Réseaux de neurones

Bruno Bouzy

18 octobre 2005

Introduction

Ce document est le chapitre « réseau de neurones » du cours d'apprentissage automatique donné en Master MISV, parcours IPCC. Il donne des éléments sur le problème de la classification, problème sur lequel les réseaux de neurones donnent de bons résultats. Ensuite, il montre l'exemple d'un réseau avec 4 neurones résolvant le problème du XOR avec l'algorithme de back-propagation (Rumelhart & al 1986). Cet exemple permet d'introduire cet algorithme et d'en montrer les avantages et limites. Ensuite, ce chapitre présente les fonctions discriminantes linéaires, le perceptron ou réseau de neurones à une couche, le MPL, (Multi-Layer Perceptron) ou réseau de neurones multi-couches.

Généralités

Cette partie présente quelques généralités sur la classe de problèmes sur laquelle les réseaux de neurones s'appliquent avec succès : la tâche de classification et la tâche de régression.

La classification et la régression

La tâche de classification et celle de régression sont différentes par leurs sorties. Dans la tâche de *régression*, les sorties sont des fonctions numériques des entrées, généralement continues. Dans la tâche de *classification*, les sorties sont des fonctions booléennes indiquant l'appartenance ou pas de l'exemple à une classe. Vocabulaire : pour un exemple, la fonction booléenne vaut vrai, et pour un « contre-exemple », elle vaut faux. En revanche, pour les deux tâches, la méthode d'apprentissage est la même.

Ensemble d'apprentissage, ensemble de test.

Pour les deux tâches, on souhaite que le réseau de neurones donnent les bonnes sorties sur un ensemble d'exemples, appelé ensemble de test. Pour cela, on va utiliser un ensemble d'exemples sur lequel le réseau va s'entraîner et apprendre. Cet ensemble s'appelle l'ensemble d'apprentissage. Les exemples de l'ensemble de test n'appartiennent pas à l'ensemble d'apprentissage. L'ensemble de test est beaucoup plus grand que l'ensemble d'apprentissage. L'ensemble opérationnel est composé de tous les exemples réels rencontrés au cours de l'utilisation finale du réseau de neurones. Avec cette méthode basée sur la distinction entre ensemble d'apprentissage et ensemble de test, on constate que le réseau de neurones doit être capable de généralisation.

Généralisation

Si une image de 256x256 possède 256 niveaux de gris, alors on peut obtenir $2^{8x256x256}$ images. Au lieu de stocker toutes les images avec leur classification, on a un ensemble d'apprentissage avec un millier d'exemples seulement. A partir de cet ensemble petit d'apprentissage, le réseau de neurones doit classer correctement toutes les images de l'ensemble grand de test. Il doit donc posséder une bonne propriété de ce que l'on appelle la *généralisation*. (cf cours sur l'approximation polynomiale de courbe).

Pré-traitement, extraction, sélection de propriétés.

Habituellement, on pré-traite les exemples en entrée du système d'apprentissage afin de faciliter l'apprentissage. Un pré-traitement consiste à extraire les propriétés représentatives des exemples, ou à ne sélectionner qu'un sous-ensemble des propriétés quand celles-ci sont trop nombreuses pour le système d'apprentissage.

"curse of dimensionality"

Si l'espace d'entrée des exemples est de dimension d et que l'on découpe chaque dimension avec M intervalles, alors l'espace est découpé en M^d cubes. Ce nombre croit exponentiellement avec M. Chaque cube contient au moins un exemple. On voit que le nombre d'exemples croit exponentiellement avec la dimension. C'est ce que l'on appelle en anglais « curse of dimensionality ».

Biais variance

(cf cours sur l'approximation polynomiale de courbe).

« Backprop » sur l'exemple du XOR

Cet exemple est destiné à montrer le fonctionnement d'un réseau de neurones apprenant avec l'algorithme « backprop » (Rumelhart & al 1986). L'exemple est repris de (Tveter, chapitre 2). Le réseau est très simple. Son but est de donner la valeur de XOR(x, y) en fonction de x et y. La fonction z = XOR(x, y) est définie par la table 1.

x	0	0	1	1
y	0	1	0	1
z	0	1	1	0

Table 1 : la fonction XOR.

Le premier paragraphe montre comment le réseau fonctionne en « feed-forward », en avant, pour donner la solution z en fonction des entrées x et y. Le second paragraphe montre comment le réseau « apprend », c'est-à-dire met à jour les poids de ses connexions en fonction des exemples qui lui sont présentés. Cet apprentissage est fait suivant l'algorithme de « back-propagation » (Rumelhart, Hinton & Williams 1986) dont le support théorique est mentionné par le troisième paragraphe (qui peut être sauté).

Calcul en avant :

En mode opérationnel, lorsque l'apprentissage est terminé, le réseau ne fait que du calcul « en avant ». Le but de cette partie est de montrer ce calcul en avant.

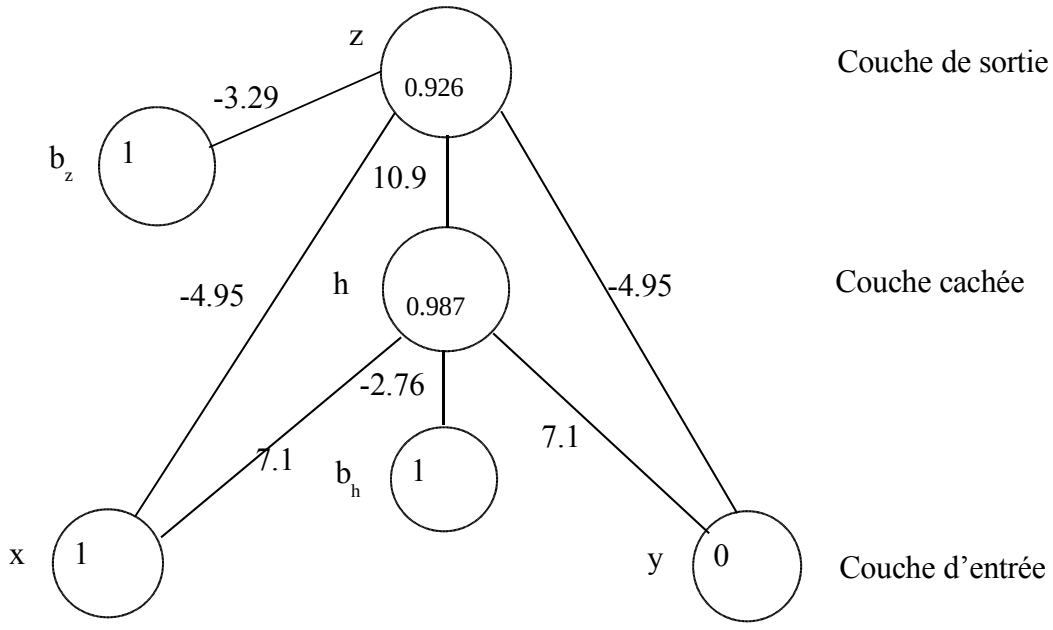

Figure 1 : Un réseau à 3 couches pour résoudre le problème du XOR, avec des poids obtenus par « back-propagation » .

La figure 1 montre un réseau de neurones à trois couches permettent de calculer z, le XOR de x et y, deux variables données en entrée du réseau. Les ronds représentent des « unités » ou « neurones ». Il y a deux unités dans la couche d'entrée : x et y. Il y a 3 unités dans la couche

cachée : h, b_h et b_z. z est la seule unité de la couche de sortie. b_h et b_z. sont des « unités de biais ». La plupart du temps les unités de biais ne sont pas représentées. Les nombres dans les ronds indique les valeurs d'activation des unités. Les lignes reliant les ronds sont les connexions entre unités. Chaque connexion a un « poids » dont la valeur est indiquée à côté de la ligne correspondante.

La plupart du temps, les réseaux ont des connexions reliant uniquement une couche à sa couche voisine, mais sur cet exemple, la couche d'entrée est connectée directement à la couche de sortie. Dans certains problèmes, comme celui du XOR, ces connexions supplémentaires rende l'apprentissage beaucoup plus rapide. En général, les réseaux sont décrits par le nombre d'unités dans chacune des couches, sans compter les unités de biais. Le réseau de la figure 1 est donc un réseau 2-1-1. On peut le noter 2-1-1-x pour mentionner le fait qu'il existe des connexions supplémentaires (de x et y vers z).

Pour calculer z, prenons l'exemple de la figure 1 en supposant que x=1.0 et y=0.0. (D'après le tableau, on doit trouver 1).

Pour calculer la valeur d'activation d'une unité en fonction de ses connexions entrantes, on commence par effectuer la somme sur toutes les connexions entrantes dans l'unité, des produits de l'unité associée à la connexion et du poids de la connexion. Pour l'unité h, on a donc :

$$\text{somme} = 1x7.1 + 1x(-2.76) + 0x7.1 = 4.34.$$

Dans les réseaux linéaires, cette somme est égale à la valeur d'activation. Dans notre exemple, le réseau est non linéaire, il utilise une fonction non linéaire appelée la fonction sigmoïde f.

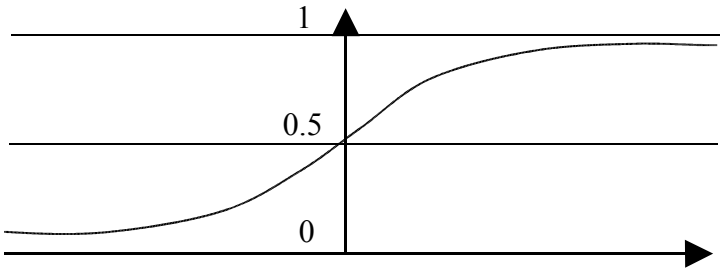

Figure 2 : la fonction sigmoïde f.

La figure 2 montre la fonction sigmoïde dont la formule est :

$$f(x) = 1/(1+e^{-x}) \qquad (1a)$$

On vérifie que f(0) = 0.5 que $f(-\infty) = 0$ et que $f(+\infty) = 1$. La valeur de l'activation d'une unité est f(somme). Dans l'exemple, on a donc h = f(4.34) = 0.987. la fonction f s'appelle la fonction sigmoïde standard ou la fonction logistique. De manière générale, la fonction f s'appelle la fonction de transfert, ou fonction d'activation, ou encore fonction « écrasante ».

Pour calculer z, on a :

$$\text{somme} = 1.0x(-4.95) + 0.0x(-4.95) + 0.987x10.9 + 1.0x(-3.29) = 2.52.$$

Donc z = f(2.52) = 0.926.

Bien sûr, 0.926 est différent de 1, mais pour notre exemple, c'est suffisamment près. Avec la fonction sigmoïde, de toute façon on ne peut pas obtenir la valeur 1, atteinte à l'infini seulement. Donc lorsque la valeur sera suffisamment proche de 1, on sera satisfait. Avec le réseau de la figure 1, on obtient la table 2, proche de la table 1 :

x	0	0	1	1
y	0	1	0	1
z	0.067	0.926	0.926	0.092

Table 2 : la sortie du réseau de neurones.

On peut écrire plus rapidement le calcul de la valeur d'activation o_j d'un neurone j en fonction de ses entrées i par la formule suivante :

$$o_j = f(net_j) \text{ avec } net_j = \Sigma_i w_{ij} o_i \qquad (1b)$$

w_{ij} est le poids de la connexion reliant le neurone i avec le neurone j. La somme s'applique pour toutes les connexions partant d'un neurone i et arrivant sur le neurone j.

Apprentissage des poids du réseau avec « backprop » :

Pour obtenir le réseau avec les poids de la figure 1, on a utilisé l'algorithme backprop. Le but de cette partie est de montrer le fonctionnement de backprop sur l'exemple du XOR.

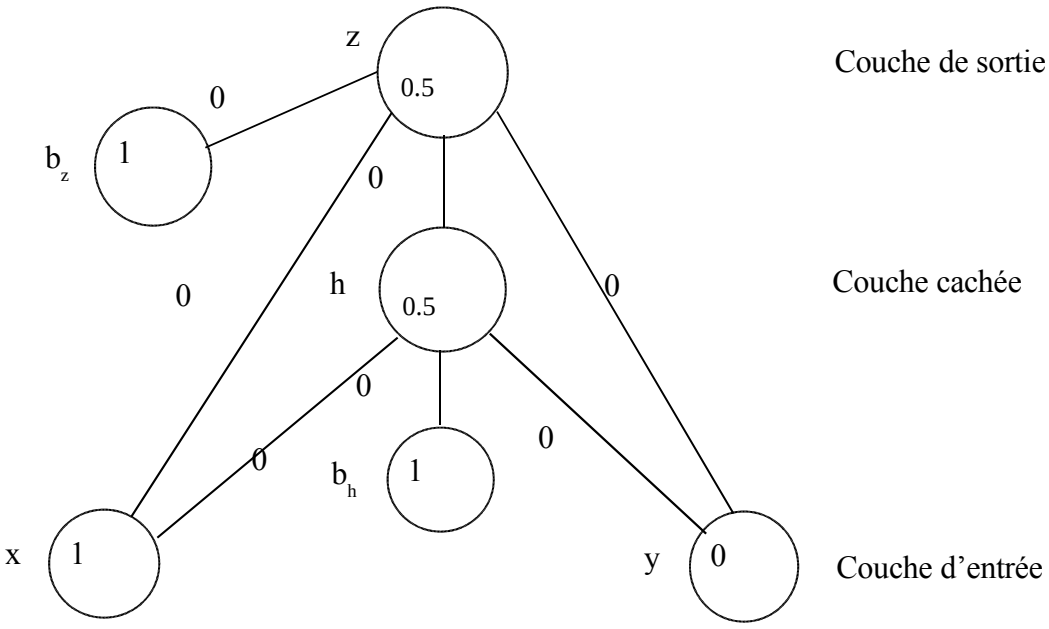

Figure 3 : Le réseau avant le démarrage de l'apprentissage, avec des poids nuls.

Le but de l'apprentissage est de trouver les poids corrects, c'est-à-dire donnant au réseau un comportement se rapprochant le plus possible de celui de la table 1. La figure 3 montre le réseau avant le démarrage de l'apprentissage : les poids des connexions sont nuls, x=1, y=0, h=0.5, z=0.5. La cible pour z est 1 et l'erreur sur cette unité est donc 0.5.

Le processus d'apprentissage « backprop » utilise le programme suivant :

1° Mettre un exemple à apprendre en entrée du réseau.
2° Calculer les valeurs d'activation des neurones cachés et de sortie avec (1a) et (1b).
3° Calculer l'erreur entre la valeur de l'exemple et celle du réseau avec (2) et (3).
4° Mettre a jour les poids des connexions allant sur l'unité de sortie avec (4).
5° Calculer l'erreur dans les unités de la couche cachée avec (5).
6° Mettre à jour les poids des connexions allant sur la couche cachée avec (6).

Répéter les pas 1 à 6 pour les autres exemples à apprendre. L'ensemble de ces répétitions se nomme une *itération*. Après une itération, la sortie du réseau sera un peu plus proche de la bonne solution. L'algorithme « backprop » consiste à effectuer plusieurs itérations jusqu'à ce que l'erreur soit suffisamment petite.

Les formules utilisées pour les pas 4, 5, et 6 sont les suivantes. Soit t_k la valeur souhaitée de l'unité k et o_k sa valeur d'activation. On appelle d_k le signal d'erreur défini par la formule :

$$d_k = (t_k - o_k)\, f'(net_k) \qquad (2)$$

où f' est la dérivée de f. Si on utilise pour f la fonction sigmoïde, alors on peut montrer facilement que :

$$f'(net_k) = o_k\,(1 - o_k) \qquad (3)$$

La formule pour changer le poids w_{jk} entre l'unité de sortie k et l'unité j est:

$$\Delta w_{jk} = \nu\, d_k\, o_j \qquad (4)$$

ν se nomme le pas d'apprentissage de l'algorithme. La formule 4 se comprend intuitivement de la manière suivante : on modifie un w poids proportionnellement à la sortie de l'unité associée à la connexion, à l'erreur calculée par 2 et le pas de l'apprentissage ν, déterminé par l'expérience. Dans l'exemple on a ν=0.1. Avec le réseau de la figure 3, on a successivement :

$$d_z = (1 - 0.5)\, 0.5\, (1 - 0.5) = 0.125$$

$$w_{zx} = 0 + 0.1 \text{x} 0.125 \text{x} 1 = 0.0125$$

$$w_{zy} = 0 + 0.1 \text{x} 0.125 \text{x} 0 = 0$$

$$w_{zh} = 0 + 0.1 \text{x} 0.125 \text{x} 0.5 = 0.00625$$

$$w_{zbz} = 0 + 0.1 \text{x} 0.125 \text{x} 1 = 0.0125$$

La formule pour calculer l'erreur d_j d'une unité cachée j est:

$$d_j = f'(net_j) \sum_k d_k\, w_{jk} \qquad (5)$$

La somme s'applique pour toutes les connexions arrivant sur un neurone k et partant du neurone j. Dans notre exemple, l'unité h ne possède qu'une seule connexion sortante (vers z). On a :

$$d_h = 0.5 \text{ x } (1 - 0.5) \text{ x } 0.125 \text{ x } 0.00625 = 0.000195313$$

La formule pour changer le poids w_{ij} entre l'unité cachée j et l'unité d'entrée i est similaire à (4) :

$$\Delta w_{ij} = \nu \, d_j \, o_i \qquad (6)$$

Les nouveaux poids des connexions arrivant sur l'unité h seront:

$$w_{hx} = 0 + 0.1x0.000195313x1 = 0.0000195313$$

$$w_{hy} = 0 + 0.1x0.000195313x0 = 0$$

$$w_{hbh} = 0 + 0.1x0.000195313x1 = 0.0000195313$$

Avec ces nouveaux poids, la valeur d'activation de z est 0.507031. Si on répète la même procédure pour les trois autres exemples, on obtient les poids de la première itération. Et on obtient la table 3 :

x	0	0	1	1
y	0	1	0	1
z	0.499893	0.499830	0.499830	0.499768

Table 3 : la sortie du réseau de neurones après une itération.

On observe que les valeurs d'activation ont très légèrement changé. Pour aboutir à des sorties égales à la vraie fonction XOR de la table 1, avec une erreur de 0.1, il faut 20,682 itérations, ce qui est un nombre très grand pour un problème aussi simple. Heureusement, de nombreuses choses peuvent être faites pour accélérer le processus. En particulier, on peut augmenter le pas de l'apprentissage ν. La table 4 montre que l'on peut réduire le nombre d'itérations à 480 avec ν=2.0.

ν	0.1	0.5	1.0	2.0	3.0
N	20,682	2,455	1,060	480	-

Table 4 : le nombre d'itérations N nécessaires pour obtenir une erreur de 0.1,
pour différentes valeurs du pas d'apprentissage ν.

Lorsque le pas d'apprentissage est trop grand, la méthode rate. La table 5 montre la valeur de z après 10,000 itérations.

x	0	0	1	1
y	0	1	0	1
z	0.009	0.994	0.994	0.999

Table 5 : la sortie du réseau de neurones après convergence incorrecte
avec un pas d'apprentissage trop grand.

On observe que z vaut 1 au lieu de 0 si x=1 et y=1. La méthode d'apprentissage a convergé vers une mauvaise valeur. Elle a trouvé un minimum local de la fonction d'erreur au lieu de trouver le minimum global.

Dans la méthode présentée, lors de la mise à jour avec un exemple, on a calculé et mis à jour les poids des connexions de la couche de sortie pour calculer l'erreur de la couche cachée. On aurait pu calculer les poids de la couche de sortie, ne pas les mettre à jour tout de suite, calculer l'erreur de la couche cachée avec les anciens poids, puis mettre à jour les poids de la couche de sortie. Cela aurait été plus « propre » (?) théoriquement, mais en pratique, on préfère la méthode que nous avons utilisée.

La remarque précédente peut être faite au niveau de l'ensemble des exemples présentés dans l'apprentissage. On aurait pu faire les calculs d'erreurs sans mettre à jour les poids. Puis lorsque tous les exemples auraient été présentés, mettre à jour le poids. Une controverse existe entre les « théoriciens » favorables à la mise à jour retardée des poids et les « praticiens » favorables à la mise à jour directe des poids.

Reproduction de l'expérience:

Cette expérience a été reproduite. La convergence dépend des valeurs initiales du réseau et de la valeur de ν. le nombre d'itérations dépend du critère d'arrêt que nous avons fixé à $\varepsilon = 0.01$.

Avec des valeurs initiales du réseau *nulles*, j'ai obtenu les résultats suivants:

ν	***convergence ?***	***nombre d'itérations***
0,1	oui	459000
0,25	oui	174000
0,5	oui	85000
1	oui	42000
2	oui	21000
4	non	-

Dans les cas de convergence, j'ai obtenu le réseau suivant:

w_{hx}	w_{hy}	w_{hb}	w_{zx}	w_{zy}	w_{zh}	w_{zb}
8,9	8,9	-3,88	-9,46	-9,46	19,56	-5,23

Avec les valeurs initiales *aléatoires* suivantes:

w_{hx}	w_{hy}	w_{hb}	w_{zx}	w_{zy}	w_{zh}	w_{zb}
0,34	-0,1	0,28	0,29	0,41	-0,3	-0,16

j'ai obtenu des résultats identiques pour savoir si il y a convergence ou pas, mais avec un nombre d'itérations légèrement (10%) plus grand. Cependant, point très important, il faut remarquer que le réseau obtenu possède des poids différents de ceux obtenus avec les valeurs initiales nulles:

w_{hx}	w_{hy}	w_{hb}	w_{zx}	w_{zy}	w_{zh}	w_{zb}
9,88	-9,09	4,62	9,68	-9,37	-19,5	14,42

Comment interpréter ce résultat en terme de descente de gradient ? Le minimum est-il local ? Y-a-t-il plusieurs minima globaux ?