de couches cachées, ainsi que le nombre de neurones par couche cachées que nous souhaitons, puis d'en faire l'analyse.

Pour se faire, nous rapperelons les différents résultats des parties précédentes au fur et à mesure pour nous permettre de faire le lien plus facilement avec les formules générales. Nous présenterons en premier les formules pour un réseau à trois couches cachées, puis à c couches cachées.

6.1. Dimension des paramètres

$$\text{Première couche : } \begin{cases} W^{[1]} : \text{Dimension } (n^{[1]}, n^{[0]}) \\ b^{[1]} : \text{Dimension } (n^{[1]}, 1) \end{cases}$$

$$\text{Deuxième couche : } \begin{cases} W^{[2]} : \text{Dimension } (n^{[2]}, n^{[1]}) \\ b^{[2]} : \text{Dimension } (n^{[2]}, 1) \end{cases}$$

$$\text{Troisième couche : } \begin{cases} W^{[3]} : \text{Dimension } (n^{[3]}, n^{[2]}) \\ b^{[3]} : \text{Dimension } (n^{[3]}, 1) \end{cases}$$

$$\text{c - ième couche : } \begin{cases} W^{[c]} : \text{Dimension } (n^{[c]}, n^{[c-1]}) \\ b^{[c]} : \text{Dimension } (n^{[c]}, 1) \end{cases}$$

En exemple, prenons un réseau de neurones avec quatre couches cachées composé de :
- 2 Neurones sur la 1ère couche
- 12 Neurones sur la 2nd couche
- 28 Neurones sur la 3ème couche
- 16 Neurones sur la 4ème couche

A l'aide de *python*, nous pouvons implémenter notre formule générale pour vérifier les dimensions de l'exemple du réseau de neurones ci-dessus.

```
## W1 (12, 2)
## b1 (12, 1)
## W2 (28, 12)
## b2 (28, 1)
## W3 (16, 28)
## b3 (16, 1)
```

On constate bien que les dimensions sont correctes et suivent notre formule des **c-èmes couches** établi précédement.

6.2. Forward Propagation à c couche(s) et n neurone(s)

A partir des formules (19), (20, (21) et (22) traitant la **Forward Propagation**, nous pouvons définir les formules pour une troisième couche, puis une c-ième couche.

$$\text{Sur la 1ère couche : } \begin{cases} Z^{[1]} = W^{[1]}.X + b^{[1]} \\ \\ A^{[1]} = \frac{1}{1+e^{-Z^{[1]}}} \end{cases}$$

$$\text{Sur la 2nd couche : } \begin{cases} Z^{[2]} = W^{[2]}.A^{[1]} + b^{[2]} \\ \\ A^{[2]} = \frac{1}{1+e^{-Z^{[2]}}} \end{cases}$$

$$\text{Sur la 3ème couche : } \begin{cases} Z^{[3]} = W^{[3]}.A^{[2]} + b^{[3]} \\ \\ A^{[3]} = \frac{1}{1+e^{-Z^{[3]}}} \end{cases}$$

$$\text{Sur la c - ème couche : } \begin{cases} Z^{[c]} = W^{[c]}.A^{[c-1]} + b^{[c]} \\ \\ A^{[c]} = \frac{1}{1+e^{-Z^{[c]}}} \end{cases}$$

Attention, que pour cette formule générale puisse fonctionner, il nous faut imposer une condition qui est que $X = A^{[0]}$.

Avec *python*, nous allons vérifer que nos dimensions sont vérifiées, c'est à dire si chaque couche s'active 100 fois (car il y a 100 données au totale).

```
## A0 (2, 100)
## A1 (12, 100)
## A2 (28, 100)
## A3 (16, 100)
```

On remarque bien que nous dimensions concordent avec ce que nous attendions.

6.3. Back Propagation à c couche(s) et n neurone(s)

Ici, nous partons de la troisième couche cachée (donc de la dernière couche ici), pour revenir à la première couche cachée. Nous utiliserons les formules (24) jusque (29).

$$\text{Sur la 3ème couche : } \begin{cases} dZ_3 = A^{[3]} - Y \\ dW^{[3]} = \frac{1}{m} dZ_3 . (A^{[2]})^T \\ db^{[3]} = \frac{1}{m} \sum dZ_3 \end{cases}$$

$$\text{Sur la 2nd couche : } \begin{cases} dZ_2 = (W^{[3]})^T \times dZ_3 \times A^{[2]}(1 - A^{[2]}) \\ dW^{[2]} = \frac{1}{m} dZ_2 . (A^{[1]})^T \\ db^{[2]} = \frac{1}{m} \sum dZ_2 \end{cases}$$

$$\text{Sur la 1ère couche : } \begin{cases} dZ_1 = (W^{[2]})^T \times dZ_2 \times A^{[1]}(1 - A^{[1]}) \\ dW^{[1]} = \frac{1}{m} dZ_1 . X^T \\ db^{[1]} = \frac{1}{m} \sum dZ_1 \end{cases}$$

On en déduit la formule général suivante :

$$\begin{cases} dZ_{\text{couche finale}} = A^{[\text{couche finale}]} - Y \\ dW^{[c]} = \frac{1}{m} dZ_c . (A^{[c-1]})^T \\ db^{[c]} = \frac{1}{m} \sum dZ_c \\ dZ_{c-1} = (W^{[c]})^T \times dZ_c \times A^{[c-1]}(1 - A^{[c-1]}) \end{cases}$$

Avant de s'attaquer à notre descente de gradient, il va falloir que nous verifions les dimensions de nos gradients. Nous continuons de reprendre nos exemples à trois couches cachées.

```
## dW3 (16, 28)
## db3 (16, 1)
## dW2 (28, 12)
## db2 (28, 1)
## dW1 (12, 2)
## db1 (12, 1)
```

Les dimensions sont effectivement celles que nous voulions en partant de la dernière couche cachée pour revenir à la première.

Pour la suite, nous pourrons utiliser nos formules (30) jusque (33) pour notre méthode de descente de gradient.

6.4. Exemples de différents réseaux de neurones

Nous repartons de notre deuxième base de données du package `make_circles` de la librairie `sklearn.datasests` (figure 9). Nous pouvons tester notre réseau de neurone de toute à l'heure composé de :
- 2 Neurones sur la 1ere couche
- 12 Neurones sur la 2nd couche
- 28 Neurones sur la 3ème couche
- 16 Neurones sur la 4ème couche

Avec un pas **alpha de 0.001** et **3000 itérations**.

```
## neurones cachées (en anglais : Hidden layers) : (2, 12, 28, 16)
## La precision du modele est de 73.0 %
```

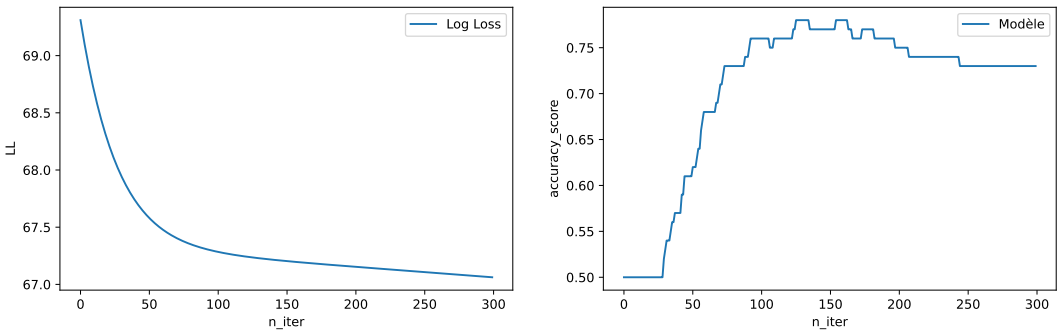

FIGURE 20 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

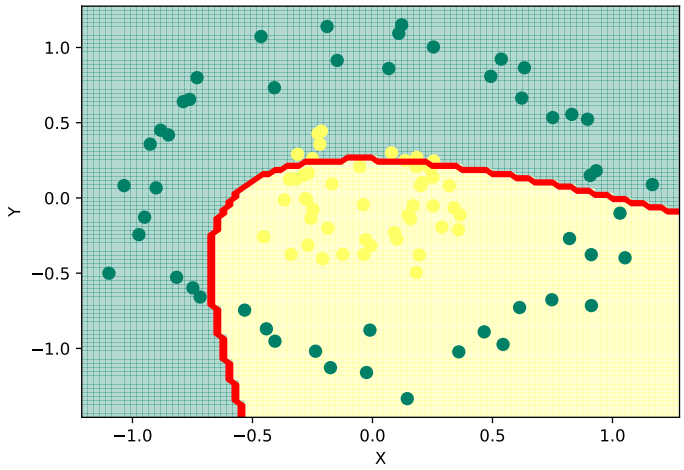

FIGURE 21 – Régression obtenue avec comme hidden layers = (2, 12, 28, 16)

Nous avons donc un modèle ayant une perfomance de 73 % avec une stabilisation de notre fonction Log Loss (figure 20 partie gauche) ce qui nous informe que notre apprentissage n'évoluera pas plus (figure 20 partie droite).

Nous allons tester différents programmes pour choisir lequel serait le meilleur, à savoir combien de couches ? combien de neurones par couches ? Nous garderons le pas **alpha = 0.001** et un nombre de **3000 itérations** pour chaque programme. Pour présenter nos paramètres, on écrira sous la forme suivante nos réseaux : $(n^{[0]}, n^{[1]}, ..., n^{[c]})$ dans lequel nous mettrons par exemple

: $n^{[0]} = 2, n^{[1]} = 6$ ce qui signifie 2 neurones sur la première couche et 6 neurones sur la deuxième couche.

```
## neurones cachées (en anglais : Hidden layers) : (16, 16, 32)
## La precision du modele est de 68.0 %
```

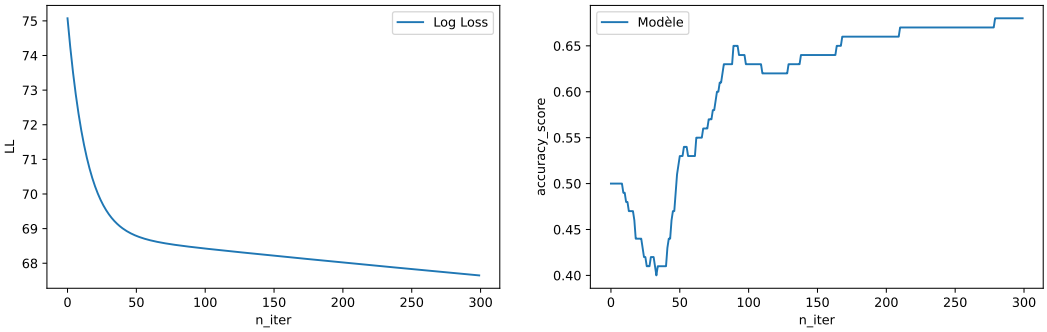

FIGURE 22 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

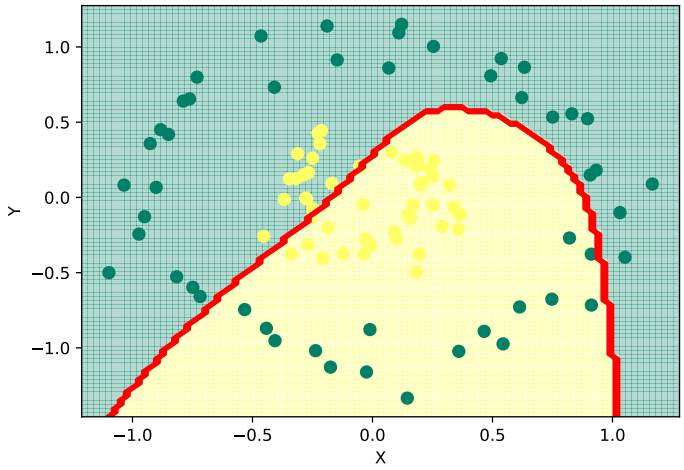

FIGURE 23 – Régression obtenue avec comme hidden layers = (16,16,32)

Avec seulement trois couches cachées, notre fonction Log Loss (figure 22 partie gauche) se stabilise, mais l'apprentissage est quant à lui loin d'être finis.

```
## neurones cachées (en anglais : Hidden layers) : (16, 16, 16, 16)
## La precision du modele est de 49.0 %
```

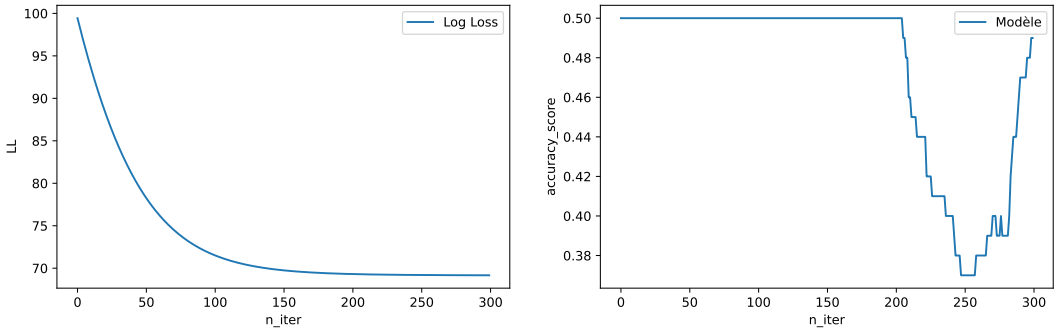

FIGURE 24 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

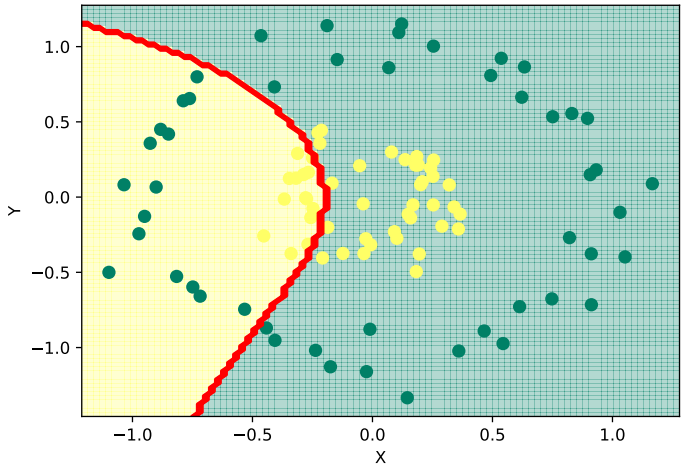

FIGURE 25 – Régression obtenue avec comme hidden layers = (16,16,16,16)

Ici, nous avons quatres couches cachées composées chacun de 16 neurones, on constate que l'apprentissage n'évolue pas et même, chute brutalement (figure 24 partie droite) ce qui nous donne un modèle médiocre.

```
## neurones cachées (en anglais : Hidden layers) : (32, 32, 32, 32)
## La precision du modele est de 81.0 %
```

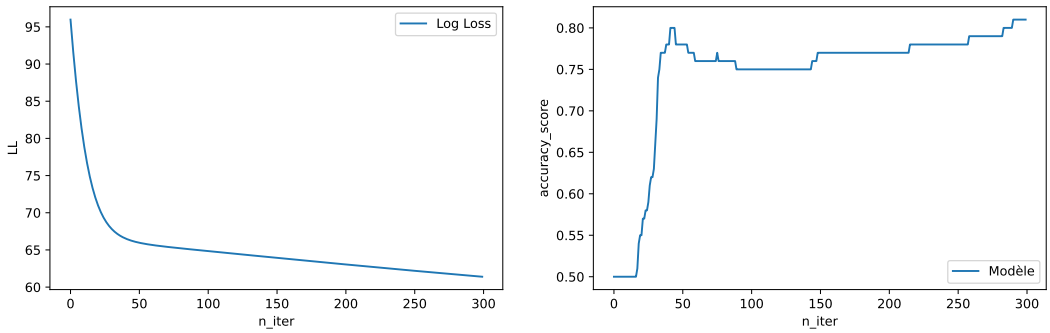

FIGURE 26 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

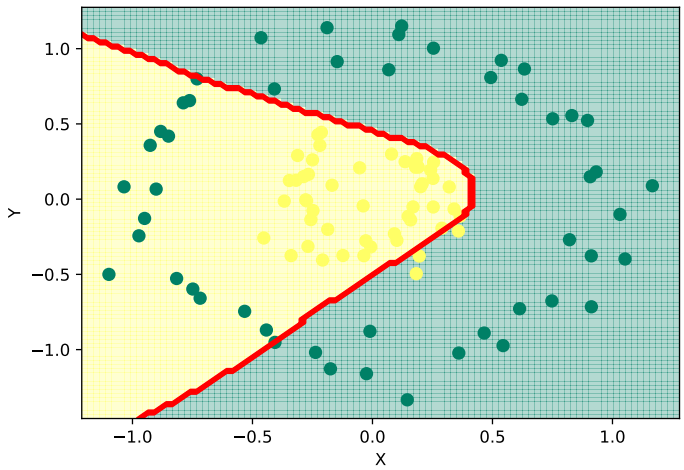

FIGURE 27 – Régression obtenue avec comme hidden layers = (32,32,32,32)

A la différence, de l'ancien réseau de neurones, ici nous avons 32 neurones sur les quatres couches cachées, ce qui nous un apprentissage plus tôt correct de 81% et une stabilisation de la fonction Log Loss et de l'apprentissage (figure 26).

```
## neurones cachées (en anglais : Hidden layers) : (2, 4, 6, 6, 8, 10)
## La precision du modele est de 53.0 %
```

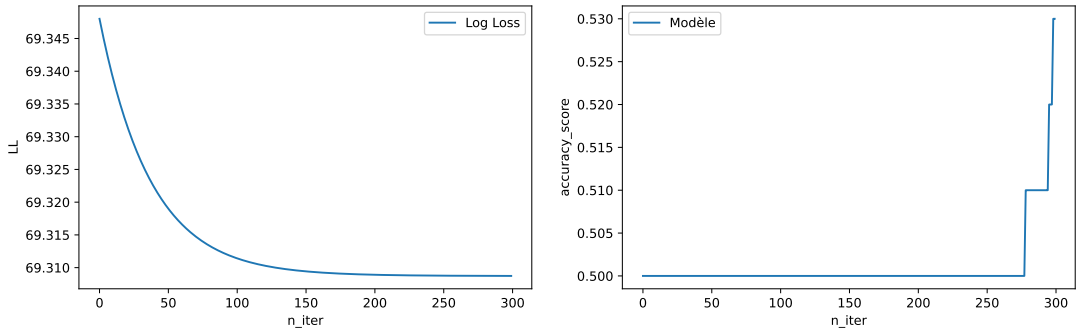

FIGURE 28 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

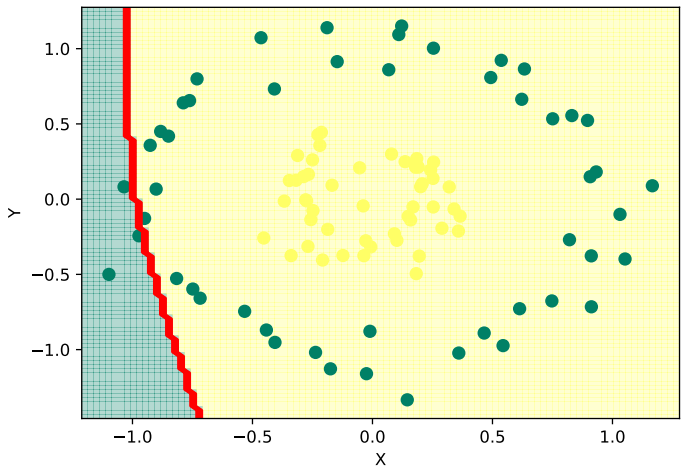

FIGURE 29 – Régression obtenue avec comme hidden layers = (2,4,6,6,8,10)

Nous voulions voir si en augmentant le nombre de couches, nous aurions un bon apprentissage, ce n'est pas le cas ici, car l'apprentissage n'a évolué qu'à partir de la fin du réseau (figure 28).

```
## neurones cachées (en anglais : Hidden layers) : (16, 40, 90, 100)
## La precision du modele est de 94.0 %
```

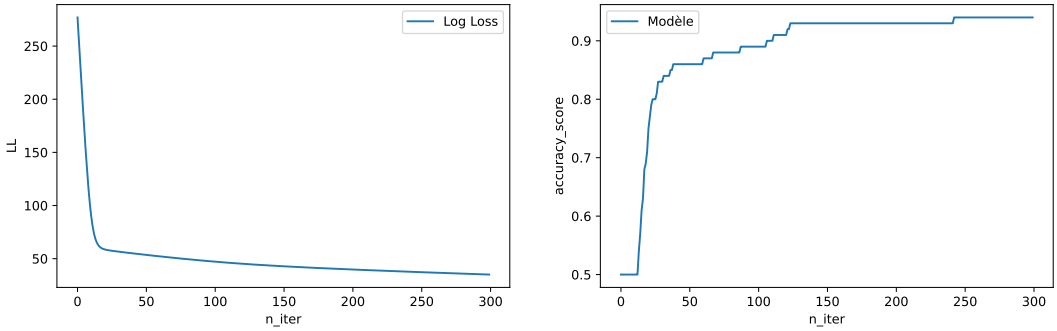

FIGURE 30 – Evolution de la fonction Log Loss (à gauche), Evolution/apprentissage du modèle (à droite)

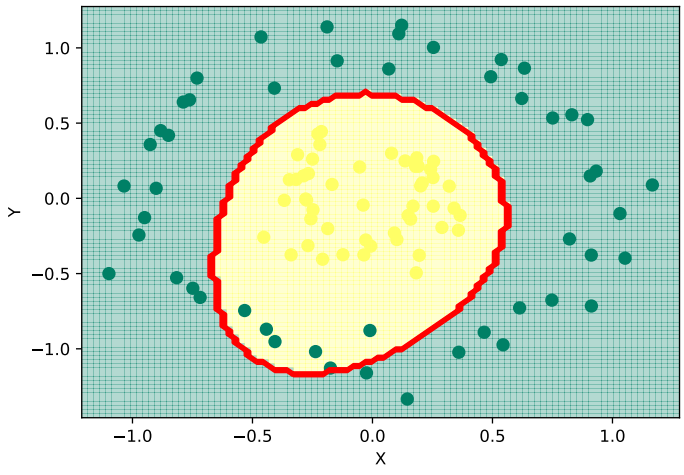

FIGURE 31 – Régression obtenue avec comme hidden layers = (16,40,90,100)

On constate que le meilleur modèle de réseau est celui ayant quatre couches composées d'un très grand nombre de neurones chacun. En effet, on pourrait croire qu'en construisant des modèles de réseaux de neurones à plusieurs couches et plusieurs neurones, on aurait un modèle très perfomant, sauf que ce n'est pas forcément le cas. On peut voir que notre modèle ne s'améliore pas d'avantage dès lorsqu'on dépasse les 4 couches, voir même que cela le rend moins performant. Nous verrons par la suite un phénonème très connu nommé le **vanishing gradient problem** qui explique cette situation qui nous montre que la perfomance de notre modèle peut être très basse en ayant de nombreuse couches de neurones.

7. Réseau de Neurone récurrent (RNN)

Chaque schéma de cette partie a été produit par le scientifique Jaouad Dabounou.

7.1. Origines

Un réseau de neurone récurrent (en anglais : Recurrent Neural Network (RNN)) est un réseau de neurones artificiels présentant des connexions récurrentes. Les RNN sont très exploités en Deep learning et dans le développement de modèles qui simulent l'activité des neurones dans le cerveau humain (MORERE, Y 1998). Ils sont utiles pour prédire un résultat et se distinguent des autres types de réseaux de neurones, car ils utilisent des boucles de **rétroaction** pour analyser des données qui informe sur la sortie finale. Ces boucles de **rétroaction** telles que la **rétropropagation** permettent aux informations de persister, ce qui nous fais penser à un phénomène décrit comme une mémoire à court terme.

Les réseaux de neurones sont créés avec des composants permettant le traitement de données interconnectées qui sont créés de manière à fonctionner comme les réseaux neuronaux d'un cerveau humain. Ils sont composés de couches de neurones qui ont la mission d'analyser les données d'entrée et de transmettre les données de sortie aux autres couches de neurones du réseau. Les différentes couches de neurones sont reliées par des poids W qui influencent le traitement et la sortie finale du réseau de neurone. D'une manière plus scientifique, les RNN permettent de traiter des données séquentielles et temporelles, ce qui nous permet de résoudre différents types de problèmes.

7.2. Représentation d'un RNN

Regardons de plus près la composition d'une cellule d'un RNN.

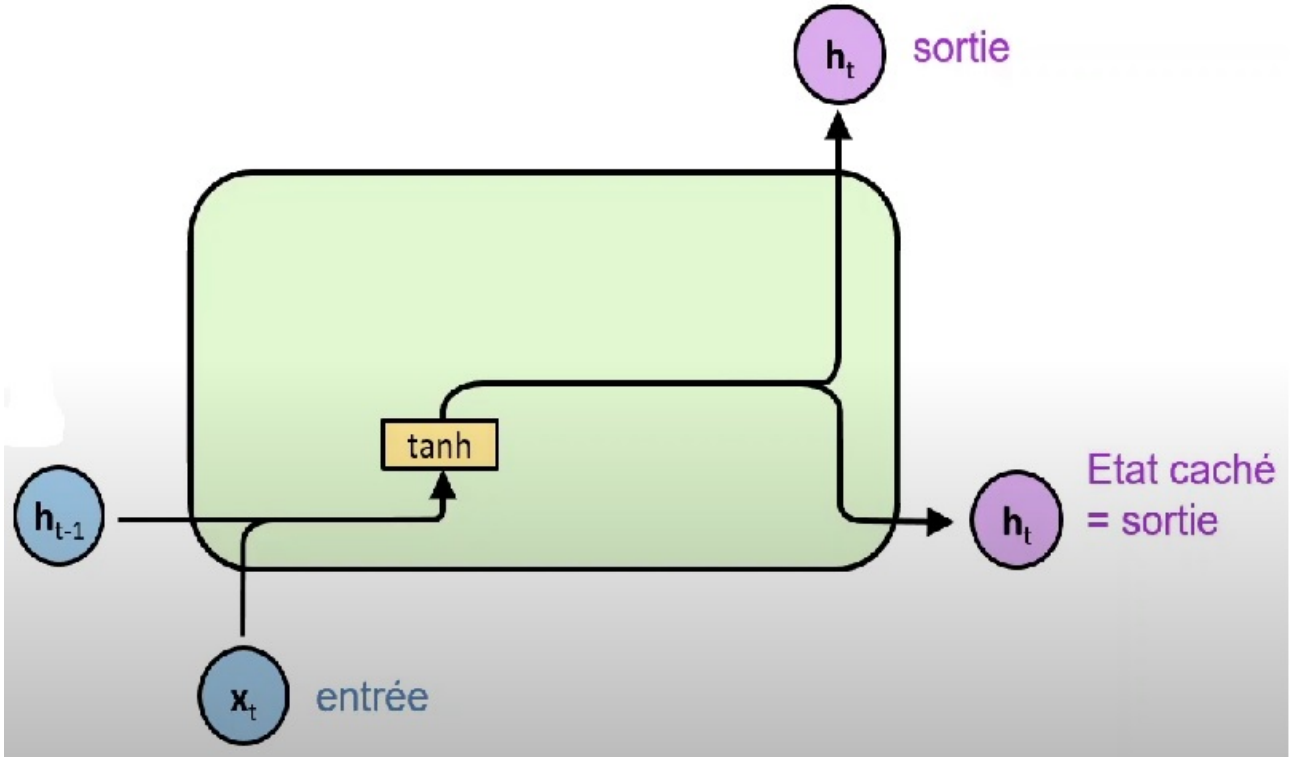

FIGURE 32 – Schéma d'un RNN au complet avec les entrées, fonctions d'activation tanh et les sorties

Cette cellule est composée de deux entrées que l'on nomme x_t et h_t, qui nous donneront deux sorties h_t. A l'intérieur de la cellule, nous allons retrouver une fonction d'activation de type *tanh* qui pour rappel nous renvoies des valeurs entre -1 et 1 symbolisé par la case jaune. Le choix de *tanh* repose sur le fait que cette fonction apporte une convergence plus rapide que la fonction *sigomïde*. Pour apprendre, un RNN utilise la méthode de la descente du gradient afin de mettre à jour les poids W entre ses neurones. De plus, la fusion de lignes indique une concaténation, tandis qu'une bifurcation de ligne indique que son contenu est copié et que les copies vont à des emplacements différents (figure 32).

Pour comprendre cette cellule, nous proposons donc de faire le lien avec les visualisations de réseau de neurones que nous avions vu durant cette étude, voici donc la représentation de la cellule d'un RNN par les couches d'un réseau de neurones (figure 33).

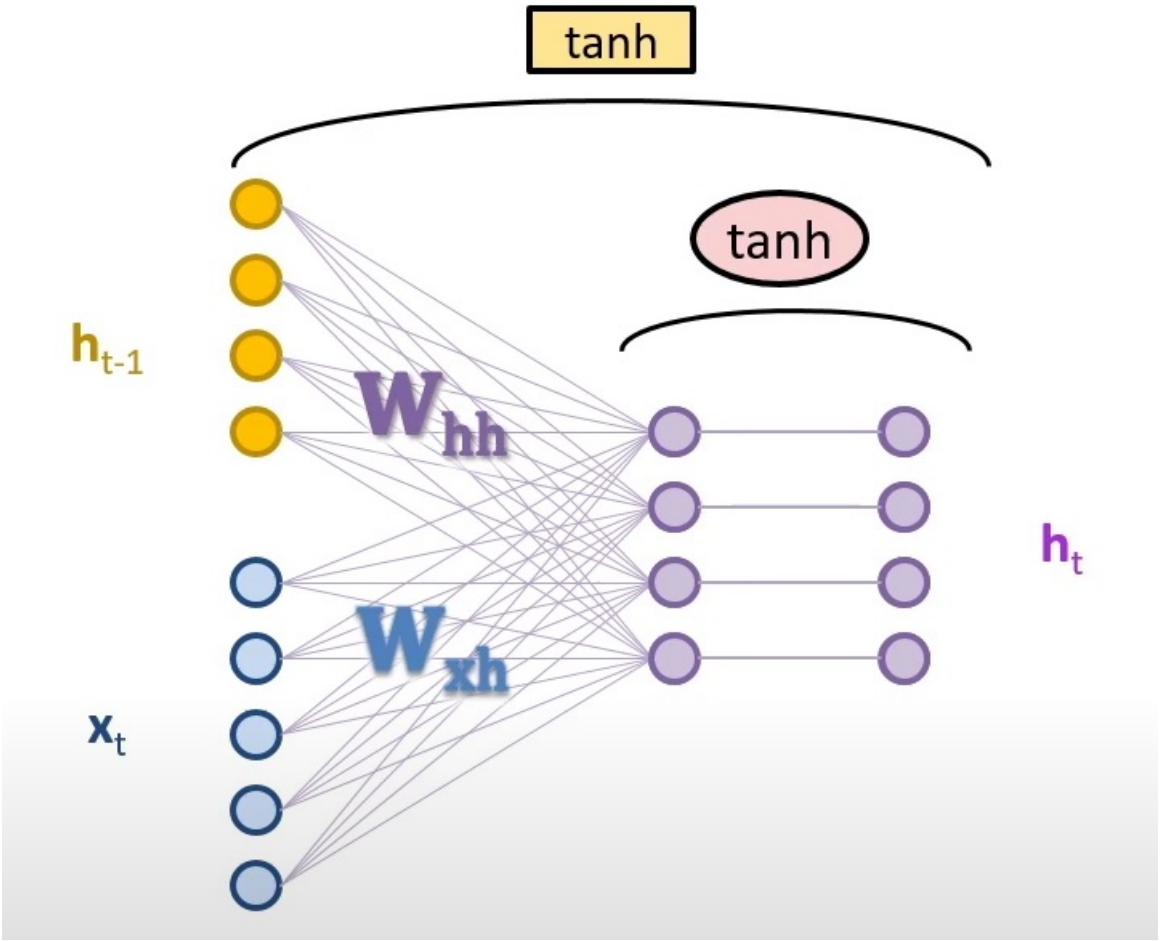

FIGURE 33 – Détails des couches d'un RNN faisant le lien avec les représentations des réseaux de neurones

Nous avons que W_{hh} et W_{xh} représentent les poids tel que les poids W dont nous avons déjà étudié. Suite à l'étape *tanh* se situant dans la cellule, notre valeur h_t à pour formule en post traitement :

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{34}$$

En effet, *tanh* permet de réguler les valeurs circulant dans le réseau.

L'intérêt ensuite est de créer une chaîne de cellules répétitifs de réseau de neurones d'où le terme de **récurrent** (figure 34).

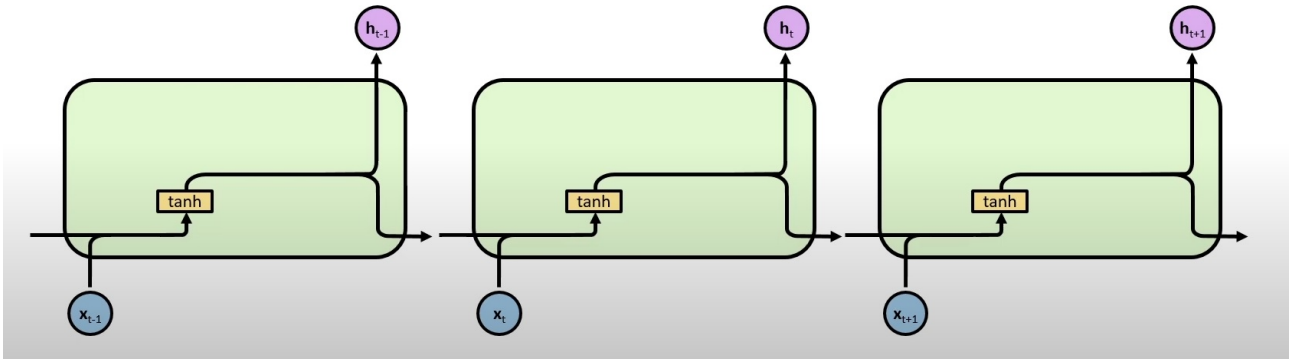

FIGURE 34 – Schéma récurrent des RNN qui symbolise bien le principe de répétition (récurrent)

Les réseaux de type RNN ne sont rien d'autre qu'un ensemble de calculs répétitifs muni d'une structure très convenable composé d'une seule couche et d'une seule fonction d'activation.

7.3. Inconvénients d'un RNN

Le plus gros inconvénient des RNN est le problème du gradient de disparition (en anglais : **Vanishing gradient problem**), dans lequel la performance du réseau de neurone souffre du fait qu'il peut rencontrer des problèmes d'apprentissage lors de la rétropopagation (back propagation). On retrouve ce problème lors que les réseaux de neurones sont composés de très nombreuses couches de neurones que l'on nomme un réseau de neurone profond, qui est souvent utilisé pour résoudre des données qui sont de plus en plus complexes (ZERHOUNI, N 2002). Donc plus la mise à jour des poids est faite par la méthode du gradient, moins ils apprenent, voire ils ne sont plus modifiés à cause du **Vanishing gradient problem**.

Les RNN qui utilisent donc une méthode d'apprentissage de retropopagation basée sur le gradient se dégradent au fur et à mesure qu'ils deviennent plus grands et plus complexes. Du point de vue des calculs, cela devient plus long et coûteux pour les premières couches.

C'est pour cela que dans l'intérêt de régler ces problèmes de gradients et de mémoire, les informaticiens Sepp Hochreiter [9] et Jurgen Schmidhuber [10] ont inventé en 1997, les unités de mémoire longue et courte durée qu'on notera LSTM (en anglais : Long Short Time Memory).

Les RNN construites à partir des unités LSTM répartissent les données en cellules de mémoire à court terme et à long terme. Les RNN peuvent ainsi déterminer quelles sont les données qui sont importantes et doivent être mémorisées sur le long terme puis réintégrées dans le réseau, et quelles données peuvent être oubliées sur le court terme.

9. Josef « Sepp » Hochreiter né en 1967 est un informaticien allemand spécialisé en IA.
10. Jürgen Schmidhuber né en 1963 est un informaticien surtout connu pour ses travaux dans le domaine de l'IA, de l'apprentissage en profondeur et des réseaux de neurones artificiels.

8. Réseau LSTM

Chaque schéma de cette partie a été produit par le scientifique Jaouad Dabounou.

Le principe d'un réseau LSTM (Long Short Time Memory) repose sur le même fonctionnent qu'un réseau RNN. Cependant, comme nous allons le voir, le réseau LSTM n'est pas composé d'une seule couche de neurone dans sa cellule, mais de quatre couches de neurones au total! L'intérêt repose sur le principe de garder des données en mémoire à long terme et de savoir quelles sont les données qui passeront d'une cellule à une autre.

8.1. Construction d'un LSTM

8.1.1 Etape 0

On va partir d'une cellule de RNN, on rappel que la formule qui actualise l'état caché est donné par :

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t) \quad (34)$$

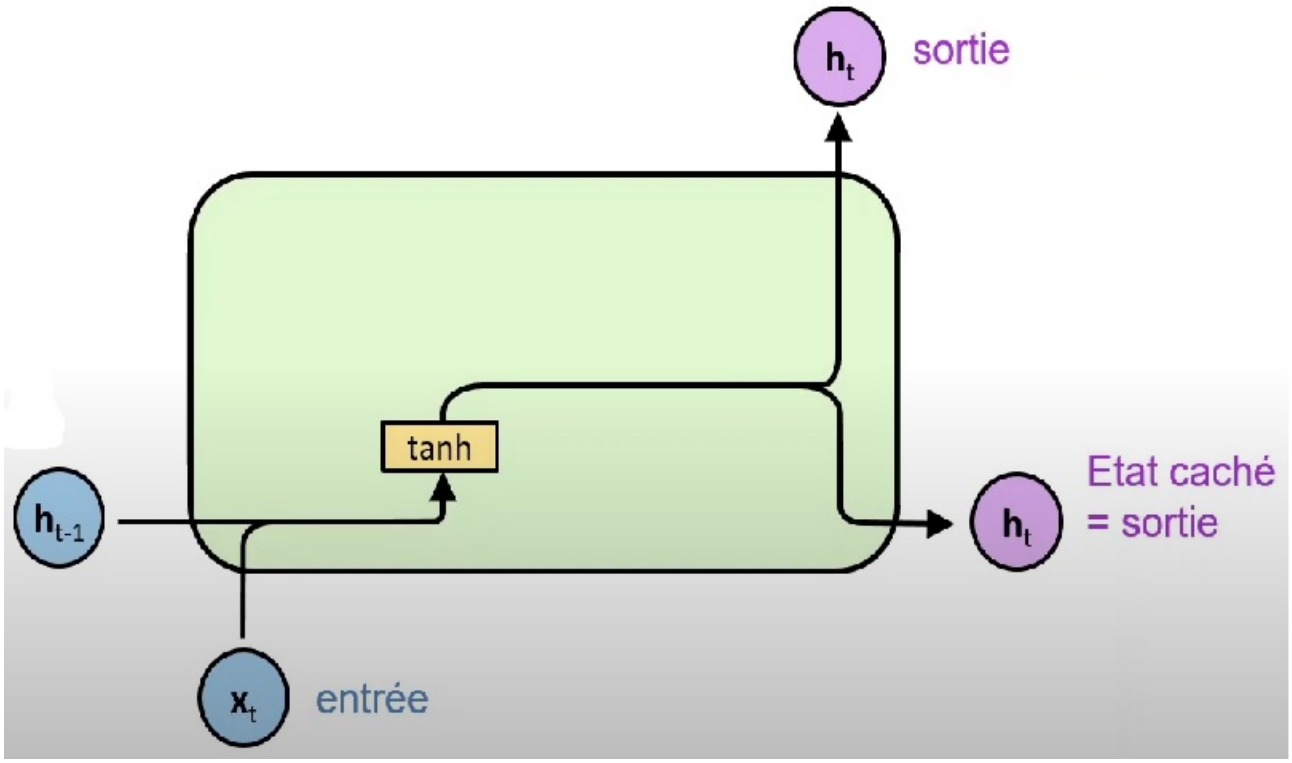

FIGURE 35 – Construction LSTM étape 0

On rappel que cette cellule génére souvent le problème du **Vanishing Gradient** à cause de *tanh* pour les réseaux de type profond qui cause donc des problèmes de mémoire à long terme.

8.1.2 Etape 1

Pour palier à ce problème de mémoire, nous allons introduire une nouvelle variable dans notre cellule qui est c_{t-1} en entrée et c_t en sortie qui représentera l'état de notre cellule. Grâce à cela, nous pourrons

garder des informations sur de longue durée. Les cercles roses représentent des opérations ponctuelles, comme l'addition/multiplication de vecteurs.

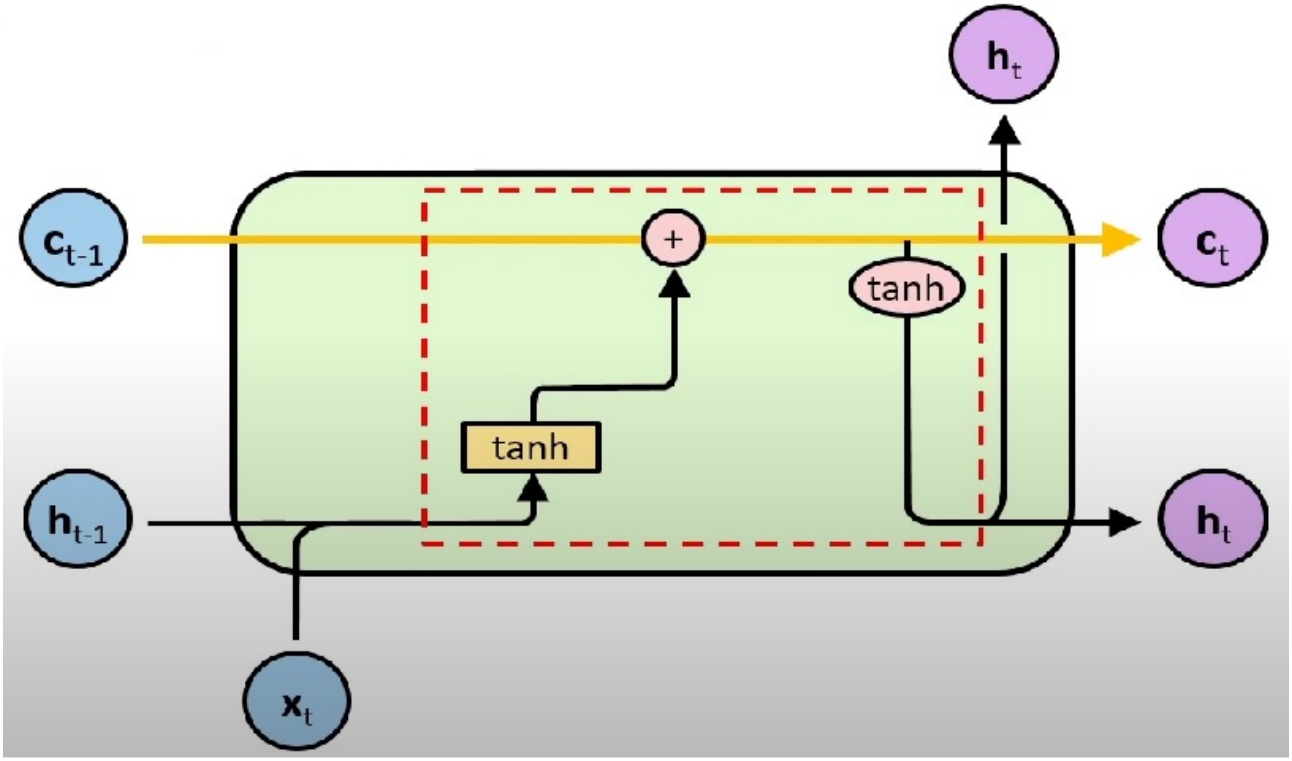

FIGURE 36 – Construction LSTM étape 1 : Ajout de la mémoire à long terme

Les variables se mettront à jour selon les formules suivantes :

$$c_t = c_{t-1} + tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{35}$$

$$h_t = tanh(c_t) \tag{36}$$

Nous serons donc assuré que notre état h_t à l'instant t puisse contenir tous les états cachés des cellules précédentes. Cependant, en gardant toutes ces informations des états précédents, nous allons rencontrer le problème que le passé va écraser nos données du présent, ce qui ne nous arrange pas.

8.1.3 Etape 2

Il va falloir ici qu'on puisse permettre au réseau de garder ou d'oublier les informations des données du passé selon leur importance si elles sont déterminantes ou pas pour la suite de nos calculs. C'est pour ça que nous allons introduire une nouvelle couche qui sera une nouvelle fonction d'activation représentée par la *sigmoïde* σ pour la variable c_t. Elle aura pour mission de mettre à jour la mémoire à long terme. Le réseau pourra oublier les informations non pertinentes et garder celles qui sont pertinentes.

Nos formules s'écrivent de la forme :

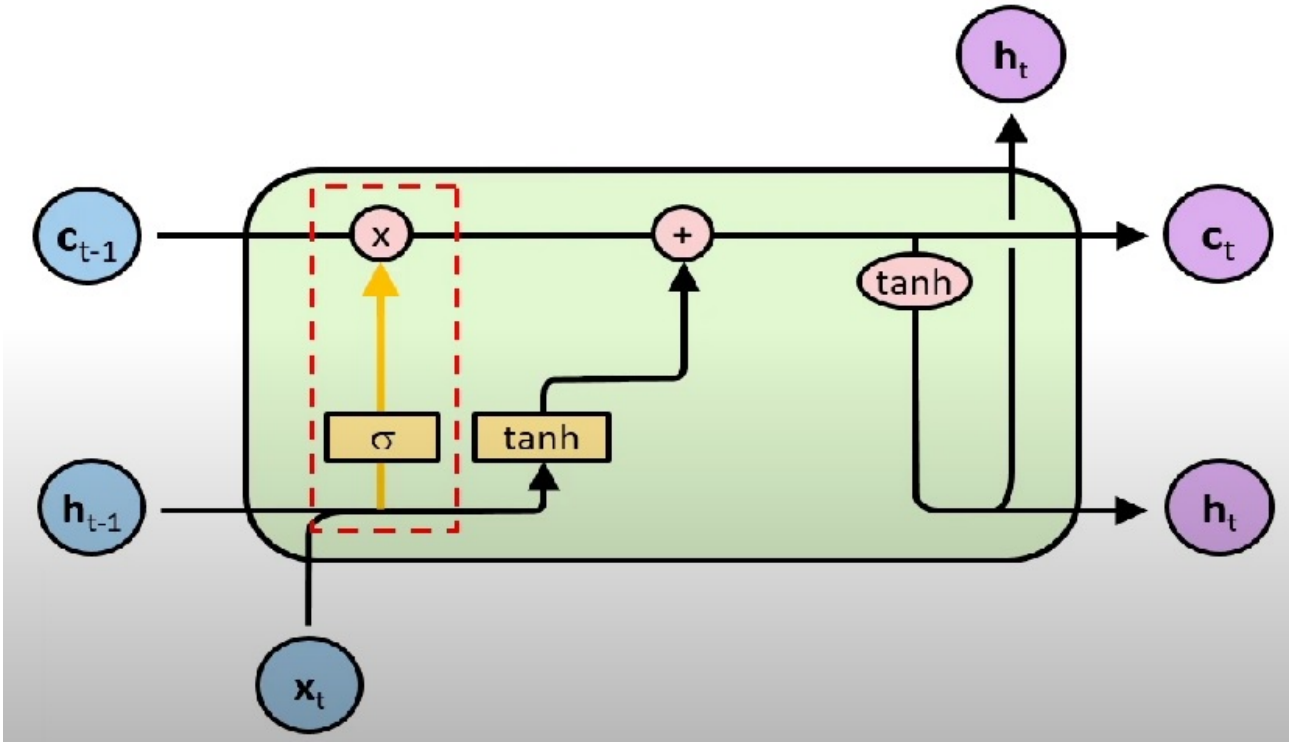

FIGURE 37 – Construction LSTM étape 2 : Ajout de la forget gate

$$f_t = \sigma(W_{hf} h_{t-1} + W_{xf} x_t) \tag{37}$$

$$c_t = f_t \times c_{t-1} + tanh(W_{hh} h_{t-1} + W_{xh} x_t) \tag{38}$$

$$h_t = tanh(c_t) \quad (35)$$

On rappel que la fonction *sigmoïde* renvoi des valeurs entre 0 et 1. Le calcul $f_t \times c_{t-1}$ représente le passé mise à jour, ce qui permet de répondre au problème de l'étape 1. On parle alors de porte oublie ou forget gate pour cette étape de calcul du réseau, car les informations qui auront une valeur proche de 0 seront oubliées et celles de proche de 1 seront gardées dans la mémoire à long terme.

Voici un schéma permettant d'illustrer cette étape de calcul par l'affichage des couches du réseau de neurones.

Il s'agit en faite d'un calcul similaire à celui de *tanh* comme dans l'étape 0, mais ici on préfère la *sigmoïde* par rapport à ces valeurs comprises entre 0 et 1 qui a été expliqué plus haut.

8.1.4 Etape 3

Nous venons de faire une mise à jour multiplicative de la mémoire à long terme par l'intermédiaire de la forget gate. Ici, nous allons rajouter une mise à jour additive. Pour cela, il va nous falloir rajouter une couche par la fonction d'activation *sigmoïde* σ que nous multiplierons par la fonction d'activation

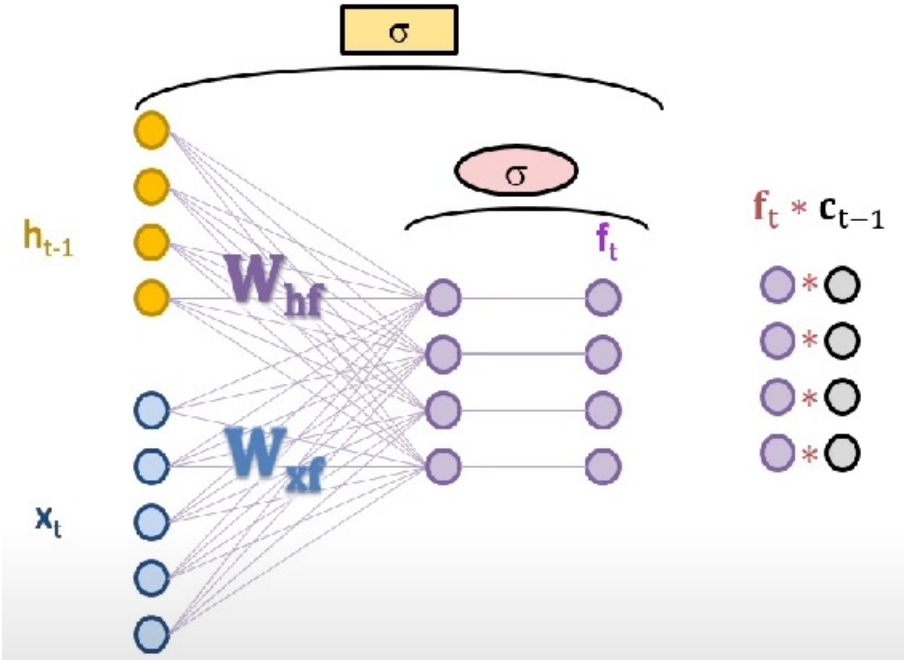

FIGURE 38 – Schéma couche LSTM de la forget gate

tanh (notre calcul de base depuis le début), puis nous additionnerons le tout avec les mises à jour effectuer par la forget gate.

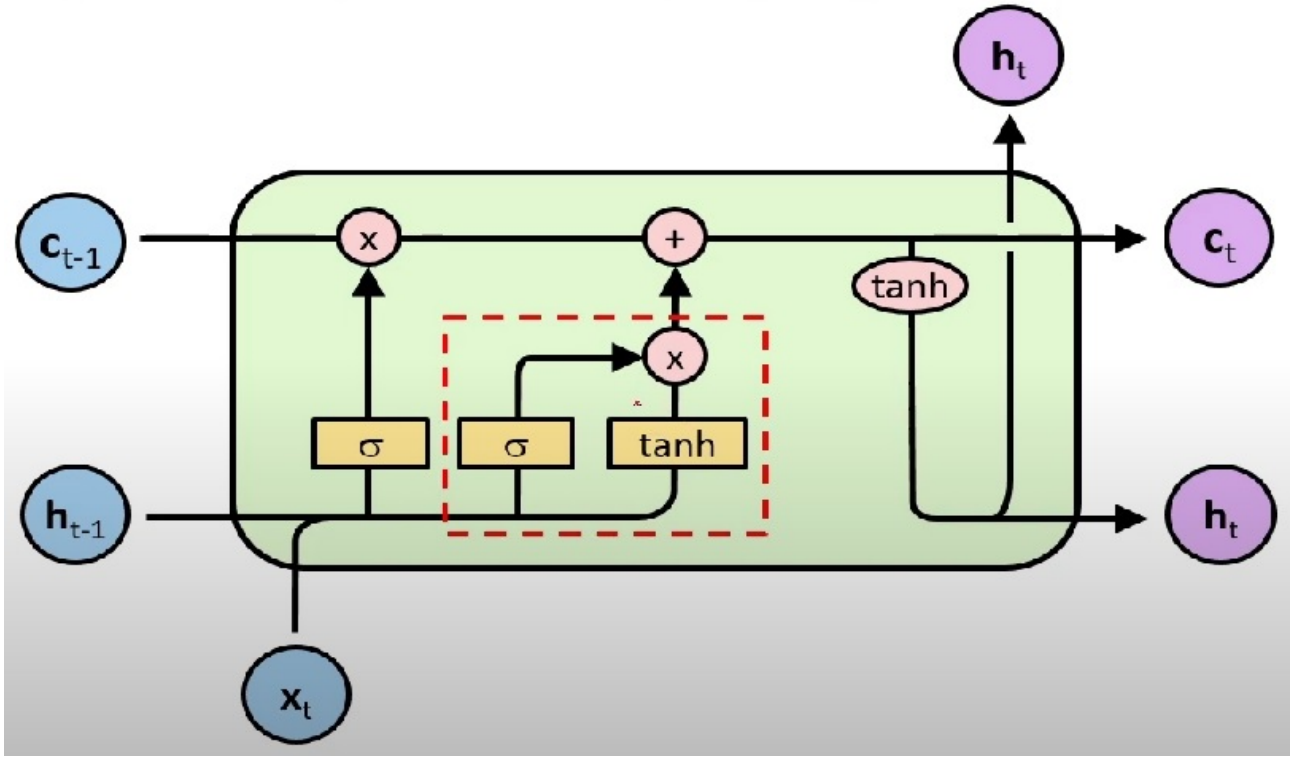

FIGURE 39 – Construction LSTM étape 3 : Ajout de l'input gate

Nos formules s'écrivent donc :

$$f_t = \sigma(W_{hf} h_{t-1} + W_{xf} x_t) \quad (37)$$

$$i_t = \sigma(W_{hi}h_{t-1} + W_{xi}x_t) \tag{39}$$

$$c_t = f_t \times c_{t-1} + o_t \times tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{40}$$

$$h_t = tanh(c_t) \quad (35)$$

La couche supplémentaire qui vient d'être rajoutée se nomme la input gate qui va permettre au réseau d'apprendre à ignorer ou utiliser les informations de l'entrée des données selon leur importance. Comme nous utilisons encore la *sigmoïde*, les valeurs de sortie de i_t seront comprises entre 0 et 1. Cette mise à jour additive à le même but que la mise à jour multiplicative.

8.1.5 Etape 4

Pour terminer, nous rajoutons une dernière couche qui est encore une fonction d'activation de type *sigmoïde* qui aura pour but de mettre à jour l'état caché h_t dont nous n'avons pas touché depuis le début. Ceci permet au réseau de savoir quelles informations notre état caché h_t devra-t-il gardé pour la suite. Le fonctionnement reste le même que dans les étapes précédentes, on termine la mise à jour par une multiplication qui gardera les valeurs proche de 1 et supprimera les valeurs proche de 0 pour la suite. On nomme cette étape la output gate.

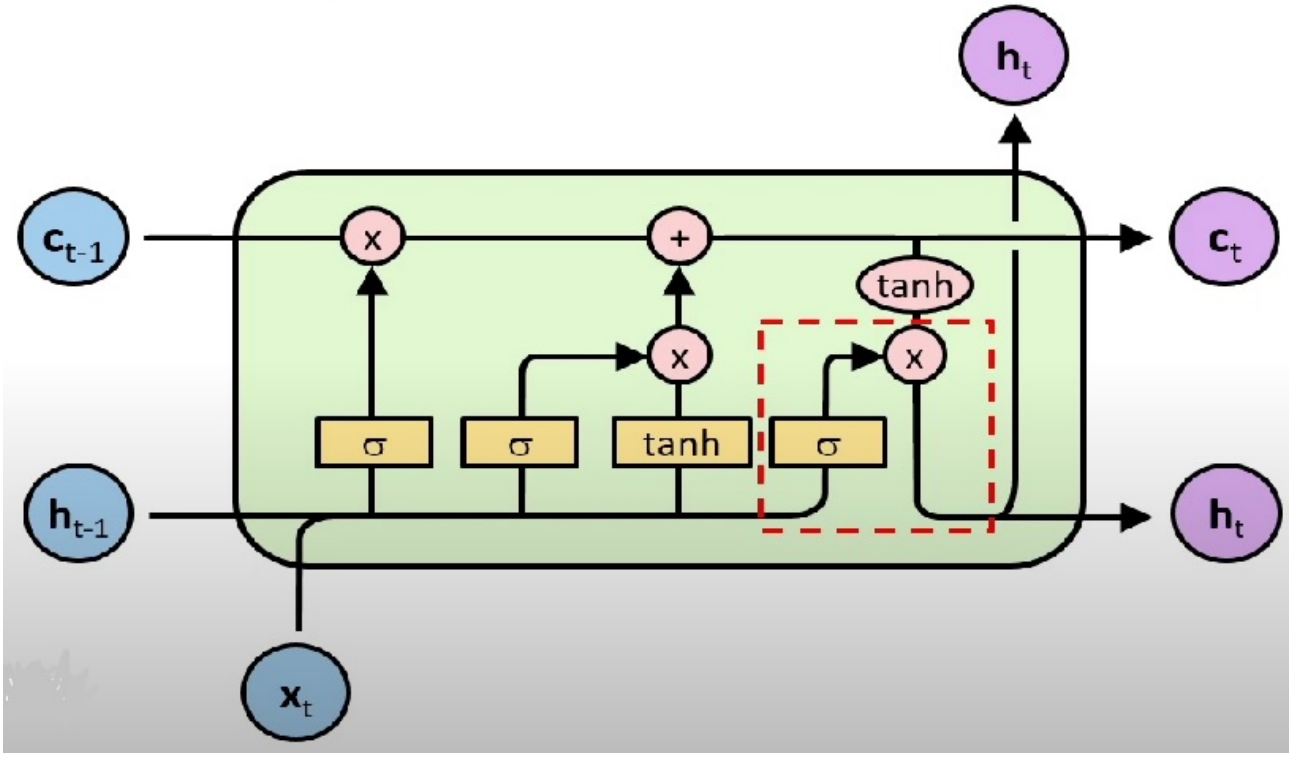

FIGURE 40 – Construction LSTM étape 4 : Ajout de l'Output gate

Voici donc la formulation génénerak d'un réseau LSTM :

$$f_t = \sigma(W_{hf} h_{t-1} + W_{xf} x_t) \quad (37)$$

$$i_t = \sigma(W_{hi} h_{t-1} + W_{xi} x_t) \quad (39)$$

$$O_t = \sigma(W_{hO} h_{t-1} + W_{xO} x_t) \quad (41)$$

$$c_t = f_t \times c_{t-1} + O_t \times tanh(W_{hh} h_{t-1} + W_{xh} x_t) \quad (40)$$

$$h_t = O_t \times tanh(c_t) \quad (42)$$

8.2. Représentation d'un LSTM

Voici la représentation finale d'un réseau LSTM dont nous allons faire une synthèse de chaque étape clé après ce schéma.

— Forget gate :

$$f_t = \sigma(W_{hf} h_{t-1} + W_{xf} x_t) \quad (37)$$

Permet une première mise à jour multiplicative qui permet d'oublier les valeurs proche de 0 et garder celle proche de 1 pour la mémoire à long terme.

— Input gate :

$$i_t = \sigma(W_{hi} h_{t-1} + W_{xi} x_t) \quad (39)$$

Mettre à jour les informations transmises par la couche *tanh* pour voir qu'elles informations peuvent rentrées dans la mémoire à long terme par le même principe qu'au dessus.

— Output gate :

$$O_t = \sigma(W_{hO} h_{t-1} + W_{xO} x_t) \quad (41)$$

Dernière mise à jour pour savoir quelles informations de l'état caché h_t vont passer pour la cellule suivante avec le même raisonnement que lors des Forget et Input gate.

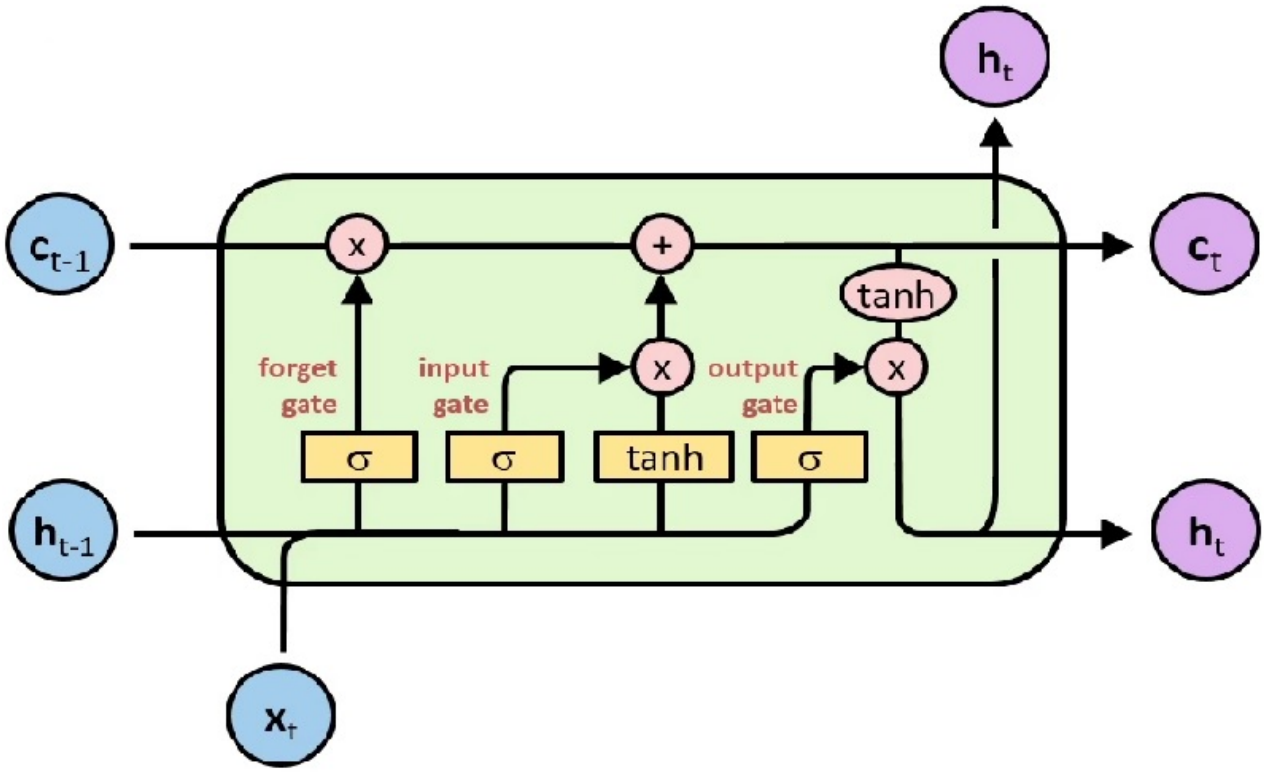

FIGURE 41 – Schéma LSTM complet

— Caclul Final de c_t et h_t :

$$c_t = f_t \times c_{t-1} + O_t \times tanh(W_{hh}h_{t-1} + W_{xh}x_t) \quad (40)$$

Nous avons que c_t parcourt quasiment toutes les couches du réseau avec des calculs très simples (multilplication et addition) composés de plusieurs mises à jour.

$$h_t = O_t \times tanh(c_t) \quad (42)$$

L'étape caché h_t est quant à lui mise à jour une seule fois.

Comme pour les RNN, les LSTM sont un ensemble de calculs répétitifs muni d'une structure plus adéquate composés de quatre couches munie des fonctions d'activations *tanh* et *sigmoïde*.

Grâce aux *sigmoïde*, l'information est régulée intelligement qui permet de mettre à jour les poids W que nous avons vus lors des étapes de **back propagation**.

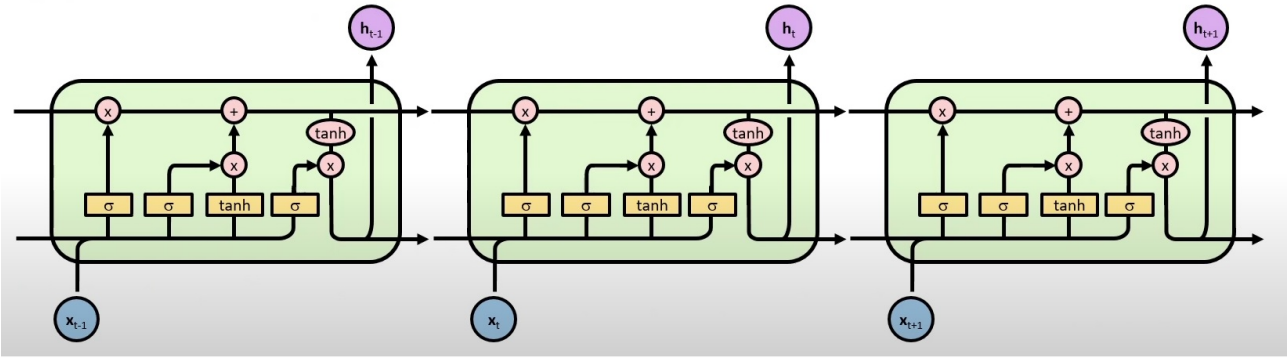

FIGURE 42 – Schéma récurrent LSTM

9. Conclusion

Depuis de nombreuses décennies, l'IA ne cesse de progrésser, notamment par l'intermédiaire des réseaux de neurones que nous avons pû étudier qui ont abouti aux RNN et LSTM. Cependant, de nos jours beaucoup de scientifique, data scientist et d'autres se servent de ces réseaux sans savoir comment ils fonctionnent. C'est pour cela que nous avons menés cette étude, dans l'intérêt de pouvoir expliquer de manière mathématiquement les points clés des réseaux de neurones à c couches et n neurones à travers les outils que l'on retrouve en mathématiques.

Les réseaux RNN et LSTM ne sont pas les seuls réseaux qui permettent à l'IA de progrésser, par exemple les GRU qui sont des LSTM plus perfomant et plus rapide en terme de calcul permettent une meilleure approche pour résoudre des problèmes complexes. Tout comme les réseaux de neurones convolutifs qui permettent de reconnaître à travers une banque de données, une image en donnant sa nature (exemple de savoir différencier un chat d'un chien à travers des images).

Il reste néanmoins une analyse plus poussée à étudier pour les réseaux de neurones, à savoir trouver la bonne combinaison permettant de traiter au mieux un problème, à savoir, combien de couches composées de combien de neurones. Ici, nous n'avons étudié qu'un morceau des réseaux de neurones, comme le fait de ne pas choisir plus de quatre couches sous peine d'avoir le problème du **Vanishing Gradient** qui surgisse et freine nos apprentissages.

Il sera donc capitale de pouvoir à l'avenir déterminer ces combinaisons de couches et de neurones pour résoudre tout type de problème à travers les réseaux de neurones pour permettre à l'IA de faire un pas de plus vers l'avant.

10. Bibliographie

CHARNIAK, Eugene. Introduction au Deep Learning. Dunod, 2021.

DABOUNOU, Jaouad. Réseaux de neurones récurrents et LSTM. 2020.

MORERE, Yann. Les réseaux de neurones récurrents. 1998.

SAINT-CIRGUE, Guillaume. Machine Learnia. 2021.

ZEMOURI, Ryad A., RACOCEANU, Daniel, et ZERHOUNI, Noureddine. Réseaux de neurones récurrents à fonctions de base radiales: RRFR Application au pronostic. Rev. d'Intelligence Artif., 2002, vol. 16, no 3, p. 307-338.