Implémentation d'un réseau de neurones

Amina El Bachari, Israa Ben Sassi, Camille Goujet,
Mehdi Helal et Rafael Quilbier

à l'attention de Jean-Philippe Kotowicz

Table des matières

Résumé

Depuis une dizaine d'années, les réseaux de neurones sont au cœur de l'intelligence artificielle et sont désormais un enjeu majeur dans de nombreux secteurs tels que la santé, l'énergie, les industries.... Avec l'amélioration de la puissance de calcul des machines et des résultats spectaculaires obtenus avec le deep learning,

Pour toutes ces raisons, vous trouverez dans le présent rapport, une description détaillée d'un réseau de neurones, l'explication de chacun des modules qui le composent et finalement, l'implémentation de notre propre réseau de neurones que nous . PAS FINI

Chapitre 1

Introduction

1.1 Bibliographie

1.1.1 Définitions importantes

1.1.1.1 Neurone dit formel ou artificiel

Le neurone formel est l'unité élémentaire des réseaux de neurones artificiels dans lesquels il est associé à ses semblables pour calculer des fonctions arbitrairement complexes, utilisées pour diverses applications en intelligence artificielle. Mathématiquement, le neurone formel est une fonction à plusieurs variables et à valeurs réelles.

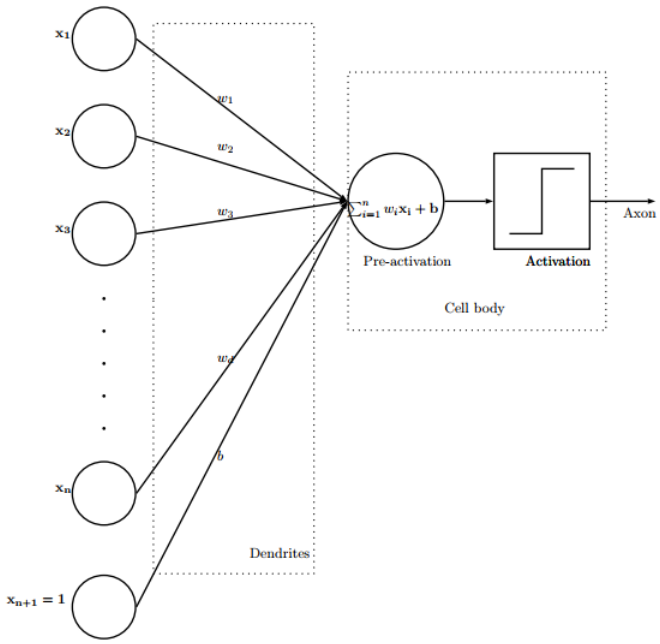


FIGURE 1.1 – Schématisation algorithmique d'un neurone artificiel

Cette représentation est appelée le perceptron, un algorithme d'apprentissage supervisé pour les

classifications binaires linéaires. Cela fait beaucoup de mots inconnus alors arrêtons nous un instant sur chacun de ces termes car ils seront importants pour la suite !

Algorithme : le perceptron est une suite d'opérations et de calcul = la somme des entrées, leur pondération, la vérification d'une condition et la production d'un résultat d'activation.

Apprentissage : l'algorithme doit être "entraîné", c'est à dire qu'en fonction d'une prédiction voulue, le poids des différentes entrées va évoluer et il faudra trouver une valeur optimale pour chacune.

Supervisé : l'algorithme trouve les valeurs optimales de ses poids à partir d'une base de données d'exemples dont on connaît déjà la prédiction. Par exemple on a une base de données de photos de banane et on "règle" notre algorithme jusqu'à ce que chaque photo (ou presque) soit classé comme banane.

Classification : l'algorithme permet de prédire une caractéristique en sortie et cette caractéristique sert à classer les différentes entrées entre elles. Par exemple, trouver toutes les bananes dans un panel de photos de fruits.

1.1.1.2 Réseau de neurones

Un réseau de neurones est en général composé d'une succession de couches dont chacune prend ses entrées sur les sorties de la précédente. Chaque couche (i) est composée de Ni neurones, prenant leurs entrées sur les Ni-1 neurones de la couche précédente.

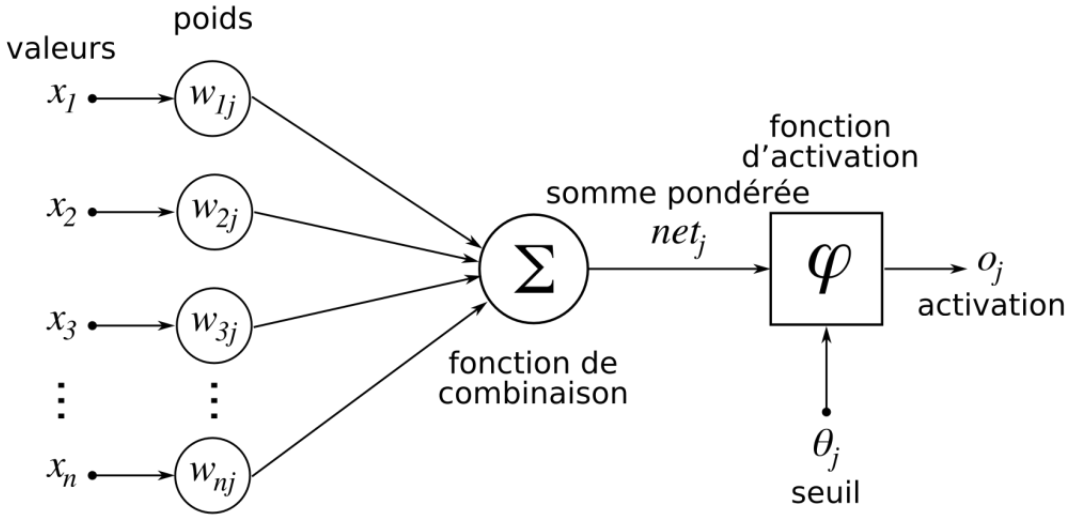


FIGURE 1.2 – Illustration d'un réseau de neurones simple

1.1.1.3 Couche

En effet, un réseau de neurones est composé d'une couche d'entrées ("inputs layer"), d'une couche de sortie ("outputs layers") et d'au moins une couche cachée ("hidden layer") qui fait le lien entre entrée et sortie. Toutes ces couches sont composés de plusieurs neurones qui sont eux-mêmes reliés les uns aux autres par des poids.

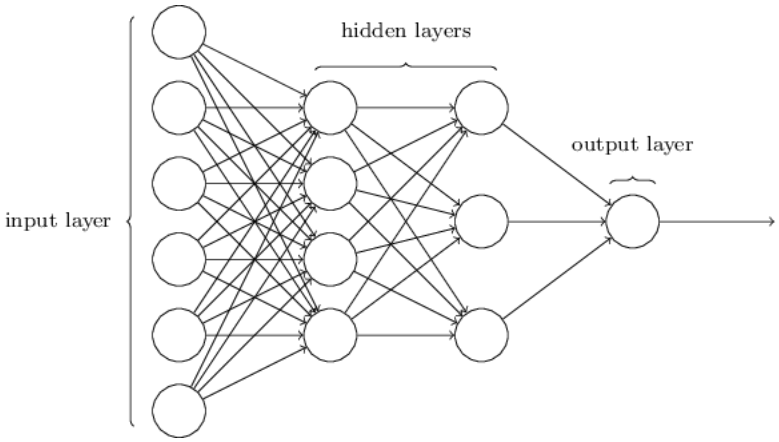


FIGURE 1.3 – Illustration des différentes couches

1.1.1.4 Fonction d'activation

La fonction d'activation (ou fonction de seuillage, ou encore fonction de transfert) sert à introduire une non-linéarité dans le fonctionnement du neurone. Les fonctions de seuillage présentent généralement trois intervalles : en dessous du seuil, le neurone est non-actif (souvent dans ce cas, sa sortie vaut 0 ou -1) ; aux alentours du seuil, une phase de transition ; au-dessus du seuil, le neurone est actif (souvent dans ce cas, sa sortie vaut 1). Des exemples classiques de fonctions d'activation sont :

- La fonction sigmoïde.
- La fonction tangente hyperbolique.
- La fonction de Heaviside.

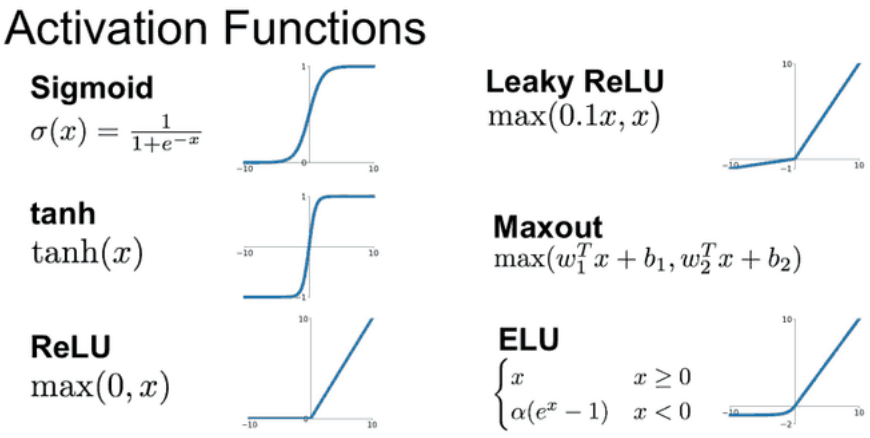


FIGURE 1.4 – Différentes fonctions d'activation

1.1.2 Fonctionnement d'un réseau de neurones

Pour comprendre son fonctionnement, étudions un modèle mathématique simple d'un neurone biologique : le modèle de McCulloch et Pitts (1943).

Considérons n entrées $x_1, \ldots, x_n \in \mathbb{R}$. Un neurone fonctionne en 2 phases. Tout d'abord, il effectue la somme pondérée des entrées :

$$I = \sum_{i=1}^{n} \omega_i x_i$$

avec $\omega_i \in \mathbb{R}$ le poids de la $i^{ème}$ entrée.
Ensuite, la fonction d'activation f vérifie si la valeur calculée est supérieure au seuil requis et détermine si le neurone est actif ou non. Pour ce faire, on compare I à un seuil T : si $I \geq T$ alors le neurone est actif et transmet le signal, sinon il est inactif. Dans le modèle de McCulloch et Pitts, on a :

$$f(I) = \begin{cases} 1 & si\, I \geq T \\ -1 & sinon \end{cases}$$

Notons que lorsque les neurones sont reliés uniquement par des connexions directes (i.e vers un neurone de la couche suivante), l'activation des différentes couches est réalisée de manière synchrone : la sortie de tous les neurones de la $1^{ère}$ couche est calculée, puis celle de la 2^e... Dans le cas d'un réseau possédant des connexions latérales (de haut en bas vers un neurone d'une même couche ou bien vers un neurone d'une couche précédente), l'ordre de mise à jour des sorties des différents neurones est important, car chaque nouvelle sortie va pouvoir modifier le calcul de la sortie d'un autre neurone. On peut mettre à jour des sorties de manière asynchrone et aléatoire (i.e les neurones sont choisis aléatoirement) ou bien de manière synchrone (toutes les sorties sont mises à jour en même temps).

Au fur et à mesure des itérations, les poids sont modifiés par apprentissage. Il les calcule en fonction des couples entrée/sortie désirées (on parle d'apprentissage supervisé) ou indépendamment d'une sortie désirée (on parle d'auto-organisation) par des petites adaptations successives.
Soit ω_{ij} le poids de la connexion entre les neurones i et j à un instant donné discret t. Après une itération d'apprentissage, la nouvelle valeur est $\omega_{ij}(t+1) = \omega_{ij}(t) + \Delta\omega_{ij}$.
Il existe plusieurs règles d'apprentissage courantes comme la règle de Hebb, la règle d'apprentissage compétitif, etc que nous détaillerons dans notre projet lorsque nous les utiliserons.

1.1.3 Architecture des réseaux de neurones

On appelle architecture d'un réseau de neurones sa forme. On distingue 4 types de réseaux de neurones :
- les réseaux de neurones Feed-forwarded
- les réseaux de neurones récurrents (RNN)
- les réseaux de neurones à résonance
- les réseaux de neurones auto-organisés.

Le choix de telle ou telle architecture est une question essentielle lors de la construction d'un réseau. Chacune possède ses forces et ses faiblesses. Ainsi, en fonction de l'application que l'on souhaite faire de notre algorithme, on se portera sur une architecture en particulier.
On peut aussi souligner que souvent, afin d'obtenir un meilleur résultat, plusieurs types d'architectures sont combinées. Nous avons préalablement cité 4 grandes familles d'architectures, mais il existe des dizaines de réseaux particuliers pour chacune d'elles. Voyons plus en détails ces 4 architectures :

1.1.3.1 Réseaux de neurones feed forwarded

L'appellation Feed-Forward signifie que l'information traverse le réseau de neurones de l'entrée à la sortie sans retour en arrière. En français, il est nommé : « réseau de neurones à propagation avant ». Dans ce type de réseaux, on distingue les réseaux monocouches et les réseaux multicouches.

Dans les réseaux monocouches, le perceptron est dit « simple », car il est constitué que d'une couche d'entrée et une couche de sortie.
À l'inverse, dans les réseaux multicouches, les perceptrons sont qualifiés de multicouches, car ils disposent de plusieurs couches cachées entre l'entrée et la sortie.

Ces 2 types de réseaux de neurones ont des intérêts différents : le réseau multicouche est plus adapté pour traiter des informations non-linéaires.

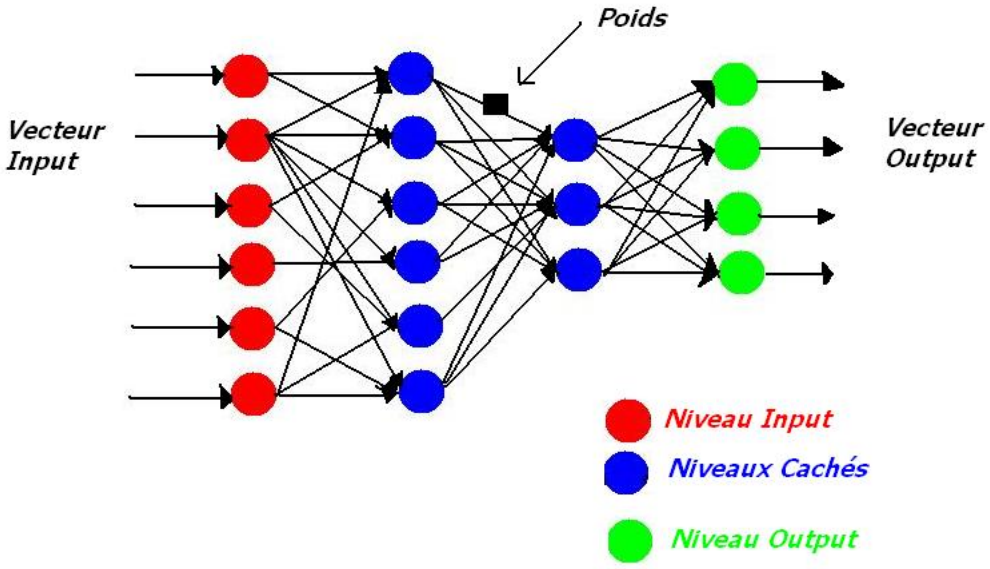


FIGURE 1.5 – Réseau Feed-Forwarded

1.1.3.2 Réseaux de neurones récurrents

Plus complexe et moins intuitif, les réseaux de neurones récurrents traitent les informations de manière circulaire. Un réseau est qualifié de récurrent si sa structure possède au moins un cycle. Contrairement l'architecture de type Feed Forwarded, dans les réseaux récurrents, l'information circule dans les deux sens. Ils peuvent posséder une couche ou plusieurs. L'intérêt est de conserver de l'information en mémoire et de la laisser accessible à tout instant ultérieur. C'est pourquoi les réseaux de neurones récurrents sont particulièrement bien adaptés aux applications faisant intervenir un contexte, comme la reconnaissance de forme.

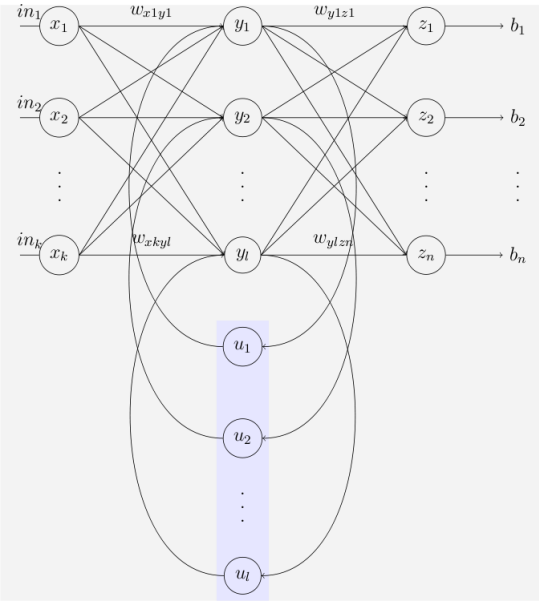


FIGURE 1.6 – Réseau récurrent de type Elman

1.1.3.3 Réseaux de neurones à résonance

L'appellation de ce type de réseaux fait référence à son fonctionnement. Lorsqu'un neurone est activé, son activation est renvoyée à tous les autres neurones du réseau. Cela provoque des oscillations, d'où l'emploi du terme « résonance ».
Pour mieux comprendre l'objectif de cette architecture, étudions le modèle ART (Adaptative Resonance Theory) conçu par Gail Carpenter et Stephen Grossberg. Il existe de nombreux modèles ART, (ART1, ART2, fuzzy ART etc.) qui utilisent l'apprentissage supervisé, ou non supervisé. L'objectif général des modèles ART est de résoudre le dilemme entre stabilité et plasticité.
Selon un article du laboratoire d'Analyse Cognitive de l'information à Montréal : «la plasticité rend compte de la capacité du réseau à s'adapter aux informations nouvelles, et la stabilisation mesure la capacité du réseau à organiser les informations connues en ensembles stables.».
Finalement, le principe des modèles ART est d'apprendre de manière autonome, à s'adapter, et à se stabiliser en même temps. Ainsi, ces modèles sont capables de choisir entre une information pertinente à prendre en compte, et une information superflue, qui pourrait donner lieu à un surapprentissage.

1.1.3.4 Réseaux de neurones auto-organisés

Ce type de réseau de neurones est surtout utilisé dans le traitement d'informations spéciales. En effet grâce à des méthodes d'apprentissage non supervisé, ce type de réseau est capable de répartir en différentes classes de grands espaces de données.

1.2 Objectifs et fonctionnalités

Tout au long de ce projet, nos objectifs seront les suivants :

1.2.1 Création d'un réseau de neurones

1.2.2 Application d'un réseau de neurones à un exemple de la liste suivante :

1.2.2.1 Les classifications de données, texte ou image

L'objectif est d'élaborer un système capable d'affecter à une donnée, une image ou à un texte non-structuré, un tag qui correspond à une classe bien précise.
On cite deux exemples d'utilisation très importants.
D'une part, la classification de documents imprimés est une tâche cruciale dans de nombreuses chaînes de traitement ; par exemple, l'automatisation de tâches bureautiques afin de classifier les documents imprimés selon des catégories telles que : lettres, publicités, plans et cartes, articles de presse, etc...
Il faut tout d'abord extraire les données textuelles ou les images utilisées pour ensuite les classifier.

D'autre part, la reconnaissance des émotions, très utilisée sur les réseaux sociaux. Qu'il s'agisse de la détection de comportement violent à travers un texte ou une image (DeepBreath) ou de l'analyse de l'impact émotionnel sur des compagnes de marketing, etc...

FIGURE 1.7 – Illustration avec les visages

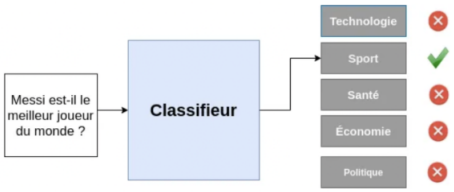


FIGURE 1.8 – Illustration avec le classifieur

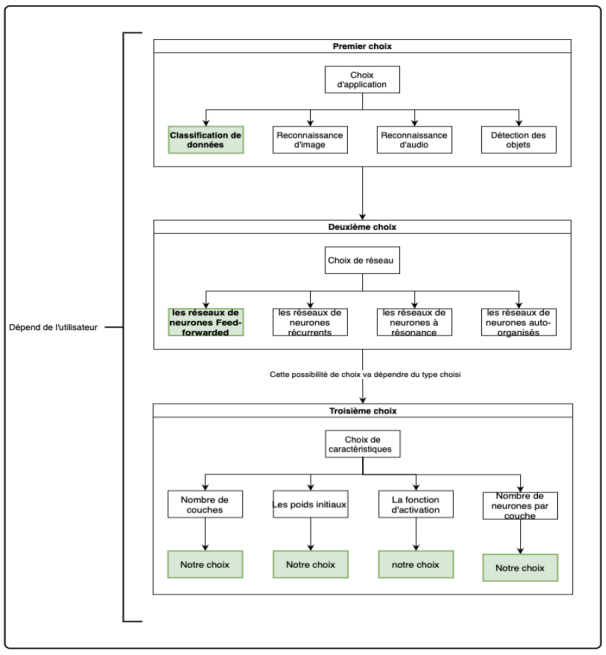
1.2.2.2 Prédiction de données

Enfin les réseaux de neurones sont très utilisés pour mettre au point des modèles de prédictions à partir d'échantillons de données. La prédiction d'indicateurs financiers est un exemple très parlant. En effet « en 2011, 40 % des ordres donnés sur le CAC40 sont totalement automatisés à l'aide d'algorithmes informatiques ». Aujourd'hui, cet outil fournit d'excellents résultats dans le domaine de détection des entreprises en future difficulté.
Les réseaux de neurones peuvent aussi prédire des phénomènes qui nous semblent plus aléatoires et

non linéaires tout en prenant en compte d'éventuelles imprécisions des données fournies en entrée. Une équipe de scientifiques s'est intéressée à la prévision des crues du bassin-versant de l'Eure à la station de Louviers en Normandie grâce à des réseaux de neurones. Le modèle pluie-débit en résultant a permis de prendre en compte l'imprécision des données fournies en entrée et ainsi d'établir des prévisions fiables en quelques secondes sur les grandes crues à venir dans les 48 heures. Cet exemple illustre parfaitement l'importance que peuvent avoir les réseaux de neurones dans la prédiction de phénomène non-linéaires.

1.3 Manuel d'utilisation préliminaire

- Choisir une application :
  - Classification
  - Prédiction
- Choisir un type de réseau :
  - Réseau Feed Forwarded
  - Réseau Récurrent
- Saisir les caractéristiques du réseau :
  - Importation des données à partir d'un fichier.
  - Saisie manuelle
    - Saisir le nombre de couches cachées
    - Saisir le nombre de neurones par couches
    - Choix des poids initiaux
      - Remplissage de manière aléatoire
      - Remplissage à la main (une valeur pour tous les poids)
    - Choix des biais (nbNeuroneparCoucheCachee*nbCoucheCachee+nbNeuroneCoucheSortie)
      - Remplissage de manière aléatoire
      - Remplissage à la main (une valeur pour tous les poids)
- Donner le nom de fichier de données



En pratique, voici ce que nous obtenons :

```
Voulez-vous entrer votre reseau :
         1) Manuellement
        2) Avec un fichier
1
Saisir l'entier correspondant au type de reseau souhaite
1: reseau forwarded
2: reseau recurent
0:quitter
1
Saisir l'entier correspondant au cas d'utilisation souhaite
1: classification
2: prediction
2
Saisir l'entier correspondant au nombre de couches cachees du reseau
1
Saisir le nombre d'entrees
4
Saisir un a un le nombre de neurones dans chaques couches cachees
2
Comment voulez vous intisaliser votre matrice de poid ?
0: aleatoirement
x un double: tous à x
0
```

FIGURE 1.9 – Manuel d'utilisation manuelle

```
Voulez-vous entrer votre reseau :
         1) Manuellement
        2) Avec un fichier
2
Entrez le nom du fichier ou 0 pour sortir
TBF.csv
le reseau est conforme
Que voulez vous faire ?
        1) Apprentissage du reseau
        2) Tester le reseau sur un exemple
        3)Quitter
1
Entrez le nom du fichier pour l'apprentissage
FA.txt
Que voulez vous faire ?
        1) Apprentissage du reseau
        2) Tester le reseau sur un exemple
        3)Quitter
2
Entrez le nom du fichier contenant l'entree
TE.txt
La classe est : 2
```

FIGURE 1.10 – Manuelle d'utilisation Fichier

Chapitre 2

Spécification

Maintenant que nous nous sommes familiariser avec toutes les notions que peuvent avoir leur importance dans la compréhension d'un réseau de neurones, nous allons à présent étudier les côtés plus techniques de ces objets afin de mener à bien l'implémentation de notre propre réseau.

2.1 Conception préliminaire

2.1.1 Diagramme de cas d'utilisation

Voici notre diagramme de cas d'utilisation final :

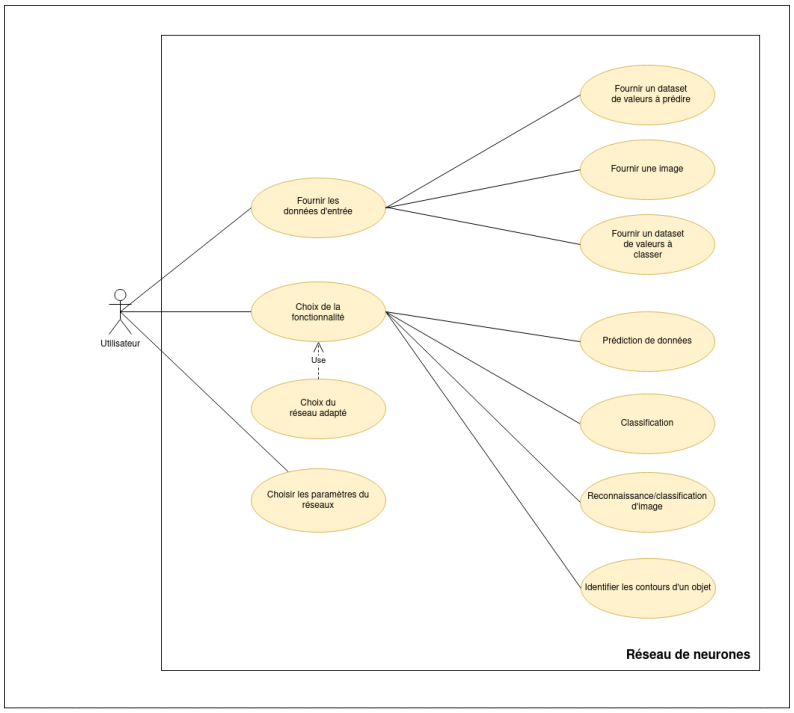


FIGURE 2.1 – Diagramme de cas d'utilisation

2.1.2 Descriptions des cas d'utilisation

— Fournir les données d'entrées : pour faire ceci, on construit un tableau de la dimension adéquate correspondant à celle des données que l'on donne au réseau. Si nos données sont sous forme de fichier ou autre, il faut les convertir dans le langage utilisé avant de pouvoir les utiliser. Ce travail sera fait dans la classe Entrées.
— Choix des fonctionnalités, des paramètres et du type de réseau : notre programme propose les différentes options à l'utilisateur de manière intéractive pour lancer la partie du programme correspondant au choix de l'utilisateur.

2.1.3 Diagrammes de séquence

Afin de visualiser les interactions entre les composants, nous avons réalisé plusieurs diagrammes de séquence pour chaque phase.
Nous commençons avec un diagramme de séquence assez général :

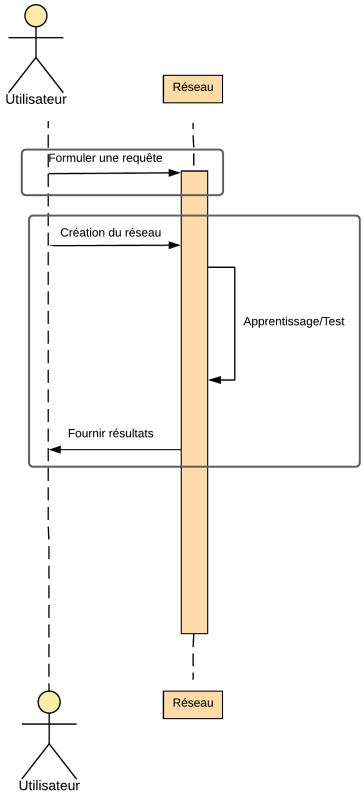


FIGURE 2.2 – Diagramme de séquence

Ensuite, nous avons détaillé les différentes phases du premier diagramme. Notamment, pour la création d'un réseau :

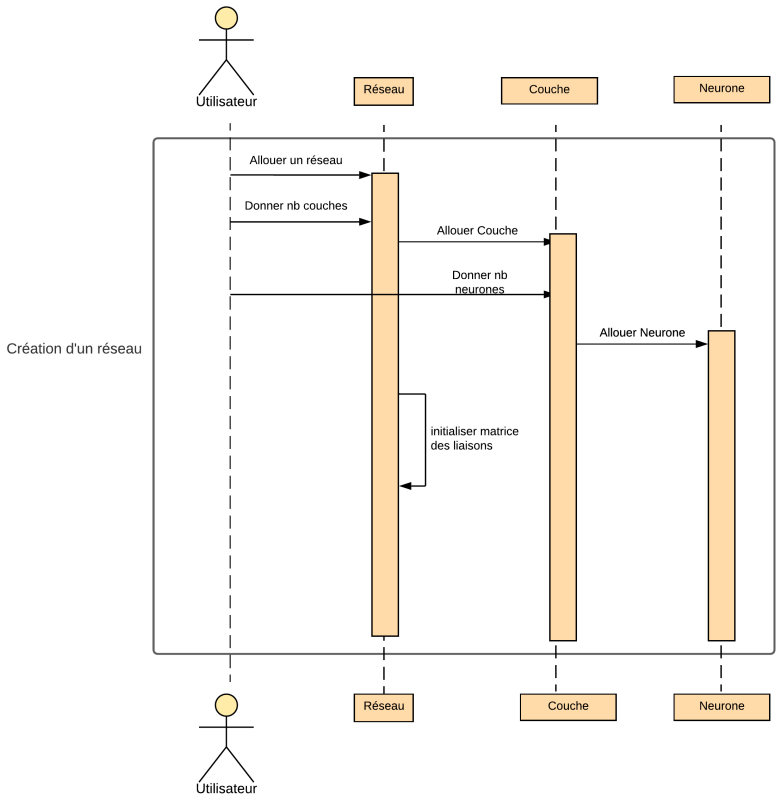


FIGURE 2.3 – Diagramme de séquence

ainsi que pour l'initialisation d'un réseau prédéfini dans un fichier

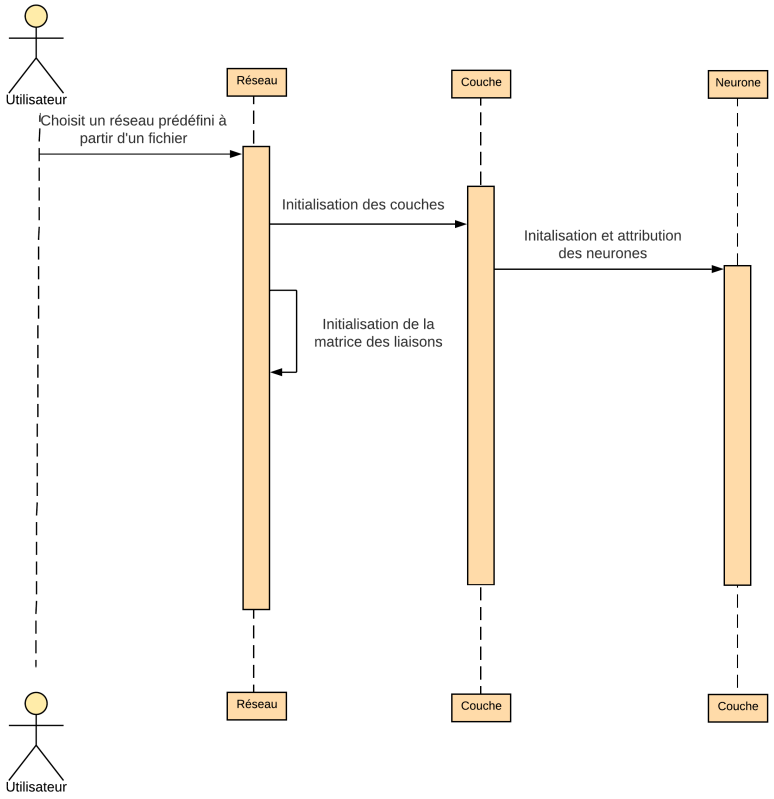


FIGURE 2.4 – Diagramme de séquence

. Finalement, nous avons réalisé un diagramme de séquence lorsqu'un utilisateur formule sa requête :

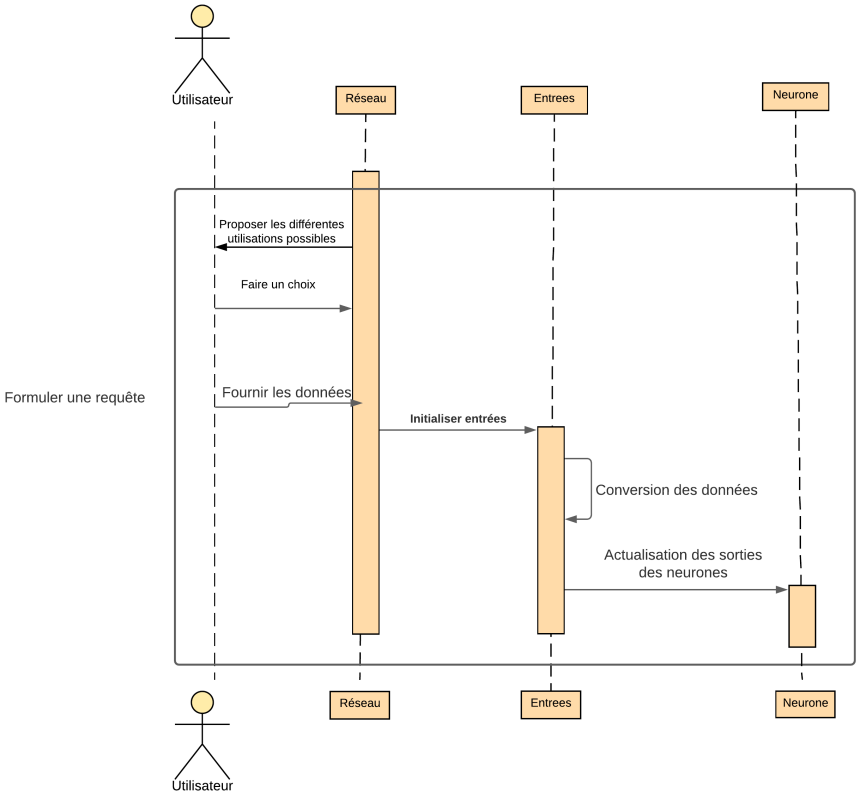


FIGURE 2.5 – Diagramme de séquence

2.1.4 Le choix des données

Pour le moment, nous allons se concentrer sur une classification supervisée des données. Voyons cela sur l'exemple des iris de Fisher.
Les Iris de Fisher correspondent à 150 fleurs décrites par 4 variables quantitatives :

- longueur du sépale
- largeur du sépale
- longueur du pétale
- largeur du pétale

Les 150 fleurs sont réparties en 3 différentes espèces :

- iris setosa
- iris versicolor
- iris virginica.

Le fichier va être sous la forme suivante :

longueur des ...en cm) (Sepal length)	largeur des sépales (en cm) (Sepal width)	longueur des pétales (en cm) (Petal length)	largeur des pétales (en cm) (Petal width)	Espèce (Species)
6.3	3.3	6.0	2.5	*I. virginica*
5.8	2.7	5.1	1.9	*I. virginica*
7.1	3.0	5.9	2.1	*I. virginica*
5.1	3.5	1.4	0.2	*I. setosa*
4.9	3.0	1.4	0.2	*I. setosa*
4.7	3.2	1.3	0.2	*I. setosa*
4.6	3.1	1.5	0.2	*I. setosa*

FIGURE 2.6 – exemple du fichier

On peut donc voir nos données (entrée du programme) comme :

— Pour une fleur (entrée du réseau) : un vecteur de dimension 4 représentant les 4 variables explicatives d'une fleur

$$f = (x_1, x_2, x_3, x_4)^T$$

— Pour les classe (la sortie du réseau) : un vecteur de dimension 3 représentant les probabilités des 3 potentielles classes d'appartenance (setosa, versicolor, virginica)

$$c = (y_1, y_2, y_3)^T$$

Il faut donc lire les mesures, les mettre en entrée du réseau, calculer le résultat en propageant l'information de l'entrée vers la sortie couche par couche, prendre la sortie maximale parmi les neurones de sortie et afficher le résultat.
Donc notre réseau, cela ressemblera à quelque chose comme ça :

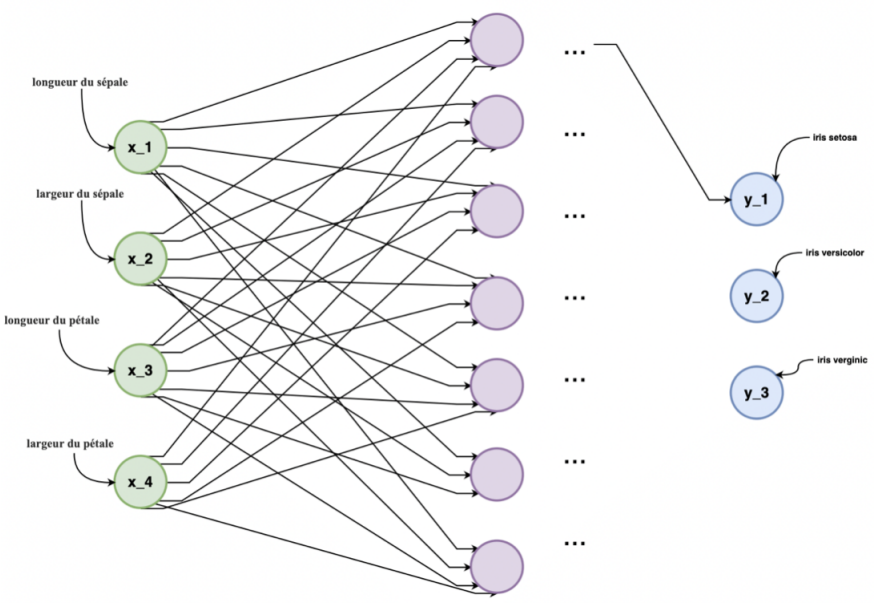


FIGURE 2.7 – Réseau de neurones appliqué aux iris de Fisher

Se pose le problème suivant : comment à partir des mesures réalisées sur une fleur inconnue trouver le type d'iris à l'aide d'un réseau de neurones ?

2.1.5 Tests d'intégration

Ici, on explicite tous les différents scénarios possibles engageant un réseau de neurones. Cela nous permettra de n'oublier aucune fonctionnalité dans notre code et dans nos diagrammes. Ces scénarios seront ensuite repris à la fin pour effectuer différents tests, et ainsi vérifier le bon fonctionnement de notre programme.

2.1.5.1 Scénarios de création d'un réseau :

Réseau vide : Si dans le choix des paramètres, on choisit 0 neurone et donc 0 couche, les utilisations que l'on pourra faire de ce réseau seront très limitées pour ne pas dire inexistantes. Pas de couche d'entrée, ni d'arrivée. Il serait utile de prévoir une exception pour éviter ce cas de figure/de potentielles erreurs.

Perceptron simple : On entre en machine : 2 pour le nombre de couches, autant de neurones que l'on veut pour la couche d'entrée et un seul neurone sur la couche de sortie. Le perceptron simple est un modèle de prédiction (supervisé) linéaire. Ce schéma simple a un effet plus éducatif qu'autre chose, il sert de modèle pour le perceptron multicouches qui permet de régler des problèmes plus complexes.

Perceptron multicouche : L'utilisateur entre en machine un nombre de couches supérieures à 2. Notons tout de même que choisir au dela de 6 à 10 couches entraine très souvent des problèmes d'overfitting, aussi appelée surapprentissage (le réseau est incapable se généraliser, car il est trop adapté aux données d'appretissage). Il serait peut être bon de définir un nombre de couches par défaut pour

orienter les utilisateurs novices. Puis l'utilisateur saisit de manière itérative le nombre de neurones pour chaque couche y compris la couche de sortie qui peut comporter plusieurs neurones. Ensuite les neurones des couches i seront reliés à la couche i+1 suivant le type de réseau sélectionné par l'utilisateur.

2.1.5.2 Scénarios relatifs aux fonctions d'activation :

Ici, il est très difficile de délimiter notre projet puisqu'il n'existe pas de règles propres au choix de la fonction d'activation utilisée pour telle couche, tel réseau ou telle fonctionnalité de ce réseau.
On peut tout de même émettre quelques pistes :

— Dans le cas d'un problème de *régression*, il n'est pas nécessaire de transformer la somme pondérée reçue en entrée. La fonction d'activation est la fonction identité, elle retourne ce qu'elle a reçu en entier. (pour un perceptron simple)

— Dans le cas d'un problème de *classification binaire*, on peut utiliser une fonction de seuil :

$$s\left(w_0+\sum_{j=1}^{p} w_j x_j\right) = \begin{cases} 0 & si \left(w_0+\sum\limits_{j=1}^{p} w_j x_j\right) < 0 \\ 1 & sinon \end{cases}$$

Comme dans le cas de la *régression logistique*, on peut aussi utiliser une fonction **sigmoïde** pour prédire la *probabilité* d'appartenir à la classe positive.

— Dans le cas d'un problème de *classification multi-classe,* nous allons modifier l'architecture du perceptron. Au lieu d'utiliser une seule unité de sortie, il va en utiliser autant que de classes. Chacune de ces unités sera connectée à toutes les unités d'entrée. On aura donc ainsi $K(p+1)$ poids de connexion, où K est le nombre de classes. On peut alors utiliser comme fonction d'activation la fonction **softmax**. Il s'agit d'une généralisation de la sigmoïde.
Si la sortie pour la classe k est suffisamment plus grande que celles des autres classes, son activation sera proche de1 tandis que l'activation des autres sera proche de 0. On peut donc aussi considérer qu'il s'agit d'une version différentiable du maximum, ce qui nous aidera grandement pour l'apprentissage.

2.1.5.3 Scénarios relatifs aux poids :

Pour entraîner un perceptron, c'est-à-dire apprendre les poids de connexion, nous allons chercher à minimiser l'erreur de prédiction sur le jeu d'entraînement. On peut initialiser les poids de différentes manières, soit on initialise les poids manuellement (souvent on prend la valeur 0,5 (valeur moyenne entre 0 et 1)) soit on initialise les poids en créant une fonction qui nous renverra un nombre aléatoire entre 0 et 1 pour chaque poids.
Puis pour les faire évoluer à chaque itération jusqu'à convergence de l'algorithme, on implémente la méthode de descente du gradient. Grâce à celle-ci, le poids à l'étape (i) devient à l'étape $(i+1)$:

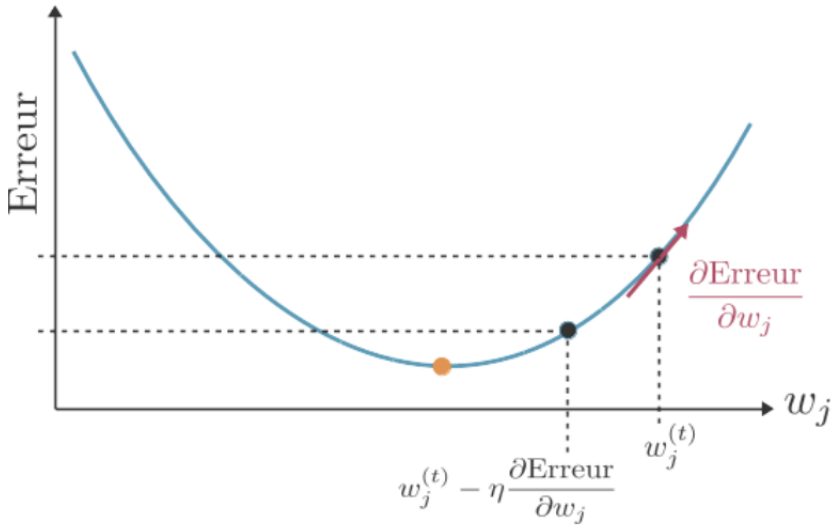


FIGURE 2.8 – Courbe de l'erreur en fonction du poidsoù η est la vitesse d'apprentissage.

2.1.5.4 Scénarios relatifs aux utilisations :

On compte 4 cas d'utilisations pour un réseau de neurones. Nous nous sommes limités à proposer la prédiction de données et la classification. Lorsque l'utilisateur saisit le type d'utilisation, un algorithme type va se lancer. Voyons chaque cas un par un en détail :

Classification Il est recommandé à l'utilisateur d'utiliser un perceptron multi-couches. Puis deux modélisations lui sont proposées.

- Soit l'utilisateur saisit le nombre K de classes qui correspond au nombre de sorties du réseau de neurones. Chaque neurone de sortie indique la probabilité d'appartenance à la $k^{ème}$ classe. La classe avec la probabilité la plus forte est la classe d'appartenance de la donnée fournie en entrée.
- Soit l'utilisateur choisit de modéliser un réseau pour chaque classe $n = 1..K$. Dans ce cas, il y a donc un unique neurone de sortie pour chacun des réseaux indiquant la probabilité d'appartenance des données à la nième classe. Dans cette modélisation, une donnée passe par K réseau de neurones. Puis la construction du réseau débute, avec comme nombre de neurones dans la couche d'entrée le cardinal des variables explicatives des données sélectionnées et le nombre de neurones de la couche de sortie égale à 1 (pour K réseaux).

Ensuite, l'utilisateur peut choisir un apprentissage supervisé ou non du réseau. S'il choisit un apprentissage supervisé il devra importer les données nécessaire (toujours au bon format et organisé). Enfin, la dernière étape consiste à importer les données organisées au bon format que l'on souhaite classifier, puis à lancer l'algorithme.

Prédiction Dans le cas d'une prédiction, nous considérons une unique modélisation. La couche d'entrée comporte n neurones, égale à la cardinalité du nombre de variables explicatives, et la couche de sortie possède un unique neurone qui retourne la valeur prédite. L'utilisateur va donc saisir le nombre de variables explicatives que comporte son jeu de données. Puis la construction du réseau débute. Un fois construit, l'utilisateur spécifie s'il souhaite faire apprendre son réseau. Si oui, il rentre le fichier

de données d'apprentissage. Une fois l'apprentissage effectué, il faut importer les valeurs des variables explicatives de la donnée que l'utilisateur souhaite prédire, puis lancer l'algorithme.

2.2 Conception détaillée

Dans cette partie, nous expliquons en détail chaque module/classe composant notre code et les tests unitaires qui lui sont associés.
Cela permet d'avoir une vision précis, exhaustive et claire de l'implémentation produite.

2.2.1 Diagramme de classe

Voici notre diagrammde de classe :

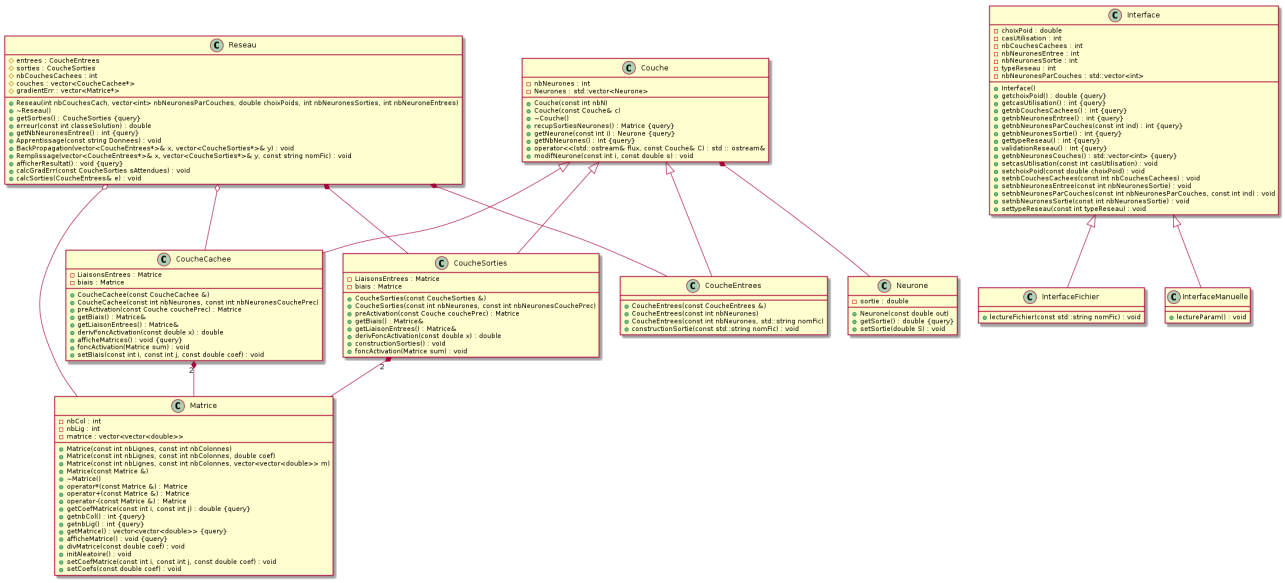


FIGURE 2.9 – Diagramme de classe

2.2.2 Réseau

2.2.2.1 La classe Réseau

Dans cette classe, on retrouve les méthodes qui permettent de manipuler un réseau de neurones, de sa création à sa destruction. Une première remarque importante sur ces codes : nous avons souvent été obligés de "dupliquer" certains codes car certaines couches cachées sont liées à la couche d'entrée ou bien à la couche de sortie, ce qui veut dire que les codes de la forme :
`couchecachee[i] = f(couchecachee[i-1], couchecachee[i+1])` ne fonctionne pas dans ce cas particulier. Par exemple, dans le constructeur de Réseau, nous avons traité le cas de la première couche cachée `couches[0]` à part. La fonction `erreur(int classeSolution)` calcule la norme au carré de la différence entre la sortie attendue (qui est donc fournie par l'utilisateur) et la sortie calculée par le réseau. Ensuite, la méthode `Apprentissage(string Donnees)` est une méthode qui modifie les poids et les biais du réseau à l'aide du fichier Donnees qui est un échantillon d'apprentissage. Elle

construit l'échantillon d'apprentissage complet composé de `lesEntrees` et de `lesSorties` en faisant appel à la méthode `Remplissage(lesEntrees, lesSorties, Donnees)`. Ensuite, elle divise cet échantillon en sous-échantillons appelés `EntreesSousEchan` et `SortiesSousEchan` puis elle appelle la méthode `BackPropagation(vector<CoucheEntrees*>& x, vector<CoucheSorties*>& y)` sur chaque sous-échantillon. Cette dernière méthode va calculer le gradient de l'erreur moyen sur le sous-échantillon qui lui est donné en paramètres et modifier les poids et les biais en conséquences. C'est la stratégie de modification par lots ou batch. Pour être plus précis, la méthode `Backpropagation()` appelle la méthode `calcGradErr(CoucheSorties sAttendues)` qui permet de calculer le gradient de l'erreur sur un seul exemple sur chaque élément du sous-échantillon d'apprentissage, puis elle fait la moyenne des gradients obtenus et elle modifie les poids et les biais. Cette méthode
`calcGradErr(CoucheSorties sAttendues)` est la plus compliquée. Pour faire simple, elle utilise le principe de la rétropropagation du gradient : elle calcule en premier les derniers termes du gradient de l'erreur, c'est-à-dire les dérivées partielles de la fonction d'erreur par rapport aux poids et aux biais de la dernière couche du réseau (donc la couche de sortie). Une fois ces termes calculés, on en déduit la valeur des dérivées partielles par rapport aux poids et aux biais de la couche précédente, et ainsi de suite jusqu'à avoir calculé les dérivées partielles par rapport à tous les poids et tous les biais. À chaque étape, on stocke les dérivées partielles par rapport aux poids de la i-ème couche dans la variable `gradientErr[2*i]` et les dérivées partielles par rapport aux biais de la i-ème couche dans `gradientErr[2*i+1]`. Enfin, chaque dérivée partielle est calculée en utilisant la formule suivante obtenue à l'aide de la chain rule :

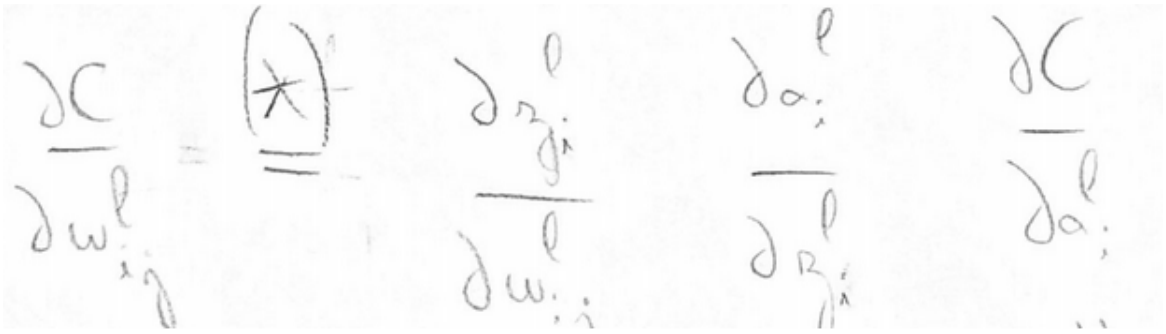


FIGURE 2.10 – Chain rule

avec C qui désigne la fonction d'erreur, ${w_{ij}}^l$ qui est le poids reliant le i-ème neurone de la couche l au j-ème neurone de la couche $l-1$, z_i^l est le résultat de la pré-activation du i-ème neurone de la couche l et a_i^l est la fonction d'activation du i-ème neurone de la couche l. On cacule de la même manière les dérivées partielles par rapport aux biais en remplaçant les poids ${w_{ij}}^l$ par b_i^l qui désigne le biais du i-ème neurone de la couche l.
Voir documentation doxygen.

2.2.3 Couche

La fonction Couche a pour unique but de construire les objets Couche peu importe leur type (entrée, cachée, sortie).
Pour notre implémentation, nous avons choisi de coder les méthodes utiles dans les classes CoucheCachée et CoucheSorties uniquement puisqu'elles sont inutiles dans CoucheEntrées.
voir la documentation doxygen

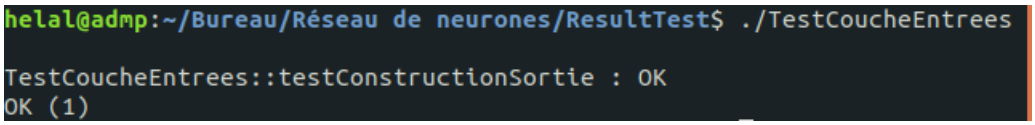


FIGURE 2.11 – Resultat testConstructionSortie

2.2.4 CoucheEntree

2.2.4.1 La classe CoucheEntree

voir la documentation doxygen

2.2.4.2 Les tests de la classe CoucheEntree

testConstructionSortie : prototype de la méthode : void constructionSortie()

Soit un fichier de données test qui contient les 4 valeurs utilisées dans l'exemple précédent.
il faut vérifier que la fonction constructionSorite remplit un tableau de neurone de dimension 4 comme suit :

x_1	x_2	x_3	x_4
$6,3$	$3,3$	$6,0$	$2,5$

Donc il faut créer un objet coucheEntrée qu'on initialise à l'aide d'un constructeur avec les valeurs du tableau et vérifier que le tableau de neurone de cet objet est égal au tableau de neurone de notre fonction test.

Resultat :

2.2.5 CoucheSortie et CoucheCachee

2.2.5.1 La classe CoucheSortie

voir la documentation doxygen.

2.2.5.2 La classe CouchCachee

voir la documentation doxygen.
Comme vous allez le constater, nous avons fait le choix d'implémenter les fonctions preActivation() et activation() dans la classe CoucheCachée mais aussi dans CoucheSortie. Cela nous permet en fait de pouvoir modifier la fonction activation qui, bien souvent n'est pas la même selon si c'est une couche cachée ou de sortie.

2.2.5.3 Les tests de la classe CoucheCachee et CoucheSortie

testPreactivation : prototype de la méthode : double[] preActivation()

Il faut tester si elle renvoie bien la somme des produits de la valeur du neurone avec le poids qui le relie au neurone étudié ajouté au biais de ce neurone.
Pour l'exemple des iris de Fisher :

- une couche d'entrée qui comprend 4 entrées x1, x2, x3 et x4
- une couche cachée qui comprend 2 neurones n1 et n2

⇒ Il y a alors 8 poids et 2 biais

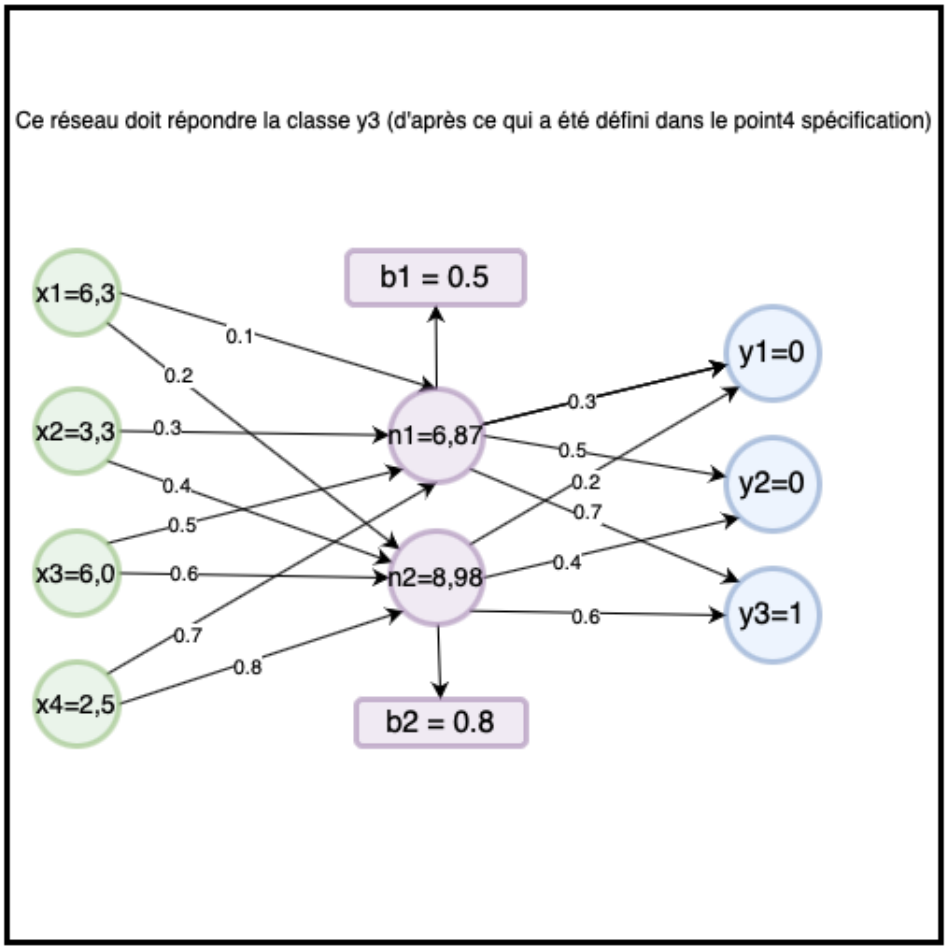


FIGURE 2.12 – Exemple des iris de Fisher

La fonction de préactivation doit donnée comme valeur pour n1 :
$x1 \times 0,1 + x2 \times 0,3 + x3 \times 0,5 + x4 \times 0,7 + b1 = 0,63 + 0,99 + 3 + 1,75 + 0,5 = \mathbf{6,87}$
Pour n2 : $\mathbf{8,98}$
⇒ Donc on doit avoir

6,87	8,98

testActivation : prototype de la méthode : double activation(double[])
Nous allons considérer le cas de la fonction sigmoïde donnée par :

$$f(x) = \frac{1}{1+e^x}$$

Il faut tester si elle renvoie la bonne valeur.
Pour cela, reprenons le résultat de la fonction de préactivation, on a alors :

$$f(6,87) = 0.001$$

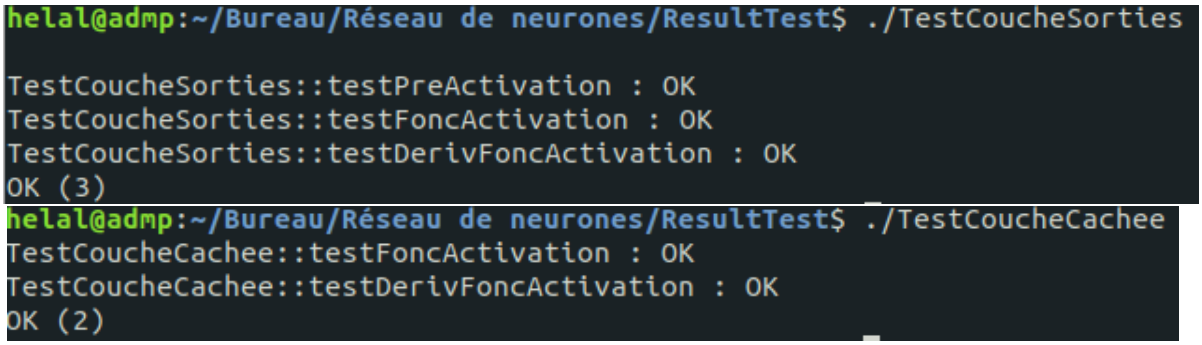


FIGURE 2.13 – Resultat Activation et testPreactivation

Resultat :

2.2.6 Interface :

2.2.6.1 Classe Interface

voir la documentation doxygen.

2.2.6.2 Les tests de la classe Interface

testLectureFichier : prototype de la méthode : listeParamètres lectureFichier(String 'nomFich.csv')
Nous allons d'abord tester la fonction lectureFichier(). Pour cela, nous allons définir le jeu de données que nous allons utiliser ainsi que les résultats attendus.
Tout d'abord, faisons un point sur la manière de remplir un fichier des paramètres du futur réseau à construire :

- *Le premier paramètre* du fichier sera le type de réseau. Dans notre projet, seul le réseau forwarded sera fonctionel. Il sera représenté par l'entier 1.
- *Le second paramètre* du fichier est le cas d'utilisation du réseau (1 :classification, 2 :prédiction, 3 :identification des objets et 4 : reconnaissance d'image). Rappelons que seule le choix 1 et potentiellement 2 seront implémentés.
- *Le troisième paramètre* compte le nombre de couches cachées, nbCouche.
- Le *quatrième paramètre* compte le nombre de neurones par couche cachée (classe CoucheCachée).
- *Le cinquième paramètre* du fichier est 0 ou x un entier positif non nul :
  - 0 si l'initialisation de la matrice de poids initiaux est faite *aléatoirement*.
  - x si l'utilisateur choisit d'initaliser tous les poids à x.

1. **Test avec un fichier vide :** Entrer en paramétres un fichier en format .txt vide.
   La méthode doit renvoyer un message d'erreur sur la console pour prévenir l'utilisateur, puis revenir sur le menu précédent.
   Renvoie 0 comme message d'erreur.
2. **Test avec un fichier au mauvais format :** Entrer en paramètres un fichier en binaire, donc illisible par notre programme.
   La méthode doit renvoyer un message d'erreur sur la console pour prévenir l'utilisateur, puis revenir sur le menu précédent (ainsi l'utilisateur pourra entrer manuellement les paramètres si son fichier n'est pas au bon format).
   Renvoie -2 comme message d'erreur.

3. **Test avec un fichier mal rempli :** Plusieurs tests sont envisageables.
   Voyons en détails :
   Fichier test 1 : (2 1 3 2 5 2 0) renvoie un message pour dire que le fichier est mal rempli (exemple : réseau non implémenté).
   Retourne une liste avec la valeur -1.
4. **Test avec un fichier bien écrit :**
   Fichier test 1 : (1 1 3 2 5 2 0) renvoie une liste de paramètres=(1 1 3 2 5 2 0).

testlectureParam() : prototype de la méthode : listeParamètres lectureParam()

Tout d'abord, faisons un point sur la manière dont l'utilisateur va saisir les paramètres, soit comment va fonctionner la fonction listeParametreslectureParam().
L'utilisateur entrera à la suite les différents paramètres dans l'ordre suivant (il sera guidé par le programme $param1$: type de réseau, $param2$: cas d'utilisation du réseau (rappel : 1 :classification, 2 :prédiction, 3 :identification des objets et 4 : reconnaissance d'image), $param3$: nombre de couches cachées, $param4$: nombre de neurones par couche (une suite d'entiers séparés par des espaces), $param5$: choix de la matrice de poids initiaux.)

1. Test quand l'utilisateur entre de mauvais paramètres (par exemple entre des string au lieu d'entier etc.. problème de type)
   (a) Test 1 : saisie du premier paramètre : si $param1 = a$ erreur de type, si $param1 = 4$ erreur réseau non implémenté, (propose une resaisie, ou quitter).
   (b) Test 2 : saisie du second paramètre : avec $param1 = 1$ si $param2 = a$ erreur de type, si $param2 = 4$ erreur cas d'utilisation non implémenté, (propose une resaisie, ou quitter).
   (c) Test 3 : saisie du troisième paramètre : avec $param1 = 1$, $param2 = 2$, si $param3 = a$ erreur de type.
   (d) Test 4 : saisie du quatrième paramètre : avec $param1 = 1$, $param2 = 2$, $param3 = 4$, si $param4 = 4\ 5\ 6\ 9\ 5\ 7$, le programme ne sauvegarde que les 4 premier entiers.
   (e) Test 5 : saisie du quatrième paramètre : avec $param1 = 1$, $param2 = 2$, $param3 = 4$, si $param4 = 4\ 5$, erreur trop couches sans neurones.
   (f) Test 6 : saisie du quatrième paramètre : avec $param1 = 1$, $param2 = 2$, $param3 = 4$, $param4 = 4\ 5\ 6\ 1$, si $param5 = a$ erreur de type.
2. Test quand l'utilisateur entre de bons paramètres :
   (a) Test 1 : saisie suivante : $param1 = 1$, $param2 = 1$, $param3 = 4$, $param4 = 4\ 5\ 6\ 1$, $param5 = 0$ renvoie la liste : (1 1 4 4 5 6 1 0).

Résultat :

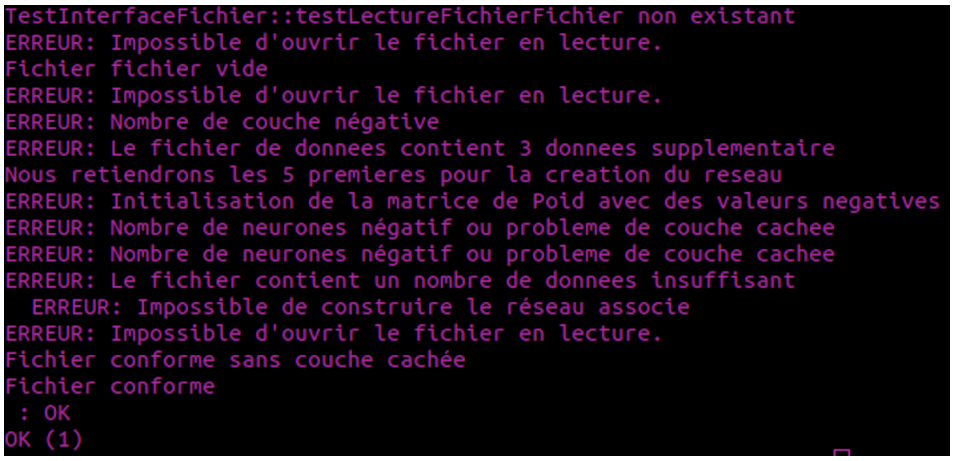


FIGURE 2.14 – Resultat Test interface

2.2.7 Matrice

2.2.7.1 La classe Matrice

voir la documentation doxygen

2.2.7.2 Les tests de la classe Matrice

Les tests sur la classe Matrice sont de 2 types :

1. Fournit le bon résultat
2. Traite le cas où les matrices ne sont pas de la même taille et vérifie si le nombre colonne est égal au nombre de lignes de la matrice passé en paramètre.

testProduit : prototype de la méthode : testProduit(Matrice)

1. Test pour 2 matrices 3x3 quelconques et retourne le bon résultat

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} . \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

$$\Downarrow$$

$$\begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{pmatrix}$$

FIGURE 2.15 – 1ère partie de test

2. Test pour 2 matrices de taille différentes (1x2 et 2x3) et retourne le bon résultat (une matrice 1x3)

$$\begin{pmatrix} 1 & 3 \end{pmatrix} . \begin{pmatrix} 1 & 8 & 4 \\ 7 & 1 & 6 \end{pmatrix}$$

$$\Downarrow$$

$$\begin{pmatrix} 22 & 11 & 22 \end{pmatrix}$$

FIGURE 2.16 – 2ème partie de test

3. Test pour 2 matrices de tailles incompatibles pour un produit matriciel et retourner erreur.

```
helal@admp:~/Bureau/Réseau de neurones/ResultTest$ ./TestMatrice
TestMatrice::testProduit : OK
OK (1)
```

FIGURE 2.17 – Resultat TestMatrice

Resultat :

2.2.8 ResForwarded

voir la documentation doxygen

2.2.9 Neurone

voir la documentation doxygen

Chapitre 3

Conclusion

Ce projet nous a d'abord permis d'appliquer et d'appronfondir nos connaissances en $C++$ et en génie logiciel (*Github, Agile...*). Nous avons pris conscience de la nécessité de mettre en oeuvre des tests unitaires, nottament avec *cppunit*, afin de vérifier au fur et à mesure le bon fonctionnement des différentes parties de notre code.
Attirés par la science des données, nous avons également pris plaisir à découvrir les réseaux de neurones et leur implémentation. Cependant, malgré un long travail, nous avons manqué de temps pour terminer notre projet. Un travail préliminaire important était nécessaire pour maitriser le sujet.
Enfin, nous avons su nous organiser afin de répartir la charge de travail et mettre à profit les qualités de chacun.

Table des figures

Bibliographie

[1] http ://benoit.decoux.free.fr/ENSEIGNEMENT/PROGRAMMATION/projet_RN_CPP.pdf

[2] https ://www.juripredis.com/fr/blog/id-19-demystifier-le-machine-learning-partie-2-les-reseaux-de-neurones-artificiels

[3] https ://www.lebigdata.fr/perceptron-machine-learning

[4] https ://www.youtube.com/watch ?v=sK9AbJ4P8ao

[5] https ://zestedesavoir.com/forums/sujet/6567/classe-generique-pour-des-reseaux-de-neurones-en-c/

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

[7] http ://lexicometrica.univ-paris3.fr/jadt/jadt2000/pdf/47/47.pdf

[8] http ://nicolascormier.com/documentation/ia/cours_IA/node102.html

[9] https ://dataanalyticspost.com/Lexique/reseaux-de-neurones-recurrents/

[10] https ://halshs.archives-ouvertes.fr/halshs-02096266/document

[11] https ://www.tandfonline.com/doi/pdf/

[12] https ://www.rncan.gc.ca/cartes-outils-publications/imagerie-satellitaire-photos-aer/tutoriels-sur-la-teledetection/analyse-interpretation-dimages/classification-et-analyse-des-images/9362

[13] https ://weave.eu/deep-learning-service-de-linformatique-affective/

[14] https ://ledatascientist.com/introduction-a-la-categorisation-de-textes/

[15] https ://www.aquiladata.fr/classification-dimages-et-detection-dobjets-par-cnn/