Rapport de Veille (2024)

Les Réseaux Neuronaux

19 janvier 2024

Beroud Dylan

École d'ingénieurs du CESI, filière informatique

TABLE DES MATIÈRES

1 Introduction **2**
Un peu d'histoire 4

2 Perceptron **8**
Forward Propagation 9
Calcul de l'erreur 12
Backward Propagation 15
Résultat 18

3 Perceptron Multi-Couches **20**
Vectorisation des équations 21
Forward Propagation 24
Backward Propagation 28

4 Méthodologie **35**
Organisation des Données 36
Quel modèle choisir ? 39

5 Conclusion **43**

6 Code Source (Python) **45**

7 Bibliographie **53**

CHAPITRE 1

INTRODUCTION

L'évolution rapide de l'intelligence artificielle (IA) a révolutionné la manière dont les machines apprennent et accomplissent des tâches complexes, évoquant des possibilités insoupçonnées il y a encore quelques décennies. Au cœur de cette révolution se trouvent les **réseaux neuronaux**, des structures algorithmiques qui semblent être au coeur de ces innovations, allant de la **reconnaissance d'image** à la **traduction automatique**.

Toutefois, la magie opérée par les réseaux neuronaux suscite souvent plus de questions que de réponses. Comment ces systèmes complexes parviennent-ils à assimiler l'information et à prendre des décisions intelligentes ? Quels **mécanismes** sous-jacents guident leur apprentissage et leur compréhension du monde qui les entoure ? Cette veille s'attellera à **démystifier ces interrogations** cruciales, offrant une exploration approfondie du fonctionnement des réseaux neuronaux dans le domaine toujours en expansion de l'intelligence artificielle.

À travers cette analyse, nous plongerons dans l'anatomie virtuelle des réseaux neuronaux, dévoilant les principes fondamentaux qui sous-tendent leur capacité à apprendre, généraliser et résoudre des problèmes complexes.

Cette exploration nous guidera à travers les mécanismes de traitement de l'information, les différentes couches de neurones et les stratégies d'optimisation qui confèrent aux réseaux neuronaux leur pouvoir de calcul impressionnant.

Ainsi, nous verrons dans un premier temps le fonctionnement même d'un simple **neurone artificiel**, puis nous généraliserons pour arriver à un **réseau neuronal sur plusieurs couches** et nous finirons par une **méthodologie** pour bien exploiter cette nouvelle technologie.

Mais avant de rentrer dans le vif du sujet, il est important de noter que nous utiliserons les **termes anglophones** dans cette présentation. Bien évidemment, nous donnerons également les termes francophones pour des raisons évidentes, mais cela ne se limitera qu'à un simple rôle de traduction.

En effet, que ce soit dans le monde de l'ingénierie en intelligence artificielle ou tout simplement en termes de ressources disponibles, la plupart du temps, le vocabulaire employé se trouve être en anglais. Cette veille n'y fera donc pas exception.

Un peu d'histoire

Que ce soit pour la création des avions, l'utilisation des biocarburants ou de matériaux isolants, toutes ces inventions prennent racine dans une forme d'**inspiration de la Nature**, que ce soit respectivement par les ailes des oiseaux, le processus de photosynthèse ou la fourrure animale.

Le domaine de l'intelligence artificielle n'y fait absolument pas exception. Ce sont deux mathématiciens et neuro-scientifiques du nom de Warren McCulloch et Walter Pitts, qui en 1943 dans un article scientifique[1], expliquent comment ils ont programmé des neurones artificiels en s'inspirant de neurones biologiques.

Faisons un peu de rappel en biologie. Les neurones sont des cellules excitables, liées entre elles, composées de **dendrites**, d'un **corps cellulaire** et d'un **axone** et ayant pour rôle de **transmettre des informations** dans notre système nerveux.[2]

C'est au niveau des dendrites que l'information est reçue sous forme de signaux. Ces derniers peuvent être excitateurs (comme si nous recevions une valeur positive) ou inhibiteurs (comme si nous recevions une valeur négative). On peut voir les dendrites comme les **entrées** du neurone.

Ces signaux sont ensuite traités par le corps cellulaire. Si la somme de ceux-ci dépasse un certain seuil, le neurone va **s'activer** et produire un signal électrique.

Enfin, ce signal électrique va parcourir l'axone pour atteindre les autres neurones du système.

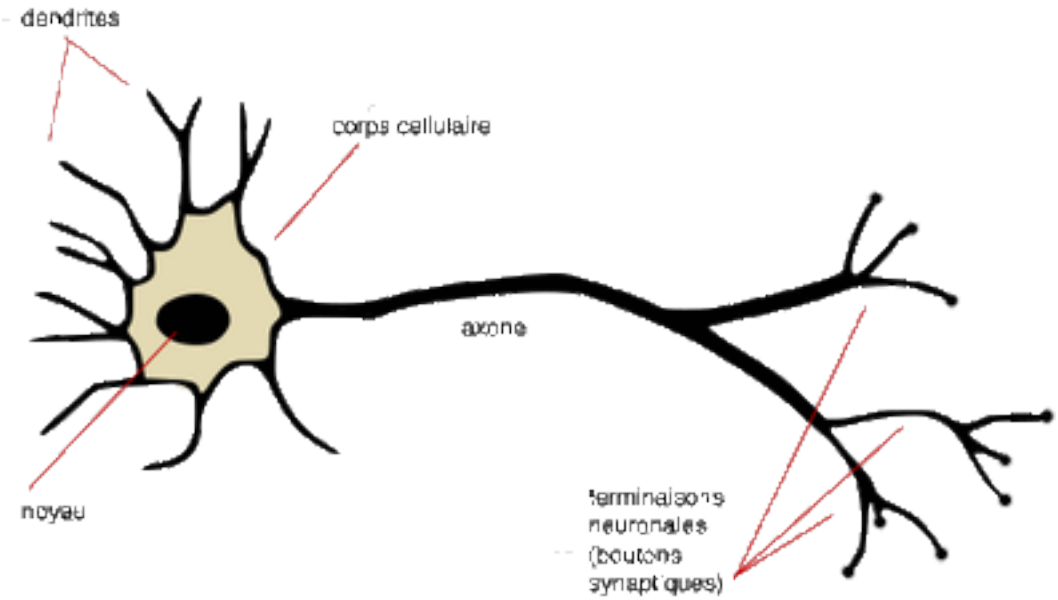


FIGURE 1.1 – Structure d'un neurone biologique.

Ainsi, ce que nos deux mathématiciens ont essayé de faire, c'est de **modéliser** ce schéma en considérant des entrées $(x_n)_{n\in\mathbb{N}}$, une fonction de transfert f et une sortie y, suivant le schéma ci-dessous.

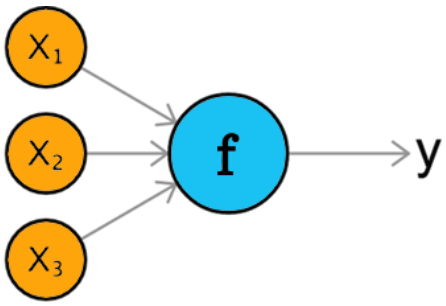


FIGURE 1.2 – Structure d'un neurone artificiel.

La fonction de transfert est simple, il s'agit d'une agrégation des entrées, auxquelles on y associe un **poids** spécifique. En d'autres termes, si on reprend notre schéma, en notant w_1, w_2, w_3 les poids associés respectivement aux entrées x_1, x_2, x_3, la fonction f est définie comme suit :

$$f(x_1, x_2, x_3) = w_1x_1 + w_2x_2 + w_3x_3$$

Enfin, pour déterminer la sortie y, la condition est simplement de savoir si f est positive ou négative, ainsi on écrit :

$$\begin{cases} y = 1 & \text{si } f(x_1, x_2, x_3) \geq 0, \\ y = 0 & \text{sinon} \end{cases}$$

Mais ce modèle comporte énormément de problèmes, notamment parce qu'il ne possède pas d'algorithme d'apprentissage. Les poids doivent être calculés manuellement pour déterminer les sorties voulues à partir de nos entrées.

C'est alors qu'en 1957, Frank Rosenblatt, propose non seulement une version améliorée du neurone artificiel, mais aussi d'un algorithme d'apprentissage. Ce nouveau modèle est appelé le **perceptron de Frank Rosenblatt**.

Son algorithme d'apprentissage est inspiré de la théorie de Hebb qui énonce que si deux neurones sont excités conjointement, alors leur connexion (ou leur **poids**) se renforcent. Ainsi, il détermine qu'il est possible d'**entraîner** un neurone artificiel sur des **données de références** (X, y), de sorte que celui-ci affine la valeur de ses poids.

Il ajoute alors une simple formule, permettant de modifier les poids en fonction de l'entrée, de la sortie trouvée et de la sortie attendue :

$$w_i = w_i + \alpha(y_{true} - y)x_i$$

avec w_i le poids associé à l'entrée x_i, α une valeur arbitraire nommée coefficient d'apprentissage, y_{true} la sortie attendue et y la sortie déterminée par le perceptron.

C'est alors qu'il a rendu possible ce qu'on appelle le **Machine Learning** (ou **Apprentissage Automatique** en français).

Bien évidemment, le **perceptron de Frank Rosenblatt** a été amélioré pour atteindre aujourd'hui un modèle relativement étendu sous le simple nom de **perceptron**. Ce modèle de neurone artificiel est la base même d'énormément de réseaux neuronaux, c'est pourquoi nous allons nous intéresser spécifiquement à celui-ci.

Nous pouvons ainsi rentrer dans le vif du sujet et expliquer en détail le fonctionnement d'un perceptron sur une seule couche, ce qui ne représente en réalité qu'un simple neurone artificiel.

CHAPITRE 2

PERCEPTRON

Pour ce chapitre, nous allons concevoir un percetron composé de 2 entrées nommées x_1, x_2 et d'une seule sortie nommée y.

L'objectif de notre perceptron est qu'il puisse classifier des données en deux groupes. On peut s'imaginer que nous essayons de classifier deux types de plantes à partir de deux données : la taille de sa tige et la taille de ses pétales par exemple.

Le code source écrit en langage python sera disponible dans la section à son effet, au chapitre 6. Le code *PERCEPTRON_GENERATION* permet de générer des données aléatoires pour nos plantes fictives et de les classer en deux groupes. Ces données serviront ensuite pour l'apprentissage de notre perceptron.

Forward Propagation

Dans un premier temps, l'information va devoir passer à travers le réseau neuronal. Cette étape se nomme la **Forward Propagation** (ou **Propagation Avant** en français). Dans celle-ci, les entrées vont passer d'un neurone à l'autre jusqu'à enfin atteindre la sortie.

Pour un perceptron, la **fonction de transfert** est la même que celle du perceptron de Frank Rosenblatt à la différence près que notre neurone artificiel va posséder ce qu'on appelle un **biais**, que l'on note b.

Ainsi, cette fonction, généralement notée z, est définie par :

$$z = w_1 x_1 + w_2 x_2 + b$$

En rappelant que w_1, w_2 sont les poids associés respectivement aux entrées x_1, x_2 par le neurone.

Cependant, nous remarquons que la fonction de transfert n'est qu'une **fonction affine**. Au mieux, par composition, dans un réseau neuronale, il nous sera possible de créer une sortie sous forme d'un polynôme d'un certain degré. Mais lorsque nous voulons résoudre des problèmes non-linéaires, tels que de la classification d'images, de la traduction automatique, et bien d'autres, il est important d'introduire de la non-linéarité.

Ainsi à l'image des neurones biologiques, on utilise une autre fonction appelée **fonction d'activation** qui permet de donner une sortie non-linéaire.

Il en existe plusieurs types, par exemple, la fonction **ReLU** ("Rectified Linear Unit" ou "Unité de rectification linéaire") permet de garder les valeurs de la fonction de transfert, mais en ajoutant simplement une couche de non-linéarité.

Elle se définie simplement par :

$$ReLU(z) = \begin{cases} z & \text{si } z \geq 0 \\ 0 & \text{sinon} \end{cases}$$

Cependant, l'un des grands désavantages de cette fonction, c'est son ensemble d'arrivée : $\mathbb{R}_+$.

Cela peut engendrer de gros problèmes d'**overflow** (ou de **débordement**) car à mesure que les neurones s'envoient leurs activations, les valeurs ne cessent de grandir, atteignant un seuil trop élevé par rapport à ce qu'un ordinateur peut garder en mémoire.

Ainsi, d'autres fonctions qui ont la particularité d'avoir un ensemble d'arrivée beaucoup plus petit existent comme la fonction **sigmoïde** ou la **tangente hyperbolique**.

Cependant le désavantage de ces méthodes est aussi leur plage de valeur très petite qui cause des problèmes de **vanishing gradient** (ou **gradient qui disparaît**) lors de la **back propagation**, mais nous reviendrons sur cette partie là plus tard.

Pour notre perceptron, nous utiliserons la fonction **sigmoïde** définie sur $\mathbb{R}$ dans $[0; 1]$ car, pour notre exemple, cette activation permet de répondre plus logiquement à un problème de classification binaire.

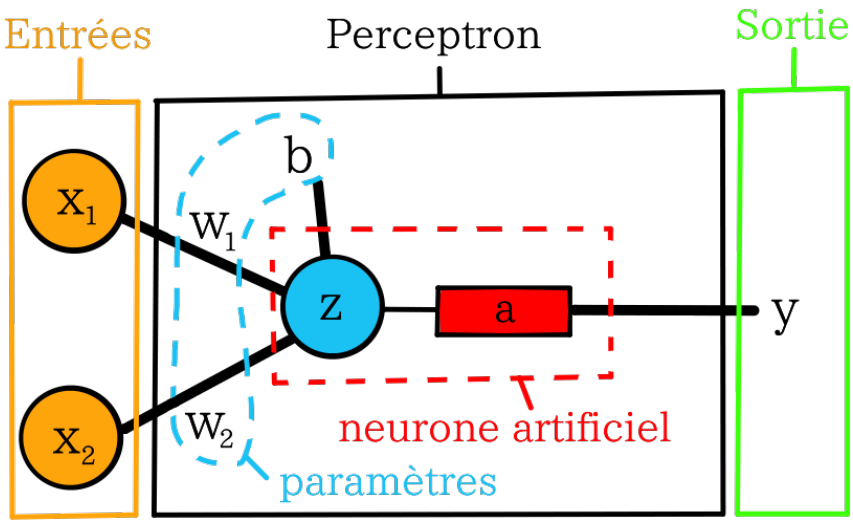
Notre fonction d'activation, notée généralement a, se définit alors par :

$$a = \frac{1}{1 + e^{-z}}$$

Enfin, après avoir obtenu notre activation, il nous est possible de définir une sortie y par une simple comparaison. On définit donc arbitrairement :

$$y = \begin{cases} 1 & \text{si } a \geq 0.5 \\ 0 & \text{sinon} \end{cases}$$

Pour résumer, voici un schéma ci-dessous qui illustre notre perceptron sur une seule couche, prenant deux entrées x_1, x_2 et renvoyant une sortie y.



Les équations importantes permettant la mise en oeuvre de ce système sont les suivantes :

$$z = w_1 x_1 + w_2 x_2 + b \qquad a = \frac{1}{1 + e^{-z}} \qquad y = \begin{cases} 1 & \text{si } a \geq 0.5 \\ 0 & \text{sinon} \end{cases}$$

À la section associée, au nom de *PERCEPTRON_FORWARD_PROPAGATION*, vous pouvez retrouver un script en langage python mettant en oeuvre ce processus.

Calcul de l'erreur

Une partie très importante dans l'apprentissage de notre perceptron est le **calcul de l'erreur**. Il faut une manière de **quantifier** à quel point notre modèle s'est trompé par rapport aux valeurs attendues.

Pour ce faire, nous allons avoir besoin de plonger dans le monde des probabilités. Vous pouvez percevoir notre fonction d'activation comme renvoyant la probabilité qu'une observation appartienne à une classe particulière, disons la classe où $y = 1$. Imaginons que notre fonction d'activation donne un résultat de $a = 0.9$; cela pourrait être interprété comme notre perceptron indiquant avec une certitude de 90% que la fleur appartienne à la classe $y = 1$.

Mais cela signifie également qu'il est sûr à 10% que la fleur appartienne à la classe $y = 0$, car dans une classification binaire, les événements "une plante appartient à la classe 0" et "une plante appartient à la classe 1" forment une partition complète de l'univers des classes possibles.

Ainsi, les sorties possibles Y suivent une loi de Bernoulli, de paramètres $p = a$, notre fonction d'activation. La probabilité que notre système ne se trompe pas, c'est-à-dire, que Y prenne la valeur de la sortie y de notre perceptron est donc définie par :

$$P(Y = y) = a^y * (1 - a)^{1-y}$$

Pour mieux comprendre, nous pouvons simplement faire une disjonction des cas. La probabilité que la plante appartienne à la classe 1 est $P(Y = 1)$, et c'est bien égal à a. Réciproquement, $P(Y = 0) = 1 - a$.

Ensuite, nous entraînons notre perceptron à l'aide d'un ensemble de données, ou d'**échantillons**. Nous noterons m le nombre d'échantillons fourni à notre neurone artificiel pour l'entraînement.

Ce que nous cherchons à faire, c'est que notre modèle ait juste sur l'ensemble des données fournies simultanément. Durant la phase de forward propagation, il n'y a pas de modification de ses propres poids et donc les sorties associées à chaque fleur envoyée en entrée sont indépendantes.

Définissons alors $Y_1, Y_2, ..., Y_m$ les variables aléatoires des sorties possibles pour les m plantes. La probabilité que les $y_1, y_2, ..., y_m$ sorties soient en accord avec la bonne classe de plantes se note $P(Y_1 = y_1 \cap Y_2 = y_2 \cap ... \cap Y_m = y_m)$.

Puisque, comme nous l'avons énoncé précédemment, ces évènements sont tous indépendants, alors il en résulte la formule suivante :

$$P(Y_1 = y_1 \cap Y_2 = y_2 \cap ... \cap Y_m = y_m) = \prod_{i=1}^{m} P(Y_i = y_i) = \prod_{i=1}^{m} a_i^{y_i} * (1 - a_i)^{1-y_i}$$

Nous définissons alors la **vraissemblance** comme étant cette probabilité. Cependant, un problème crucial découle de cette vraissemblance : étant le produit de nombres inférieurs à 1, à mesure que nous rajoutons des données d'entraînement (donc que m est grand), le résultat tend vers 0.

Pour pallier à ce problème, il est intéressant d'appliquer une fonction **logarithmique** à ce produit, car cela va permettre de le transformer en somme. De plus, étant une fonction strictement monotone sur $\mathbb{R}_+^*$, l'ordre est conservé par l'application de cette fonction, ce qui fait que chercher à **maximiser la vraisemblance** revient également à **maximiser son logarithme**.

Écrivons alors VL ce résultat, pour vraissemblance logarithmé. Nous avons alors :

$$VL = log(\prod_{i=1}^{m} a_i^{y_i} * (1 - a_i)^{1-y_i}) = \sum_{i=1}^{m} y_i log(a_i) + (1 - y_i) log(1 - a_i)$$

Ensuite, en informatique, il n'existe pas vraiment d'algorithme de **maximisation**, mais plutôt de **minimisation**. Cela ne pose pas vraiment de problème car **maximiser une fonction** revient à **minimiser son opposé**. Et enfin, pour pallier au problème d'une somme éventuellement trop grande à mesure que l'on donne plus de données, il suffit de la normaliser en la multipliant par le facteur $\frac{1}{m}$.

On définit alors le **Log Loss** (ou **fonction d'erreur logarithmé** en français) par la relation suivante :

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

À la section associée, au nom de *PERCEPTRON_LOG_LOSS*, vous pouvez retrouver un script en langage python permettant le calcul de cette erreur.

Backward Propagation

Après avoir réussi à quantifier l'erreur de notre perceptron, il va nous être possible de l'entraîner grâce à un algorithme nommé la **descende de gradient**.

Nous allons partir de l'erreur calculée pour ensuite revenir en arrière jusqu'à chaque paramètre de notre système, pour pouvoir enfin les modifier. C'est suite à cette "vague en arrière" que cette partie de la réalisation de notre modèle s'appelle la **backward propagation**.

Ce que nous nous apprêtons à faire, c'est de déterminer à quel point l'erreur changerait si on modifie un certain paramètre. Ensuite, il suffit de modifier ce paramètre en fonction : si l'erreur ne diminue pratiquement pas, c'est que ce paramètre est bien adapté, sinon au contraire, il faut le modifier.

On peut d'ores et déjà comprendre que cela va impliquer l'utilisation des dérivées. En effet, nous modifierons nos paramètres sur la base suivante :

$$w_1 = w_1 - \alpha * \frac{\partial \mathcal{L}}{\partial w_1}, \qquad w_2 = w_2 - \alpha * \frac{\partial \mathcal{L}}{\partial w_2}, \qquad b = b - \alpha * \frac{\partial \mathcal{L}}{\partial b}$$

Comme dans le modèle de Frank Rosenblatt, α est une constante arbitraire nommée coefficient d'apprentissage.

Les seules inconnues qu'ils nous restent à déterminer pour mettre à jour nos paramètres sont les dérivées partielles suivantes : $\frac{\partial \mathcal{L}}{\partial w_1}$, $\frac{\partial \mathcal{L}}{\partial w_2}$ et $\frac{\partial \mathcal{L}}{\partial b}$.

Pour ce faire, on décompose les dérivées partielles comme suit :

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial a} * \frac{\partial a}{\partial z} * \frac{\partial z}{\partial w_1}, \qquad \frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial a} * \frac{\partial a}{\partial z} * \frac{\partial z}{\partial w_2}, \qquad \frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial a} * \frac{\partial a}{\partial z} * \frac{\partial z}{\partial b}$$

On remarque que le terme $\frac{\partial \mathcal{L}}{\partial a} * \frac{\partial a}{\partial z}$ est récurrent, on le notera alors dZ. Ainsi, nous avons :

$$dZ = \frac{\partial \mathcal{L}}{\partial a} * \frac{\partial a}{\partial z}, \qquad \frac{\partial \mathcal{L}}{\partial w_1} = dZ * \frac{\partial z}{\partial w_1}, \qquad \frac{\partial \mathcal{L}}{\partial w_2} = dZ * \frac{\partial z}{\partial w_2}, \qquad \frac{\partial \mathcal{L}}{\partial b} = dZ * \frac{\partial z}{\partial b}$$

Commençons alors simplement par calculer la dérivée partielle du **Log Loss** par rapport à la **fonction d'activation** a :

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

Ensuite, nous pouvons calculer la dérivée partielle de la **fonction d'activation** a par rapport à la **fonction de transfert** z :

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

On peut alors déterminer notre dZ :

$$dZ = \frac{\partial \mathcal{L}}{\partial a} * \frac{\partial a}{\partial z} = -\frac{1}{m}\sum_{i=1}^{m}(\frac{y_i}{a_i} - \frac{1-y_i}{1-a_i}) * a_i(1-a_i) = \frac{1}{m}\sum_{i=1}^{m}(a_i - y_i)$$

Enfin, nous calculons de manière triviale les dérivées partielles suivantes :

$$\frac{\partial z}{\partial w_1} = x_1, \qquad \frac{\partial z}{\partial w_2} = x_2, \qquad \frac{\partial z}{\partial b} = 1$$

En rassemblant toutes nos données, nous pouvons enfin déterminer nos inconnus :

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m}[(a_i - y_i) * x_1], \qquad \frac{\partial \mathcal{L}}{\partial w_2} = \frac{1}{m}\sum_{i=1}^{m}[(a_i - y_i) * x_2], \qquad \frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a_i - y_i)$$

En voyant ces équations, nous pouvons remarquer donc le problème du **vanishing gradient** comme énoncé précédemment. Si notre activation est suffisamment petite, il y a des chances pour que nos dérivées partielles le soient également et à cause d'erreurs de précisions, soit mises à 0, ce qui empêche alors tous les autres paramètres de notre système de se mettre à jour.

Résultat

Pour finir, nous avons réussi à mettre en place un **perceptron** sur une seule couche, prenant **2 entrées** et renvoyant une **seule sortie**. Celui-ci est capable, après un entraînement, de classifier deux types de données distinctes en fonction de deux paramètres qui leur sont caractéristiques.

Un programme python permettant la finalisation de ce perceptron avec l'étape de la backward propagation se trouve dans la section associée, sous le nom de *PERCEPTRON*.

Voici par exemple le tracé d'une ligne de séparation que notre perceptron a développé suite à un entraînement sur 1000 itérations, avec 128 fleurs distinctes :

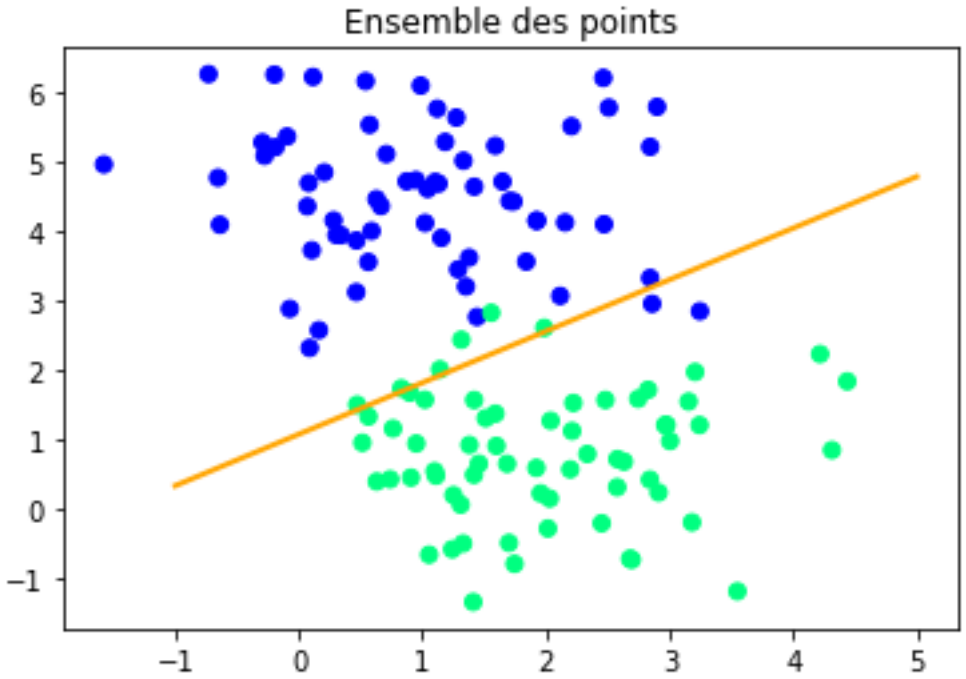
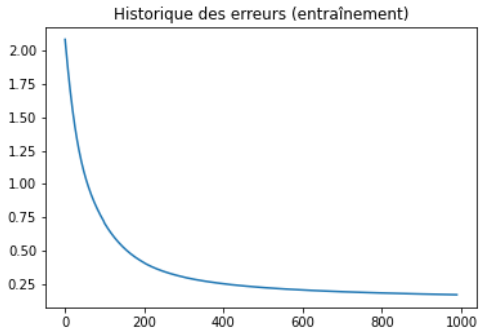
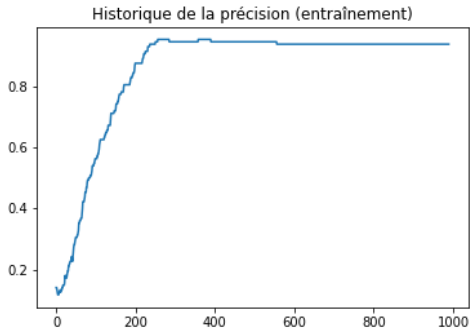


FIGURE 2.1 – Performance d'un perceptron.

Ce n'est certes pas parfait, mais réussir une telle prouesse avec un seul neurone artificiel est déjà un bel exploit !

Il est également possible de déterminer une **courbe d'apprentissage**, au fil des itérations, en voici donc l'évolution des erreurs et de la précision de notre perceptron :





La création de ce perceptron a également fait l'objet d'un mini-projet dont l'entièreté du code source se trouve dans le répositoire Github associé à ce document, dans le dossier *Code_Source\Perceptron*.

CHAPITRE 3

PERCEPTRON MULTI-COUCHES

Pour ce chapitre, nous allons concevoir un percetron multi-couches, prenant également deux entrées nommées x_1, x_2 et renvoyant une seule sortie nommée y.

Nous nous fixerons les mêmes objectifs, à la différence près, que la répartition des données des deux types de fleurs sera beaucoup moins linéaire :

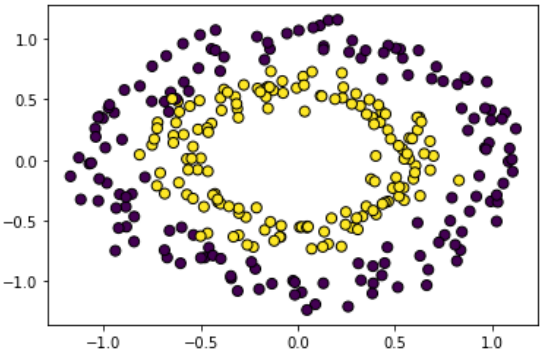


FIGURE 3.1 – Données d'Entraînement.

Le code de la génération de ces données est disponible dans la section à son effet, sous le nom de *MLP_GENERATION*.

Vectorisation des équations

Avant de se lancer pleinement dans la réalisation de notre perceptron multi-couches (ou **MLP** pour Multi-Layer Perceptron), il est important d'aborder la question de la vectorisation des équations.

Jusqu'à présent, nous avons traité chaque variable comme comportant une unique valeur. Étant donné le peu de variables que nous avions, les équations n'étaient clairement pas compliquées à appliquer. Mais imaginons maintenant que notre réseau neuronal prenne en entrée une image de 1024x1024 pixels. Cela fait un total de plus d'un million de variables à traiter. C'est simplement inimaginable de mettre en place un tel système à la main.

C'est là qu'intervient le concept de **vectorisation des équations**. Au lieu de voir les entrées comme des variables séparées, on peut les représenter comme une **matrice** d'un certain nombre d'éléments. Ainsi, il suffit simplement de traiter **une seule matrice**.

Plus concrètement, au lieu de considérer les entrées x_1 et x_2, on va considérer la **matrice d'entrées** $X = \begin{pmatrix} x_1 & x_2 \end{pmatrix}$.

Mais un autre problème se pose alors si nous utilisons cette simple notation : notre MLP va s'entraîner non pas sur une seule fleur, mais un nombre m d'entre elles. Et c'est justement là que les matrices sont d'autant plus pratiques. Il suffit simplement de voir X comme une matrice de $\mathcal{M}_{m,2}(\mathbb{R})$.

Introduisons la notation $x_1^{(i)}, i \in [1, m]$ étant la valeur de x_1 pour la i-ième plante et réciproquement, $x_2^{(i)}, i \in [1, m]$ étant la valeur de x_2 pour la i-ième plante.

Nous pouvons représenter notre matrice d'entrées ainsi :

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

Pour la suite, nous allons redéfinir certaines opérations matricielles, car utilisant le langage Python, la librairie numpy nous permet d'exécuter d'autres opérations qui nous facilitent énormément le travail et qui orientera donc notre recherche dans cette direction.

Comme vous le savez probablement, la somme de deux matrices A et B de **même dimension** s'effectue en ajoutant chaque élément de la matrice A à chaque élément homologue de la matrice B. Par exemple $\begin{pmatrix} 1 & 2 \\ 0 & 3 \end{pmatrix} + \begin{pmatrix} -1 & 5 \\ 7 & -5 \end{pmatrix} = \begin{pmatrix} 0 & 7 \\ 7 & -2 \end{pmatrix}$.

Cependant, il nous est possible d'ajouter **une matrice et un vecteur**, suivant l'exemple suivant :

$$\begin{pmatrix} a_1 & a_2 & \dots & a_n \\ b_1 & b_2 & \dots & b_n \end{pmatrix} + \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a_1 + c & a_2 + c & \dots & a_n + c \\ b_1 + d & b_2 + d & \dots & b_n + d \end{pmatrix}$$

De la même manière, en ce qui concerne le produit, on utilisera l'opérateur $\times$ pour parler du **produit matriciel**, définit par :

$$A = (a_{ij})_{(i,j)\in[1,n]\times[1,p]} \text{ et } B = (b_{ij})_{(i,j)\in[1,p]\times[1,m]}, \qquad A \times B = \sum_{k=1}^{p} a_{ik} b_{km}$$

Exemple : $\begin{pmatrix} 1 & 2 \\ 0 & 3 \end{pmatrix} \times \begin{pmatrix} -1 & 5 \\ 7 & -5 \end{pmatrix} = \begin{pmatrix} 1*(-1)+2*7 & 1*5+2*(-5) \\ 0*(-1)+3*7 & 0*5+3*(-5) \end{pmatrix} = \begin{pmatrix} 13 & -5 \\ 21 & -15 \end{pmatrix}$

On peut également multiplier une matrice par un **scalaire** de la manière suivante :

Soient $\lambda \in \mathbb{R}$ et $A = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{m1} & a_{m2} & ... & a_{mn} \end{pmatrix}$, alors $\lambda A = \begin{pmatrix} \lambda a_{11} & \lambda a_{12} & ... & \lambda a_{1n} \\ \lambda a_{21} & \lambda a_{22} & ... & \lambda a_{2n} \\ ... & ... & ... & ... \\ \lambda a_{m1} & \lambda a_{m2} & ... & \lambda a_{mn} \end{pmatrix}$

Et enfin, à l'image de la somme, il est également possible grâce à numpy de multiplier **une matrice avec un vecteur** avec l'opérateur $*$:

$$\begin{pmatrix} a_1 & a_2 & ... & a_n \\ b_1 & b_2 & ... & b_n \end{pmatrix} * \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a_1 c & a_2 c & ... & a_n c \\ b_1 d & b_2 d & ... & b_n d \end{pmatrix}$$

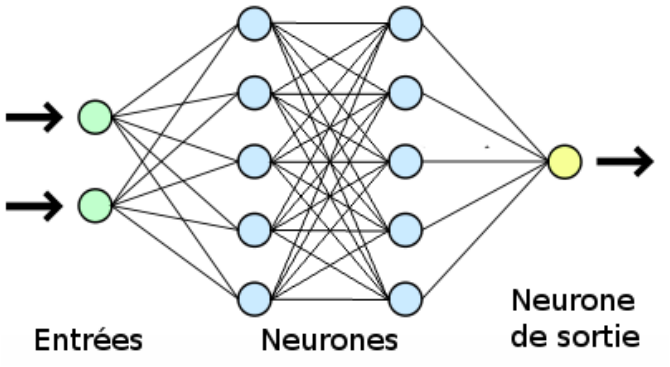
Maintenant que nous avons bien mis en place le vocabulaire et les opérations que nous utiliserons, il sera important tout au long de l'étude de faire attention aux dimensions des matrices que nous utiliserons.

Nous pouvons à présent étudier ensemble la **forward propagation** d'un MLP.

Forward Propagation

Lors de la **forward propagation** dans un réseau neuronal, nous distinguons trois parties : les **entrées**, les **neurones artificiels** et les **sorties**.

Pour passer d'un neurone à l'autre, il suffit simplement de relier les **entrées** d'un neurone à la **fonction d'activation** d'un autre neurone. Voici donc un schéma typique du MLP que nous allons mettre en place :



Nous pouvons alors procéder par récurrence pour déterminer le passage de l'information : d'abord il y a le passage des **entrées aux neurones** puis des **neurones d'une couche i aux neurones de la couche $i+1$**.

Il est également très important de définir l'architecture de notre réseau neuronal. Dans notre cas, nous allons bêtement relier tous les neurones d'une couche à tous les neurones de la couche suivante.

Pour caractériser notre réseau, notons la suite $(l_n)_{n\in\mathbb{N}}$, dont le terme l_i représente le nombre de neurones à la couche i. l_0 représente le nombre d'entrées et l_n le nombre de sorties. Nous noterons également n le nombre de couches de notre réseau.

Les couches qui ne représentent ni les entrées ni les sorties sont appelées des **hidden layer** (ou **couches cachées**).

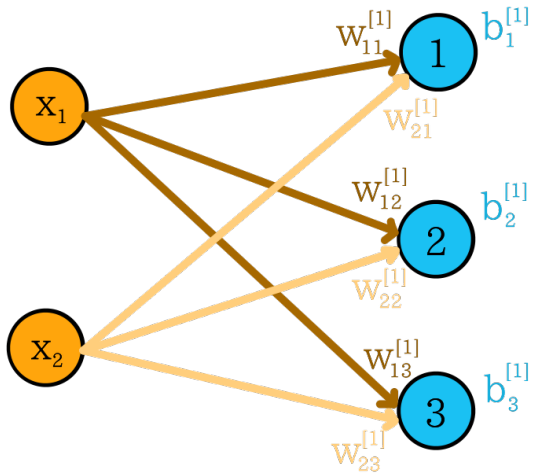
Nous allons devoir définir de nouvelles variables, cette fois-ci matriciellement, pour caractériser les poids, les biais, les fonctions de transferts et les activations de notre réseau.

Nous noterons $W^{[i]}$, $b^{[i]}$, $Z^{[i]}$, $A^{[i]}$ pour $1 \leq i \leq n$ respectivement l'ensemble des poids, des biais, des fonctions de transfert et des fonctions d'activation de la couche i.

Par exemple, pour la couche 1, nous avons les poids et biais suivants :

$$W^{[1]} = \begin{pmatrix} w_{11}^{[1]} & w_{12}^{[1]} & \dots & w_{1l_1}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & \dots & w_{2l_1}^{[1]} \end{pmatrix} \in \mathcal{M}_{2,l_1}(\mathbb{R}) \qquad b^{[1]} = \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \dots \\ b_{l_1}^{[1]} \end{pmatrix} \in \mathcal{M}_{l_1,1}(\mathbb{R})$$

Où $w_{1i}^{[1]}$ correspond au poids de l'entrée x_1 pour le neurone i de la couche 1, $w_{2i}^{[1]}$ correspond au poids de l'entrée x_2 pour le neurone i de la couche 1, et $b_i^{[1]}$ correspond au biais du neurone i de la couche 1. Voici un exemple de la couche 1 pour $l_1 = 3$:



Maintenant, étudions comment se comporte la fonction de transfert pour la couche 1. Nous aurons autant de fonctions de transfert qu'il y a de neurones et ceux, sur m échantillons. Ainsi nous avons la matrice suivante :

$$Z^{[1]} = \begin{pmatrix} z_1^{[1](1)} & z_2^{[1](1)} & \dots & z_{l_1}^{[1](1)} \\ z_1^{[1](2)} & z_2^{[1](2)} & \dots & z_{l_1}^{[1](2)} \\ \dots & \dots & \dots & \dots \\ z_1^{[1](m)} & z_2^{[1](m)} & \dots & z_{l_1}^{[1](m)} \end{pmatrix} \in \mathcal{M}_{m,l_1}(\mathbb{R})$$

Or $z_i^{[1](k)}$ pour $1 \leq i \leq l_1$ et $1 \leq k \leq m$ correspond à la fonction de transfert du i-ième neurone de la couche 1 pour le k-ième échantillon. Ainsi $z_i^{[1](k)} = x_1^{(k)} * w_{1i}^{[1]} + x_2^{(k)} * w_{2i}^{[1]} + b_i^{[1]}$.

Vous l'aurez peut-être reconnu, mais le terme $x_1^{(k)} * w_{1i}^{[1]} + x_2^{(k)} * w_{2i}^{[1]}$ correspond au produit matriciel $X \times W^{[1]}$. Ensuite il suffit d'ajouter le vecteur $b^{[1]}$.

Nous obtenons alors une équation matricielle bien plus élégante que voici :

$$Z^{[1]} = X \times W^{[1]} + b^{[1]}$$

On peut également vérifier les dimensions de nos matrices. X étant une matrice de $\mathcal{M}_{m,2}(\mathbb{R})$ et $W^{[1]}$ une matrice de $\mathcal{M}_{2,l_1}(\mathbb{R})$, leur produit matriciel donne une matrice de $\mathcal{M}_{m,l_1}(\mathbb{R})$. Ajouter le vecteur $b^{[1]}$ de dimension l_1 ne change pas la dimension finale, et l'on vérifie bien que $Z^{[1]} \in \mathcal{M}_{m,l_1}(\mathbb{R})$.

Ensuite, appliquer la fonction sigmoïde ne change également pas les dimensions de la matrice, donc on a également :

$$A^{[1]} = \frac{1}{1 + e^{-Z^{[1]}}} \in \mathcal{M}_{m,l_1}(\mathbb{R})$$

Il ne nous reste plus qu'à déterminer la **relation de récurrence** d'une couche i à la couche $i+1$.

Utilisons alors la même logique. Nos poids et nos biais sur la couche i sont les suivants :

$$W^{[i]} = \begin{pmatrix} w_{11}^{[1]} & w_{12}^{[1]} & ... & w_{1l_i}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & ... & w_{2l_i}^{[1]} \\ ... & ... & ... & ... \\ w_{l_{i-1}1}^{[1]} & w_{l_{i-1}2}^{[1]} & ... & w_{l_{i-1}l_i}^{[1]} \end{pmatrix} \in \mathcal{M}_{l_{i-1},l_i}(\mathbb{R}) \qquad b^{[i]} = \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \\ ... \\ b_{l_i}^{[1]} \end{pmatrix} \in \mathcal{M}_{l_i,1}(\mathbb{R})$$

Cette fois-ci en revanche, les entrées de nos neurones ne seront pas la matrice X, mais la matrice $A^{[i-1]}$, étant les activations de la couche précédente. Cette matrice est de dimension $m \times l_{i-1}$ car elle possède m échantillons des l_{i-1} neurones de la couche précédente.

Étant donné que nous appliquons les mêmes opérations pour chaque neurone, on s'attend à ce que la relation de récurrence soit du type :

$$Z^{[i]} = A^{[i-1]} \times W^{[i]} + b^{[i]}, \text{ pour } i \in [2, n]$$

Vérifions alors si les dimensions de nos matrices collent avec cette équation. D'une part, nous nous attendons à ce que $Z^{[i]}$ soit de dimension $m \times l_i$, car nous avons m échantillons pour l_i neurones.

D'autre part, le produit matriciel de $A^{[i-1]}$ par $W^{[i]}$ donne une matrice de dimension $m \times l_i$. En ajoutant le vecteur $b^{[i]}$ de dimension l_i, on ne change pas la dimension de la matrice finale. Ainsi, on a bien $(A^{[i-1]} \times W^{[i]} + b^{[i]}) \in \mathcal{M}_{m,l_i}(\mathbb{R})$.

Et enfin, pour déterminer la matrice des fonctions d'activation de la couche i, il suffit simplement d'appliquer aux fonctions de transferts, la fonction sigmoïde :

$$A^{[i]} = \frac{1}{1 + e^{-Z^{[i]}}}$$

Ainsi, nous avons réalisé l'étape de **forward propagation**. À la section associée, au nom de *MLP_FORWARD_PROPAGATION*, vous pouvez retrouver un script en langage python mettant en oeuvre ce processus.

Backward Propagation

Maintenant que nous avons parcouru l'ensemble du système, il va nous falloir le remonter pour lui permettre d'ajuster ses paramètres.

Dans un premier temps, il nous sera utile de calculer l'erreur. Rappelons la formule que nous utilisions pour le perceptron sur une seule couche :

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

Nous avons transformé nos activations pour obtenir les matrices $A^{[i]}$. De la même manière, nos sorties $y^{(1)}, y^{(2)}, ..., y^{(m)}$ pour chaque échantillon peuvent être représentées par la matrice $Y = \begin{pmatrix} y^{(1)} & y^{(2)} & ... & y^{(m)} \end{pmatrix}$

Heureusement pour nous, il existe des fonctions numpy qui permettent de prendre le logarithme d'une matrice et de faire la somme de chacun de ses éléments. Nous noterons alors notre nouveau **Log Loss** matriciellement :

$$\mathcal{L} = -\frac{1}{m}\sum Y * log(A^{[n]}) + (1 - Y) * log(1 - A^{[n]})$$

Pour la méthode de la descente des gradient, nous allons également procéder par récurrence pour trouver l'ensemble des **gradients** de nos paramètres. Cependant cette fois-ci également, nos équations seront des **équations matricielles**. Il n'est pas question de déterminer le gradient d'un certain poids $w_{ij}^{[k]}$, mais plutôt de l'ensemble des poids d'une couche i, donc la matrice $W^{[i]}$. Par chance, il est possible d'allier l'analyse différentielle à l'algèbre linéaire pour effectuer ces calculs !

Ainsi, pour chaque couche i, nous devons déterminer les gradients $\frac{\partial \mathcal{L}}{\partial W^{[i]}}$ et $\frac{\partial \mathcal{L}}{\partial b^{[i]}}$. Puisque nous procédons par récurrence, mais en partant de la fin, nous allons d'abord déterminer les gradients de la couche n, puis nous procéderons par récurrence en propageant les gradients de la couche i à la couche $i-1$.

Pour la couche n, nous devons déterminer $\frac{\partial \mathcal{L}}{\partial W^{[n]}}$ et $\frac{\partial \mathcal{L}}{\partial b^{[n]}}$. Or, nous pouvons décomposer ces dérivées partielles comme suit :

$$\frac{\partial \mathcal{L}}{\partial W^{[n]}} = \frac{\partial \mathcal{L}}{\partial A^{[n]}} * \frac{\partial A^{[n]}}{\partial Z^{[n]}} * \frac{\partial Z^{[n]}}{\partial W^{[n]}}, \qquad \frac{\partial \mathcal{L}}{\partial b^{[n]}} = \frac{\partial \mathcal{L}}{\partial A^{[n]}} * \frac{\partial A^{[n]}}{\partial Z^{[n]}} * \frac{\partial Z^{[n]}}{\partial b^{[n]}}$$

On remarque à nouveau le terme commun $\frac{\partial \mathcal{L}}{\partial A^{[n]}} * \frac{\partial A^{[n]}}{\partial Z^{[n]}}$ dans les deux équations, notons le alors dZ_n et commençons par le déterminer :

$$\frac{\partial \mathcal{L}}{\partial A^{[n]}} = \frac{\partial}{\partial A^{[n]}}(-\frac{1}{m}\sum Y*log(A^{[n]})+(1-Y)*log(1-A^{[n]})) = -\frac{1}{m}\sum \frac{Y}{A^{[n]}} - \frac{1-Y}{1-A^{[n]}}$$

$$\frac{\partial A^{[n]}}{\partial Z^{[n]}} = \frac{\partial}{\partial Z^{[n]}}(\frac{1}{1+e^{-Z^{[n]}}}) = -\frac{-e^{-Z^{[n]}}}{(1+e^{-Z^{[n]}})^2} = \frac{1+e^{-Z^{[n]}}}{(1+e^{-Z^{[n]}})^2} - \frac{1}{(1+e^{-Z^{[n]}})^2} = A^{[n]}(1-A^{[n]})$$

Ainsi, nous obtenons finalement une équation similaire à la **back propagation** du perceptron sur une seule couche : $dZ_n = -\frac{1}{m}\sum(\frac{Y}{A^{[n]}} - \frac{1-Y}{1-A^{[n]}}) * A^{[n]}(1-A^{[n]}) = \frac{1}{m}\sum A^{[n]} - Y$.

Cependant, lors de calculs d'équations matricielles, il faut **toujours** vérifier les dimensions de nos matrices. dZ_n représente une matrice des gradients des $Z^{[n]}$ par rapport à $\mathcal{L}$. Ainsi, la matrice dZ_n et $Z^{[n]}$ sont toutes deux de la même dimension $m \times l_n$.

Or si l'on regarde nos calculs, la matrice $A^{[n]}$ est également de dimension $m \times l_n$ et Y de dimension $1 \times l_n$, ce qui donne par soustraction une matrice de $m \times l_n$. Cependant, l'opérateur $\sum$ effectue la somme de tous les éléments de cette matrice pour obtenir finalement un simple nombre, ce que nous ne voulons pas. Ainsi, la formule final de notre dZ_n est la suivante :

$$dZ_n = \frac{1}{m} * (A^{[n]} - Y)$$

Pour déterminer finalement les gradients de $W^{[n]}$ et de $b^{[n]}$, il suffit de calculer les dérivées partielles suivantes : $\frac{\partial Z^{[n]}}{\partial W^{[n]}}$ et $\frac{\partial Z^{[n]}}{\partial b^{[n]}}$. Ce calcul s'effectue très simplement et l'on obtient respectivement $A^{[n-1]}$ et 1. Ainsi on détermine alors :

$$\frac{\partial \mathcal{L}}{\partial W^{[n]}} = dZ_n * A^{[n]}, \qquad \frac{\partial \mathcal{L}}{\partial b^{[n]}} = dZ_n$$

Vérifions à nouveau les dimensions. $dZ_n \in \mathcal{M}_{m,l_n}(\mathbb{R})$, $A^{[n-1]} \in \mathcal{M}_{m,l_{n-1}}(\mathbb{R})$ et $\frac{\partial \mathcal{L}}{\partial W^{[n]}}$ est de même dimension que $W^{[n]} \in \mathcal{M}_{l_{n-1},l_n}(\mathbb{R})$. Pour obtenir les mêmes dimensions, il suffit de transposer $A^{[n]}$ et d'effectuer le produit matriciel entre $A^{[n]}$ et dZ_n. En ce qui concerne la matrice $b^{[n]} \in \mathcal{M}_{l_n,1}(\mathbb{R})$, on va d'abord transposer dZ_n puis rassembler ses m lignes en une seule en utilisant l'opérateur $\sum_{\text{axe}=1}$.

Finalement, voici les formules associées :

$$\frac{\partial \mathcal{L}}{\partial W^{[n]}} = (A^{[n]})^T \times dZ_n, \qquad \frac{\partial \mathcal{L}}{\partial b^{[n]}} = \sum_{\text{axe}=1} dZ_n^T$$

Nous venons alors de déterminer le cas **initial**. Nous utiliserons le même raisonnement pour déterminer les couches plus basses. Nous pouvons définir la suite $(dZ_k)_{k\in[1,n]}$, représentant la dérivée partielle du **Log Loss** par rapport aux **fonctions de transfert de la couche** k par récurrence avec la formule suivante :

$$dZ_n = \frac{1}{m} * (A^{[n]} - Y), \qquad dZ_{i-1} = dZ_i * \frac{\partial Z^{[i]}}{\partial A^{[i-1]}} * \frac{\partial A^{[i-1]}}{\partial Z^{[i-1]}}$$

Or nous trouvons facilement que $\frac{\partial Z^{[i]}}{\partial A^{[i-1]}} = W^{[i]}$ et $\frac{\partial A^{[i-1]}}{\partial Z^{[i-1]}} = A^{[i-1]}(1 - A^{[i-1]})$. En passant les vérifications de dimension, et en rappelant que $\times$ est l'opérateur du produit matriciel et $*$ est un produit qui multiplie un à un les éléments de deux matrices, nous nous retrouvons avec la formule générale de cette suite :

$$dZ_n = \frac{1}{m} * (A^{[n]} - Y), \qquad dZ_{i-1} = [dZ_i \times (W^{[i]})^T] * A^{[i-1]}(1 - A^{[i-1]})$$

Enfin, pour déterminer les gradients de $W^{[i]}$ et de $b^{[i]}$, il suffit de décomposer nos dérivées partielles, et grâce aux calculs précédents, nous obtenons simplement :

$$\frac{\partial \mathcal{L}}{\partial W^{[i]}} = dZ_i * \frac{\partial Z^{[i]}}{\partial W^{[i]}}, \qquad \frac{\partial \mathcal{L}}{\partial b^{[i]}} = dZ_i * \frac{\partial Z^{[i]}}{\partial b^{[i]}}$$

Et en utilisant les mêmes procédés que précédemment, on détermine finalement :

$$\frac{\partial \mathcal{L}}{\partial W^{[i]}} = (A^{[i]})^T \times dZ_i, \qquad \frac{\partial \mathcal{L}}{\partial b^{[i]}} = \sum_{\text{axe}=1} dZ_i^T$$

Il est aussi important de noter, pour le bien de nos calculs, que $A^{[0]} = X$. Cela découle simplement d'un fait logique : finalement nos toutes premières activations ne sont que les entrées du réseau de neurones.

Je vous félicite si vous avez réussi à tout comprendre jusqu'ici, n'hésitez surtout pas à relire s'il y a encore quelques points qui vous paraîssent obscurs, car ce chapitre est très intense en termes d'abstraction et de mathématiques. Néanmoins, nous allons de ce pas faire un petit inventaire des formules utilisées pour la **backward propagation** :

Initialisation :

$$dZ_n = \frac{1}{m} * (A^{[n]} - Y) \qquad A^{[0]} = X$$

Récurrence ($\forall i \in [1, n]$) :

$$dZ_{i-1} = [dZ_i \times (W^{[i]})^T] * A^{[i-1]}(1 - A^{[i-1]})$$
$$\frac{\partial \mathcal{L}}{\partial W^{[i]}} = (A^{[i]})^T \times dZ_i$$
$$\frac{\partial \mathcal{L}}{\partial b^{[i]}} = \sum_{\text{axe}=1} dZ_i^T$$

À la section associée, se trouve au nom de *MLP_BACK_PROPAGATION* l'ensemble du programme réalisant la **back propagation**, puis au nom de *MLP* le programme qui réalise l'entraînement de notre modèle.

Après avoir entraîné notre **MLP** sur les 300 données simulées de fleurs réparties en deux groupes selon la manière suivante :

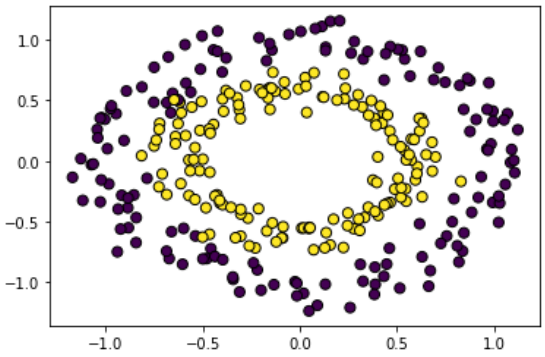
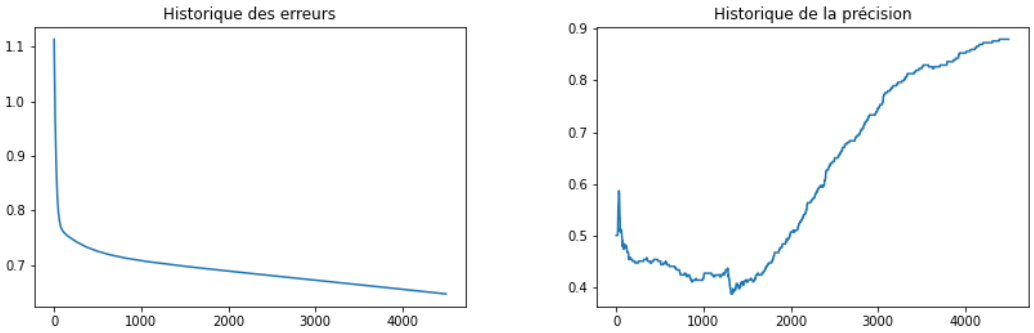


FIGURE 3.2 – Données d'Entraînement.

Celui-ci, après 5000 itérations de **forward propagation** puis **backward propagation**, et un réseau neuronal composé de **2 entrées**, une unique couche de **32 neurones** et **1 sortie**, a obtenu les courbes d'apprentissage suivantes :



Nous avons également pu tester notre **MLP** sur 500 données simulées de tests suivant une même répartition, dont le programme se trouve à section associée, au nom de *MLP_TEST*, et dont la visualisation est la suivante :

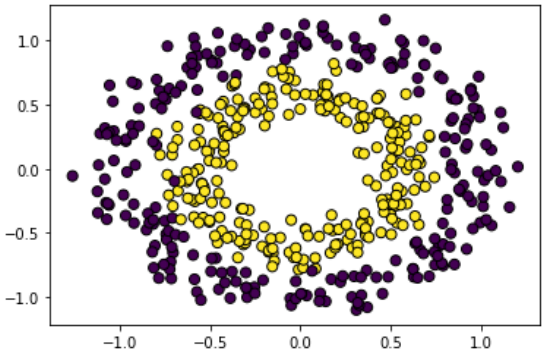


FIGURE 3.3 – Données de Test.

Enfin, comme pour le perceptron sur une seule couche, la création de ce MLP a également fait l'objet d'un mini-projet dont l'entièreté du code source se trouve dans le répositoire Github associé à ce document, dans le dossier *Code_Source\Multi-Layer-Perceptron*.

CHAPITRE 4

MÉTHODOLOGIE

Ce chapitre est fait à l'attention des personnes qui peuvent et utiliseront les réseaux neuronaux que ce soit ou non à but lucratif. L'un des plus grands aspects novateurs de l'IA, c'est justement sa capacité à apprendre. Nous venons de voir à quel point tout ceci ne tient que sur de simples équations mathématiques, mais qui n'en retirent pas l'aspect complexe des tâches qu'elles peuvent réaliser.

Notre réseau neuronal s'est non seulement approprié le schéma résultant des données que nous lui avons fournies, mais il a également été capable de **généraliser** pour classifier d'autres données suivant le même schéma.

Ainsi, nos MLP peuvent être utilisés dans des domaines comme par exemple la prédiction, ce qui peut être un atout majeur dans la gestion d'une entreprise. D'autres exemples bien plus connus et d'actualité comme la classification d'images, la traduction automatique, ou moins connus tels que la reconnaissance de billets dans une caisse automatique peuvent faire l'objet de l'utilisation d'une telle technologie.

Organisation des Données

Mais donc, comment peut-on réellement s'approprier cette technologie ? En effet, nous avons étudié en détail son fonctionnement, mais qu'en est-il de l'aspect pratique ?

Comme vous avez pu le remarquer pour la réalisation de nos perceptrons, des centaines de données ont dû être nourries à notre modèle pour qu'il s'entraîne correctement. En effet, la collecte de données est l'un des aspects majeurs de la mise en place d'une intelligence artificielle, et la gestion de celles-ci est tout aussi importante que la réalisation de l'IA en elle-même.

La faisabilité d'une IA dépend en faite de deux facteurs clés : le nombre de données que nous pouvons lui fournir et la puissance de calcul que nous pouvons lui accorder.

Dans cette section, nous verrons comment on peut utiliser les données pour entraîner efficacement notre réseau de neurones.

En effet, l'un des premiers aspects que l'on recherche dans une IA, c'est sa capacité à **généraliser** à partir d'un nombre de données réduit. Mais alors cela pose le problème du fait que l'IA ne peut pas être testée sur les mêmes données qui lui ont été fournies pour l'entraînement, car elle s'est déjà spécialisée à les reconnaitre.

Il est donc nécessaire de séparer les données que nous avons en deux parties : le **train set** (ou **données d'entraînement**) et le **test set** (ou **données de test**), à une répartition respective de 80% et 20%.[3]

Cependant, lorsque nous entraînons un réseau de neurones, nous pouvons **modifier son architecture** ou **choisir un modèle** plus adapté à nos besoins. Nous reviendrons sur cette partie dans la section suivante. Dans tous les cas, nous risquons de modifier certains paramètres initiaux pour guider notre IA vers la généralisation que nous recherchons.

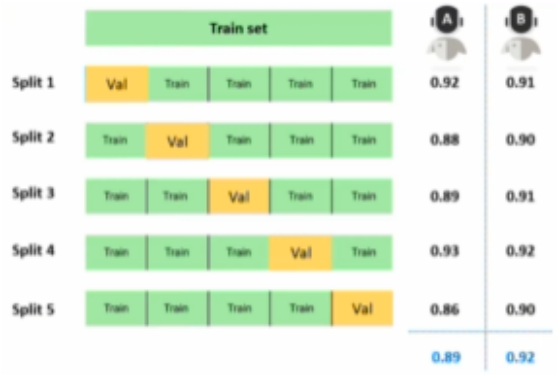
Pour ainsi dire, nous allons l'entraîner avec 80% de nos données, la tester avec les 20% restantes, mais si sa précision ne nous convient pas, nous allons modifier des paramètres initiaux pour qu'elle puisse avoir un meilleur **score** dans la phase de test.

Vous l'aurez peut-être compris, le problème derrière cela, c'est que nous risquons d'adapter notre **modèle** en fonction des données de test et finalement, faire la même erreur que si nous l'avions testée avec les données d'entraînements. Ce problème dans lequel notre modèle se spécialise trop sur des données d'entrainement s'appelle l'**overfitting**.

C'est pourquoi, nous découpons également en même proportion le **train set** pour en soutirer ce qu'on appelle le **validation set** (ou **données de validation**). Ensuite le processus est simple : nous allons entraîner plusieurs configurations de notre réseau de neurones artificiels avec le **train set**, les tester avec le **validation set** et ensuite, nous prenons le modèle ayant le meilleur **score** pour le tester avec le **test set** et se faire une idée de son efficacité dans un cas réel.

Mais encore une fois, d'autres problèmes se posent. Qu'est-ce qui nous garantit que la manière dont nous découpons nos données permettent un entraînement idéal pour notre modèle ?

Prenons l'exemple d'un modèle permettant de classifier divers groupes. Si dans le train set, un groupe est très représenté par rapport aux autres, notre modèle risque de s'adapter à celui-ci spécifiquement, ce qui risque d'influencer nos prédictions. C'est ainsi que nous faisons ce qu'on appelle de la **cross validation**[4] (ou **validation croisée**). Cette méthode consiste en un découpage du **train set** de sorte que nous puissons entraîner un même modèle avec différentes découpes possibles. Il suffit ensuite de prendre par exemple la moyenne des scores pour définir le score total de ce modèle en particulier. Voici un schéma peut-être plus parlant d'une **cross validation** :



Expliquer plus en détail les différentes méthodes de **cross validation** va au delà de l'étude de cette veille. Néanmoins, nous laissons à disposition le vocabulaire ainsi que les ressources supplémentaires permettant de se renseigner ultérieurement sur ces processus. Voici donc une liste non-exhaustive[5] de méthodes utilisées :

— **K-Fold**

— **Stratified K-Fold**

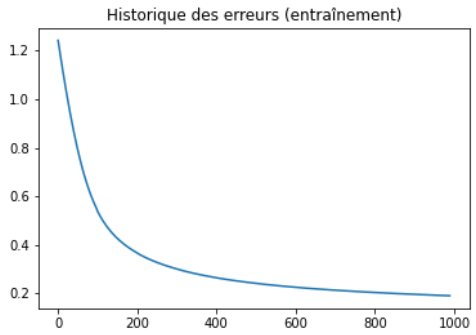
— **Group K-Fold**

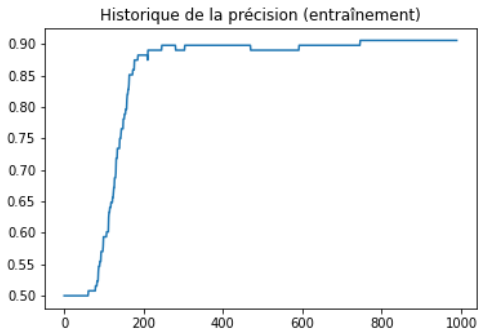
— **Shuffle Split**

Ensuite, d'autres notions sont très importantes lors de l'entraînement d'une intelligence artificielle, notamment les **learning curves** (ou **courbes d'apprentissage**).

En effet, il faut savoir quand est-ce que nos modèles ont atteint leur limite. Est-ce que fournir plus de données changera significativement la précision ou est-ce trop coûteux et donc faut-il réfléchir à concevoir un autre modèle ?

C'est pour cette raison qu'il est intéressant de pouvoir visualiser les **courbes d'erreurs** et de **précision** au fil de l'apprentissage. Celles-ci ressemblent généralement aux schémas ci-dessous. Nous voyons avec évidence qu'il y a un plafond à partir duquel fournir plus de données est trop coûteux par rapport aux gains de précision qu'elles pourraient accorder :



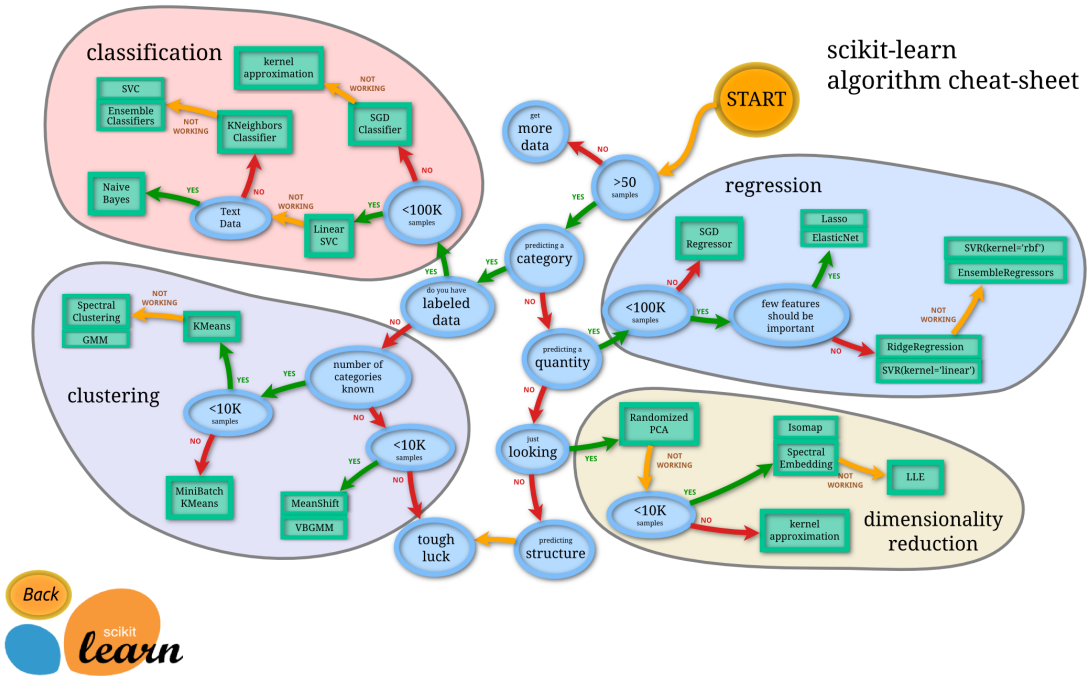


Quel modèle choisir ?

Enfin, après avoir expliqué l'importance d'étudier son réseau de neurones artificiels et d'une bonne gestion des données à lui fournir, il reste encore à expliquer les différents modèles qu'il existe.

Comme nous avons pu le remarquer dans la conception de notre MLP, l'architecture même au sein des hidden layers est importante : la manière dont nous connectons les neurones entre eux et les formules que nous appliquons en tant que fonction de transfert et d'activation peuvent permettre des apprentissages plus performants et spécialisés dans certains domaines. Il existe énormément de modèles, allant d'une simple **LinearRegression** spécialisée dans la prédiction de données suivant une simple droite, jusqu'à des modèles **BERT** permettant la "compréhension de textes".

La librairie Python nommée Scikit-Learn spécialisée dans l'apprentissage automatique propose une carte mentale permettant de choisir le modèle associé à nos besoins :



Nous avons vu le modèle du perceptron multi-couches, mais il existe par exemple le modèle de **réseaux neuronaux convolutifs** conçu pour traiter des données sous forme de grille telles que des images ou des motifs spatiaux (reconnaître un "3" dans une écriture manuscrite par exemple).

Il existe aussi ce que nous appelons des **réseaux neuronaux récurrents** permettant d'avoir des entrées "dynamiques" au sens où l'on peut envoyer des entrées au modèle au fil du temps. Ce genre de modèles sont utilisés pour ce qu'on appelle du **NLP** (**Natural Langage Processing**) qui représente le traitement du langage humain. Pour ce genre de modèles, d'autres systèmes tels que les **mécanismes d'attention**[6] sont aussi mis en place avec le réseau neuronal pour permettre de meilleurs résultats. Cela se base sur le fait que chaque mot dans une phrase n'a pas la même importance, et qu'il faut donc mettre une certaine **attention** envers certains mots.

Nous ne rentrerons pas dans les détails de ces architectures, mais des ressources consultables à la section *Bibliographie* y sont présentes.

Enfin et pour cloturer cette veille, nous pouvons parler légèrement d'optimisation. En informatique, c'est un des problèmes les plus fréquents : comment créer un algorithme capable d'effectuer un maximum de tâches sans effectuer énormément d'opérations élémentaires ?

Pour revenir à notre MLP, nous avons utiliser des fonctions de transfert affine. Sans l'utilisation de fonctions d'activations et si nous avions voulu traiter des problèmes de régressions polynomiales, il aurait été intéressant de se poser la question suivante : est-il plus intéressant de composer nos fonctions affines pour créer le polynôme voulu ou alors partir d'un polynôme général et en trouver ses paramètres ?

Nous pouvons résumer très grossièrement la question à la suivante : est-il plus avantageux de mettre en place un réseau neuronal utilisant des fonctions simples ou alors un seul neurone utilisant une fonction complexe ?

Lors de la back propagation, nous avons vu l'algorithme de la descente des gradients, mais bien évidemment il en existe plein d'autres. Pour répondre à ce problème d'optimisation, Arunachalam Gopalakrishnan, un doctorant en ingénierie électronique à l'université du Texas, propose de réfléchir à cette optimisation en **nombre de paramètres à entraîner**.

Dans sa thèse[7], il compare un MLP à un modèle de réseau neuronal utilisant les **polynômes de Gabor** et en déduit que ce dernier est bien plus efficace. Ajoutons à cela que l'utilisation de polynôme permet l'implémentation de l'algorithme des gradients conjugués étant bien plus efficace que la descente de gradients.

Il a également montré que lorsque l'on traite des polynômes de degré 2 ou 3, il existe **toujours** un moyen de trouver les bons paramètres initiaux, mais que ce n'était pas le cas pour des degrés pour élevés.

En somme, cela met bien en évidence l'importance de créer des modèles **spécialisés** pour nos besoins. Il est impératif de réfléchir à la manière dont nous devons **collecter et organiser nos données**, **choisir des modèles** ainsi que des **algorithmes d'optimisation** adaptés au problème à solutionner.

CHAPITRE 5

CONCLUSION

En conclusion, cette exploration détaillée des réseaux neuronaux, du simple perceptron aux architectures plus complexes telles que les perceptrons multi-couches, nous a permis de plonger dans le monde fascinant de l'apprentissage automatique. La compréhension approfondie du fonctionnement des neurones artificiels, des différentes couches et des mécanismes d'apprentissage a jeté les bases de notre appréhension des modèles de réseaux neuronaux.

La méthodologie présentée dans le chapitre dédié a souligné l'importance cruciale de l'organisation des données en **train set**, **validation set** et **test set**. La manipulation judicieuse de ces ensembles, associée à l'analyse des courbes d'apprentissage, constitue une étape incontournable dans la conception et l'évaluation rigoureuse des modèles neuronaux.

Cette veille nous offre une vision globale des outils et concepts fondamentaux nécessaires à la conception, à l'entraînement et à l'évaluation de modèles de réseaux neuronaux. Toutefois, l'univers en constante évolution de l'apprentissage automatique exige une vigilance continue, incitant à explorer de nouvelles architectures et à s'adapter aux dernières avancées technologiques.

En définitive, cette veille constitue une plongée profonde et éclairante dans le domaine des réseaux neuronaux, établissant une base solide pour des recherches futures et pour l'application pratique de ces connaissances dans des projets d'apprentissage automatique. La rencontre entre la théorie et la méthodologie renforce notre capacité à façonner l'avenir de l'intelligence artificielle et à relever les défis passionnants qui se profilent devant nous.

CHAPITRE 6

CODE SOURCE (PYTHON)

PERCEPTRON_GENERATION

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Initialisation de donnees aleatoires
X, y = make_blobs(n_samples=128, n_features=2, centers=2,
                                    random_state=0)
y = y.reshape((y.shape[0], 1))
# On recupere chaque entree separement
x_1 = X[:, 0]
x_2 = X[:, 1]
# Affichage du graphique associe
plt.scatter(x_1, x_2, c=y, cmap='summer')
plt.show()
```

PERCEPTRON_FORWARD_PROPAGATION

```
# Fonction qui permet la forward propagation
def forward_propagation(x_1, x_2, w_1, w_2, b):
    z = w_1 * x_1 + w_2 * x_2 + b
    a = 1/(1+np.exp(-z))
    return a

# Fonction renvoyant la sortie a partir de l'activation
def get_output(a):
    return (a >= 0.5)
```

PERCEPTRON_LOG_LOSS

```
# Fonction renvoyant le log loss a partir des sorties y et
# des activations a
def log_loss(y, a):
    # On definit une valeur toute petite pour ne pas avoir
    # d'erreurs lorsque la valeur dans le log est trop
    # petite.
    EPSILON = 1e-15
    # On recupere le nombre d'echantillon car y est une
    # matrice contenant m fleurs.
    m = len(y)
    # On calcule le cout
    L = -(1 / m) * np.sum(y * np.log(a + EPSILON) + (1 - y) *
                                        np.log(1 - a + EPSILON))
    # On renvoie le cout
    return L
```

PERCEPTRON

```
    # Fonction qui renvoie les gradients
    def get_gradients(x_1, x_2, a, y):
        dW1 = (1/len(y)) * np.sum(x_1 * (a - y)[0])
        dW2 = (1/len(y)) * np.sum(x_2 * (a - y)[1])
        db = (1/len(y)) * np.sum(a - y)
        return (dW1, dW2, db)

    # Fonction qui permet d'entrainer un perceptron, en renvoyant
    # sous forme de tuple ses differents parametres.
    def perceptron(x_1, x_2, y, learning_rate = 0.1, nb_iter = 100
                                        ):
        # Initialisation des poids et du biais
        w_1 = np.random.randn(1)
        w_2 = np.random.randn(1)
        b = np.random.randn(1)
        # Pour chaque iteration
        for i in range(nb_iter):
            """ FORWARD PROPAGATION """
            a = forward_propagation(x_1, x_2, w_1, w_2, b)
            """ BACKWARD PROPAGATION """
            dW1, dW2, db = get_gradients(x_1, x_2, a, y)
            w_1 = w_1 - learning_rate * dW1
            w_2 = w_2 - learning_rate * dW2
            b = b - learning_rate * db
        # On renvoie les parametres du perceptron
        return (w_1, w_2, b)
```

MLP_GENERATION

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

# On genere des donnees
X, Y = make_circles(n_samples=300, factor=0.6, noise=0.1,
                                    random_state=12)
X, Y = X.T, Y.reshape((1, Y.shape[0]))

# On affiche le set d'entrainement
print("Train set : ")
plt.scatter(X[0, :], X[1, :], c=Y, cmap='viridis', edgecolors=
                                    'k', marker='o', s=50)
plt.show()
```

MLP_FORWARD_PROPAGATION

```
# Initialisation du nombre de neurones pour chaque couche
dimensions = [X.shape[0], 32, Y.shape[0]]
# Fonction qui initialise les parametres
def init_params(dimensions):
    parameters = {}
    # On initialise les couches
    nbCouches = len(dimensions)
    for c in range(1, nbCouches):
        parameters['W' + str(c)] = np.random.randn(dimensions[
                                            c], dimensions[c-1])
        parameters['b' + str(c)] = np.random.randn(dimensions[
                                            c], 1)
    return parameters
# Fonction qui execute la forward propagation
# avec les entrees X et les parametres
# "parameters".
def forward_propagation(X, parameters):
    # Initialisation des activations
    activations = {'A0' : X}
    nbCouches = len(parameters) // 2
    for c in range(1, nbCouches + 1):
        Z = np.dot(parameters['W' + str(c)], activations['A' +
                                             str(c-1)]) +
                                            parameters['b' + str(
                                            c)]
        activations['A' + str(c)] = 1 / (1 + np.exp(-Z))
    # On renvoie les activations
    return activations
```

MLP_BACK_PROPAGATION

```
# Fonction qui execute la back propagation avec les entrees X,
# les sorties Y attendues, les parametres et les activations
# de chaque couche.
def back_propagation(X, Y, parameters, activations):
    # On recupere le nombre d'echantillons
    m = Y.shape[1]
    # Et le nombre de couche
    nbCouches = len(parameters) // 2
    # On calcule notre dZn
    dZ = activations['A' + str(nbCouches)] - Y
    # On initialise le dictionnaire
    gradients = {}

    # On bouche de la couche finale vers la
    # couche initiale
    for c in reversed(range(1, nbCouches + 1)):
        A = activations['A' + str(c-1)]
        # On calcule les gradients
        gradients['dW' + str(c)] = (1/m) * np.dot(dZ, A.T)
        gradients['db' + str(c)] = (1/m) * np.sum(dZ, axis=1,
                                           keepdims=True)
        # Il n'y a pas de dZ0
        if(c > 1):
            # On modifie le dZ pour la prochaine couche
            dZ = np.dot((parameters['W' + str(c)]).T, dZ) * A
                                                * (1 - A)
    # On renvoie le dictionnaire de gradients
    return gradients
```

MLP

```
    # Fonction qui permet d'entrainer un mlp, en renvoyant
    # sous forme de tuple ses differents parametres.
    def mlp(X, Y, dimensions, learning_rate = 0.01, nb_iter = 5000
                                        ):
        parameters = init_params(dimensions)
        nbCouches = len(parameters) // 2
        # Pour chaque iteration
        for i in range(nb_iter):
            """ FORWARD PROPAGATION """
            activations = forward_propagation(X, parameters)
            """ BACK PROPAGATION """
            gradients = back_propagation(X, Y, parameters,
                                         activations)
            # On met a jour les parametres
            for c in range(1, nbCouches+1):
                parameters['W' + str(c)] = parameters['W' + str(c)
                                                ] - learning_rate
                                                 * gradients['dW'
                                                 + str(c)]
                parameters['b' + str(c)] = parameters['b' + str(c)
                                                ] - learning_rate
                                                 * gradients['db'
                                                 + str(c)]

        return parameters
```

MLP_TEST

```
from sklearn.metrics import accuracy_score
# Fonction qui permet de predire les sorties a partir
# des entrees X et des parametres donnees.
def predict(X, parameters):
    nbCouches = len(parameters) // 2
    # On recupere les activations
    activations = forward_propagation(X, parameters)
    Af = activations['A' + str(nbCouches)]
    # On "True" (ou 1) si l'activation est superieure a 0.5,
    # et "False" (ou 0) dans le cas contraire
    return Af >= 0.5
# On entraine notre mlp
parameters = mlp(X, Y, [X.shape[0], 32, Y.shape[0]])
# On genere des donnees de test
newX, newY = make_circles(n_samples=500, factor=0.6, noise=0.1
                                    , random_state=69)
newX, newY = newX.T, newY.reshape((1, newY.shape[0]))
# On affiche le set de test
print("Test set : ")
plt.scatter(newX[0, :], newX[1, :], c=newY, cmap='viridis',
                                    edgecolors='k', marker='o', s
                                    =50)
plt.show()
# On fait la prediction
prediction = predict(newX, parameters)
print(f"Precision : {accuracy_score(newY.flatten(), prediction
                                    .flatten()) * 100}%.")
```

CHAPITRE 7

BIBLIOGRAPHIE

[1] A logical calculus of the ideas immanent in nervous activity, par Warren McCulloch et Walter Pitts.

[2] Structure d'un Neurone Biologique.

[3] Vidéo de Guillaume Saint-Cirgue, développeur en Machine Learning, sur la séparation des données (jusqu'à 4:22).

[4] Vidéo de Guillaume Saint-Cirgue, développeur en Machine Learning, sur la cross validation.

[5] Documentation de la bibliothèque Python Scikit-Learn, expliquant différentes méthodes de Cross Validation.

[6] Attention in Natural Langage Processing, par les ingénieurs informatiques italiens Andrea Galassi, Marco Lippi et Paolo Torroni.

[7] Efficient Nonlinear Mapping Using The Multi-Layer Perceptron, par le doctorant en ingénierie électrique Arunachalam Gopalakrishnan.