Erik PERNOD
Calcul Scientifique
3ème Année

RESEAUX DE NEURONES

TABLE DES MATIERES

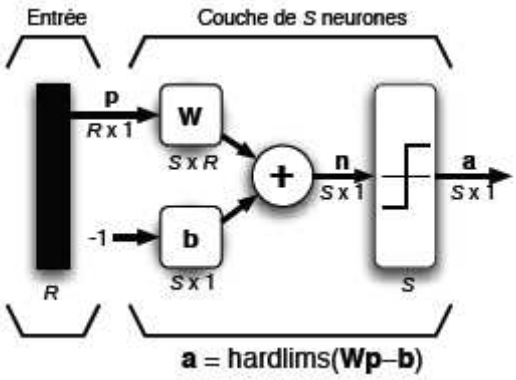
I – PERCEPTRON SIMPLE

I.1 – Introduction

Le perceptron est l'un des réseaux de neurones les plus utilisés pour des problèmes d'approximation, de classification et de prédiction. Dans un premier temps nous allons étudier le perceptron simple, puis un cas de perceptron multicouche avec une couche de deux perceptrons en parallèles.

Puis pour finir nous étudierons le perceptron multicouche de taille (3x3). Dans les trois cas, le perceptron sera assimilé à une classe c++ comportant ses propres éléments. Une classe Vecteur est également utilisée. Elle permet entre autre l'égalité directe de deux vecteurs, l'affichage direct d'un vecteur.

Perceptron simple :



Composition :

- Deux entrées (dans notre cas, des coordonnées x et y d'un point)
- Un biais b, assimilable à une troisième entré, mais variable.
- Un système de trois poids liées à ces entrés sachant que le 3ème poids correspond au biais et est toujours égal à -1.
- Une fonction de transfert (ici nous utiliserons la fonction seuil)
- Une seule sortie

Sachant que la fonction de transfert utilisé est une fonction seuil, les sorties possibles de ce perceptron sont 1 ou -1. Il ne peut donc faire que de la classification, puis de la prédiction.

On se donne un nombre N de points. Certains valant 1 et d'autres -1. Nous allons étudier comment le perceptron simple permet de séparer ces deux groupes. Si ces deux groupes sont bien distincts, sinon nous verrons que ce type de perceptron est vite limité.

I.2 – Algorithme

- <u>Initialisation des données :</u>
  - Nbr = Nombre de points

- Variables : utilisation de vecteurs de float pour les coordonnées des points, leur affinité (1 ou -1) et les poids du perceptron.
- Initialisation du biais du perceptron à 0.
- Utilisation de fichier pour imprimer les données.

- <u>Insertion des points pour la phase d'apprentissage :</u>
  - On insère les coordonnées de chaque point et son affinité dans 3 vecteurs (X, Y, F). Ainsi le triplet [X(1),Y(1),F(1)] défini le 1er point.
  - Par soucis de simplicité, les poids du perceptron auront comme valeur initiale l'affinité du 1er point.

- <u>Phase d'apprentissage (Pour chaque point i) :</u>
  - Calcul de la somme : $S = x_i.w_1 + y_i.w_2 - 1.b$
  - Calcul de f(S) où f est la fonction seuil.
  - On compare le résultat obtenu à celui voulu (vecteur F)
  - Si le résultat est identique, on ne change pas les poids. S'il est différent, on modifie les poids de la façon suivante :

$$\begin{cases} w_1 = w_1 + \eta.(\frac{w_1 + x_i}{2}).F_i \\ w_2 = w_2 + \eta.(\frac{w_2 + y_i}{2}).F_i \\ b = b + \eta.(\frac{-1 + b}{2}).F_i \end{cases}$$

    eta étant le coefficient d'apprentissage.

  - On passe au point suivant

- <u>Critère d'arrêt :</u>
  - A chaque itération, les poids sont modifiés pour satisfaire chacun des points. Lorsque ces poids ne sont plus modifiés par aucun des points, on arrête le programme.
  - Il se peut que l'ensemble des points ne soient jamais satisfait par les poids. Ce sont les cas limite que je montrerai par la suite.

- <u>Prédiction :</u>
  - Une fois les poids bien calibrés (concrètement, le domaine est séparé en deux parties, les points 1 d'un coté, les -1 de l'autre), il est possible de rentrer les coordonnées d'un point pour connaître son affinité.

I.3 – Résultats

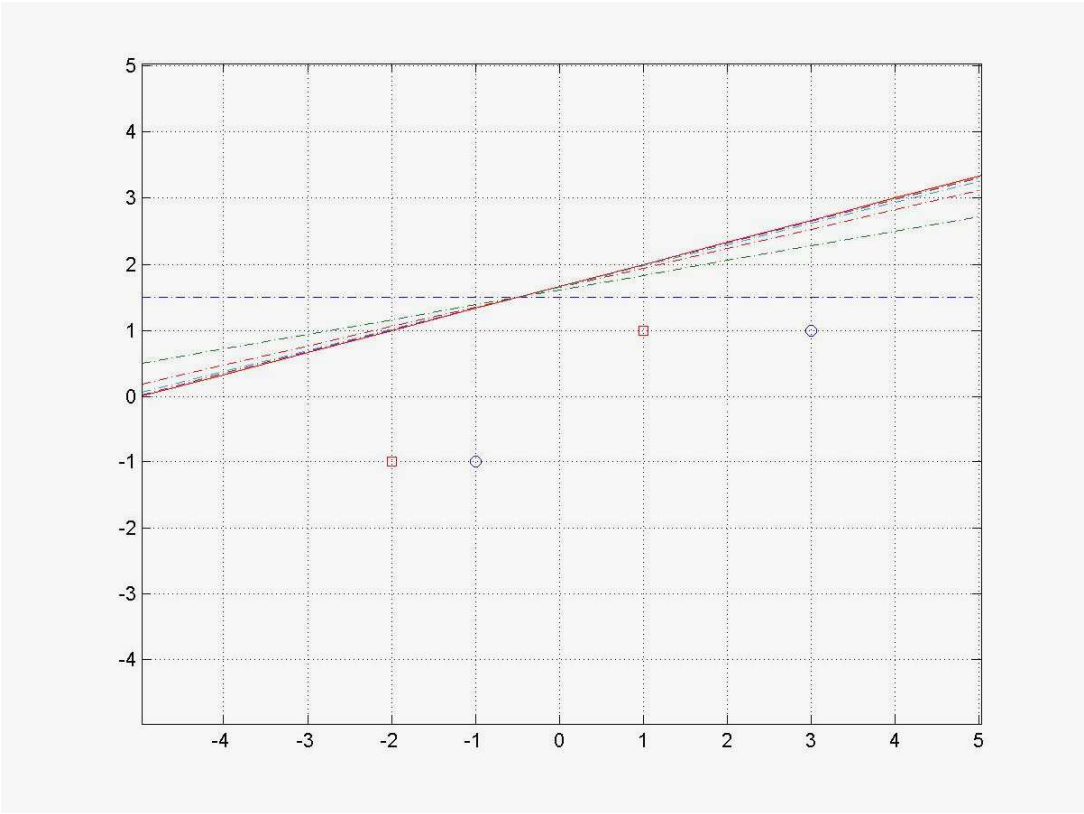
Les coordonnées des points ainsi que leur affinité seront enregistré dans un fichier Points.dat. Les poids ainsi que la valeur de b seront quant à eux enregistré itération par itération dans un fichier Droites.dat

Les graphiques illustrant les résultats suivant sont obtenus à l'aide d'un programme Matlab (joint avec le programme c++)

Je ne montrerais que des cas simplifiés avec un nombre limité de points (4 points). Sachant que rajouter des points ne modifie pas la méthode.

Les carrés rouges indiquent les points ayant une affinité égale à 1 et les ronds bleus indiquent ceux ayant une affinité de -1.

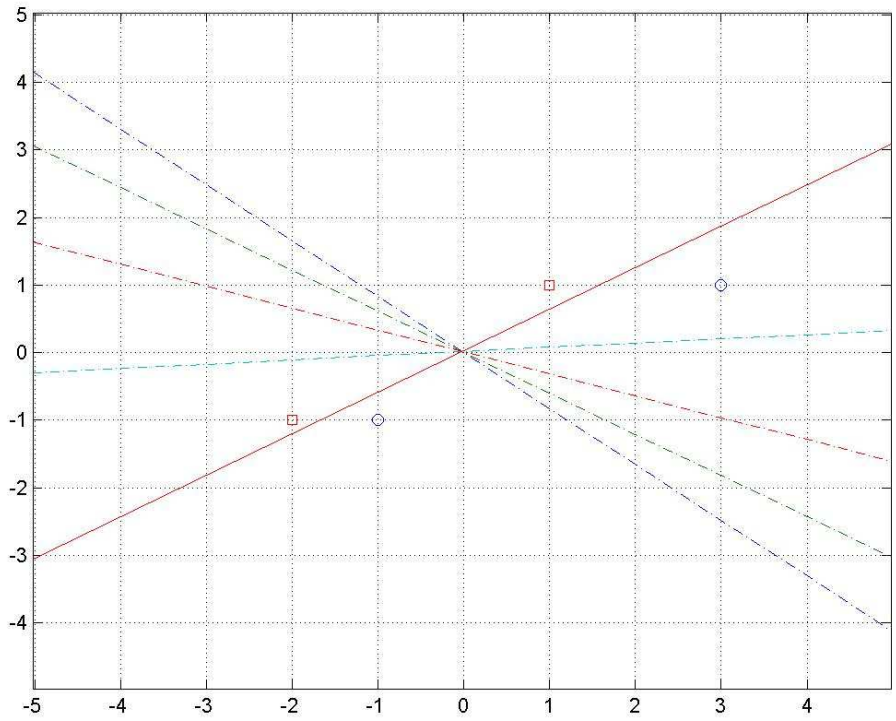
1er exemple : paramètres : 4 points, eta =1.



Dans ce cas, on constate que le biais a été modifié trop rapidement, b/w2 (ordonnée a l'origine) est passé dès la première itération à 1. Du coup le programme n'arrive pas à revenir en arrière et ne s'arrête plus.

2ème exemple : paramètres : 4 points, eta=0.1.

On réutilise les mêmes points mais cette fois ci avec un coefficient d'apprentissage moins élevé :



Cette fois ci, le biais n'a pas été modifié trop rapidement, et la courbe a eu le temps de tourner pour satisfaire les 4 points.
La courbe en trait plein représente la position des poids à la dernière itération.

Rappel :

$x_i.w_1 + y_i.w_2 - 1.b$

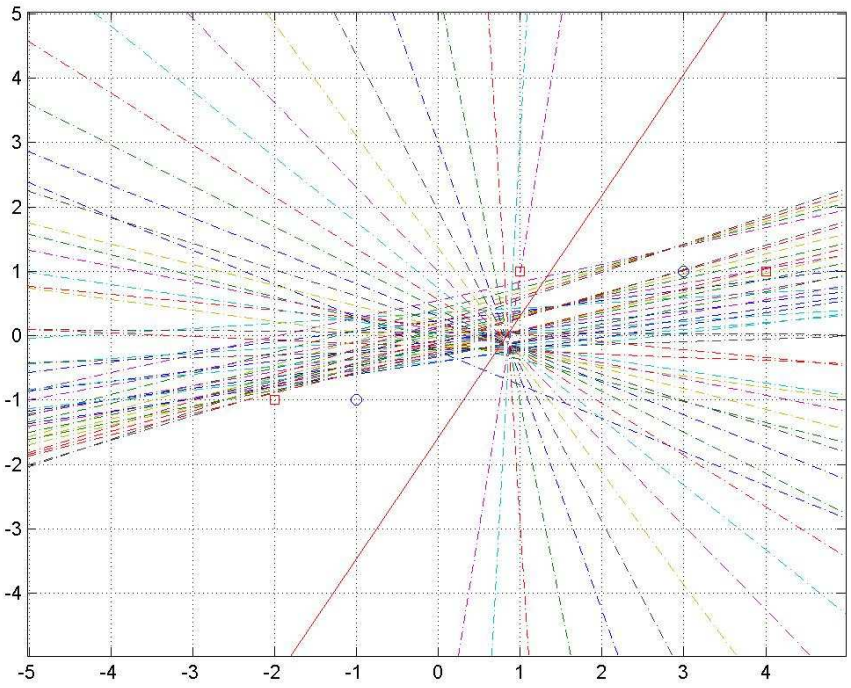
$donne$:

$$y_i = \frac{w_1}{w_2} x_i + \frac{b}{w_2}$$

On a bien une droite du type y=a.x+b.

3ème exemple : paramètres : 5 points eta =0.1.

Dans ce cas, on constate que le dernier point rajouté : (4,1,-1), ne permet pas de séparer le domaine en deux parties distinctes. L'algorithme ne peut donc pas converger. Il s'agit là de la limite du perceptron simple.



I.4 - Conclusion

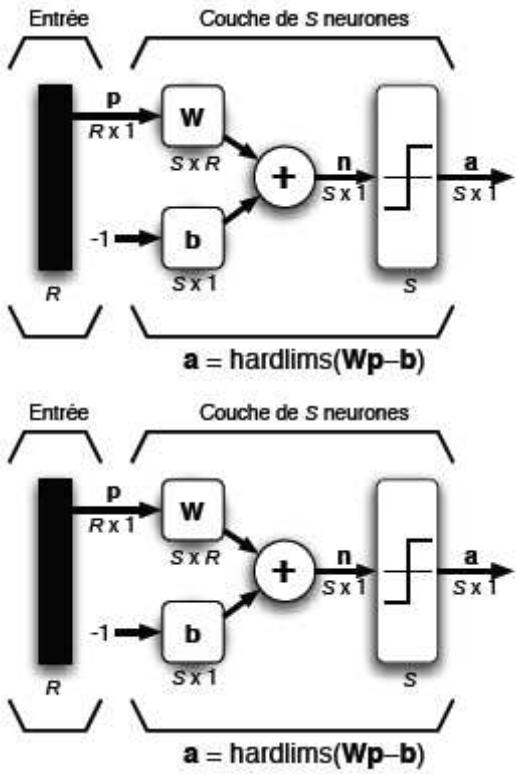
La méthode du perceptron simple permet de classifier des éléments compris dans deux catégories différentes. A condition que ces deux catégories soient linéairement séparables. Par la suite il est également possible de prédire à quelle catégorie appartient un élément.

II – PERCEPTRON PARALLELE

II.1 – Introduction

Nous allons à présent étudier deux perceptrons en parallèle. (Non liés)

Soit, deux Perceptron simple en parallèle :



Composition pour chaque perceptron :

- Deux entrées (dans notre cas, des coordonnées x et y d'un point)
- Un biais b, assimilable à une troisième entré, mais variable.
- Un système de trois poids liées à ces entrés sachant que le 3ème poids correspond au biais et est toujours égal à -1.
- Une fonction de transfert (ici nous utiliserons la fonction seuil)
- Une seule sortie

Cette fois, chaque point de notre domaine aura deux affinités, une pour chaque perceptron. On se donne donc un nombre N de points. Il y aura donc quatre possibilités : (1,1); (1,-1); (-1,1) ; (-1,-1) pour chaque point. Le domaine sera donc séparé par deux droites.

II.2 – Algorithme

- Initialisation des données :
  - Nbr = Nombre de points
  - Variables : utilisation de vecteurs de float pour les coordonnées des points, leurs affinités (1 ou -1) et les poids des deux perceptrons.
  - Initialisation des biais des perceptrons à 0.
  - Utilisation de fichier pour imprimer les données.

- Insertion des points pour la phase d'apprentissage :
  - On insère les coordonnées de chaque point et leurs affinités dans 4 vecteurs (X, Y, F1, F2). Ainsi le quadruplet [X(1),Y(1) ,F1(1),F2(1)] défini le 1er point.
  - Par soucis de simplicité, les poids des perceptrons auront comme valeur initiale l'affinité respective du 1er point.

- Phase d'apprentissage (Pour chaque point i) pour une perceptron :
  - Calcul de la somme : $S = x_i.w_1 + y_i.w_2 - 1.b$
  - Calcul de f(S) où f est la fonction seuil.
  - On compare le résultat obtenu à celui voulu (vecteur F)
  - Si le résultat est identique, on ne change pas les poids. S'il est différent, on modifie les poids de la façon suivante :

$$\begin{cases} w_1 = w_1 + \eta.(\frac{w_1 + x_i}{2}).F_i \\ w_2 = w_2 + \eta.(\frac{w_2 + y_i}{2}).F_i \\ b = b + \eta.(\frac{-1 + b}{2}).F_i \end{cases}$$

    eta étant le coefficient d'apprentissage.

  - On passe au point suivant

- Critère d'arrêt :
  - A chaque itération, les poids sont modifiés pour satisfaire chacun des points. Lorsque ces poids ne sont plus modifiés par aucun des points, on arrête le programme.
  - Il se peut que l'ensemble des points ne soient jamais satisfait par les poids. Ce sont les cas limite que je montrerai par la suite.

- Prédiction :
  - Une fois les poids bien calibrés (concrètement, le domaine est séparé en deux parties, les points 1 d'un coté, les -1 de l'autre), il est possible de rentrer les coordonnées d'un point pour connaître ses affinités.

II.3 – Résultats

Les coordonnées des points ainsi que leur affinité seront enregistré dans un fichier Points.dat. Les poids ainsi que la valeur de b seront quant à eux enregistré itération par itération dans un fichier Droites.dat pour le 1er perceptron et dans un fichier Droites2.dat pour le 2ème.

De la même manière, grâce au programme Matlab, on obtient les figures suivantes :

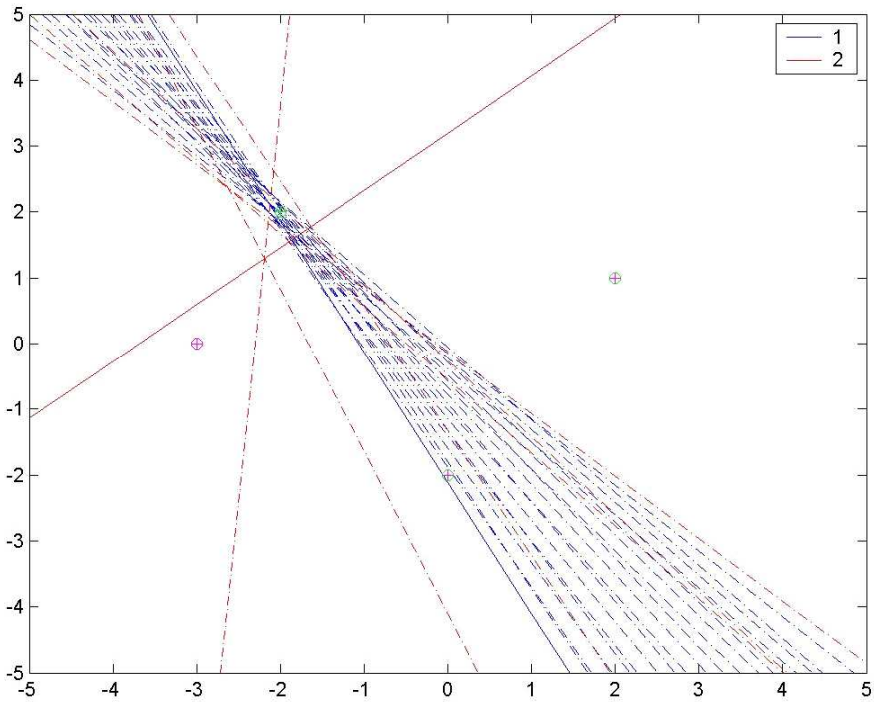
Légende pour les affinités des points :

- Cercle vert o : affinité de 1 pour le 1er perceptron.
- Cercle mauve o : affinité de -1 pour le 1er perceptron.
- Croix verte + : affinité de 1 pour le 2ème perceptron.
- Croix mauve + : affinité de -1 pour le 2ème perceptron.

Courbes bleus : 1er perceptron
Courbes rouges : 2ème perceptron

Les courbes pleines représentes les poids de la dernière itération (les deux perceptrons étant indépendant, ils n'ont pas le même nombre d'itération).

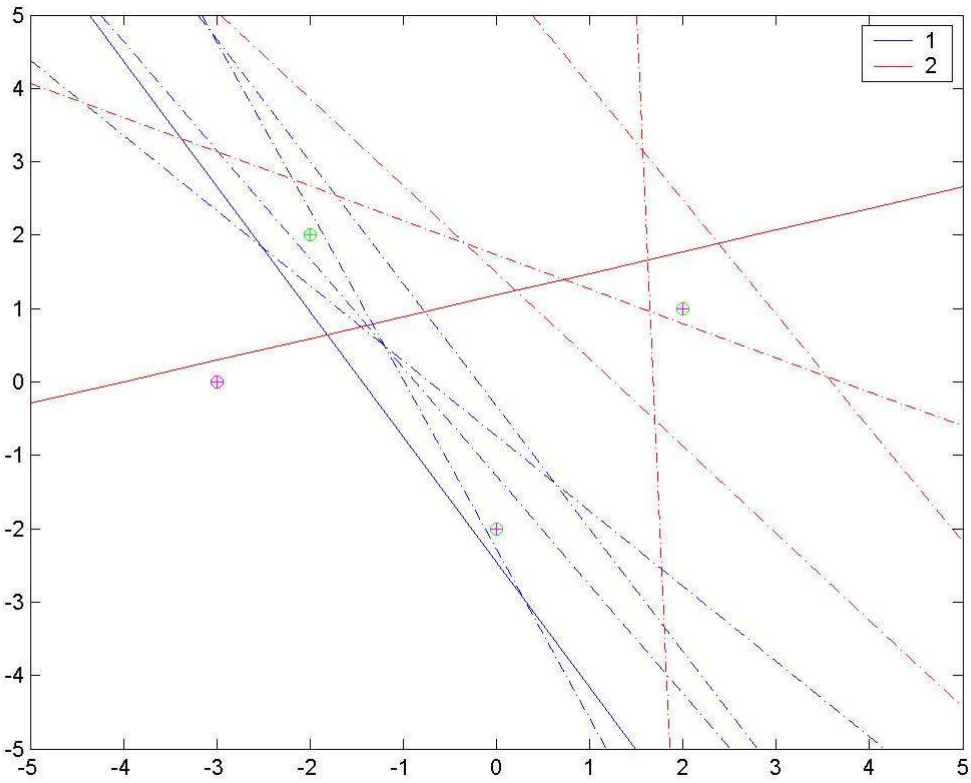
1er exemple : paramètres : 4 points, eta =0.1.



L'algorithme converge lentement. La courbe bleue sépare bien les cercles verts o du cercle mauve o. Et la courbe rouge sépare la croix verte + des Croix mauves +.

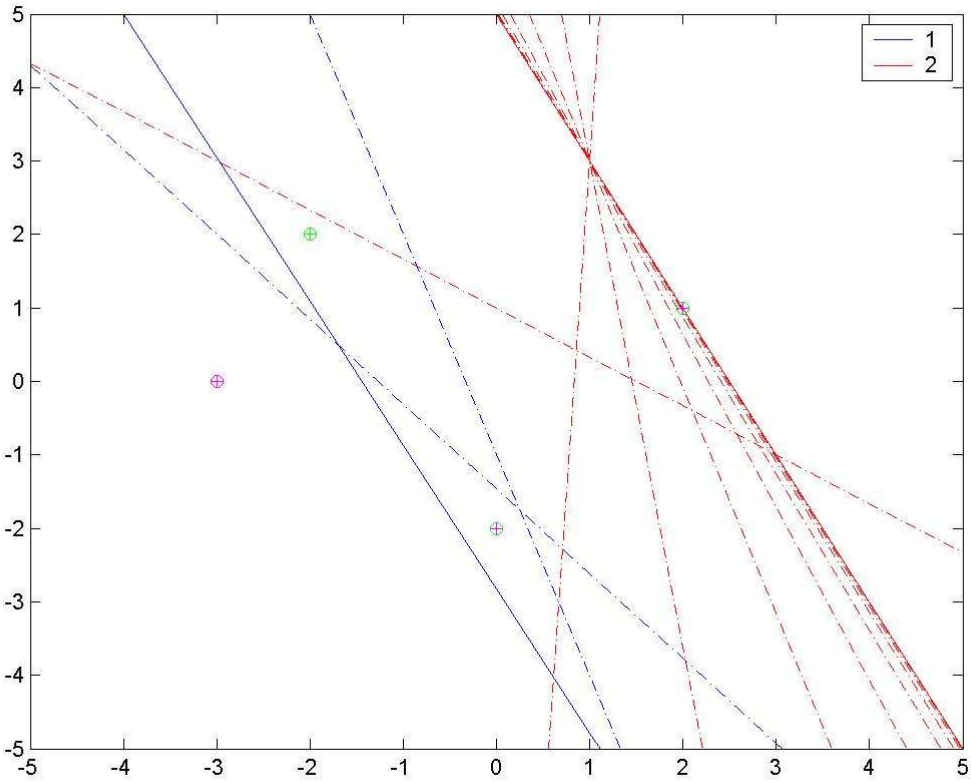
2ème exemple : paramètres : 4 points, eta=0.5.

On réutilise les mêmes points mais cette fois ci avec un coefficient d'apprentissage plus élevé :



L'algorithme converge plus rapidement, on constate bien que le biais est lui aussi modifié plus rapidement. En effet, les ordonnées à l'origine des courbes varient de manière importante.

3ème exemple : paramètres : 4 points eta =1.



On constate que le perceptron 1 converge très rapidement (courbe bleu séparant les cercles de couleurs). Alors que le perceptron 2 subit le phénomène de « sur apprentissage » et diverge. Comme nous l'avons vu précédemment, la variation des poids est trop importante et l'algorithme ne peut pas converger.

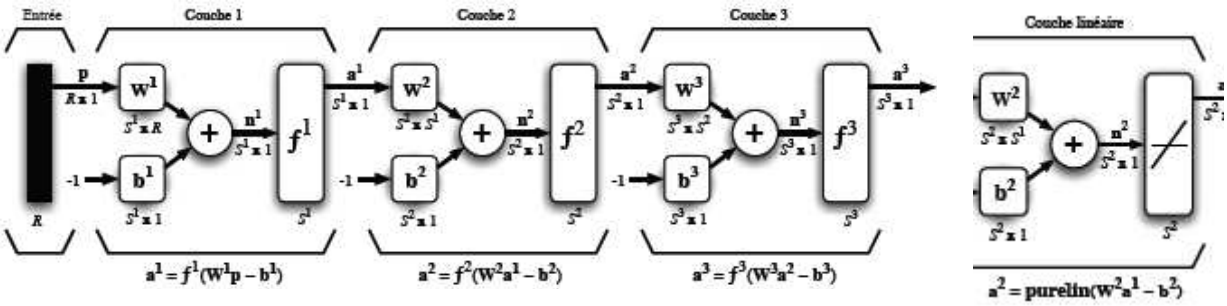
II.4 - Conclusion

Il s'agit de deux perceptrons indépendants. Il n'y a donc pas de différence par rapport a la partie précédente. Nous avons classé les éléments suivant deux critères : (1,1); (1,-1); (-1,1) ; (-1,-1). Il faudrait donc un perceptron de plus pour faire la synthèse et ainsi obtenir trois catégories dans ce cas précis.

III – PERCEPTRON MULTICOUCHE

III.1 – Introduction

Nous allons à présent étudier le perceptron multicouche de taille 3x3.



C'est-à-dire que les coordonnées des points sont rentrées dans la couche 1 qui représente 3 perceptrons en parallèles. Cette fois ci les fonctions utilisées sont des fonctions sigmoïdes.

Par la suite, les sorties de la couche k sont les entrés des perceptrons de la couche k+1. Puis, les trois sorties des perceptrons de la couche 3 sont récupérées par un dernier perceptron ayant une fonction linéaire comme fonction de transfert. Ce dernier perceptron fourni donc la valeur finale du perceptron multicouche.

Composition pour chaque perceptron :

- Deux entrées (dans le cas de la couche 1), trois entrées sinon.
- Un biais b variable.
- Un système de poids liées à ces entrés sachant que le dernier poids correspond au biais et est toujours égal à -1.
- Une fonction de transfert (la fonction sigmoïde)
- Une seule sortie.

Dans un premier temps nous testerons l'apprentissage des poids du perceptron multicouche pour un seul point. Puis pour un nombre N de point. Sachant que nous connaissons la valeur désiré de ces points. (exemple : approximation d'une fonction définie cos(x+y)

III.2 – Algorithme

Classe perceptron (Neurone dans le prog.) :

On défini le perceptron comme un objet, on lui associe :
Un vecteur poids, de taille variable, un biais b et la variable som qui vaut $\sum_{i=1}^{N} w_i.x_i - b$

On lui associe les fonctions suivantes :

- float Set_LVL_UP(Vecteur <float>, int);

Permet de faire passer un vecteur de coordonnées dans le perceptron. La fonction renvoi la valeur en sortie du perceptron. On peut choisir le type de fonction de transfert. (il faut que les poids soient déjà initialisé)

- void Set_Poids(Vecteur <float>);

Permet de modifier manuellement les poids. (sous forme d'un vecteur de taille variable + la valeur du biais)

- float Get_Resultat(Vecteur <float>);

Permet de calculer puis de récupérer la variable som.

- float Fonction_Sig(float);

Calcul de la valeur de la fonction sigmoïde en un point donné.

- int Fonction_Seuil(float);

Calcul de la valeur de la fonction seuil en un point donné.

Classe reseau :

Il s'agit en réalité d'un tableau de perceptron. (A la base de taille variable mais le codage n'a été testé que pour un perceptron de taille 3x3)

La classe comporte donc un tableau de taille 3x3 d'objet « perceptron » et un perceptron « final » qui se trouve en fin du réseau.

On lui associe également un coefficient d'apprentissage, et des matrices temporaires pour stocker les valeurs pour la rétropropagation de l'erreur.

Une matrice de taille 3x3 pour les variations des poids.

Une matrice de taille 3x3 pour les poids de la couche k+1 lors du calcul des poids de la couche k.
Une matrice de taille 3x3 pour stocker les valeurs des sorties de chaque perceptron du réseau.

On lui associe les fonctions suivantes :

- float Initialisation(Vecteur <float>);

Permet à l'aide d'un vecteur de coordonnées d'initialiser aléatoirement tout les poids des 10 perceptrons du réseau.

- void Bourrage_De_Crane(Vecteur <float>, Vecteur <float>, Vecteur <float>, float,float);

Permet En entrant un vecteur contenant les coordonnées x, un autre les coordonnées y et un dernier les valeurs des points voulues en sortie, ainsi que le coefficient d'apprentissage et la précision voulue, de calibrer les poids du réseau.

- float Propagation_Point(Vecteur <float>);

Permet de faire passer un point (2 coordonnées) dans le réseau et d'obtenir sa valeur en sortie.

- float Fonction_DSeuil(float);

Calcul de la valeur de la fonction dérivé de sigmoïde en un point donné.

- Vecteur <float> Sensibilite_Finale(float, float);

Calcul de la sensibilité du dernier perceptron « final ».

- Vecteur <float> Sensibilite(Vecteur <float>, int, Vecteur <float>);

Calcul de la sensibilité de la couche k en fonction de la couche k+1.

- void Variation_Poids_Initial(Vecteur <float>, Vecteur <float>, float);

Calcul des poids initiaux de la 1ère couche prenant en entré les coordonnées des points

- void Variation_Poids(Vecteur <float>, float, int);

Calcul des poids de la couche k en fonction de la sensibilité de la couche k et des sorties de la couche k-1.

Algorithme :

- Initialisation des données :
  - Nbr = Nombre de points
  - Variables : utilisation de vecteurs de float pour les coordonnées des points et leur valeur voulue.
  - Utilisation de fichier pour imprimer les données.

- Initialisation du réseau :
  - On insère les coordonnées d'un point dans le réseau
  - On donne des poids aléatoire à la première couche. Puis à l'aide des fonctions de la classe perceptron, on calcul la sortie des perceptrons de la couche 1. Ces sorties sont stockées dans la matrice A.
  - La couche suivante récupère ces sorties et les prend en valeur d'entrées. Sachant que si la couche 1 est composé de 3 perceptrons en parallèles. Alors il y aura 3 entrées par perceptron de la couche suivante. Et ainsi de suite pour les couches suivante.
  - A la dernière couche on récupère les 3 sorties et le perceptron « finale » nous renvoi une seule valeur (sachant qu'il est composé d'une fonction de transfert linéaire).
  - Tout les biais sont initialisé à 0.

- Phase d'apprentissage Pour chaque point :
  - On fait passer les coordonnées de ce point dans le réseau.
  - On compare le résultat obtenu (variable a) à celui voulu (vecteur D)
  - On calcul l'erreur commise (sensibilité) : S^M=-2x(d-a)
  - On calcul les sensibilité de la couche k :

$$\mathbf{s}^k = \dot{\mathbf{F}}^k\left(\mathbf{n}^k\right)\left(\mathbf{W}^{k+1}\right)^T \mathbf{s}^{k+1}$$

  - Puis on met à jour les poids et les biais de la couche k :

$$\Delta\mathbf{W}^k = -\eta\mathbf{s}^k\left(\mathbf{a}^{k-1}\right)^T$$

$$\Delta\mathbf{b}^k = \eta\mathbf{s}^k$$

  - Une fois tout les poids mis à jour on passe au point suivant.
- Critère d'arrêt :
  - A chaque itération, les poids des perceptrons sont modifiés pour satisfaire à chacun des points. Lorsque ces poids ne sont plus modifiés par aucun des points, on arrête le programme Comparaison avec une valeur epsilon très faible.

- Prédiction :
  - Une fois les poids bien calibrés il est possible de rentrer les coordonnées d'un point pour connaître sa valeur en sortie du réseau.

En réalité, pour un perceptron de taille 3x3, j'ai du dissocier la couche initiale des autres couches car elle ne reçoit pas le même nombre d'entré que les autres.

Un autre problème est la gestion des anciens poids. En effet, lors de la rétropropagation de l'erreur, au moment où les poids de la couche k+1 sont mis à jour, il faut garder les anciennes valeurs des poids, car elles sont nécessaire au calcul de la sensibilité de la couche k.

III.3 – Résultats

Le réseau ne convergeant pas toujours très bien. Je n'utilise pas de critère d'arrêt pour le moment mais une simple boucle avec un nombre d'itération donnée.

1er exemple : Pour un point (5,2) valant 10, eta =1

Pour une valeur de eta trop forte, l'algorithme diverge. En effet, on trouve les valeurs suivantes :

Pour 1 itérations : 64.9
Pour 10 itérations : 410
Pour 100 itérations : 1310

En regardant de plus près les valeurs des poids, on constate que les poids des dernières couches explosent.

2ème exemple : Même point, eta=0.4.

Pour 1 itérations : 1.94
Pour 10 itérations : 10.0004
Pour 100 itérations : 10

3ème exemple : Pour plusieurs points, eta=0.4.

On effectue un test trivial :
Y=2*X

On prend 10 valeurs de la fonction qu'on apprend au réseau.

Apparemment il y a une erreur dans le programme car l'algorithme semble converger d'après l'affichage de la console.

Mais Lorsque je lui demande de sortir une valeur d'un point, par exemple : 1,1 il donne : 18,5 et ce pour n'importe qu'elle valeur de point.

Je n’ai pas eu le temps de debugger ce cas avant l’échéance.

III.4 - Conclusion

On constate qu’il ne faut pas utiliser de coefficient d’apprentissage trop élevé, sinon le réseau diverge complètement. Dans notre cas, pour une valeur de eta supérieur à 0.5, l’algorithme n’est pas très stable. Cependant il ne faut pas prendre de valeur trop faible non plus, sinon le programme ne converge presque pas. Ou alors pour un nombre d’itérations très grand.

L’algorithme tourne bien pour un seul point. Par la suite, ce type de programme permettrait d’approximer une fonction complète.