Scala & Java

Perceptron multi couches

Vérifiez les conventions à suivre pour l'écriture du code (cf Moodle).

1. **Lire, comprendre et apprendre le cours sur la retro propagation du gradient d'erreur.**

2. **Vous allez fabriquer un simulateur de réseau connexionniste en Java ou en Scala.**

Ce travail ne nécessite ni la récursivité, ni la programmation fonctionnelle. Cependant, pour certaines parties du programme, il est possible que l'impératif classique ou le fonctionnel soient plus adaptés (et plus faciles à mettre en œuvre). L'intérêt de la programmation multi paradigme est justement de vous apporter les outils adéquats : donc choisissez ce que vous maitrisez le mieux et qui vous semble le plus adapté.

Normalement, au niveau Master, vous devez être capables de définir vous-même la structure de données, la structure objet et le découpage en méthodes ou en fonctions du problème (évidemment sans s'inspirer lourdement de solutions glanées à droite et à gauche ☺)

Ce travail s'accompagne forcément de notes sur papier, de schémas, de dessins de la structure de données et d'une compréhension préalable de ce qu'est un perceptron (cf cours).

3. **A vous de tester votre librairie. L'expérimentation guidée sera pour le TD suivant (il faut donc que la librairie fonctionne correctement pour le TD suivant).**

Suggestions pour le test :

Un exemple classique : le ET, le OU et le OU Exclusif (OR, AND, XOR)

Par exemple pour le ET :

1 .0, 1.0 -> 1.0
0.0 0.0 -> 0.0
1.0 0.0 -> 0.0
0.0 1.0 -> 0.0

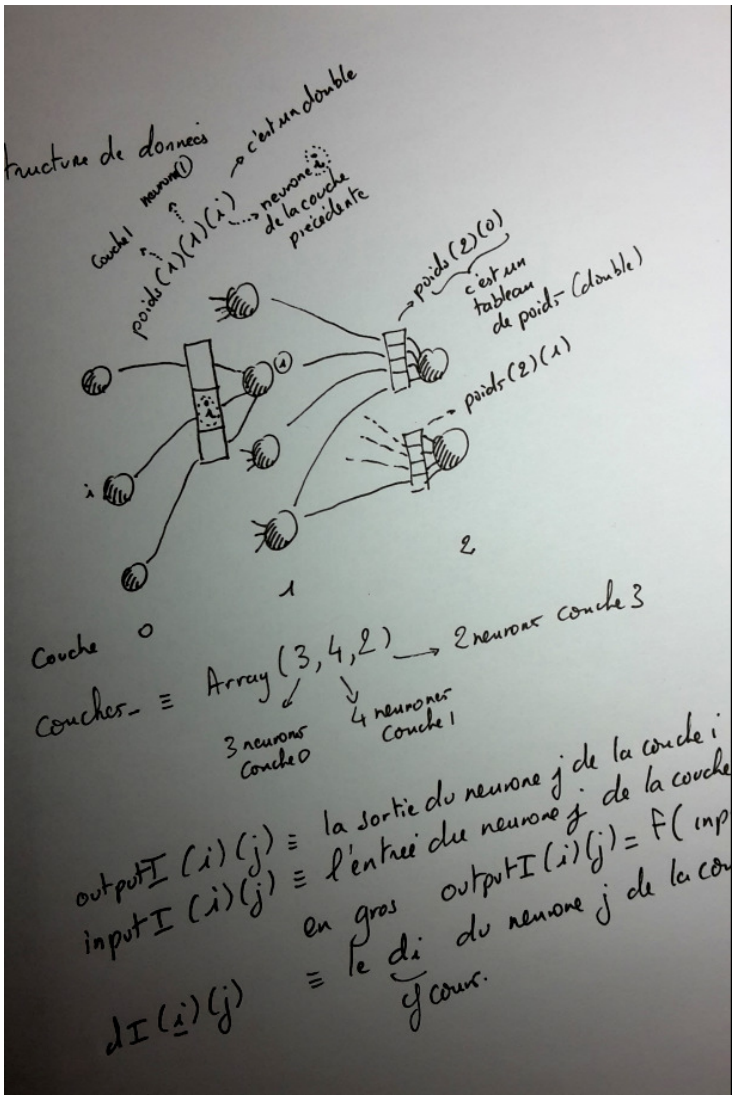
Vérifiez que l'erreur diminue pendant l'apprentissage. Est-il possible d'apprendre les 3 avec seulement la couche d'entrée et une couche de sortie (par de couche cachée) ? Pourquoi ?

Si vous n'avez aucune idée de comment commencer (ce qui, encore une fois, au niveau Master devrait vous inquiéter sérieusement), voilà quelques indices d'une réalisation possible : ce n'est pas la meilleure solution ; elle reste volontairement très proche des explications du cours. Vous n'êtes pas obligé de suivre ces indices, normalement, vous devez être capable de réaliser vous-même, sans guidage, ce travail.

Je vous explique comment, moi j'ai réalisé ce travail – et encore une fois, ce n'est pas une correction ni une solution à suivre de manière aveugle : c'est une solution très proche du cours et non optimisée.

J'utilise Scala (ça me permet de choisir entre impératif et fonctionnel en cours de route)

1ère chose : je cherche les structures de données en fonction du traitement que j'aurai à faire sur les poids. Le réseau est défini par les poids, donc ça sera ma structure la plus importante. Pour faire cela, je commence par dessiner sur une feuille de papier, mon projet de structure de données.



J'écris en francais la signification des indexes parce qu'il faut que je sois absolument certain de savoir comment atteindre les bonnes informations pour l'implémentation de l'algorithme d'apprentissage :

je n'ai pas de temps à perdre et je ne souhaite pas tenter au hasard un indice ou l'autre pour voir si ca marche (j'ai très peu de chances d'y arriver là). Il faut que ma structure de données soit très claire.

Donc le tableau poids va avoir 3 dimensions. La première est la couche : donc poids(2) renvoie une structure de données qui me permet d'avoir les poids qui atterrissent sur la couche 2. Je le fais dans ce sens car la manipulation des poids va se faire de la sortie vers l'entrée. Je prends des Arrays parce que l'accès est simple et qu'il n'est pas question ici de faire des structures de données qui contiennent des milliards de poids : d'autant plus que l'objectif est de faire fonctionner le perceptron en priorité.

Donc poids(2) me donne la structure de données qui correspond aux poids qui atterrisent sur la couche 2 (qui viennent de la couche 3 – cf le cours, on prend comme hypothèse que les seuls poids de la couche 2 concernent la couche 3). Dans la couche 3, si j'ai n neurones, et m neurones dans la couche 2, alors j'aurai un tableau de n*m poids. Donc par exemple, poids(2)(i) me donne le tableau (de taille m) des poids qui relient le neurone i de la couche 3 aux neurones de la couche 2.

De la même manière, et cette fois-ci pour simplifier l'implémentation de l'apprentissage, je créé un tableau inputI(a)(b) qui contient les entrées des neurones b de la couche a. Idem pour outputI (qui est seulement la valeur de inputI passée dans la fonction de transfert). Et enfin, dI(a)(b) qui contient les dI calculés (cf cours).

Première chose : je créé une classe Perceptron et un objet Perceptron. Dans l'objet Perceptron, je vais mettre mes méthodes de classe (celles qui ne dépendent pas de l'état d'une instance).

Par exemple, j'y mets :

```
// f fonction de transfert
def f(x_ : Double):Double = 1/(1+Math.exp(-x_))

// f'
def fp(x_ : Double):Double = f(x_)*(1-f(x_))
```

En théorie, si un jour mon programme doit évoluer, je me dis que ça serait bien d'en faire une variable fonction (comme ça l'utilisateur pourra changer la fonction), mais dans l'immédiat, je fige la fonction de transfert pour les tests.

En ce qui concerne l'activation du réseau, je fais une opération très souvent : je multiplie chaque entrée par un poids puis je fais la somme des résultats. C'est très exactement un produit scalaire entre un vecteur d'entrée et un vecteur qui contient des poids. J'utilise des tableaux, donc, je rajoute une méthode pour faire un produit scalaire entre deux tableaux :

```
 // produit scalaire
 def prod(p1_: Array[Double], p2_: Array[Double]): Double = {
  require(p1_.length == p2_.length, "pour le produit les vecteurs doivent avoir la même taille")
  p1_.zip(p2_).map{ case (a,b) => a*b }.sum
 }
```

Notez qu'en ce qui me concerne, je préfère ici, utiliser du fonctionnel pour faire le boulot. Mais on peut tout à fait utiliser une variable d'accumulation et une boucle pour multiplier élément par élément, puis sommer ensuite.

Faites tourner le code précédent sur un exemple simple pour voir comment est utilisé zip ici.

Comme je souhaite éviter les problèmes de tableaux qui n'ont pas la même taille, j'indique une précondition (require) pour bloquer ce problème (et éviter de gérer une exception).

Je rajoute le calcul de l'erreur quadratique :

```
// calcul d'erreur quadratique
 def errQuad(p1_: Array[Double], p2_:Array[Double]):Double = {
  require(p1_.length == p2_.length, "pour l'erreur quadratique les vecteurs doivent avoir la même taille")
  p1_.zip(p2_).map{ case (a,b) => (a-b)*(a-b)}.sum
 }
```

Et enfin, je souhaite pouvoir créer un perceptron avec la ligne :

```
val monPerceptron = Perceptron( 2, 3, 4, 1)
```

// un perceptron avec 2 entrées, et des couches cachées de 3 et 4 neurone, puis un neurone de sortie.

Le nombre d'arguments est variable comme l'indique Int* (qui permet d'avoir accès à la collection des paramètres) :

```
// construction
 def apply( couches_ : Int* ): Perceptron = {
  new Perceptron( couches_.toArray )
 }
```

Je vais maintenant m'intéresser à la classe Perceptron : je passe au constructeur un tableau qui contient le nombre de neurones de chaque couche.

```
class Perceptron(couches_ : Array[Int]) {
```

Il faut ensuite créer les structures de données adaptées. J'utilise encore une manière fonctionnelle ici :

```
// pour rester très proche du cours
 var outputI = (0 until couches_.length).map( i => new Array[Double](couches_(i))).toArray
```

```
var inputI  = (0 until couches_.length).map( i => new Array[Double](couches_(i))).toArray
var dI      = (0 until couches_.length).map( i => new Array[Double](couches_(i))).toArray

var poids = (1 until couches_.length)
                    .map( i => Array.ofDim[Double](couches_(i), couches_(i-1))).toArray
```

Prenez un morceau de papier et décomposez les lignes précédentes pour bien comprendre ce que cela fait.

Notez qu'ici, je n'ai pas utilisé des structures de données immuables parce que les valeurs vont être modifiées de manière très conséquentes et sans arrêt pour l'apprentissage. Aucun intérêt ici, d'autant plus que je ne vais pas utiliser le fonctionnel pour les algo d'apprentissage, mais l'impératif.

Ensuite, je créé des poids au hasard :

```
this.poidsHasard()
```

avec la méthode :

```
def poidsHasard() { // met au hasard les poids du réseau
   for( c <- poids.indices)  // chaque couche
    for (n <- poids(c).indices)  // chaque neurone
     for(p <- poids(c)(n).indices) poids(c)(n)(p) = 1-2*Math.random()   //   [-1.0, 1.0]
 } // poidsHasard
```

Ici, j'utilise bien de l'impératif (les for sont des itérateurs sans yield et sans transformation)

Comme j'utilise des Arrays, je simplifie mon écriture en renommant Array :

```
import scala.{Array => $}
```

Je souhaite utiliser mon instance de la manière suivante :

```
val reponse = monPerceptron($(1.0, 2.0, 4.0))
```

J'active lacouche d'entrée avec les valeurs passées en argument : la première couche a donc 3 neurones dans cet exemple. Je récupère un tableau des états d'activation des neurones en sortie (donc de la taille de la couche de sortie).

Je défini donc la méthode apply :

```
def apply(in_ : Array[Double]) : Array[Double] = { // 12 lignes maxi
```

Cette méthode commence par activer les neurones de la couche 0 en remplissant outputI(0) puis propage les activations couche par couche (en utilisant le produit scalaire) jusqu'à la couche de sortie. Je gère donc les inputI, les outputI – mais pas les dI évidemment. Pas encore.
Je renvoie en sortie le tableau des états d'activation de la couche de sortie.

Pour cette fonction, j'utilise de l'impératif, des boucles pour gérer les modifications de tableau. Ca permet d'être beaucoup plus lisible et de pouvoir commenter. Les commentaires s'adressent ici plus à moi qu'à un autre utilisateur. Je veux être bien certain de bien comprendre ce que je fais à chaque étape pour être certain que c'est bien ce que je veux faire (!).

Je vérifie que cette fonction apply fonctionne correctement en essayant avec un réseau avec un neurone en entrée un en sortie, puis avec deux neurones en entrée et un en sortie, en fixant à la main les poids et en vérifiant (tous) les calculs à la main. J'essaye ensuite avec 3 couches. JE NE DEPASSE PAS CETTE ETAPE TANT QUE JE NE SUIS PAS CERTAIN QUE L'ACTIVATION FONCTIONNE CORRECTEMENT. Pourquoi ? parce que si je vais trop vite, comme l'apprentissage dépend de l'activation, je ne saurai pas si mes bugs sont liés à l'activation ou bien à l'apprentissage – et encore une fois, je n'ai pas de temps à perdre.

L'étape suivante consiste à réaliser l'apprentissage sur le réseau :

def retroPropag(observe_ : Array[Double], souhaite_ : Array[Double], pas_ : Double = 0.1): Unit = {
// 25 lignes maxi

Les paramètres représentent pour observe_ le tableau des états d'activation observé sur le réseau (les états de la couche de sortie) et souhaite_ le tableau des valeurs que je veux. Le paramètre pas_ est la valeur e(k) du cours. Je la fixe pour l'instant à une valeur par défaut, ce qui me permet de l'oublier en phase de mise au point, mais de la modifier si nécessaire.

J'utilise encore des boucles en impératif ici (des itérateurs for (<-)). Scala le fait très bien également.

Je commence par calculer les dI de la couche de sortie, puis je remonte de la couche de sortie vers la première couche (en calculant les autres dI) et je fais attention car la première couche du réseau (la couche d'entrée) n'est pas connectée à une couche précédente. A chaque couche, je corrige les poids. Je fais très attention en regardant sur mon dessin, de ne pas me tromper dans les indices des poids. En particulier pour le calcul de la somme des dH, il y a un piège avec les indices des poids : les poids sont sur les connexions suivantes et pas les connexions précédentes comme avec l'activation.

Pour réaliser l'apprentissage, il faut que je représente les couples (X,Y) du cours (les exemples à apprendre).
Je défini des case class :

case class X(x_ : Array[Double]) // entree
case class Y(y_ : Array[Double]) // sortie

et je décide de former une liste de couples (X,Y) pour les exemples à apprendre.

List[Tuple2[X, Y]]

Je pourrais faire des couples de tableau, mais le fait de rajouter un case class, me permet de bien différencier ce qui est entrée du réseau et sortie du réseau – et en plus, ça me permet de coller au cours, ce qui simplifie ma compréhension et ma relecture de mon code.

Je rajoute la méthode :

```
def erreur( ex_ : List[Tuple2[X, Y]]) : Double = { // 3 Lignes maxi
   ex_.map{
    case (X(entree), Y(sortieSouhaitee)) => Perceptron.errQuad(sortieSouhaitee, this(entree))
   }.sum
 }
```

qui calcule l'erreur quadratique observée sur l'ensemble d'apprentissage complet. Remarquez que cette fois, j'utilise du fonctionnel. Pourquoi pas.

Enfin, je rajoute :

def apprendreUneFois(ex_ : List[Tuple2[X, Y]]) { // 3 lignes maxi

qui réalise l'apprentissage de l'ensemble d'apprentissage une fois : c'est-à-dire que chaque exemple est activé puis retropropagé une fois. J'utilise également le fonctionnel.

Une remarque : il est souhaitable de ne pas présenter systématiquement les exemples d'apprentissage dans le même ordre. C'est-à-dire qu'il faut mélanger la liste ex_ avant d'utiliser ses éléments. Faire une fonction qui mélange aléatoirement une liste est simple : c'est un bon exercice à faire – QUE VOUS DEVEZ SAVOIR FAIRE ET EN IMPERATIF ET EN FONCTIONNEL. Ici, si vous travaillez en fonctionnel (et en Scala), j'attire votre attention sur **scala.util.Random.shuffle**.

L'étape suivante (après les tests), consiste à utiliser des fonctions plus générales qui encapsulent les fonctions précédentes et les variables qui ne sont pas des variables d'état : par exemple inputI, outputI et dI ne servent que pour l'apprentissage et n'ont pas à être des variables d'instances : elles ne doivent exister que dans le contexte d'une fonction générale d'apprentissage (qui contient les fonctions apprendreUneFois etc). Le seul état à conserver pour un perceptron, c'est la collection des poids (et donc sa structure).