Projet : un réseau de neurones simple : le perceptron multicouches

Notions : Développement logiciel, allocation dynamique

1 Avant propos

Le projet clôture le premier semestre. Il met en oeuvre l'ensemble des concepts que vous avez vu tout au long du semestre : tableau, fichiers, structures, allocation dynamique. Il ajoute une dimension : travailler en équipe, en utilisant bien sur le gestionnaire de version git.

Ce projet se réalise donc en binôme, que vous formez avant le début du projet. Lisez le sujet et préparez vos questions pour la première séance qui est encadrée par un enseignant. Vous gérez ensuite votre projet à 2 en autonomie et posez vos questions de préférence sur l'outil `riot.ensimag.fr` aux enseignants. Vous déposerez sur `gitlab.ensimag.fr` votre code. Les 2 séances prévues dans l'emploi du temps ne sont pas suffisantes pour réaliser ce projet, qui nécessite du travail supplémentaire. Il n'y a pas de rapport à rendre, uniquement votre code.

Quelques conseils pour travailler en binôme

- Commencez par analyser le problème posé, définissez les structures de données dont vous avez besoin et/ou complétez celles que nous vous suggérons, définissez les fonctions, leur rôle et leur prototype ensemble. Ensuite, répartissez vous les fonctions à écrire en indiquant dans quels fichiers elles se trouvent.
- Travailler de manière incrémentale : écrivez une première fonction, compilez cette fonction, testez cette fonction avec un programme principal `main` spécifique à cette fonction. Déposer le fichier sur `gitlab.ensimag.fr`. Passez ensuite à une autre fonction.
- Utilisez les outils de debug pour traiter les cas de `segmentation fault`, du à des accès mémoire inadéquats : `gdb`, `ddb` et `valgrind` dont vous trouverez une initiation sur le site :http://tdinfo.phelma.grenoble-inp.fr/1AS1/site.
- Evitez de travailler à 2 sur les mêmes fichiers. Bien que cela soit possible, cela peut générer des conflits au moment de déposer votre code sur git qu'il faut ensuite gérer correctement. Cela demande un peu d'habitude avec `git`.
- Déposer régulièrement votre code sur gitlab, et faites aussi régulièrement un `git pull` pour récupérer le travail de votre binôme.

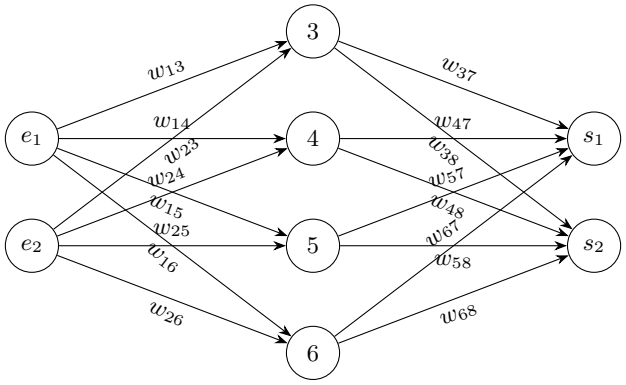
2 Introduction

L'Intelligence Artificielle s'est développée avec pour objectif la simulation des comportements du cerveau humain. Les premières tentatives de modélisation du cerveau datent de 1943 avec les premières notions de neurone formel. Ce concept fut ensuite mis en réseau avec une couche d'entrée et une sortie par Rosenblatt en 1959 pour reconnaître des formes : c'est le perceptron. Compte tenu de la puissance de calcul de l'époque et des limites théoriques de ce modèle (incapable de reproduire un OU exclusif), l'approche a été mise en sommeil jusque dans les années 1980.

Les avancées technologiques et l'algorithme de retro-propagation ont relancé le perceptron multi-couches (MLP) (figure 1) à ce moment. C'est un type de réseau de neurones artificiel formé de plusieurs

couches. L'information circule de la couche d'entrée vers la couche de sortie uniquement. Chaque couche est constituée d'un nombre variable de neurones, les neurones de la dernière couche de sortie sont les sorties du réseau.

FIGURE 1 – Un réseau avec 2 entrées e_1 et e_2, 2 sorties s_1 et s_2 et une couche cachée de 4 neurones



Pour mettre en place ce type d'outil d'apprentissage supervisé, il y a deux étapes :

1. apprentissage à partir d'un grand ensemble de données déjà classées, pour régler les poids qui lient les différents neurones d'une couche aux neurones de la couche suivante
2. Utilisation du réseau dont les poids ont été appris à de nouvelles données

Le projet consiste à programmer un des réseaux de neurones les plus simples, le perceptron, composé de plusieurs couches de neurones. La couche 0 est la couche d'entrée. La couche n est la couche de sortie. Tous les neurones d'une couche i sont reliés à la couche inférieure $i-1$, exceptée la couche d'entrée 0.

Un exemple classique et simple est la classification des iris de Fisher.

3 Le perceptron et la classification

3.1 Le neurone formel

Un neurone est composé de :

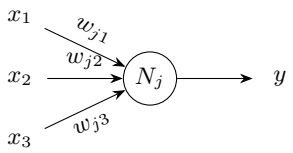
— les n entrées x_i, correspondant aux sorties des neurones de la couche inférieure.

— une sortie y

— un poids w_i à appliquer à l'entrée x_i lors du calcul de la valeur de sortie du neurone. Ce poids est celui de la connexion entre ce neurone et ceux de la couche inférieure.

— une fonction d'activation f. Cette fonction sera ici une sigmoide $f(x) = \frac{1}{1+e^{-x}}$

FIGURE 2 – Un neurone



Lorsque l'on applique un signal ou une valeur aux entrées x_i du neurone j, il est activé et la valeur y de sortie du neurone est définie par :

$$y = \frac{1}{1 + e^{-\sum_{i=1}^{n} w_{ji} \times x_i}} \tag{1}$$

3.2 Le réseau et la classification

Un réseau peut être utilisé pour la classification. Voyons cela sur l'exemple des iris de Fisher.

les Iris de Fisher correspondent à 150 fleurs décrites par 4 variables quantitatives : longueur du sépale, largeur du sépale, longueur du pétale et largeur du pétale. Les 150 fleurs sont réparties en 3 différentes espèces : iris setosa, iris versicolor et iris virginica.

A partir de mesures réalisées sur une fleur inconnue, comment trouver le type d'iris à l'aide d'un réseau de neurones ? Mathématiquement, il s'agit de construire une partition de $\mathbb{R}^4$ en 3 sous-ensembles disjoints. Une fleur correspond à un point de $\mathbb{R}^4$, donné par ses 4 mesures. A quelle espèce appartient ce point ?

Pour créer un réseau, il faut définir le nombre de couches, le nombre de neurones de chaque couche. La couche 0 ou entrée contient donc le même nombre de neurones que le nombre de caractéristiques ou mesures, la couche de sortie contient le nombre de classes différentes. Dans le cas des iris de Fisher, il y a donc 4 entrées et 3 sorties.

Si l'apprentissage a été réalisé (voir la section 4), les poids des connexions entre les neurones sont définis et pour trouver le type d'une fleur dont on connait les 4 mesures, il suffit de :

1. mettre en entrée du réseau les 4 mesures de la fleur examinée,
2. calculer la valeur de sortie de chaque neurone de la couche 1
3. calculer la valeur de sortie de chaque neurone de la couche 2
4. calculer la valeur de sortie de chaque neurone de la couche i
5. prendre la valeur de sortie maximale obtenue sur la dernière couche comme classe de cette fleur

3.3 Travail à réaliser

Le travail à réaliser dans cette partie consiste à utiliser des réseaux déjà appris dont la configuration est stockée dans un fichier et à classifier un ensemble de données en utilisant ce réseau.

Il faut donc construire un réseau comportant n couches de neurones à partir d'un fichier contenant ce réseau. Un autre fichier contient des mesures correspondant à des fleurs : pour trouver le type de chaque fleur, il faut lire les mesures, les mettre en entrée du réseau, calculer le résultat en propageant l'information de l'entrée vers la sortie couche par couche, prendre la sortie maximale parmi les neurones de sortie et afficher le résultat.

3.3.1 Réseau et structures de données

Pour construire les réseaux, nous avons besoin d'une structure pour représenter un neurone. Une couche de neurones est un tableau de neurones et le réseau de tableau de couches. Tous les tableaux doivent être alloués dynamiquement. Les types de données sont donc les suivants :

```
  /* Le type definissant un neurone */
typedef struct Neurone {
  double  x;      /* la valeur de sortie du neurone */
  double* w;     /* tableau des poids des connexions entre ce neurone et ceux de la couche inferieur */
} t_neurone;

  /* Le type definissant une couche */
typedef struct Couche {
  int     nombreNeurones;   /* Nombre de neurones de la couche */
  t_neurone* tabNeurones; /* Tableau des neurones de la couche */
} t_couche;

  /* Le type definissant un reseau */
typedef struct Perceptron {
  int    nombreCouches;   /* Nombre de couches du reseau, en comptant les couches d'entree et de sortie
    */
  t_couche* tabCouches;   /* Tableau des couches de neurones */
} t_perceptron;
```

Pour créer un réseau, il faut donc d'abord connaitre le nombre de couches et allouer dynamiquement un tableau de couches (le champ `tabNeurones`) d'une variable de type `t_perceptron`.

Ensuite, il faut connaitre le nombre de neurones de chaque couche et pour chaque couche d'une variable de type `t_perceptron`, allouer dynamiquement un tableau de couches (le champ `tabCouches`).

Toutes ces informations sont contenues dans un fichier décrit section 3.3.5, avec les poids du réseau appris.

Pour construire le réseau, il faut donc :

— ouvrir le fichier,
— lire les informations de la configuration du réseau (nombre de couches, nombre de neurones de chaque couche),
— allouer les tableaux nécessaires
— lire tous les poids des neurones dans le bon ordre et les mettre dans le réseau
— fermer le fichier

Cette fonction aura pour prototype :

```
t_perceptron* perceptronLecture( char* fichier);
```

Remarque : la couche d'entrée ou couche 0 n'a pas de poids, puisque non reliée à une couche inférieure.

3.3.2 Classification

Une fois le réseau construit, il faut pouvoir classifier un exemple (i.e. 4 mesures extraites du fichier des données sur les Iris de Fisher) à l'aide de ce réseau. Pour cela, il faut :

— pour toutes les couches i ($i \geq 1$) du réseau
  — parcourir les neurones de la couche i et mettre à jour leur sortie à l'aide de l'équation 1
— trouver le neurone de sortie qui a la plus forte valeur
— retourner l'indice de ce neurone (l'indice correspondra à sa classe)

Cette fonction suppose que les valeurs de sortie (champ `x`) des neurones de la couche d'entrée contiennent les mesures relatives à une fleur. Cette fonction aura pour prototype :

```
int perceptronPropagation(t_perceptron* p);
```

3.3.3 Test du réseau sur un ensemble de données

Les mesures relatives à une fleur sont stockées dans un fichier de données (voir section 3.3.6). Il faut donc lire une ligne de ce fichier qui contient les mesures. Ce fichier contient aussi les sorties attendues du réseau, afin de pouvoir vérifier l'exactitude du résultat. Dans une application réelle, ces sorties attendues ne seraient bien sûr pas disponibles. Elles ne sont ici que pour donner un score de réussite du réseau.

Tester le réseau sur l'ensemble des données disponibles se fera donc de la manière suivante :

— Ouvrir le fichier de données
— Tant qu'on n'est pas à la fin du fichier
  — lire les mesures et les résultats attendus d'une fleur, ie une ligne du fichier
  — mettre les mesures dans la couche d'entrée du reseau
  — propager l'information avec la fonction `perceptronPropagation` précédente
  — afficher le type correspondant à cette fleur
  — si le type n'est pas correct, afficher l'indice de cette fleur et la couche de sortie du réseau
  — calculer l'erreur globale commise par le réseau (somme des carrés des écarts entre sortie attendue et sortie obtenue)
— afficher l'erreur globale commise
— fermer le fichier

Cette fonction aura pour prototype :

```
void perceptronTest(t_perceptron* p,char* fichier);
```

3.3.4 Autres fonctions utiles

Vous pourrez avoir besoin d'autres fonctions comme :

`perceptronDestruction(t_perceptron* p);` : destruction du réseau avec libération mémoire de tous les tableaux alloués.

`perceptronAffichage(t_perceptron* p);` : affichage des poids et/ou des sorties des neurones du réseau

3.3.5 Les poids du réseau

Nous fournissons des fichiers contenant les poids de réseaux ayant déjà été appris sur les données. Le format de ce fichier est le suivant :

- Sur la première ligne, le nombre de couches du réseau suivi de la chaîne " : Nombre de couches"
- Sur la deuxième ligne, le nombre x de neurones de la couche 0 du réseau suivi de la chaîne " : Nombre de neurone de la couche 0"
- Sur la troisième ligne, le nombre y de neurones de la couche 1 du réseau suivi de la chaîne " : Nombre de neurone de la couche 1"
- ...
- ...pour toutes les couches
- une chaine contenant "Poids du neurone 0 de la couche 1"
- les x réels correspondant aux poids reliant le neurone 0 de la couche 1 aux neurones de la couche 0
- une chaine contenant "Poids du neurone 1 de la couche 1"
- les x réels correspondant aux poids reliant le neurone 1 de la couche 1 aux neurones de la couche 0
- ....
- ... pour tous les neurones de toutes les couches

```
3 : Nombre de couches
4 : Nombre de neurone de la couche 0
3 : Nombre de neurone de la couche 1
3 : Nombre de neurone de la couche 2
Poids du neurone 0 de la couche 1
-1.100405 -3.059170 4.695029 1.976174
Poids du neurone 1 de la couche 1
-15.597073 -34.393907 32.496463 25.175091
Poids du neurone 2 de la couche 1
18.336535 -7.513683 -16.874983 -9.708140
Poids du neurone 0 de la couche 2
-13.557511 -3.526711 6.814055
etc....
```

3.3.6 Les fichiers de données

Le fichier qui contient les données à tester est au format suivant :

- un entier qui est le nombre d'entrées suivi de la chaîne de caractères "// nombre d'entrees"
- un entier qui est le nombre de sorties suivi de la chaîne de caractères "// nombre de sorties"
- les réels correspondant aux entrées puis les réels correspondant aux sorties attendues du premier exemple
- les réels correspondant aux entrées puis les réels correspondant aux sorties attendues du deuxième exemple
- ... jusqu'au dernier exemple

```
4  // nombre d'entrees
3 // nombre de sorties
5.1000000e+00   3.5000000e+00   1.4000000e+00   2.0000000e-01   1.0000000e+00   0.0000000e+00
   0.0000000e+00
4.9000000e+00   3.0000000e+00   1.4000000e+00   2.0000000e-01   1.0000000e+00   0.0000000e+00
   0.0000000e+00
4.7000000e+00   3.2000000e+00   1.3000000e+00   2.0000000e-01   1.0000000e+00   0.0000000e+00
   0.0000000e+00
```

3.3.7 Les exemples disponibles

Un premier jeu de données concerne les iris de Fisher et date de 1936. Le jeu de données comprend 50 échantillons de trois espèces d'iris (Iris setosa, Iris virginica et Iris versicolor). Quatre caractéristiques ont été mesurées : la longueur et la largeur des sépales et des pétales. Le nom du fichier est *iris.txt*. Deux réseaux sont proposés : un premier réseau avec 1 couche cachée de 3 neurones avec une erreur de 4 fleurs mal classées, fichier *reseau-iris-3.txt* et un deuxième réseau avec 2 couches cachées de 5 et 3 neurones respectivement fichier *reseau-iris-5-3.txt* avec une erreur de 1 fleur mal classée.

Un deuxième jeu de données concerne la base de données MNIST de chiffres manuscrits . C'est une base de données de chiffres écrits à la main (figure 3). Elle regroupe 60000 images d'apprentissage (fichier *mnist-train.txt*) et 10000 images de test (fichier *mnist-test.txt*). Ce sont des images en niveau de gris, normalisées centrées de 28 pixels de côté. Chaque exemple contient donc 784 mesures et 10 classes possibles (chiffres 0 à 9), soit 794 données par ligne. Chaque mesure est le niveau de gris d'un pixel, normalisé entre 0 et 1. Deux réseaux sont proposés : un premier réseau avec 1 couche cachée de 400 neurones et une erreur de classification de 176 chiffres sur le fichier de tests, fichier *reseau-mnist-400.txt* et un deuxième réseau avec 2 couches cachées de 80 et 20 neurones respectivement et une erreur de classification de 231 chiffres, fichier *reseau-mnist-80-20.txt*.

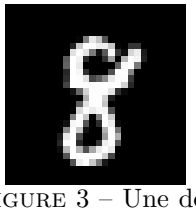


FIGURE 3 – Une des images

4 Facultatif : pour aller plus loin, l'apprentissage du réseau

Si vous avez réalisé la première partie, vous pouvez aborder l'apprentissage des poids du réseau. Il utilise ici l'algorithme de rétro-propagation.

Nous disposons d'un ensemble X d'exemples dont nous connaissons la classe Y. En prenant un réseau dont les poids sont initialisés au hasard et en mettant un exemple en entrée X_p, la sortie du réseau pour cet exemple contient donc des erreurs par rapport aux sorties attendues Y_p. L'algorithme de rétro-propagation minimise cette erreur par rapport aux poids du réseau en dérivant cette erreur :

1. remontant l'erreur de la couche de sortie jusqu'à la couche d'entrée : chaque neurone de chaque couche contient ainsi une erreur calculée à partir de la couche supérieure. Pour chaque neurone, l'erreur est la somme des erreurs de chaque neurone de la couche suivante, pondérée par le poids qui le lie au neurone dont on calcule l'erreur et par la dérivée de la fonction d'activation.
2. ajustant les poids du réseau de manière à minimiser cette erreur (descente de gradient)

Il faut réaliser ces étapes pour tous les exemples connus et itérer toute cette procédure de nombreuses fois jusqu'à ce que le réseau converge vers une erreur totale minimale. Cette convergence n'est cependant jamais assurée et dépend des données et des paramètres de l'algorithme.

4.1 Algorithme

En notant X_p les exemples et Y_p les sorties attendues, x_k^i la valeur de sortie du neurone k de la couche i, w_{kj}^i le poids de la connexion entre le neurone k de la couche i et le neurone j de la couche inférieure $i-1$, l'algorithme s'écrit de la manière suivante 1 :

Algorithm 1 Apprentissage du réseau

1: **repeat**
2: **for** tous les couples exemples/sorties (X_p, Y_p) **do**
3: Propager l'information de la couche d'entree jusqu'a la couche de sortie (activer le reseau)
4: **for** tous les neurones k de la couche de sortie n **do**
5: Calculer l'erreur de chaque neurone de la couche de sortie $\delta_k^n = x_k^n \times (1 - x_k^n) \times (Y_{pk} - x_k^n)$
6: **end for**
7: **for** Toutes les couches cachées j, de $j = n - 1$ à $j = 0$ **do**
8: **for** tous les neurones k de la couche cachée j **do**
9: Calculer l'erreur $\delta_k^j = x_k^j \times (1 - x_k^j) \times \sum_{i \,:\, neurones\ couche\ j+1} \delta_i \times w_{ik}^j$
10: **end for**
11: **end for**
12: **for** Toutes les couches j, de $j = 0$ à $j = n$ **do**
13: **for** tous les neurones k de la couche j **do**
14: Ajuster les poids $w_k^j = w_k^j + \eta \times \delta_k^j \times (1 - x_k^j)$
15: **end for**
16: **end for**
17: **end for**
18: Calculer l'erreur totale de la couche sortie n du reseau $E = \sum_{k \,:\, neurones\ couche\ n} (Y_{pk} - x_k^n)$
19: **until** $E \leq \epsilon$ ou nombre d'itérations maximale

Le paramètre η est le pas de l'algorithme de descente du gradient de la rétro-propagation : s'il est petit, le réseau est très long à converger, s'il est trop grand, le réseau ne converge pas. Il est en général compris entre 0 et 1.

Lorsque l'on augmente le nombre de couches, l'algorithme d'apprentissage nécessite de plus en plus d'itérations pour converger vers un résultat. On dépasse donc rarement deux couches cachées dans un perceptron multi-couche.

Attention : les temps de calcul de l'apprentissage deviennent rapidement importants (plusieurs heures) dès que le nombre de neurones augmentent.

4.2 Travail à réaliser

Mettre en place cet algorithme pour apprendre les poids d'un réseau avec les données disponibles. Pour un exemple issu d'un fichier de test, dont les valeurs d'entrées sont déjà dans le réseau `p` et dont les sorties sont données par le tableau `cible`, vous pouvez écrire une fonction qui réalise les étapes 2 à 17 de l'algorithme 1.

```
void perceptronEntrainementUnExemple(t_perceptron* p, double* cible)
```

Il est conseillé de décomposer ces étapes en plusieurs fonctions comme par exemple :

— la propagation, déjà disponible

— le calcul de l'erreur de la sortie par rapport aux sorties désirées :

```
void perceptronErreur(t_perceptron* p, double* cible);
```

— le calcul de l'erreur des couches cachées par rétropropagation du gradient (étapes 7 à 11) :

```
void perceptronRetroPropagation(t_perceptron* p);
```

— l'ajustement des poids (étapes 12 à 17) :

```
void perceptronAdjustePoids(t_perceptron* p);
```

Pour éviter de devoir toujours faire un apprentissage, écrire une fonction qui sauvegarde le réseau `p` dans un fichier de nom `fichier` au format décrit en section 3.3.5. Le prototype de cette fonction sera :

```
int perceptronSauve(t_perceptron* p, char* fichier);
```

4.2.1 Variante

Vous pouvez ajouter un neurone supplémentaire pour chaque couche, qui est un neurone de biais. Il est relié comme les neurones de sa couche j à tous les neurones de la couche $j+1$, mais n'est relié à aucun neurone de la couche $j-1$. Son entrée est toujours à 1. Il est traité comme les autres neurones pour calculer l'erreur et corriger les poids entre lui et les neurones de la couche $j+1$.

4.3 Sur-apprentissage du réseau

Les réseaux dépendent de paramètres comme le nombre de couches, de neurones, d'itérations maximales. Il est toujours possible d'arriver à une réseau qui apprend parfaitement ces poids sur un ensemble de données (i.e. classifie parfaitement ces données) mais qui se comporte très mal avec de nouvelles données : c'est le phénomène de sur-apprentissage. Le réseau est incapable de généralisation et ne classifie que ce qu'il a appris.

Pour éviter ce phénomène, il est classique de séparer les données d'apprentissage en 2 parties : un ensemble d'apprentissage et un ensemble de validation. L'erreur totale faite par le réseau sur l'ensemble d'apprentissage décroit avec la complexité du réseau (les paramètres tels que nombre de couches, de neurones, d'itérations maximales). Si on calcule l'erreur sur l'ensemble de validation, cette erreur décroit pendant un certain temps avec la complexité du réseau, puis se met à croitre à partir du moment où le réseau commence à être trop *spécialisé* et à sur-apprendre. C'est par exemple le cas du réseau proposé sur les données des iris qui sont trop peu nombreuses pour 2 couches cachées.

Pour réaliser un apprentissage correct et pouvoir comparer divers classifieurs (que ce soient des réseaux de neurones ou d'autres classifieurs), il faut donc répartir l'ensemble des données en 3 jeux : données d'apprentissage (50 % des données disponibles), données de validation (25 % des données disponibles) et données de tests (25 % des données disponibles) [1]. Les données d'apprentissage et de validation servent à faire apprendre le réseau au mieux, les données de test à comparer les performances de ce réseau avec les autres classifieurs.

5 Bibliographie

Fisher, R.A. 1936. The use of multiple measurements in taxonomic problems. Annals of Eugenics 7 : 179-188

Dua, D. Karra Taniskidou, E. 2017. UCI Machine Learning Repository. Iris Data Set. Irvine, University of California, School of Information and Computer Science. https ://archive.ics.uci.edu/ml/datasets/Iris

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11) :2278-2324, November 1998.

1. D'autres solutions existent pour la validation de manière à ne pas utiliser trop de données utiles à l'apprentissage comme la validation croisée ou K-fold.