Rapport UE LO21

HAUTEFAYE Corentin
JOURDAN Lorna

Janvier 2025

Table des matières

1 Introduction

Dans le cadre de l'UE LO21 (*Semestre A24*), nous avons realisé un projet qui consiste à faire découvrir aux étudiants le principe de réseau neuronal. En effet, dans un monde où l'on doit traiter de plus en plus de données, il devient nécessaire de poser un modèle informatique afin de reconnaître des motifs et des relations dans des données d'entrée.

Ainsi, le sujet propose d'implémenter un réseau de couches de neurones déterministe (*cf. fonction d'activation 3*). Ce dernier sera entre-autres en mesure d'évaluer à l'aide d'une entrée, une assertion logique, i.e. un énoncé prenant la valeur 0 ou 1.

Pour ce projet, nous avons choisi de travailler en utilisant le contrôle de versions *Git*, car cela était plus simple et efficace afin de travailler à plusieurs. Vous pouvez accéder au dépôt GitHub en cliquant ici.

2 Conventions & Lexique

Dans cette partie, nous allons expliciter les conventions d'écriture que nous avons choisies, ainsi que le lexique associé.

Tout d'abord, nous avons choisi une typographie de type "*Camel Case*" ; en effet, il est important dans tout projet informatique, qu'il soit théorique ou pratique, de rester cohérent dans les notations utilisées mais également dans les noms donnés aux objets. De plus, les types dits "*enregistrement*" prennent une majuscule, tandis que les types standards commencent par une minuscule. Les champs d'un type enregistrement commencent par une minuscule. Par ailleurs, dans un souci de bonne compréhension, toutes les fonctions et procédures présentées ici utilisent uniquement des noms français, ce qui n'est pas le cas du code en C où tout est en Anglais.

Ensuite, il est **très** important de garder en mémoire que les noms des algorithmes, qui doivent obligatoirement figurer dans ce rapport, ont été changés. Ainsi, `InitNeur` → `allouerNeurone`, `Outneurone` → `obtenirSortieNeurone`, `InitCouche` → `allouerCouche`, `Outcouche` → `obtenirSortieCouche` et `CreerResNeur` → `allouerReseau`, en notant que le symbole → se lit dans ce paragraphe "est remplacé par".

Puis, on suppose que les primitives `creer`, `liberer` et `afficher` sont définies de la manière suivante :

1. `creer` est une fonction qui prend en paramètre le nom d'un type (*e.g entier*) et qui crée en mémoire un objet de même nature initialisé par défaut et le renvoit.
2. `liberer` est une procédure qui prend un objet en paramètre et qui détruit ce dernier en mémoire.
3. `afficher` est une procédure *variadique* qui prend **au moins** un paramètre et qui affiche à l'utilisateur, peu importe la manière, la chaîne de caractères correctement formatée. Le premier paramètre doit être obligatoirement une chaîne de caractères. Les autres, s'ils existent, peuvent être de type quelconque, pourvu que ça ait du sens. De plus, pour tout $i \in \mathbb{N}$, la présence du métacaractère '$\%i$' indique le remplacement de celui-ci par la valeur du i-ème paramètre dans la chaîne de caractères affichée.

Ensuite, les principes mathématiques de quantification ont bien été respectés. Ainsi, le lecteur devra partir du principe que si une fonction (resp. procédure, variable) est rencontrée sans avoir été déclarée avant, alors celle-ci est indéfinie et n'existe pas.

De plus, pour un type enregistrement S donné, on admet que pour tout champ c, l'écriture $c(S)$ est valide et une expression prenant la valeur du champ c. De plus, pour tout champ c et objet e du type du champ c, $c(S) \leftarrow e$ est une écriture valide et écrase l'ancienne valeur du champ c par e.

Quand ce n'est pas précisé, le pas d'une boucle "Pour" est par défaut de 1. De même pour une boucle "Repète... jusqu'à..." quand une expression numérique positive se trouve en condition.

L'utilisation du caractère "$\emptyset$" indique un objet inexistant ou non défini. Cette notation est analogue au pointeur nul en C.

En sus de cela, on part du principe que l'évaluation des assertions booléennes est "*paresseuse*" et se fait de gauche à droite. En conséquence, cela implique que dès lors que l'on rencontre "Faux" en évaluant une proposition normale, alors on arrête l'évaluation.

De plus, les retours anticipés sont valides. Dans le cas d'une fonction `nomFonction` qui doit renvoyer un objet X, il suffit d'écrire `nomFonction` $\leftarrow X$ pour effectuer cette opération.

Enfin, pour les types dits *polymorphes*, l'usage de chevrons "<>" avec le nom du type utilisé entre permet de définir correctement l'objet. Les types définis par défaut sont **entier**, **entier non signé**, **réel**, **réel non signé**, **booléen**, **caractère**, **chaîne de caractères** et **tableau<>**.

3 Définition de types

Tout d'abord, il est nécessaire de définir le type Liste<T> dont on note T le type. On choisit la représentation doublement chaînée avec descripteur de liste. Ainsi, on définit un *noeud* (ou *élement*) de type T, noté Noeud<T>, comme étant un enregistrement contenant les champs suivants : $\begin{cases} \texttt{valeur} & : \texttt{T} \\ \texttt{suivant} & : \texttt{Noeud<T>} \\ \texttt{precedent} & : \texttt{Noeud<T>} \end{cases}$.

On définit une *liste* (ou *descripteur de liste*) de type T, noté Liste<T>, comme étant un enregistrement contenant les champs suivants : $\begin{cases} \texttt{taille} & : \texttt{entier non signé} \\ \texttt{tete} & : \texttt{Noeud<T>} \\ \texttt{queue} & : \texttt{Noeud<T>} \end{cases}$.

Pour notre projet, nous n'avons pas besoin de rendre le type liste polymorphe ; en effet, bien que ce soit possible en C, cela reste trop inefficace et conséquent pour un si petit projet (*Un langage d'implémentation tel que le C++ ou Java aurait été plus propice dans ce cas*). Ainsi, nous étudierons uniquement le type Liste<réel>, une liste doublement chaînée de réels. De plus, le lecteur notera l'abus de langage précédent ; pour être rigoureux, il faudrait dire "*liste doublement chaînée de noeuds de réels*". Pour des soucis d'écriture, on admet que pour toute liste L : Liste<réel> et pour tout $i \in [\![0; \texttt{taille}(L) - 1]\!]$, L_i désigne le i-ème noeud dans la liste L.

D'après l'énoncé, on définit le type enregistrement Neurone qui contient les champs suivants : $\begin{cases} \texttt{poids} & : \texttt{Liste<réel>} \\ \texttt{seuil} & : \texttt{réel} \end{cases}$. Plus généralement, un neurone aurait un champ supplémentaire `fonctionActivation`. Or, comme ici cette dernière est constante, il est inutile et inefficace de stocker $n \in \mathbb{N}^*$ fois cette information.

Ici, on définit la *fonction d'activation d'un neurone* N comme l'application

$$f : \begin{array}{ll} \texttt{Neurone} \times \texttt{Liste<réel>} & \to \{0; 1\} \\ (N, E) & \mapsto \left(\sum_{i=0}^{\texttt{taille}(E)} E_i \times \texttt{poids}(N)_i\right) \geq \texttt{seuil}(N) \end{array}.$$

Notons que comme f est une fonction au sens mathématique, pour un couple de valeurs données, f renverra toujours la même image, d'où le caractère déterministe.

On définit une couche de neurones comme une liste doublement chaînée de noeuds de neurones. D'où les types `Noeud<Neurone>` : $\begin{cases} \texttt{neurone} & : \texttt{Neurone} \\ \texttt{suivant} & : \texttt{Noeud<Neurone>} \\ \texttt{precedent} & : \texttt{Noeud<Neurone>} \end{cases}$ et `Couche` : $\begin{cases} \texttt{taille} & : \texttt{entier non signé} \\ \texttt{tete} & : \texttt{Noeud<Neurone>} \\ \texttt{queue} & : \texttt{Noeud<Neurone>} \end{cases}$.

On suppose de plus que chaque neurone au sein d'une même couche dispose du même nombre d'entrées.

De manière analogue, on définit un réseau comme une liste doublement chaînée de noeuds de couches de neurones. D'où les types `Noeud<Couche>` : $\begin{cases} \texttt{couche} & : \texttt{Couche} \\ \texttt{suivant} & : \texttt{Noeud<Couche>} \\ \texttt{precedent} & : \texttt{Noeud<Couche>} \end{cases}$ et `Reseau` : $\begin{cases} \texttt{taille} & : \texttt{entier non signé} \\ \texttt{nombreEntrees} & : \texttt{entier non signé} \\ \texttt{tete} & : \texttt{Noeud<Couche>} \\ \texttt{queue} & : \texttt{Noeud<Couche>} \end{cases}$. Notons ici que dans le cas d'une propagation avant, la tête d'un réseau correspond à la couche des entrées tandis que la queue correspond à la couche de sorties.

Pour finir, nous avons fait le choix de la liste doublement chaînée, car les opérations de lecture, d'ajout et de suppression en queue (*comme en tête*) se font en $o(1)$, de même pour récupérer la taille d'une liste. De plus, cela est intéressant dans le cas de l'évaluation d'un réseau par propagation arrière.

4 Algorithmes

Dans cette section, nous allons vous présenter les algorithmes nécessaires à la réalisation du projet. Il est toutefois à noter qu'une partie a été omise car ils ne présentent pas de réel intérêt pour l'exposé (*e.g les procédures d'affichage de listes*), ou bien car ils sont fortement similaires à d'autres déjà présentés. Cela permet également de rendre la lecture plus agréable.

4.1 Liste de réels

Algorithme 1: Création d'un noeud (réel) de liste

Entrée: $c \in \mathbb{R}$
Sortie: Le noeud créé
Fonction *allouerNoeud(c : réel) : Noeud<réel>*
Début
  res : Noeud<réel> ← creer(Noeud<réel>);
  valeur(res) ← c;
  suivant(res) ← $\emptyset$;
  precedent(res) ← $\emptyset$;
  allouerNoeud ← res;
Fin

Algorithme 2: Création d'une liste de réels

Entrée: $\emptyset$
Sortie: La liste créée
Fonction *allouerListe() : Liste<réel>*
Début
  res : Liste<réel> $\leftarrow$ creer(Liste<réel>);
  taille(res) $\leftarrow$ 0;
  tete(res) $\leftarrow$ $\emptyset$;
  queue(res) $\leftarrow$ $\emptyset$;
  allouerListe $\leftarrow$ res;
Fin

Algorithme 3: Insertion d'un réel en tête de liste

Entrée: L : Liste<réel>, $v \in \mathbb{R}$
Sortie: La liste avec v en tête
Fonction *insererTeteListe(L : Liste<réel>, v : réel) : Liste<réel>*
Début
  res : Liste<réel>;
  Si $L \neq \emptyset$ **Alors**
    $res \leftarrow L$;
  Sinon
    res $\leftarrow$ allouerListe();
  Fin Si
  $newel$: Noeud<réel> $\leftarrow$ allouerNoeud(v);
  suivant($newel$) $\leftarrow$ tete(res);
  Si *tete(res)* $= \emptyset$ **Alors**
    queue(res) $\leftarrow$ $newel$;
  Sinon
    precedent(tete(res)) $\leftarrow$ $newel$;
  Fin Si
  tete(res) $\leftarrow$ $newel$;
  taille(res) $\leftarrow$ taille(res) + 1;
  insererTeteListe $\leftarrow$ res;
Fin

Algorithme 4: Insertion d'un réel en queue de liste

```
Entrée: L : Liste<réel>, v ∈ ℝ
Sortie: La liste avec v en queue
Fonction insererQueueListe(L : Liste<réel>, v : réel) : Liste<réel>
Début
    res : Liste<réel>;
    Si L ≠ ∅ Alors
        res ← L;
    Sinon
        res ← allouerListe();
    Fin Si
    newel : Noeud<réel> ← allouerNoeud(v);
    precedent(newel) ← queue(res);
    Si queue(res) = ∅ Alors
        tete(res) ← newel;
    Sinon
        suivant(queue(res)) ← newel;
    Fin Si
    queue(res) ← newel;
    taille(res) ← taille(res) + 1;
    insererQueueListe ← res;
Fin
```

Algorithme 5: Suppression de la tête d'une liste de réels

```
Entrée: L : Liste<réel>
Sortie: La liste L avec la tête en moins
Fonction supprimerTeteListe(L : Liste<réel>) : Liste<réel>
Début
    Si L = ∅ OU taille(L) = 0 Alors
        supprimerTeteListe ← ∅;
    Fin Si
    res : Liste<réel> ← L;
    tmp : Noeud<réel> ← tete(res);
    tete(res) ← suivant(tmp);
    Si tete(res) = ∅ Alors
        queue(res) ← ∅;
    Sinon
        precedent(tete(res)) ← ∅;
    Fin Si
    liberer(tmp);
    taille(res) ← taille(res) - 1;
    supprimerTeteListe ← res;
Fin
```

Algorithme 6: Suppression de la queue d'une liste de réels

```
Entrée: L : Liste<réel>
Sortie: La liste L avec la queue en moins
Fonction supprimerQueueListe(L : Liste<réel>) : Liste<réel>
Début
    Si L = ∅ OU taille(L) = 0 Alors
        supprimerQueueListe ← ∅;
    Fin Si
    res : Liste<réel> ← L;
    tmp : Noeud<réel> ← queue(res);
    queue(res) ← precedent(tmp);
    Si queue(res) = ∅ Alors
        tete(res) ← ∅;
    Sinon
        suivant(queue(res)) ← ∅;
    Fin Si
    liberer(tmp);
    taille(res) ← taille(res) - 1;
    supprimerQueueListe ← res;
Fin
```

Algorithme 7: Création d'une liste de n réels assignés à une valeur réelle donnée

```
Entrée: (n, v) ∈ ℕ⁺ × ℝ
Sortie: Une liste contenant n fois la valeur v
Fonction creerListe(n : entier non signé, v : réel) : Liste<réel>
Début
    res : Liste<réel> ← allouerListe();
    Répéter
        res ← insererTeteListe(res, v);
    Jusqu'à n;
    creerListe ← res;
Fin
```

Algorithme 8: Suppression d'une liste de réels

```
Entrée: L : Liste<réel>
Sortie: ∅
Procédure libererListe(L : Liste<réel>)
Début
    Si L ≠ ∅ Alors                                        /* L existe */
        tmp : Liste<réel> ← L;
        Tant que taille(tmp) > 0 Faire
            tmp ← supprimerTeteListe(tmp);
        Fin Tant que
        liberer(tmp);
    Fin Si
Fin
```

4.2 Neurone

Algorithme 9: Création d'un neurone

Entrée: L : Liste<réel>, $s \in \mathbb{R}$
Sortie: Le neurone créé de poids L et de seuil s
Fonction *allouerNeurone(L : Liste<réel>, s : réel) : Neurone*
Début
    res : Neurone $\leftarrow$ creer(Neurone);
    poids(res) $\leftarrow L$;
    seuil(res) $\leftarrow s$;
    allouerNoeud $\leftarrow res$;
Fin

Algorithme 10: Suppression d'un neurone

Entrée: N : Neurone
Sortie: $\emptyset$
Procédure *libererNeurone(N : Neurone)*
Début
    Si $N \neq \emptyset$ **Alors**
        LibererListe(Poids(N));
        Liberer(N);
    Fin Si
Fin

Algorithme 11: Obtention de la sortie d'un neurone

Entrée: N : Neurone, L : Liste<réel>
Sortie: Renvoie un réel (0 ou 1) selon si la somme pondérée de l'entrée dépasse le seuil du neurone
Fonction *obtenirSortieNeurone(N : Neurone)*
Début
    Si $N = \emptyset$ **OU** *poids(N)* $= \emptyset$ **OU** $L = \emptyset$ **OU** *taille(L)* $\neq$ *taille(poids(N))* **Alors**
    `/* Le neurone` N `n'existe pas ou sa liste de poids est indéfinie ou la liste des entrées` L `est indéfinie ou erreur de dimension */`
        afficher("Impossible de produire une sortie !");
        obtenirSortieNeurone $\leftarrow 0$;
    Fin Si
    x : réel $\leftarrow 0$;
    W : Noeud<réel> $\leftarrow$ tete(poids(N));
    E : Noeud<réel> $\leftarrow$ tete(L);
    Tant que $W \neq \emptyset$ **Faire** `/* Pas besoin de vérifier sur` E `car listes de même taille et dans lesquelles on itère "en phase" */`
        $x \leftarrow x +$ valeur(W) $\times$ valeur(E);
        $W \leftarrow$ suivant(W);
        $E \leftarrow$ suivant(E);
    Fin Tant que
    obtenirSortieNeurone $\leftarrow$ ($x \geq$ seuil(N));
Fin

4.3 Couche

Les fonctions d'insertion et de suppression en tête (resp. en queue) d'un élément d'une couche de neurones sont analogues à celles des listes de réels et ont donc été omises.

Algorithme 12: Création d'un noeud de couche

Entrée: N : Neurone
Sortie: Un nouvel élément de couche contenant un neurone N
Fonction *allouerNoeudCouche(L : Liste<réel>, s : réel) : Noeud<Neurone>*
Début
    res : Noeud<Neurone> ← creer(Noeud<Neurone>);
    neurone(res) ← N;
    suivant(res) ← $\emptyset$;
    precedent(res) ← $\emptyset$;
    allouerNoeudCouche ← res;
Fin

Algorithme 13: Suppression d'un noeud de couche

Entrée: E : Noeud<Neurone>
Sortie: $\emptyset$
Procédure *libererNoeudCouche(E : Noeud<Neurone>)*
Début
    Si $E \neq \emptyset$ **Alors**
        libererNeurone(neurone(E));
        liberer(E);
    Fin Si
Fin

Algorithme 14: Création d'une couche de neurones

Entrée: $(n, e) \in (\mathbb{N})^2$
Sortie: Une couche de n neurones à e entrées
Fonction *allouerCouche(n : entier non signé, e : entier non signé) : Couche*
Début
    Si $n = 0$ **OU** $e = 0$ **Alors**
        allouerCouche ← $\emptyset$;
    Fin Si
    C : Couche ← creer(Couche);
    tete(C) ← $\emptyset$;
    queue(C) ← $\emptyset$;
    taille(C) ← 0;
    Pour i **de** *0* **à** *n-1* **Faire**
        afficher("Acquisition des poids du neurone %0", i);
        W : Liste<réel> ← $\emptyset$;
        Pour j **de** *0* **à** *e-1* **Faire**
            afficher("Entrez le %0 poids :", j);
            p : réel ← acquisition();
            W ← insererQueueListe(W, p);
        Fin Pour
        afficher("Entrez le seuil du neurone :");
        s : réel ← acquisition();
        N : Neurone ← allouerNeurone(W, s);
        C ← insererTeteCouche(C, N);
    Fin Pour
    allouerCouche ← C;
Fin

Algorithme 15: Suppression d'une couche de neurones

Entrée: C : Couche
Sortie: $\emptyset$
Procédure *libererCouche(C : Couche)*
Début
  Si $C \neq \emptyset$ **Alors**
    tmp : Couche $\leftarrow C$;
    Tant que *taille(tmp) > 0* **Faire**
      $tmp \leftarrow$ supprimerTeteCouche(tmp);
    Fin Tant que
    liberer(tmp);
  Fin Si
Fin

Algorithme 16: Obtention de la sortie d'une couche de neurones

Entrée: C : Couche, L : Liste<réel>
Sortie: Une liste contenant la sortie de chaque neurone dans la couche C
Fonction *obtenirSortieCouche(C : Couche, L : Liste<réel>) : Liste<réel>*
Début
  Si $C = \emptyset$ **OU** *taille(C)* $= 0$ **OU** $L = \emptyset$ **OU** *taille(L) = 0* **Alors**
    obtenirSortieCouche $\leftarrow \emptyset$;
  Fin Si
  res : Liste<réel> $\leftarrow$ creerListe(taille(C), 0);
  No : Noeud<réel> $\leftarrow$ tete(res);
  It : Noeud<Neurone> $\leftarrow$ tete(C);
  Tant que $It \neq \emptyset$ **Faire**
    valeur(No) $\leftarrow$ obtenirSortieNeurone(neurone(It), L);
    $No \leftarrow$ suivant(No);
    $It \leftarrow$ suivant(It);
  Fin Tant que
  obtenirSortieCouche $\leftarrow res$;
Fin

4.4 Réseau

Les fonctions d'insertion et de suppression en tête (resp. en queue) d'un élément dans un réseau de couches de neurones sont analogues à celles des listes de réels et ont donc été omises.

Algorithme 17: Création d'un noeud de réseau

Entrée: C : Couche
Sortie: Un nouvel élément de réseau contenant une couche C
Fonction *allouerNoeudReseau(C : Couche) : Noeud<Couche>*
Début
  res : Noeud<Couche> $\leftarrow$ creer(Noeud<Couche>);
  couche(res) $\leftarrow C$;
  suivant(res) $\leftarrow \emptyset$;
  precedent(res) $\leftarrow \emptyset$;
  allouerNoeudReseau $\leftarrow res$;
Fin

Algorithme 18: Suppression d'un noeud de réseau

```
Entrée: E : Noeud<Couche>
Sortie: ∅
Procédure libererNoeudRéseau(E : Noeud<Couche>)
Début
    Si E ≠ ∅ Alors
        libererCouche(couche(E));
        liberer(E);
    Fin Si
Fin
```

Algorithme 19: Création d'un réseau de neurones

```
Entrée: c ∈ ℕ, L : Liste<réel>, e ∈ ℕ
Sortie: Un réseau à e entrées de c couches de L_i neurones
Fonction allouerReseau(c : entier non signé, L : Liste<réel>, e : entier non signé) :
  Reseau
Début
    Si L = ∅ OU c = 0 OU taille(L) ≠ c Alors
        allouerReseau ← ∅;
    Fin Si
    res : Reseau ← creer(Reseau);
    tete(res) ← ∅;
    queue(res) ← ∅;
    taille(res) ← 0;
    nombreEntrees(res) ← e;
    i : entier non signé ← e;
    It : Noeud<réel> ← tete(L);
    Tant que It ≠ ∅ Faire
        afficher("Allocation de la couche %0", 1 + taille(res));
        C : Couche ← allouerCouche(valeur(It), i);
        res ← insererQueueReseau(res, C);
        i ← valeur(It);
        It ← suivant(It);
    Fin Tant que
    allouerReseau ← res;
Fin
```

Algorithme 20: Suppression d'un réseau de neurones

```
Entrée: R : Reseau
Sortie: ∅
Procédure libererReseau(R : Reseau)
Début
    Si R ≠ ∅ Alors
        tmp : Reseau ← R;
        Tant que taille(tmp) > 0 Faire
            tmp ← supprimerTeteCouche(tmp);
        Fin Tant que
        liberer(tmp);
    Fin Si
Fin
```

Algorithme 21: Création d'un noeud de réseau

Entrée: C : Couche
Sortie: Un nouvel élément de réseau contenant une couche C
Fonction *allouerNoeudReseau(C : Couche) : Noeud<Couche>*
Début
    res : Noeud<Couche> $\leftarrow$ creer(Noeud<Couche>);
    couche(res) $\leftarrow C$;
    suivant(res) $\leftarrow \emptyset$;
    precedent(res) $\leftarrow \emptyset$;
    allouerNoeudReseau $\leftarrow res$;
Fin

Algorithme 22: Obtention de la liste des sorties d'un réseau

Entrée: R : Reseau, L : Liste<réel>
Sortie: Une liste contenant les sorties de la dernière couche du réseau par propagation avant
Fonction *obtenirSortiesReseau(R : Reseau, L : Liste<réel>) : Liste<réel>*
Début
    Si $R = \emptyset$ **OU** *taille(R) = 0* **OU** $L = \emptyset$ **OU** *taille(L)* $\neq$ *nombreEntrees(R)* **Alors**
        obtenirSortiesReseau $\leftarrow \emptyset$;
    Fin Si
    res : Liste<réel> $\leftarrow L$;
    It : Noeud<Couche> $\leftarrow$ tete(R);
    Tant que $It \neq \emptyset$ **Faire**
        tmp : Liste<réel> $\leftarrow$ obtenirSortieCouche(couche(It), res);
        libererListe(res);
        $res \leftarrow tmp$;
        $It \leftarrow$ suivant(It);
    Fin Tant que
    obtenirSortiesReseau $\leftarrow res$;
Fin

5 Explications implémentation en C

Pour implémenter ce projet en C, nous avons travaillé avec la norme C99 avec les bibliothèques standard. Nous avons également utilisé le logiciel *CLion* de JetBrains afin de pouvoir travailler ensemble et de debugger la mémoire plus efficacement et facilement qu'avec GDB. La version du projet dans le dépôt GitHub est un projet CLion tandis que le code source fourni avec le livrable se compile via Make. Le code peut être compilé avec GCC sur des distributions Linux et avec MinGW sur systèmes Windows. Enfin, le code source est en Anglais (*plus élégant ainsi*) ; toutefois les noms restent analogues à ceux en Français.

5.1 Structures

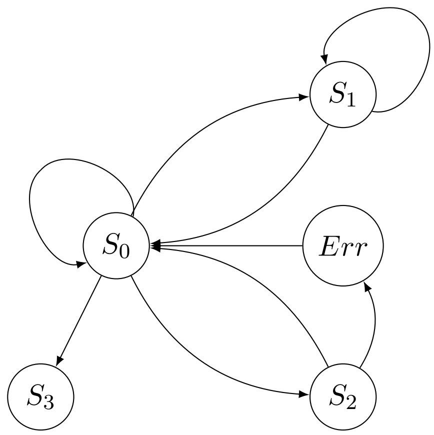
Dans cette sous-partie, on donne les structures en C des types définis en 3.

```c
typedef struct STRUCT_NODE {
    float value;
    struct STRUCT_NODE *next;
    struct STRUCT_NODE *prev;
} Node;

typedef struct {
    int length;
    Node *head;
    Node *tail;
} List;

typedef struct {
    List *weights;
    float threshold;
} Neuron;

typedef struct STRUCT_CNODE
{
    Neuron *neuron;
    struct STRUCT_CNODE *next;
    struct STRUCT_CNODE *prev;
} ClusterNode;

typedef struct
{
    int length;
    ClusterNode *head;
    ClusterNode *tail;
} Cluster;

typedef struct STRUCT_NNODE
{
    Cluster *cluster;
    struct STRUCT_NNODE *next;
    struct STRUCT_NNODE *prev;
} NetworkNode;

typedef struct
{
    int length;
    int numOfInputs;
    NetworkNode *head; // Input cluster
    NetworkNode *tail; // Output cluster
} Network;
```

5.2 Programme principal

Afin de ne pas coder les réseaux de neurones "en dur", nous avons cherché à faire l'acquisition des poids, entrées et autres données pour générer ou évaluer un réseau de neurones. Nous utilisons pour cela un automate simple à cinq états :

1. S_0 : état initial -> acquisition du choix de l'utilisateur dans le menu
   (a) Si l'acquisition est non valide, on reste dans l'état S_0
   (b) Si l'utilisateur rentre "1", alors on passe à l'état S_1
   (c) Si l'utilisateur rentre "2", alors on passe à l'état S_2
   (d) Si l'utilisateur rentre "3", alors on passe à l'état final S_3
2. S_1 : (Choix 1) création d'un réseau (et suppression s'il existe)
   (a) Si l'acquisition est invalide lors de la création du réseau, on reste dans l'état S_1
   (b) Si le réseau a été créé et configuré, on retourne dans l'état S_0
   (c) Toute erreur "grave" dans cet état conduit à un réseau nul, et donc à un retour à l'état S_0
3. S_2 : (Choix 2) obtention de la sortie d'un réseau
   (a) Si le réseau n'a pas été défini avant, on passe dans l'état d'erreur Err
   (b) Si le réseau a produit une sortie, on retourne dans l'état S_0
4. S_3 : (Choix 3) état final -> arrêt du programme
5. Err : état d'erreur -> retour à l'état S_0



Nota Bene : Pour plus de lisibilité, les étiquettes de texte ont été omises sur le schéma de l'automate.

6 Test de réseaux

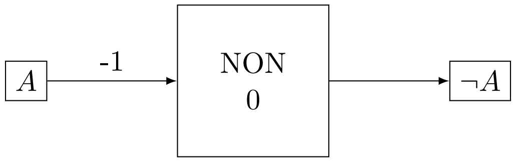
Le choix 1 (*ou état* S_1) permet de créer un réseau de neurones et supprime le précédent s'il existe. Pour cela, le programme effectue l'acquisition des paramètres suivants dans l'ordre :

1. Nombre de couches $n \in \mathbb{N}^*$
2. Nombre d'entrées (pour la première couche) $e \in \mathbb{N}^*$
3. Nombre de neurones couche par couche L : `Liste<réel>`
4. Pour tout $(i, j) \in [\![0; n-1]\!] \times [\![1; \text{valeur}(L_i)]\!]$, acquisition des poids du j-ème neurone de la $i+1$-ème couche, puis du seuil

Dans la suite, la donnée d'une chaîne d'entiers pour un réseau indique l'ordre dans lequel il faut passer les valeurs au programme.

6.1 Réseau NON

Soit $A \in \{0;1\}$. On définit un réseau NON logique par une couche contenant un neurone. Ce dernier a un seuil égal à 0 et un poids de -1.



Ce réseau peut être reproduit dans le programme avec la chaîne $1, 1, 1, -1, 0$.

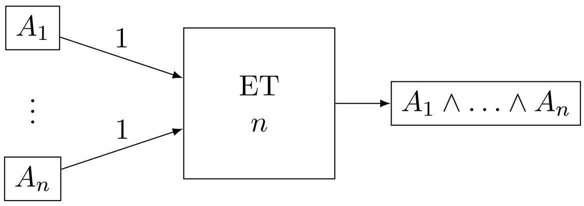
On a la table de vérité suivante :

A	$\neg A$	Somme pondérée	Sortie
0	1	0	1
1	0	-1	0

On constate que la deuxième colonne coïncide avec la dernière. Ainsi, le modèle est cohérent.

6.2 Réseau ET

Soit $n \in \mathbb{N}^*$ et $(A_i)_{i \in [\![1;n]\!]} \in \{0;1\}^n$. On définit un réseau ET logique par une couche contenant un neurone à n entrées. Ce dernier a un seuil égal à n et chaque entrée a un poids de 1.



Ce réseau peut être reproduit dans le programme avec la chaîne $1, n, 1, \underbrace{1, \ldots, 1}_{n}, n$.

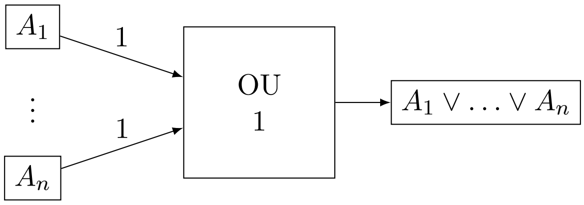
Sans perte de généralité, prenons $n = 3$. Le seuil vaut 3. On a ainsi la table de vérité suivante :

A_1	A_2	A_3	$A_1 \wedge A_2 \wedge A_3$	Somme pondérée	Sortie
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	2	0
1	0	0	0	1	0
1	0	1	0	2	0
1	1	0	0	2	0
1	1	1	1	3	1

On constate que la quatrième colonne coïncide avec la dernière. Ainsi, le modèle est cohérent.

6.3 Réseau OU

Soit $n \in \mathbb{N}^*$ et $(A_i)_{i \in [\![1;n]\!]} \in \{0;1\}^n$. On définit un réseau OU logique par une couche contenant un neurone à n entrées. Ce dernier a un seuil égal à 1 et chaque entrée a un poids de 1.



Ce réseau peut être reproduit dans le programme avec la chaîne $1, n, 1, \underbrace{1, \ldots, 1}_{n}, 1$.

Sans perte de généralité, prenons $n = 3$. Le seuil vaut 3. On a ainsi la table de vérité suivante :

A_1	A_2	A_3	$A_1 \vee A_2 \vee A_3$	Somme pondérée	Sortie
0	0	0	0	0	0
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	1	2	1
1	0	0	1	1	1
1	0	1	1	2	1
1	1	0	1	2	1
1	1	1	1	3	1

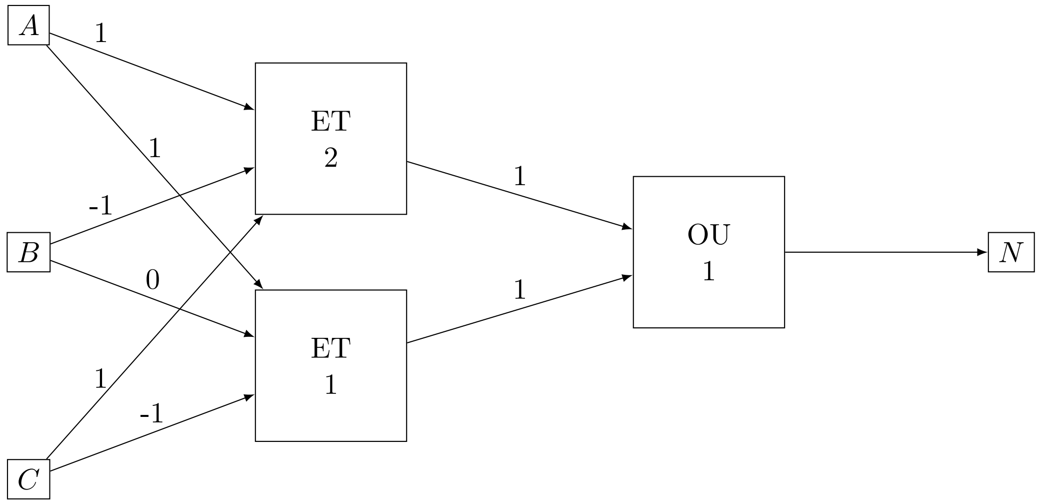
On constate que la quatrième colonne coïncide avec la dernière. Ainsi, le modèle est cohérent.

6.4 Réseau multi-couches

Soit $(A, B, C) \in \{0;1\}^3$. On pose la proposition $N \equiv (A \wedge \neg B \wedge C) \vee (A \wedge \neg C)$. Le but est de définir un réseau multi-couches qui évalue l'assertion N. On propose alors un réseau à deux couches. La première effectue l'opération ET logique, tandis que la deuxième fait l'opération OU logique entre les sorties de la couche 1. La couche 1 prend trois entrées, et renvoie deux sorties. Ainsi, la couche 2 prend deux entrées et renvoie une sortie. La sortie de la couche 2 dans le cas de la propagation avant est le résultat attendu.

Un poids de 1 pour une entrée, n'a aucun effet ; on parle d'entrée identité. Un poids de -1 pour une entrée, simule un NON logique. Un poids de 0 pour une entrée, permet de la masquer. Il reste à calculer le seuil. Pour une porte ET à $n \in \mathbb{N}^*$ entrées de poids P : `Liste<réel>`, le seuil vaut $1 + \sum_{i=0}^{n} P_i$. Cette formule peut être démontrée à l'aide d'une itération finie sur n. Pour une porte OU à $n \in \mathbb{N}^*$ entrées de poids P : `Liste<réel>`, le seuil vaut $1 - \sum_{i=0}^{n} \delta_{L_i,-1}$ où pour tout $(i,j) \in \mathbb{Z}^2, \delta_{i,j} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{sinon} \end{cases}$ (*symbole de Kronecker, ici compte le nombre d'entrées au poids -1*). Cette formule peut être également démontrée par une itération finie sur n.

On peut ainsi utiliser le réseau suivant pour évaluer N :



Ce réseau peut être reproduit dans le programme avec la chaîne $2, 3, 2, 1, 1, -1, 1, 2, 1, 0, -1, 1, 1, 1, 1$.

Sans perte de généralité, prenons $n = 3$. Le seuil vaut 3. On a ainsi la table de vérité suivante :

A	B	C	$A \wedge \neg B \wedge C$	$A \wedge \neg C$	N
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	1	0	0	0

S.p. [a] ET 1	S.p. ET 2	Sortie couche 1	S.p. OU 1	Sortie couche 2
0	0	{0 ;0}	0	0
1	-1	{0 ;0}	0	0
-1	0	{0 ;0}	0	0
0	-1	{0 ;0}	0	0
1	1	{0 ;1}	1	1
2	0	{1 ;0}	1	1
0	1	{0 ;1}	1	1
1	0	{0 ;0}	0	0

a. Somme pondérée

On constate que l'évaluation de N coïncide avec la sortie du réseau associé. Le modèle proposé est donc cohérent.

7 Conclusion

Travailler sur ce projet de réseau de neurones a été une expérience particulièrement enrichissante, tant sur le plan technique que sur l'organisation. Cependant, ce projet a également mis en lumière plusieurs difficultés que nous avons dû surmonter.

La première difficulté majeure concernait l'organisation et la planification du projet. En effet, il s'agissait de l'un des trois gros projets à finaliser avec sensiblement la même échéance. Il nous a ainsi fallu prioriser certaines tâches et apprendre à gérer notre temps de manière efficace. Cela nous a donc permis de développer notre capacité d'adaptation et de gestion, ainsi que sur la collaboration. Nous avons entre autres organisé des sessions en distanciel afin de pouvoir travailler.

De plus, certaines consignes étaient ambiguës, ce qui a entraîné une confusion quant aux attendus.

Une amélioration que nous aurions pu implémenter est de pouvoir sauvegarder et charger des réseaux, afin de ne pas avoir à faire l'acquisition à chaque fois.

In fine, ce projet nous a permis de développer des compétences variées et utiles en allant de la programmation, à l'algorithmique mais aussi de la gestion de projet. Cela nous sera très utile dans la poursuite de nos études.