INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Réseaux de neurones

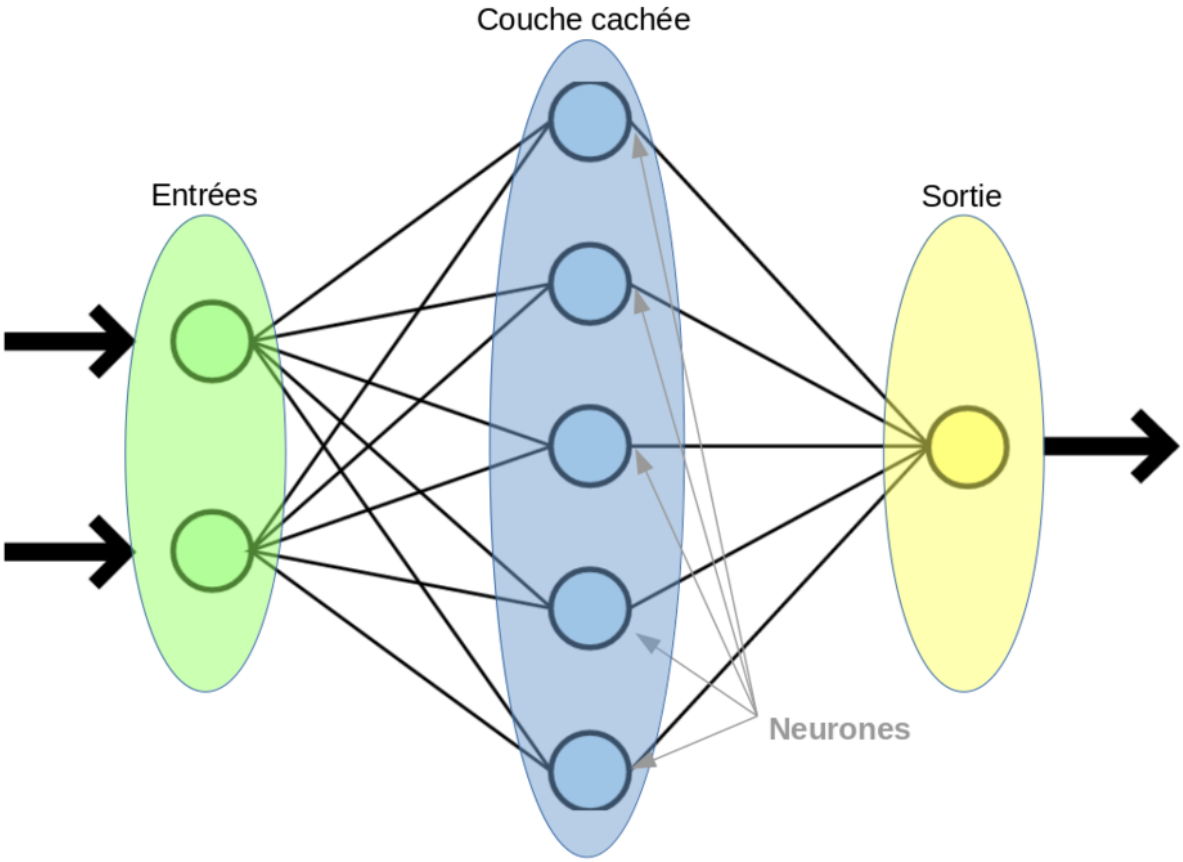


FIGURE 1 – Réseaux de neurones

Réalisé par :
Charlotte VIENNEY
Louis PADÉ
Hengshuo LI
Yuchen MO
Peiyao LI
Awa FAYE
Ndeye Fatou DIOP
Roxane LEDUC

Encadré par :
M. KOTOWICZ

GM4 / Février 2023 - Mai 2023

Table des matières

1 Présentation **3**

2 Cahier des charges **4**

2.1 Présentation générale 4

2.2 Description des fonctionnalités 6

2.3 Versions du projet 9

3 Cahier de spécification **10**

3.1 Diagramme de cas d'utilisation 10

3.2 Diagramme de classe 12

3.3 Diagrammes de séquence 18

3.3.1 Cas "Construire le réseau de neurone" 18

3.3.2 Cas "Faire apprentissage" 20

3.3.3 Cas "Tester les performances de l'apprentissage" 22

3.3.4 Cas "Observer la sortie du réseau" 23

4 Spécifications techniques **24**

4.1 Documentation des interfaces de chaque module 24

4.2 Tests unitaires 25

5 Qu'est ce qu'un réseau de neurones ? **32**

5.1 Réseaux de neurones à propagation avant (feedforward) 34

5.2 Réseaux de neurones récurrents (recurrent neural networks - RNN) 35

5.3 Auto-encodeurs et auto-encodeurs varitationnels 38

5.3.1 Auto-encodeurs 38

5.3.2 Auto-encodeurs varitationnels 38

5.4 Réseaux de neurones convolutifs (convolutional neural networks - CNN) 40

5.5 Réseaux antagonistes génératifs (generative neural networks) 42

5.6 Réseau de l'État d'Écho (Echo state networks - ESN) 44

5.6.1 Introduction 44

5.6.2 Equation 44

5.6.3 Phase d'échantillonnage 44

5.6.4 La phase de calcul des poids 44

6 Détails d'implémentation **45**

6.1 Classe Matrix 45

6.2 Classe TrainingFonction 45

6.3 Classe Layer 46

6.4 Classe Output_layer 47

6.5 Classe Network 47

6.6 Classe Datasource 48

7 Conclusion **50**

8 Bibliographie **51**

1 Présentation

Dans le cadre de notre cours de C++, nous allons réaliser un projet qui a pour objectif l'implémentation d'un code permettant la création de divers types de réseau de neurones.

Dans un premier temps, des recherches auront lieu afin de pouvoir comprendre les réseaux de neurones, c'est-à-dire leur fonctionnement, leurs composantes. Une fois les recherches effectuées et différents types de réseau de neurones trouvés, il faudra trouver leurs points communs afin de pouvoir mettre en place par la suite le principe d'héritage et avoir une idée de comment coder cela.

Puis, dans un second temps, nous passerons à la partie de conception avec le cahier des charges, les diagrammes UML et pour conclure la conception d'un code.

Bien évidemment, le sujet étant très général, nous ne pourrons implémenter en C++ l'ensemble des réseaux de neurones que nous aurons découvert et présenté. C'est pourquoi, nous nous concentrerons sur un de ces réseaux de neurones en l'appliquant sur un exemple.

Notre exemple ici sera la reconnaissance de caractères manuscrits dans une image par rapport à une base connue de caractères.

2 Cahier des charges

2.1 Présentation générale

Passons désormais à une analyse plus complète de notre projet afin de pouvoir concrètement comprendre ce que nous pourrons faire par le truchement de notre projet.

Notre projet traitant des réseaux de neurones, nous avons pour but “général” de mettre en œuvre un code offrant la possibilité de créer divers types de réseaux de neurones comme nous avons pu le dire précédemment dans notre brève introduction. Pour développer un peu cette partie, nous savons qu'il existe un grand nombre de type de réseaux de neurones, que cela soit avec les réseaux à apprentissages supervisés sans rétropropagation avec par exemple le perceptron ou la machine de Cauchy. Dans le même type d'apprentissage nous avons les réseaux de neurones avec cette fois-ci rétropropagation, nous pouvons citer pour exemple le perceptron multicouche. La seconde grande classe est quant à elle l'apprentissage supervisée, cette classe n'étant apparemment composée que de réseaux de neurones avec rétropropagation.

Nous voyons bien qu'il existe énormément de réseaux de neurones différents, proposant chacun divers avantages et inconvénients, il est évident qu'ils ont tous une utilité claire dans des domaines précis ou dans des applications précises. Ainsi, si nous pouvons proposer à l'utilisateur d'utiliser le réseau de neurone qu'il souhaite, celui-ci pourra utiliser notre code pour un très grand nombre d'applications et par conséquent ne pas avoir à se soucier de quels codes utiliser pour quelle application.

Ce projet est très ambitieux, d'une part, vouloir coder la plupart des réseaux de neurones existants est une tâche longue et demandant une grande connaissance par rapport à l'ensemble de ces réseaux de neurones et non un seul d'entre eux. De plus, nous pourrons effectuer des mises à jour afin de rajouter des nouveaux réseaux de neurones.

Toutefois, si ce projet arrive à être réalisé, il pourrait apporter une grande aide à de nombreuses personnes, comme dit ci-dessus, en regroupant et en proposant la possibilité d'utiliser n'importe quel réseau de neurone à partir d'un "même" code avec une base "simple". Les personnes auraient alors déjà la possibilité d'utiliser le réseau de neurones souhaité et aussi le fait de pouvoir découvrir d'autre type de réseaux de neurones de manière "simplifiée" sans avoir besoin de passer par d'autres bibliothèques notamment.

La partie de recherche aura principalement deux grandes utilités, permettre aux dévelloppeurs de découvrir les réseaux de neurones et les différents types existants et aussi permettre à la création de la documentation de chaque réseau dans l'application, cette documentation qui sera par la suite utilisée par les utilisateurs.

Pour résumer ce projet, le but serait de créer une "boîte à outils" de réseaux de neurones, avec une interface graphique permettant à l'utilisateur de trier les réseaux de neurones disponibles entre les types, les domaines d'application ou encore la difficulté de ceux-ci. L'utilisateur aurait aussi la possibilité d'utiliser des réseaux de neurones "pré-définis" avec un nombre de couches précis, des fonctions d'activation déjà connues etc... Cela lui permettant d'avoir une base utilisable, bien sûr celui-ci aura aussi la possibilité de créer manuellement son réseau de neurones.

A côté, nous offrirons aussi des bases de données prêtes à utilisation qui seront observables et modifiables. Finalement, nous pourrions aussi ajouter des exemples d'application et pourquoi pas rajouter une manière de vérifier le bon fonctionnement de notre réseau de neurones et de voir le temps de calcul.

Maintenant que le projet est présenté, nous allons présenter plus particulièrement ce que nous allons faire, ce que nous pourrons ajouter si nous avons la possibilité et finalement ce que nous pourrions faire avec plus de temps.

Tout d'abord, nous allons implémenter un type de réseau de neurones voir deux afin de pouvoir répondre à deux problèmatiques : la première est de créer la fonction 'XOR' et la seconde est quant à elle la plus intéressante, l'objectif sera de pouvoir reconnaître un caractère écrit de manière manuscrite par une personne ou non, dans une image. Bien entendu, nous mettrons aussi en place le code général décrivant un réseau de neurone de façon généraliste et une description de la méthode, de plus nous mettrons des bases de données afin de pouvoir entraîner et tester nos modèles.

Si nous avons le temps, nous ajouterons possiblement une interface graphique ou un autre type de réseau de neurones, une autre possibilité serait de rajouter la fonctionnalité permettant d'afficher le temps de calcul ou l'évolution de notre réseau de neurones en voyant les erreurs.

Il y a énormément de fonctionnalités, de données, de contenus que nous pourrions ajouter avec plus de temps et plus de moyens, ce que nous allons faire peut-être vu comme une ébauche, le début d'un projet beaucoup plus grand et développé.

2.2 Description des fonctionnalités

Concentrons-nous sur les fonctionnalités que nous allons implémenter :

- Fonctionnalité 1 : Choisir les données d'apprentissage
  - F1.1 : Choisir la base de données
  - F1.2 : Choisir un nombre de données initiales
  - F1.3 : Choisir les données initiales de deux manières : aléatoire ou les X premières
  - F1.4 : Choisir la taille des lots pour le calcul des nouveaux poids

- Fonctionnalité 2 : Choisir les données de test
  - F2.1 : Choisir les données provenant d'une base de données
    - F2.1.1 : Choix de la base de données
    - F2.1.2 : Choix du nombre de test

- Fonctionnalité 3 : Choisir les données d'application
  - F3.1 : Choix des données d'application : données de la base de données ou données de l'utilisateur

- Fonctionnalité 4 : Afficher les résultats
  - F4.1 : Afficher le pourcentage d'erreur
  - F4.2 : Afficher pour chaque donnée l'image avec le résultat du réseau de neurone

- Fonctionnalité 5 : Choisir les paramètres du réseau de neurones
  - F5.1 : Le type principal du réseau de neurones
  - F5.2 : Choisir les spécificités du réseau de neurones
    - F5.2.1 : Choisir le nombre de couches cachées
      - F5.2.1.1 : Choisir le nombre de neurones pour chaque couche cachée
      - F5.2.1.2 : Choisir les fonctions d'activation pour chaque couche cachée
    - F5.2.2 : Choisir les neurones en entrée : le format
    - F5.2.3 : Choisir les sorties du réseau : le format
    - F4.2.4 : Choisir la fonction de perte
  - F5.3 : Choisir les poids initiaux du réseau de neurones
    - F5.3.1 : Choisir les poids : manière aléatoire ou en fonction d'un modèle existant si possible

Nous allons maintenant décrire de manière plus précise les fonctionnalités listées ci-dessus.

Choisir les données d'apprentissage :

Données d'apprentissage : les données initiales sur lesquelles le réseau de neurones va s'entraîner.

L'utilisateur doit pouvoir choisir les données sur lesquelles le réseau de neurones va s'entraîner, celui-ci aura le choix entre plusieurs fichiers déjà présents dans l'application. Il n'y aura donc pas, pour les données d'apprentissage, la possiblité de choisir diverses bases de données, cependant l'utilisateur pourra tout de même choisir l'une des bases disponibles et le nombre de données servant cet apprentissage. On pourra par exemple supposer que celui-ci souhaite prendre 500 données parmis toutes celles de la base choi-

sie ou plutôt un pourcentage de cette base, on pourrait aussi proposer des quantités si celui-ci ne sait pas vraiment combien de données choisir.

De plus, on peut aussi proposer à l'utilisateur de choisir les données de manière aléatoire dans la base, ce qui est le mieux, ou de les choisir dans un ordre précis, par exemple on pourrait imaginer qu'il veuille prendre les x premières ou dans un intervalle etc..., en fonction de ses envies cela revient à choisir indirectement un nombre de données...

Un autre point à noter est le fait de choisir la taille des lots de données qui seront utilisés pour l'apprentissage et notamment pour le calcul des nouveaux poids à chaque étape.

Choisir les données de test :

Données de test : données sur lesquelles nous allons tester le réseau de neurones lors de son apprentissage afin de savoir si celui-ci est bien entraîné ou non.

Pour le choix des données de test, ce choix va en partie dépendre du choix des données initiales si l'utilisateur veut rester avec la même base de données dans laquelle il a entraîné son modèle ou si il souhaite plutôt choisir une autre base de données disponibles dans les fichiers. Ainsi, on retrouve, comme lors de la sélection des données d'apprentissage, le choix de ces fameuses données.

Choisir les données "d'application" :

Données "d'application" : données sur lesquelles l'utilisateur va utiliser le réseau de neurones.

Il est à noter ici que l'utilisateur possède une option supplémentaire pour les données d'application, en effet celui-ci aura aussi la possibilité de donner ses propres données, pour cela il aura plusieurs choix. Bien sur, il pourra aussi prendre des données provenant des bases de données disponibles avec l'application.

Le premier choix étant de donner simplement une image au programme dans un format précis, le programme traitera alors l'image pour l'avoir sous le format nécessaire afin de la donner au réseau de neurones. Le second, quant à lui, est de directement donner au programme le format pour le réseau de neurone et non une image, de cette manière le programme n'aura pas besoin de traiter l'image mais l'utilisateur aura quant à lui ce devoir.

Affichage des résultats :

Lorsque l'on entraîne un réseau de neurones, le but est que celui-ci puisse par la suite donner des résultats corrects dans le contexte dans lequel nous l'entraînons, ainsi dans notre cas, nous allons proposer à l'utilisateur plusieurs possibilités pour l'affichage des résultats sur les données test.

En premier lieu, il pourra afficher le pourcentage d'erreur entre les données observées et ce que le réseau propose comme résultats, ce qui est finalement un affichage plutôt ba-

sique permettant simplement de voir l'efficacité de notre réseau de neurones. Une autre possibilité s'offrant à l'utilisateur serait de pouvoir, d'une part, voir le pourcentage d'erreur mais aussi, et surtout, de pouvoir voir la donnée qui a été testé, qui correspondra ici à une image comportant un chiffre, et le résultat proposé par le réseau, qui sera ici le chiffre qu'il pense avoir reconnu. Notons aussi que l'affichage lors de "l'application" du réseau de neurones, par l'utilisateur, sur une image qu'il a rentré sera de la forme d'un vecteur où la première composante correspondra à la probabilité, en pourcentage, selon le réseau de neurones que cela soit un 0 dans l'image, puis la deuxième composante pour la probabilité que cela soit un 1 et ainsi de suite jusqu'à 9.

De cette manière, nous pourrons concrètement voir si notre modèle reconnaît ou non, cela est aussi utile pour parfois expliquer certains résultats, on pourrait imaginer par exemple un chiffre que même nous ayons du mal à le reconnaître alors ce n'est pas forcément problèmatique si le réseau n'a pas non plus réussi à le reconnaître.

Choisir le type de réseau de neurones :

Ce projet traitant des réseaux de neurones, l'utilisateur aura nécessairement besoin de pouvoir travailler sur ces derniers. Cet utilisateur va avoir plusieurs choix que nous allons voir, le premier choix étant tout d'abord de choisir le type de réseau de neurones qu'il souhaite. Nous avons vu dans la partie bibliographique au début de ce rapport qu'il existe de nombreux types de réseau de neurones, nous allons donc proposer à l'utilisateur le choix entre divers réseaux que nous aurons dans l'application.

Cependant, une question qui vient naturellement se poser est de savoir si l'utilisateur veut utiliser un réseau de neurones déjà implémenté ou le créer lui-même ?

La réponse à cette question est alors la suivante : l'utilisateur pourra tout d'abord choisir le type de réseau l'intéréssant, une fois ce choix effectué, celui-ci aura deux possibilités. La première étant de prendre un modèle déjà existant et alors il n'apporte pas de modifications, il aura juste la possibilité d'entraîner ce modèle sur des données.

La seconde possiblité, et la plus pertinente, est de proposer la création du réseau de neurones à l'utilisateur, comme on peut retrouver dans certaines bibliothèques. Ainsi, il pourra choisir l'entrée et la sortie du réseau, le nombre de couches cachées, le nombre de neurones par couches cachées ou encore la fonction d'activation pour les couches.

Pour décrire de manière plus précise le choix des paramètres du réseau de neurones, il se passera selon la méthode suivante : dans un premier temps nous allons demander le nombre de couches cachées puis pour chaque couche chachée nous allons choisir le nombre de neurones par couche et la fonction d'activation. Une fois la "base" du réseau créée, il ne restera plus qu'à choisir le format des données en entrée et le format des données en sortie, le format faisant référence au nombre de neurones pour ces couches finalement. Pour la sortie, il y a aussi le choix de la fonction de perte pour calculer l'erreur une fois l'estimation obtenue, ainsi que le choix de la fonction d'optimisation. Il est à noter que les poids initiaux seront un paramètre à choisir dans les deux cas, pour ces poids nous retrouverons encore deux possiblités : les choisir de manière aléatoire ou se baser sur des modèles déjà existants pour les initialiser.

2.3 Versions du projet

Cette partie est consacrée à la présentation des diverses versions du projet, l'idée est de présenter l'évolution, que nous souhaitons, du projet au fil du temps, avec initialement une application ne possédant pas forcément beaucoup de fonctionnalités jusqu'à finalement une application proposant de nombreuses fonctionnalités.

Version 1

Dans la première version de ce projet nous retrouverons les fonctionnalités suivantes :

- Un type de réseau de neurones d'implémenté sans paramètres modifiables
- Une base de données pour l'entraînement, les tests lors de l'entraînement et pour l'utilisation de l'utilisateur
- La possibilité de choisir le nombre de données pour l'apprentissage et les tests
- Le choix de l'initialisation des poids

Version 2

La deuxième version de ce projet comportera les fonctionnalités de la première version avec l'ajout des fonctionnalités suivantes :

- La possibilité de donner en entrée une image sous un format précis
- L'affichage de l'entrée et de la sortie dans la console
- L'ajout d'autres bases de données pour l'apprentissage des réseaux de neurones
- La création de son propre réseau de neurone, modification des paramètres (fonction d'activation, fonction de perte, nombre de neurones par couches cachées, nombre de couches cachées)

Version 3

La troisième et, normalement, dernière version apportera les nouveautés suivantes, cette version ne sera sûrement pas faites par manque de temps :

- Une interface graphique
- L'ajout de nouveaux type de réseaux de neurones
- L'affichage de l'entrée et de la sortie de manière propre (hors console)
- La possibilité de passer des images en entrée sous divers formats

3 Cahier de spécification

3.1 Diagramme de cas d'utilisation

FIGURE 2 – Diagramme de cas d'utilisation

Description du diagramme de cas d'utilisation :

L'utilisateur pourra effectuer plusieurs opérations dans le système, à savoir construire le réseau de neurone, faire l'apprentissage du réseau de neurone, tester les performances de l'apprentissage pour évaluer si l'apprentissage a été efficace et finalement tester le réseau avec une image pour voir les performances du réseau de neurones.

— **Construction du réseau de neurone :**
Dans la phase de construction du réseau de neurone, l'utilisateur devra choisir les fonctions de perte et d'optimisation ainsi que le nombre de couches cachées du réseau. Pour chaque couche, il devra définir le nombre de neurones associé, les fonctions d'activation, la valeur des poids initiaux de liaison entre chaque neurone d'une couche et les autres neurones de la couche qui suit, sans oublier la valeur des biais qui pourra se faire de maniere aléatoire ou non.

— **Faire l'apprentissage :**
Pour faire l'apprentissage, il faut obligatoirement avoir construit au préalable le réseau de neurone. Ensuite, l'utilisateur pourra choisir sa base de donnée parmi celles présentes dans l'application puis ses données initiales qui peuvent être selectionnées dans la base choisie de manières aléatoires ou en considérant les X premières données.

— **Tester les performances de l'apprentissage :**
Dans la phase de test des performances de l'apprentissage il faudra choisir des données dites de tests, différentes des données utilisées lors de l'apprentissage afin de tester la capacité de généralisation du réseau de neurone. Le choix de ces données pourra être fait en selectionnant des données de la base existante ou en créant son propre jeu de données.

— **Observer la sortie du réseau sur image :**
On pourra également après la phase de test de l'apprentissage vouloir tester le comportement du réseau de neurone face à une image donnée. Dans ce cas là, en sortie nous aurons le chiffre dont la probablité de correspondance à l'image d'entrée est la plus élevée ainsi qu'un vecteur de taille 10 correspondant à la probabilité que l'image d'entrée corresponde à chaque chiffre.

3.2 Diagramme de classe

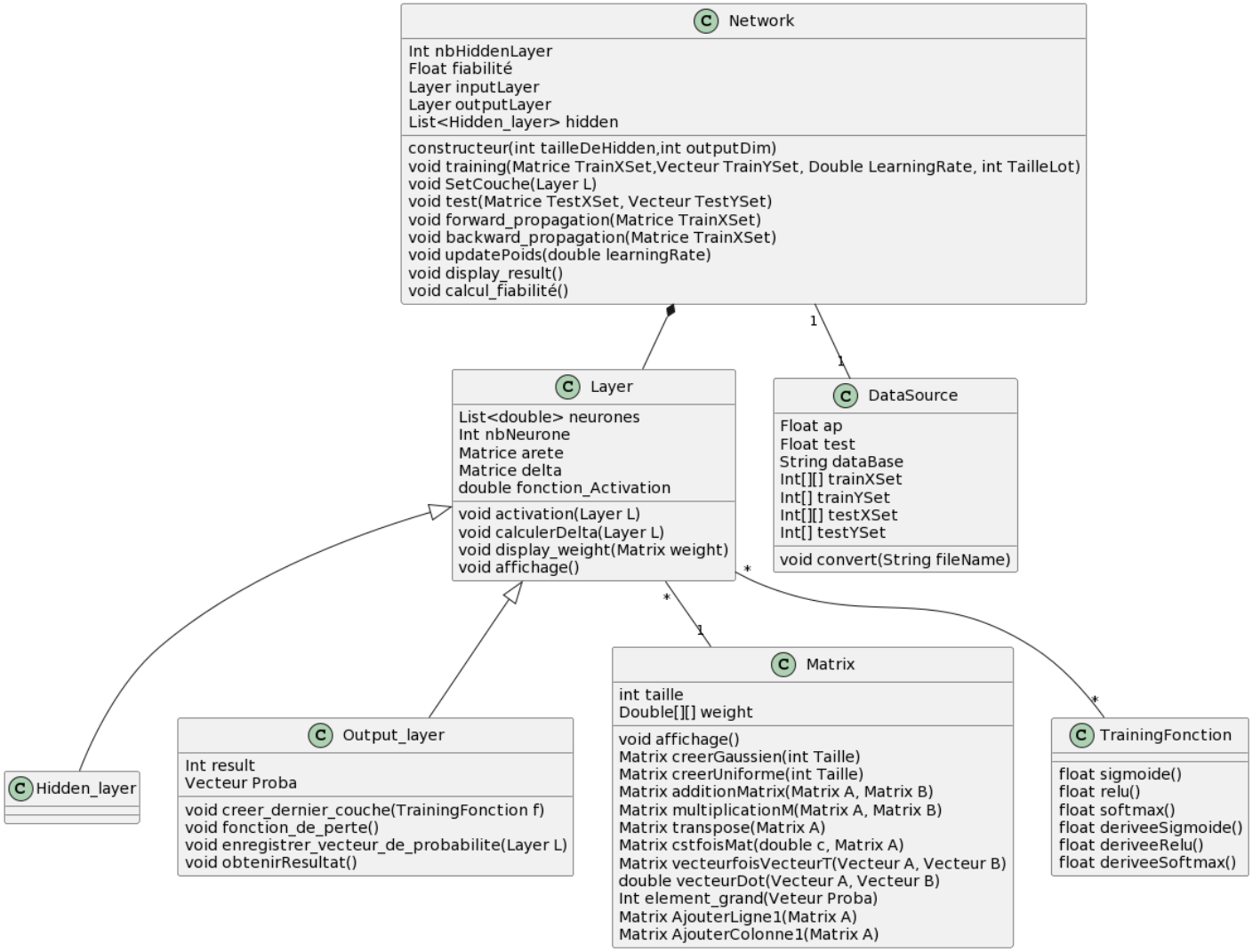


FIGURE 3 – Diagramme de classe

Description du diagramme de classes :

Classe Network :

Notre réseau de neurones **Network** est composé de couches **Layer**, si il n'y a plus de réseau de neurones alors nous allons partir du principe qu'il n'y a pas plus de couches du tout, une couche ne peut exister si il n'y a plus de réseau, ceci nous permet d'avoir une relation de composition entre **Network** et **Layer**. En suivant ce même principe nous avons une autre relation de composition entre nos couches **Layer** et nos neurones **Neurone**.

Un réseau de neurones est constitué d'un nombre de couches cachées **nbHiddenLayer**. Afin de pouvoir représenter l'ensemble des couches se trouvant dans notre réseau de neurones nous avons décidé d'utiliser une liste de couches cachées **hidden**, cela permettant d'accéder plus facilement aux diverses couches. Nous aurons aussi une couche de sortie qui sera ici **outputLayer**.

Une procédure que nous avons dans notre classe réseau est celle permettant de voir la sortie de celui-ci, elle permettra notamment d'afficher le fameux vecteur comportant la probabilité d'avoir les chiffres de 0 à 9 dans une image par exemple, soit la procédure **display_result()**.

Une autre procédure dont nous avons besoin est celle s'occupant d'entraîner le réseau de neurones, celle-ci s'occupera de toute la phase de mise à jour des poids, du test du réseau de neurones etc... Cette fonction est nommée **training()**. Cette fonction prendra en paramètres une matrice de données en entrée, le vecteur de sortie, un réel de vitesse d'apprentissage et le nombre de données dans le lot servant à l'apprentissage.

Une fois l'entraînement fini nous pouvons tester notre réseau de neurones à l'aide de la fonction **test()** qui prendre en paramètres une matrice de données en entrée et le vecteur de sortie de test.

Il est aussi plus que nécessaire de pouvoir mettre à jour les poids comme nous avons pu le dire juste avant, c'est pourquoi nous retrouvons la fonction **update_Poids()** qui prendre en paramètre une vitesse d'apprentissage sous la forme d'un réel **learning rate**. Cette fonction mettra à jour les poids une fois avoir donné à notre réseau un lot de données.

Les neurones jouant un rôle important dans les réseaux de neurones, il est important de pouvoir afficher les sorties de ceux-ci si l'on le souhaite, c'est pourquoi nous avons la fonction **forward_propagation()** qui a pour but d'afficher la valeur de sortie de chaque neurone et la fonction **backward_propagation()** qui, quant à elle, aura pour but d'afficher l'erreur au niveau de chaque couche. Ces deux fonctions prenant en paramètre une matrice de données.

Il ne faut pas oublier d'avoir un indicateur permettant de savoir si notre réseau de neurones est efficace ou non et aussi si celui-ci est fiable ou non, c'est pourquoi nous avons en attribut un réel **fiabilite** afin de représenter la fiabilité de ce réseau de neurones, ce réel sera compris entre 0 et 1 où 0% une fiabilité nulle et 1 sera une fiabilité de 100%.

Afin de connaître la fiabilité du réseau nous aurons besoin d'une fonction pour la calculer qui sera ici **calcul_fiabilité()** bien sur nous pourrons afficher cet indicateur si nous le voulons.

Afin de mettre à jour une couche si nous le souhaitons nous avons une fonction **SetCouche()** qui aura en paramètre une couche, il y a logiquement le constructeur afin de pouvoir créer l'ensemble des couches de notre réseau de neurones. Le constructeur qui aura, quant à lui, comme paramètres le nombre de couches cachées et la "taille" de la sortie.

Classe Layer, HiddenLayer et OutputLayer :

Un point important est la gestion des couches de notre réseau, en effet, elles constituent la principale base de celui-ci, nous sommes finalement parti sur le principe de créer deux classes spécifiques, une pour la couche de sortie et une autre pour les couches cachées, nous partons sur le principe d'avoir une classe **Layer** afin de gérer les couches de manière générale puis des classes héritant de cette classe avec une classe **HiddenLayer** et une classe **OutputLayer**.

La classe **Layer** sera composée d'une liste de neurones la composant **neurones**, du nombre de neurones dans cette couche qui sera ici **nbNeurone** et d'une matrice contenant les poids des arêtes reliant cette couche à la suivante, soit **arete**. Nous reviendrons sur ce qu'est une "matrice" dans la suite de ce document.

En plus d'avoir une matrice contenant les poids de chaque arête nous aurons aussi besoin d'une autre matrice contenant les Δ de chaque poids qui sera la matrice **delta**.

Il faut pouvoir calculer ces fameux Δ, la fonction **calculer_Delta()** permet de faire cela en enregistrant dans la matrice **delta**.

Tout de même, nous aurons une fonction permettant d'afficher cette fameuse matrice des poids qui sera **display_weight()** prenant en paramètre une matrice.

Une autre information nécessaire est la fonction d'activation étant utilisée par les neurones de la couche, étant donné qu'il existe un grand nombre de fonction d'activation nous aurons besoin de savoir laquelle est utilisée par telle couche. Ainsi, nous aurons un attribut **fonction_Activation** en plus dans la classe **Layer**.

Découlant directement du fait de savoir quelle fonction d'activation est utilisée dans une couche, nous aurons aussi besoin d'une procédure **activation()** permettant de calculer l'activation sur la couche et prenant en paramètre une couche.

Pour ce qui est de la classe **OutputLayer** nous retrouvons dans celle-ci un réel **result** correspondant à la sortie du réseau de neurones et une procédure **fonction_de_perte()** prenant en paramètre une fonction correspondant à la fontion de perte du réseau de neurones, qui sera en particulier ici l'entropie croisée.

De plus nous aurons une fonction **enregistrer_vecteur_de_probabilite()** qui enregistrera le résultat dans l'attribut **Proba**.

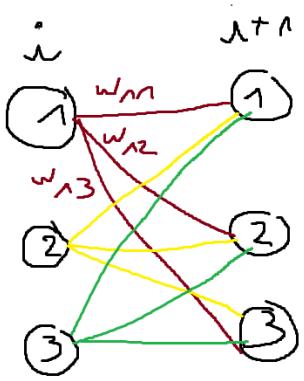
La création de la couche de sortie est aussi nécessaire pour la création de notre réseau de neurones, nous aurons la fonction **creer_dernier_couche()** prenant en paramètres la taille d'entrée de la couche, la taille de sortie et la fonction d'activation.

Classe Matrix :

Dans chaque réseau de neurones nous retrouvons des liaisons entre les neurones d'une couche et ceux de la couche suivante, pour le cas des couches cachées nous avons fait le choix de partir sur le fait que chaque neurone d'une couche i est relié à l'ensemble des neurones de la couche suivante, soit la couche $i + 1$.

En suivant ce choix, nous voyons que nous allons avoir pour chaque couche un ensemble de liens entre cette couche et la couche suivante, de plus en supposant que toutes les couches cachées ont le même nombre de neurones n alors nous aurons un total de n^2 liaisons d'une couche à la suivante.

Le point primordial étant que chacune de ces liaisons possède un poids, nous pouvons en déduire une manière de représenter cela : l'utilisation de tableaux de deux dimensions que nous nommerons ici **Matrix** qui seront composés de la manière suivante :



La matrice qui serait alors associée à la couche i serait la suivante pour l'exemple ci-dessus :

$$\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}$$

Où la première ligne correspondra au poids des liaisons entre le neurone 1 de la couche i et les neurones de la couche $i + 1$.

Suite à cela, nous en avons déduit deux choses, premièrement que nous aurions besoin d'une classe **Matrix** qui correspondrait à un tableau de réels à deux dimensions que nous venons de décrire ci-dessus.

Et surtout que pour cette classe nous aurions aussi besoin de diverses méthodes permettant de travailler sur des matrices. Les principales méthodes étant les suivantes : addition de matrice, addition de vecteur, transposition de matrice, multiplication matricielle, produit scalaire de deux vecteurs et de quelques autres opérations mathématiques.

Il est à remarquer que chaque couche possède une matrice des poids des arêtes entre cette dite couche et la couche suivante sauf pour la couche de sortie. Ajoutons aussi à cela que lors de l'initialisation des poids il est intéressant de donner des valeurs "uniformes" au poids ou totalement aléatoire.

Il est donc nécessaire d'avoir une procédure permettant d'initialiser des objets de la classe **Matrix**, cela pouvant être fait par le biais du constructeur où l'on pourrait passer en paramètre un entier faisant référence à la méthode d'initialisation de la matrice, par exemple **Matrix(int choix_initialisation)**.

Pour résumer nous aurons besoin de pouvoir effectuer des calculs élémentaires sur les objets de la classe **Matrix** comme l'addition avec **additionMatrix(Matrix A,Matrix B)**, la multiplication avec **multiplicationM(Matrix A,Matrix B)** ou encore la transposition avec **transpose(Matrix A)** ou encore **multiplicationHadamard(Matrix A, Matrix B)**. L'ensemble de ces opérations qui seront par la suite utilisée afin de pouvoir effectuer le calcul des poids ou encore de la sortie de la couche actuelle.

De plus, nous avons aussi fait le choix d'ajouter une procédure permettant de voir les poids des liaisons entre la couche actuelle et la couche suivante, ce qui revient tout simplement à afficher la matrice **weights**. Nous retrouvons ainsi une fonction d'affichage dans la classe **Matrix** ainsi que dans la classe **Layer**.

Pour les matrices de poids nous aurons besoin du nombre de colonnes et de lignes de ces matrices qui seront représentées par des entiers **nbColumns** et **nbRows**.

Classe TrainingFonction :

En parlant du calcul des poids, cela nous fait penser aux principales fonctions dont nous avons besoin dans un réseau de neurones, parmi ces "fonctions" nous retrouvons déjà la fonction d'apprentissage ou fonction d'activation qui aura ici une classe **TrainingFunction**, le principal but de cette classe est de comporter les principales fonctions d'activation que nous pourrons utiliser par la suite ainsi que les dérivées de ces fonctions

Un exemple de fonction d'activation serait la fonction "sigmoïde", "relu" ou encore la fonction "softmax". Ces fonctions prendront en paramètre des vecteurs.

Une particularité à citer par rapport aux fonctions d'activation est qu'une fonction d'activation sera associée à un neurone d'une couche, pour simplifier nous partons du principe que tous les neurones d'une même couche possède la même fonction d'activation, nous retrouvons donc bien ce que nous avions écrit dans la description de la classe **Layer**.

Cependant la couche qui correspondra à la couche d'entrée n'aura pas de fonction d'activation et la couche qui correspondra, quant à elle, à la couche de sortie aura une seule fonction d'apprentissage possible qui sera la fonction "softmax" et qui correspondra aussi à la fonction de perte. Par conséquent, l'utilisateur ne pourra choisir la fonction de perte.

Pour conclure sur les fonctions d'activation, nous rappelons que chaque **Layer** aura une fonction d'activation, cela explique pourquoi nous retrouvons dans cette classe un attribut correspondant à la fonction d'activation.

Classe DataSource :

La classe **DataSource** permet de gérer les données en entrée du réseau de neurones, nous retrouvons naturellement deux réels qui sont **ap** et **test** correspondant, respectivement, au pourcentage de données que le réseau utilisera pour l'apprentissage et pour le test.

Nous aurons aussi besoin d'une chaine de caractères afin de savoir quelle base de données a été choisi, cet attribut sera **dataBase**. Afin d'avoir en mémoire les données nécessaires pour l'apprentissage nous aurons un tableau d'entiers **trainXSet** et un tableau de données pour les test **testXSet**. De même nous aurons pour les sorties du réseau de neurones pour l'apprentissage **trainYSet** et pour les tests **testYSet**.

Finalement, nous aurons besoin d'une fonction afin de convertir les données de la base de données vers les données au format dont nous avons besoin, cette fonction sera **convert(String dataBase,Tableaux d'entiers dataArray)** transformant les données de dataBase dans le tableau dataArray.

3.3 Diagrammes de séquence

3.3.1 Cas "Construire le réseau de neurone"

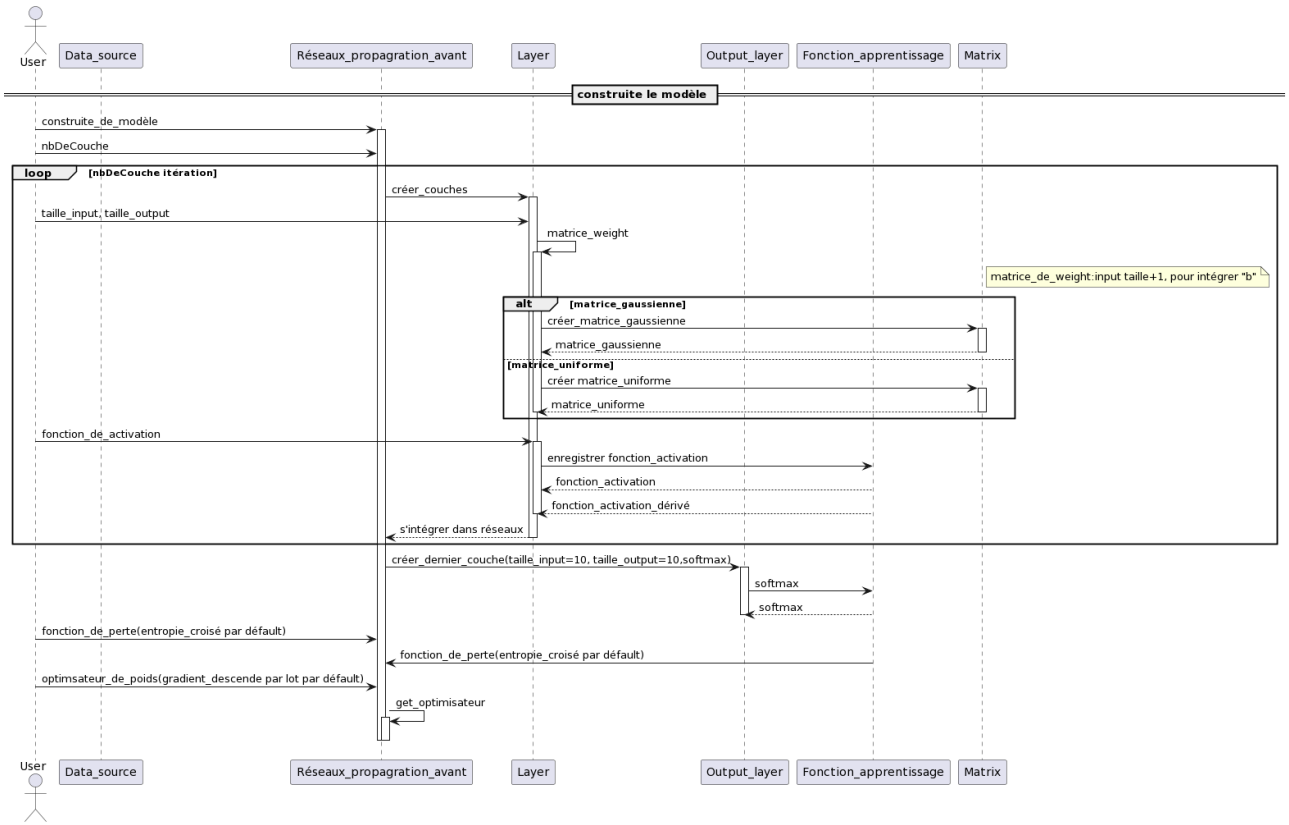


FIGURE 4 – Diagramme de séquence du cas "Construire le réseau de neurone"

Chaque flèche signifie la transmission d'une information ou le fait de solliciter une méthode.

Partie "Construire le réseau de neurone" :

Tout d'abord , "Réseaux propagation avant, qui correspond à la classe "Network", reçoit une indication de créer le modèle, simultanément l'utilisateur va indiquer à "Réseaux propagation avant" combien de couches il souhaite créer.

"Réseaux propagation avant" va créer une liste de couche, pour chaque couche, l'utilisateur va fournir la quantité de neurones de la couche précédente (taille input) et la quantité de neurones de cette couche pour construite la matrice de poids. (Pour la première couche, la "taille input" est le nombre de pixels et il faut faire attention que cette "taille input" de chaque couche coïncide avec "taille output" de la couche précédente).

Le "layer" va créer la matrice de poids par deux méthodes, les valeurs pourront suivre des lois soit gaussiennes soit uniformes.

Puis l'utilisateur va fournir le nom de la fonction d'activation de cette couche et le "layer" va enregistrer cette fonction d'activation et sa dérivée. Enfin, une fois que cette couche sera construite parfaitement, elle sera intégrée à la liste de couches.

Après avoir finir de créer la couche cachée, "Réseaux propagation avant" va créer la couche de sortie en fournissant la fonction de softMax pour unifier le résultats comme une loi de probabilité.

Pour finir de construire le modèle, le "Réseaux propagation avant" va enregistrer un pointeur sur la fonction de perte et la méthode d'optimissation des poids fournis par l'utilisateur.

3.3.2 Cas "Faire apprentissage"

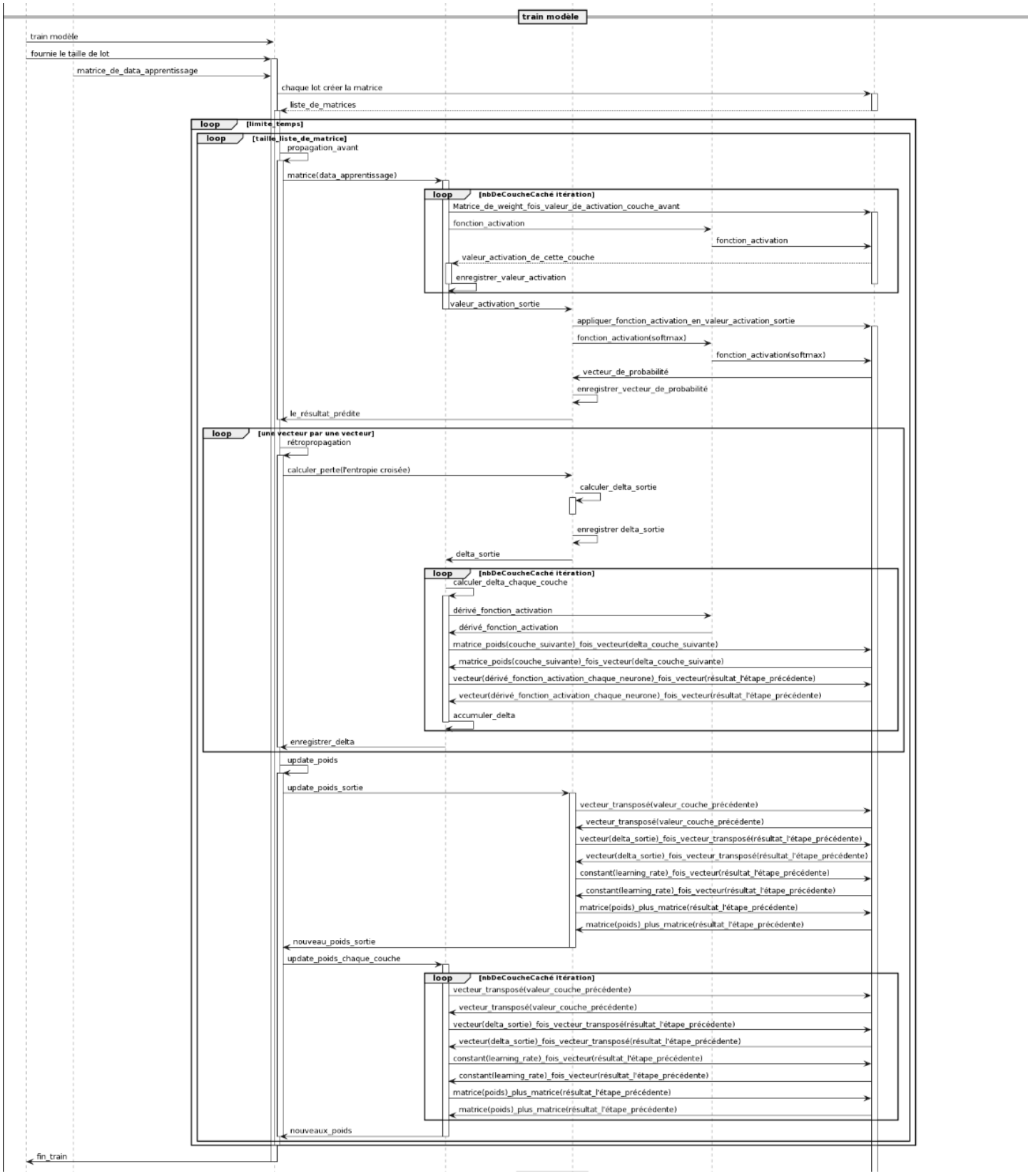


FIGURE 5 – Diagramme de séquence du cas Faire apprentissage

Nous avons divisé toutes les données en lots. Pour la matrice composée de chaque lot de données, nous effectuons une propagation vers l'avant et chaque couche se voit attribuer une fonction d'activation par l'utilisateur. Dans la dernière couche, nous utilisons la

fonction "softmax" pour créer plusieurs classifications. Avec la fonction "softmax", on a la probabilité que le $i-ème$ donnée soit dans la classe j comme

$$p_{i,j} = \frac{e^{z_{i,j}}}{\sum_{k=1}^{K} e^{z_{i,k}}}$$

où $z_{i,j}$ est le score d'origine que le modèle prédit que le $i-ème$ échantillon appartient à la $j-ème$ classe et on a K classes. Nous enregistrons la distribution de probabilité de chaque donnée sur toutes les classes en tant que valeurs prédites.

Ensuite, nous rétropropageons chaque élément de données dans le lot séparément. La première chose à faire est de calculer la perte. Les modèles qui utilisent la fonction softmax pour la multi-classification utilisent généralement la fonction de perte d'entropie croisée. La formule est la suivante. Pour chaque donnée i

$$L_i = -\sum_{j=1}^{K} t_{i,j} log(y_{i,j})$$

où $y_{i,j}$ est la probabilité que le $i-ème$ donné appartienne à la $j-ème$ catégorie, et utilise $t_{i,j}$ pour indiquer si le vrai label du $i-ème$ donné appartient à la $j-ème$ catégorie (si oui, c'est 1, sinon c'est 0).

L'algorithme de rétropropagation peut être utilisé pour calculer le gradient de chaque paramètre (y compris les poids et les biais des couches cachées) à travers le réseau, en les mettant à jour pour minimiser la fonction de perte. Supposons que notre réseau de neurones ait L couches, où la $l-ème$ couche est la couche cachée, et $w_{i,j}^{(l)}$ est le poids reliant le $i-ème$ neurone de la $(l-1)-ème$ couche au $j-ème$ neurone de la $l-ème$ couche.

Nous les mettons à jour à l'aide d'algorithmes d'optimisation tels que la descente de gradient. Le processus de chaque mise à jour peut utiliser la formule suivante :

$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \frac{\partial L}{\partial w_{i,j}^{(l)}}$$

où η est taux d'apprentissage.

Par conséquent, pour chaque donnée, nous pouvons obtenir les formules suivantes :

$$\frac{\partial L}{\partial w_{i,j}^{(l)}} = \delta_j^l = \begin{cases} (y_j - t_j) h_i^l & \textit{pour la couche sortie} \\ f^{'}(h_j^l) \sum_k w_{j,k}^{l+1} \delta_k^{l+1} & \textit{pour chaque couche cachée} \end{cases}$$

où $f^{'}(h_j^l)$ est la dérivée de la fonction d'activation du $j-ème$ neurone dans la $l-ème$ couche, et h_j^l est l'entrée pondérée du $j-ème$ neurone dans la $l-ème$ couche.

Donc, dans notre diagramme de séquence, nous devons d'abord calculer le δ^l de chaque couche, et l'accumuler jusqu'à ce que toutes les données de ce lot soient calculées, et nous sauvegardons le δ^l accumulé. Nous effectuons une mise à jour de poids sur la matrice de poids de chaque couche.

$$W^{(l)} \leftarrow W^{(l)} - \eta \delta^l$$

Puisque nous utilisons tous des calculs matriciels, nous devons entrer dans la classe Matrix pour appeler les méthodes requises pour les calculs matriciels.

Nous ferons une propagation vers l'avant et une rétropropagation de chaque lot de données jusqu'à ce que tous les lots aient été traités. Ensuite, recommençons à partir du premier lot de données, jusqu'à ce que le temps d'entraînement défini arrive, nous terminons l'entraînement.

3.3.3 Cas "Tester les performances de l'apprentissage"

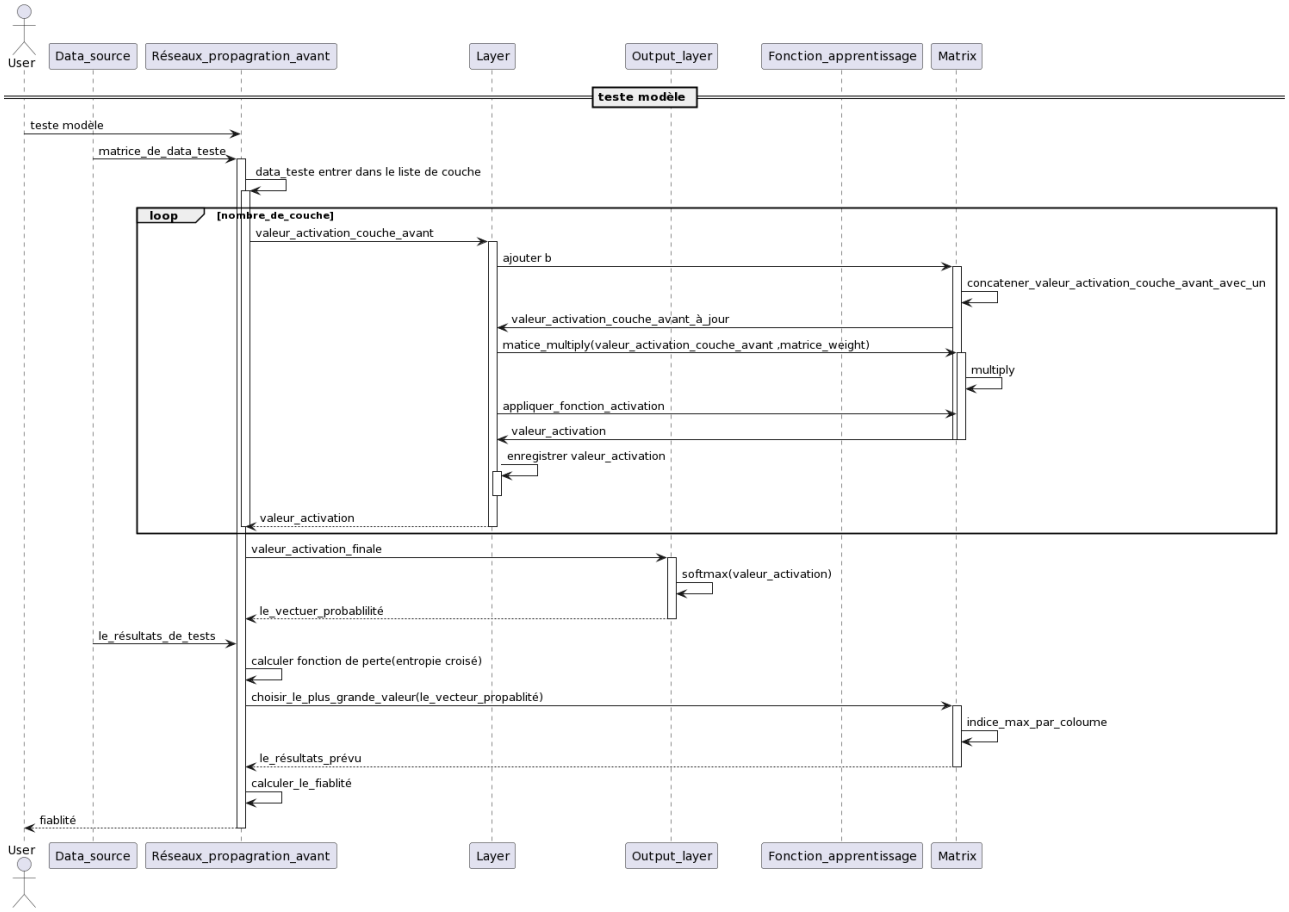


FIGURE 6 – Diagramme de séquence du cas "Tester les performances de l'apprentissage"

Partie "Tester les performances de l'apprentissage" :

Les données de test vont être fournies par une grande matrice et devenir comme des valeurs d'activation de la couche avant la première couche.

Dans la loop, la matrice "valeur activation couche avant" va concaténer une ligne de 1 (ajouter b) à la fin de matrice (si calculer dans l'ordre WX) et puis faire $f(WX)$ pour obtenir la valeur d'activation de cette couche et l'enregistrer,(f est la fonction de activation, W la matrice de poids et X la matrice de "valeur activation couche avant").

Après avoir passé toutes les couches cachées, on obtient la valeur d'activation finale, et on la transforme (unification) en un vecteur de probablilité (somme des composantes est égale 1) pour chaque colonne de la matrice.

Et puis il faut calculer l'erreur par la fonction de perte enregistrée et les résultats de test

sont alors fournis.

Choisir l'indice de probabilité de la plus grande de chaque colonne pour former une vecteur "le résultats prévu" et comparer avec "le résultats de tests" pour calculer le fiabilité.

3.3.4 Cas "Observer la sortie du réseau"

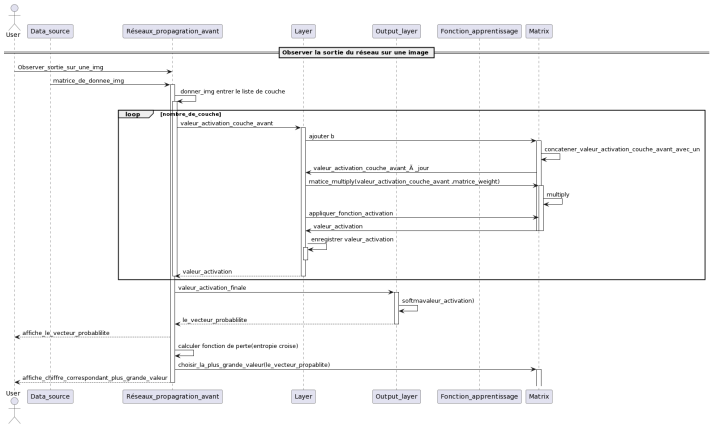


FIGURE 7 – Diagramme de séquence du cas "Observer la sortie du réseau"

Pour observer la sortie finale du réseau de neurones, l'utilisateur utilise la fonction application qui prend en paramètre une image. Le système répond à l'appel de cette fonction en affichant le chiffre qu'il croit correspondre à cette image un vecteur de taille 10 dont chaque ligne va correspondre à la probabilité selon le réseau de neurones que l'image donnée corresponde à chaque chiffre, tout ceci dans le but de vérifier les performances du réseau. Ainsi, on pourra vérifier si la phase d'apprentissage du réseau s'est bien passée et donc , l'efficacité du réseau de neurone.

4 Spécifications techniques

4.1 Documentation des interfaces de chaque module

Doxygen est un outil de documentation de code source qui permet de générer automatiquement une documentation en différents formats (HTML, LaTeX, RTF, etc.) à partir de commentaires dans le code. La documentation Doxygen est utile pour :

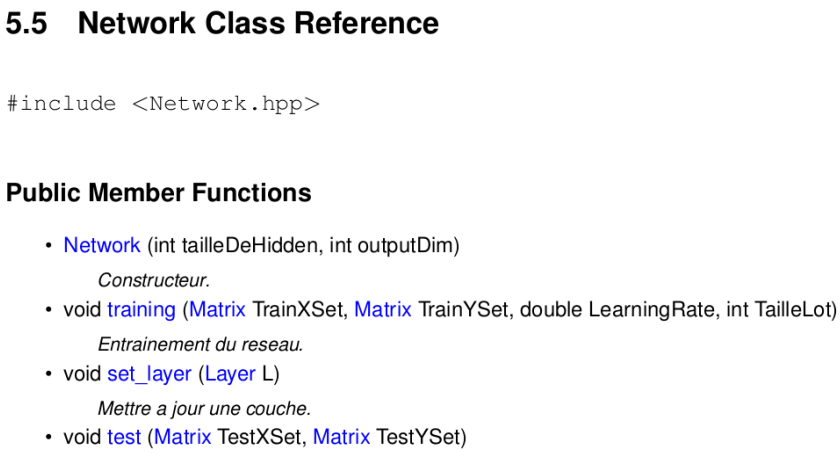
— Comprendre rapidement comment utiliser une bibliothèque, une API ou un module en lisant la documentation générée automatiquement.

— Faciliter la maintenance du code en fournissant une documentation claire et précise qui explique le but de chaque fonction, classe et variable, ainsi que les paramètres d'entrée et de sortie.

— Améliorer la collaboration entre les membres d'une équipe en fournissant une documentation commune.

Pour utiliser Doxygen, nous avons commencé par ajouter des commentaires dans notre code source. Nous avons ensuite généré la documentation en exécutant la commande doxygen sur le fichier de configuration (fichier texte qui contient des paramètres pour configurer la génération de la documentation).

La documentation générée peut alors être consultée en ouvrant le fichier index.html (présente dans le répertoire html) dans notre navigateur Web ou directement depuis le répertoire latex.



5.5 Network Class Reference

`#include <Network.hpp>`

Public Member Functions

- Network (int tailleDeHidden, int outputDim)
  Constructeur.
- void training (Matrix TrainXSet, Matrix TrainYSet, double LearningRate, int TailleLot)
  Entrainement du reseau.
- void set_layer (Layer L)
  Mettre a jour une couche.
- void test (Matrix TestXSet, Matrix TestYSet)

FIGURE 8 – Documentation générée par Doxygen

4.2 Tests unitaires

Tests unitaires dans Layer

— *void activation(Layer L)* : cette procédure permet de calculer l'activation dans chaque couche. Elle prend en paramètre une couche L. Pour simplifier le réseau de neuronnes, il n'y aura qu'une seule fonction d'activation pour chaque couche, c'est-à-dire que tous les neuronnes d'une même couche auront la même fonction d'activation.
Cas limites : Nous avons une itération sur chaque neurone de chaque couche.
Cas erreurs : Si nous divisons par 0 lors du calcul de la fonction d'activation, cela peut causer une erreur. Avec la fonction sigmoïde, la formule étant $\forall x \in R, g(x) = \frac{1}{1+e^{-x}}$, il n'y aura pas de problème.

— *void calculerDelta(Layer L)* : cette fonction permet de calculer les Δ de chaque poids. Cela ce calcule grâce à la formule $\Delta w_i j = \eta \delta_j x_i$ avec $\delta_j = f'(I_j)(y_j - x_j)$ pour une sortie j ou $\delta_j = f'(I_j) \sum_k \delta_k w_{jk}$ pour un neurone caché j. Cette fonction enregistre également les résultats dans une matrice **delta**.
Cas limites : Nous avons une itération sur chaque neurone.
Cas erreurs : Si la couche L n'existe pas dans le réseau, il peut y avoir une erreur. De plus, pour la phase enregistrement des Δ dans la matrice **delta**, il peut y avoir des erreurs de dimension : il faut s'asssurer que la dimension de la matrice enregistrant les calculs soit correctement définie. Il peut également y avoir des problèmes d'ordre : il faut s'assurer que les calculs enregistrés soit correctement ordonnés et correspondent bien à chaque neurone de chaque couche cachée. Finalement, il faut s'assurer que la matrice **delta** soit bien initialisée.

— *void display_ weight(Matrix weight)* : cette fonction permet d'afficher la matrice des poids.
Cas limites : Nous pouvons avoir des matrices de poids avec une seule ligne (respectivement une seule colonne non nulle) quand on calcule les Deltas de la couche d'entrée (respectivement de la couche de sortie) par exemple.
Cas erreurs : Il faut vérifier la dimension de la matrice. De plus, il peut y avoir des erreurs de précisions : à cause de certains arrondis dû aux différents calculs que nous aurons pour les poids, il peut y avoir des erreurs d'arrondis lors de l'affichage des poids. Enfin, certains poids peuvent être égaux à 0. Il ne faudrait pas que la machine pensent qu'il s'agit d'une valeur manquante et donc que le poids affiché devra être bien égal à 0.

Tests unitaires dans Output_layer

— *void creer_ derniere_ couche(int input, int output, fonctionActivation f)* : Cette fonction permet de créer la couche de sortie. Des poids arrivent à cette couche mais ne "repartent" pas de cette dernière. Ainsi, il faut initialiser les poids et les biais de chaque neuronne de cette couche. Dans notre cas, la couche de sortie est composée de 10 neuronnes. Elle prend en entrée une fonction propagation : ici nous prendrons la fonction *softmax* qui permet une classification multiclasse.
Cas d'erreurs : Il faut que la dimension de la couche de sortie corresponde correctement au nombre de neuronnes que nous avons. Il faut qu'elle soit correctement initialisée avant d'être créée. De plus, la *TrainingFonction f* utilisée dans ce cas là est la fonction *softmax* donc si l'utilisate rentre une autre fonction, il y aura une erreur.

— *void fonction_ de_ perte()* : elle permet de calculer l'écart entre le résultat attendu et celui obtenu par le réseau de neurones . Dans notre cas le résultat obtenu va etre récupéré en s'aidant de la fonction enregistrer$_{v}ecteur_{d}e_{p}robabilite()$.$Ellevacorrespondreauchiffredo$
Cas d'erreurs :Sil'utilisateurprendunparamtreautrequel'entropiecroiseonpourraseretrouverd
— *void enregistrer_ vecteur_ de_ probabilite()* :

— *void obtenirResultat()* :

Tests unitaires dans Matrix

— *void affichage()* : cette procédure étant appelée sur un objet de la classe Matrix et ayant pour but d'afficher une matrice, nous pourrons ici tester l'affichage des matrices dans divers cas :
- une matrice carrée : avoir un retour à la ligne à chaque nouvelle ligne pour avoir un affichage d'une matrice carrée
- une matrice nulle : afficher que des 0
- une matrice diagonale : afficher correctement le fait qu'elle est diagonale
- une matrice vide : le programme informe l'utilisateur que la matrice est vide
- une matrice de taille un par un : le programme affiche seulement un réel à l'écran

— *Matrix creerGaussien(int nbRows, int nbColums)* : cette fonction permet de remplir une matrice de taille nbRows par nbColumns par des données obtenues grâce à une loi gaussienne :

- l'un des paramètres nuls en paramètre : le programme informe l'utilisateur que cela n'est pas possible
- les deux paramètres valant un : le programme renvoie une matrice correspondant à un réel avec nbRows = 1 et nbColumns = 1 et un tableau de deux dimensions n'ayant qu'une case
- une taille quelconque : les éléments de la matrice suivent bien une loi gaussienne connue du programmeur
- modifier les paramètres de la loi gaussienne : vérifier que les données obtenues correspondent bien à la nouvelle loi définie par le programmeur

— *Matrix creerUniforme(int nbRows, int nbColums)* : cette fonction permet de remplir une matrice de taille nbRows par nbColumns par des données obtenues grâce à une loi uniforme :

- l'un des paramètres nuls en paramètre : le programme informe l'utilisateur que cela n'est pas possible
- les deux paramètres valant un : le programme renvoie une matrice correspondant à un réel nbRows = 1 et nbColumns = 1 et un tableau de deux dimensions n'ayant qu'une case
- une taille quelconque : les éléments de la matrice suivent bien une loi uniforme connue du programmeur
- modifier les paramètres de la loi uniforme : vérifier que les données obtenues correspondent bien à la nouvelle loi définie par le programmeur

— *Matrix additionMatrix(Matrix A, Matrix B)* : cette fonction doit permettre de faire la somme de deux matrices et renvoyer le résultat dans une nouvelle matrice créée pour cela :

- somme de deux matrices carrées de taille différente : la fonction renvoie un message d'erreur à l'utilisateur l'informant que cela n'est pas possible
- la première matrice de la somme est nulle : la fonction renvoie une nouvelle matrice égale à la seconde matrice qui est non nulle
- la seconde matrice de la somme est nulle : la fonction renvoie une nouvelle matrice égale à la première matrice qui est non nulle
- les deux matrices sont de la même taille : la fonction renvoie une nouvelle matrice avec la bonne taille en paramètre
- somme de deux matrices de taille 1 : la fonction renvoie une matrice de taille 1 avec la bonne valeur
- somme de deux matrices opposés : la somme est censée être nulle, la fonction renvoie donc une nouvelle matrice dont tous les termes sont nuls
- somme de deux matrices dont l'une est négative : cela correspond à une différence de matrice, la fonction ne voit pas de problème et renvoie la nouvelle matrice
- vérifier que la somme est correcte dans des cas "normaux" : les calculs sont corrects

— *Matrix multiplicationMatrix(Matrix A, Matrix B)* : la fonction renvoie une nouvelle matrice égale au produit des deux matrices passées en paramètre :

- produit de deux matrices de taille empêchant le produit : la fonction renvoie un message d'erreur à l'utilisateur l'informant que cela n'est guère possible
- produit de deux matrices de taille nulle : la fonction renvoie un message d'erreur afin d'informe l'utilisateur que cela n'est pas possible
- le produit entre une matrice et un vecteur colonne : le programme renvoie un vecteur colonne
- le produit entre un vecteur ligne et une matrice : le programme renvoie un vecteur ligne
- la première matrice est nulle : la fonction renvoie une matrice nulle de la bonne taille
- la seconde matrice est nulle : la fonction renvoie une matrice nulle de la bonne taille
- l'une des deux matrices est l'identité : la fonction renvoie une nouvelle matrice égale à la matrice qui n'est pas l'identité
- produit d'une matrice avec son inverse : la fonction renvoie une nouvelle matrice correspondant à la matrice identité
- produit entre une matrice et elle-même : la fonction renvoie la matrice élevée au carré
- vérifier que le produit est correcte dans des cas "normaux" : les calculs sont corrects

— *Matrix transpose(Matrix A)* : la fonction renvoie la matrice transposée de celle passée en paramètre :

- la transposée d'une matrice de taille nulle : la fonction renvoie une erreur à l'utilisateur lui indiquant que cela n'est pas possible
- la transposée d'un vecteur colonne (d'un vecteur ligne) : la fonction renvoie un vecteur ligne(un vecteur colonne)
- la transposée d'une matrice de taille un : la fonction renvoie la même matrice

- la transposée d'une matrice symétrique : la fonction renvoie la même matrice
- vérifier que la transposée d'une matrice est correct : les lignes et les colonnes ont bien été "échangées"

— *Matrix cstfoisMatrix(double c, Matrix A)* : la fonction renvoie une nouvelle matrice obtenue en faisant la constante fois chaque terme de la matrice :

- la matrice est de taille nulle : la fonction renvoie un message d'erreur pour informer l'utilisateur du problème
- la constante est nulle : la fonction renvoie une matrice nulle de la bonne taille
- la matrice est de taille un : la fonction renvoie une matrice de taille un correspondant au produit entre la constante et l'élément de la matrice en paramètre
- la constante est égal à un : la fonction renvoie une matrice égale à celle passée en paramètre
- vérifier que le produit constante matrice est correct au niveau de chaque élément de la matrice

— *Matrix vecteurFoisVecteur(Matrix A, Matrix B)* : cette fonction s'occuper de faire le produit entre un vecteur colonne et un vecteur ligne ayant respectivement le même nombre de ligne et de colonne que l'autre afin d'avoir en sortie une matrice carrée de taille ce nombre de ligne ou de colonne :

- le vecteur colonne possède n lignes et le vecteur ligne ne possède pas n colonnes : la fonction renvoie une erreur indiquant à l'utilisateur que cette opération n'est pas correcte dans ce contexte
- les deux vecteurs sont vides : la fonction renvoie une erreur indiquant que cela n'est pas possible
- l'un des vecteurs est remplie de zéros : la matrice renvoyée par la fonction est la matrice nulle
- vérifier que le produit des deux vecteurs est bien correct : les éléments de la matrice obtenue sont les bons

— *double vecteurDot(Matrix A, Matrix B)* : cette fonction renvoie le produit scalaire entre deux vecteurs de même taille :

- les deux vecteurs ne font pas la même taille : la fonction indique à l'utilisateur que cela n'est pas possible
- les deux vecteurs sont vides : la fonction renvoie un message d'erreur afin d'informer l'utilisateur que cela n'est pas correct comme opération
- l'un des deux vecteurs est nul : la fonction renvoie zéro
- les deux vecteurs sont orthogonaux : la fonction renvoie zéro
- les deux vecteurs sont le même vecteur : la fonction renvoie la norme deux de ce vecteur au carré
- vérifier que le produit scalaire entre deux vecteurs est correct

— *int elementGrand(Matrix A)* : cette fonction renvoie l'indice du plus grand entier d'un vecteur passé en paramètre :

- le vecteur est vide : la fonction renvoie un message d'erreur prévenant l'utilisateur que cela n'est pas possible

- le vecteur ne contient qu'un seul élément : la fonction renvoie un comme indice peu importe la valeur de l'élément
- le vecteur comporte plusieurs fois la plus grande valeur : la fonction renvoie le premier indice auquel cette valeur est rencontrée (ou le dernier indice, c'est au choix lors de la conception)
- vérifier que la valeur renvoyée est bien la plus grande : si elle est négative ou même nulle

— *Matrix ajouter1Ligne(Matrix A)* : cette fonction renvoie une matrice dans laquelle une ligne a été ajouté au début de la matrice :

- la matrice est nulle : la fonction renvoie un vecteur ligne
- vérifier que l'ajout a bien été pris en compte et qu'aucune ligne n'a été perdu lors du processus

— *Matrix ajouter1Colonne(Matrix A)* : cette fonction renvoie une matrice dans laquelle une colonne a été ajouté au début de la matrice :

- la matrice est nulle : la fonction renvoie un vecteur colonne
- vérifier que l'ajout a bien été pris en compte et qu'aucune colonne n'a été perdu lors du processus

Tests unitaires dans Network

— *void training(Matrice TrainXSet,Vecteur TrainYSet, Double LearningRate, int TailleLot)* :La méthode training est utilisée pour entraîner le réseau de neurones. Nous entrons les données de l'ensemble d'apprentissage, l'étiquette de l'ensemble d'apprentissage, le taux d'apprentissage et la taille de lot comme paramètres. Parmi eux, taille de lot est utilisé pour spécifier la quantité de données utilisée pour mettre à jour le poids une fois. Après cette méthode, des poids raisonnables seront stockés dans chaque couche.
Cas d'erreurs : La taille de lot est un entier, et il est compris entre 1 et le nombre de lignes de la matrice de l'ensemble d'apprentissage. Si la valeur d'entrée est inférieure ou égale à 0 ou supérieure au nombre de lignes de la matrice de l'ensemble d'apprentissage, une erreur sera retourné. Le nombre de lignes de la matrice de l'ensemble d'apprentissage et les dimensions des étiquettes de l'ensemble d'apprentissage doivent être égaux, et une erreur sera renvoyée s'ils ne sont pas égaux.

— *void test(Matrice TestXSet, Vecteur TestYSet)* :La méthode de test est utilisée pour tester le réseau de neurones. Nous entrons les données de l'ensemble de test et l'étiquette de l'ensemble de test comme paramètres. Utilisez les matrices de poids stockées dans l'ensemble d'apprentissage pour effectuer une propagation vers l'avant du réseau de neurones sur les données de l'ensemble de test et stockez les résultats dans OutputLayer.
Cas d'erreurs : Le nombre de lignes de la matrice de l'ensemble de test et la dimension de le vecteur d'étiquettes de l'ensemble de test doivent être égaux, et une erreur sera renvoyée s'ils ne sont pas égaux.
La méthode ForwardPropagation est utilisée pour propager les données en avant, et l'entrée est un élément de données d'ensemble d'apprentissage ou d'ensemble de test, qui est un vecteur. Nous stockons ces données dans la première couche, puis nous utilisons la forme Wx+b pour la propagation vers l'avant dans chaque couche, où W est la matrice de poids, x est les valeurs de chaque neurone dans la couche précédente, et à chaque fois nous allons ajoutez une ligne de 1 à la matrice de poids, et obtenez enfin le résultat de Wx+b, et stockez-le dans le vecteur neurone de cette couche.

Cas d'erreurs : Le nombre de neurones dans la première couche est la dimension du vecteur de données d'entrée, et une erreur est renvoyée s'ils ne sont pas égaux.

— *void backpropagate()* : On analyse de son contexte d'abord, c'est à dire quelle information on possède déja quand on appelle cette méthode, et puis le fonctionnalité de cette fonction, à la fin les cas limites et les erreurs.
Contexte : nous avons le matrice delta pour la dernière couche de caché. et tous les matrices de activations pour chaque couche, et les dérives de fonction d'apprentissage pour chaque couche.
fonctionnalité : propager les delta vers avant par la formule suivante :

$$\frac{\partial L}{\partial h_j^{(l)}} = \delta_j^l = \begin{cases} (y_j - y_j^*) & \textit{pour la couche sortie} \\ f^{'}(h_j^l) \sum_k w_{j,k}^{l+1} \delta_k^{l+1} & \textit{pour chaque couche cachée} \end{cases}$$

les erreur : on a pas d'erreur de taille si les matrices activation sont calculé par propagation-forward.
la cas limite : on rétropropager zéro fois!!! donc il n' y a que une couche caché, dans cette cas il n' y pas le matrice de delta à calculer, donc le backpropagate() va faire rien.
Dans le test unitaire
on composer notre réseaux par que trois couche, deux couche caché et une couche sortie ; donc le boucle fait que une tour pour rétropropagation de delta, et on prends que une image de "1".
On prends le fonction d'activation tanh(x) pour deux couche dont dérivée de zéro égale un, et le matrice de poids pour deux couche caché égale zéro , après le propagation-avant les deux matrices de activation sont égale 0 aussi, et on prends $(\frac{9}{10}, \frac{-1}{10}, \frac{-1}{10}, \frac{-1}{10}, ...)^t$ pour les δ_j^2 à cause de le matrice de poids égale 0, on a $\delta_j^1 = 0$
on mock les fonction dérive est égale 1 et matrices fois matrices par 0 on regarde si le matrice de delta de premier couche égale 0, c'est à dire cohérent avec valeur nous attendons.

— *void updatePoids(double learing$_r$ate)* : *onsuivelemmedmarchequebackpropagation.*
Contexte : nous avons tous les matrice delta et tous les matrices de activations pour chaque couche.
fonctionnalité : calculer les incrément de matrice de poids et mettre à jour par la formule suivante : $\Delta w_i j = \eta \delta_j x_i$
les erreur : on a pas d'erreur de taille si les matrices activation sont calculé par propagation-forward et les matrices de delta calculer par propagation-backward.
la cas limite : on update zéro fois!!! donc il n' y a de couche caché, dans cette cas il n' y pas le matrice de poids à mise à jour, donc le updatePoids() va retourner message "cette modèle peut pas entrainner" et abort le programme.
Dans le test unitaire
tous d'aborde on teste le cas limite, on composer notre réseau par une couche sortie, et tester le méthode .
Dans le cas normale,on prends une image, donc matrices devient un vecteur on composer notre réseaux par que deux couches, une couche caché et une couche sorties ; donc le boucle fait que une tour pour update poids. et on prends matrice de activation de couche égale 1
On prends le matrice de poids pour couche caché égale 1 , pour δ_j^1, on prends $(0.1, 0.1, 0.1, ...)$.
À cause de le matrice de activation égale 1, on a le matrice de poids à jour tous coefficient égale 1.1.
on mock matrices fois matrices par matrice de 0.1, on regarde si le matrice de poids de premier couche égale 1.1, c'est à dire cohérent avec valeur nous attendons.

— *double calculfiabilité(int[] resultats$_e$xacte)*

Contexte : nous avons le résultats prévue dans le couch dérnier et nous avons le résultats exacte
fonctionalité : calculer le cohérent des cette vecteur .
les erreurs : deux vecteur résultats n'est pas le même taille, retourner message "imcomparabe" et aborter le programme.
les cas limites : deux vecteur taille 1(une image), si cohérent 100
le teste unitaire
on considère le erreur d'aborde, construite un réseaux, et mettre le résultats de couche sortie vecteur taille 3 et vecteur de résultats exacte égale taille 2, tester le méthode.
le cas limite on le considère pas (trivale)
le cas normale on choisie deux même vecteur et tester le méthode si la résultats est 100

5 Qu'est ce qu'un réseau de neurones ?

Les réseaux de neurones sont des modèles de traitement de l'information inspirés du fonctionnement du cerveau humain. Ils sont utilisés dans de nombreux domaines pour résoudre des problèmes complexes tels que la classification d'images, la reconnaissance de la parole, la prédiction de séries temporelles, la traduction automatique, la recommandation de produits, la détection de fraudes et bien d'autres encore.

Plus précisément, les réseaux de neurones servent à apprendre à partir de données d'entrée afin de produire des prédictions ou des classifications précises. Pour se faire, le réseau de neurones est entraîné à partir d'un ensemble de données d'entraînement, où il ajuste progressivement les poids et les biais de ses neurones jusqu'à ce que les prédictions produites soient aussi précises que possible.

De manière générale, voici les étapes du fonctionnement des réseaux de neurones :

— Les données d'entrée sont alimentées dans le réseau de neurones. Chaque neurone de la première couche est connecté à l'ensemble des données d'entrée.

— Chaque neurone de la première couche effectue une opération mathématique sur les données d'entrée qui lui sont connectées. Cette opération implique la multiplication des données d'entrée x_i par des poids, notés w_i, qui sont initialement définis de manière aléatoire, puis l'addition d'un biais w_0, qui permet de réguler l'activation du neurone. La somme pondérée des entrées, aussi appelée activation, se calcule de la manière suivante :
$y = \overrightarrow{w}.\overrightarrow{x} = (\sum_{i=1}^{n} w_i x_i) - w_0$

— Les sorties des neurones de la première couche sont alors alimentées à la deuxième couche, et ainsi de suite, jusqu'à atteindre la dernière couche. De façon générale, une fonction de transfert, aussi appelée fonction d'activation, que nous noterons g, permet de relier l'activation au signal de sortie. Les plus couramment utilisées sont la fonction de Heaviside ($\forall x \in R, g(x) = 1$ si $x \geq 0, 0$ sinon) et la fonction sigmoïde ($\forall x \in R, g(x) = \frac{1}{1+e^{-x}}$). Notons que les fonctions d'activations sont croissantes.
La sortie d'un neurone s'exprime avec la formule suivante : $g((\sum_{i=0}^{k} w_i x_i) - w_0)$ avec g.

— La dernière couche du réseau de neurones produit la sortie du modèle, qui peut être une classification, une prédiction, une génération de données, etc.

— En cours d'apprentissage, le réseau de neurones ajuste les poids et les biais de ses neurones pour minimiser une fonction de coût, qui mesure l'écart entre la sortie du modèle et la sortie attendue. Ce processus est appelé la rétropropagation de l'erreur, et il permet au réseau de neurones d'apprendre à partir des exemples d'entraînement.

— Une fois que le réseau de neurones a été entraîné, il peut être utilisé pour produire des prédictions sur de nouvelles données qui n'ont pas été vues lors de l'entraînement.

Pour savoir quand le neurone est actif ou non, il faut regarder ce que nous appelons le seuil d'activation. Un neurone est actif quand le seuil est atteint ou dépassé. Le seul est atteint quand l'entrée vaut 0. C'est à dire quand $(\sum_{i=0}^{k} w_i x_i) - w_0 \geq 0$ donc quand $\sum_{i=0}^{k} w_i x_i \geq w_0$.

Les réseaux de neurones sont particulièrement efficaces pour traiter des données non linéaires ou hautement complexes, ce qui en fait une technique d'apprentissage automatique puissante pour résoudre des problèmes qui ne peuvent pas être facilement résolus avec des méthodes traditionnelles de programmation.

Il s'agit d'un type particulier d'algorithmes d'apprentissage automatique (comme les machines à vecteur de support, arbres de décision, K plus proches voisins, etc.) caractérisés par un grand nombre de couches de neurones, dont les coefficients de pondération sont ajustés au cours d'une phase d'entraînement (apprentissage profond).

Il existe plusieurs types de réseaux de neurones, chacun étant conçu pour répondre à des problématiques spécifiques. Nous en présenterons quelques uns plus en détail dans ce qui va suivre. Ces différents types de réseaux de neurones peuvent être combinés pour créer des modèles plus complexes et plus performants pour des tâches spécifiques. Ils peuvent être également sujets à des problèmes tels que le surapprentissage et le biais algorithmique, ce qui nécessite une attention particulière lors de leur utilisation.

5.1 Réseaux de neurones à propagation avant (feedforward)

Il s'agit d'un réseau de neurones artificiels acyclique. Il n'y a donc pas de cycles ou de boucles dans le réseaux. Les informations ne se déplacent que vers l'avant, dans une seule direction. Elles partent des noeuds d'entrées et se dirigent vers les noeuds de sorties. Elles peuvent passer par des couches cachées.

Il s'agit d'un réseau de neurone monocouche, ou perceptron, feed-forward, notamment utilisé dans le domaine des statistiques.

De façon théorique, un réseau de neurones monocouche se caractèrise de la façon suivante : nous y retrouvons n entrées et p sorties qui sont généralement alignés de façon verticale. Chaque p neurone est connecté aux n entrées. Ce type de perceptron n'alimente pas ses entrées avec ses sorties, se distinguant ainsi des perceptrons récurrents.

Chaque neurone est connecté à tous les neurones de la couche précédente. Chaque neurone d'entrée se voit attribuer une valeur numérique. Sur la première couche, celle qui suit la couche d'entrée, chaque neurone va calculer sa valeur en utilisant celles de tous les neurones de la couche d'entrée. Les neurones de la deuxième couche utiliseront les valeurs ainsi obtenues à la première couche pour calculer leurs valeurs. Ce procéssus se répète sur toutes les autres couches du système. Ce type de réseau de neurones se distingue des réseaux de neurones récurrents qui contiennent au moins un cycle. Dans ce genre de réseaux, les neurones interagissent donc non-linéairement.

Dans les réseaux feedforward on retrouve des fonctions de transfert linéaires.

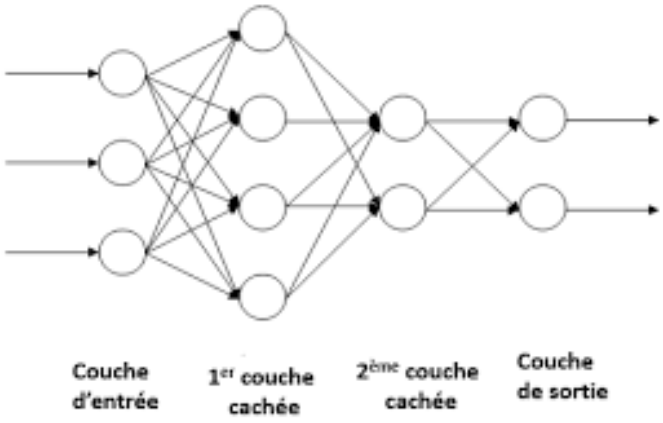


FIGURE 9 – Exemple de réseau de neurones *feed-forward* multicouches

C'est le type le plus simple de réseau de neurones, où les informations circulent dans une seule direction, de l'entrée à la sortie, sans qu'il y ait de boucle de rétroaction.

5.2 Réseaux de neurones récurrents (recurrent neural networks - RNN)

Un réseau de neurones récurrent (RNN) est un type de réseau de neurones qui prend en compte les dépendances séquentielles dans les données. Contrairement aux réseaux de neurones feedforward, les RNN peuvent prendre en compte les entrées précédentes pour influencer les sorties actuelles.
Le concept clé derrière les RNN est l'utilisation d'un état caché qui est mis à jour à chaque pas de temps en fonction de l'entrée actuelle et de l'état caché précédent. Cette mise à jour peut être décrite par une équation de récurrence.
Les RNN sont couramment utilisés pour la modélisation de séquences telles que le traitement du langage naturel et la reconnaissance de la parole, ainsi que pour la prédiction de séries chronologiques.
Bien qu'ils sont très pratiques comparé à une architecture de réseau de neurones classique pour le traitement des données séquentielles, il s'avère qu'ils sont extrêmement difficiles à entraîner pour gérer la dépendance à long terme en raison du problème de la disparition du gradient (Gradient Vanishing). Une explosion de gradient peut également se produire mais très rarement. Pour surmonter ces lacunes, de nouvelles variantes RNN ont été introduites comme le Long short term memory(LSTM) ou réseau récurrent à mémoire court et long terme et le Gated Recurrent Unit (GRU) ou réseau récurrent à portes.

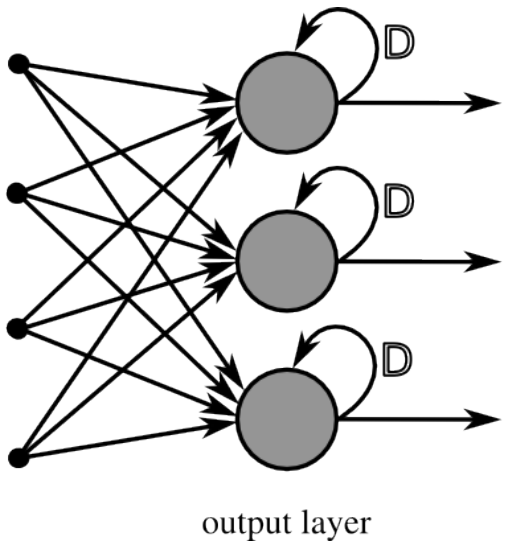


FIGURE 10 – Auto-encodeurs

Le LSTM

Le réseau récurrent à mémoire court et long terme est l'architecture de réseau de neurones récurrents la plus utilisée en pratique qui permet de répondre au problème de disparition de gradient. Il a été proposé par Sepp Hochreiter et Jürgen Schmidhuber en 1976. L'idée d'un LSTM est que chaque unité computationnelle est liée non seulement à un état caché h mais également à un état c de la cellule qui joue le rôle de mémoire. Le passage de c_{t-1} à c_t se fait par transfert à gain constant et égal à 1. De cette façon les erreurs se propagent aux pas antérieurs (jusqu'à 1 000 étapes dans le passé) sans phénomène de disparition de gradient. L'état de la cellule peut être modifié à travers une porte qui autorise ou bloque la mise à jour (input gate). De même une porte contrôle si l'état de cellule est communiqué en sortie de l'unité LSTM (output gate). La version la plus répandue des LSTM utilise aussi une porte permettant la remise à zéro de l'état de la cellule (forget gate).

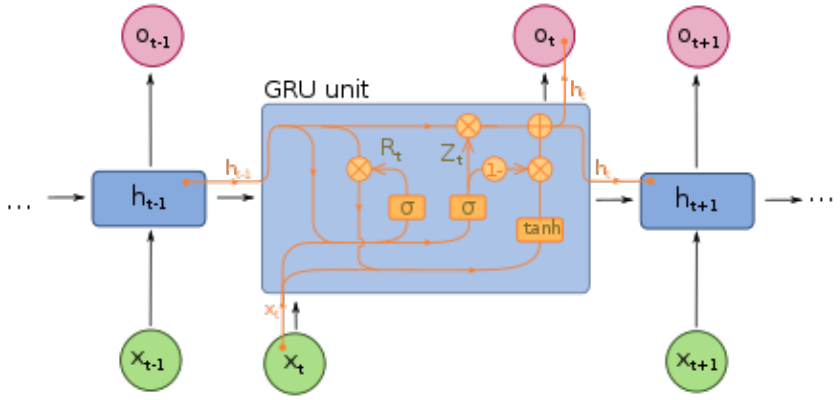


FIGURE 11 – Auto-encodeurs

Le GRU

Un réseau de neurones récurrents à portes, est une variante des LSTM introduite en 2014 par Kyunghyun Cho et Al. Les réseaux GRU ont des performances comparables aux LSTM pour la prédiction de séries temporelles (ex : partitions musicales, données de parole). Une unité requiert moins de paramètres à apprendre qu'une unité LSTM. Un neurone n'est associé plus qu'à un état caché (plus de cell state) et les portes d'entrée et d'oubli de l'état caché sont fusionnées (update gate). La porte de sortie est remplacée par une porte de réinitialisation (reset gate).

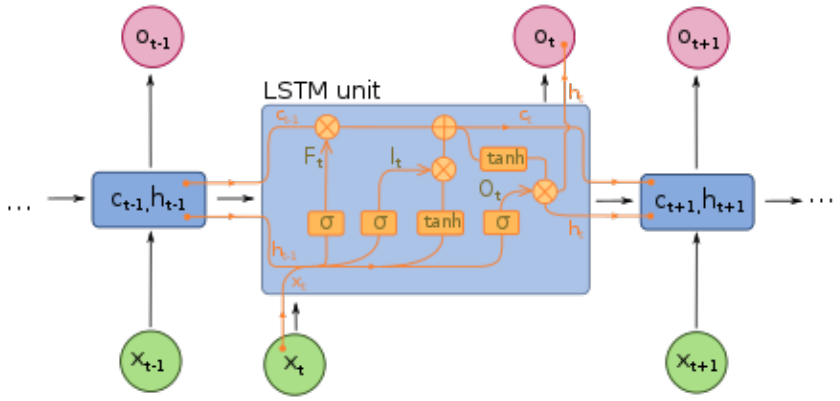


FIGURE 12 – Auto-encodeurs

Dépliement d'un réseau de neurones récurrent

Qu'elle soit détaillée ou simplifiée, la représentation d'un réseau récurrent n'est pas aisée, car il est difficile de faire apparaître la dimension temporelle sur le schéma. C'est notamment le cas pour les connexions récurrentes, qui utilisent l'information du temps précédent.
Pour solutionner ce problème, on utilise souvent une représentation du réseau "déplié dans le temps", afin de faire apparaître explicitement celui-ci.

5.3 Auto-encodeurs et auto-encodeurs varitationnels

5.3.1 Auto-encodeurs

Un réseau de neurones auto-encodeurs est un type de réseau de neurones artificiels qui apprend à représenter des données d'entrée de manière comprimée en utilisant une structure d'encodage-décodage.

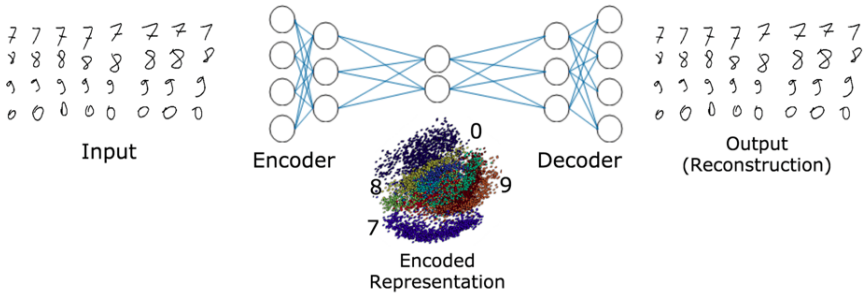


FIGURE 13 – Auto-encodeurs

Un auto-encodeur peut être représenté comme une fonction f qui prend en entrée un vecteur de données $\mathbf{x}$ et renvoie un vecteur de sortie $\mathbf{y}$ de même dimension. Il possède une structure interne (bottleneck - espace latent) implémentée par les couches cachées (le nombre de neurones dans ces couches est inférieur aux nombres de neurones des couches d'entrée/sortie). L'objectif de l'auto-encodeur est de minimiser la différence entre l'entrée et la sortie, également appelée erreur de reconstruction.
Un réseau d'auto-encodeurs est composé de plusieurs couches d'encodeurs et de décodeurs, où chaque couche représente une transformation non linéaire de la couche précédente. Les encodeurs réduisent la dimensionnalité de l'entrée tandis que les décodeurs la restaurent à sa dimensionnalité d'origine. Un auto-encodeur s'entraîne à extraire les parties les plus importantes d'une entrée afin de générer une sortie qui présente moins de descripteurs. En d'autres termes, le réseau ignore le bruit, généralement pour réduire la dimensionnalité de l'entrée. Le réseau entier est formé en minimisant la somme des erreurs de reconstruction pour toutes les couches.
Les auto-encodeurs peuvent être utilisés pour des tâches telles que la réduction de dimensionnalité, la compression de données, la détection d'anomalies et la génération de données. Ils ont également été utilisés avec succès pour la préparation de données en vue d'une utilisation ultérieure dans des modèles d'apprentissage en profondeur plus avancés. A la différence d'un grand nombre de réseaux de neurones, les auto-encodeurs peuvent être entraînés de manière non-supervisée, ce qui permet d'appliquer ces méthodes à des jeux de données non annotés.

5.3.2 Auto-encodeurs varitationnels

Un auto-encodeur variationnel (VAE) est un type particulier de réseau d'auto-encodeurs qui ajoute une contrainte de régularisation pour générer des représentations latentes qui suivent une distribution spécifique, généralement une distribution normale multivariée.
La principale utilisation d'un VAE est la génération de données. En effet, en apprenant une représentation latente qui suit une distribution connue, le VAE peut être utilisé pour générer de nouvelles données en échantillonnant à partir de cette distribution. Ces données générées peuvent être utilisées dans diverses applications telles que la synthèse de données pour l'entraînement de modèles de deep learning, la génération de contenu

pour des applications de réalité virtuelle, ou même la création de nouvelles œuvres d'art numériques.
Un autre avantage des VAE est qu'ils permettent également d'interpoler entre des exemples existants dans l'espace latent, ce qui permet de générer des données intermédiaires qui ne sont pas nécessairement présentes dans l'ensemble de données d'origine. Cette fonctionnalité peut être utilisée pour la création de nouvelles images ou de nouveaux sons en combinant des caractéristiques de plusieurs exemples existants.
En résumé, les VAEs sont utilisés pour générer de nouvelles données à partir d'une distribution latente connue, ce qui peut être utile dans de nombreuses applications, notamment la synthèse de données pour l'entraînement de modèles de deep learning, la création de contenu pour la réalité virtuelle et la création artistique.

5.4 Réseaux de neurones convolutifs (convolutional neural networks - CNN)

Ces réseaux sont particulièrement adaptés à la classification d'images, où les informations visuelles sont représentées sous forme de matrices. Les réseaux de neurones convolutifs (Convolutional Neural Networks ou CNN) sont des réseaux de neurones artificiels qui sont particulièrement efficaces pour traiter des données de haute dimensionnalité telles que des images ou des vidéos. Les CNN sont inspirés de la façon dont le cortex visuel du cerveau traite les informations visuelles.
Voici les principales étapes du fonctionnement d'un CNN :
Convolution : La première couche d'un CNN est une couche de convolution, qui applique des filtres (également appelés noyaux ou kernels) sur l'image d'entrée. Chaque filtre extrait une caractéristique spécifique de l'image, telle que les bords, les textures ou les formes. La sortie de la couche de convolution est une carte de caractéristiques.
Fonction d'activation : Après chaque couche de convolution, une fonction d'activation est appliquée à la sortie de la couche pour introduire de la non-linéarité dans le modèle. La fonction d'activation la plus couramment utilisée dans les CNN est la fonction ReLU (Rectified Linear Unit).
Pooling : La couche de pooling réduit la dimensionnalité de la carte de caractéristiques en appliquant une opération de sous-échantillonnage (par exemple, le max-pooling). Cette opération extrait les valeurs les plus importantes de chaque zone de la carte de caractéristiques.
Couches fully connected : Les couches fully connected (ou couches denses) sont similaires à celles des réseaux de neurones traditionnels. Elles relient les sorties de la couche précédente à une sortie finale qui est une prédiction pour la tâche donnée, telle que la classification d'images.
Rétropropagation : Après l'évaluation de la sortie, le modèle utilise la rétropropagation pour ajuster les poids des filtres et des connexions pour minimiser l'erreur de prédiction. Ces étapes sont généralement répétées plusieurs fois dans le réseau pour améliorer les performances du modèle. Les CNN ont été utilisés avec succès dans de nombreuses applications, telles que la reconnaissance d'images, la classification d'objets, la détection d'objets et la segmentation sémantique.
Les réseaux de neurones convolutifs sont spécialement conçus pour travailler sur des données d'images et de vidéos. Ils utilisent des couches de convolution pour extraire des caractéristiques à partir de l'image, suivies de couches de pooling pour réduire la dimensionnalité dle pourcentage d'erreur es caractéristiques extraites. Les réseaux de neurones convolutifs sont également capables d'apprendre des hiérarchies de caractéristiques à partir des images, ce qui leur permet de reconnaître des motifs complexes tels que des formes, des textures et des contours.
La principale différence entre les réseaux de neurones convolutifs et les réseaux de neurones généraux réside dans leur capacité à travailler avec des données complexes telles que des images et des vidéos. Les réseaux de neurones convolutifs sont mieux adaptés à ce type de données grâce à leur capacité à extraire des caractéristiques et à apprendre des hiérarchies de caractéristiques à partir de ces données.

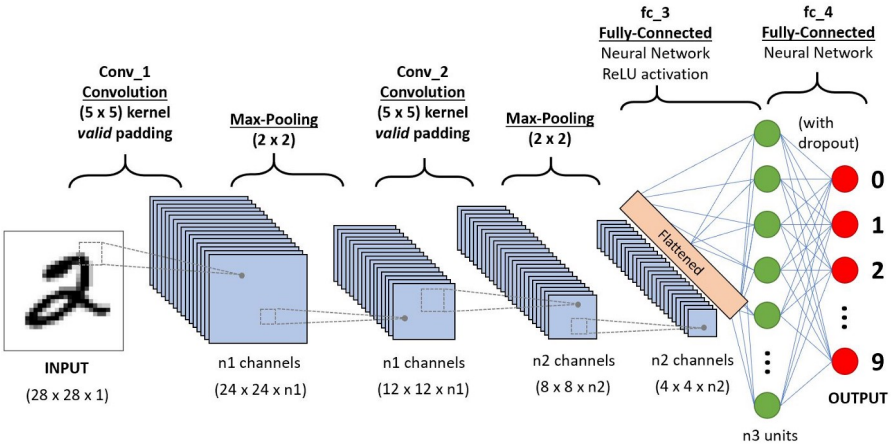


FIGURE 14 – Réseaux de neurones convolutifs

5.5 Réseaux antagonistes génératifs (generative neural networks)

Les réseaux antagonistes génératifs (en anglais, generative adversarial networks ou GAN) sont des modèles de réseaux de neurones non supervisés qui fonctionnent en mettant en concurrence deux modèles de réseaux de neurones. Par example, dans le cas du jeu de données en dessous, le premier réseau (appelé le générateur) générerait une image de chiffre. Le second, le discriminateur, reçoit la sortie du générateur ou un exemple d'un véritable image de chiffre. Sa sortie est sa propre estimation de la probabilité que l'entrée qui lui a été fournie provienne d'un exemple créé par le générateur. La décision du discriminateur agit alors comme un signal d'erreur adressé aux deux modèle mais dans des "directions opposées", en ce sens que s'il est certain que sa décision est correcte, cela implique une erreur importante pour le générateur (pour ne pas avoir réussi à tromper le discriminateur), tandis que si le discriminateur a été largement trompé, la perte importante est pour le discriminateur.

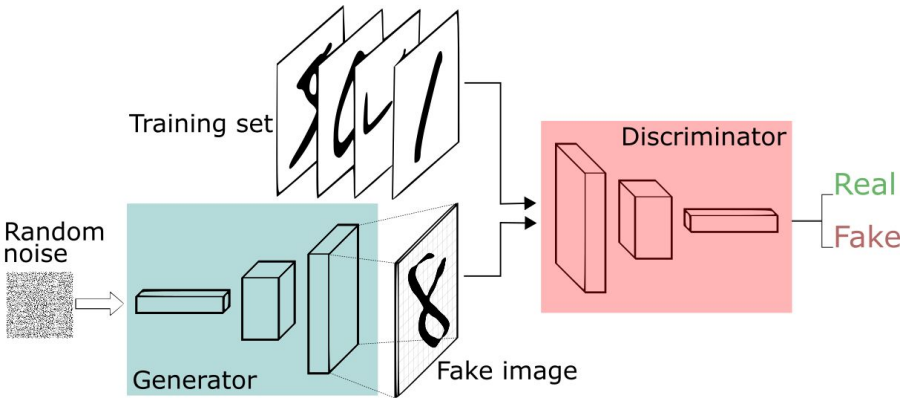


FIGURE 15 – GAN

L'approche d'apprentissage du GAN
Un générateur a été défini pour générer des chiffres manuscrits, un discriminateur pour déterminer si les chiffres manuscrits sont réels ou non, et un jeu de données de chiffres manuscrits réels. Alors comment faire l'entraînement ?
À propos des générateurs
Pour le générateur, l'entrée doit être un vecteur à n dimensions et la sortie une image de la taille du pixel de l'image exemple. Nous devons donc d'abord obtenir le vecteur d'entrée.
Conseils : Le générateur peut être n'importe quel modèle capable de produire des images, comme le réseau neuronal entièrement connecté le plus simple, ou un réseau déconvolutionnel.
Le vecteur d'entrée est considéré ici comme porteur de certaines informations sur la sortie, comme le nombre de chiffres dans l'écriture, le degré de gribouillage de l'écriture, etc. Nous n'avons pas besoin d'informations spécifiques sur les chiffres de sortie, mais seulement qu'ils ressemblent le plus possible aux chiffres manuscrits réels (pour tromper le discriminateur). Nous utilisons donc un vecteur généré aléatoirement comme entrée, où l'entrée aléatoire devrait idéalement satisfaire les distributions courantes telles que la distribution moyenne, la distribution gaussienne, etc.
Astuce : Si nous devons obtenir des nombres de sortie spécifiques par la suite, nous pouvons analyser la sortie générée par le vecteur d'entrée et avoir une idée des dimensions utilisées pour contrôler la numérotation des nombres, c'est-à-dire la sortie spécifique. Souvent, cela n'est pas précisé avant la formation.
À propos des discriminateurs

Il va sans dire que les discriminateurs sont souvent des discriminateurs communs, où l'entrée est l'image et la sortie est l'étiquette d'authenticité de l'image.
Conseils : De même, les discriminateurs, comme les générateurs, peuvent être n'importe quel modèle de discriminateur, comme un réseau entièrement connecté, ou un réseau contenant une convolution, etc.

5.6 Réseau de l'État d'Écho (Echo state networks - ESN)

5.6.1 Introduction

Le réseau d'état d'écho, un nouveau type de réseau neuronal récurrent (comme illustré ci-dessus), se compose également d'une couche d'entrée, d'une couche cachée (c'est-à-dire une réserve) et d'une couche de sortie. Le processus de formation ESN est le processus de formation des poids de connexion (Wout) de la couche cachée à la couche de sortie. Trois caractéristiques sont résumées comme suit :
(1) La structure de base est une réserve constante et générée aléatoirement .
(2) Les poids de sortie sont la seule partie qui doit être ajustée.
(3) Une simple régression linéaire peut être utilisée pour entraîner le réseau.

5.6.2 Equation

A chaque instant de l'entrée u(t), la réserve met à jour son état, et son équation de mise à jour d'état est la suivante : $x(t+1) = f(W_{in} * u(t+1) + W_{back} * x(t))$
L'équation d'état de sortie de l'ESN est la suivante : $y(t+1) = f_{out} * (W_{out} * (u(t+1), x(t+1)))$
Le processus d'apprentissage de l'ESN est le processus de détermination de la matrice de poids de connexion de sortie du coefficient W_{out} sur la base des échantillons d'apprentissage donnés. L'apprentissage est divisé en deux phases : la phase d'échantillonnage et la phase de calcul des poids.
Pour simplifier, on suppose que W_{back} est égal à 0 et que les poids de connexion entrée-sortie et sortie-sortie sont également égaux à 0.

5.6.3 Phase d'échantillonnage

La phase d'échantillonnage commence par une sélection arbitraire de l'état initial du réseau, mais généralement l'état initial du réseau est choisi comme étant 0, c'est-à-dire x(0) = 0.
(1) Les échantillons d'apprentissage (u(t), t=1,2,.... ,P) sont ajoutés à la réserve après la matrice de poids des connexions d'entrée Win.
(2) Le calcul et la collecte de l'état du système et de la sortie y(t) sont effectués tour à tour selon les deux équations d'état précédentes.
Afin de calculer la matrice des droits de connexion de sortie, les variables d'état internes doivent être collectées (échantillonnées) à partir d'un certain moment m et former une matrice B(P-m+1, N) avec des vecteurs comme lignes, tandis que les données d'échantillon correspondantes y(t) sont également collectées et forment un vecteur colonne T(P-m+1, 1).

5.6.4 La phase de calcul des poids

La phase de calcul des poids consiste à calculer les poids de connexion de sortie Wout sur la base de la matrice d'état du système et des données collectées pendant la phase d'échantillonnage. Comme la relation entre la variable d'état x(t) et la sortie prédite est linéaire, l'objectif à atteindre est d'approcher la sortie désirée y(t), en utilisant la sortie prédite.

6 Détails d'implémentation

Dans le cadre de ce projet, nous devions implémenter un réseau de neurone en utilisant le langage C++, pour ce faire nous avons codé les différentes classes présentes dans le diagramme de classe en mettant dans le dossier include tous les fichiers .hpp, dans le dossier src tous les fichiers .cpp et dans le fichier obj, tous les fichiers .o créé suites à l'execution de notre makefile. Nous allons maintenant donner une explication plus détaillée des méthodes les plus importantes de nos differentes classes.

6.1 Classe Matrix

Cette classe est essentielle puisqu'elle représente les différentes liaisons entre les neuronnes.

Dans cette classe, nous retrouvons les différentes méthodes permettant de faire des opérations entre matrices. C'est pourquoi sont surchargés les opérateurs $=$, $+$, $-$ et $*$, ce dernier permettant la multiplication entre deux matrices.
De plus, certaines méthodes sont implémentées pour générer des lois pour les poids :

— la méthode *void creerGaussien(double mu, double stddev)* permet de créer une matrice de valeurs aléatoires suivant une distribution gaussienne.
mu correspond à la moyenne de la loi normale et **stddev** correspond à l'écart type.
Le générateur de nombres aléatoires est initialisée à l'heure actuelle, **time(0)** pour s'assurer que les nombres aléatoires générés sont différents à chaque exécution du programme.
Finalement, à chaque itération de la boucle **for**, un nombre aléatoire est généré à l'aide de la distribution gaussienne et du générateur de nombres aléatoires. Ce nombre aléatoire est ensuite assigné à l'élément correspondant de la matrice **weight[i][j]**.
— la méthode *void creerUniforme(double a, double b)* crée une matrice de valeurs aléatoires suivant une distribution gaussienne avec **a** la borne inférieure de la distribution uniforme et **b** la borne supérieure de la loi uniforme.

Nous retrouvons également des méthodes pour la transposition de la matrice, de la multiplication d'une matrice avec une constance, une multiplication d'un vecteur avec une matrice, produit de vecteurs pour avoir un réel,...

Des méthodes d'ajout de colonne vide ou de ligne vide sont également présentes.

Nous retrouvons des méthodes permettant de calculer la somme des éléments d'une matrice et finalement une méthode pour appliquer une fonction sur chaque élément de la matrice.

6.2 Classe TrainingFonction

Dans cette classe, nous retrouvons les différentes méthodes qui implémentent les fonctions utiles à notre réseau : les fonctions d'activations.

Nous retrouvons par exemple la méthode *softmax* qui est implémenté de la sorte :

```
Matrix TrainingFunction::softmax(const Matrix & x)
{

    Matrix y = x;
    double sum = 0.0;
    int c;
    for(int j= 0;j<y.nbColumns;j++){
    for (int i = 0; i < y.nbRows; i++)
    {
        y.weight[i][j] = exp(y.weight[i][j]);

        sum += y.weight[i][j];
    }
    for (int i = 0; i < y.nbRows; i++)
    {
        c= y.weight[i][j]/ sum;
    }
    }
    return y;
}
```

FIGURE 16 – méthode softmax de la classe TrainingFunction

Cette fonction s'est avérée utile dans notre projet car c'est une fonction mathématique utilisée dans les domaines de l'apprentissage automatique par exemple. Elle est utilisée dans l'implémentation de réseaux de neuronnes pour générer des probabilités pour des problèmes de classification multiclasse, où il faut attribuer une probabilité à chaque classe possible en fonction des caractéristiques d'entrées.
Dans cette classe, nous retrouvons également l'implémentation de la fonction sigmoïde, fonction que nous retrouvons classiquement dans les réseaux de neuronnes.

6.3 Classe Layer

La classe layer est celle qui permet de créer une couche dans le réseau de neurone. Pour ce faire, mis à part les getters, les setters et les constructeurs nous avons la méthode displayWeight() qui permet d'afficher la matrice de poids, la méthode Sortie() qui nous permet permet de calculer la sortie de la couche, la méthode activation(Layer dernier) qui prend en paramètre une couche sur laquelle on applique la fonction d'activation et qui permet d'appliquer la fonction d'activation à tous les éléments du vecteur colonne et CalculerDelta() qui permet de calculer les Δw_{ij} qui vont servir à modifier les poids lors de la rétropropagation. Dans la methode activation on applique d'abord la fonction d'activation à toutes les entrées, après l'avoir sauvegardé dans le vecteur I, on ajoute une ligne supplémentaire à la matrice des entrées.

```
void Layer::activation(Layer& dernier)
{
        dernier.Sortie();
        dernier.delta = I;
        entree = dernier.I.apply_function(dernier.Fonction);
        entree.ajouterLigneUn();

}
```

FIGURE 17 – Application de la fonction d'activation

CalculerDelta() permet de calculer les Δw_{ij} afin de corriger la valeur des poids lors de la retropropagation. Pour ce faire, on utilise la formule suivante : $\Delta w_{ij} = \eta \delta_j x_i$. Sachant que les x_i sont la sortie du neurone i et les $\delta_j = F'(I_j) \sum_k \delta_k w_{kj}$. C'est ce travail que l'on fait successivement dans les 3 boucles.

```
void Layer::calculerDelta(Layer& L)
{
        double Ij;
        double dj;
        double s;
        double n = 1;

        for (int j = 0; j < L.delta.nbColumns; j++)
        {
                for (int i = 0; i < arete.nbColumns; i++)
                {
                        delta.weight[i][j] =0;
                        for(int k =0 ; k< arete.nbRows;k++)
                                delta.weight[i][j] += L.delta.weight[k][j] * arete.weight[k][i];
                        delta.weight[i][j] = delta.weight[i][j]*(*deriveFonction)(I.weight[i][j]);
                }


        }
}
```

FIGURE 18 – Calcul des delta

6.4 Classe Output_layer

Cette classe permet d'implémenter la couche de sortie de notre réseau.

```
typedef Matrix& (*gen)(const Matrix&);
```

FIGURE 19 – Création d'un atlas de type

Nous retrouvons cet alias de type appelé **gen** pour un pointeur de fonction qui prend une référence constante vers un objet Matrix en tant que paramètre et renvoie une référence mutable vers un objet Matrix. Ce type de pointeur de fonction est utilisé pour définir des fonctions génériques qui prennent une Matrix en entrée et renvoient une nouvelle Matrix modifiée. Pour pouvoir tester le réseau il est important de télécharger OpenCV et boost pour c++.

6.5 Classe Network

Commençons par détailler ce que fait le constructeur de cette classe. Il est appelée lors de la création d'une instance de la classe Network et est chargée d'initialiser les couches et la couche de sortie du réseau neuronal. Le constructeur prend trois paramètres : inputDim (la dimension de la couche d'entrée du réseau neuronal), hidDim (un vecteur contenant les dimensions des couches cachées du réseau neuronal) et outputDim (la dimension de la couche de sortie du réseau neuronal).

Il est ensuite important de noter qu'une méthode centrale de cette classe est la méthode train. Cette méthode est utilisée pour entraîner le réseau de neurone. Elle prend en paramètres les données d'entraînement (TrainXset et TrainYset), le taux d'apprentissage (learningRate), la taille des lots (batchsize), et le nombre d'époques (epochs). Elle effectue la phase d'entraînement en utilisant les méthodes forward_propagation, backward_propagation, et update_weight pour ajuster les poids du réseau.

La fonction forward_propagation propage les données d'entrée à travers les différentes couches du réseau neuronal, en activant chaque couche à partir de la deuxième couche jusqu'à la dernière. La dernière couche effectue des opérations spécifiques à la sortie, et

enfin, la sortie est obtenue en appelant la méthode sortie de la couche finale.

La méthode backward_propagation calcule les deltas (erreurs) de chaque couche du réseau neuronal en partant de la couche de sortie et en remontant jusqu'à la première couche cachée. Cela permet de quantifier l'erreur commise par chaque neurone et d'utiliser ces informations pour ajuster les poids lors de la mise à jour ultérieure du réseau. Ci-dessous une rapide explication de cette méthode :

— Matrix delta_L = this->fcouche.getProba() - TrainYset ; : Cette ligne calcule la différence entre les probabilités prédites par la couche de sortie (fcouche.getProba()) et les vraies valeurs d'entraînement (TrainYset). Cela donne le delta (erreur) de la couche finale.
— fcouche.setDeltas(delta_L) : Cette ligne appelle la méthode setDeltas de la couche de sortie (fcouche) pour mettre à jour les deltas de cette couche avec le delta calculé précédemment.
— for(int i = this->couches.size() - 1 ; i >= 0 ; i–) : Cette boucle parcourt les couches cachées du réseau neuronal en commençant par la dernière couche et en remontant jusqu'à la première couche.
— couches[i].calculerDelta(couches[i+1]) : Pour chaque couche cachée (couches[i]), la méthode calculerDelta est appelée en lui passant la couche suivante (couches[i+1]). Cette méthode calcule le delta de la couche actuelle en utilisant les deltas de la couche suivante et les poids entre les deux couches.

Ensuite, la méthode update_weight est responsable de la mise à jour des poids du réseau neuronal à l'aide de la descente de gradient par lot. Elle calcule les variations des poids en fonction des deltas des couches et des taux d'apprentissage, puis les applique pour mettre à jour les poids de chaque couche du réseau. Cette étape est cruciale pour l'apprentissage du réseau neuronal, car elle permet d'ajuster les poids afin de minimiser l'erreur du modèle pendant la phase d'entraînement.

La méthode test sera quant-à-elle utilisée pour tester le réseau sur des données de test. Elle prend en paramètres les données de test (TestXset et TestYset) ainsi que la taille des lots (batchsize). Elle retourne une mesure de performance du réseau neuronal, par exemple, l'exactitude (accuracy) ou l'erreur moyenne.

6.6 Classe Datasource

Cette classe permet de charger les images du jeu de données MNIST, les convertit en matrices de pixels normalisés et les associe à leurs étiquettes correspondantes, stockées dans Xtrain et Ytrain.

La fonction prend en paramètres un chemin vers le répertoire contenant les images MNIST, ainsi que des matrices Xtrain et Ytrain pour stocker les données d'entraînement. Puis, elle boucle sur chaque classe d'étiquettes (de 0 à 9). Pour chaque classe, elle utilise la fonction cv : :glob pour récupérer les noms de fichiers correspondant à cette classe dans le répertoire spécifié. Ensuite, elle boucle sur un nombre spécifié d'images par classe (déterminé par nr_images / LABELS). Pour chaque image, elle lit l'image à l'aide de cv : :imread et la stocke dans une matrice cv : :Mat appelée img. Ensuite, elle crée une nouvelle instance de la classe Matrix pour stocker l'image (de taille height * width). Elle parcourt les pixels de l'image et les convertit en valeurs entre 0 et 1 en les divisant par

255.0. Ces valeurs sont ensuite stockées dans la matrice image. La matrice image est ensuite ajoutée à la matrice Xtrain en utilisant emplace_back. Elle crée également une matrice vr de taille LABELS pour représenter l'étiquette correspondante à l'image. La valeur correspondant à la classe de l'image est définie à 1.0 et les autres valeurs sont à 0.0. La matrice vr est ajoutée à la matrice Ytrain en utilisant emplace_back. Une fois toutes les images traitées pour toutes les classes, la fonction renvoie 0.

7 Conclusion

Ce projet nous a tout d'abord permis d'apprendre à travailler en groupe de huit personnes, ce qui a été une expérience enrichissante. En effet, nous avons appris l'importance d'une communication efficace. En se partageant nos connaissances et nos idées, nous avons pu surpasser les différentes difficultés rencontrées. De plus, le court délai pour réaliser ce projet nous a permis de nous mettre dans la peau d'un ingénieur en informatique : nous avons appris à gérer notre temps de manière plus efficace et appris à nous concentrer sur les tâches importantes. Chacun codait de son côté les différentes classes qui lui avait été données à implémenter et nous organisions des réunions de groupe pour voir notre avancée et discuter certains points qui pouvaient ne pas être clairs.

De plus, il était important pour nous de coder un réseau de neuronnes car nous voulions éviter de coder un jeu, chose que nous avons déjà pu faire dans des projets précédents. Nous avions déjà pu voir la théorie concernant les réseaux de neuronnes grâce au cours de Machine Learning du semestre 8 mais ce projet nous a permis de comprendre cette théorie en la mettant en pratique.

Finalement, ce projet a été entièrement codé en C++, un langage que nous avons appris cette année. Avant d'appliquer la théorie vue dans le cours de M. Kotowicz, nous avons fait des diagrammes UML pour modéliser notre réseau de neuronnes. Nous nous sommes finalement aperçu, en commençant à coder, que certaines de nos idées de modélisation étaient incompatible avec l'implémentation de notre réseau. Ce langage s'est donc avéré être un langage efficace pour l'implémentation de notre réseau de neurone.

8 Bibliographie

— Aflak Omar : Réseau de neurones en partant de zéro en Python : https ://medium.com/france-school-of-ai/mathC3A9matiques-des-rC3A9seaux-de-neurones-code-python-613d8e83541
consulté le 24/04/2023

— Moutot Etienne : Reconnaissance de caractères à l'aide des réseaux de neurones : https ://emoutot.perso.math.cnrs.fr/static/etudes/MP/TIPE/rapport_ens.pdf
consulté le 24/04/2023

— Mestan Alp : Introduction aux résaux de neurones - Artificiels Feed Forward : https ://alp.developpez.com/tutoriels/intelligence-artificielle/reseaux-de-neurones/
consulté 22/04/2023

— Desruisseaux Martin, Pétit Michel, Constantin de Magny Guillaume et Huynh Frédéric : Essai d'utilisation des réseaux de neurones : https ://books.openedition.org/irdeditions/10038 ?lang=fr
consulté le 03/04/2023

— Mohammed Msaaf, Fouad Belmajdoub. L'application des réseaux de neurone de type " feedforward " dans le diagnostic statique. Xème Conférence Internationale : Conception et Production Intégrées, Dec 2015, Tanger, Maroc. ffhal-01260830
consulté le 22/04/2023