Spécifications techniques
Implémentation d'un réseau de neurones

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

Chapter 1

Documentation des interfaces de chaque module

(voir les .html pour l'interface graphique dans le dossier *DOXYGEN HTML* et les .hpp dans le dossier *CodeDoxygen.hpp* pour l'implémentation des codes)

Chapter 2

Les tests unitaires

2.1 module Interface:

Cette classe a deux méthodes importantes, qui peuvent générer des erreurs donc il y a deux tests à effectuer :

2.1.1 testLectureFichier

prototype de la méthode: listeParamètres lectureFichier(String 'nomFich.csv')

Nous allons d'abord tester la fonction lectureFichier(). Pour cela, nous allons définir le jeu de données que nous allons utiliser ainsi que les résultats attendus.

Tout d'abord, faisons un point sur la manière de remplir un fichier des paramètres du futur réseau à construire :

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

1. **Test avec un fichier vide:** Entrer en paramétres un fichier en format .txt vide.
   La méthode doit renvoyer un message d'erreur sur la console pour prévenir l'utilisateur, puis revenir sur le menu précédent.
   Renvoie 0 comme message d'erreur.

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

3. **Test avec un fichier mal rempli:** Plusieurs tests sont envisageables. Voyons en détails :

   Fichier test 1: (2 1 3 2 5 2 0) renvoie un message pour dire que le fichier est mal rempli (exemple : réseau non implémenté).
   Retourne une liste avec la valeur -1.

4. **Test avec un fichier bien écrit:**

   Fichier test 1: (1 1 3 2 5 2 0) renvoie une liste de paramètres=(1 1 3 2 5 2 0).

2.1.2 testlectureParam();

prototype de la méthode: listeParamètres lectureParam()

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

1. Test quand l'utilisateur entre de mauvais paramètres (par exemple entre des string au lieu d'entier etc.. problème de type)

   (a) Test 1: saisie du premier paramètre: si $param1 = a$ erreur de type, si $param1 = 4$ erreur réseau non implémenté, (propose une resaisie, ou quitter).

   (b) Test 2: saisie du second paramètre: avec $param1 = 1$ si $param2 = a$ erreur de type, si $param2 = 4$ erreur cas d'utilisation non implémenté, (propose une resaisie, ou quitter).

   (c) Test 3: saisie du troisième paramètre: avec $param1 = 1$, $param2 = 2$, si $param3 = a$ erreur de type.

(d) Test 4: saisie du quatrième paramètre: avec $param1 = 1$, $param2 = 2$, $param3 = 4$, si $param4 = 4\ 5\ 6\ 9\ 5\ 7$, le programme ne sauvegarde que les 4 premier entiers.

(e) Test 5: saisie du quatrième paramètre: avec $param1 = 1$, $param2 = 2$, $param3 = 4$, si $param4 = 4\ 5$, erreur trop couches sans neurones.

(f) Test 6: saisie du quatrième paramètre: avec $param1 = 1$, $param2 = 2$, $param3 = 4$, $param4 = 4\ 5\ 6\ 1$, si $param5 = a$ erreur de type.

2. Test quand l'utilisateur entre de bons paramètres:

   (a) Test 1: saisie suivante: $param1 = 1$, $param2 = 1$, $param3 = 4$, $param4 = 4\ 5\ 6\ 1$, $param5 = 0$ renvoie la liste: $(1\ 1\ 4\ 4\ 5\ 6\ 1\ 0)$.

ajouter -lcCppUnit

2.2 module Couche

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

2.3 module CoucheEntree

Cette classe a une seule méthode importante, qui peut générer des erreurs donc le seul test à effectuer est :

2.3.1 testConstructionSortie

prototype de la méthode: void constructionSortie()

Soit un fichier de données test qui contient les 4 valeurs utilisées dans l'exemple précédent.

il faut vérifier que la fonction constructionSorite remplit un tableau de neurone de dimension 4 comme suit :

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

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

2.4 module CoucheSortie et CoucheCachee

Comme vous allez le constater, nous avons fait le choix d'implémenter les fonctions preActivation() et activation() dans la classe CoucheCachée mais aussi

dans CoucheSortie. Cela nous permet en fait de pouvoir modifier la fonction activation qui, bien souvent n'est pas la même selon si c'est une couche cachée ou de sortie.

2.4.1 testPreactivation

prototype de la méthode: double[] preActivation()

Il faut tester si elle renvoie bien la somme des produits de la valeur du neurone avec le poids qui le relie au neurone étudié ajouté au biais de ce neurone.

Pour l'exemple des iris de Fisher :

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

⇒ Il y a alors 8 poids et 2 biais

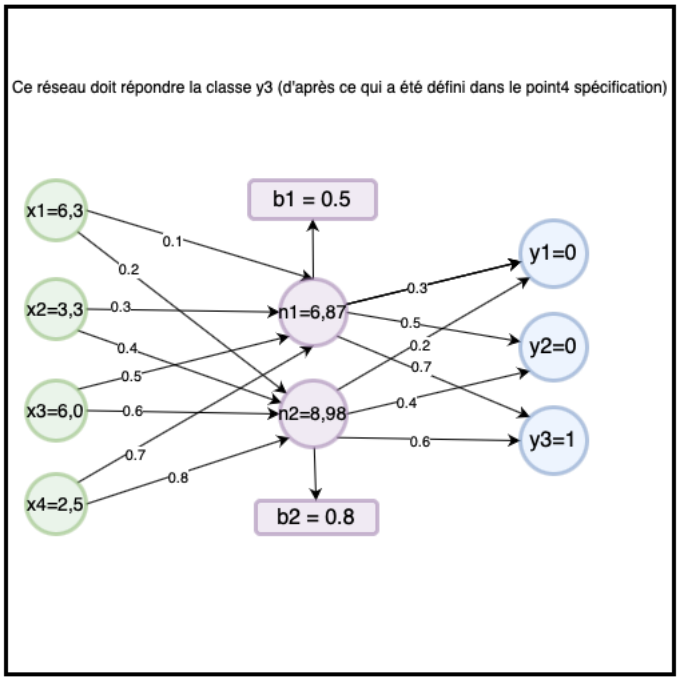

Figure 2.1: Exemple des iris de Fisher

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

6,87	8,98

2.4.2 testActivation

prototype de la méthode: double activation(double[])

Nous allons considérer le cas de la fonction sigmoïde donnée par :

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

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

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

2.4.3 testConstructionSortie

prototype de la méthode: void constructionSortie( )

2.5 Module Reseau

2.5.1 testConstructeurReseau

Pour créer un réseau, les choix sont multiples. Comme on l'a vu plus haut dans la partie InterfaceUtilisateur, c'est par le biais d'un fichier au bon format que l'utilisateur peut obtenir un type de réseau particulier.
Réseau(Liste listeParametres) est le constructeur de la classe Reseau et prend en paramètres la liste *listeParametres* qui contient :

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

On prend alors 2 exemples :
Un réseau R1 de type Forwarded avec 3 couches cachées, 2 neurones par couches et des poids initialisés aléatoirement.
Un réseau R2 de type Forwarded avec 1 couches cachées, 4 neurones par couches et des poids initialisés à 0.5.

Remarque : Il est impossible de calculer l'égalité entre 2 flottants en machine. On utilise pour parer cela : la norme de la différence entre la valeur attendue et la valeur recupérée dans le fichier pour obtenir le booléen qui indique si le test est réussi ou non.
C'est cette technique que l'on utilise ici pour le test concernant les poids initialisés à 0.5.

2.5.2 testAjouterCouche

2.5.3 testErreur

2.5.4 calcGradC

2.5.5 testApprentissage

2.5.6 testBackPropagation

2.6 Module ResForwarded

Pas de test à effectuer

2.7 Module ResRecurrent

Pas de test à effectuer

2.8 Module Matrice

Les tests sur la classe Matrice sont de 2 types :
- Fournit le bon résultat
- Traite le cas où les matrices ne sont pas de la même taille et vérifie si le nombre colonne est égal au nombre de lignes de la matrice passé en paramètre.

2.8.1 testProduit

prototype de la méthode testProduit(Matrice)

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

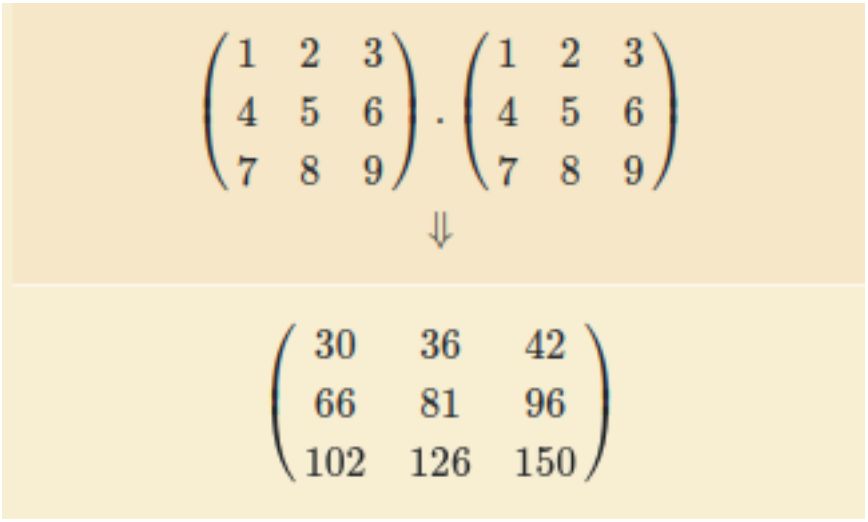

Figure 2.2: 1ère partie de test

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

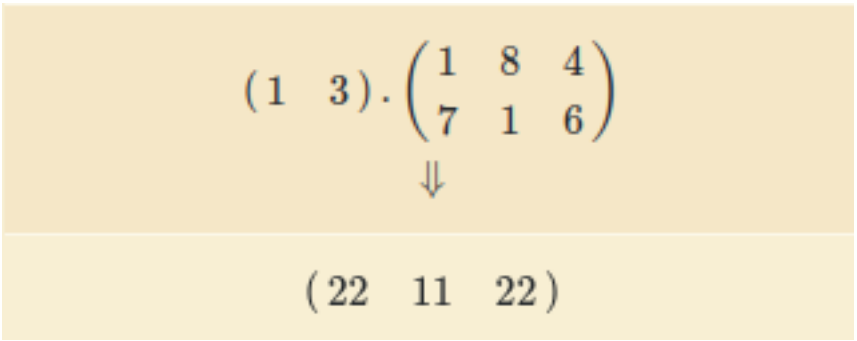

Figure 2.3: 2ème partie de test

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

2.9 Module Neurone

Pas de test à effectuer