TP10 POO - C++

gilles.scarella@univ-cotedazur.fr, simon.girel@univ-cotedazur.fr

1 Classe *Neurone*

L'objectif de cette première partie du TP est de réaliser un discriminateur linéaire. Nous allons pour cela implémenter un neurone artificiel tel qu'il est souvent modélisé dans les réseaux de neurones informatiques. Le problème est décrit dans l'*Annexe* (en p.5).

1.1 Déclaration et définitions pour la classe *Neurone*

Indication: Dans cette partie, vous pouvez tester la syntaxe de votre code comme suit, (seul un fichier objet non exécutable (il manque le *main*) est créé).

```
g++ -c Neurone.cpp
```

La classe *Neurone* sera déclarée dans un fichier *Neurone.hh*. Le code des fonctions de la classe *Neurone* sera écrit dans un fichier *Neurone.cpp*.
La classe *Neurone* comprend deux données privées:

- Un pointeur sur *double*, *_poids*, représentant les poids d'entrées du neurone.
- Un entier, *_n*, qui sera le nombre d'entrées du neurone.

1.1.1 Les constructeurs

La classe *Neurone* comprend deux constructeurs.

- Implémenter le premier constructeur qui prend uniquement un paramètre entier et qui initialisera *_n* avec cette valeur. Ce constructeur devra également allouer l'espace mémoire nécessaire pour *_poids* et donner des valeurs aléatoires à ces poids, entre 0 et 1 (voir la fonction *get_rand_number* du TP7).
- Implémenter le deuxième constructeur comme constructeur par copie de la classe.

1.1.2 Le destructeur

- Implémenter le destructeur de la classe *Neurone* qui devra libérer l'espace alloué.

1.1.3 L'opérateur +

Dans notre fonction d'apprentissage statistique du fichier *main.cpp*, on a besoin d'un opérateur + qui prend comme opérande un tableau de *double* supposé déjà existant.

- Implémenter un tel opérateur comme fonction membre de la classe *Neurone*. Cet opérateur devra recevoir un tableau de *double* (*double* *) en paramètre et il additionnera les éléments du tableau avec les éléments du tableau *_poids* de l'instance courante.

1.1.4 L'opérateur ()

On a également besoin d'un opérateur permettant de transformer notre objet en foncteur c'est-à-dire un objet capable d'être vu comme une fonction.
Cet opérateur calculera la sortie du neurone en fonction des entrées passées en paramètre sous forme d'un pointeur sur *double*.
Il doit donc commencer par calculer la somme pondérée des entrées et appliquer à cette somme la fonction d'activation.

1.1.5 L'opérateur <<

Nous avons vu en cours l'opérateur << qui permet un affichage propre via *cout* d'un objet placé en second opérande.
Ecrire un tel opérateur pour la classe *Neurone*. Il s'agira d'afficher les poids du neurone.
Il ne s'agit pas ici d'écrire une fonction membre car le premier opérande de cet opérateur est de type *ostream*. Cependant, on peut le déclarer en tant que fonction amie de la classe.

1.1.6 L'opérateur =

La surcharge de l'opérateur d'affectation est la plus délicate car il faut bien faire attention aux différentes allocations et libérations de l'espace mémoire.

Cet opérateur permet d'affecter un *Neurone* à un autre *Neurone*, en recopiant correctement *_n* et *_poids* (un peu comme dans le constructeur par copie). Son prototype est le suivant:

```
Neurone& operator= (const Neurone& other);
```

Il s'agit d'un opérateur agissant sur deux *Neurones* (l'instance courante et celui passé en paramètre) et qui:

- Vérifie que l'instance courante n'est pas la même que *other*: on vérifiera si les adresses mémoires sont les mêmes ou pas. L'adresse de l'instance courante est donnée par le pointeur *this*. L'adresse d'*other* est donnée par *&other*.
- Si l'adresse n'est pas la même, alors on libérera le tableau *_poids* de l'instance courante, on affectera la nouvelle valeur à *_n*, on allouera le nouvel espace pour *_poids* et on copiera les valeurs contenues dans *other._poids* dans le tableau *_poids*.
- Enfin la valeur de retour de l'opérateur = est *Neurone&*. Il conviendra de retourner l'instance courante en utilisant *"return *this"*, car la valeur d'une opération d'affectation est la valeur affectée.

1.2 Tests de la classe *Neurone*

Indication: Pour compiler et exécuter le code de cet exercice afin de tester notre neurone, on exécutera dans le terminal

```
g++ main.cpp Data.cpp Neurone.cpp -o Neurone
./Neurone
```

2 Exercice sur les fonctions et classes Template [à commencer]

Indication: Pour compiler et exécuter le code de cet exercice, on exécutera dans le terminal

```
g++ main_sort.cpp -o main_sort
./main_sort
```

On se propose d'implémenter un algorithme de tri sur des tableaux de type générique. Pour cela, on va utiliser un des algorithmes de tri les plus simples (loin d'être le plus performant !) appelé "tri à bulle" (déjà cité à la fin du TP8). Le pseudo code de cet algorithme est disponible dans le lien suivant (c'est le 1er algo de la page):
Source Wikipédia : `http://fr.wikipedia.org/wiki/Tri_à_bulles`

2.1 Fonction *MySwap*

- Pour cela, dans un fichier *MySort.hpp*, commencer à écrire une fonction patron *MySwap* sur un type générique T, prenant en paramètre deux références de ce type T, et échangeant leur valeur.
- Tester dans une fonction *main* définie dans le fichier *main_sort.cpp*, sur au moins deux types.

2.2 Fonction *MySort*

- Ecrire une fonction *MySort* implémentant l'algorithme présenté dans le lien Wikipédia. Ce sera une fonction template sur un type générique T, prenant en paramètre un pointeur sur ce type, ainsi que la longueur de ce tableau (un entier).

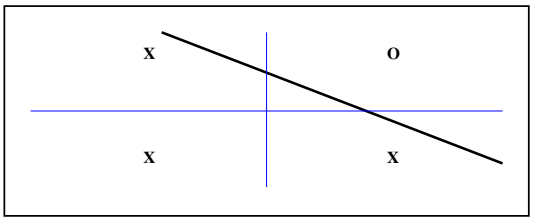
- Tester cette fonction de tri dans le fichier *main_ sort.cpp* avec des tableaux de type divers : un tableau d'entiers, de réels et de chaînes de caractères (*string*) (pour les *string*, on ajoutera #include<string> au début du fichier contenant le *main*).

Indication: Pour comparer deux *string*, les premières lettres sont utilisées et la comparaison s'appuie sur l'ordre alphabétique. De plus, toute majuscule est considérée plus petite que toute autre minuscule.

Annexe (à lire seulement)

Rappel du problème

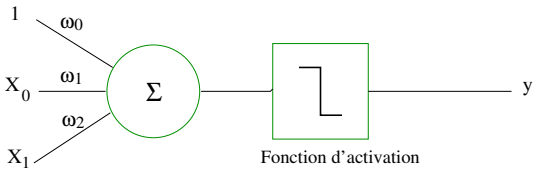
Soit un ensemble de points dans l'espace. Les coordonnées des points x et y sont les "variables descriptives".
Les points ont comme valeur possible 1 ou -1, ce sont les classes possibles du problème. Si on représente les 1 par des X et les -1 par des O, on aurait par exemple un ensemble de points comme ci-dessous.



Le discriminateur doit donc partitionner l'espace, linéairement dans cet exemple, de manière à séparer les O et les X. Un exemple de séparation est le trait en gras sur le schéma.
Dans un exemple plus concret, on pourrait voir les variables descriptives, par exemple comme la longueur et le bruit émis par un véhicule, et chercher par exemple à discriminer, en fonction de ces entrées, si le véhicule est un camion ou autre chose (voiture, moto etc.)

Un neurone

Un neurone se présente sous la forme schématisée suivante :



Formellement, en appelant la fonction d'activation f, on a la formule suivante:

$$y = f\left(\sum_{i=0}^{n-1} X_i\,\omega_i\right)$$

où X_i est la i-ème entrée du neurone. Par convention $X_0 = 0$. Et ω_i est le i-ème poids de l'entrée i.
La fonction d'activation choisie dans ce TP est la fonction suivante :

$$f(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ -1 & \text{si } x < 0 \end{cases}$$

Mise en place

Différents fichiers sont mis à votre disposition sur Moodle: *main.cpp*, *Data.hh* et *Data.cpp*. Il conviendra de copier le contenu de ces fichiers dans un répertoire donné de votre environnement de travail (par ex: TP10)

Le jeu de données est modélisé par une classe *Data* qui vous est fournie. Ce jeu de données comprend deux variables, appelées les descripteurs d'un élément du jeu. Chaque donnée peut valoir 1 ou -1.
Dans l'exemple de la fonction *main* donnée, ce jeu est rempli avec 4 points.
-1, -1, -1, 1, 1, -1 et 1,1 chacun ayant pour valeur 1 ou -1.
Le nombre d'entrées du neurone correspond au nombre de variables descriptives du jeu de données plus une qui sera alimentée systématiquement par une entrée fictive valant 1.
Avec un neurone, il est possible de discriminer linéairement un jeu de données ayant deux classes de valeurs possibles.
Un fichier *main.cpp* contenant la fonction *main*, ainsi que deux fonctions servant à l'apprentissage du neurone sont fournis.
L'algorithme d'apprentissage repose sur la règle du Delta :

- Initialiser les poids du neurone aléatoirement.
- Tirer au hasard un jeu de données X (dans ce TP, il s'agira donc d'un tableau contenant trois valeurs). On notera y la valeur correspondant à ce jeu de données (1 ou -1).
- Si $y * N(X) < 0$ alors cela veut dire que X est mal classé et on modifie le neurone selon la règle suivante: ($*$ désigne la multiplication habituelle)

  $$W_{t+1} = W_t + \lambda * y * X$$

  où W_t et W_{t+1} sont les poids du neurone avant et après itération respectivement. Il s'agit d'une somme terme à terme des poids avec un vecteur de réels. λ est choisi dans le code arbitrairement à 0.001.
  La notation $N(X)$ signifie que l'on a appliqué les données X à notre neurone. La sortie est donc de type *double*.
  $\lambda * y * X$ est un produit terme à terme du vecteur X par le réel $\lambda * y$.
- Test : Si le jeu de données X est bien classé quel que soit X, alors l'algorithme a convergé et on peut terminer.