Licence Maîtrise d'Informatique de Lille

Maîtrise d'informatique – 2003/2004

Intelligence Artificielle

mars 2004

USTL

Réseaux de neurones
Travaux Pratiques

Cette séance de travaux pratiques se propose d'étudier d'abord l'utilisation des réseaux de neurones par l'intermédiaire de l'interface graphique de `Weka`*, puis d'aller un peu plus loin, en écrivant des petits programmes qui iront chercher des informations supplémentaires à l'intérieur de la classe* `MultiLayerPerceptron`*.*

1 Les options

Sous `Weka`, l'implémentation des réseaux de neurones utilisant l'algorithme de rétropropagation du gradient s'appelle `MultiLayerPerceptron` (dans `classifiers`,puis `functions`). Les principales options disponibles dans l'`Explorer` sont les suivantes :

GUI : permet l'utilisation d'une interface graphique (pour plus de détails, lire la doc en ligne (bouton `more` dans la fenêtre de choix des options)).

autobuild : Connecte les couches cachées : le laisser à `True`.

decay : si vrai, faire décroître le taux d'apprentissage : les poids sont moins modifiés au fur et à mesure de l'apprentissage.

hiddenLayers : permet de décrire le nombre et la taille des couches cachées. La description est :

- Soit une suite d'entiers (le nombre de neurones par couche) séparés par des virgules.
- Soit les valeurs spéciales déterminant une seule couche cachée :
  - `a` : (nombre d'attributs+nombre de classes)/2
  - `i` : nombre d'attributs
  - `o` : nombre de classes
  - `t` : nombre d'attributs+nombre de classes

learning rate : le η du cours ...

momentum : le α du cours ... :

$$\Delta w(n) = \eta w x + \alpha \Delta(n-1)$$

Pour observer l'algorithme de rétro-propagation du gradient dans toute sa pureté, fixez-le à 0.

nominalToBinaryFilter : transforme les attributs nominaux en attributs binaires : un attribut pouvant prendre k valeurs différentes sera transformé en k attributs binaires, un seul de ces attributs valant 1.*je ne sais pas comment fait* `Weka` *lorsque cette option est mise à* `False` *!*

normalizeAttributes : les valeurs des attributs (y compris les attributs nominaux qui seront passés dans le filtre nominalToBinaryFilter) seront toutes ramenées entre -1 et 1.

normalizeNumericClass : si la classe est numérique, on la normalise (de façon interne) : permet d'améliorer les résultats ...

training time : le nombre de fois (epochs en anglais) où l'on fera passer l'ensemble d'apprentissage à travers le réseau.

2 Exemple

Le premier exemple est le problème des Iris de Fischer, déjà rencontré plusieurs fois : il est facile (la plupart des classifieurs s'en sortent plutôt bien) , ses attributs sont continus, et tous du même ordre de grandeur.

Après avoir ouvert l'`Explorer`, prenez le fichier iris.arff : tous les attributs sont continus, ils ne sont pas normalisés, mais ont tous le même ordre de grandeur.

2.1 Paramètres par défaut (presque. . .)

Dans la liste des options, mettez le moment (`momentum`) à zéro, pour appliquer l'algorithme de rétro-propagation sans amélioration. Lancez le réseau de neurones, en fixant `Use training set` : afin de ne pas attendre trop longtemps le résultat à cause de la validation croisée.

2.1.1 Lecture des résultats

Si les générateurs aléatoires ne diffèrent pas selon les machines, vous devriez tous avoir les mêmes résultats, sinon, il sera facile de faire le lien entre les différents résultats.

Le première chose qui nous intéressera, c'est l'erreur en classification :

```
=== Evaluation on training set ===
=== Summary ===

Correctly Classified Instances         148               98.6667 %
Incorrectly Classified Instances         2                1.3333 %
```

que l'on peut aussi lire par l'intermédiaire de la matrice de confusion :

```
=== Confusion Matrix ===

  a  b  c   <-- classified as
 50  0  0 |  a = Iris-setosa
  0 49  1 |  b = Iris-versicolor
  0  1 49 |  c = Iris-virginica
```

On peut remarquer que les résultats sont relativement bons (deux erreurs seulement).

Question 2.1 : En demandant de visualiser les erreurs (click droit dans `Result List`, puis `Visualize classifiers errors`), retrouvez les deux instances sur lesquels le classifieur se trompe. Quelle est à votre avis la cause de son erreur ?

Question 2.2 : Relancer l'apprentissage, en fixant maintenant un temps de calcul de 25 *epochs*, au lieu de 500. Regardez de nouveau où se situent les instances mal classées.

Question 2.3 : Combien faut-il d'*epochs* pour que tous les exemples soient bien classés ?

Question 2.4 : Que se passe-t-il si on ne normalise pas les attributs ?

2.1.2 Voir le réseau

Reprenez un apprentissage de 500 *epochs*, mais demandez maintenant la visualisation du réseau (`GUI` dans la fenêtre des options du `MultilayerPerceptron`)

Maintenant, lorsqu'on appuie sur `Start` une fenêtre s'ouvre, qui nous montre le réseau construit par `Weka` (figure 1)

Quelques petites remarques :

1. La couche cachée contient
$$(3+3)/2 = 3$$
neurones dans la couche cachée : c'est la politique choisie par défaut.
2. On a codé les trois sorties sur trois neurones : la classe choisie est celle pour laquelle la sortie a la valeur la plus élevée.
3. Il n'y a pas besoin de codage des entrées, puisqu'elles sont continues

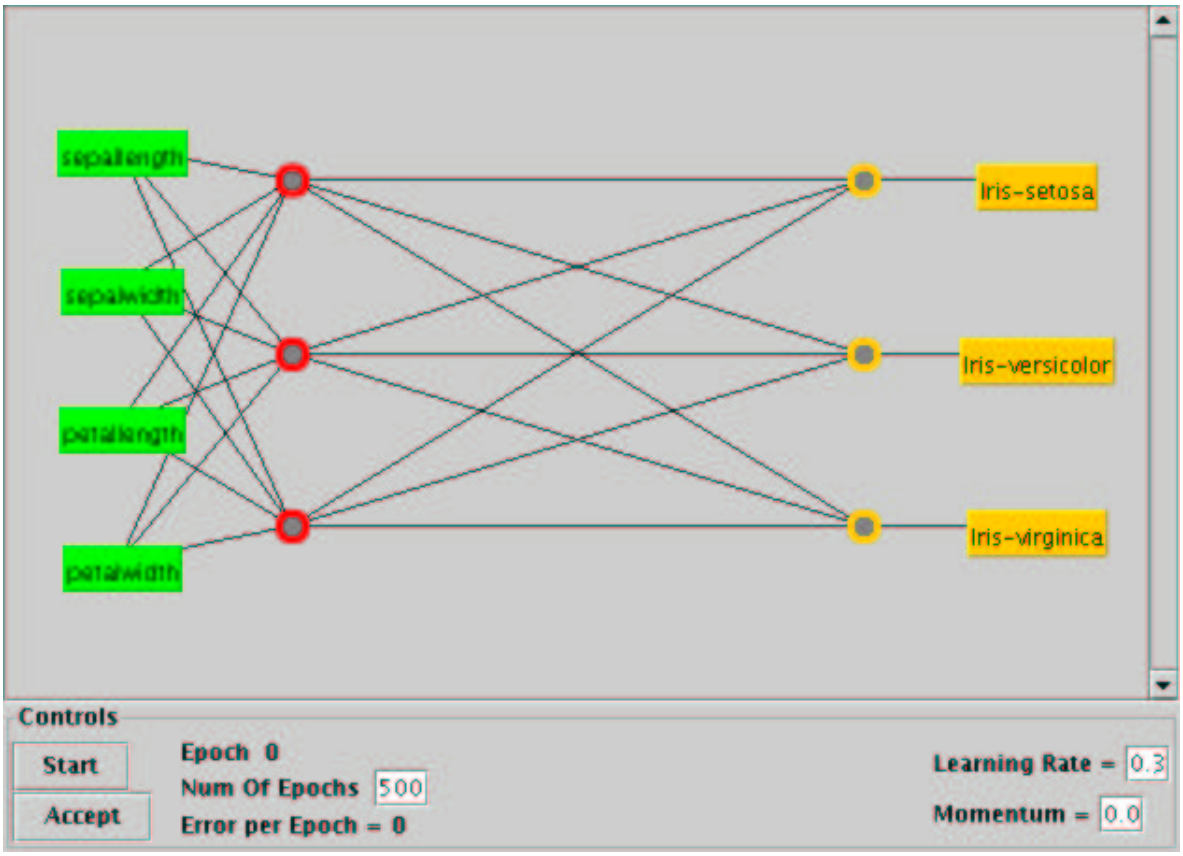

FIG. 1 – *Le réseau pour le problème des iris*

Pour lancer l'apprentissage, il faut maintenant, dans la fenêtre représentant le réseau, appuyer sur `Start`, puis sur `Accept` pour fermer la fenêtre et obtenir les statistiques dans la fenêtre `Classifier output`.

L'interface graphique ne devient intéressante que si l'on veut éditer le réseau, ce que nous ne ferons pas, au moins pour l'instant.

2.1.3 Etudier le réseau

Les nœuds sont numérotés de 0 à n, en commençant par la couche cachée la plus à droite, et par le neurone du haut. Après apprentissage, la fenêtre `Classifier output` nous donne la valeur des poids dans tout le réseau. Par exemple, les poids en entrée du nœud `node 0`, celui qui correspond à la sortie `iris setosa`, ont les valeurs indiquées dans la figure 2

```
Sigmoid Node 0
    Inputs    Weights
    Threshold    -3.407862566565817
    Node 3    -0.9122514567058606
    Node 4    8.759379585222362
    Node 5    -4.138044195407552
```

FIG. 2 – *Les poids en entrée de* `Node 0`

Question 2.5 : A partir de ces informations en chaque nœud, pourrait-on lire ce que le réseau a appris, ou au moins, pouvoir dire ce que chaque neurone calcule ?

Question 2.6 : En étudiant le rôle de chaque neurone interne, pensez-vous que l'on ait vraiment besoin de trois neurones dans la couche interne ?

2.2 Modifier les paramètres

Question 2.7 : Etudiez l'influence des divers paramètres sur l'apprentissage du réseau.

Question 2.8 : Essayez aussi d'avoir une estimation de l'erreur par validation croisée, et de voir si l'algorithme apprend toujours aussi bien lorsqu'on réduit la taille de l'ensemble d'apprentissage (on dispose alors d'un ensemble test).

2.3 Comparaison

Question 2.9 : Comparez les résultats avec ceux obtenus par un arbre de décision.

2.4 Comparaison plus fine

Question 2.10 : Les exemples mal classés par le réseau de neurones sont-ils aussi mal classés par un arbre de décision ? [1] :

Pour obtenir la classification d'un exemple, on utilise la méthode `distributionForInstance`, qui retourne un tableau contenant les prédictions du réseau pour les différentes valeurs de la classe. La classe attribuée est celle correspondant à la plus grande prédiction.

Question 2.11 : Comment pourrait-on faire pour voir ce qui distingue les exemples bien classés par un des deux classifieurs et par pas l'autre ?

1. Nécessite un minimum de programmation...