Rapport du projet : « Réseaux de neurones »

Ganne Charles

24 décembre 2024

Table des matières

1 Introduction

Ce projet vise à introduire aux réseaux de neurones et à développer une implémentation simplifiée en langage C. Ce rapport présentera les choix de conception et d'implémentation qui ont été fait relativement aux structures de donnée et à la démarche adoptée, les algorithmes des principaux sous programmes et leurs applications ainsi que le jeux de tests qui a été confectionné pour éprouver ce pourquoi a été fait ce projet (évaluer des expressions logiques).

L'objectif principal est d'illustrer le fonctionnement des réseaux de neurones dans un cadre discret (utilisation d'entiers exclusivement pour les poids, biais et l'unique fonction d'activation). Seule la propagation avant sera réalisée par le réseau, ainsi il faudra choisir méticuleusement les poids et biais car ces derniers ne seront pas corrigés lors d'une propagation arrière.

2 Description des choix de conception et d'implémentation relatifs aux structures de données utilisées et à la démarche adoptée

2.1 Conception

Comme indiqué précédemment ainsi que dans l'énoncé, ce projet ne traite que des entiers et ne réalise pas de propagation arrière dans le réseau par souci de simplicité. Ainsi sa seule application que nous en feront ici sera l'évaluation d'expressions logiques booléennes. Il faudra donc bien choisir les poids et biais ainsi que la fonction d'activation lors de l'initialisation du réseau.

2.2 Implémentation

Le langage dans lequel l'implémentation doit se faire est le langage C. C'est un langage compilé qui génère du code machine, donc assez optimisé. Dans ce projet nous avons eu besoin à plusieurs reprises d'utiliser le type de donnée abstrait liste. Son implémentation en C a été réalisée avec les listes doublement chaînées, cela permet d'insérer et de supprimer des éléments ainsi que de connaître la taille de la liste avec une complexité constante. Nous aurons ainsi une meilleure complexité temporelle du code une fois le réseau initialisé en mémoire.

Enfin, par souci de lisibilité du code, la convention snake_case a été adoptée pour le nommage de tous les identificateurs utilisés dans le code, et non UpperCamelCase comme suggéré par l'énoncé.

3 Définition des types abstraits de donnée utilisés dans ce projet et des opérations associées

3.1 Définition du type abstrait LISTE (toutes ces opérations ne seront pas forcément implémentées ou utilisées)

Type abstrait : LISTE (ELT)

Constructeurs :

- **creer :** $\varnothing \rightarrow LISTE$ // Construit une liste vide.
- **ajoutert :** $LISTE \times ELT \rightarrow LISTE$ // Ajoute un élément en tête de liste.
- **ajouterq :** $LISTE \times ELT \rightarrow LISTE$ // Ajoute un élément en queue de liste.
- **supprimert :** $LISTE \rightarrow LISTE$ // Supprime le premier élément de la liste.
- **supprimerq :** $LISTE \rightarrow LISTE$ // Supprime le dernier élément de la liste.

Observateurs et fonctions d'accès :

- **vide :** $LISTE \rightarrow$ BOOLEEN // Teste si une liste est vide.
- **tete :** $LISTE \rightarrow ELT$ // Renvoie l'élément en tête de liste (et non sa place.)
- **queue :** $LISTE \rightarrow ELT$ // Renvoie l'élément en queue de liste (et non sa place.)
- **reste :** $LISTE \rightarrow LISTE$ // Retourne la même liste privée de son premier élément (tête) si elle en a un.

3.2 Définition du type abstrait NEURONE

Type abstrait : NEURONE

Attributs (éléments du struct en C) :

- **Poids :** Liste des poids : LISTE(ENTIER) // Chaque élément correspond au poids d'une entrée.
- **Biais :** Entier : ENTIER // Représente le biais du neurone.

Constructeurs :

- **InitNeur :** $ENTIER \times LISTE(ENTIER) \rightarrow NEURONE$ // Initialise un neurone avec un biais (l'entier) et une liste de poids (la liste d'entiers).

Opérations :

- **OutNeur :** $NEURONE \times LISTE(ENTIER) \rightarrow INT$ // Calcule la sortie du neurone à partir d'une liste d'entiers entrée, des poids et du biais (dans le neurone). La fonction d'activation fais la somme pondérée des entrées, si elle est supérieur ou égale au biais elle retourne 1, sinon 0.

Observateurs et fonctions d'accès :

- **poids :** $NEURONE \rightarrow LISTE(ENTIER)$ // Renvoie la liste des poids du neurone.
- **biais :** $NEURONE \rightarrow ENTIER$ // Renvoie le biais du neurone.

4 Principaux algorithmes utilisés et implémentés dans ce projet

4.1 InitNeur : Création d'un neurone

Entrées : Un entier *biais* qui sera le biais du neurone et une liste d'entiers *poids* dont les éléments correspondent aux poids du neurone.

Sortie : Un neurone correctement intialisé. Cf figure 1.

```
Fonction InitNeur (biais : ENTIER, poids : LISTE(ENTIER)) : NEURONE
Début
        n : NEURONE
        l : LISTE(ENTIER) <- creer()
        biais(n) <- biais
        tant que NON(vide(poids)) faire
                ajouterq(l, tete(poids))
                poids <- reste(poids)
        fait
        poids(n) <- l
        InitNeur <- n
Fin
```

FIGURE 1 – InitNeur

4.2 Outneurone : Calcul de la sortie d'un neurone

Entrée : Un neurone n à travers lequel passeront les entrées et une liste d'entiers *entrees* qui sont les entrées du neurone.

Sortie : Un entier correspondant à la sortie du neurone. Cf figure 2.

```
Fonction Outneurone(n : NEURONE, entrées : LISTE(ENTIRE)) : ENTIER
Début
        x : ENTIER <- 0
        tant que NON vide(entrées) faire
                x <- x + (tete(entrées) * tete(poids(n)))
                entrées <- reste(entrées)
        fait
        si x >= biais(n) alors
                Outneurone <- 1
        sinon
                Outneurone <- 0
        finsi
Fin
```

FIGURE 2 – Outneurone

4.3 InitCouche : Permet de créer une couche et d'initialiser ses neurones

Entrée : *biais* la liste des biais dans l'ordre souhaité dans la couche et *poids* la liste des listes d'entiers qui sont les poids de chaque neurone.

Sortie : Une liste de neurones symbolisant la couche. Cf figure 3.

```
Fonction InitCouche(biais : LISTE(ENTIER), poids LISTE(LISTE(ENTIER))) :
LISTE(NEURONE)
Début
        nbNeur : ENTIER <- taille(poids)
        C : LISTE(NEURONE) <- creer()
        tant que NON(vide(biais) OU vide(poids))  faire
                n : NEURONE <- InitNeur(tete(biais), tete(poids))
                biais <- reste(biais)
                poids <- reste(poids)
                ajouterq(c, n)
        fait
        InitCouche <- c
Fin
```

FIGURE 3 – InitCouche

4.4 OutCouche : Calcul des sorties d'une couche

Entrée : *couche* une liste de neurones et *entrees* une liste de listes d'entiers, chaque sous liste étant l'entrée d'un des neurones de la première liste.

Sortie : Une liste d'entiers, chacun étant la sortie d'un neurone. Cf figure 4.

```
Fonction Outcouche(couche : LISTE(NEURONE), entrées : LISTE(LISTE(ENIER))) :
LISTE(ENTIER)
Début
        sorties : LISTE(ENTIER) <- creer()
        tant que NON(vide(couche) OU vide(entrées)) faire
                ajouterq(sorties, Outneurone(tete(couche), tete(entrees))
                couche <- reste(couche)
                entrées <- reste(entrées)
        fait
        Outcouche <- sorties
Fin
```

FIGURE 4 – OutCouche

4.5 Explication de la gestion du réseau

Pour le réseau, nous traiterons simplement avec une liste doublement chaînée de couches. Ainsi il s'agira simplement d'ajouts en queue d'une liste doublement chaînée, les éléments étants des couches. Il n'y a pas d'algorithme spécifique qui a été implémenté.

5 Compilation et exécution

Pour compiler ce projet, il faut mettre tous les fichiers dans un même dossier, ouvrir un terminal dans le dossier et effectuer la commande : `gcc *.c` par exemple pour compiler avec le compilateur gcc. Pour ensuite exécuter le code, taper la commande `./a.out`. Une fois que l'exécutable n'est plus nécessaire, taper la commande `rm ./a.out`

6 Jeux d'essais

Afin d'approuver le code, j'ai implémenté en dur une expression logique traduite en un réseau de neurones d'abord sur papier et j'ai pu constater que le résultat est correct en lui donnant toutes les combinaisons d'entrées possibles et en constatant que la table de vérité complète générée était la bonne.

J'ai ensuite commenté ce code, et créé un menu afin qu'un utilisateur puisse créer son propre réseau en choisissant chaque paramètre. Il serait impossible de tester le projet avec toutes les entrées possibles, mais les tests sur des réseaux très petits, très grands ou avec des valeurs incohérentes me permettent de conclure que le code est quand même solide.

7 Commentaires sur les résultats.

Ce projet a permis d'implémenter une version très simplifiée d'un réseau de neurones avec des entiers et sans la propagation arrière. Le code ne gère pas forcément les problèmes tels que les mauvaises saisies des utilisateurs (chaînes de caractères au lieu de nombres, dernière couche constituée de plusieurs neurones, interruptions clavier sans libération mémoire, ...) et doit donc être utilisé en fonction nominale. Cependant, si je reprends ce projet plus tard, afin de l'améliorer, je pense que je me concentrerai dans un premier temps sur le fait de prendre directement des expressions logiques, de créer leur arbre et de créer automatiquement le réseau associé, cela m'a l'air beaucoup plus intéressant.