République Algérienne Démocratique et Populaire

Ministère de l'enseignement et de la Recherche Scientifique

Université Mentouri de Constantine

Faculté des Sciences de l'ingénieur

Département d'électrotechnique

N° Ordre : .....
Série : .....

Thèse Présentée pour l'obtention du Titre de MAGISTÈRE en

ÉLECTROTECHNIQUE

Par : M. DRAIDI ABDELLAH

Titre de la Thèse :

Répartition économique de l'énergie électrique utilisant les techniques d'intelligence artificielle

Devant le jury composé de :

Président :	M. Benalla Hocine	Professeur	Université de Constantine
Rapporteur :	M. Labed Djamel	Maître de conférences	Université de Constantine
Examinateur :	M. Bouzid Aissa	Professeur	Université de Constantine
Examinateur :	M. Boutamina Brahim	Maître de conférences	Université de Constantine

Année : 2010

Je dédie ce modeste travail à la mémoire de ma grand-mère

À toute ma famille

À tous mes amis…

Je tiens à exprimer ma profonde gratitude à Monsieur Labed Djamel, Maître de conférences à l'université de Constantine, Pour m'avoir accueilli et accepter au sein de son équipe et pour avoir dirigé ce travail.

Je remercie sincèrement Monsieur Peter Kelen, propriétaire de Power Optimisation Software en Angleterre, pour l'aide impressionnante en matière de documentation et sites utiles.

Je tiens à remercier Monsieur Benalla Hocine Professeur à l'université de Constantine, pour m'avoir fait l'honneur de présider mon jury. Ainsi que tous les membres du Jury.

Je remercie Monsieur Boucharma Mohamed, Maître de conférences en électrotechnique, pour son aide.

Enfin, je remercie tout particulièrement mes parents, pour leur soutien inconditionnel tout au long de ces longues années d'études…

Table des matières

INTRODUCTION GÉNÉRALE :

Le développement de l'informatique, de l'automatisme et des interfaces homme machine a créé une innovation dans le domaine de génie électrique et surtout dans les réseaux électriques ; on parle aujourd'hui de réseaux électriques intelligents « smart grids » qui sont une combinaison d'une infrastructure électrique avec une intelligence embarquée que l'on peut lui associer (logiciel, automatismes, transmission et traitement de l'information).

Cette évolution du réseau électrique est liée à divers facteurs, parmi lesquels on peut citer : les préoccupations environnementales de nos sociétés modernes, les tensions sur les énergies primaires et les conséquences sur la sécurité d'approvisionnement, le vieillissement des infrastructures, l'ouverture des marchés de l'énergie, la multiplication d'acteurs conjuguée à l'accès non-discriminatoire au réseau, Ces facteurs se sont renforcés ces dernières années en devenant de plus en plus des éléments de levier des changements à venir dans le système électrique.[1]

L'introduction des techniques de l'intelligence artificielle, dans les logiciels de commande et de décision est un élément essentiel dans la recherche et dans le développement des réseaux de demain. Ces techniques sont principalement : les réseaux de neurones, la logique floue, les algorithmes génétiques,…, etc.

Les réseaux de neurones figurent parmi les techniques les plus répondues dans le domaine de l'intelligence artificielle, pour cela ils sont utilisés dans plusieurs programmes et logiciels de commande des réseaux électriques ; dans des domaines d'application variés : la stabilité, la sécurité, le diagnostic des défauts, la protection, le dispatching économique, l'analyse des harmoniques, l'écoulement des puissances, la prédiction de la demande électrique, la prédiction des pannes [2],…, etc.

Le chapitre 01 (état de l'art) résume quelques exemples illustrant l'utilisation des techniques d'intelligence artificielle dans les réseaux électriques.

La répartition économique de l'énergie électrique, ou le dispatching économique est un secteur d'étude essentiel dans les réseaux électriques, où on doit générer moins d'énergie pour la même demande, avec une bonne gestion et une distribution pré-calculée des paquets d'énergie avec le moindre coût possible, en diminuant les pertes d'énergie donc la diminution de la production totale, c'est-à-dire moins de combustible consommé, cela apporte des gains économiques énormes avec une baisse du prix du kWh et une bonne contribution à la

préservation de l'environnement par la réduction des émissions en CO_2(centrales thermiques), la diminution des déchets nucléaires (centrales nucléaires),…, etc.

Nous allons traiter dans notre thèse le dispatching économique utilisant les réseaux de neurones, où on va opter pour un dispatching économique plus rapide et plus fiable par rapport à celui fait avec un programme classique.

Le chapitre 02 traite la solution du dispatching économique avec les méthodes de calcul classiques, où on va définir le dispatching économique, le 'unit commitment' et la fonction coût d'une unité de production et comment la déterminer avec des méthodes d'analyse de régression à partir des données production / coût disponible. Ensuite, on va étudier la solution du dispatching économique avec les méthodes d'optimisation de Kuhn-Tucker, la méthode du Gradient, et la méthode d'itération de Lambda. La formule des pertes linéiques sera ensuite ajoutée à la solution du dispatching économique (deux approches possibles) à partir des équations de l'écoulement de puissance des réseaux maillés, où on va calculer les pertes et le facteur de pénalité pour construire notre programme classique de dispatching économique avec pertes. Finalement, on va donner une petite description illustrant le transite de l'énergie réactive dans le réseau.

Au chapitre 03 on va donner une définition et un historique des réseaux de neurones, avec des notions sur les neurones biologiques. Ensuite on va traiter les modèles de neurones artificiels et les fonctions de transfert, les règles d'apprentissage supervisé et les taches d'apprentissage. Après on définit le perceptron monocouche avec leur limitation et le perceptron multicouche qui donne les solutions suite à ces limitations. Pour minimiser l'erreur, on va s'intéresser à la méthode de rétropropagation de l'erreur, celle qu'on va utiliser dans notre programmation Matlab (*newff*) en donnant l'algorithme d'entrainement et toutes les caractéristiques telles que le calcul des sensibilités, les critères d'arrêt, le phénomène de saturation, le groupage, le momentum et le taux d'apprentissage.

Dans l'étude pratique du chapitre 04, le but est de pouvoir créer un réseau de neurones capable de reproduire les résultats du programme classique de dispatching économique en bénéficiant des avantages des réseaux de neurones. La programmation se divise en deux : programmation classique (avec des boucles while, if, for) qui traite l'énergie demandée par paquets en prenant en compte les pertes linéiques ; programmation des réseaux de neurones qui utilise comme données d'apprentissage les résultats de la programmation classique.

Trois exemples vont être étudiés, le premier sera le dispatching économique sans pertes d'un réseau de trois centrales (Skikda, Annaba et Fkirina) alimentant une charge (Constantine) ; le même réseau sera étudié en incluant le calcul des pertes des lignes, cette étude sera faite avec un programme classique puis par un programme de réseau de neurones à rétropropagation de l'erreur. Le dernier exemple sera le dispatching économique d'un réseau maillé IEEE de 9 jeux de barres dont 3 centrales et 5 charges avec le programme classique puis par un programme de réseau de neurones à rétropropagation de l'erreur. Finalement on va comparer les deux programmes en termes de vitesse d'exécution et de fiabilité.

Deux Appendices vont être associés, l'appendice A donne quelques exemples des domaines d'utilisation de l'intelligence artificielle et l'appendice B traite La règle LMS (règle d'apprentissage d'un réseau supervisé monocouche).

CHAPITRE 01

État de l'art

L'utilisation des techniques d'intelligence artificielle dans les réseaux électriques est un domaine d'étude et d'application très vaste et varié pour les chercheurs qui optent pour un réseau très sûr, économique et stable. Nous allons présenter quelques exemples non exhaustifs illustrant ces recherches.

Exemple 01:

ECONOMICAL DISPATCH OF POWER UNITS UNDER FUZZINESS

Abstract «*The problem of economical dispatch of condensing power units on the basis of fuzzy information about units' characteristics is considered in the paper. The goal of optimization is minimization of maximum losses caused by uncertainty and fuzziness of the initial information. The paper introduces fuzzy models of characteristics, presents a mathematical model of the problem, the conditions of optimality and the method of solution. Some examples of calculations are presented. Especially important is economical dispatch of power units at coal and oil shale-fired power plants*».

Source: M. VALDMA, M. KEEL, H. TAMMOJA, J. SHUVALOVA , *Oil Shale, 2007, Vol. 24, No. 2 Special, pp. 249–263.*

Dans cet article l'auteur traite le dispatching économique utilisant la logique flou en minimisant les pertes maximales.

Exemple 02:

REPARTITION ECONOMIQUE DES PUISSANCES PAR UN ALGORITHME GENETIQUE EN CODE REEL

Résumé «*Dans cet article, nous présentons une solution au problème de la répartition économique des puissances actives(REPA) basée sur un algorithme génétique. Notre objectif est de minimiser Le coût du combustible nécessaire pour la production de l'énergie électrique qui se présente sous forme d'une fonction non linéaire, en tenant compte de certaines contraintes de type égalité et inégalité. Le choix d'un codage approprié est un élément critique dont dépend grandement l'efficacité d'un algorithme génétique. Pour cela nous présentons une étude comparative entre deux types de codage : le codage binaire classique et le codage réel*».

Source: M. YOUNES, M. RAHLI, M. KANDOUCI, *Acta Electrotechnica, Vol 50, No. 2, 2009, pp. 128-136.*

Dans cet article l'auteur fait introduire les algorithmes génétiques dans la solution du dispatching économique.

Exemple 03:

POWER SYSTEM STATIC SECURITY ASSESSEMENT USING THE KOHONEN NEURAL NETWORK CLASSIFIER

Abstract « *This paper presents the application of an artificial neural network, kohonen's self-organizing feature map, for the classification of power system states. This classifier maps vectors of an N-dimensional space to a 2-dimentional neural net in a nonlinear way preserving the topological order of the input vectors. Therefore, secure operating points, that is vector inside the boundaries of these cure domain are mapped to a different region of the neural map than insecure operating points* ».

Source: D. Niebur, A.J. Germond, *IEEE Transactions on Power Systems, Vol. 7, No. 2, 1992, pp. 865 - 872*

Cet article traite l'évaluation de la sécurité d'un réseau électrique utilisant un réseau de neurones avec apprentissage non supervisé.

Exemple 04:

COMPARISON OF DYNAMIC LOAD EXTRAPOLATION USING NEURAL NRTWORKS AND TRADITIONAL METHODS.

Abstract« *Up to now the representation of load dynamic characteristics continues to be an area of great uncertainty and it becomes a limiting factor of power systems dynamic performance …this paper deals with groups of data measured in Chinese power systems using two models: a multi-layer feedforward neural network using back propagation learning, and difference equations using recursive extended least square identification …*»

Source: He Ren-mu. A.J. Germond, *Neural network computing for the electric power industry, proceeding of the 1992 INNS summer workshop, pp. 70-80.*

Cet article traite la stabilité dynamique des réseaux électriques en comparant la méthode des réseaux de neurones avec la méthode classique.

Exemple 05:

NEURAL NETWORK SYNTHESIS OF TANGENT HYPERSURFACES FOR TRANSIENT SECURITY ASSESSEMENT OF ELECTRIC POWER SYSTEMS

Abstract« *In this paper a new direct method for transient security assessement of multimachine power system is presented. A local approximation of the stability boundary is*

made by tangent hypersurfaces … Artificial neural networks are used to determine the unknown coefficients of the hypersurfaces independently of operating conditions ».

Source: D.J. Sobajic, Y. Pao, *Neural network computing for the electric power industry, proceeding of the 1992 INNS summer workshop, p. 87-92.*

Cet article traite la sécurité des réseaux électriques utilisant les réseaux de neurones.

Exemple 06:

VOLTAGE STABILITY MONITORING BY ARTIFICIAL NEURAL NETWORK USING A REGRESSION-BASED FEATURE SELECTION METHOD

Abstract« *This paper proposes a methodology for online voltage stability monitoring using artificial neural network (ANN) and a regression-based method of selecting features for training the ANN …*»

Source: S. CHAKRABARTI, *Science Direct Expert Systems with Applications, Vol. 35, No. 4, Nov2008, pp. 1802-1808.*

Dans cet article, l'auteur traite la stabilité du voltage utilisant les réseaux de neurones.

Exemple 07:

SAIC UNVEILS SOFTWARE ABLE TO PREDICT GRID FAILURES

«Science Applications International Corp. (SAIC) has completed work on new software that will predict distribution and transmission system failures days, weeks and even months before they happen. The software, Distribution Monitoring Systems, took almost a year of research before it was ready for release… An artificial intelligence module, a neural network, that can determine from continuing patterns of equipment aberrations, even small ones, if failure will occur sometime in the future. The neural networks generally deliver more accurate predictions the longer they are in operation. »

Source: *News and analysis for the modernization and automation of electric power.* http://www.smartgridnews.com/artman/publish/Delivery_Distribution_Automation_News/SAIC-Unveils-Software-Able-to-Predict-T-D-System-Failures-1401.html, 10/11/2009.

La société américaine SAIC a mis au point un logiciel à base des réseaux de neurones pour prédire les pannes à partir des données recueillies sur les dysfonctionnements passés.

CHAPITRE 02

SOLUTION DU DISPATCHING ÉCONOMIQUE AVEC LES MÉTHODES DE CALCUL CLASSIQUES

2.1. Dispatching économique

Le dispatching économique est un problème d'optimisation statique qui consiste à répartir la production de la puissance active demandée entre les différentes centrales du réseau, de sorte à exploiter ce dernier de la manière la plus économique possible. Cette distribution doit évidemment respecter les limites de production des centrales. La variable à optimiser est donc le *coût de production*.

Le problème du dispatching économique sans perte est peu complexe car le seul paramètre qui influence le coût est la puissance active générée par la centrale (sans tenir compte de la puissance perdue dans les lignes lors des transits de puissance entre les centrales et les charges). Un autre problème d'optimisation statique, l' « optimal power flow » [1](OPF)

Une autre limitation du dispatching économique est l'aspect statique du problème. En effet, quand on résout un dispatching économique, on le fait pour une demande à un instant précis. Lorsque le problème prend une dimension dynamique, c'est-à-dire lorsque la demande évolue dans un intervalle de temps donné (une journée par exemple), il faut alors tenir compte des états des centrales ainsi que des changements d'états qui occasionnent des coûts supplémentaires. Par exemple, si la demande augmente au court du temps, il faudra sans doute faire fonctionner une centrale qui était à l'arrêt afin de satisfaire cet accroissement de la demande, et le coût pour faire démarrer cette centrale doit être pris en compte dans l'optimisation. Le traitement d'un tel problème est appelé « unit commitment ». [3]

2.2. Unit commitment

Le « Unit commitment » ou '*la planification de l'opération des unités de production'* [4], est le processus de décider quand et quelle unité de génération doit fonctionner ou pas, donc on doit programmer les générateurs ('*on*' ou '*off*') pour répondre aux charges nécessaires à un coût minimum soumis aux pertes du réseau. [5]

2.3. La fonction coût

Le coût de production d'une centrale est généralement modélisé par une fonction polynomiale du second degré en P_G (puissance active générée par la centrale) dont les coefficients sont des constantes propres à chaque centrale :

[1] *l'objectif de l'OPF est de garder une limite de performance acceptable de la puissance active et réactive de sortie des générateurs, du voltage des jeux de barres, des capacitances et réactances shunt, réglage des transformateurs , et de l'écoulement de puissance des lignes de transmission*. [6]

$$C_{centrale}(P_G) = a + bP_G + cP_G^2 \quad (2.1)$$

Dans le cas où ces coefficients (a, b et c) ne sont pas connus et les données des unités de production sont en forme de tableaux associant chaque puissance active générée à son cout de production. On doit utiliser plusieurs techniques mathématiques pour déterminer les constantes *a*, *b* et *c* comme : l'analyse de régression « *regression analysis* » avec la méthode des moindres carrés « *least-squares method* », la logique flou «*fuzzy logic* » etc. [3]

2.3.1. Analyse de régression par la méthode des moindres carrés

La méthode des moindres carrés permet de comparer des données expérimentales, généralement entachées d'erreurs de mesure à un modèle mathématique censé décrire ces données.

Cette méthode est attribuée au mathématicien allemand *Carl Friedrich Gauss (1809)*. Sous certaines hypothèses, *les moindre carrés* ordinaires ont quelques propriétés statistiques très séduisantes, propriétés qui ont rendu la méthode l'une des plus populaires de *l'analyse de régression*. [7]

Soit $y = f(x, a, b, c, \dots)$ l'équation de la courbe que l'on cherche à ajuster au nuage statistique (*figure 2.1*). Nous voudrions que les erreurs entre la valeur *observée* y_i et la valeur *ajustée* $f(x_i)$ soit *minimale*. Appelons e_i la différence :

$$e_i = y_i - f(x_i) \quad (2.2)$$

e_i est le résidu de la $i^{ème}$ observation et sa valeur absolue représente la distance entre les points $M_i(x_i, y_i)$ et $P_i\big(x_i, f(x_i)\big)$.

Les résidus étant positifs ou négatifs, leur somme peut être de faible valeur pour une courbe mal ajustée. On évite cette difficulté en considérant *la somme des carrés des résidus*.

Cette somme :

$$S(a, b, c, \dots) = \sum_{i=1}^{n} e_i^2 \quad (2.3)$$

dépend des paramètres *a*, *b*, *c*,… à ajuster. On choisira ces paramètres de manière qu'elle soit *minimale*. $\sum_{i=1}^{n} e_i^2$ est appelé *variation résiduelle* et nous donne une mesure de l'ampleur de l'éparpillement des observations y_i autour de la courbe d'ajustement. [8]

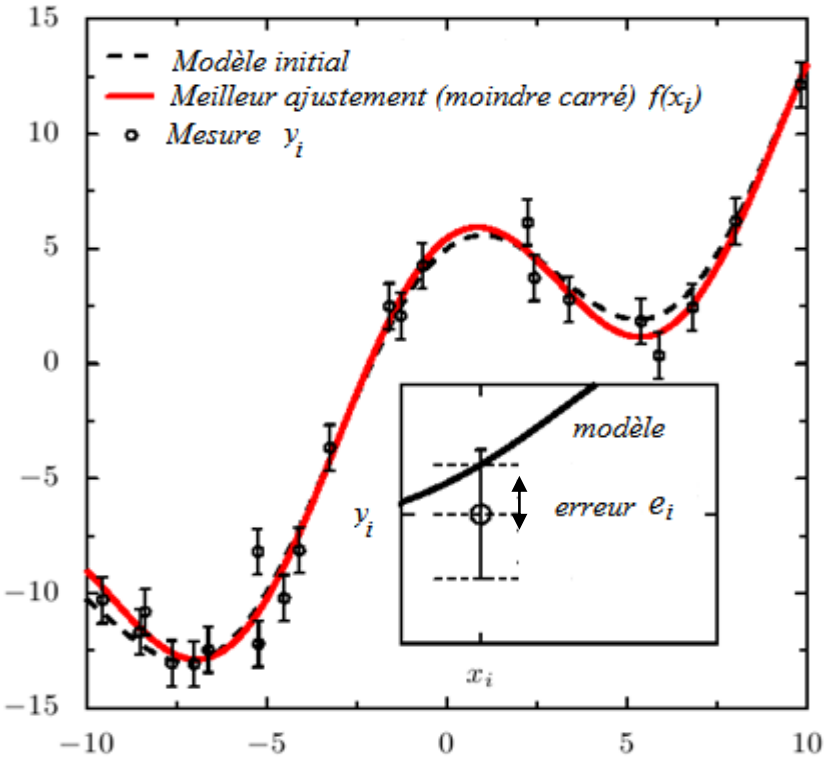

FIG.2.1– *Le meilleur ajustement déterminé par la méthode des moindres carrés est représenté en rouge (ligne épaisse continue). Il s'agit de la fonction qui minimise la somme quadratique des écarts (appelés résidus) entre les données et le modèle.* [9]

Application aux centrales électriques

Les propriétés des centrales électriques comme *la puissance* et *le coût* du carburant peuvent être exprimées en tant que variables en mathématique. Dans cet exemple comme dans bien d'autres, on peut établir un lien entre ces variables. En format mathématique, ce lien est exprimé généralement par une équation polynomiale du deuxième degré (équation du coût).

Pour déterminer l'équation qui relie les variables, on doit d'abord découvrir dans l'échantillon, les couples de valeurs qui correspondent à ces variables donc les variables *x* et *y* pourraient représenter respectivement la puissance et le coût correspondant d'un générateur. Donc, pour un échantillon N, on obtiendrait les valeurs $x_1, x_2, x_3, \ldots, x_N$ *et* $y_1, y_2, y_3, \ldots, y_N$.

L'ensemble des points répartis sur un système d'axes donne l'aspect d'une nuée de particules de poussière qui se disperse dans l'air. Ce système est d'ailleurs appelé diagramme de dispersion (*figure 2.2*). Dans la plupart des cas, on peut, à partir de ce diagramme, faire la représentation d'une courbe continue autour de laquelle les points semblent se regrouper. Cette courbe est nommée *courbe d'ajustement* en utilisant la méthode des moindres carrés.

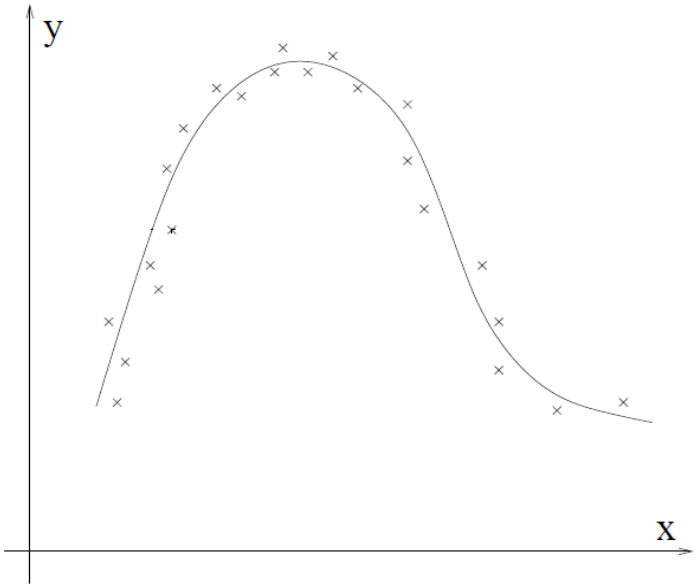

FIG.2.2– *fonction de 2ème degré d'une forme parabolique*.

Dans le cas d'une parabole ajustant les points (X_1, Y_1), (X_2, Y_2),…, (X_N, Y_N) l'équation est:

$$Y = a_0 + a_1 X + a_2 X^2 \tag{2.4}$$

Les constantes a_0, a_1 et a_2 peuvent être déterminées en résolvant les équations suivantes simultanément:

$$\begin{cases} \sum Y = a_0 N + a_1 \sum X + a_2 \sum X^2 \\ \sum XY = a_0 \sum X + a_1 \sum X^2 + a_2 \sum X^3 \\ \sum X^2 Y = a_0 \sum X^2 + a_1 \sum X^3 + a_2 \sum X^4 \end{cases} \tag{2.5}$$

Tel que : $a_0 = a$

$a_1 = b$

$a_2 = c$ et N= le nombre de couples de valeurs

Où : a, b *et* c sont les constantes caractérisant chaque alternateur ou unité de production. [10]

2.3.2. Analyse de régression par la Logique floue (régression floue)

Dans l'analyse de régression conventionnelle, la déviation entre les valeurs observées et celles estimées est censée être due aux erreurs aléatoires, néanmoins, généralement cette déviation est due à l'absence de définition de structures de système ou à des observations imprécises. Donc cette déviation est *flou* « fuzzy » et ce type de régression doit minimiser ce flou. Les études concernant la régression floue linéaire peuvent être classées en deux approches :

1) la méthode basée sur la programmation linéaire.
2) la méthode des moindres carrés flous (Fuzzy least squares method). [11]

2.4. Solution du dispatching économique sans pertes

La solution du dispatching économique est obtenue à l'aide de deux types de méthodes d'optimisation, le premier type utilise le gradient (fonction de Lagrange) comme : La méthode de Kuhn-Tucker et la méthode de gradient. Le deuxième type utilise les itérations (minimisation sans gradient) : la méthode d'itération de Lambda. Dans certains cas on peut considérer le problème de dispatching comme linéaire par parties (Piecewise linear)

Si on considère le problème du dispatching comme un problème linéaire on utilise la programmation linéaire.

2.4.1. La méthode de Kuhn-Tucker

On peut constater que le problème d'optimisation est non-linéaire et soumis à des contraintes d'égalité et d'inégalité. En effet, il faut que

$$\sum_i P_{Gi} = P_D \tag{2.6}$$

et que

$$P_{Gi} - P_{Max\,i} \le 0 \text{ et } P_{Min\,i} - P_{Gi} \le 0\,. \tag{2.7}$$

tel que :

P_{Gi}: La puissance générée dans la centrale i

P_D: La puissance demandée.

$P_{Max\,i}$: La puissance maximale générée dans la centrale i

$P_{Min\,i}$: La puissance minimale générée dans la centrale i

La méthode de« Kuhn-Tucker » consiste à construire le *Lagrangien* qui tient compte des contraintes d'égalité et des contraintes d'inégalité :

$$L(x,\lambda,\beta) = f(x) + \lambda.h(x) + \sum_i \beta_i.g_i(x)$$
$$\Rightarrow L(P_{Gi},\lambda) = C + \lambda(P_D - \sum_i P_{Gi}) \quad (2.8)$$

où $f(x) = C$ est la *fonction à optimiser*,

$h(x) = P_D - \sum_i P_{Gi}$ est la *contrainte d'égalité* mis sous la forme $h(x) = 0$,

et $g(x)$ sont les *contraintes d'inégalité (équation 2.7)* mis sous la forme $g(x) \leq 0$

Notre fonction à optimiser est bien entendu *le coût total* défini par

$$C = \sum_i C_i(P_{Gi}) \text{ avec } C_i(P_{Gi}) = a_i + b_i.P_{Gi} + c_i.P_{Gi}^2 \quad (2.9)$$

Pour ensuite atteindre l'optimum, il suffit pour commencer de l'évaluer en négligeant les contraintes d'inégalité ($\beta_i = 0$). Si cet optimum vérifie les contraintes d'inégalité, il s'agit de la solution recherchée. Dans le cas contraire, on transforme certaines inégalités non-vérifiées en égalités (pour imposer ces inégalités à leurs limites) et on recalcule un nouvel optimum en tenant compte de ces nouvelles égalités. L'optimum sera atteint dès que toutes les contraintes d'inégalités seront vérifiées.

En effet, pour trouver le premier optimum des PGi (en négligeant donc les contraintes d'inégalité), il faut dériver notre Lagrangien en fonction des PGi et du coefficient de Lagrange λ, et annuler ces dérivées de sorte à obtenir les conditions sur l'optimum suivantes :

$$\frac{\partial L}{\partial P_{Gi}} = \frac{dC_i}{dP_{Gi}} - \lambda = 0 \quad (2.10)$$

$$\frac{\partial L}{\partial \lambda} = P_D - \sum_i P_{Gi} = 0 \quad (2.11)$$

La dérivée $\frac{dC_i}{dP_{Gi}}$ est connue sous le nom de « coût incrémental ». Elle représente l'accroissement du coût correspondant à la production d'une unité de puissance supplémentaire.

Si on reprend la première condition, on peut calculer :

$$\lambda = \frac{dC_i}{dP_{Gi}} = b_i + 2c_i P_{Gi}$$

$$\Rightarrow \quad P_{Gi} = \frac{(\lambda - b_i)}{2c_i} \tag{2.12}$$

En reprenant ensuite la seconde condition, on a :

$$P_D = \sum_i P_{Gi} = \sum_i \frac{(\lambda - b_i)}{2c_i} \tag{2.13}$$

$$\Rightarrow \quad P_{Gi} = \frac{1}{2c_i}\left(\left(\sum_i \frac{1}{2c_i}\right)^{-1}\left(P_D + \sum_i \frac{b_i}{2c_i}\right) - b_i\right) \tag{2.14}$$

L'expression (2.14) nous donne donc l'ensemble des P_{Gi} minimisant le coût total (contraintes d'inégalité négligées) et constituant notre premier optimum, n'est pas calculable dans le cas ou c_i est nul. Or ce coefficient pourrait être nul pour quelques centrales.

Nous arrivons donc à la conclusion que la méthode d'optimisation de « Kuhn-Tucker » n'est pas adaptée à tous les problèmes. [3]

2.4.2. La méthode du gradient

Le problème consiste donc à trouver un minimum global de la fonction erreur E entre la fonction *f(t)* et les points « *target* ». La méthode du gradient est l'une des principales méthodes pour déterminer cette région d'optimisation.

En effet, afin de minimiser une fonction à partir d'une solution approchée, le plus simple est de suivre la ligne de *plus grande pente*. D'un point de vue mathématique, la pente d'une fonction correspond à la dérivée de cette dernière. Si l'on se place dans le cadre d'une fonction ayant plusieurs paramètres, la dérivée devient un vecteur : le gradient de la fonction. Chaque élément de ce vecteur correspond alors à la dérivée partielle de la fonction selon l'un de ses paramètres.

Soit f une fonction (suffisamment dérivable) dont on recherche un minimum. La méthode du gradient construit une suite x_n qui doit en principe s'approcher du minimum. Pour cela, on part d'une valeur quelconque $x_0$et l'on construit la suite :

$$x_{n+1} = x_n - \rho f'(x_n) \tag{2.15}$$

où ρ est une valeur "bien" choisie réelle non nulle.

On a $f(x_{n+1}) = f\big(x_n - \rho f'(x_n)\big) \approx f(x_n) - \rho\big(f'(x_n)\big)^2$ d'après le théorème des approximations finies si $\rho f'(x_n)$ est "suffisamment" petit.

On voit que, sous réserve de la correction de l'approximation, $f(x_{n+1})$ est inférieur à $f(x_n)$.

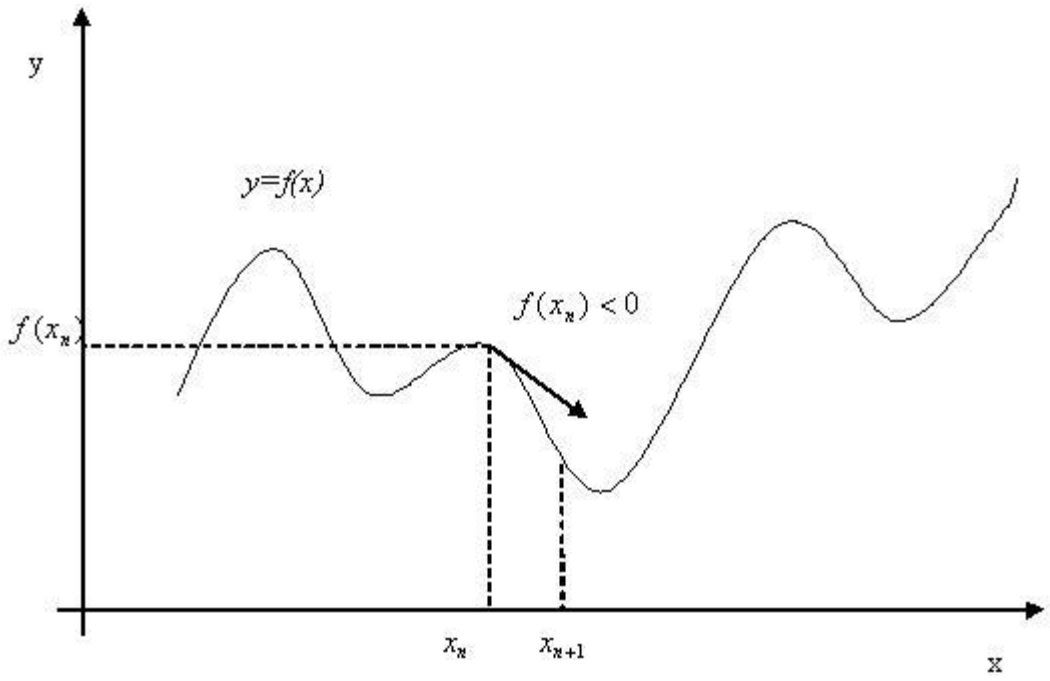

FIG.2.3– *La méthode du gradient*

On remarque que x_{n+1} est d'autant plus éloigné de x_n que la pente de la courbe en $x_n$est grande. On peut décider d'arrêter l'itération lorsque cette pente est suffisamment faible (en dimension 2 par exemple, un minimum correspond à une *pente nulle*). [14]

Application au dispatching économique :

Appliquant la méthode du gradient au dispatching économique, la fonction d'objet sera :

$$minF = \sum_{i=1}^{N} f_i(P_{Gi}) \tag{2.16}$$

La contrainte est la fonction de puissances actives :

$$\sum_{i=1}^{N} P_{Gi} = P_D \tag{2.17}$$

Premièrement, on doit construire la fonction de Lagrange

$$L = F + \lambda\left(P_D - \sum_{i=1}^{N} P_{Gi}\right) = \sum_{i=1}^{N} f_i(P_{Gi}) + \lambda\left(P_D - \sum_{i=1}^{N} P_{Gi}\right) \tag{2.18}$$

Le gradient de la fonction de Lagrange est :

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial P_{G1}} \\ \frac{\partial L}{\partial P_{G2}} \\ \vdots \\ \frac{\partial L}{\partial P_{GN}} \\ \frac{\partial L}{\partial \lambda} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1(P_{G1})}{\partial P_{G1}} - \lambda \\ \frac{\partial f_2(P_{G2})}{\partial P_{G2}} - \lambda \\ \vdots \\ \frac{\partial f_N(P_{GN})}{\partial P_{GN}} - \lambda \\ P_D - \sum_{i=1}^{N} P_{Gi} \end{bmatrix} \tag{2.19}$$

Pour utiliser le gradient ∇L pour la solution du dispatching économique, les valeurs initiales $P_{G1}^0, P_{G2}^0, \dots, P_{GN}^0$, et λ^0 doivent être données. Les nouvelles valeurs seront calculées avec les équations suivantes :

$$x^1 = x^0 - \varepsilon \nabla L \tag{2.20}$$

où les vecteurs $x^1,\ x^0$ sont :

$$x^0 = \begin{bmatrix} P_{G1}^0 \\ P_{G2}^0 \\ \vdots \\ P_{GN}^0 \\ \lambda^0 \end{bmatrix} \tag{2.21}$$

$$x^1 = \begin{bmatrix} P_{G1}^1 \\ P_{G2}^1 \\ \vdots \\ P_{GN}^1 \\ \lambda^1 \end{bmatrix} \tag{2.22}$$

L'expression générale de la recherche du gradient est :

$$x^n = x^{n-1} - \varepsilon \nabla L \tag{2.23}$$

où *n* est le nombre d'itérations. [15]

2.4.3. La méthode d'itération de Lambda (Lambda iteration method)

La méthode d'itération de Lambda est une des méthodes utilisées pour trouver la valeur de Lambda du système et trouver le dispatching économique optimal des générateurs.

Contrairement aux autres méthodes d'itération, comme : Gausse-Seidel et Newton – Raphson, Lambda itération n'utilise pas la valeur précédente de l'inconnue pour trouver la valeur suivante c'est-à-dire il n'y a pas une équation qui calcule la valeur suivante en fonction de la valeur précédente. La valeur suivante est prédéfinie par *intuition*, elle est projetée avec interpolation de la bonne valeur possible jusqu'à ce que le décalage spécifié soit obtenu. [12]

On va maintenant discuter comment trouver le dispatching économique optimal utilisant la méthode d'itération de Lambda.

- la méthode exige qu'il y ait une correspondance entre une valeur lambda et l'output (en MW) de chaque générateur
- la méthode commence avec des valeurs de lambda en-dessous et en-dessus de la valeur optimale (qui est inconnue), puis par itération limite la valeur optimale

On choisit λ^L et λ^Htel que :

$$\sum_{i=1}^{m} P_{Gi}(\lambda^L) - P_D < 0 \qquad \sum_{i=1}^{m} P_{Gi}(\lambda^H) - P_D > 0 \tag{2.24}$$

on pose

$$\lambda^M = \frac{(\lambda^H + \lambda^L)}{2} \tag{2.25}$$

si

$$\sum_{i=1}^{m} P_{Gi}(\lambda^M) - P_D > 0 \tag{2.26}$$

On pose

$$\lambda^H = \lambda^M \tag{2.27}$$

si

$$\sum_{i=1}^{m} P_{Gi}(\lambda^M) - P_D < 0 \tag{2.28}$$

On pose

$$\lambda^L = \lambda^M \tag{2.29}$$

On refait le calcul jusqu'à

$$|\lambda^L - \lambda^H| > \varepsilon \tag{2.30}$$

D'où ε est la tolérance de convergence

Dans la figure ci-dessous pour chaque valeur de lambda il y a une P_{Gi} unique pour chaque générateur. Cette relation est la fonction $P_{Gi}(\lambda)$. [13]

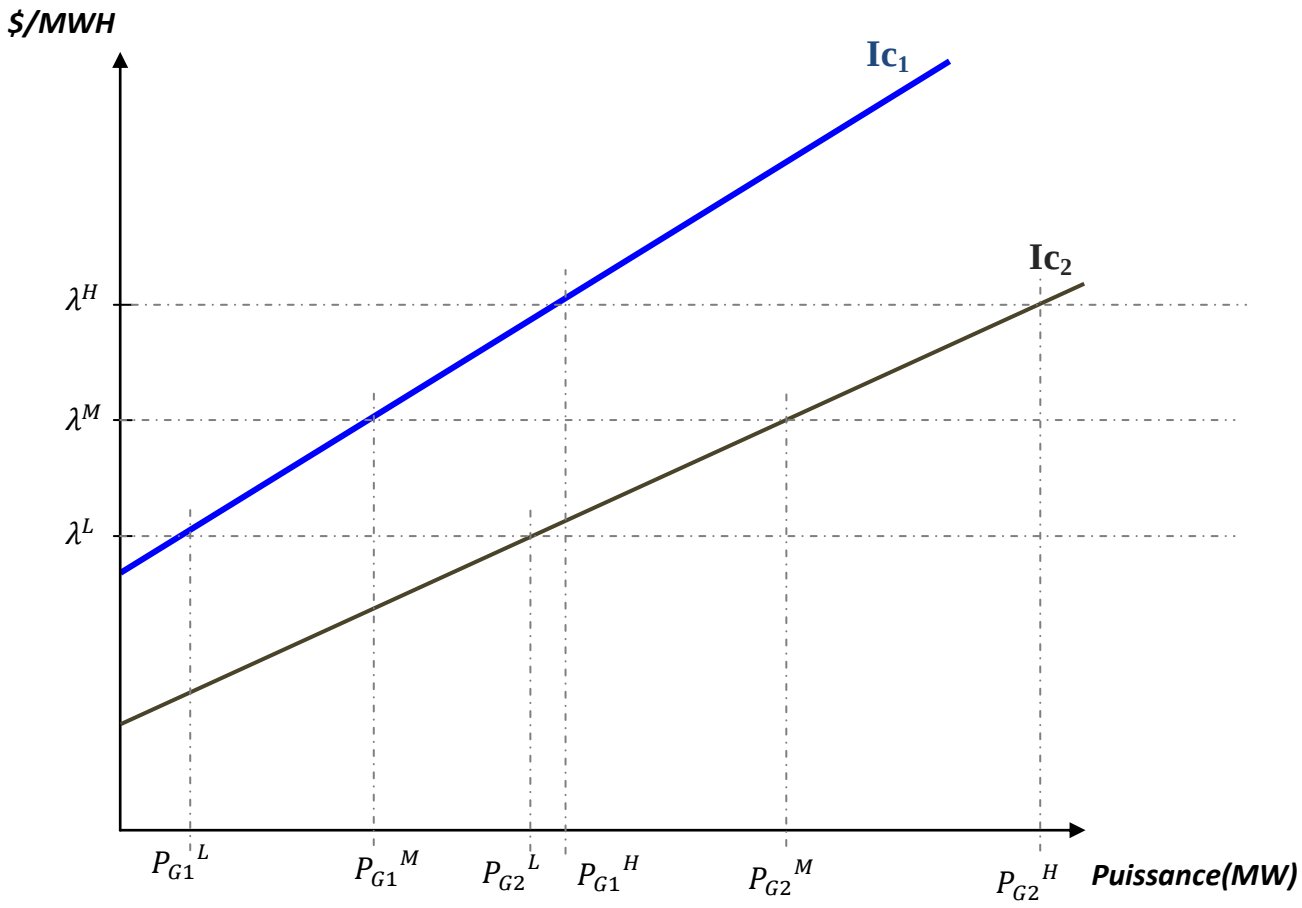

FIG.2.4 – *changement de λ en fonction de la puissance de sortie*. [13]

2.5. Dispatching économique avec pertes

Deux approches sont essentiellement utilisées pour la solution de dispatching économique avec pertes, la première est le développement d'une expression mathématique des pertes en fonction des puissances de sortie de chaque unité de production. La deuxième approche consiste à utiliser les équations de l'écoulement de puissances optimal (optimal power flow).

2.5.1. Première approche :(Utilisation d'une expression mathématique des pertes)

La fonction à optimiser reste la même de la *fonction 2.9*, cependant, les pertes de transmission doivent être ajoutées aux contraintes d'égalité tel que :

$$P_D + \boldsymbol{P_L} - \sum_{i=1}^{N} P_{Gi} = 0 \qquad (2.31)$$

P_L : Les pertes de la ligne de transmission.

N : le nombre des unités de production.

La fonction de Lagrange pour ce nouveau cas est donnée par :

$$L(P_i, \lambda) = C + \lambda(P_D + \boldsymbol{P_L} - \sum_{i=1}^{N} P_{Gi}) \tag{2.32}$$

Les dérivées de la fonction de Lagrange par rapport aux variables indépendantes nous donne :

$$\frac{\delta L}{\delta P_{Gi}} = \frac{dC_i(P_{Gi})}{dP_{Gi}} - \lambda\left(1 - \frac{\delta \boldsymbol{P_L}}{\delta P_{Gi}}\right) = 0$$

$$\frac{dC_i(P_{Gi})}{dP_{Gi}} - \lambda\frac{\delta \boldsymbol{P_L}}{\delta P_{Gi}} = \lambda$$

$$\lambda = \frac{dC_i(P_{Gi})}{dP_{Gi}}\left(\frac{1}{1 - \frac{\delta \boldsymbol{P_L}}{\delta P_{Gi}}}\right) \tag{2.33}$$

$$\frac{\delta L}{\delta \lambda} = P_D + \boldsymbol{P_L} - \sum_{i=1}^{N} P_{Gi} = 0 \tag{2.34}$$

Les équations (2.33) et (2.34) sont des conditions nécessaires pour solutionner le problème de dispatching avec pertes.

Prenant un exemple d'une fonction des pertes totales d'une ligne de transmission avec deux centrales :

$$P_L = 1.5 \times 10^{-4}P_1^2 + 2 \times 10^{-5}P_1P_2 + 3 \times 10^{-5}P_2^2 \ MW \text{ [33]}$$

Remarque : dans la pratique la fonction de pertes n'est pas donnée directement donc on doit l'extraire à l'aide des équations de l'écoulement de puissances.

2.5.2. Deuxième approche : (utilisation de l'OPF)

Le dispatching économique avec pertes est un procédé itératif qui doit s'il est réalisé correctement converger vers la solution optimale. Pour tenir compte des pertes, nous allons évaluer celles-ci et les inclure dans la demande. Elles varient en fonction de la répartition des puissances entre les centrales et de la consommation locale de puissance. Ainsi, contrairement à celui sans pertes, le dispatching économique avec pertes tient compte de la topographie du réseau. Pour pénaliser les centrales qui produisent de la puissance dont le transit provoque d'importantes pertes, nous multiplions leur coût incrémental par un *facteur de pénalité*. La

justification physique de ce facteur de pénalité s'explique par le fait qu'à cause des pertes, il peut être plus intéressant de produire pour plus cher près du lieu de consommation que loin et pour moins cher.

Pour l'appliquer au dispatching économique avec perte, il nous faut :

- Calculer les pertes
- Calculer le facteur de pénalité
- Déterminer un critère de convergence

a. Schéma bloc

Voici le schéma bloc de l'algorithme appliqué :

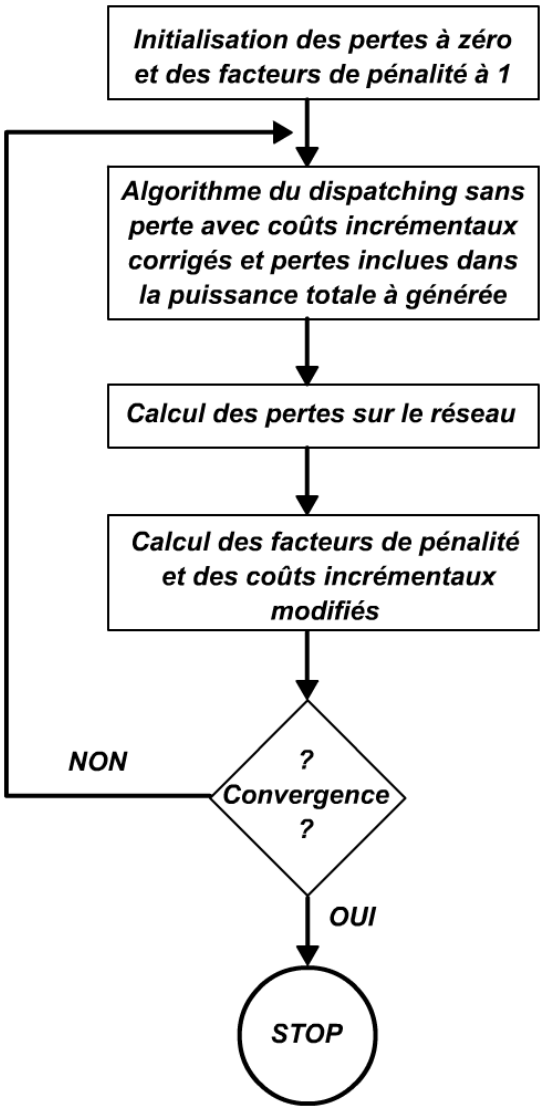

FIG.2.5- *Schéma bloc de l'algorithme de dispatching économique* [3]

b. Calcul des pertes(P_L)

À partir des équations de l'écoulement de puissance (Power Flow) entre deux nœuds i et j[34]

T_{ij} représente la puissance qui quitte le nœud *i* en direction du nœud *j*.

T_{ji} représente la puissance qui quitte le nœud *j* en direction du nœud *i*.

$$T_{ij} = V_i^2 G_{ij} - V_i V_j (G_{ij}\cos(\delta_i - \delta_j)) + B_{ij}\sin(\delta_i - \delta_j)) \quad (2.35)$$

$$T_{ji} = V_j^2 G_{ij} - V_i V_j (G_{ij}\cos(\delta_j - \delta i)) + B_{ij}\sin(\delta_j - \delta_i)) \quad (2.36)$$

Leur somme est donc égale aux pertes sur la ligne qui lie les deux nœuds.

$$\Rightarrow \boldsymbol{P_{Lij}} = \boldsymbol{T_{ij}} + \boldsymbol{T_{ji}} \quad (2.37)$$

$$\begin{aligned} &= G_{ij}(V_i^2 - 2V_i V_j cos(\delta_i - \delta_j) + V_j^2) \\ &\approx G_{ij}((V_i - V_j)^2 + V_i V_j(\delta_i - \delta_j)^2) \end{aligned} \quad (2.38)$$

Avec $(\delta_i - \delta_j) \approx 0$ (utilisation de Taylor sur le cosinus). Si de plus les nœuds sont à leur tension nominale $V_i = V_j = 1 p.u.$

$$\Rightarrow P_{Lij} \approx G_{ij}(\delta_i - \delta_j)^2 \quad (2.39)$$

$$P_L = \sum_{toutes\_les\_lignes} P_{Lij} \quad (2.40)$$

Ce qui peut s'écrire sous forme matricielle

$$P_L = \Psi^T G \Psi \quad (2.41)$$

Avec $\Psi = M\delta$. M étant *la matrice d'incidence des lignes*, *ẟ la matrice des phases des nœuds* et G *la matrice diagonale des conductances des lignes* :

$$G = \begin{bmatrix} G_{12} & 0 & 0 & \dots & 0 \\ 0 & G_{13} & 0 & \dots & 0 \\ 0 & 0 & \ddots & \dots & 0 \\ \dots & \dots & \dots & \ddots & \dots \\ 0 & 0 & 0 & \dots & G_{(n-1)n} \end{bmatrix} \quad (2.42)$$

Or ; δ peut-être approximé par un *DC Load Flow*[2]. Ainsi,

$$P_G - P_D = A\delta$$

et donc,

$$\delta = A^{-1}(P_G - P_D) \tag{2.43}$$

Avec A : matrice du DC Load flow.

Nous trouvons donc :

$$\begin{aligned} P_L &= \Psi^T G \Psi \\ &= (P_G - P_D)^T A^{-1} M^T G M A^{-1} (P_G - P_D) \\ &= P_D^T B P_D - 2P_D^T B P_G + P_G^T B P_G, \end{aligned} \tag{2.44}$$

où

$$B = A^{-1} M^T G M A^{-1} \tag{2.45}$$

c. Facteur de pénalité

Soit, P_{Gi} la puissance générée par la centrale *i*

C le coût de production

P_{Ci} la partie de la puissance générée par la centrale i qui est réellement consommée par les charges

P_{Li} la partie de la puissance générée qui est perdue dans les lignes

Nous savons que

$$\frac{dC}{dP_{Gi}} = b_i + 2.c_i.P_{Gi}. \tag{2.46}$$

et que

$$P_G = P_{Ci} + P_{Li} \tag{2.47}$$

Donc,

$$\begin{aligned} dC &= b_i.dP_{Gi} + 2.c_i.P_{Gi}.dP_{Gi} \\ dC &= b_i.(dP_{Ci} + dP_{Li}) + 2.c_i.P_{Gi}.(dP_{Ci} + dP_{Li}) \end{aligned}$$

[2] *DC load flow est la résolution du problème non linéaire de 'AC load flow' (répartition des puissances en courant alternatif) en introduisant des approximations rendant le problème linéaire.*

$$\Rightarrow$$

$$\frac{dC}{dP_{Ci}} = (b_i + 2.c_i.P_{Gi}).(1 + \frac{dP_{Li}}{dP_{Ci}})$$

$$\frac{dC}{dP_{Ci}} = (b_i + 2.c_i.P_{Gi}).(\frac{dP_{Ci} + dP_{Li}}{dP_{Ci}})$$

$$\frac{dC}{dP_{Ci}} = (b_i + 2.c_i.P_{Gi}).(\frac{dP_{Gi}}{dP_{Ci}})$$

$$\frac{dC}{dP_{Ci}} = (b_i + 2.c_i.P_{Gi}).(\frac{dP_{Ci}}{dP_{Gi}})^{-1}$$

$$\frac{dC}{dP_{Ci}} = (b_i + 2.c_i.P_{Gi}).(\frac{dP_{Gi} - dP_{Li}}{dP_{Gi}})^{-1}$$

$$\frac{dC}{dP_{Ci}} = (b_i + 2.c_i.P_{Gi}).(1 - \frac{dP_{Li}}{dP_{Gi}})^{-1} \qquad (2.48)$$

puisque

$$\frac{dP_{Li}}{dP_{Gi}} = \frac{dP_L}{dP_{Gi}}$$

$$\frac{dC}{dP_{Ci}} = (b_i + 2.c_i.P_{Gi}).(1 - \frac{dP_L}{dP_{Gi}})^{-1} \qquad (2.49)$$

On obtient

$$\frac{dC}{dP_{Ci}} = (b'_i + 2.c'_i.P_{Gi}) \qquad (2.50)$$

avec

$$b'_i = b_i.f_i \quad \text{et} \quad c'_i = c_i.f_i$$

Ainsi $f_i = (1 - \frac{dP_L}{dP_{Gi}})^{-1}$ est le facteur de pénalité du coût incrémental.

$\frac{dC}{dP_{Ci}}$ représente l'accroissement du coût de production pour une augmentation de la puissance consommée par les charges. Cette valeur est plus intéressante pour nous que le coût incrémental défini précédemment car il tient compte des pertes. Ainsi, le critère pour la distribution des paquets de puissance sera dorénavant de trouver le coût incrémental corrigé le plus faible.

d. Critère de convergence

Si $\left| \sum P_{Gi} - P_D - P_L \right| \leq \varepsilon$, le système a convergé.

e. Hypothèses et approximations

Plusieurs hypothèse et approximations sont utilisées lors du calcul d'erreur. On considère les déphasages entre nœud négligeable ($(\delta_i - \delta_j) \approx 0$) ce qui nous a permis d'approcher un cosinus par son développement en série de Taylor. Tous les nœuds sont supposés à tension nominative. L'usage du DC load flow implique aussi que l'on suppose les conductances négligeables. $G_{ij} \approx 0$ [3]

2.6. Dispatching économique de la puissance réactive

Plusieurs techniques sont utilisées pour ajouter l'écoulement de puissance réactive dans la formule de dispatching économique pour minimiser le coût total de l'énergie. En pratique, les injections d'énergie réactive par *les FACTS* et *les compensateurs synchrones*[3] peuvent être utilisés pour régler la circulation de l'énergie réactive dans le réseau, cette énergie réactive est utilisée pour contrôler la magnitude du voltage dans les jeux de barres du réseau, donc il n'y a pas besoin d'un dispatching central de l'énergie réactive. [33]

2.7. Conclusion

Nous avons présenté dans ce chapitre des notions sur le dispatching économique, la fonction du cout et comment faire extraire cette fonction à l'aide des méthodes de régression. Nous avons traité aussi les méthodes d'optimisation (minimisation) qui utilisent le gradient ou les itérations pour solutionner le problème de dispatching économique. On a calculé les pertes linéiques en se basant sur les équations de l'écoulement de puissance dans les réseaux interconnectés.

Dans le chapitre suivant, on va étudier l'une des techniques d'intelligence artificielle ; celle des réseaux de neurones à rétropropagation de l'erreur, qui va être utilisée dans le chapitre 04 comme méthode novatrice pour le calcul du dispatching économique.

---

[3] - *FACTS (flexible alternating current transmission system) ou, Système de transmission flexible en courant alternatif, est un équipement d'électronique de puissance d'appoint comme : les bancs de capacitances commutées (switched capacitor banks) et les compensateurs statiques d'énergie réactive (static var systems) avec des transformateurs à plots réglables (transformer tap settings), utilisé pour contrôler la tension, augmenter les capacités de transit, ou assurer la stabilité dynamique des réseaux de transmission d'électricité. Il agit généralement en fournissant ou en consommant dynamiquement de la puissance réactive sur le réseau. Ceci a pour effet d'augmenter ou de diminuer l'amplitude de la tension à son point de connexion, et par conséquent la puissance active maximale transmissible. Les FACTS sont utilisés aussi pour le filtrage des courants harmoniques.*

- Le compensateur synchrone est une machine synchrone tournant à vide dont la seule fonction est de consommer ou de fournir de la puissance réactive au réseau. C'est en ajustant le courant d'excitation qu'il est possible de fournir de l'énergie réactive (la machine est surexcitée) ou de consommer de l'énergie (si la machine est sousexcitée). De telles machines sont utilisées notamment pour fournir de l'énergie réactive lorsque le réseau est chargé, et pour absorber l'énergie réactive générée par les lignes lorsque la consommation est faible.

CHAPITRE 03

LES RESEAUX DE NEURONES

3.1. Introduction :

Comment l'homme fait-il pour raisonner, parler, calculer, apprendre, ...? Comment s'y prendre pour créer de l'intelligence artificielle ? Deux types d'approches ont été essentiellement explorés :

- procéder d'abord à l'analyse logique des tâches relevant de la cognition humaine et tenter de les reconstituer par programme. C'est cette approche qui a été privilégiée par l'Intelligence Artificielle et la psychologie cognitive classiques. Cette démarche est étiquetée sous le nom de *cognitivisme*.
- puisque la pensée est produite par le cerveau ou en est une propriété, commencer par étudier comment celui-ci fonctionne. C'est cette approche qui a conduit à l'étude de réseaux de neurones formels. On désigne par *connexionnisme* la démarche consistant à vouloir rendre compte de la cognition humaine par des réseaux de neurones.

La seconde approche a donc menée à la définition et l'étude de réseaux de neurones formels qui sont des réseaux complexes d'unités de calcul élémentaire interconnectées. Il existe deux courants de recherche sur les réseaux de neurones : un premier motivé par l'étude et la modélisation des phénomènes naturels d'apprentissage à l'aide de réseaux de neurones, la pertinence biologique est importante ; un second motivé par l'obtention d'algorithmes efficaces ne se préoccupant pas de la pertinence biologique. Nous nous plaçons du point de vue du second groupe. En effet, bien que les réseaux de neurones formels aient été définis à partir de considérations biologiques, pour la plupart d'entre eux, et en particulier ceux étudiés dans ce cours, de nombreuses caractéristiques biologiques (le temps, la mémoire, ...) ne sont pas prises en compte.

3.1.1. Quelques éléments de physiologie du cerveau

La physiologie du cerveau montre que celui-ci est constitué de cellules (les *neurones*) interconnectées. Quelques étapes de cette découverte :

- Van Leuwenhook (1718) : première description fidèle de ce qu'on appellera plus tard les *axones*,
- Dutrochet (1824) : observation du corps cellulaire des neurones
- Valentin : découverte des *dendrites*,
- Deiters (1865) : image actuelle de la cellule nerveuse

- Sherington (1897) : les *synapses*,
- les neurotransmetteurs (première moitié du siècle).

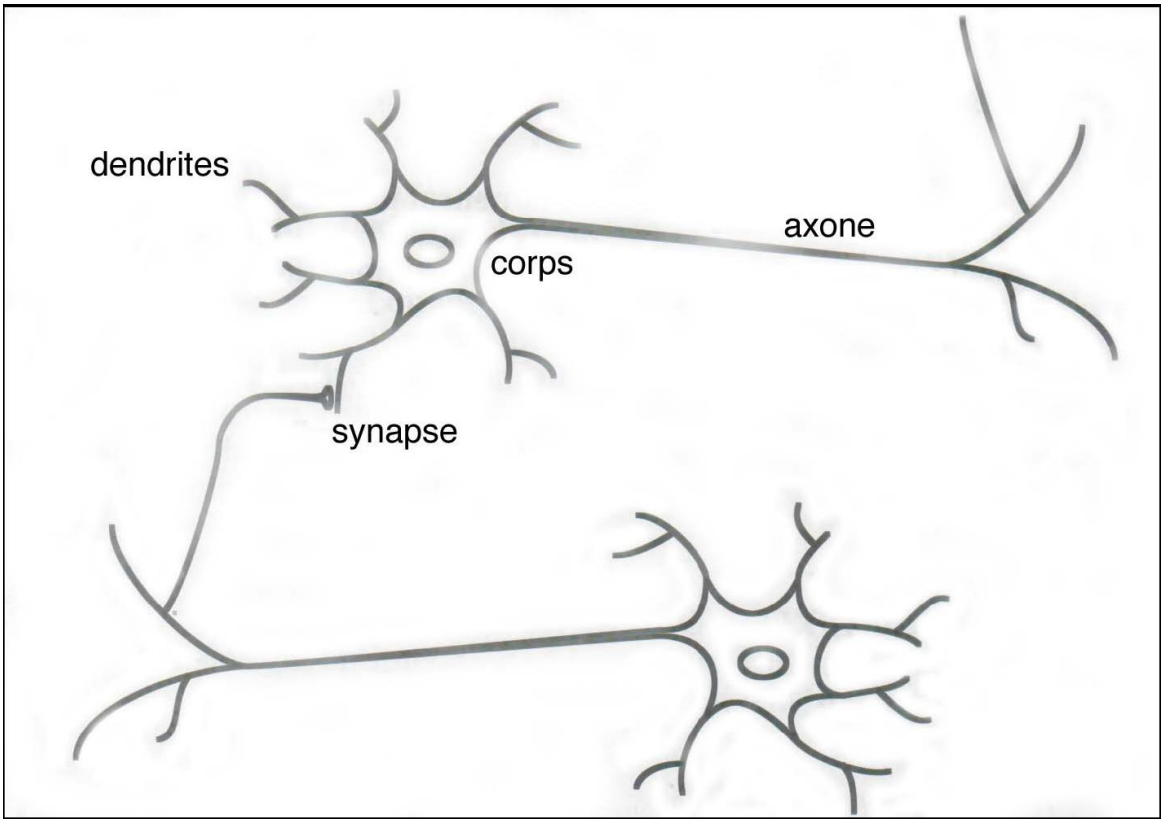

FIG.3.1 – *Schéma d'un neurone biologique*. [16]

Les *neurones* reçoivent les signaux (impulsions électriques) par des extensions très ramifiées de leur corps cellulaire (*les dendrites*) et envoient l'information par de longs prolongements (*les axones*). Les impulsions électriques sont régénérées pendant le parcours le long de l'axone. La durée de chaque impulsion est de l'ordre d'1 ms et son amplitude d'environ 100 *mVolts*.

Les contacts entre deux neurones, de l'axone à une dendrite, se font par l'intermédiaire des *synapses*. Lorsqu'un potentiel d'action atteint la terminaison d'un axone, des neuromédiateurs sont libérés et se lient à des récepteurs post-synaptiques présents sur les dendrites. L'effet peut être excitateur ou inhibiteur.

Chaque neurone intègre en permanence jusqu'à un millier de signaux synaptiques. Ces signaux n'opèrent pas de manière linéaire (*effet de seuil*).

Quelques informations en vrac :

- le cerveau contient environ *100 milliards* de neurones.
- on ne dénombre que quelques dizaines de catégories distinctes de neurones.
- aucune catégorie de neurones n'est propre à l'homme.
- la vitesse de propagation des influx nerveux est de l'ordre de 100m/s. C'est à dire bien inférieure à la vitesse de transmission de l'information dans un circuit électronique.
- on compte de quelques centaines à plusieurs dizaines de milliers de contacts synaptiques par neurone. Le nombre total de connexions est estimé à environ 10^{15}.
- la connectique du cerveau ne peut pas être codée dans un << document biologique >> tel l'ADN pour de simples raisons combinatoires. La structure du cerveau provient donc en partie des contacts avec l'environnement. L'apprentissage est donc indispensable à son développement.
- le nombre de neurones décroit après la naissance. Cependant, cette affirmation semble remise en question.
- on observe par contre une grande plasticité de l'axone, des dendrites et des contacts synaptiques. Celle-ci est surtout très importante après la naissance (on a observé chez le chat un accroissement des contacts synaptiques de quelques centaines à 12000 entre le 10ème et le 35ème jour). Cette plasticité est conservée tout au long de l'existence.
- les synapses entre des neurones qui ne sont pas simultanément actifs sont affaiblies puis éliminées.
- il semble que l'apprentissage se fasse par un double mécanisme : des connections sont établies de manière redondantes et aléatoires puis seules les connexions entre des neurones simultanément actifs sont conservés (phase de sélection) tandis que les autres sont éliminés. On parle de stabilisation sélective. [17]

3.1.2. Historique

Les neurologues *Warren McCulloch* et *Walter Pitts* menèrent les premiers travaux sur les réseaux de neurones à la suite de leur article fondateur : '*What the frog's eye tells to the frog's brain*'. Ils constituèrent un modèle simplifié de neurone biologique communément appelé neurone formel. Ils montrèrent également théoriquement que des réseaux de neurones formels simples peuvent réaliser des fonctions logiques, arithmétiques et symboliques complexes.

La fonction des réseaux de neurones formels à l'instar du modèle vivant est de résoudre des problèmes. À l'opposé des méthodes traditionnelles de résolution informatique, on ne doit pas construire un programme pas à pas en fonction de la compréhension de celui-ci. Les

paramètres les plus importants de ce modèle sont les coefficients synaptiques. Ce sont eux qui construisent le modèle de résolution en fonction des informations données au réseau. Il faut donc trouver un mécanisme qui permette de les calculer à partir des grandeurs que l'on peut acquérir du problème. C'est le principe fondamental de l'apprentissage. Dans un modèle de réseaux de neurones formels, apprendre, c'est d'abord calculer les valeurs des coefficients synaptiques en fonction des exemples disponibles.

Les travaux de *McCulloch* et *Pitts* n'ont pas donné d'indication sur une méthode pour adapter les coefficients synaptiques. Cette question au cœur des réflexions sur l'apprentissage a connu un début de réponse grâce aux travaux du physiologiste canadien *Donald Hebb* sur l'apprentissage en 1949 décrits dans son ouvrage '*The Organization of Behaviour'*. *Hebb* a proposé une règle simple qui permet de modifier la valeur des coefficients synaptiques en fonction de l'activité des unités qu'ils relient. Cette règle aujourd'hui connue sous le nom de « règle de Hebb » est presque partout présente dans les modèles actuels, même les plus sophistiqués. À partir de cet article, l'idée se sema au fil du temps dans les esprits, et elle germa dans l'esprit de *Franck Rosenblatt* en 1958 avec le modèle du perceptron. C'est le premier système artificiel capable d'apprendre par expérience, y compris lorsque son instructeur commet quelques erreurs (en quoi il diffère nettement d'un système d'apprentissage logique formel). [18]

En 1969, un coup grave fut porté à la communauté scientifique gravitant autour des réseaux de neurones : *Marvin Lee Minsky* et *Seymour Papert* publièrent un ouvrage mettant en exergue quelques limitations théoriques du Perceptron, notamment l'impossibilité de traiter des problèmes non linéaires ou de connexité. Ils étendirent implicitement ces limitations à tous modèles de réseaux de neurones artificiels. Paraissant alors dans une impasse, la recherche sur les réseaux de neurones perdit une grande partie de ses financements publics, et le secteur industriel s'en détourna aussi. Les fonds destinés à l'intelligence artificielle furent redirigés plutôt vers la logique formelle et la recherche piétina pendant dix ans. Cependant, les solides qualités de certains réseaux de neurones en matière adaptative leur permettant de modéliser de façon évolutive des phénomènes eux-mêmes évolutifs les amèneront à être intégrés sous des formes plus ou moins explicites dans le corpus des systèmes adaptatifs, utilisés dans le domaine des télécommunications ou celui du contrôle de processus industriels.

En 1982, *John Joseph Hopfield*, physicien reconnu, donna un nouveau souffle au neuronal en publiant un article introduisant un nouveau modèle de réseau de neurones (complètement récurrent). Cet article eut du succès pour plusieurs raisons, dont la principale était de teinter la

théorie des réseaux de neurones de la rigueur propre aux physiciens. Le neuronal redevint un sujet d'étude acceptable, bien que le modèle de Hopfield souffrait des principales limitations des modèles des années 1960, notamment l'impossibilité de traiter les problèmes non-linéaires.

À la même date, les approches algorithmiques de l'intelligence artificielle furent l'objet de désillusion, leurs applications ne répondant pas aux attentes. Cette désillusion motiva une réorientation des recherches en intelligence artificielle vers les réseaux de neurones (bien que ces réseaux concernent la perception artificielle plus que l'intelligence artificielle à proprement parler). La recherche fut relancée et l'industrie reprit quelque intérêt au neuronal (en particulier pour des applications comme le guidage de missiles de croisière). En 1984, *le système de rétropropagation du gradient de l'erreur* était le sujet le plus débattu dans le domaine.

Une révolution survient alors dans le domaine des réseaux de neurones artificiels : une nouvelle génération de réseaux de neurones, capables de traiter avec succès des phénomènes non-linéaires : *le perceptron multicouche* ne possède pas les défauts mis en évidence par *Marvin Minsky*. Proposé pour la première fois par *Werbos*, le Perceptron Multicouche apparait en 1986 introduit par *Rumelhart*, et, simultanément, sous une appellation voisine, chez *Yann le Cun*. Ces systèmes reposent sur la rétropropagation du gradient de l'erreur dans des systèmes à plusieurs couches, chacune de type *Adaline* de *Bernard Widrow*, proche du Perceptron de *Rumelhart*. [19]

3.2. Modèle d'un neurone

Le modèle mathématique d'un neurone artificiel est illustré à la *figure 3.2*. Un neurone est essentiellement constitué d'un intégrateur qui effectue la somme pondérée de ses entrées. Le résultat n de cette somme est ensuite transformé par une fonction de transfert *f* qui produit la sortie *a* du neurone. Les *R* entrées du neurone correspondent au vecteur $p = [p_1 p_2 \dots p_R]^T$ alors que $w = [w_1 w_2 \dots w_{1,R}]^T$ représente le vecteur des poids du neurone. La sortie *n* de l'intégrateur est donnée par l'équation suivante :

$$n = \sum_{j=1}^{R} w_{1,j} p_j - b \tag{3.1}$$

$$= w_{1,1} p_1 + w_{1,2} p_2 + \cdots + w_{1,R} p_R - b \tag{3.2}$$

que l'on peut aussi écrire sous forme matricielle :

$$n = w^T p - b \tag{3.3}$$

T : marque le transposée de la matrice ou du vecteur.

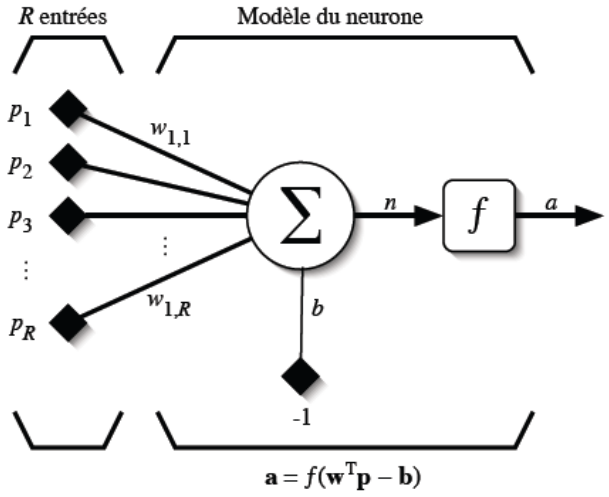

FIG.3.2 – *Modèle d'un neurone artificiel*

Cette sortie correspond à une somme pondérée des poids et des entrées moins ce qu'on nomme *le biais b* du neurone. Le résultat *n* de la somme pondérée s'appelle *le niveau d'activation du neurone*.

Le biais *b* s'appelle aussi *le seuil d'activation* du neurone. Lorsque le niveau d'activation atteint ou dépasse le seuil *b*, alors l'argument de *f* devient positif (ou nul). Sinon, il est négatif. On peut faire un parallèle entre ce modèle mathématique et certaines informations que l'on connait à propos du neurone biologique. Ce dernier possède trois principales composantes : les dendrites, le corps cellulaire et l'axone (voir figure 3.1). Les dendrites forment un maillage de récepteurs nerveux qui permettent d'acheminer vers le corps du neurone des signaux électriques en provenance d'autres neurones. Celui-ci agit comme une espèce d'intégrateur en accumulant des charges électriques. Lorsque le neurone devient suffisamment excité (lorsque la charge accumulée dépasse un certain seuil), par un processus électrochimique, il engendre un potentiel électrique qui se propage à travers son axone pour éventuellement venir exciter d'autres neurones. Il semble que c'est l'arrangement spatial des neurones et de leur axone, ainsi que la qualité des connexions synaptiques individuelles qui détermine la fonction précise d'un réseau de neurones biologique. C'est en se basant sur ces connaissances que le modèle mathématique a été défini.

Un poids d'un neurone artificiel représente donc l'efficacité d'une connexion synaptique. Un poids négatif vient inhiber une entrée, alors qu'un poids positif vient l'accentuer. Il importe de retenir que ceci est une grossière approximation d'une véritable synapse qui résulte en fait d'un processus chimique très complexe et dépendant de nombreux facteurs extérieurs encore mal connus. Il faut bien comprendre que notre neurone artificiel est un modèle pragmatique qui, comme nous le verrons plus loin, nous permettra d'accomplir des tâches intéressantes. La vraisemblance biologique de ce modèle ne nous importe peu. Ce qui compte est le résultat que ce modèle nous permettra d'atteindre.

Un autre facteur limitatif dans le modèle que nous nous sommes donnés concerne son caractère *discret*. En effet, pour pouvoir simuler un réseau de neurones, nous allons rendre le temps discret dans nos équations. Autrement dit, nous allons supposer que tous les neurones sont *synchrones*, c'est-à-dire qu'à chaque temps *t*, ils vont simultanément calculer leur somme pondérée et produire une sortie $a(t) = f\big(n(t)\big)$. Dans les réseaux biologiques, tous les neurones sont en fait *asynchrones*.

Revenons donc à notre modèle tel que formulé par *l'équation 3.1* et ajoutons la fonction d'activation f pour obtenir la sortie du neurone :

$$a = f(n) = f(w^T p - b) \tag{3.4}$$

En remplaçant w^T par une matrice $W = w^T$ d'une seule ligne, on obtient la forme générale :

$$a = f(Wp - b) \tag{3.5}$$

L'équation 3.5 nous amène à introduire un schéma de notre modèle plus compact que celui de *la figure 3.2*. *La figure 3.3* illustre celui-ci. On y représente les R entrées comme un rectangle noir (le nombre d'entrées est indiqué sous le rectangle). De ce rectangle sort le vecteur p dont la dimension matricielle est $R \times 1$. Ce vecteur est multiplié par une matrice W qui contient les poids (synaptiques) du neurone. Dans le cas d'un neurone simple, cette matrice possède la dimension $1 \times R$. Le résultat de la multiplication correspond au niveau d'activation qui est ensuite comparé au seuil b (un scalaire) par soustraction. Finalement, la sortie du neurone est calculée par la fonction d'activation f. La sortie d'un neurone est toujours *un scalaire*.

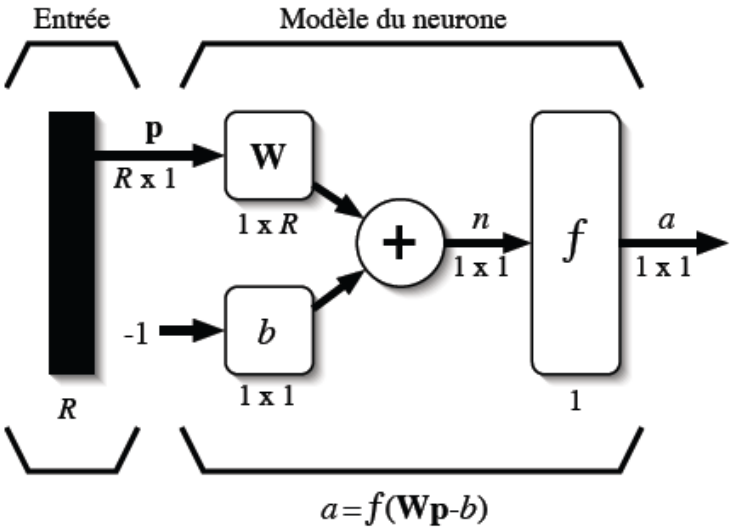

FIG.3.3 – *Représentation matricielle du modèle d'un neurone artificiel.*

3.3. Fonctions de transfert

Différentes fonctions de transfert pouvant être utilisées comme fonction d'activation du neurone sont énumérées au tableau 3.1. Les trois les plus utilisées sont les fonctions «*seuil*» « *hard limit* », «*linéaire*» et «*sigmoïde*».

Nom de la fonction	Relation d'entrée / sortie	Icône	Nom Matlab
Seuil	$a = 0 \quad si \quad n < 0$ $a = 1 \quad si \quad n \geq 0$		*hardlim*
Seuil symétrique (seuils)	$a = -1 \quad si \quad n < 0$ $a = 1 \quad si \quad n \geq 0$		*hardlims*
Linéaire	$a = n$		*purelin*
Linéaire saturée	$a = 0 \quad si \quad n < 0$ $a = n \quad si \quad 0 \leq n \leq 1$ $a = 1 \quad si \quad n > 1$		*satlin*
Linéaire saturée symétrique	$a = -1 \quad si \quad n \leq -1$ $a = n \quad si \quad -1 \leq n \leq 1$ $a = 1 \quad si \quad n > 1$		*satlins*
Linéaire positive	$a = 0 \quad si \quad n < 0$ $a = n \quad si \quad n \geq 0$		*poslin*

Sigmoïde	$a = \frac{1}{1 + exp^{-n}}$		*logsig*
Tangente hyperbolique	$a = \frac{e^n - e^{-n}}{e^n + e^{-n}}$		*tansig*
Compétitive	$a = 1 \ \ si \ \ n \ \ maximum$ $a = 0 \ \ autrement$		*compet*

TAB. 3.1 – *Fonctions de transfert* $a = f(n)$.

Comme son nom l'indique, la fonction seuil applique un seuil sur son entrée. Plus précisément, une entrée négative ne passe pas le seuil, la fonction retourne alors la valeur 0 (on peut interpréter ce 0 comme signifiant *faux*), alors qu'une entrée positive ou nulle dépasse le seuil, et la fonction retourne 1 (*vrai*). Utilisée dans le contexte d'un neurone, cette fonction est illustrée à la *figure 3.4a*. On remarque alors que le biais b dans l'expression de $a = hardlim(W^T p - b)$ (*équation 3.5*) détermine l'emplacement du seuil sur l'axe $W^T p$, où la fonction passe de 0 à 1.

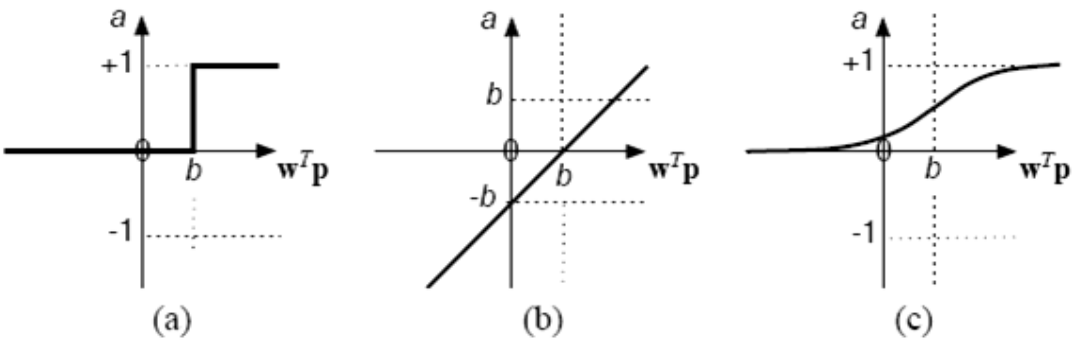

FIG. 3.4 – *Fonction de transfert : (a) du neurone «seuil» ; (b) du neurone «linéaire», et (c) du neurone «sigmoïde».*

La fonction linéaire est très simple, elle affecte directement son entrée à sa sortie : $a = n$

Appliquée dans le contexte d'un neurone, cette fonction est illustrée à *la figure 3.4b*. Dans ce cas, la sortie du neurone correspond à son niveau d'activation dont le passage à zéro se produit lorsque $W^T p = b$.

La fonction de transfert sigmoïde est quant à elle illustrée à *la figure 3.4c*. Son équation est donnée par :

$$a = \frac{1}{1 + exp^{-n}} \tag{3.6}$$

Elle ressemble soit à la fonction seuil, soit à la fonction linéaire, selon que l'on est loin ou près de *b*, respectivement. La fonction seuil est très non-linéaire car il y a une discontinuité lorsque $W^T p = b$. De son côté, la fonction linéaire est tout à fait linéaire. Elle ne comporte aucun changement de pente. La sigmoïde est un compromis intéressant entre les deux précédentes. Notons finalement, que la fonction «tangente hyperbolique» est une version symétrique de la sigmoïde.

3.4. Architecture de réseau :

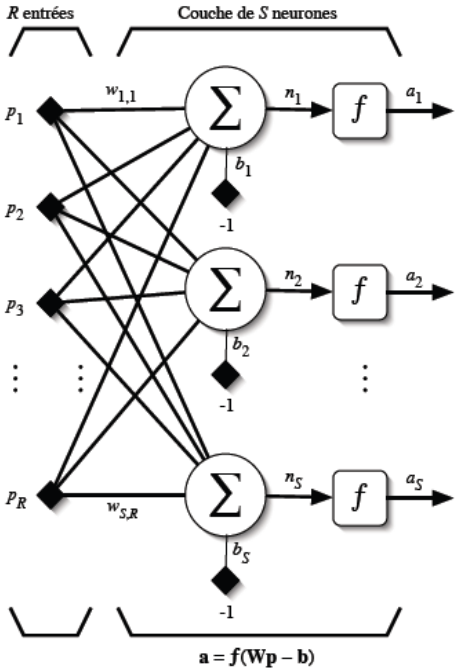

FIG. 3.5 – *Couche de S neurones.*

Un réseau de neurones est un maillage de plusieurs neurones, généralement organisé en couches. Pour construire une couche de S neurones, il s'agit simplement de les assembler comme à la *figure 3.5*. Les S neurones d'une même couche sont tous branchés aux R entrées. On dit alors que la couche est totalement connectée. Un poids $w_{i,j}$ est associé à chacune des connexions. Nous noterons toujours le premier indice par i et le deuxième par j (jamais l'inverse). Le premier indice (rangée) désigne toujours le numéro de neurone sur la couche, alors que le deuxième indice (colonne) spécifie le numéro de l'entrée. Ainsi, $w_{i,j}$ désigne le poids de la connexion qui relie le neurone i à son entrée j. L'ensemble des poids d'une couche forme donc une matrice W de dimension $S \times R$:

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & \ddots & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix} \tag{3.7}$$

Notez bien que $S \neq R$, dans le cas général (les nombres de neurones et d'entrées sont indépendants). Si l'on considère que les S neurones forment un vecteur de neurones, alors on peut créer les vecteurs $b = [b_1 b_2 \ldots b_s]^T$, $n = [n_1 n_2 \ldots n_s]^T$ et $a = [a_1 a_2 \ldots a_s]^T$. Ceci nous amène à la représentation graphique simplifiée, illustrée à la *figure 3.6*. On y retrouve, comme à la *figure 3.3*, les mêmes vecteurs et matrice. La seule différence se situe au niveau de la taille, ou plus précisément du nombre de rangées (S), de b, n, a et W.

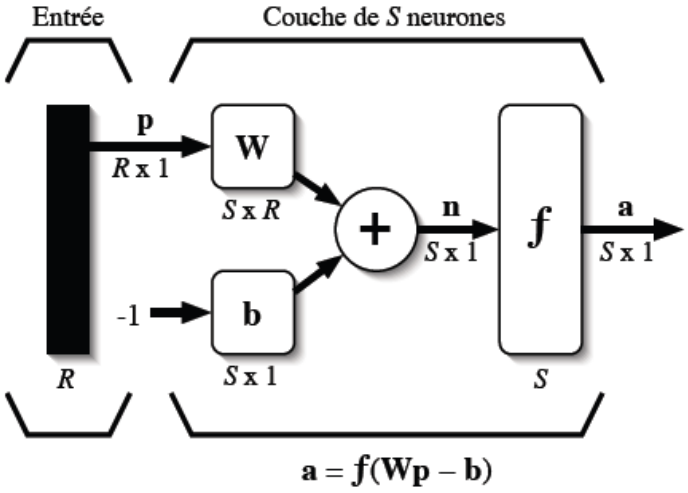

FIG. 3.6 – *Représentation matricielle d'une couche de* S *neurones.*

Finalement, pour construire un réseau, il ne suffit plus que de combiner des couches comme à la *figure 3.7*. Cet exemple comporte R entrées et trois couches de neurones comptant respectivement $S^1, S^2 \; et \; S^3$ neurones. Dans le cas général, de nouveau $S^1 \neq S^2 \neq S^3$. Chaque couche possède sa propre matrice de poids W^k, où k désigne l'indice de couche. Dans le contexte des vecteurs et des matrices relatives à une couche, nous emploierons toujours un exposant pour désigner cet indice. Ainsi, les vecteurs $b^k, n^k \; et \; a^k$ sont aussi associés à la couche *k*.

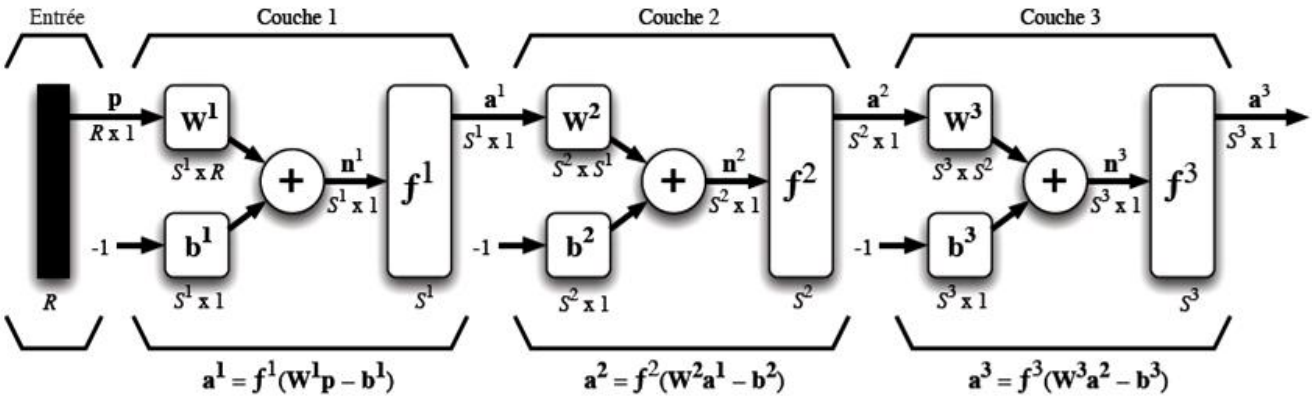

FIG. 3.7 – *Représentation matricielle d'un réseau de trois couches.*

Il importe de remarquer dans cet exemple que les couches qui suivent la première ont comme entrée la sortie de la couche précédente. Ainsi, on peut enfiler autant de couche que l'on veut, du moins en théorie. Nous pouvons aussi fixer un nombre quelconque de neurones sur chaque couche. En pratique, il n'est cependant pas souhaitable d'utiliser trop de neurones. Finalement, notez aussi que l'on peut changer de fonction de transfert d'une couche à l'autre. Ainsi, toujours dans le cas général $f^1 \neq f^2 \neq f^3$.

La dernière couche est nommée «couche de sortie». Les couches qui précèdent la couche de sortie sont nommées «couches cachées».

Les réseaux multicouches sont beaucoup plus puissants que les réseaux simples à une seule couche. En utilisant deux couches (une couche cachée et une couche de sortie), à condition d'employer une fonction d'activation *sigmoïde* sur la couche cachée, on peut entrainer un réseau à produire une approximation de la plupart des fonctions, avec une précision arbitraire (cela peut cependant requérir un grand nombre de neurones sur la couche cachée). Sauf dans de rares cas, les réseaux de neurones artificiels exploitent deux ou trois couches.

Entrainer un réseau de neurones signifie modifier la valeur de ses poids et de ses biais pour qu'il réalise la fonction entrée/sortie *désirée*. Pour spécifier la structure du réseau, il faut aussi choisir le nombre de couches et le nombre de neurones sur chaque couche. Tout d'abord, rappelons que le nombre d'entrées du réseau *(R)*, de même que le nombre de neurones sur la couche de sortie est fixé par *les spécifications du problème* que l'on veut résoudre avec ce réseau. Par exemple, si la donnée du problème comporte quatre variables en entrée et qu'elle exige de produire trois variables en sortie, alors nous aurons simplement $R = 4$ et $S^M = 3$, où *M* correspond à l'indice de la couche de sortie (ainsi qu'au nombre de couches). Ensuite, la nature du problème peut aussi nous guider dans le choix des fonctions de transfert. Par exemple, si l'on désire produire des sorties binaires *0* ou *1*, alors on choisira probablement une fonction seuil (voir *tableau 3.1*) pour la couche de sortie. Il reste ensuite à choisir le nombre de couches cachées ainsi que le nombre de neurones sur ces couches, et leur fonction de transfert. Il faudra aussi fixer les différents paramètres de l'algorithme d'apprentissage.

Finalement, la *figure 3.8* illustre le dernier élément de construction que nous emploierons pour bâtir des réseaux dit «récurrents». Il s'agit d'un registre à décalage qui permet d'introduire un retard dans une donnée que l'on veut acheminer dans un réseau. La sortie retardée $a(t)$ prend la valeur de l'entrée u au temps $t - 1$. Cet élément de retard présuppose que l'on peut initialiser la sortie au temps $t = 0$ avec la valeur $a(0)$. Cette condition initiale est indiquée à la *figure 3.8* par une flèche qui entre par le bas de l'élément.

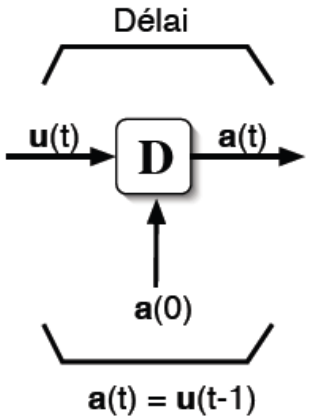

FIG. 3.8 – *élément de retard.* [20]

3.5. Apprentissage d'un réseau de neurones :

3.5.1. Différents types d'apprentissage :

Il existe essentiellement deux types d'apprentissage, l'apprentissage non supervisé et l'apprentissage supervisé.

a. Apprentissage non supervisé :

Dans ce cas, des exemples ou *prototypes* ou *patrons* sont présentés au réseau qu'on laisse s'auto-organiser au moyen de lois locales qui réagissent l'évolution des poids synaptiques. Ce mode d'apprentissage est aussi appelé « *apprentissage par compétition* ».

b. Apprentissage supervisé :

Dans ce type d'apprentissage, on cherche à imposer au réseau un fonctionnement donné en forçant à partir des entrées qui lui sont présentées, les sorties du réseau à prendre des valeurs données en modifiant les poids synaptiques. Le réseau se comporte alors comme un filtre dont les paramètres de transfère sont ajustés à partir des couples *entrée /sortie* présentés.

L'adaptation des paramètres du réseau s'effectue à partir d'un algorithme d'optimisation, l'initialisation des poids synaptiques étant le plus souvent aléatoire. [21]

3.5.2. Règles d'apprentissage non supervisé

Les principales règles de l'apprentissage non supervisé sont : La règle de Hebb, la règle de Kohonen, la règle Instar et la règle Outstar [21]

3.5.3. Apprentissage supervisé :

L'une des applications de l'apprentissage supervisé et le perceptron. La règle de correction d'erreur, donne une illustration aux règles : LMS (*Appendice B*) et la rétropropagation d'erreur (*section 3.8*), qui sont des méthodes utilisées pour l'apprentissage supervisé du perceptron monocouche et multicouches.

Règle de correction d'erreur

La règle est fondée sur la correction de l'erreur observée en sortie. Soit $a_i(t)$ la sortie que l'on obtient pour le neurone *i* au temps *t*. Cette sortie résulte d'un stimulus $p(t)$ que l'on applique aux entrées du réseau dont un des neurones correspond au neurone i. Soit $d_i(t)$ la sortie que l'on désire obtenir pour ce même neurone i au temps t. Alors, $a_i(t)$ et $d_i(t)$ seront

généralement différents et il est naturel de calculer l'erreur $e_i(t)$ entre ce qu'on obtient et ce qu'on voudrait obtenir :

$$e_i(t) = d_i(t) - a_i(t) \tag{3.8}$$

et de chercher un moyen de réduire autant que possible cette erreur. Sous forme vectorielle, on obtient :

$$\mathrm{e(t)} = \mathrm{d(t)} - \mathrm{a(t)} \tag{3.9}$$

Avec $\mathrm{e}(t) = [e_1(t)\ e_2(t)\ \ldots\ e_i(t)\ \ldots\ e_S(t)]$ qui désigne le vecteur des erreurs observées sur les S neurones de sortie du réseau. L'apprentissage par correction des erreurs consiste à minimiser un *indice de performance F* basé sur les signaux d'erreur $e_i(t)$, dans le but de faire converger les sorties du réseau avec ce qu'on voudrait qu'elles soient. Un critère très populaire est *la somme des erreurs quadratiques* :

$$F\big(e(t)\big) = \sum_{i=1}^{S} e_i^2(t) = e(t)^T e(t) \tag{3.10}$$

Maintenant, il importe de remarquer que les paramètres libres d'un réseau sont ses poids. Prenons l'ensemble de ces poids et assemblons les sous la forme d'un vecteur $w(t)$ au $temps\ t$. Pour minimiser $F\big(e(t)\big) = F\big(w(t)\big) = F(t)$, nous allons commencer par choisir des poids initiaux ($t = 0$) au hasard, puis nous allons modifier ces poids de la manière suivante :

$$w(t+1) = w(t) + \eta x(t) \tag{3.11}$$

où le vecteur $x(t)$ désigne la direction dans laquelle nous allons chercher le minimum et η est une constante positive déterminant l'amplitude du pas dans cette direction *(la vitesse d'apprentissage)*. L'objectif est de faire en sorte que $F(t+1) < F(t)$. Mais comment peut-on choisir la direction *x* pour que la condition précédente soit respectée ?

Considérons la série de *Taylor* de 1er ordre autour de $w(t)$:[4]

$$F(t+1) = F(t) + \nabla F(t)^T \Delta w(t) \tag{3.12}$$

où $\nabla F(t)$ désigne *le gradient* de F par rapport à ses paramètres libres (les poids w) et $\Delta w(t) = w(t+1) - w(t)$. Or, pour que $F(t+1) < F(t)$, il faut que la condition suivante soit respectée :

$$\nabla F(t)^T \Delta w(t) = \eta \nabla F(t)^T x(t) < 0 \tag{3.13}$$

N'importe quel vecteur $x(t)$ qui respecte l'inégalité de l'équation précédente pointe donc dans une direction qui *diminue F*. On parle alors d'une direction de «*descente*». Pour obtenir une descente maximum, étant donné $\eta > 0$, il faut que le vecteur $\mathrm{x(t)}$ *pointe dans le sens opposé au gradient* car c'est dans ce cas que le produit scalaire sera minimum :

$$\mathrm{x}(t) = -\nabla F(t) \tag{3.14}$$

Ce qui engendre la règle dite de «*descente du gradient*» :

$$\Delta \mathrm{w}(t) = -\eta \nabla F(t) \tag{3.15}$$

illustrée à la *figure 3.9*. Dans l'espace des poids, cette figure montre les courbes de niveau de *F* représentées par des *ellipses hypothétiques*. La flèche en pointillés montre la direction optimale pour atteindre le minimum de *F*. La flèche pleine montre la direction du gradient qui est perpendiculaire à la courbe de niveau en $\mathrm{w}(t)$.

---

[4] *La série de Taylor de F(x) du 1er ordre de la fonction F(x) autour de a est :*

$$F(x) = F(a) + \frac{F'(a)}{1!}(x-a)$$

Dans notre cas $F(x) = F(t+1) = F(w(t+1))$ *et on cherche la décomposition de Taylor autour de* $w(t)$

$$F\big(w(t+1)\big) = F\big(w(t)\big) + \acute{F}\big(w(t)\big)\big(w(t+1) - w(t)\big)$$

$$F\big(w(t+1)\big) = F\big(w(t)\big) + \acute{F}\big(w(t)\big)\Delta w(t)$$

$$F(t+1) = F(t) + \acute{F}(t)\Delta w(t)$$

Avec $\acute{F}(t) = \nabla(F(t))$ *désigne le gradient de F, donc :* $F(t+1) = F(t) + \nabla F(t)^T \Delta w(t)$

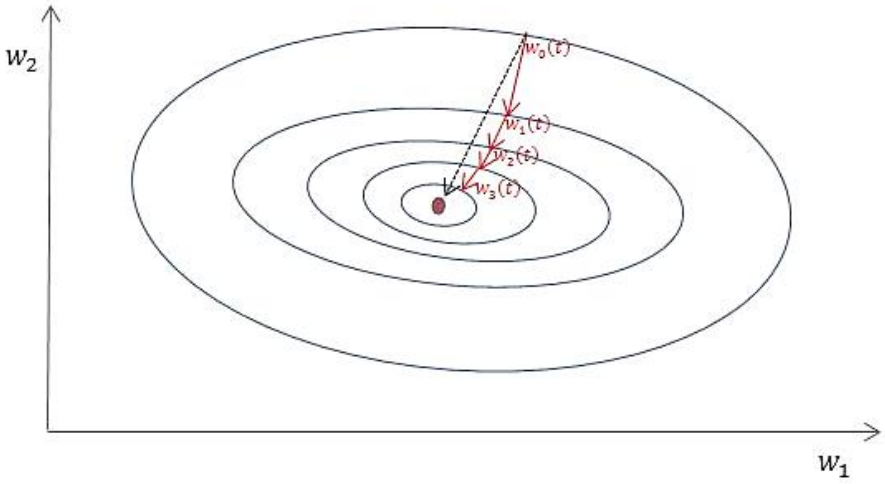

FIG.3.9–*Trajectoire de la descente du gradient*[5]

La règle de la correction des erreurs est utilisée pour beaucoup de réseaux de neurones artificiels, bien qu'elle ne soit pas plausible biologiquement. En effet, comment le cerveau pourrait-il connaître a priori les sorties qu'il doit produire ? Cette règle ne peut être utilisée que dans un contexte d'apprentissage supervisé.

3.5.4. Tâches d'apprentissage :

Les différentes catégories de tâches que l'on peut vouloir réaliser avec un réseau de neurones sont énumérées ci-dessous:

a) Approximation :

Soit la fonction g telle que :

$$d = g(p) \tag{3.16}$$

où p est l'argument de la fonction (un vecteur) et d la valeur (un scalaire) de cette fonction évaluée en p. Supposons maintenant que la fonction $g(\cdot)$ est inconnue. La tâche d'approximation consiste alors à concevoir un réseau de neurones capable d'associer les

[5] *Dans cette figure, on suppose que F est défini sur le plan $\mathbb{R}^2$, et qu'elle représente une forme de bol. Les ellipses correspondent aux lignes de niveaux, c'est à dire les lignes sur lesquelles F est constante. Le vecteur montre la direction de l'opposé du gradient à leur point d'origine. Il est intéressant de noter également que le gradient (et son opposé) en un point donné est orthogonal à la ligne de niveau en ce point. On peut donc observer ici comment la descente de gradient progresse vers le fond du bol, c'est à dire vers le point où la valeur de F est minimale.* [22]

éléments des couples entrée-sortie : $\{(p_1, d_1),\ (p_2, d_2),\ \dots, (p_Q, d_Q)\}$. Ce problème peut être résolu à l'aide d'un apprentissage supervisé sur les Q exemples, avec les p_i représentant les stimuli, et les d_i représentant les sorties désirées pour chacun de ces stimuli, avec $i = 1, 2, \dots, Q$. Ou inversement, on peut aussi dire que l'apprentissage supervisé est un problème d'approximation de fonction ;

b) *Association* :

Il en existe deux types : *l'auto-association* et *l'hétéro-association*. Le problème de l'auto-association consiste à mémoriser un ensemble de patrons (vecteurs) en les présentant successivement au réseau. Par la suite, on présente au réseau une version partielle ou déformée d'un patron original, et la tâche consiste à produire en sortie le patron original correspondant. Le problème de l'hétéro-association consiste quant à lui à associer des paires de patrons : un patron d'entrée et un patron de sortie. L'auto-association implique un apprentissage non supervisé, alors que l'hétéro-association requiert plutôt un apprentissage supervisé.

c) *Classement* :

Pour cette tâche, il existe un nombre fixe de catégories (classes) de stimuli d'entrée que le réseau doit apprendre à reconnaître. Dans un premier temps, le réseau doit entreprendre une phase d'apprentissage supervisée durant laquelle les stimuli sont présentés en entrée et les catégories sont utilisées pour former les sorties désirées, généralement en utilisant une sortie par catégorie. Ainsi, la sortie 1 est associée à la catégorie 1, la sortie 2 à la catégorie 2, etc. Pour un problème comportant Q catégories, on peut par exemple fixer les sorties désirées $d = [d_1, d_2, \dots, d_Q]^T$ à l'aide de l'expression suivante :

$$d_i = \begin{cases} 1 & si\ le\ stimulus\ appartiens\ \text{à}\ la\ catégorie\ i \\ 0 & autrement \end{cases}, \quad i = 1, \dots, Q. \qquad (3.17)$$

Par la suite, dans une phase de reconnaissance, il suffira de présenter au réseau n'importe quel stimulus inconnu pour pouvoir procéder au classement de celui-ci dans l'une ou l'autre des catégories. Une règle simple de classement consiste, par exemple, à choisir la catégorie associée avec la sortie maximale.

d) Prédiction

La notion de prédiction est l'une des plus fondamentales en apprentissage. Il s'agit d'un problème de traitement temporel de signal. En supposant que nous possédons M échantillons passés d'un signal, $x(t-1), x(t-2), \ldots, x(t-M)$, échantillonnés à intervalle de temps fixe, la tâche consiste à prédire la valeur de x au temps t. Ce problème de prédiction peut être résolu grâce à un apprentissage par correction des erreurs, mais d'une manière non supervisée (*sans professeur*), étant donné que les valeurs de sortie désirée peuvent être inférées directement de la série chronologique. Plus précisément, l'échantillon de $x(t)$ peut servir de valeur désirée et le signal d'erreur pour l'adaptation des poids se calcule simplement par l'équation suivante :

$$e(t) = x(t) - \hat{x}(t \mid t-1, t-2, \ldots, t-M), \tag{3.18}$$

où $x(t)$ désigne la sortie désirée et $\hat{x}(t \mid t-1, t-2, \ldots, t-M)$ représente la sortie observée du réseau étant donné les M échantillons précédents. La prédiction s'apparente à la construction d'un modèle physique de la série chronologique. Dans la mesure où le réseau possède des neurones dont la fonction de transfert est non-linéaire, le modèle pourra lui-aussi être non-linéaire.

e) Commande :

La commande d'un processus est une autre tâche d'apprentissage que l'on peut aborder à l'aide d'un réseau de neurones. Considérons un système dynamique non-linéaire $\{u(t), y(t)\}$ où $u(t)$ désigne l'entrée du système et $y(t)$correspond à la réponse de celui-ci. Dans le cas général, on désire commander ce système de manière à ce qu'il se comporte selon un modèle de référence, souvent un modèle linéaire, $\{r(t), d(t)\}$, où pour tout temps $t \geq 0$, on arrive à produire une commande $u(t)$ telle que :

$$\lim_{t \to \infty} |d(t) - y(t)| = 0 \tag{3.19}$$

de manière à ce que la sortie du système suit de près celle du modèle de référence. Ceci peut se réaliser grâce à certains types de réseaux supervisés. [19]

Dans les sections qui suivent, nous allons aborder des réseaux spécifiques en commençant par l'un des plus connus et des plus utilisés : le perceptron simple, le perceptron multicouches et son algorithme de rétropropagation des erreurs.

3.6. Le perceptron

Dans notre programme Matlab de dispatching économique nous allons utiliser un perceptron à deux couches donc nous allons nous intéresser à étudier le « perceptron multicouche » (*Multilayer perceptron*) (PMC). Ce type de réseau est dans la famille générale des réseaux à «propagation vers l'avant» (*feedforward networks*), c'est-à-dire qu'en mode normal d'utilisation, l'information se propage dans un *sens unique*, des entrées vers les sorties sans aucune rétroaction. Son apprentissage est de type supervisé, par correction des erreurs. Dans ce cas uniquement, le signal d'erreur est «rétropropagé» vers les entrées pour mettre à jour les poids des neurones.

Le perceptron multicouche est un des réseaux de neurones les plus utilisés pour des problèmes d'approximation, de classification et de prédiction. Il est habituellement constitué de deux ou trois couches de neurones totalement connectés. Avant d'en étudier le fonctionnement global, nous allons nous attarder à divers cas particuliers plus simples. En particulier, nous allons aborder le cas du *perceptron simple*, c'est-à-dire le perceptron à une seule couche de neurones dont les fonctions d'activation sont de type seuils (*hardlims*).

Le perceptron simple

Le perceptron simple est illustré à la *figure.3.10*. Il s'agit d'une seule couche de S neurones totalement connectée sur vecteur p de R entrées. La matrice $W = [w_1 w_2 \dots w_s]^T$ de dimension $S \times R$ represente l'ensemble des poids de la couche, avec les vecteur-rangées w_i (dimension $R \times 1$) représentant les R poids des connexions reliant le neurone i avec ses entrées. Le vecteur b (dimension $S \times 1$) désigne l'ensemble des S biais de la couche. Les niveau d'activation $n = Wp - b = [n_1 n_2 \dots n_s]^T$ des neurones de la couche servent d'argument à la fonction d'activation qui applique un seuil au niveau 0 pour produire le vecteur des sorties $a = [a_1 a_2 \dots a_s]^T$, où :

$$a_i = \begin{cases} +1 & si\ n_i \geq 0 \\ -1 & autrement \end{cases} \tag{3.20}$$

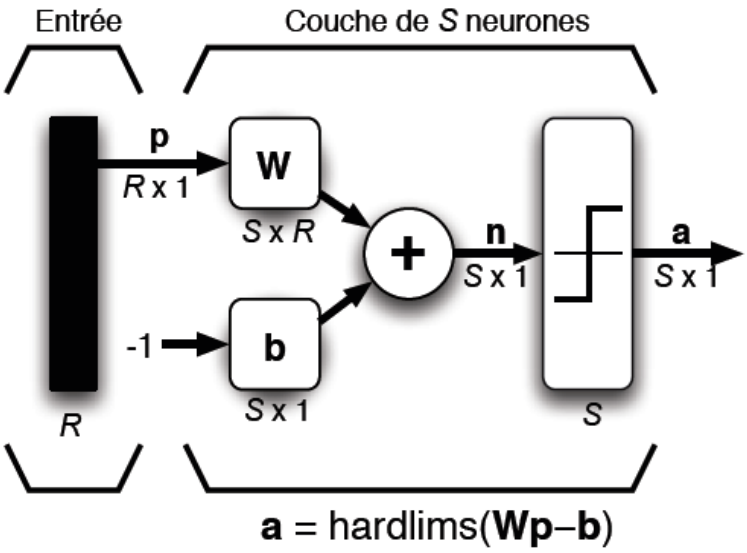

FIG. 3.10 – *Perceptron à une seule couche avec fonction seuil.*

Considérons maintenant le cas le plus simple, à savoir lorsque $R = 2\ et\ S = 1$, c'est-à-dire lorsque la couche n'est formé que d'un seul neurone relié à deux entrées. Dans ce cas, nous aurons $P = [p_1\ p_2],\ W = [w_1]^T = [w_{1,1}\ w_{1,2}],\ b = [b_1]\ et\ a = [a_1],$ où:

$$a = \begin{cases} +1 & si \quad w_{1,1}p_1 + w_{1,2}p_2 \geq b_1 \\ -1 & autrement \end{cases} \tag{3.21}$$

Cette dernière équation nous indique clairement que la sortie du réseau (neurone) peut prendre seulement deux valeurs distinctes selon le niveau d'activation du neurone : −1 lorsque ce dernier est strictement inférieur à 0 ; +1 dans le cas contraire. Il existe donc dans l'espace des entrées une frontière délimitant deux régions correspondantes. Cette frontière est définie par la condition $w_{1,1}p_1 + w_{1,2}p_2 = b_1$ de *l'équation 3.21* qui correspond à l'expression générale d'une droite, telle qu'illustrée à la *figure 3.11*. Etant donné un certain vecteur de poids $W = [w_{1,1}\ w_{1,2}]$, il est aisé de montrer que ce vecteur doit être perpendiculaire à cette droite. En effet, pour tous les points p de la droite, nous avons la relation $w^T P = b,$ où $b = b_1$. Or le terme $w^T P$ correspend à un produit scalaire et l'on sait que : $< x, y >= \|x\|\|y\|cos\theta$, où θ représente l'angle entre les vecteurs x et y. Nous avons donc :

$$< w, p >= \|w\|\|p\|cos\theta = b \tag{3.22}$$

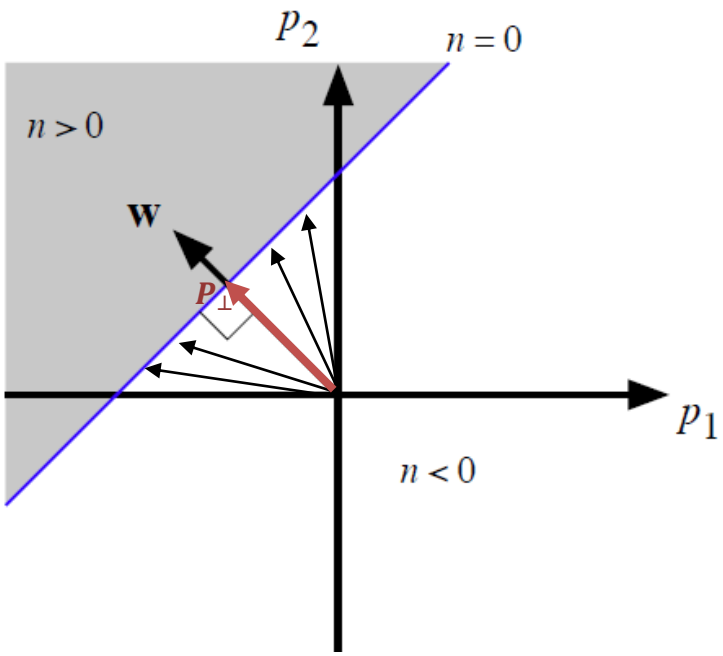

FIG.3.11 – *Frontière de décision pour un perceptron simple à 1 neurone et deux entrées.*

pour tous les points p qui appartiennent à la droite, le produit scalaire doit rester constant. Mais s'il reste constant alors que la norme de p change, c'est parce que l'angle entre les vecteurs doit aussi changer. Soit $P_\perp$, le point de la droite dont le vecteur correspondant possède la plus petite norme. Ce vecteur est perpendiculaire à la droite et sa norme correspond à la distance perpendiculaire entre la droite et l'origine. Maintenant, si sa norme est minimale, c'est que $cos\theta$ est maximal et, par conséquent, que l'angle θ entre $P_\perp$ et w est *nul*. Ainsi, w pointe dans la *même direction* que $P_\perp$et :

$$\|P_\perp\| = \frac{b}{\|\mathrm{w}\|} \tag{3.23}$$

Nous pouvons également déduire que l'origine appartiendra à la région grisée $(n > 0)$ si, et seulement si $(b < 0)$. Autrement, comme à la *figure 3.11*, l'origine appartiendra à la région $(n < 0)$. Si $(b = 0)$, alors la frontière de décision passera par l'origine.

Si l'on considère maintenant le cas où $(S > 1)$, alors chaque neurone i possédera son propre vecteur de poids w_i et son propre biais b_i, et nous nous retrouverons avec S frontières de décision distinctes. Toutes ces frontières de décision seront linéaires. Elles permettront chacune de découper l'espace d'entrée en deux régions infinies, de part et d'autre d'une droite. Chaque neurone d'un perceptron simple permet donc de résoudre parfaitement un problème de classification à deux classes, à condition que celles-ci soient linéairement

séparables. Il ne reste plus qu'à trouver une règle d'apprentissage pour pouvoir déterminer les poids et les biais du réseau permettant de classer au mieux Q couples d'apprentissage :

$$\{(\mathrm{p}_1, \mathrm{d}_1), (\mathrm{p}_2, \mathrm{d}_2), \dots, (\mathrm{p}_\mathrm{Q}, \mathrm{d}_\mathrm{Q})\} \tag{3.24}$$

Pour fixer les idées, considérons le problème particulier, illustré à la *figure 3.12*, consiste à discriminer entre le point noir (p_1) et les points blancs (p_2 et p_3) définis par :

$$\left\{\left(\mathrm{p}_1 = \begin{bmatrix}1\\2\end{bmatrix}, d_1 = +1\right), \left(\mathrm{p}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, d_2 = -1\right), \left(\mathrm{p}_3 = \begin{bmatrix}0\\-2\end{bmatrix}, d_3 = -1\right)\right\} \tag{3.25}$$

Et fixons $S = 1$ (un seul neurone). Il s'agit de trouver un vecteur de poids w correspondant à l'une ou l'autre des frontières de décision illustrées à la *figure 3.12 a*. Pour simplifier davantage, nous supposons pour cet exemple que $b = 0$, de sorte que les frontières de décision induites par w passent toutes par l'origine. Nous allons donc l'initialiser aléatoirement, par exemple $w = \begin{bmatrix}1\\-1\end{bmatrix}$ (voir *figure 3.12 b*)

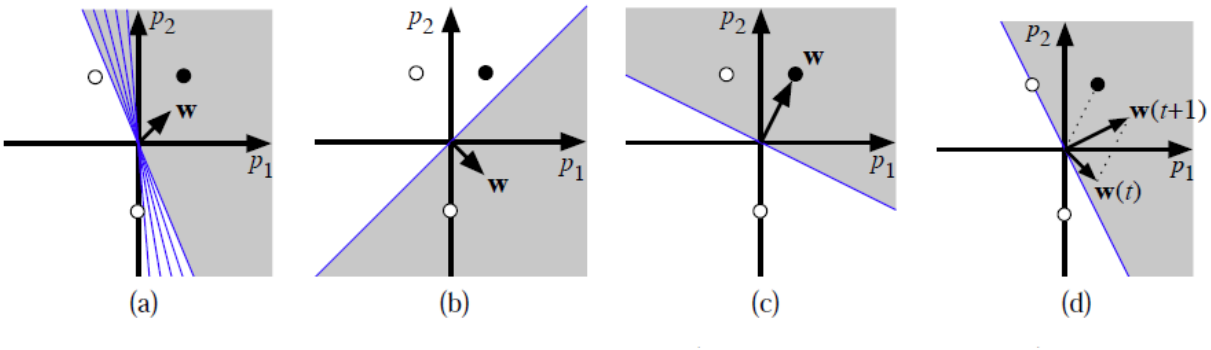

FIG. 3.12 – *Exemple d'un problème à deux classes (points noirs vs points blancs).*

Considérons le point p_1 (point noir). La sortie du réseau pour ce point est donnée par :

$$a = hardlims(\mathrm{w}^T\mathrm{p}_1) = hardlims\left(\begin{bmatrix}1 & -1\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix}\right) = hardlims(-1) = -1 \tag{3.26}$$

Or, la sortie désirée pour ce point est +1 (les zones grises à la *figure 3.12* produisent une sortie +1). Le réseau n'a donc pas le comportement désiré, il faudra modifier le vecteur w. on peut remarquer que dans le cas particulier de ce problème simplifié, la norme de w ne compte pas car le biais est nul, seule son orientation importe.

Comment pourrions-nous modifier w pour que le réseau puisse classifier adéquatement le point p_1 ? Une solution consisterait à fixer $\mathrm{w} = \mathrm{p}_1$, tel qu'illustré à la *figure 3.12c*. De cette manière, le point p_1 serait parfaitement classer. Mais le problème avec cette approche est que la frontière de décision bondirait d'un stimulus à l'autre au fil de l'apprentissage ce qui pourrait engendrer des oscillations et empêcher la convergence dans certains cas. La solution consiste donc à prendre une position intermédiaire en approchant la direction de $w$de celle de p_1:

$$\mathrm{w}(t+1) = \mathrm{w}(t) + \mathrm{p}_1 \tag{3.27}$$

Tel qu'illustré à la *figure .3.12 d*. Cette règle fonctionne bien pour la catégorie de stimulus où l'on désire obtenir une sortie +1. Dans la situation inverse, il faut au contraire éloigner w de p_1. Définissons un signal d'erreur $e = \frac{d-a}{2}$ où $e \in \{-1, 0, -1\}$. Alors, nous avons l'ensemble suivant de règle :

$$\Delta\mathrm{w} = \begin{Bmatrix} \mathrm{p} & si\ e = +1 \\ 0 & si\ e = 0 \\ -\mathrm{p} & si\ e = -1 \end{Bmatrix} \tag{3.28}$$

où $\Delta w = w(t+1) - w(t)$ et p est le stimilus que l'on cherche à apprendre. Dans le cas où $\neq 0$, on peut aussi mettre à jour le biais en observant simplement que celui-ci n'est rien d'autre qu'un poids comme les autres, mais dont l'entrée est fixée à -1. Ainsi :

$$\Delta b = -e \tag{3.29}$$

Et dans le cas général où l'on dispose de *S* neurones, on peut réécrire *l'équation 3.28* sous forme matricielle de la façon suivant :

$$\Delta\mathrm{W} = \mathrm{ep}^T \tag{3.30}$$

$$\Delta\mathrm{b} = -\mathrm{e} \tag{3.31}$$

où $\mathrm{e} = [e_1 e_2 \ \ldots \ e_s]^T = \mathrm{d} - \mathrm{a}$ est le vecteur des erreurs que l'on observe en sortie pour le stimulus p.

Malgré sa relative simplicité, la règle du perceptron s'avère très puissante. On peut facilement expérimenter avec cette règle grâce à la « Neural Network toolbox » de Matlab, programme de démonstration nnd4pr.

Nous ne démontrerons pas ici qu'elle converge toujours vers une solution en un nombre fini d'itérations, mais pour savoir qu'une telle preuve existe. Il importe cependant de connaître les hypothèses sous-jacentes à cette preuve :

1. Le problème doit être linéairement séparable ;
2. Les poids ne sont mis à jour que lorsqu'un stimulus d'entrée est classé incorrectement ;
3. Il existe une borne supérieure sur la norme des vecteurs de poids.

La première hypothèse va de soit car s'il n'existe aucune solution linéaire au problème, on ne peut pas s'attendre à ce qu'un réseau qui ne peut produire que des solutions linéaires puisse converger.

La deuxième hypothèse est implicite dans *l'équation 3.30*. Lorsque le signal d'erreur e est nul, le changement de poids ΔW est également nul. La troisième hypothèse est plus subtile mais non limitative. Si l'on s'arrange pour conserver le ratio $\frac{\|w\|}{b}$ constant, sans changer l'orientation de w pour un neurone donné, on ne change aucunement la frontière de décision que ce neurone engendre. Sans perte de généralité, on peut donc réduire la norme des poids lorsque celle-ci devient trop grande.

Mais qu'entend-on par un problème à deux classes «linéairement séparables» ? Et bien simplement un problème de classification dont la frontière de décision permettant de séparer les deux classes peut s'exprimer sous la forme d'un *hyperplan* (plan dans un espace à n dimensions). Par exemple, les problèmes de la *figure 3.13* ne sont pas séparables en deux dimensions (par de simples droites). Des frontières possibles sont dessinées en pointillés. Elles sont toutes non linéaires.

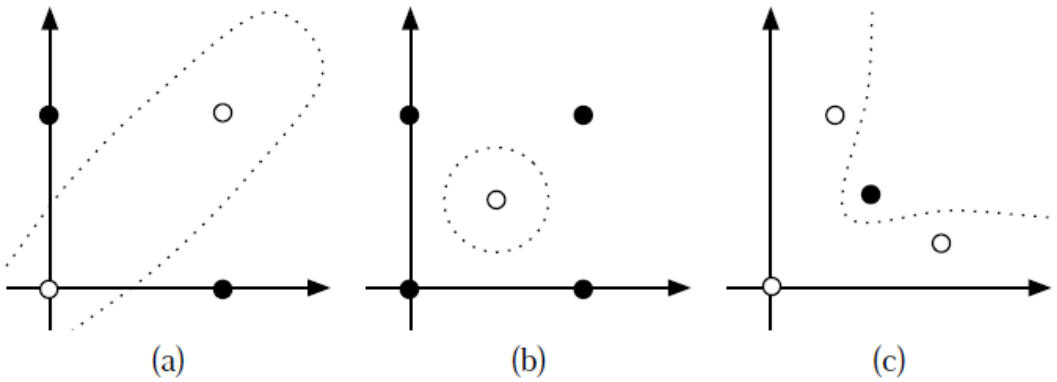

FIG. 3.13 – *Exemples de problèmes non linéairement séparables.*

3.7. Réseau multicouche

Jusqu'à présent, nous n'avons traité que des réseaux à une seule couche de neurones. Nous avons aussi vu que ces réseaux ne pouvaient résoudre que des problèmes de classification linéairement séparables. Les réseaux multicouches permettent de lever cette limitation. On peut même démontrer qu'avec un réseau de trois couches (deux couches cachées + une couche de sortie), comme celui de la *figure 3.7*, on peut construire des frontières de décision de complexité quelconque, ouvertes ou fermées, concaves ou convexes, à condition d'employer une fonction de transfert non linéaire et de disposer de suffisamment de neurones sur les couches cachées. Un réseau multicouche n'est rien d'autre qu'un assemblage de couches concaténées les unes aux autres, de la gauche vers la droite, en prenant les sorties d'une couche et en les injectant comme les entrées de la couche suivante. A la section suivante, nous allons développer l'algorithme dit de «rétropropagation des erreurs» qui permet d'entrainer un réseau multicouche. Mais pour l'instant nous allons tenter d'illustrer à quoi servent les couches supplémentaires. Une chose que l'on peut déjà remarquer est qu'il ne sert à rien d'assembler plusieurs couches ADALINE (Appendice B) car la combinaison de plusieurs couches linéaires peut toujours se ramener à une seule couche linéaire équivalente. C'est pourquoi, pour être utile, un réseau multicouche doit toujours posséder des neurones avec fonctions de transfert non-linéaires sur ses couches cachées. Sur sa couche de sortie, selon le type d'application, il pourra comporter des neurones linéaires ou non-linéaires.

3.7.1. Problème du ou exclusif

A la *figure 3.13 a*, nous avons illustré un problème de classification non séparable linéairement. Il s'agit du problème classique du «ou exclusif» (xor) que l'on ne peut pas résoudre ni avec un perceptron simple, ni avec un réseau ADALINE, car les points noirs ne peuvent pas être séparés des blancs à l'aide d'une seule frontière de décision linéaire. Dans ce problème, les points noirs représentent *le vrai* (valeur 1) et les points blancs *le faux* (valeur 0). Le «ou exclusif», pour être vrai, exige qu'une seule de ses entrées soit vraie, sinon il est faux. On peut résoudre facilement ce problème à l'aide du réseau multicouche illustré à la *figure 3.14*.

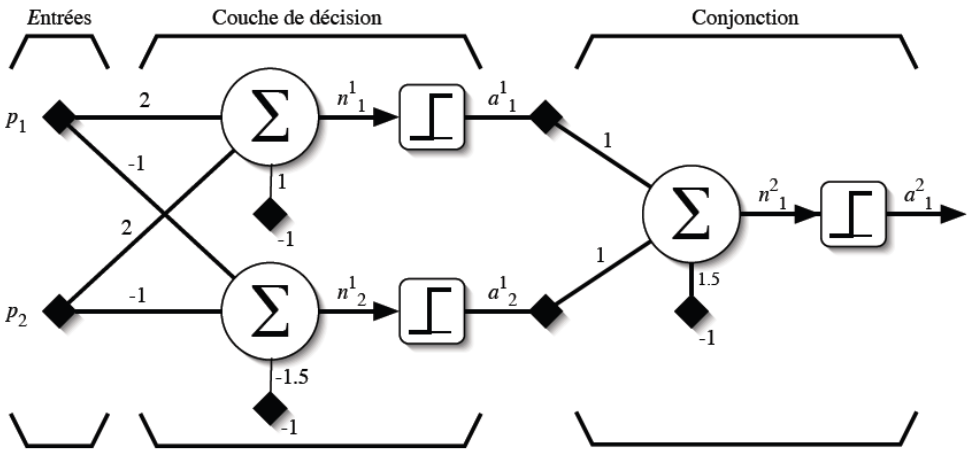

FIG. 3.14 –*Réseau multicouche pour résoudre le problème du «ou exclusif».*

Ce réseau à deux couches utilise des fonctions de transfert seuil. Sur la première couche, chaque neurone engendre les frontières de décision illustrées aux *figures 3.15 a et 3.15 b*.

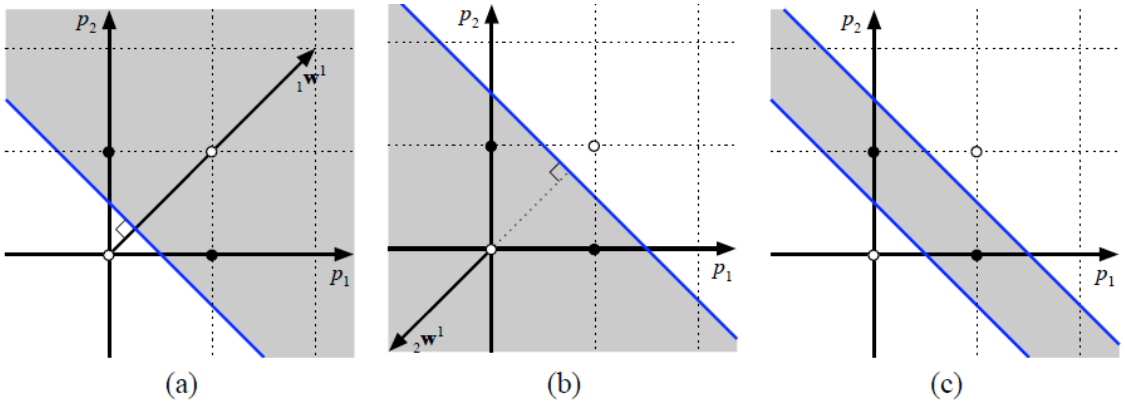

FIG. 3.15 – *Frontières de décision engendrées par le réseau de la figure 3.14 :*
(a) neurone 1 de la couche 1 ; (b) neurone 2 de la couche 1 ; (c) neurone 1 de la couche 2.

Les zones grisées représentent la région de l'espace d'entrée du réseau pour laquelle le neurone correspondant produit une réponse vrai. Le rôle du neurone sur la couche de sortie, illustré à la *figure 3.15 c*, consiste à effectuer la conjonction des deux régions produites par les neurones de la première couche. Notez bien que les entrées de la deuxième couche sont les sorties de la première couche. La *figure 3.15* représente toutes les frontières de décision dans l'espace des entrées. La frontière de décision engendrée par le neurone de la couche de sortie est aussi illustrée dans son propre espace d'entrée à la *figure 3.16*.

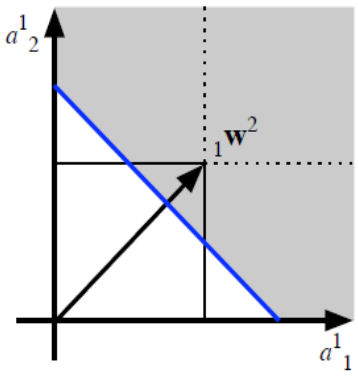

FIG. 3.16 – *Frontière de décision engendrée par le neurone qui effectue une conjonction.*

Il importe de remarquer que la sortie des fonctions seuils employées étant limitée aux valeurs {0, 1} (que l'on interprète comme étant respectivement faux et vrai), seuls les coins du carré illustré à la figure sont pertinents. Pour réaliser une conjonction, le neurone effectue donc la somme de ses deux entrées et fixe un seuil à 1.5. Si la somme est inférieure à 1.5, alors il produit vrai en sortie, sinon il produit faux. Dans ce cas, seul le coin supérieur droit du carré produit vrai en sortie.

Mentionnons finalement que le réseau de la *figure 3.14* n'est pas le seul à pouvoir résoudre ce problème du «ou exclusif». D'autres combinaisons de poids et de biais pourraient produire le même résultat.

3.7.2. Approximation de fonctions

Pour faire de l'approximation de fonction, on peut montrer qu'un réseau multicouche comme celui de la *figure 3.17*, avec une seule couche cachée de neurones *sigmoïdes* et une couche de sortie avec des neurones *linéaires* permet *d'approximer* n'importe quelle fonction d'intérêt avec une précision arbitraire, à condition de disposer de suffisamment de neurones sur la couche cachée. Intuitivement, un peu à la façon des séries de Fourier qui utilisent des sinus et cosinus, cette preuve passe par la démonstration que l'on peut approximer n'importe quelle fonction d'intérêt par une combinaison linéaire de sigmoïdes. C'est le cas dans notre réseau de neurone du dispatching économique de la partie programmation, qui est constitué d'une couche cachée avec une fonction d'activation *tansig* et d'une couche de sortie avec la fonction d'activation *purelin*.

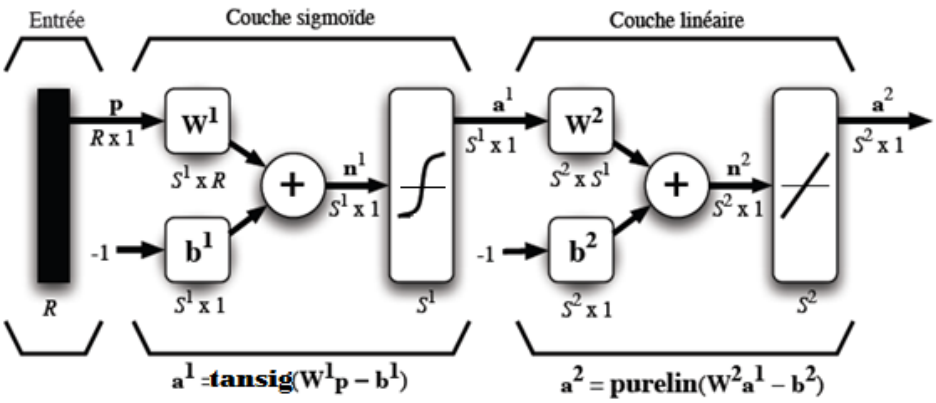

FIG. 3.17 – *Réseau multicouche permettant de faire l'approximation de fonctions.*

3.7.3. Classification

Pour faire de la classification, on utilisera des réseaux soit à deux, soit à trois couches de neurones sigmoïdes. On peut montrer qu'une seule couche cachée suffit à engendrer des frontières de décision convexes, ouvertes ou fermées, de complexité arbitraire, alors que deux couches cachées permettent de créer des frontières de décision concaves [6] ou convexes, ouvertes ou fermées, de complexité arbitraire. La *figure 3.18* montre en deux dimensions différents types de frontières de décision.

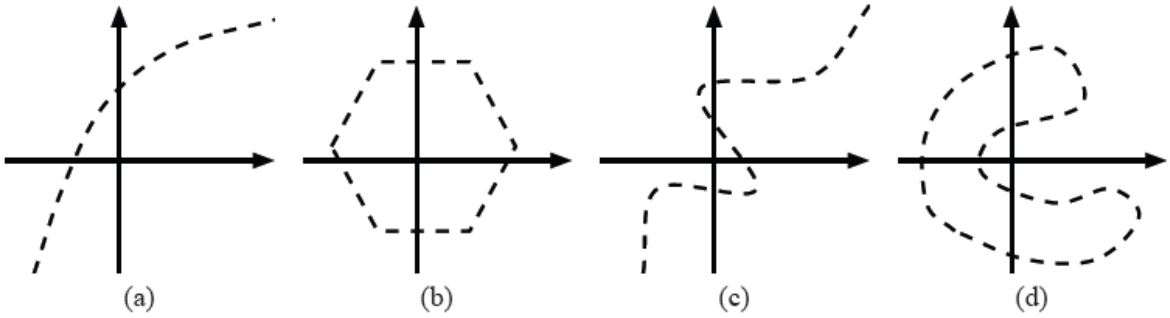

FIG. 3.18 – *Exemples de frontières de décision : (a) convexe ouverte ; (b) convexe fermée ; (c) concave ouverte; et (d) concave fermée.*

Intuitivement, on veut voir que la première couche cachée d'un tel réseau sert à découper l'espace d'entrée à l'aide de frontières de décision linéaires, comme on l'a vu pour le perceptron simple, la deuxième couche sert à assembler des frontières de décision non linéaires (*Les non-linéarités proviennent des sigmoïdes*) convexes en sélectionnant ou en retranchant des régions engendrées par la couche précédente et, de même, la couche de sortie

[6] *Une courbe (surface) convexe ne comporte aucun changement dans le signe de la courbure, alors qu'une courbe concave implique un point d'inflexion.*

permet d'assembler des frontières de décision concaves en sélectionnant ou en retranchant des régions convexes engendrées par la couche précédente.

Avant de passer à l'algorithme de rétropropagation qui nous permettra d'entraîner un réseau multicouche, que nous nommerons dorénavant *perceptron multicouche* ou PMC, mentionnons que ce n'est pas par hasard que nous avons remplacé la fonction de transfert seuil par la fonction sigmoïde, mais bien pour pouvoir procéder à un *apprentissage automatique*. Par exemple, même si nous avons pu construire à la main, avec la fonction seuil, le réseau de la *figure 3.14* pour résoudre le problème du «ou exclusif», nous ne saurions pas comment apprendre automatiquement à générer les bons poids et les bons biais de ce réseau. Le problème avec la fonction seuil est que sa dérivée est toujours nulle sauf en un point où elle n'est même pas définie. [23]

3.8. Rétropropagation de l'erreur (Backpropagation)

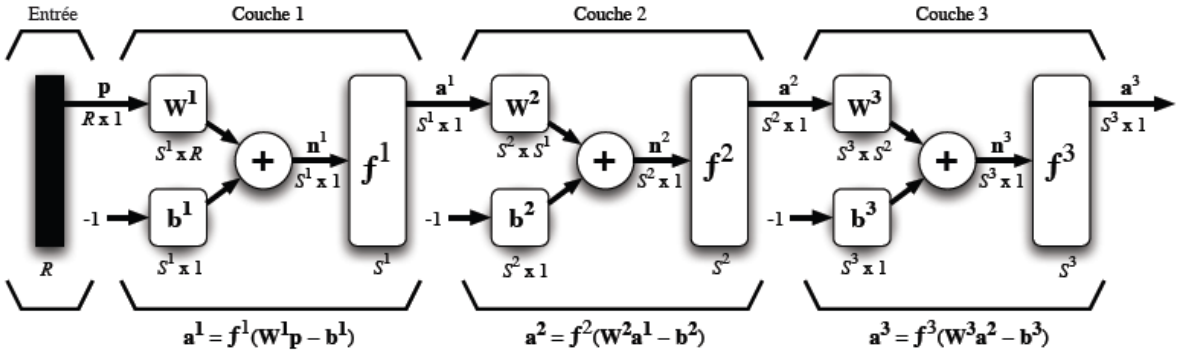

FIG. 3.19 – *Représentation matricielle d'un réseau de trois couches*

L'équation qui décrit les sorties d'une couche k dans un perceptron multicouche est donnée par :

$$a^k = f^k(W^k a^{k-1} - b^k), \quad pour\ k = 1, \dots, M, \tag{3.32}$$

où M est le nombre total de couches et $a^0 = p$ définit le cas de base de cette formule de récurrence. Les sorties du réseau correspondent alors à a^M. L'algorithme de rétropropagation utilise comme indice de performance *l'erreur quadratique moyenne*, et permet un apprentissage de type supervisé avec un ensemble d'association *stimulus/cible* $\{(\mathrm{p}_q, \mathrm{d}_q)\}$, $q = 1, \dots, Q$ où p_q représente le vecteur stimulus (entrées) et d_q un vecteur cible (sortie désirée). A chaque instant *t*, on peut propager vers l'avant un stimulus différent $\mathrm{p}(t)$ à travers le réseau de la *figure 3.19* pour obtenir un vecteur de sorties $\mathrm{a}(t)$. Ceci nous permet de

calculer l'erreur $\mathrm{e}(t)$ entre ce que le réseau produit en sortie pour ce stimulus et la cible $\mathrm{d}(t)$ qui lui est associée :

$$\mathrm{e}(t) = \mathrm{d}(t) - \mathrm{a}(t) \tag{3.33}$$

L'indice de performance F permet de minimiser l'erreur quadratique moyenne :

$$F(x) = E[\mathrm{e}^T(t)\mathrm{e}(t)] \tag{3.34}$$

où $E[.]$ désigne *l'espérance mathématique* et le vecteur *x* regroupe l'ensemble des poids et des biais du réseau. Nous allons approximer cet indice par l'erreur instantanée :

$$\hat{F}(x) \ = \mathrm{e}^T(t)\mathrm{e}(t) \tag{3.35}$$

Nous allons utiliser la méthode de descente de gradient pour optimiser *x* :

$$\Delta w_{i,j}^k(t) = -\eta \frac{\partial \hat{F}}{\partial w_{i,j}^k} \tag{3.36}$$

$$\Delta b_i^k(t) = -\eta \frac{\partial \hat{F}}{\partial b_i^k} \tag{3.37}$$

où η désigne le taux d'apprentissage.

Pour calculer la dérivée partielle de $\hat{F}$, il faudra faire appel à la règle de chaînage des dérivées :

$$\frac{df[n(w)]}{dw} = \frac{df[n]}{dn} \times \frac{dn(w)}{dw} \tag{3.38}$$

Par exemple, si $f[n] = e^n$ et $n = 2w, donc \ f[w] = e^{2w}$

$$\frac{df[n(w)]}{dw} = \left(\frac{de^n}{dn}\right) \times \left(\frac{d2w}{dw}\right) = e^n \times 2 = 2e^{2w} \tag{3.39}$$

Nous allons nous servir de cette règle pour calculer les dérivées partielles des *équations 3.36 et 3.37* :

$$\frac{\partial \hat{F}}{\partial w_{i,j}^k} = \frac{\partial \hat{F}}{\partial n_i^k} \times \frac{\partial n_i^k}{\partial w_{i,j}^k} \tag{3.40}$$

$$\frac{\partial \hat{F}}{\partial b_i^k} = \frac{\partial \hat{F}}{\partial n_i^k} \times \frac{\partial n_i^k}{\partial b_i^k} \tag{3.41}$$

Le deuxième terme de ces équations est facile à calculer car les niveaux d'activation n_i^k de la couche k dépendent directement des poids et des biais sur cette couche :

$$n_i^k = \sum_{j=1}^{s^{k-1}} w_{i,j}^k a_j^{k-1} - b_i^k \tag{3.42}$$

Par conséquent :

$$\frac{\partial n_i^k}{\partial w_{i,j}^k} = a_j^{k-1} \ , \quad \frac{\partial n_i^k}{\partial b_i^k} = -1 \tag{3.43}$$

On remarque que cette partie de la dérivée partielle de $\hat{F}$ par rapport à un poids (ou un biais) est toujours égale à l'entrée de la connexion correspondante.

Maintenant, pour le premier terme des *équations 3.40 et 3.41*, définissons *la sensibilités* s_i^k de $\hat{F}$ aux changements dans le niveau d'activation n_i^k du neurone*i*de la couche *k*:

$$s_i^k = \frac{\partial \hat{F}}{\partial n_i^k} \tag{3.44}$$

On peut alors réécrire les *équations 3.40 et 3.41* de la façon suivante :

$$\frac{\partial \hat{F}}{\partial w_{i,j}^k} = s_i^k a_j^{k-1} \tag{3.45}$$

$$\frac{\partial \hat{F}}{\partial b_i^k} = -s_i^k \tag{3.46}$$

et les expressions des *équations 3.36 et 3.37* de la façon suivante :

$$\Delta w_{i,j}^{k}(t) = -\eta s_i^k(t) a_j^{k-1}(t) \tag{3.47}$$

$$\Delta b_i^k(t) = \eta s_i^k(t) \tag{3.48}$$

ce qui donne en notation matricielle :

$$\Delta \mathrm{W}^k(t) = -\eta s^k(t)(a^{k-1})^T(t) \tag{3.49}$$

$$\Delta \mathrm{b}^k(t) = \eta s^k(t) \tag{3.50}$$

avec :

$$s^k \equiv \frac{\partial \hat{F}}{\partial \mathrm{n}^k} = \begin{bmatrix} \frac{\partial \hat{F}}{\partial n_1^k} \\ \frac{\partial \hat{F}}{\partial n_2^k} \\ \vdots \\ \frac{\partial \hat{F}}{\partial n_{sk}^k} \end{bmatrix} \tag{3.51}$$

Par rapport à la règle LMS (Appendice B), il est intéressant de noter la ressemblance des équations ci-dessus avec les *équations B.19 et B.20*. On remarque que le terme $2\mathrm{e}(t)$ est simplement remplacé par $s^M(t)$.

3.8.1. Calcul des sensibilités

Il reste maintenant à calculer les sensibilités s^k, ce qui requerra une nouvelle application de la règle de chaînage des dérivées. Dans ce cas, nous obtiendrons une formule de récurrence où la sensibilité des couches en amont (entrées) dépendra de la sensibilité des couches en aval (sorties). C'est de là que provient l'expression «rétropropagation», car le sens de propagation de l'information est inversé par rapport à celui de *l'équation 3.32*.

Pour dériver la formule de récurrence des sensibilités, nous allons commencer par calculer la matrice suivante :

$$\frac{\partial n^{k+1}}{\partial n^{k}} = \begin{bmatrix} \frac{\partial n_1^{k+1}}{\partial n_1^k} & \frac{\partial n_1^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_1^{k+1}}{\partial n_{sk}^k} \\ \frac{\partial n_2^{k+1}}{\partial n_1^k} & \frac{\partial n_2^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_2^{k+1}}{\partial n_{sk}^k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial n_{s^{k+1}}^{k+1}}{\partial n_1^k} & \frac{\partial n_{s^{k+1}}^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_{s^{k+1}}^{k+1}}{\partial n_{sk}^k} \end{bmatrix} \tag{3.52}$$

Cette matrice énumère toutes les sensibilités des niveaux d'activation d'une couche par rapport à ceux de la couche précédente. Considérons chaque élément (i, j) de cette matrice :

$$\frac{\partial n_j^{k+1}}{\partial n_j^k} = \frac{\partial}{\partial n_j^k}\left(\sum_{l=1}^{s^k} w_{i,l}^{k+1} a_l^k - b_i^{k+1}\right) = w_{i,j}^{k+1}\frac{\partial a_j^k}{\partial n_j^k} \tag{3.53}$$

$$= w_{i,j}^{k+1}\frac{\partial f^k(n_j^k)}{\partial n_j^k} = w_{i,j}^{k+1}\dot{f}^k\left(n_j^k\right) \tag{3.54}$$

avec :

$$\dot{f}^k\left(n_j^k\right) = \frac{\partial f^k\left(n_j^k\right)}{\partial n_j^k} \tag{3.55}$$

Par conséquent, la matrice de *l'équation 3.52* peut s'écrire de la façon suivante :

$$\frac{\partial \mathrm{n}^{k+1}}{\partial \mathrm{n}^{k}} = \mathrm{W}^{k+1}\dot{\mathrm{F}}^k(\mathrm{n}^k) \tag{3.56}$$

$$\dot{\mathrm{F}}^k(\mathrm{n}^k) = \begin{bmatrix} \dot{f}^k(n_1^k) & 0 & \cdots & 0 \\ 0 & \dot{f}^k(n_2^k) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \dot{f}^k(n_{sk}^k) \end{bmatrix} \tag{3.57}$$

Ceci nous permet maintenant d'écrire la relation de récurrence pour les sensibilités :

$$s^k = \frac{\partial \hat{F}}{\partial \mathrm{n}^k} = \left(\frac{\partial \mathrm{n}^{k+1}}{\partial \mathrm{n}^k}\right)^T \frac{\partial \hat{F}}{\partial \mathrm{n}^{k+1}} = \dot{\mathrm{F}}^k(\mathrm{n}^k)(\mathrm{W}^{k+1})^T \frac{\partial \hat{F}}{\partial \mathrm{n}^{k+1}}$$

$$s^k = \dot{\mathrm{F}}^k(\mathrm{n}^k)(\mathrm{W}^{k+1})^T \mathrm{s}^{k+1} \tag{3.58}$$

Cette équation nous permet de calculer s^1 à partir de s^2, qui lui-même est calculé à partir de s^3, etc., jusqu'à s^M. Ainsi les sensibilités sont rétropropagées de la couche de sortie jusqu'à la couche d'entrée :

$$s^M \rightarrow s^{M-1} \rightarrow \dots \; s^2 \rightarrow s^1 \tag{3.59}$$

Il ne nous reste plus qu'à trouver le cas de base, s^M, permettant de mettre fin à la récurrence

$$s_i^M = \frac{\partial \hat{F}}{\partial n_i^M} = \frac{\partial (\mathrm{d} - \mathrm{a}^M)^T (\mathrm{d} - \mathrm{a}^M)}{\partial n_i^M} = \frac{\partial}{\partial n_i^M} \left(\sum_{l=1}^{s^M} (d_l - a_l^M)^2 \right)$$

$$= -2(d_i - a_i^M) \frac{\partial a_i^M}{\partial n_i^M}$$

$$= -2(d_i - a_i^M) \dot{f}^M(n_i^M) \tag{3.60}$$

En notation matricielle, on écrit :

$$s^M = -2\dot{\mathrm{F}}^M(\mathrm{n}^M)(\mathrm{d} - \mathrm{a}^M) \tag{3.61}$$

3.8.2. Algorithme d'entraînement

Voici donc un résumé de la démarche à suivre pour entraîner un perceptron multicouche :

1. Initialiser tous les poids du réseau à de petites valeurs aléatoires.
2. Pour chaque association $(\mathrm{p}_q, \mathrm{d}_q)$ dans la base d'apprentissage :
   (a) Propager les entrées p_q vers l'avant à travers les couches du réseau :

$$a^0 = \mathrm{p}_q \tag{3.62}$$

$$a^k = \mathrm{f}^k(\mathrm{W}^k \mathrm{a}^{k-1} - \mathrm{b}^k), \quad pour\ k = 1, \dots, M. \tag{3.63}$$

   (b) Rétropropager les sensibilités vers l'arrière à travers les couches du réseau :

$$s^M = -2\dot{\mathrm{F}}^M(\mathrm{n}^M)(\mathrm{d}_q - \mathrm{a}^M) \tag{3.64}$$

$$s^k = \dot{\mathrm{F}}^M(\mathrm{n}^k)(\mathrm{W}^{k+1})^T s^{k+1} \;, pour\ \ k = M-1, \dots, 1. \tag{3.65}$$

(c) Mettre à jour les poids et biais :

$$\Delta \mathrm{W}^k = -\eta s^k (a^{k-1})^T \ , pour\ k = 1, \dots, M, \tag{3.66}$$

$$\Delta \mathrm{b}^k = \eta s^k \ , pour\ k = 1, \dots, M \tag{3.67}$$

3. Si le critère d'arrêt est atteint, alors *stop*.
4. Sinon, permuter l'ordre de présentation des associations de la base d'apprentissage.
5. Recommencer à l'étape 2.

3.8.3. Critères d'arrêt

Plusieurs critères d'arrêts peuvent être utilisés avec l'algorithme de rétropropagation des erreurs. Le plus commun consiste à fixer un nombre maximum de périodes d'entrainement, ce qui fixe effectivement une limite supérieure sur la durée de l'apprentissage. Ce critère est important car la rétropropagation des erreurs n'offre aucune garantie quant à la convergence de l'algorithme. Il peut arriver, par exemple, que le processus d'optimisation reste pris dans un minimum local. Sans un tel critère, l'algorithme pourrait ne jamais se terminer.

Un deuxième critère commun consiste à fixer une borne inférieure sur l'erreur quadratique moyenne, ou encore sur la racine[7] carrée de cette erreur. Dépendant de l'application, il est parfois possible de fixer a priori un objectif à atteindre. Lorsque l'indice de performance choisi diminue en dessous de cet objectif, on considère simplement que le PMC a suffisamment bien appris ses données et on arrête l'apprentissage.

Un processus d'apprentissage par correction des erreurs, comme celui de la rétropropagation, vise à réduire autant que possible l'erreur que commet le réseau. Mais cette erreur est mesurée sur un ensemble de données d'apprentissage. Si les données sont bonnes, c'est-à-dire quelles représentent bien le processus physique sous-jacent que l'on tente d'apprendre ou de modéliser, et que l'algorithme a convergé sur un optimum global, alors il devrait bien performer sur d'autres données issues du même processus physique. Cependant, si les données d'apprentissage sont partiellement corrompues par du bruit[8] ou par des erreurs de mesure, alors il n'est pas évident que la performance optimale du réseau sera atteinte en minimisant l'erreur, lorsqu'on la testera sur un jeu de données différent de celui qui a servi à

[7] *On parle alors de la racine de l'erreur quadratique moyenne. En anglais, on dit «Root Mean Square» ou RMS.*
[8] *Une information mal classée.*

l'entrainement. On parle alors de la capacité du réseau à généraliser, c'est-à-dire de bien performer avec des données qu'il n'a jamais vu auparavant.

Par exemple, la *figure 3.20* illustre le problème du surapprentissage dans le contexte d'une tâche d'approximation de fonction. La droite en pointillés montre une fonction linéaire que l'on voudrait approximer en ne connaissant que les points noirs. La courbe en trait plein montre ce qu'un réseau hypothétique pourrait apprendre. On constate que la courbe passe par tous les points d'entraînement et donc que l'erreur est nulle. De toute évidence, ce réseau ne généralisera pas bien si l'on échantillonne d'autres points sur la droite

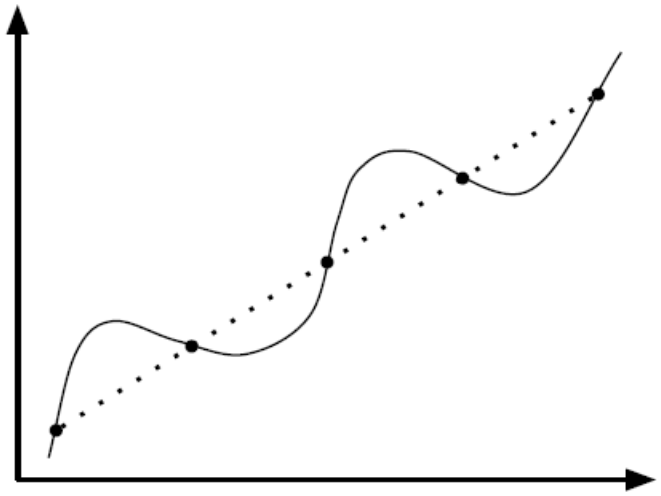

FIG. 3.20 – *Illustration du phénomène de surapprentissage pour le cas simple d'une approximation de fonction*.

Une solution à ce problème consiste à utiliser un autre critère d'arrêt basé sur une technique dite de validation croisée (*cross-validation*). Cette technique consiste à utiliser deux ensembles indépendants[9]de données pour entrainer notre réseau : un pour l'apprentissage (l'ajustement des poids) et l'autre pour la *validation*, c'est-à-dire le calcul d'un indice de performance (une erreur, un taux de reconnaissance ou tout autre mesure pertinente à l'application). Le critère d'arrêt consiste alors à stopper l'apprentissage lorsque l'indice de performance calculé sur les données de validation cesse de s'améliorer pendant plusieurs périodes d'entrainement. La *figure 3.21* illustre le critère de la validation croisée dans le cas d'un indice de performance que l'on cherche à minimiser. La courbe en pointillés de ce graphique représente l'indice de performance d'un réseau hypothétique (Des courbes semblables s'observent couramment dans la pratique) calculé sur les données d'apprentissage,

[9] *En pratique cela consiste à partitionner les données disponibles en deux ensembles distincts.*

alors que la courbe en trait plein montre le même indice mais calculé sur les données de validation. On voit qu'il peut exister un moment au cours de l'apprentissage où l'indice en validation se détériore alors que le même indice continue à s'améliorer pour les données d'entraînement. C'est alors le début du «surapprentissage».

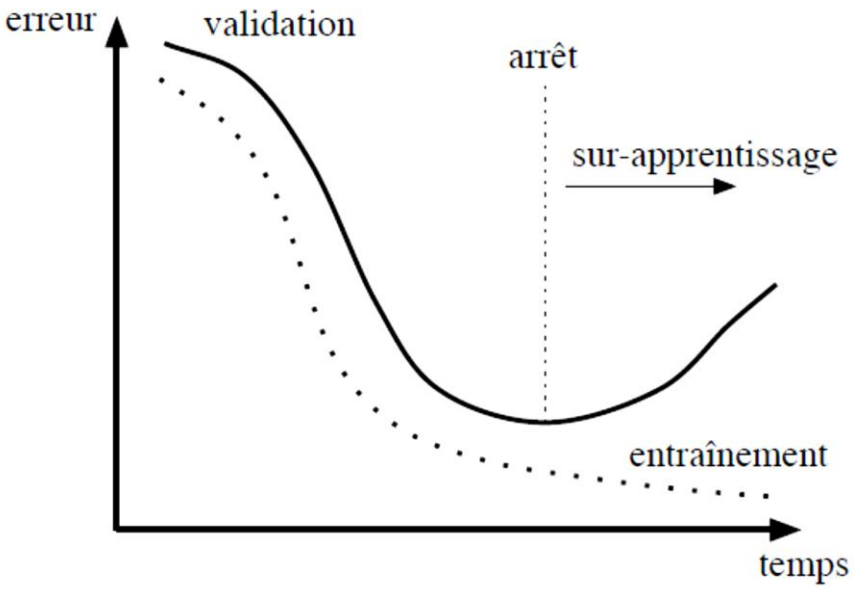

FIG. 3.21 – *Illustration de la validation croisée.*

3.8.4. Phénomène de saturation

Une autre considération pratique dont on doit tenir compte lorsqu'on entraine un PMC concerne le phénomène de saturation des neurones où, sous certaines conditions, les neurones peuvent à toute fin pratique cesser d'apprendre tellement leur convergence devient lente. Considérons par exemple le réseau de la *figure 3.22*, constitué d'un seul neurone à deux entrées avec $p_1 = p_2 = 100$. Si l'on calcule son niveau d'activation n, on obtient :

$$n = 100 \times 0.3 + 100 \times 0.4 + 0.5 = 70.5 \qquad (3.68)$$

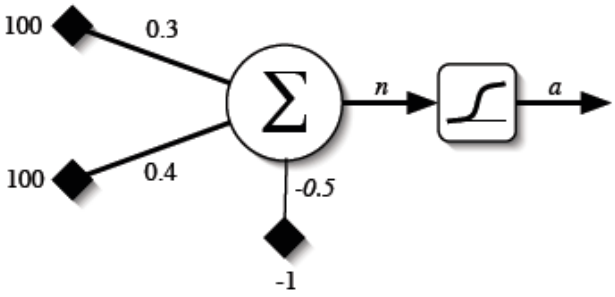

FIG. 3.22 – *Exemple d'un neurone saturé.*

On peut déjà remarquer que l'effet du biais est négligeable devant celui des deux poids d'entrée, malgré le fait qu'ils soient tous les trois du même ordre de grandeur, à cause de l'amplitude des entrées. Si l'on calcule la sortie du neurone, on obtient :

$$a = logsig(n) = \frac{1}{1+\exp(-n)} = \frac{1}{1+\exp(-70.5)} \approx 1 \qquad (3.69)$$

En effet, $\exp(-70.5) = 2.4 \times 10^{-31}$. On dit alors que le neurone est saturé. Le problème avec un tel neurone est qu'il ne peut presque plus apprendre car la dérivée de sa fonction d'activation est pratiquement nulle:

$$\dot{a} = \frac{da}{dn} = \frac{d}{dn}\left[\frac{1}{1+\exp(-n)}\right] = \frac{(-1)\frac{d}{dn}(1+\exp(-n))}{(1+\exp(-n))^2} = \frac{\exp(-n)}{(1+\exp(-n))^2}$$

$$= a \times \frac{\exp(-n)}{1+\exp(-n)} = a \times \frac{1+\exp(-n)-1}{1+\exp(-n)}$$

$$= a(1-a) \qquad (3.70)$$

Avec $a \approx 1$, on obtient :

$$a \approx 1 \times (1-1) = 0 \qquad (3.71)$$

Or, comme les variations de poids dans l'algorithme de rétropropagation des erreurs, définies aux *équations 3.66 et 3.67*, dépendent linéairement des sensibilités (voir *équations 3.64 et 3.65*) qui elles-mêmes dépendent de la dérivée de la fonction d'activation, on voit immédiatement qu'elles tendent vers zéro lorsque le neurone est saturé et que la convergence, même si elle est toujours possible, requerra beaucoup de périodes d'apprentissage.

Par conséquent, à cause de ce phénomène de saturation, il importe de normaliser les données à l'entrée d'un PMC, c'est-à-dire de les transformer de manière à éviter tout risque de saturation. Une autre façon de procéder est d'initialiser les poids sur la première couche en choisissant un intervalle de valeurs aléatoires ajusté aux stimuli d'apprentissage. Par exemple, pour l'entrée j d'un réseau à R entrées, on pourrait choisir l'intervalle suivant :

$$\left[\frac{1}{-max_q \left|p_j^q\right|}, \frac{1}{max_q \left|p_j^q\right|}\right], \qquad j = 1, \dots, R, \tag{3.72}$$

où $\{q\}$ désigne l'ensemble des stimuli d'apprentissage.

Une autre alternative serait de fixer tous les poids à zéro. Bien que ceci règle certes le problème de la saturation des neurones, ce n'est malheureusement pas une alternative viable. En effet, il se trouve que l'origine de l'espace des poids correspond souvent à un lieu d'instabilité de la fonction d'erreur du réseau. Et ceci peut facilement entraîner la divergence de l'algorithme de rétropropagation.

3.8.5. Groupage :

Au lieu de mettre à jour les poids pour chaque donnée d'entraînement, une alternative consiste à accumuler les variations de poids sur une période d'apprentissage complète et de mettre à jour les poids en une seule fois avec la moyenne de ces variations. On parle alors d'apprentissage «hors-ligne» ou par groupage (*batching*). L'idée est la suivante : l'estimation du gradient qu'engendre chaque donnée d'entraînement est peu précise, la moyenne de ces estimations devrait être plus près du gradient réel. En fait, si les données d'entraînement couvrent adéquatement l'espace des entrées, alors la moyenne de ces estimations sera exacte.
Mais le groupage n'est pas une panacée car cela peut aussi ralentir considérablement la convergence, puisque les poids changent moins souvent. Autrement dit, si l'estimation du gradient basée sur une seule donnée d'entrainement a tendance à être bonne, alors on pourrait converger jusqu'à Q fois plus lentement si l'on procède par groupage. Par contre, lorsque cette estimation est plutôt mauvaise, le groupage sert à éviter de partir dans une mauvaise direction qui, autrement, augmenterait nos chances de rester pris dans un minimum local inadéquat.

3.8.6. Momentum

Une façon d'améliorer l'algorithme de rétropropagation est de rajouter un terme d'inertie dont le rôle est de filtrer les oscillations dans la trajectoire de la descente du gradient :

$$\Delta \mathrm{W}^k(t) = \alpha \Delta \mathrm{W}^k(t-1) - (1-\alpha)\eta s^k (a^{k-1})^T \ , pour\ k = 1, \dots, M, \qquad (3.73)$$

$$\Delta \mathrm{b}^k(t) = \alpha \Delta \mathrm{b}^k(t-1) + (1-\alpha)\eta s^k \ , pour\ k = 1, \dots, M. \qquad (3.74)$$

où $0 \leq \alpha < 1$ s'appelle le momentum. Lorsque $\alpha = 0$, les *équations 3.73 et 3.74* sont équivalentes aux *équations 3.66 et 3.67*, respectivement. Lorsque $\alpha = 1$, les $\Delta \mathrm{W}^k(t)$ et $\Delta \mathrm{b}^k(t)$ ne dépendent plus des équations de rétropropagation des erreurs, mais uniquement des $\Delta \mathrm{W}^k(t-1)$ et $\Delta \mathrm{b}^k(t-1)$, c'est-à-dire des changements de poids à l'étape précédente.

Le terme du momentum produit deux effets distincts selon la situation. Premièrement, lorsque la trajectoire du gradient a tendance à osciller, il contribue à la stabiliser en ralentissant les changements de direction. Par exemple, avec $\alpha = 0.8$, cela correspend d'emblée à ajouter 80% du changement précédent au changement courant. Deuxièmement, lorsque le gradient courant pointe dans la même direction que le gradient précédent, le terme d'inertie contribue à augmenter l'ampleur du pas dans cette direction et donc à accélérer la convergence.

3.8.7. Taux d'apprentissage variable :

Une autre façon d'améliorer la vitesse de convergence pour la rétropropagation des erreurs serait de modifier le taux d'apprentissage dynamiquement tout au long de l'entraînement. Plusieurs approches peuvent être considérées. Par exemple, on peut adopter la stratégie suivante :

1. Si l'erreur quadratique totale, calculée pour toutes les associations de base d'apprentissage, augmente d'une période à l'autre par plus d'un certain pourcentage β (typiquement de 1 à 5%) à la suite d'une mise à jour des poids, alors cette mise à jour doit être abandonnée et le taux d'apprentissage doit être multiplié par un facteur $0 < \rho < 1$, et le momentum doit être fixé à zéro ;
2. Si l'erreur quadratique totale diminue à la suite d'une mise à jour des poids, alors celle-ci est conservée et le taux d'apprentissage est multiplié par un facteur $\gamma > 1$; si le momentum avait précédemment été fixé à zéro, alors on lui redonne sa valeur original ;

3. Si l'erreur quadratique totale augmente par moins de β, alors la mise à jour des poids est acceptée et le taux d'apprentissage reste inchangé ; si le momentum avait précédemment été fixé à zéro, alors on lui redonne sa valeur originale ;

Cette approche suppose que l'apprentissage fonctionne par groupage, c'est-à-dire que les mises à jour des poids sont accumulées sur l'ensemble des associations de la base d'apprentissage et appliquées une fois à la fin de chaque période. Dans certains cas cela peut accélérer grandement la convergence. Dans d'autres cas, cette approche peut aussi nuire à la convergence. Il faut comprendre que ce genre de technique ajoute des paramètres β, ρ et γ qu'il faut fixer apriori. Pour un problème donné, certaines combinaisons de paramètres peuvent être bénéfiques et d'autres non. Parfois, l'emploi d'une telle méthode peut même entraîner une divergence rapide là où la rétropropagation des erreurs avec momentum produisait une convergence lente.

3.8.8. Autre considérations pratiques :

Nous énumérons ci-dessous d'autres considérations pratiques pour l'entraînement du PMC. Selon les circonstances, celles-ci peuvent aussi avoir un effet appréciable sur la performance de l'algorithme de rétropropagation des erreurs.

1. Lorsqu'on utilise une couche de sortie non-linéaire, c'est-à-dire une couche dont les neurones possèdent des fonctions d'activation non linéaire telles que la sigmoïde ou la tangente hyperbolique, il importe de ne pas chercher à saturer les neurones en fixant des sorties désirées qui tendent vers l'asymptote de la fonction. Dans le cas de la sigmoïde, par exemple, au lieu de fixer des sorties désirées à 0 ou à 1, on peut les fixer à 0.05 et 0.95. Ainsi, la rétropropagation des erreurs ne cherchera pas à entrainer les poids dans une direction qui pourrait rendre le neurone incapable de s'adapter.
2. Les sensibilités des neurones sur les dernières couches ont tendance à être plus grandes que sur les premières couches ; le taux d'apprentissage sur ces dernières devrait donc être plus grand que sur ces premières si l'on veut que les différentes couches apprennent approximativement au même rythme.
3. A chaque période d'entrainement, il importe de permuter l'ordre de présentation des stimuli pour réduire la probabilité qu'une séquence de données pathologique nous garde prisonnier d'un piètre minimum local. En effet, la performance de la méthode de la descente du gradient peut dépendre grandement de cet ordre de présentation qui engendre des trajectoires différentes dans l'espace des paramètres, et des trajectoires différentes peuvent nous amener à des minimums locaux différents. Même s'il existe

des séquences pathologiques, le fait de permuter les données à chaque période nous garantit que l'on ne tombera pas systématiquement sur les mêmes.

4. Dans le contexte d'un problème de classification à n classes, on associe généralement un neurone de sortie distinct à chacune d'elles $(S^M = n)$. Ainsi, on interprétera chaque neurone sur la couche de sortie comme indiquant si oui ou non le stimulus d'entrée appartient à la classe correspondante. On construira les vecteur d de sorties désirées avec deux valeurs possibles pour chaque composante : une valeur pour le *oui* et une valeur pour le *non.* Si l'on choisit la fonction d'activation logistique, on pourra coder le *oui* avec une valeur proche de 1 et le *non* avec une valeur proche de 0. En mode de reconnaissance, on pourra classer un stimulus inconnu dans la catégorie associée au neurone ayant produit la sortie maximale.
5. Dans le contexte d'un problème d'approximation de fonction, on choisit généralement des neurones linéaires pour la couche de sortie. Autrement, cela force le réseau à apprendre l'inverse de la fonction d'activation utilisée, en plus de la fonction que l'on veut vraiment qu'il apprenne.
6. Effectuer l'apprentissage d'un réseau quelconque revient à estimer les bonnes valeurs pour chacun de ses poids. Pour pouvoir estimer les paramètres d'un système quelconque possédant un certain nombre de degrés de liberté (paramètres indépendants), il est nécessaire de posséder au moins un nombre équivalent de données. Mais dans la pratique il en faut bien plus ! Une règle heuristique nous indique que pour pouvoir espérer estimer correctement les n poids d'un réseau de neurones, $10n$ données d'entrainement sont requises.
7. La performance d'un réseau lorsqu'elle est évaluée avec ses données d'entraînement est presque toujours *surestimée.* Il faut bien comprendre que le réseau ne comporte aucune intelligence réelle. Il ne fait qu'apprendre les associations qu'on lui fournit. Nous avons discuté le phénomène de *surapprentissage*. nous avons vu qu'une procédure de validation-croisée peut augmenter la capacité de généralisation d'un réseau. Si l'on veut évaluer correctement la performance d'un réseau, il faut le faire avec des données qui n'ont aucunement servi au processus d'apprentissage, ni pour la rétropropagation des erreurs, ni pour la validation croisée. En pratique, ceci implique de diviser les données d'entrainement en trois sous-ensembles distincts : les données d'entrainement, de validation et de test. La proportion relative de ces ensembles peut évidemment varier selon l'application, mais une bonne proportion se situe aux alentours de 50-20-30%, respectivement. [24]

3.9. Avantages et inconvénients des réseaux de neurones

Avantages :

- Les réseaux de neurones sont applicable aux problèmes non-linéaires avec plusieurs variables ;
- La transformation des variables est automatisée dans le processus de computation.
- L'interpolation ; le réseau de neurones est capable de générer des résultats adéquats lorsque on lui présente des entrées ne figurant pas dans les échantillons à condition que ces entrées soient à l'intérieur de la marge d'échantillonnage.
- Capacité de représenter n'importe quelle dépendance fonctionnelle. Le réseau découvre (apprend, modélise) la dépendance lui-même sans avoir besoin qu'on lui "souffle" quoi que ce soit. Pas besoin de postuler un modèle, de l'amender, etc.

Inconvénients :

- Pour minimiser le surapprentissage on a besoin d'un processus intense de calcul.
- Les relations individuelles entre les variable de l'input et les variables de l'output ne sont pas développées par un jugement d'ingénierie c'est-à-dire que le modèle est assimilé à une boite noire, ou tableau (input / output) sans base analytique.
- La taille de l'échantillon doit être large. (plus le nombre d'échantillons est grand, plus la qualité du réseau est bonne)
- Le réseau de neurones ne peut pas extrapoler, c'est-à-dire, il ne peut pas générer des sorties adéquats si on lui présente des entrées ne figurant pas à l'intérieur de la marges des échantillons. [25]

3.10. Conclusion

Les réseaux de neurones à rétropropagation de l'erreur présentent des avenages importants pour la résolution des problèmes de prédiction et commande où le programme doit suivre un modèle donné en présentant des associations entrée/sortie désirée. L'algorithme de rétropropagation minimise l'erreur quadratique moyenne avec la méthode de décente de gradient jusqu'à la convergence.

Dans le chapitre suivant, on va utiliser les réseaux de neurones à rétropropagation de l'erreur dans le dispatching économique en se basant sur les données du programme classique. Avec des associations entrée/ sortie désirée (puissance demandée par les charges/puissance générée par les centrales).

CHAPITRE 04

DISPATCHING ÉCONOMIQUE UTILISANT LES RÉSEAUX DE NEURONES A RÉTROPROPAGATION DE L'ERREUR

I. Programme de dispatching économique de 3 centrales (Skikda, Annaba et Fkirina) alimentant la charge de Constantine.

1. Dispatching économique sans pertes :

Notre réseau à étudier dans la partie de simulation de dispatching économique (sans pertes) est composé de trois unités de production, qui sont les centrales de Skikda, Annaba et Fkirina (Wilaya d'Oum El Bouaghi), alimentant une charge qui est la ville de Constantine (*figure 4.1*).

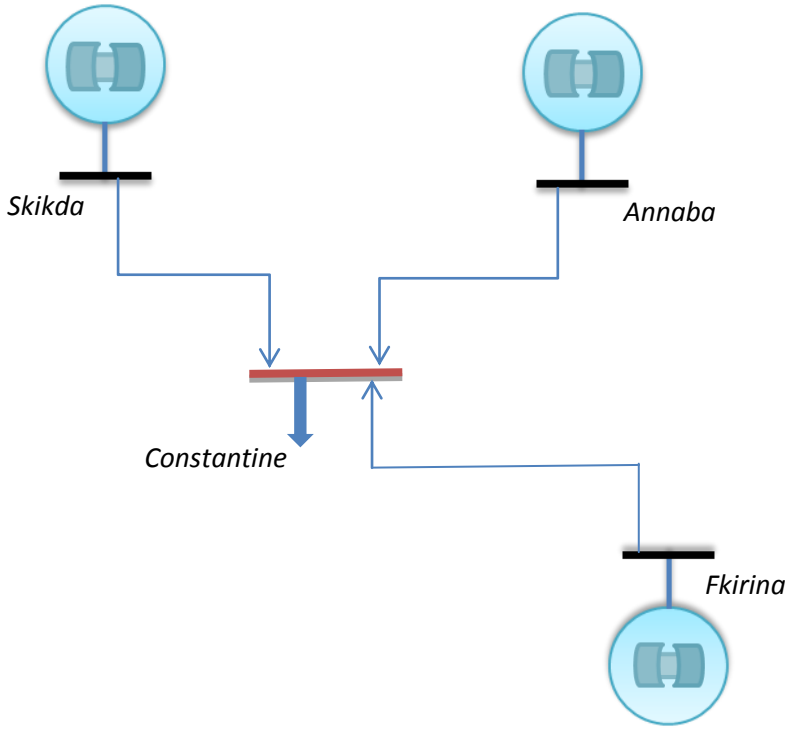

FIG.4.1– *Schéma d'un réseau composé de trois centrales (Annaba, Skikda et Fkirina) alimentant la charge de Constantine*

Comme on a déjà vue dans le *chapitre 02*, Le problème du dispatching économique sans perte est peu complexe car le seul paramètre qui influence le coût est la puissance active générée par la centrale (sans tenir compte de la puissance perdue dans les lignes lors des transits de puissance entre les centrales et les charges).

Les données des centrales sont montrées dans le tableau suivant :

Dénomination	P_{max} *(MW)*	P_{min} *(MW)*	*a (DA/h)*	*b (DA/MWh)*	*c (DA/MW²h)*
Skikda	412	10	110	3490	0,0322
Annaba	280	10	120	4880	0,04
Fkirina	290	10	100	3000	0,0628

TAB.4.1. *Données des trois centrales*

Notre programme Matlab, distribue l'ensemble de la puissance à fournir *par paquet* (de MW) aux différentes centrales. Pour chaque paquet à distribuer, on détermine la centrale qui produirait ce paquet au *plus petit coût*, c'est-à-dire celle qui a le coût incrémental le plus petit, et on la charge de la production de ce paquet. Il faut bien entendu faire attention à ne pas dépasser la puissance maximale que peut fournir une centrale. Dans le cas où une centrale atteint sa puissance maximale, on n'en tient plus compte dans la distribution des paquets restants.

Nous supposons que toutes les centrales sont en fonctionnement. Par conséquent, chaque centrale doit produire au moins sa puissance minimale et il ne reste alors plus que $P_D - \sum_i P_{Min\ i}$ à distribuer aux autres centrales. En procédant ainsi, on trouve une solution approchée de l'optimum recherché.

Notre programme Matlab utilise des paquets de 1MW. Cette distribution donne bien entendu de meilleur résultat que la distribution par paquet de 10MW ou de 5MW. Cependant, nous avons jugé inutile de descendre sous le l MW. Premièrement, parce que nous estimons que les résultats obtenus étaient déjà satisfaisant et ensuite parce qu'il s'agit d'un dispatching sans perte qui est déjà une approximation. Il n'y a donc pas lieu de descendre sous le 1 MW.

Nous avons obtenu les résultats suivantspour une demande de 900 MW (affichage Matlab)

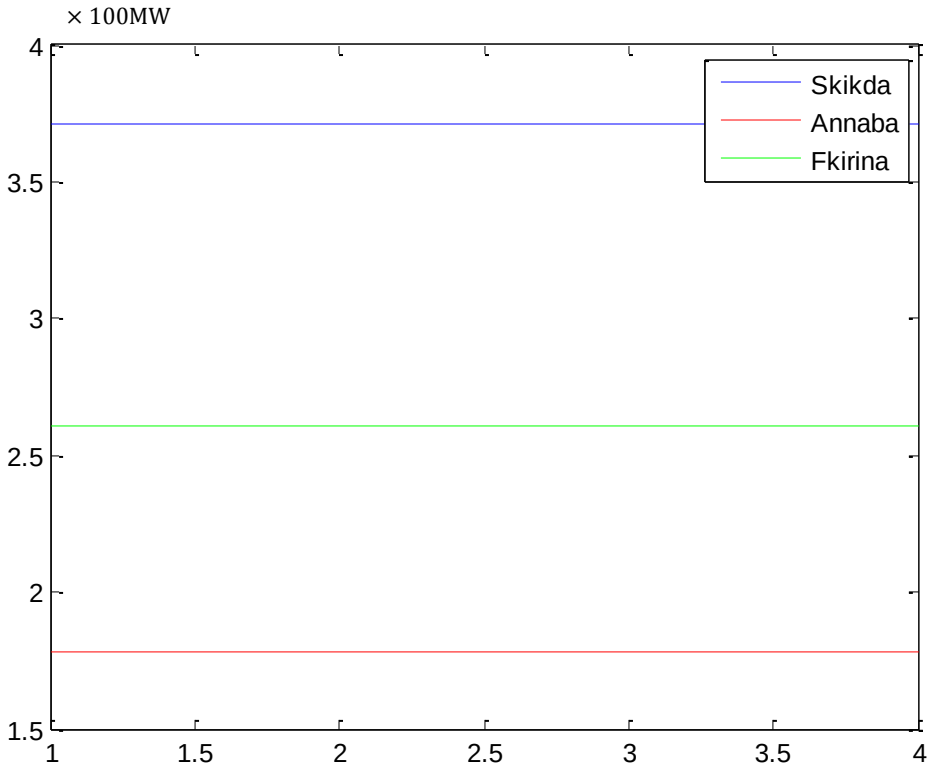

FIG.4.2- *La production des trois centrales*

ans =

La production de SKIKDA est de 412.00000 MW

ans =

La production de ANNABA est de 198.00000 MW

ans =

La production de FKIRINA est de 290.00000 MW

Pgtotale_MW =

900.0000

ans =

Le cout est de 3274450.00000 DA

Remarque 1 :

On voit clairement que la production est distribuée entre les centrales selon le paramètre *b* c'est-à-dire selon le coût de production, donc *Fkirina* et *Skikda* (3000 DA/MWh et 3490 DA/MWh) ont produit toute leur capacité de production 290 MW et 412 MW, la puissance qui reste est produite par Annaba.

Le problème de dispatching économique sans pertes reste strictement théorique, parce qu'il néglige les pertes des lignes issus des impédances linéiques lors de la distribution de l'énergie électrique.

2. Programme de dispatching économique avec pertes :

2.1. Programme classique :

Nous avons adopté la même méthode de distribution par paquet de puissance utilisée pour le dispatching économique sans perte.

Pour l'appliquer au dispatching économique avec perte, il nous faut :

- Calculer les pertes
- Calculer le facteur de pénalité
- Déterminer un critère de convergence

Le schéma bloc de l'algorithme appliqué est donné à la *figure 2.5 (chapitre 2)*. Le calcul des pertes, le facteur de pénalité et le critère de convergence sont illustrés au *chapitre 2*.

Notre réseau à étudier est le même réseau de l'exemple 1 (*figure 4.1*), cette fois, on va prendre en compte les impédances des lignes. On va prendre en moyenne les impédances linéiques $Z = (0.04 + j0.33)\Omega/km, \quad avec \quad V_{base} = 380kV$.

On a les distances *centrale-charge* suivantes : Skikda-Constantine $70\ km$, Annaba – Constantine $120\ km$ et Fkirina-Constantine $140km$.

Utilisant notre programme de dispatching économique avec pertes et introduisant les coefficients de pénalité, on trouve les résultats suivants :

Pdemandée [MW/h]	*P1 [MW/h]*	*P2 [MW/h]*	*P3 [MW/h]*	*Pgénérée [MW/h]*	*Pertes [MW/h]*	*Coût total [DA/h]*
35	10	10	15.0130	35.0130	0.0130	129069.03
50	10	10	30.0422	50.0422	0.0422	174156.74
100	10	10	80.2718	100.2718	0.2718	324845.31
200	10	10	181.3473	201.3473	1.3473	628071.97
300	10	10	283.2592	303.2592	3.2592	933807.55
400	103.8355	10	290	403.8355	3.8355	1281515.89
500	204.4126	10	290	504.4126	4.4126	1632530.13
520	224.5438	10	290	524.5438	4.5438	1702787.92
540	244.6802	10	290	544.6802	4.6802	1773064.01
600	305.1209	10	290	605.1210	5.1210	1984002.16
650	355.5243	10	290	655.5243	5.5243	2159909.84
700	405.9605	10	290	705.9605	5.9605	2335931.97
710	412	14.0106	290	716.0161	6.0161	2376608.61
715	412	19.01826	290	721.0183	6.0183	2401019.09
760	412	64.0669	290	766.0669	6.0669	2620856.51
800	412	104.1545	290	806.1546	6.1546	2816484.27
870	412	174.40895	290	876.4089	6.4089	3159325.65
900	412	204.5574	290	906.5575	6.5575	3306450.387
927	412	231.71144	290	933.7114	6.7114	3438961.83
970	412	274.9965	290	976.9965	6.9965	3650193.09
974	412	279.02555	290	981.0256	7.0256	3669854.68

TAB.4.2. *Dispatching économique local de Constantine (avec programme classique)*

Remarque 2 :

Toutes les puissances sont données par MW/h, ainsi que les pertes et la demande. Le coût total est donné par DA/h. Notre tableau décrit un dispatching statique (à un instant *t* donné). Pour déterminer la consommation journalière de *Constantine*, il suffit de multiplier la puissance générée et le coût total par le nombre des heures correspondant (unit commitment).

Prenant cet exemple :

La consommation de Constantine entre 12^h et $14\ ^h30$ varie selon le tableau 4.3

L'heure	*Pdemandée [MW]*
12h-12h15 (1h/4)	400
12h15-13h (3h/4)	500
13h-13h30 (1h/2)	520
13h30-14h30 (1h)	540

TAB.4.3. *Consommation de Constantine entre* 12^h *et* 14^h30

donc la production de chaque unité de production ainsi que la production totale et le cout total seront prises du *tableau 4.2*et multipliés selon leurs durées :

P1 [MW]	*P2 [MW]*	*P3 [MW]*	*Pgénérée [MW]*	*Pertes [MW]*	*Coût total [DA]*
103.8355*(1/4) =25.958875	10*(1/4) =2.5	290*(1/4) =72.5	403.8355*(1/4) =100.958875	3.8355*(1/4) =0.958875	1281515.89*(1/4) =320378.9725
204.4126*(3/4) =153.30945	10*(3/4) =7.5	290*(3/4) =217.5	504.4126*(3/4) =378.30945	4.4126*(3/4) =3.30945	1632530.13*(3/4) =1224397.598
224.5438*(1/2) =112.2719	10*(1/2) =5	290*(1/2) =145	524.5438*(1/2) =262.2719	4.5438*(1/2) =2.2719	1702787.92*(1/2) =851393.96
244.6802	10	290	544.6802	4.6802	1773064.01
TOTAL (de 12h à 14h30)			1286.220425	11.2204	4169234.5405

TAB.4.4. *Production et cout correspondant à la demande du tableau 4.3*

Remarque 3 :

Le dispatching économique de Constantine est dit *dispatching local*, c'est-à-dire le dispatching connecter directement à la charge (Constantine) ce dispatching local est connecté à d'autres dispatchings (dispatching de Skikda, Fkirina, Annaba, Guelma, Jijel,…etc.) qui sont reliés (gérés) par un dispatching national, ce dernier (le dispatching national) est connecté à d'autres dispatchings nationaux (Tunisie, Maroc, …).

Pour plus de précision dans notre dispatching local de Constantine (pour que notre dispatching local soit relié à d'autres dispatching locaux correctement) il faut que les valeurs de P_{max} des centrales (Skikda, Annaba, Fkirina) soient variables, par exemple : P_{max} de Skikda ne sera plus constante (412MW) elle sera variable selon ce que la ville de Skikda demande ainsi que les environs de Skikda (Dispatching local de Skikda).

2.2. Programme Réseau de neurones associé :

En se basant sur les résultats du dispatching économique classique, et comme nous sommes dans un problème de commande des centrales électriques où le système doit suivre un modèle donné, on va constituer un réseau de neurones de la façon suivante :

P : la matrice « *input* »$[1 \times 21]$ représente les données d'entrée présentée au réseau.

Cette matrice est prise du *tableau 4.2*, représentant les puissances demandées par la charge (Constantine) de 35 à 974 MW avec des intervalles plus au mois aléatoires.

T : la matrice « *target* »$[3 \times 21]$représente les données de la cible (sortie désirée)

Cette matrice est prise du *tableau 4.2*, avec21 échantillons de 3 éléments $[P1\ P2\ P3]$ représentant les puissances générées par les trois centrales suite à chaque demande de la charge (*input*)

On va utiliser un réseau de neurone à *rétropropagation de l'erreur*, à deux couches (*figure 4.3*) avec :

- *L'entrée* ; représentée par la matrice P, matrice de $[1 \times 21]$ éléments (tableau de 21 éléments)
- *Une couche cachée* ; de *21 neurones* avec une fonction d'activation *tansig (Tangente hyperbolique).*
- *Une couche de sortie ; de 3 neurones* avec une fonction d'activation *purelin* (linéaire).

Le réseau converge après 100 itérations.

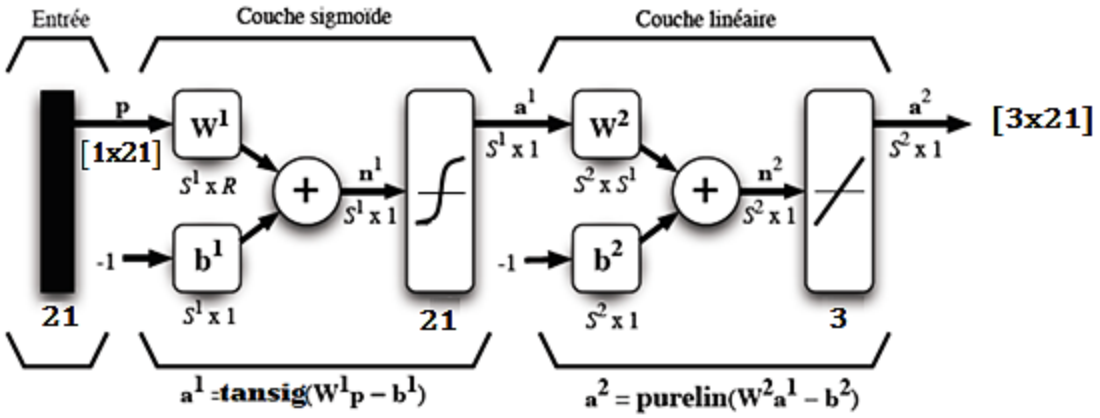

FIG.4.3. *Représentation matricielle du réseau de neurones utilisé*

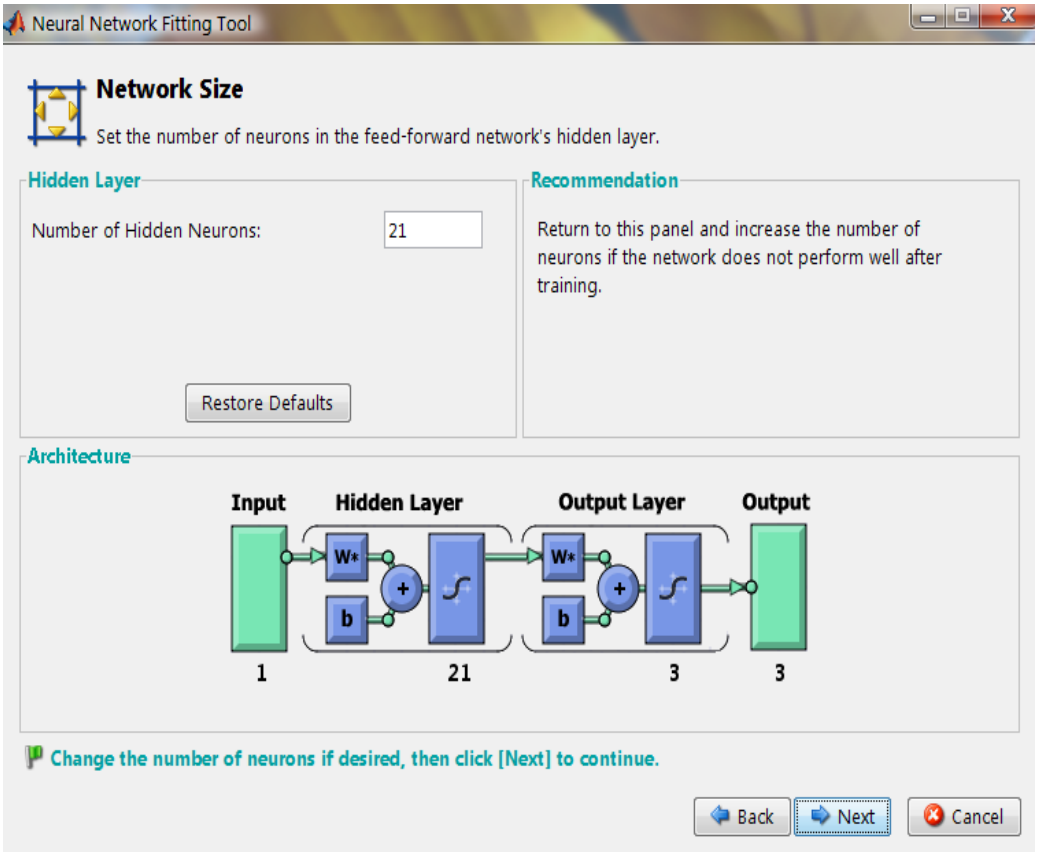

FIG.4.4. *Représentation avec* nftoo*(Matlab 2007)*

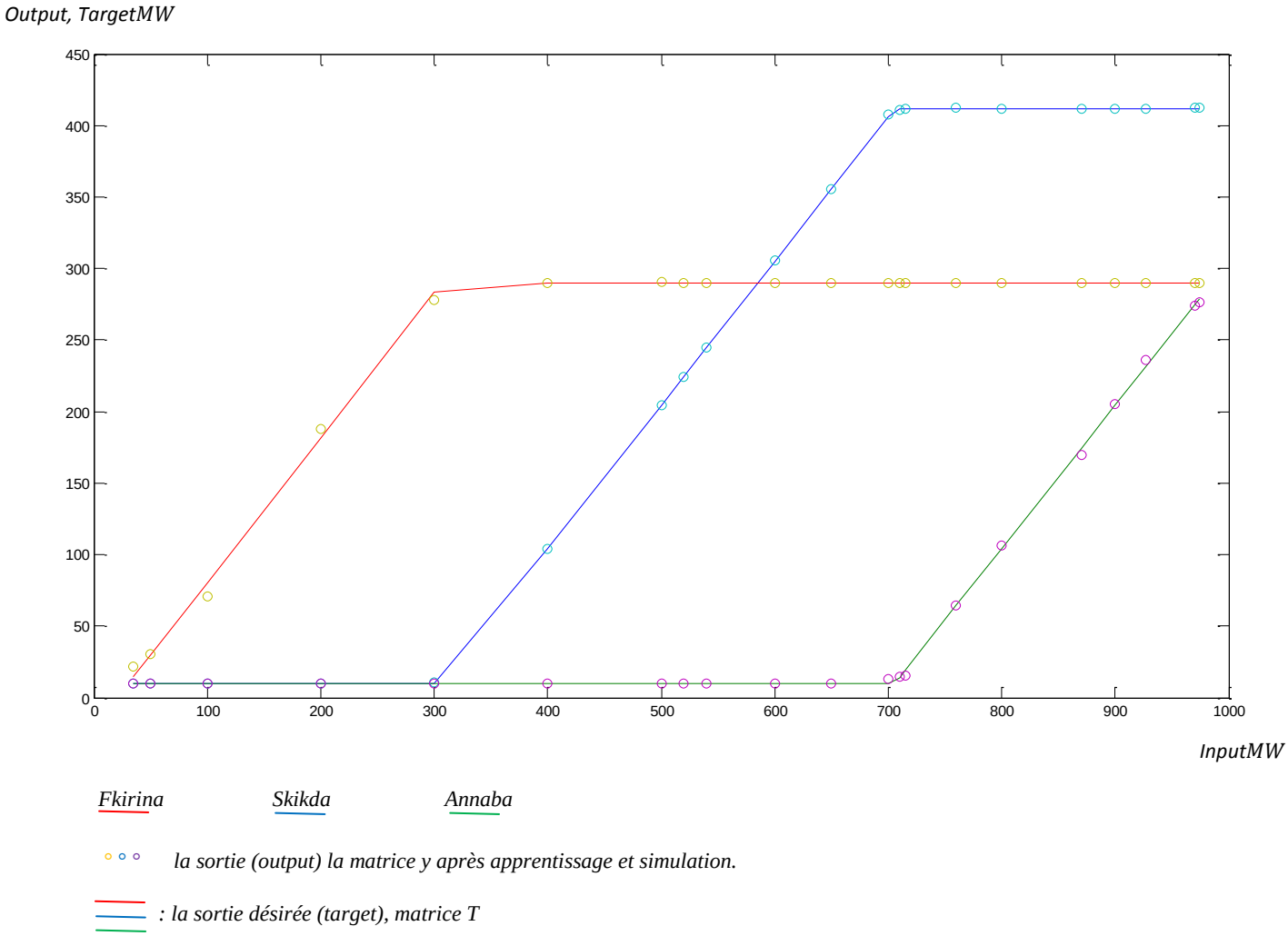

FIG.4.5- *représentation de l'output(Y) et du Target(T) en fonction de l'input(P)*

On remarque clairement que la sortie (*output*) y du modèle de réseau de neurone suit la sortie désirée (target) donc notre réseau a été bien entraîné.

➢ Test de réseau

Pour tester la fiabilité du réseau de neurone, on va donner des valeurs arbitraires ne figurant pas dans les données de l'entraînement et voir si les sorties correspondantes suivent le programme classique ou pas.

Pour cela on introduit une matrice de test P_{test} ce qui nous donne une autre matrice T_{test} (issue du programme classique), après, on fait passer P_{test} à notre réseau neurones ce qui va donner Y_{test}. Finalement on compare T_{test}avec Y_{test}.

P_{test}	T_{test}			
Pdemandée [MW]	*P1*	*P2*	*P3*	*Pgénérée [MW]*
32	10	10	13.01	33.01
70	10	10	50.10	70.10
150	10	10	130.70	150.70
510	214.47	10	290	514.47
515	219.51	10	290	519.51
712	412	16.01	290	718.01
780	412	84	290	786.10
850	412	154.32	290	856.32
1000	437.26	280	290	1007.26

TAB.4.5.*représentation des P_{test} et T_{test} (programme classique)*

P_{test}	Y_{test}			
Pdemandée [MW]	*P1*	*P2*	*P3*	*Pgénérée(RN) [MW]*
32	12.8390	9.9923	11.3349	34.16
70	10.44	6.80	80.43	97.69
150	10.41	19.78	80.53	110.72
510	214.47	9.93	290.04	514.44
515	219.46	10.05	290	519.53
712	411.92	16.15	290	718.07
780	412.58	82.57	289.99	785.15
850	412.32	154.18	289.99	856.51
1000	411.67	302.82	289.99	1004.5

TAB.4.6.*représentation des P_{test} et Y_{test} (programme RN)*

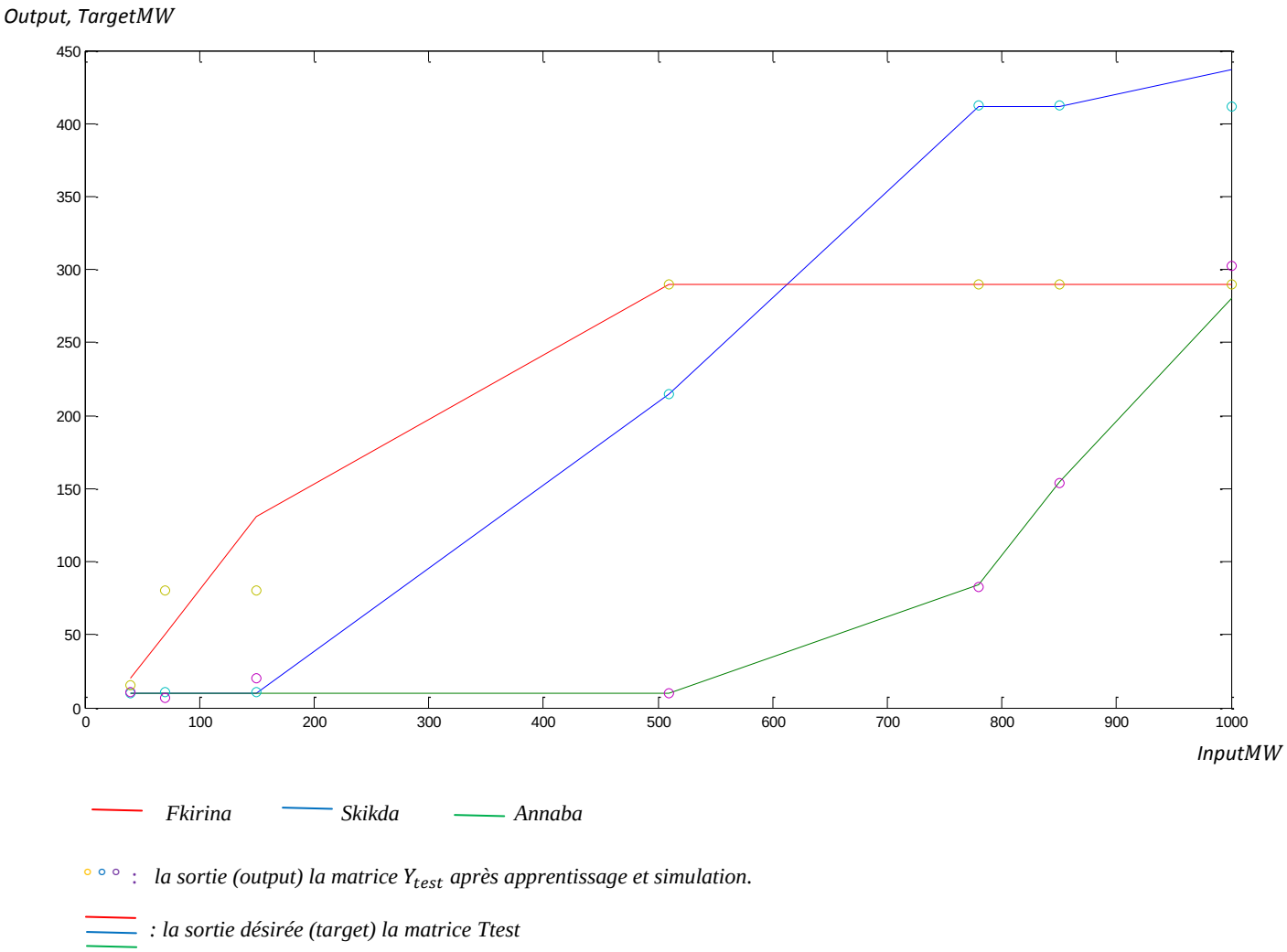

FIG.4.6- *Représentation de Y_{test} et du T_{test} en fonction de P_{test}*

après le test du réseau (*tableau4.5 et 4.6*) on a constaté que :

1- les valeurs de Y_{test} correspondant aux demandes 510MW, 515MW et 712MW, sont très proches des valeurs générées par le programme classique (514,44 - 514,47) (519.53 - 519.51) et (718.07 - 718.01) respectivement. Cela est dû au pas d'apprentissage petit (relativement) l'ors de l'apprentissage dans la matrice P, (500<510<520) et (500<515<520) (710<712<715).

2- les valeurs de Y_{test} correspondant aux demandes 70MW et 150MW, sont trop éloignées des valeurs générées par le programme classique (97,69 - 70,10) et (110,72 - 150,70) respectivement. Cela est dû au pas d'apprentissage grand l'ors de l'apprentissage dans la matrice P, (50<70<100) et (100<150<200) donc le réseau de neurones a du mal à générer des sorties proches de celles du programme classique.

3- les valeurs de Y_{test} correspondant aux demandes 32MW et 1000MW sont aussi loin de celles générées par le programme classique, (34.1661-33.0105) et (1004.5 - 1007.26).[10]

On rappelle que 32MW et 1000MW se trouvent à l'extérieur du vecteur P (vecteur des entrées de l'apprentissage) $P = [\,35, \dots, 974]$ ce qui explique l'incapacité du réseau de neurones à générer des sorties adéquates.

De cet exemple on déduit trois caractéristiques essentielles des réseaux de neurones:

1- **l'interpolation**: les réseaux de neurones (multicouches) ont la capacité d'interpolation, c'est à dire de prévoir (*prédire*, *deviner*) des sorties suites à des excitations (entrées) ne figurant pas dans l'apprentissage mais se situent dans l'intervalle d'apprentissage. [26]
2- **l'extrapolation**: les réseaux de neurones n'ont pas la capacité de l'extrapolation, c'est-à-dire, ils ne sont pas capable de générer des sorties adéquates suite à des excitations figurant hors de l'intervalle de l'apprentissage. [27]
3- **le pas d'apprentissage** (la différence entre les éléments de la matrice P) est un élément très essentiel dans la fiabilité (la convergence) du réseau de neurones, plus le pas est petit plus la fiabilité du réseau est bonne (plus le réseau est capable de générer des sorties trop proches de celles des sorties réelles) cependant, on doit garder un compromis *convergence/temps d'exécution*, parce qu'un pas trop petit mène vers une convergence lente. [28]

[10] *La valeur de la demande Pd=1000MW ne peut pas être obtenue physiquement, parce qu'elle dépasse la capacité de génération de la somme des trois centrales. On l'avait utilisé juste pour l'illustration du concept de l'extrapolation.*

II. Programme de dispatching économique d'un réseau maillé de 9 jeux de barres :

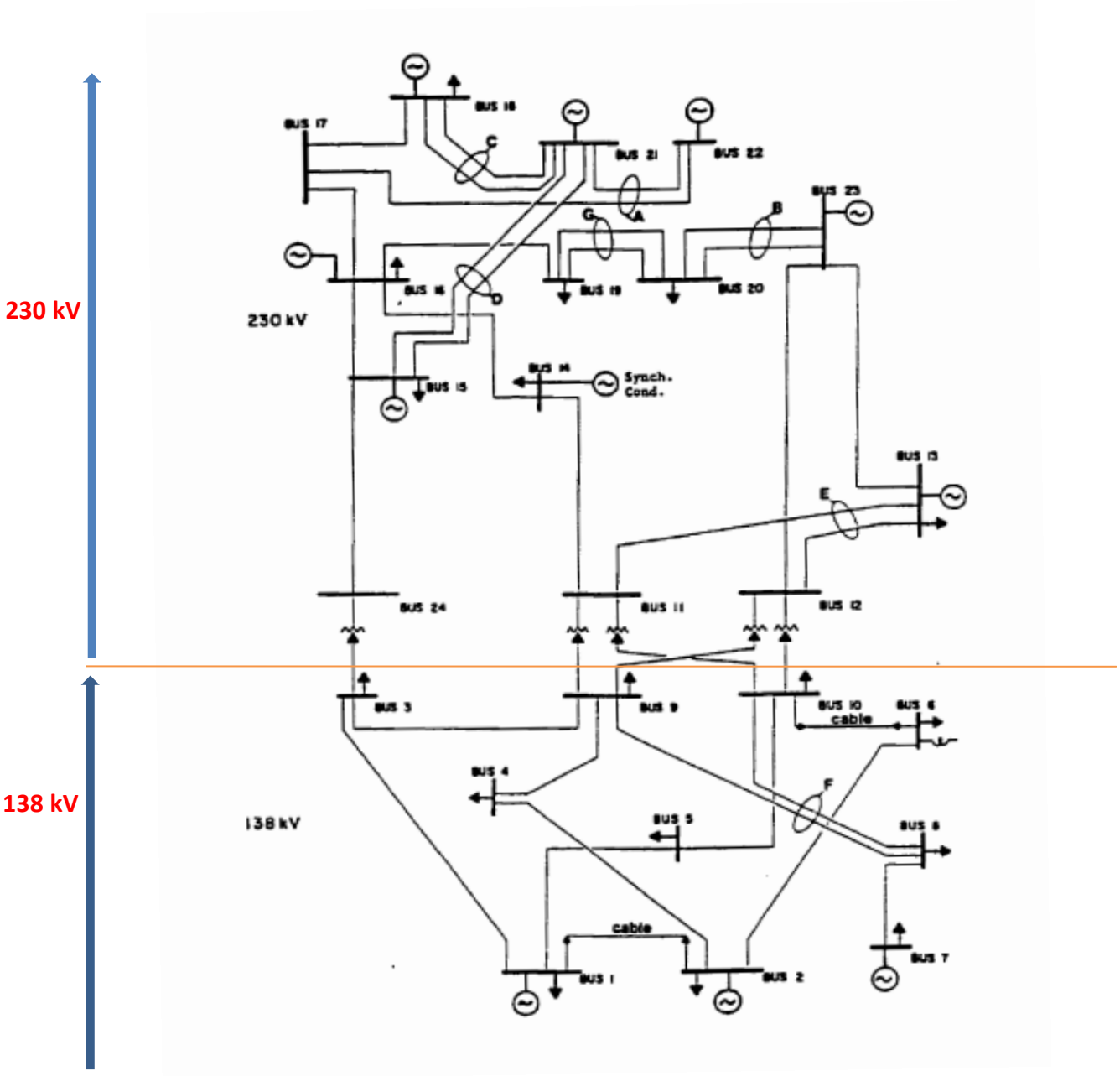

FIG.4.7. IEEE 24-Bus Test network. [29]

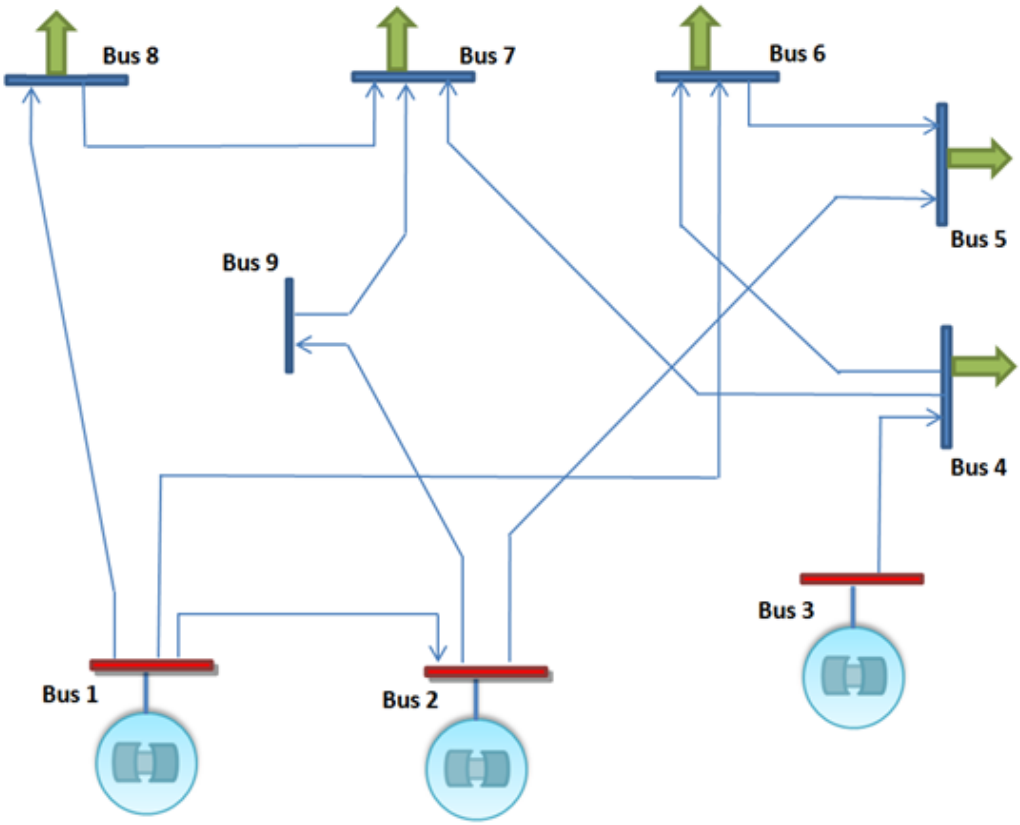

FIG.4.8. *La zone 138KV du réseau IEEE 24-Bus Test Network*

Dans cet exemple on va étudier et programmer le dispatching économique d'un réseau maillé avec le programme classique et avec le programme de réseau de neurones.

Le réseau à étudier constitue la zone *138KV* dans le *IEEE 24-Bus Test Network* (figure 4.8).

Le réseau se constitue de 9 Jeux de barres (*Busbar ou Bus*) qui se répartissent en trois catégories :

1- Trois jeux de barres de génération (*Generation buses*) (*Bus 1, Bus 2 et Bus 3*) qui sont des jeux de barres connectés directement aux sources d'énergie électrique (générateurs).
2- Cinq jeux de barres de charge (*Load buses*) (*Bus 4, Bus 5, Bus 6, Bus 7 et Bus 8*) qui constituent des jeux de barres connectés à des charges.
3- Un jeu de barres de régulation (Slack bus) (Bus 9). [29]

Premièrement, on donne les caractéristiques des générateurs ;

Dénomination	*Pmax [MW]*	*Pmin [MW]*	*a [$/h]*	*b [$/MWh]*	*c [$/MW²h]*
G1(Bus 1)	308	0	646.99	19.18	0.0322
G2(Bus 2)	350	0	646.99	19.18	0.0322
G3(Bus 3)	250	0	1829.71	27.22	0.0628

TAB.4.7. *Caractéristique des générateurs* [29]

On prend $V_{base} = 138$ kV

Deuxièmement, on donne les distances et les impédances linéiques entre jeux de barres :

Ligne	*de*	*à*	*Distance en Miles*	*R[Ω]*	*X[Ω]*
1	1	8	55	0.055	0.21
2	1	6	45	0.02	0.08
3	1	2	3	0.003	0.014
4	2	9	33	0.003	0.127
5	2	5	50	0.05	0.192
6	3	4	16	0.016	0.06
7	4	7	43	0.043	0.165
8	4	6	43	0.043	0.165
9	5	6	16	0.014	0.061
10	7	8	31	0.031	0.119
11	9	7	27	0.027	0.104

TAB.4.8.*Distances et impédances linéiques entre jeux de barres* [29]

après, on présente les demandes des jeux de barres de charge (B4, B5, B6, B7 et B8), cette demande constitue une série incrémentale[11] variant selon l'importance de la charge.

DEMANDE [MW]					*DEMANDE TOTALE*
B4	*B5*	*B6*	*B7*	*B8*	
0	46	1	15	90	152
0	56	10	25	100	191
0	66	20	35	110	231
0	76	30	45	120	271
0	86	40	55	130	311
0	96	50	65	140	351
0	106	60	75	150	391
10	116	70	85	160	441
20	126	80	95	170	491
30	136	90	105	180	541
40	146	100	115	190	591
50	156	110	125	200	641
60	166	120	135	210	691
70	176	130	145	220	741
80	186	140	155	230	791
90	196	150	165	240	841
100	206	160	175	250	891

TAB.4.9. *les demandes des jeux de barres de charge du réseau*

[11] *Il faut noter que ces demandes sont présentées avec un certain model qui garde toujours la différence entre les demandes constante.*
Ce n'est pas obligatoire de prendre les données dans cet ordre, on pourrait les maitre aléatoirement, mais les statistiques montrent que des barres consomment plus que d'autres ce qui va nous aider à diminuer les dimensions de la matrice P.

1. Le programme classique

Pour le calcul de la matrice des conductances de ligne (G), chaque tronçon est associé à sa conductance (impédance) donc on fait la conductance équivalente selon les chemins possibles de l'énergie, des générateurs vers les barres concernés.

Le programme classique (dispatching avec pertes) donne les résultats suivants :

DEMANDE TOTALE [MW]	*GENERATION [MW]*			*PERTES TOTALES [MW]*	*GENERATION TOTALE [MW]*	*COUT TOTAL*
	B1	*B2*	*B3*			
152	153.0530	0	0	1.053	153.0530	17231 $
191	192.3522	0	0	1.3522	192.3522	18126$
231	232.7053	0	0	1.7053	232.7053	19049$
271	273.1125	0	0	2.1125	273.1125	20000$
311	307.2077	6.2913	0	2.499	313.499	20944$
351	307.6334	46.1756	0	2.809	353.809	21829 $
391	288.1039	106.9353	0	4.0393	395.0393	23029 $
441	307.4126	117.1223	21.0978	4.6327	445.6327	24682 $
491	307.91	127.3267	61.0059	5.2426	496.2426	26978$
541	307.4443	137.5487	102.1364	6.1294	547.1294	29472$
591	307.2718	150.4827	140.4487	7.2032	598.2032	31923 $
641	307.7450	180.5522	160.5516	7.8488	648.8488	33167 $
691	307.2442	211.7153	180.6662	8.6258	699.6258	34463 $
741	307.7698	241.9925	200.7925	9.5548	750.5548	35873 $
791	307.3220	273.3763	220.9304	10.629	801.6286	37341$
841	307.9009	303.8653	241.08	11.846	852.8462	38918$
891	307.906	349.977	244.867	11.751	902.7514	39430$

TAB.4.10. *La production des générateurs suite aux demandes des barres de charge*

2. Programme Réseau de neurones associé :

En se basant sur les résultats du dispatching économique classique, on va constituer un réseau de neurone de la façon suivante :

P : la matrice *input* [5 × 17] représente les données d'entrée présentée au réseau.

Cette matrice est prise du tableau 4.9, représentant les puissances demandées par les jeux de barres de charge.

T : la matrice *target* [3 × 17] représente les données de la cible (sortie désirée)

Cette matrice est prise du tableau 4.10, représentant les puissances générées par les trois centrales suite à chaque demande de la charge.

On va utiliser un réseau de neurone à rétropropagation de l'erreur, à deux couches, (*figure 4.9*):

- *L'entrée*, représentée par la matrice P, matrice de [5 × 17] éléments.
- *Une couche cachée*, de *10 neurones* avec une fonction d'activation *tansig* (Tangente hyperbolique).
- *Une couche de sortie, de 5 neurones* avec une fonction d'activation *purelin* (linéaire).

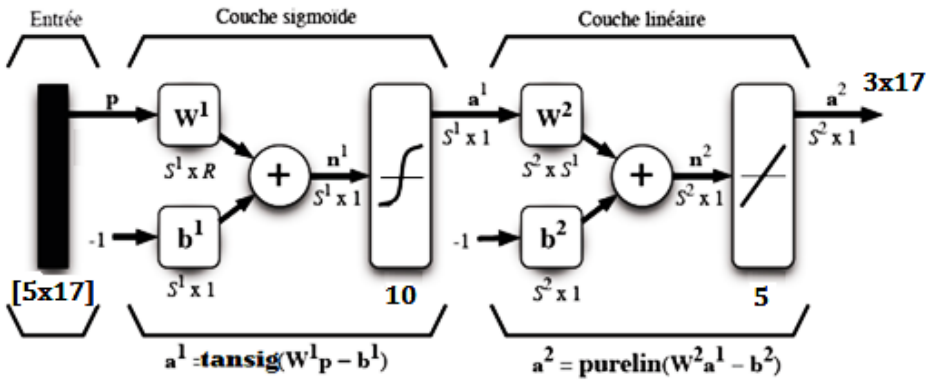

FIG.4.9. *Représentation matricielle du réseau de neurone utilisé*

Le réseau converge après 50 itérations (figure 4.10)

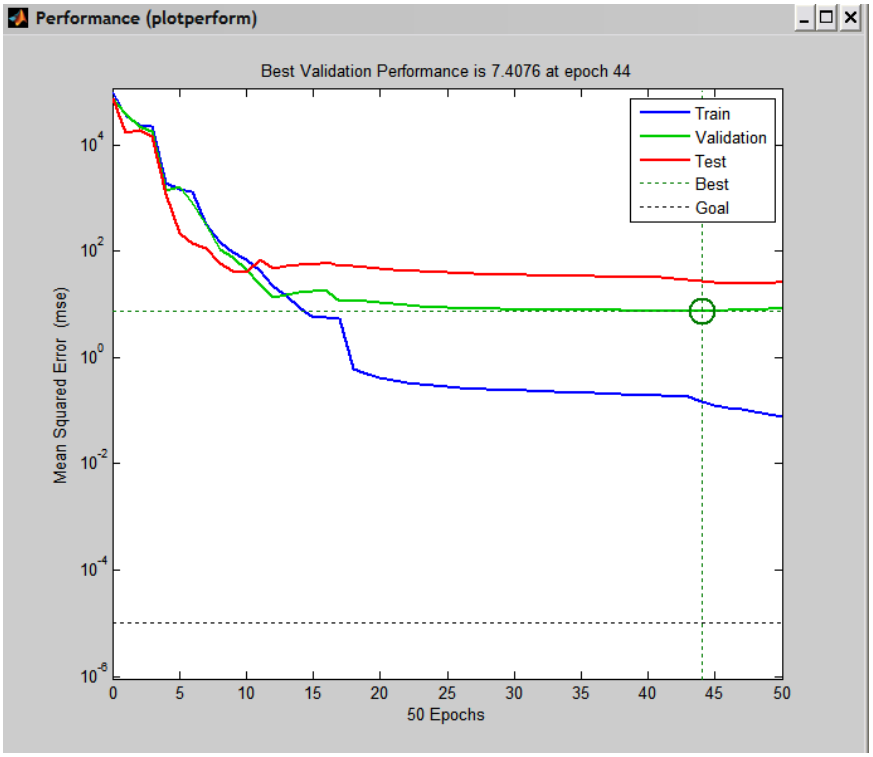

FIG.4.10. *Représentation de l'évolution de l'apprentissage, la validation et le test en fonction des 50 itérations (la descente du gradient)*

Le résultat de l'apprentissage est représenté au tableau suivant :

Demande totale [MW]	*G1*	*G2*	*G3*
151	153.0660	0.0388	0.0229
191	189.2874	3.3237	0.6751
231	232.7109	0.0452	0.0291
271	273.1185	0.0398	0.0230
311	300.9937	12.9793	0.0916
351	307.6233	46.1608	0.0175
391	288.0940	106.9250	0.0109
441	302.5176	115.3004	27.1641
491	307.9063	127.3257	61.0087

541	306.6080	125.9764	114.7316
591	307.2240	150.5320	140.4603
641	307.6713	180.8694	160.3880
691	307.5934	211.5410	180.5317
741	307.3244	242.3096	200.9332
791	307.5313	273.2441	220.8511
841	309.8659	304.3482	238.5474
891	308.067	346.016	244.817

TAB.4.11. *Les sorties du réseau de neurones suite aux entrées*

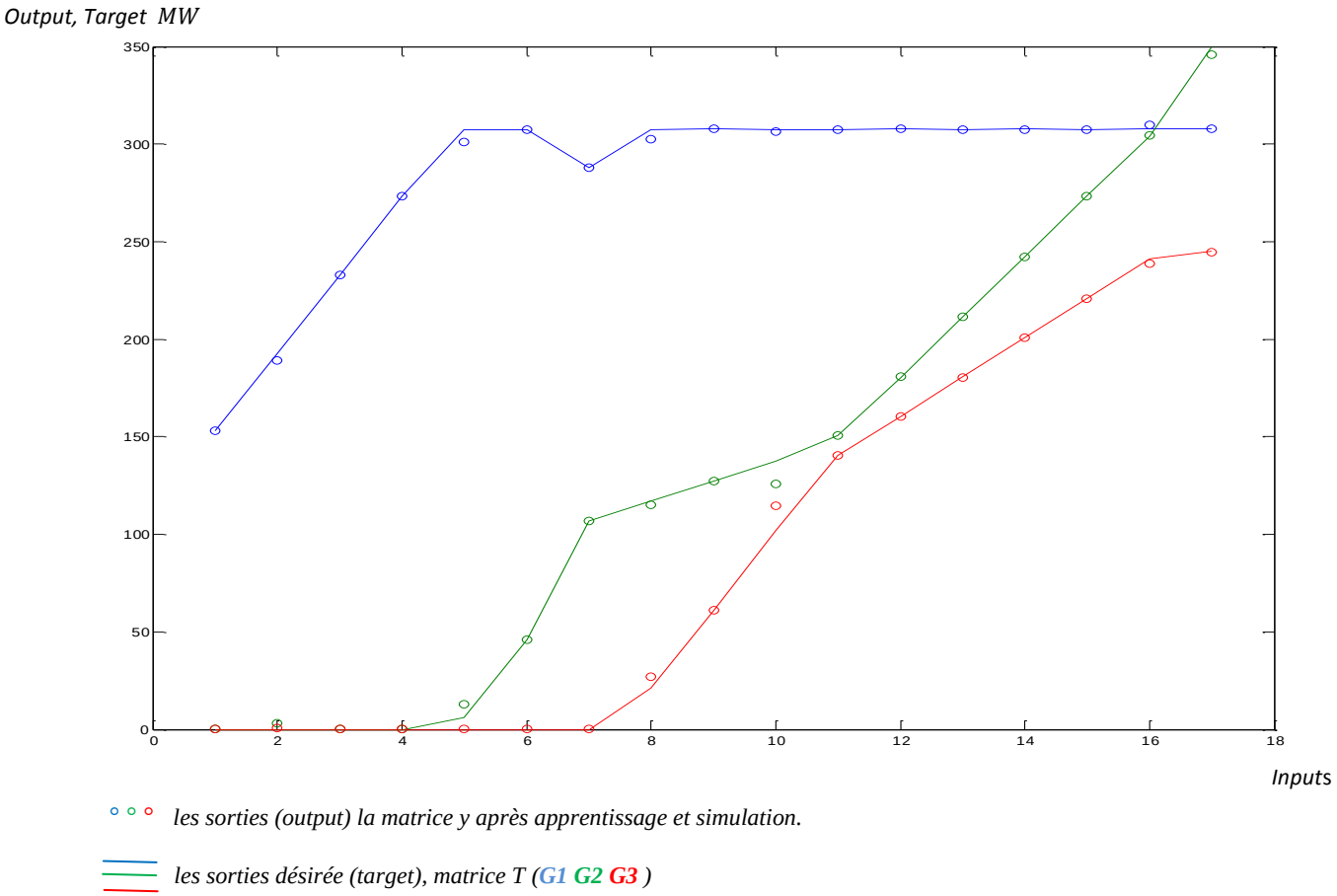

FIG.4.11-*Comparaison entre l'output(Y) et le Target(T)*

on voit du tableau 4.11 et de la figure 4.11 que les valeurs des sorties de la matrice *Y* (output) sont proches, dans la plupart des points, aux valeurs désirées de la matrice T (target).

On peut déduire que l'entrainement de notre réseau de neurones est *estimé bon*. Il faut donc tester le réseau pour juger la fiabilité de l'apprentissage.

➢ Test de réseau

On introduit la matrice P_{test} qui contient des valeurs situées dans la marge de la matrice d'entrainement P mais qui n'ont pas participé à l'apprentissage.

P_{test} [MW]				
0	51	5	20	95
0	61	15	30	105
0	70	24	39	114
0	75	29	44	119
0	77	31	46	121
11	117	71	86	161
25	131	85	100	175
89	195	149	164	239

TAB.4.12.*Les demandes des barres pour le test du réseau de neurones*

On fait passer P_{test} dans le programme classique, ce qui va nous donner la matrice T_{test}*(tableau 4.13)*

T_{test}[MW]				
P1	P2	P3	Pertes [MW]	Pgénérée [MW]
172.1958	0	0	1.195	172.1958
212.5220	0	0	1.522	212.5220
248.8617	0	0	1.8617	248.8617
269.0694	0	0	2.0694	269.0694
277.1563	0	0	2.1563	277.1563
307.4607	118.1420	25.0791	4.6818	450.6818
307.1725	132.4355	82.0445	5.6525	521.6525
307.8418	300.8117	239.0645	11.718	847.7180

TAB.4.13. *Les valeurs des générations des centrales suite aux demandes de test utilisant le programme classique*

on fait passerP_{test}dans le programme du réseau de neurones, ce qui va nous donner la matrice Y_{test}*(tableau4.14)*

Y_{test} [MW]			
P1	P2	P3	Pgénérée [MW]
169.7342	2.9799	0.0033	172.7174
210.7224	1.8963	0.0038	212.6225
249.7927	0.0000	0.0901	249.8828
269.4901	0.0095	0.0349	269.5345
276.6220	0.5433	0.0141	277.1794
303.4446	116.9157	29.7223	450.0826
307.2723	124.3478	90.1863	521.8064
307.7661	300.8741	239.6219	848.2621

TAB.4.14-*Les valeurs des générations des centrales suite aux demandes de test utilisant les réseaux de neurones*

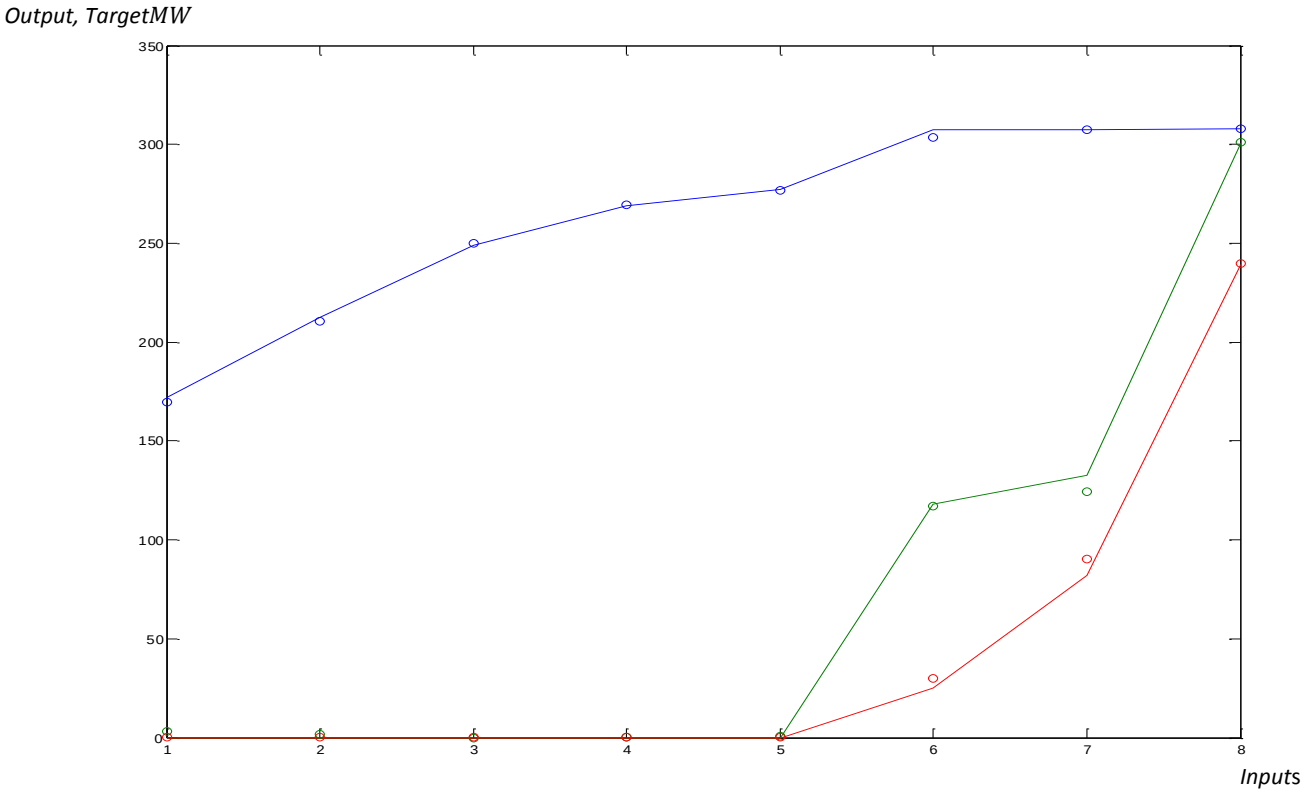

FIG.4.12-*Comparaison entre* Y_{test} *et* T_{test}

En se basant sur les données du tableau 4.13, tableau 4.14 et de la figure 4.12, on peut reconfirmer que l'apprentissage de notre réseau de neurones était bon.

Pour que l'apprentissage soit très bon voire excellent, il faut que les sorties de la matrice Y soient parfaitement identiques à la cible($Y \equiv T$), pour cela il faut :

- Utiliser un bon programme classique de dispatching économique, basé sur des données et paramètres précis des centrales, des impédances linéiques et de la topographie du réseau électrique ;
- Simuler le plus grand nombre de cas en se basant sur des données statistiques réelles des demandes d'énergie des jeux de barres de charge (Generation buses) ;
- Simuler des cas aléatoires et des cas imprévisibles (cas de défauts) ne figurant pas dans les données statistiques ;
- Diminuer le pas d'apprentissage, pour que le réseau de neurone fasse une interpolation appréciable pour les données inconnues (imprévisible) ;
- Ajouter le plus grand nombre de données, c'est-à-dire, augmenter les dimensions de la matrice P et T, donc faire un apprentissage plus détaillé qui couvre la majorité des cas issus du programme classique. [12]

Rappelons que le programme de réseau de neurones utilise ses données à partir du programme classique, donc il ne pourra pas le remplacer qu'après que le programme classique donne toutes les données nécessaires à l'apprentissage.

Dans un contexte pratique, ou, pour réaliser un logiciel de dispatching économique des réseaux électriques à base des réseaux de neurones, il faut :

Après un apprentissage complet et un test juger très bon voire excellent, exécuter (commander) le dispatching économique directement par les réseaux de neurones. C'est-à-dire que les données des demandes des charges doivent se présenter, en temps réelle, au réseau de neurones directement pour la commande, où, la décision se manifeste en deux cas :

1- Si le réseau de neurones trouve ces données (temps réel) dans sa matrice P, il donne immédiatement les sorties prédéfinies dans la matrice T.
2- Si le réseau de neurone ne trouve pas les données dans la matrice P (donc à l'intérieur de l'intervalle des éléments de P), il fait une interpolation pour faire sortir les sorties

[12] *Cela dépend essentiellement de la fréquence (vitesse d'exécution) du processeur utilisé, des matrice P et T très longues demandent un processeur plus puissant sinon on risque d'avoir un programme lent.*

(la matrice Y), cette interpolation est liée directement à l'apprentissage c'est-à-dire, plus le pas d'apprentissage est petit et les dimensions de P sont grandes, plus l'interpolation est bonne.

Conclusion Générale

Pourquoi utiliser les réseaux de neurones dans le dispatching ?

Le lecteur peut se poser cette question : pourquoi utilise-t-on les réseaux de neurones tant que le programme classique peut faire le dispatching économique tout seul, et tant que le programme de neurones est basé essentiellement sur le programme classique ?

L'élément le plus essentiel et la cause de l'utilisation des réseaux de neurones est ***le temps d'exécution rapide*** par rapport au programme classique.

Le programme classique est proportionnellement lent, parce que :

- il utilise des boucles itératives (if, *while et for*) ce qui nuit directement à la rapidité d'exécution ;
- il stock le plus grand nombre de paramètres : les paramètres des centrales, les paramètres de chaque ligne et les paramètres de chaque jeu de barres …etc.
- plus le réseau est maillé plus le programme est lent.

En revanche, pour le programme de réseaux de neurones, le temps d'exécution est très rapide de l'ordre de la milliseconde, parce que :

- il contient des boucles uniquement dans l'apprentissage, après, il rend le problème de dispatching un problème de classification ;
- c'est un exécuteur, il exécute les données déjà stockées, et interpoles les données intermédiaires.

En plus, dans la pratique, on opte pour un dispatching avec la plus rapide fréquence possible (5 ou 15 minutes plutôt que chaque heure [30]), voire en *temps réel* [31], ce qui est pratiquement impossible avec le programme classique de dispatching économique qui utilise plus d'une seconde pour le traitement, or, les données changent chaque seconde.

Pour la commande des réseaux électriques, le facteur *temps* est très essentiel et primordial, notamment dans le dispatching économique et dans bien d'autres disciplines comme : la protection, la stabilité, l'écoulement des puissances, …etc. Pour cela l'utilisation des techniques de l'intelligence artificielle et plus précisément les réseaux de neurones dans la commande des réseaux électriques réduit le temps d'exécution de quelques minutes à

quelques secondes (millisecondes), cela va apporter un gain économique énorme, par l'augmentation de la fiabilité et la réduction des pertes et des défauts, donc la réduction de la consommation des combustibles (charbons, pétrole, gaz, uranium…etc.).

La diminution de la production implique une contribution à la préservation de l'environnement par la réduction de la pollution et de l'effet de serre (CO_2 issu des centrales thermiques, déchets nucléaires issus des centrales nucléaires … etc.).

Appendice A

Les domaines de l'intelligence artificielle

- Les **systèmes experts** : un système expert est un logiciel capable de simuler le comportement d'un expert humain effectuant une tâche précise. Il s'agit là d'un domaine où le succès de l'intelligence artificielle est incontestable et cela est sans doute dû au caractère très ciblé de l'activité que l'on demande de simuler.
- Le **calcul formel** (opposé au calcul numérique) traite des expressions symboliques. Par exemple, calculer la valeur d'une fonction réelle en un point est du calcul numérique alors que calculer la dérivée d'une fonction numérique est du calcul formel.
- La **représentation des connaissances** : si l'on veut qu'un logiciel soit capable de manipuler des connaissances, il faut savoir les représenter symboliquement. C'est là un des secteurs les plus importants de la recherche en intelligence artificielle.
- La **simulation du raisonnement humain** : Les hommes sont capables de raisonner sur des systèmes incomplets, incertains et même contradictoires. On tente de mettre au point des logiques qui formalisent de tels modes de raisonnement (logiques modales, temporelles, floue, non monotones, etc.).
- Le **traitement du langage naturel** : Qu'il s'agisse de traduire un texte dans une autre langue ou de le résumer, le problème crucial à résoudre est celui de sa compréhension. On pourra dire qu'un logiciel comprend un texte lorsqu'il pourra le représenter sous une forme indépendante de la langue dans laquelle il est écrit : c'est une tâche très difficile mais, des progrès significatifs ont d'ores et déjà été réalisés.
- La **résolution de problèmes** : représentation, analyse et résolution de problèmes concrets. Les jeux fournissent une bonne illustration de ce domaine.
- la **reconnaissance de la parole** : une technique informatique qui permet d'analyser un mot ou une phrase captée au moyen d'un microphone pour la transcrire sous la forme d'un texte exploitable par une machine. La reconnaissance vocale, ainsi que la synthèse vocale, l'identification du locuteur ou la vérification du locuteur, font partie des techniques de traitement de la parole. Ces techniques permettent notamment de réaliser des interfaces vocales c'est-à-dire des interfaces homme-machine (IHM) où une partie de l'interaction se fait à la voix. Parmi les nombreuses applications, on peut citer les applications de dictée vocale sur PC où la difficulté tient à la taille du

vocabulaire et à la longueur des phrases, mais aussi les applications téléphoniques de type serveur vocal, où la difficulté tient plutôt à la nécessité de reconnaître n'importe quelle voix dans des conditions acoustiques variables et souvent bruyantes.

- La **reconnaissance de l'écriture** : un traitement informatique qui a pour but de traduire un texte écrit en un texte codé numériquement. La reconnaissance de l'écriture manuscrite fait appel à la reconnaissance de forme, mais également au traitement automatique du langage naturel. Cela veut dire que le système, tout comme le cerveau humain, reconnaît des mots et des phrases existant dans un langage connu plutôt qu'une succession de caractères. Ceci améliore grandement la robustesse.
- La **reconnaissance des visages** : longtemps considéré comme un des problèmes les plus difficiles de l'intelligence artificielle, il semble que l'on obtienne des résultats intéressants en utilisant des réseaux neuronaux.
- la **robotique** : Il y a déjà longtemps que des robots industriels ont fait leur apparition dans les usines. On appelle robot de la première génération, ceux qui sont capables d'exécuter une série de mouvements préenregistrés. Un robot de la deuxième génération est doté de moyens de perception visuelle lui permettant de prendre certaines décisions. Un robot de la troisième génération, objet des recherches actuelles, doit acquérir une plus grande autonomie comme se déplacer dans un environnement inconnu.
- l'**apprentissage** : Un logiciel devrait avoir des capacités d'apprentissage autonome pour pouvoir être véritablement qualifié d'intelligent. Donc il faut construire une gigantesque base de données censée contenir toute la pragmatique, c'est-à-dire toutes les connaissances souvent implicites partagées par les humains d'un même groupe indispensables à la communication. Il est impensable de saisir manuellement toutes ces informations donc adjoindre un module d'apprentissage à sa base de données lui permettant de travailler seule, c'est-à-dire de collecter des informations nouvelles dans des textes ou par discussion avec des humains et de réorganiser seule l'architecture de ses connaissances.
- les **systèmes complexes adaptatifs** : on regroupe sous ce vocable les algorithmes génétiques et les modèles de vie artificielle. Il s'agit là, énoncé de manière abusivement succincte, d'étudier comment des populations soumises à des lois simples et naturelles convergent naturellement vers des formes organises. [32]

Appendice B

La règle LMS (règle d'apprentissage d'un réseau supervisé monocouche)

A la *section 3.6*, nous avons traité le cas du perceptron simple où les neurones utilisent une fonction de transfert de type *seuil*. Nous allons maintenant considérer la même architecture de réseau à une seule couche mais avec cette fois-ci une fonction de transfert *linéaire* comme à la *figure B.1*. Ce réseau s'appelle «ADALINE» (en anglais «ADAptive LInear NEuron») à cause de sa fonction de transfert linéaire. Il souffre des mêmes limitations que le perceptron simple : il ne peut résoudre que des problèmes linéairement séparables. Cependant, son algorithme d'apprentissage, la règle du «Least Mean Square», est beaucoup plus puissante que la règle du perceptron original, car bien que cette dernière soit assurée de converger vers une solution, si celle-ci existe, le réseau résultant est parfois sensible au bruit[13] puisque la frontière de décision se retrouve souvent trop proche des patrons d'apprentissage (l'algorithme s'arrête dès que tous les patrons sont biens classés). En revanche, la règle LMS *minimise l'erreur quadratique moyenne*, de sorte que la frontière de décision a tendance à se retrouver aussi loin que possible des prototypes.

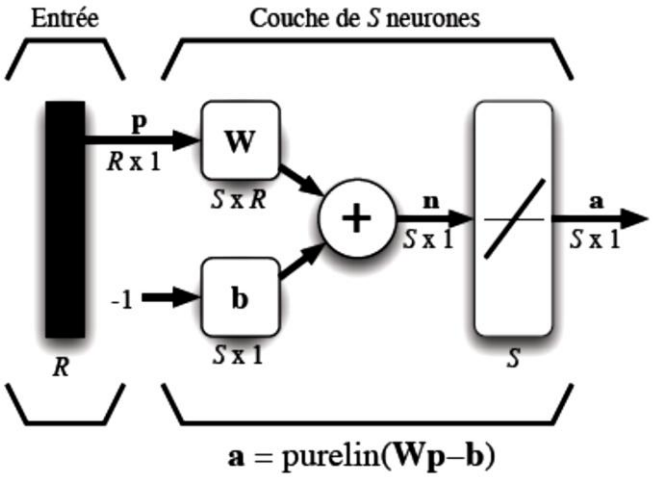

FIG. B.1 – *Réseau ADALINE*

[13] *Si du bruit (information mal classé) vient perturber les données d'entrée, le perceptron ne convergera jamais. En effet, des données linéairement séparables peuvent ne plus l'être à cause du bruit. En particulier, les problèmes* non-déterministes, *c'est-à-dire pour lesquels une même description peut représenter des éléments de classes différentes ne peuvent pas être traités à l'aide d'un perceptron* [10]

En pratique, la règle du LMS a débouché vers de nombreuses applications dont une des plus fameuses est l'annulation de l'écho pour les communications téléphoniques. Lorsque vous faites un appel inter-urbain ou outre-mer, vous vous trouvez peut-être, sans le savoir, à utiliser un réseau ADALINE.

Comme au concept de l'apprentissage par correction des erreurs, et comme son nom l'indique, la règle LMS consiste à tenter de minimiser un indice de performance F basé sur l'erreur quadratique moyenne. Possédant un ensemble d'apprentissage de Q associations *stimulus/cible* $\{(\mathrm{p}_q, \mathrm{d}_q)\}, q = 1, \ldots, Q$ où p_q représente un vecteur stimulus (entrées) et d_q un vecteur cible (*sorties désirées*), à chaque instant *t*, on peut propager vers l'avant un stimulus différent $\mathrm{p}(t)$ à travers le réseau de la *figure B.1* pour obtenir un vecteur de sorties $\mathrm{a}(t)$.

Ceci nous permet de calculer l'erreur $\mathrm{e}(t)$ entre ce que le réseau produit en sortie pour ce stimulus et la cible $\mathrm{d}(t)$ qui lui est associée :

$$\mathrm{e}(t) = \mathrm{d}(t) - \mathrm{a}(t) \tag{B.1}$$

Sachant que tous les neurones d'une même couche sont indépendants les uns des autres, et pour simplifier les équations, nous allons développer la règle LMS pour $S = 1$, c'est-à-dire le cas d'un seul neurone. Ensuite, nous pourrons facilement l'étendre au cas général de *S* neurones. Nous allons aussi regrouper tous les paramètres libres du neurone en un seul vecteur x:

$$\mathrm{x} = \begin{bmatrix} \mathrm{w} \\ b \end{bmatrix} \tag{B.2}$$

De même, nous allons regrouper en un vecteur y le stimulus p à l'entrée virtuelle -1 associé au biais du neurone :

$$\mathrm{y} = \begin{bmatrix} \mathrm{p} \\ -1 \end{bmatrix} \tag{B.3}$$

Ce qui nous permettra d'écrire la sortie a du neurone sous une forme simplifiée :

$$a = \mathrm{w}^T \mathrm{p} - b = \mathrm{x}^T \mathrm{y} \tag{B.4}$$

Nous allons donc travailler avec le signal d'erreur scalaire $e(t) = d(t) - a(t)$ et construire notre indice de performance F en fonction du vecteur x des paramètres libres du neurone :

$$F(x) = E\left(e^2(t)\right) \tag{B.5}$$

où $E[.]$ désigne *l'espérance mathématique*. Le problème avec cette équation est que l'on ne peut pas facilement calculer cette espérance mathématique puisqu'on ne connaît pas les lois de probabilité de x. On pourrait faire la moyenne des erreurs pour les Q associations d'apprentissage mais ce serait long. Une idée plus intéressante, et plus performante en pratique, consiste simplement à *estimer* l'erreur quadratique moyenne par l'erreur quadratique instantanée pour chaque association d'apprentissage :

$$\hat{F}(x) = e^2(t) \tag{B.6}$$

Alors, à chaque itération de l'algorithme, on peut calculer le vecteur gradient de cet estimé :

$$\hat{\nabla}F(x) = \nabla e^2(t) \tag{B.7}$$

où les R premiers éléments de $\nabla e^2(t)$ correspondent aux dérivés partielles par rapport aux R poids du neurone, et le dernier élément correspond à la dérivé partielle par rapport à son biais. Ainsi :

$$[\nabla e^2(t)]_j = \frac{\partial e^2(t)}{\partial w_{1,j}} = 2e(t)\frac{\partial e(t)}{\partial w_{1,j}}, \qquad j = 1, \dots, R, \tag{B.8}$$

et

$$[\nabla e^2(t)]_{R+1} = \frac{\partial e^2(t)}{\partial b} = 2e(t)\frac{\partial e(t)}{\partial b} \tag{B.9}$$

Il s'agit maintenant de calculer les deux dérivés partielles de $e(t)$ par rapport à $w_{1,j}$:

$$\frac{\partial e(t)}{\partial w_{1,j}} = \frac{\partial [d(t) - a(t)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}[d(t) - (w_1^T p(t) - b_1)]$$

$$= \frac{\partial}{\partial w_{1,j}}\left[d(t) - \left(\sum_{k=1}^{R} w_{1,k}p_k(t) - b_1\right)\right]$$

$$= -p_j(t) \tag{B.10}$$

et par rapport à b :

$$\frac{\partial e(t)}{\partial b} = 1 \tag{B.11}$$

Notez bien que les termes $p_j(t)$ et -1 sont les éléments de y, de sorte qu'on peut écrire :

$$\widehat{\nabla}F(x) = \nabla e^2(t) = -2e(t)y(t) \tag{B.12}$$

Ce résultat nous permet aussi d'apprécier la simplicité qu'engendre l'idée d'utiliser l'erreur instantanée plutôt que l'erreur moyenne. Pour calculer le gradient estimé de notre indice de performance, il suffit de multiplier l'erreur instantanée par le stimulus d'entrée.
L'équation B.12 va nous permettre d'appliquer la méthode de la descente du gradient décrite par *l'équation 3.13* (*chapitre03*) pour modifier les paramètres du neurone dans le sens d'une diminution de *F* :

$$\Delta \mathrm{x}(t) = \mathrm{x}(t+1) - \mathrm{x}(t) = -\eta \nabla F(\mathrm{x})|_{\mathrm{x}=\mathrm{x}(t)} \tag{B.13}$$

En substituant $\nabla F(x)$ par $\widehat{\nabla}F(x)$, on obtient :

$$\Delta \mathrm{x}(t) = 2\eta e(t)y(t) \tag{B.14}$$

Ce qui équivaut à :

$$\Delta \mathrm{w}(t) = 2\eta e(t)\mathrm{p}(t) \tag{B.15}$$

$$\Delta b(t) = -2\eta e(t) \tag{B.16}$$

Les deux équations précédentes définissent la règle LMS de base. On la nomme également règle de Windrow-Hoff, du nom de ses auteurs. Dans le cas d'une couche de *S* neurones, nous pourrons mettre à jour chaque rangée *i* de la matrice de poids ainsi que chaque élément *i* du vecteur de biais à l'aide des équations suivantes :

$$\Delta \mathrm{w}_i(t) = 2\eta e_i(t)\mathrm{p}(t) \tag{B.17}$$

$$\Delta b_i(t) = -2\eta e_i(t) \tag{B.18}$$

Ce qui nous permet de réécrire le tout sous la forme matricielle :

$$\Delta \mathrm{W}(t) = 2\eta \mathbf{e}(\mathrm{t})\mathrm{p}^{\mathrm{T}}(t) \tag{B.19}$$

$$\Delta \mathrm{b}(t) = -2\eta \mathbf{e}(t) \tag{B.20}$$

Même si nous ne démontrerons pas ici la convergence de l'algorithme LMS, il importe de retenir que pour les indices de performance quadratiques (comme dans le cas ADALINE), la méthode de la descente du gradient est garantie de converger vers un minimum global, à condition de restreindre la valeur du taux d'apprentissage. En pratique, nous sommes intéressés à fixer η le plus grand possible pour converger le plus rapidement possible (par de grands pas). Mais il existe un seuil à partir duquel un trop grand η peut faire diverger l'algorithme. Le gradient étant toujours perpendiculaire aux lignes de contour de $F(x)$, un petit η permettra de suivre ces lignes de contour vers le bas jusqu'à ce qu'on rencontre le minimum global. En voulant aller trop vite, l'algorithme peut sauter par-dessus un contour et se mettre à osciller. Dans le cas quadratique, les lignes de contour ont une forme elliptique comme à la *figure B.2*. Lorsque le taux est faible, la trajectoire est continue mais peut converger lentement vers l'optimum. Avec un taux plus élevé (moyen), les pas sont plus grands mais peuvent avoir tendance à osciller. On atteint normalement l'optimum plus rapidement. Lorsque le taux est trop élevé, l'algorithme peut diverger.

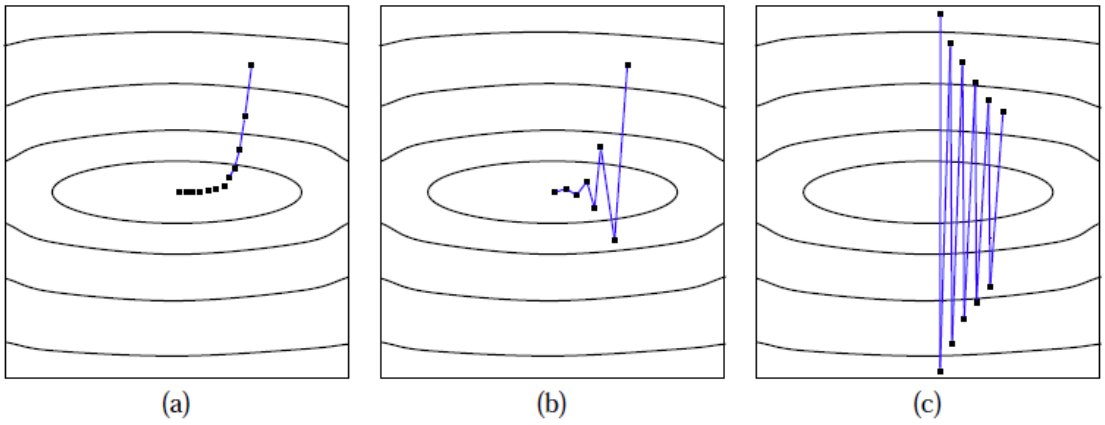

FIG. B.2 – *Trajectoire de la descente du gradient pour différents taux d'apprentissage : (a) taux faible ; (b) taux moyen ; (c) taux (trop) élevé.*

On peut montrer que pour garantir la convergence de l'algorithme LMS avec le réseau ADALINE, il faut que $0 < \eta < \frac{1}{\lambda_{max}}$ où λ_{max} est la plus grande valeur propre de la matrice $E[y\, y^T]$.

Pour initialiser l'algorithme, il s'agit simplement de fixer tous les poids et biais du réseau à zéro. Puis, pour réaliser l'apprentissage, il s'agit de lui présenter toutes les associations stimulus/cible disponible, à tour de rôle, et de mettre les poids à chaque fois en utilisant les *équations B.19 et B.20*. Une période d'entraînement correspond à appliquer ces équations une

fois pour chaque couple $(p_i, d_i),\ i = 1, \ldots, Q$. Notez qu'il peut être avantageux de permuter l'ordre de présentation à chaque période. L'algorithme itère ainsi jusqu'à un nombre maximum (fixé a priori) de périodes ou encore jusqu'à ce que la somme des erreurs quadratiques en sortie soit inférieure à un certain seuil. [24]

En conclusion, l'apprentissage par perceptron ou par la règle LMS ne sont rien d'autre que des techniques de séparation linéaire qu'il faudrait comparer aux techniques utilisées habituellement en statistiques. Ces méthodes sont non paramétriques, c'est-à-dire qu'elles n'exigent aucune autre hypothèse sur les données que la séparabilité. Une classification correcte d'un petit échantillon n'a donc aucune valeur prédictive. Par contre, lorsque l'on travaille sur suffisamment de données et que le problème s'y prête, on constate empiriquement que le perceptron appris par un des algorithmes précédents a un bon pouvoir prédictif. Il est bien évident que la plupart des problèmes d'apprentissage qui se posent naturellement ne peuvent pas être résolus par des méthodes aussi simples : il n'y a que très peu d'espoir que les exemples *naturels* se répartissent *sagement* de part et d'autre d'un hyperplan. Une manière de résoudre cette difficulté serait soit de mettre au point des *séparateurs non-linéaires* en complexifiant l'espace de représentation de manière à linéariser le problème initial. C'est ce que permettent de faire les réseaux multicouches (*section 3.7*). [17]

Références bibliographiques :

[1] Les "Smart Grids" ou "Réseaux Intelligents" http://www.enerzine.com/14/8562+les-smart-grids-ou-reseaux-intelligents+.html, 10/2009.

[2] M. T. Haque, A.M. Kashtiban, "Application of neural networks in power systems; a review", *World academy of science, Engineering and technology, 6/2005, pp 53-57.*

[3] F. Michael, L. Yun Kang, Projet de Conduite des réseaux électriques ELE234 : *dispatching économique avec/sans pertes*. Université libre de Bruxelles, 2007.

[4] J. Restrepo, *Unit commitment with primary frequency constraint in electric power systems*, McGill University, Montréal, Québec, Canada, 2/2005.

[5] A. Merev, "Comparison of the economic dispatch solutions with and without transmission losses", *JEE, Vol.2, No. 2, 2002, pp. 521-525.*

[6] T. Bouktir, L. Slimani, "Optimal power flow of the Algerian electrical network using an Ant Colony optimal method", *Leonardo Journal of Sciences ISSN 1583-0233, Issue 7, July-December 2005, pp. 43-57.*

[7] D. N. Gujarati, *Econométrie,* De Boeck, Ouvertures Economiques, 2004, 1010 p.

[8] J-P. LENOIR, "Ajustement analytique régression-corrélation", *probabilités-statistiques, Département de mathématique d'Orsay, France, pp 117-138.*

[9] N. Regnault, *Illustration de la méthode des moindres carrés.* http://fr.wikipedia.org/wiki/Fichier:Moindres_carres_introduction.png, 27/5/2007.

[10] Portail collaboratif de partage de la connaissance, *Mathématique statistiques, moindres carrés*, http://www.a525g.com/mathematiques/moindres-carres.php, 2/2008.

[11] Prajneshu, "Fuzzy regression methodology", *I.A.S.R.I., Library Avenue, New Delhi. 11/2008.*

[12] Jcmiras.Net_01, *The lambda iteration method for solving optimal dispatch* , http://www.jcmiras.net/jcm/item/15/

[13] T. Overbye, *Economic dispatch and optimal power flow*, ECE 476: Power system analysis Lecture 17, Department of Electrical and Computer Engineering, University of Illinois, 2008.

[14] Mouhamadi, *méthode de la plus grande pente*, projet n°2, http://www.math.univ-montp2.fr

[15] J. Zhu, *Optimization of power system operation*, Institute of Electrical and Electronics Engineers, IEEE press series on power engineering, 8/2009, 604 p.

[16] S. Haykin, *Neural Networks: A Comprehensive Foundation*, second edition, Prentice Hall PTR, 1998, 842 p.

[17] R. Gilleron, *Apprentissage automatique, les réseaux de neurones*, groupe de recherche sur l'apprentissage automatique, Université de Lille, http://www.grappa.univ-lille3.fr/polys/apprentissage/sortie005.html, 12/2007.

[18] G. Flavigna, " Nouveaux instruments d'évaluation pour le risque financier d'entreprise ", *Ceris-Cnr (Conseil National de Recherche), W.P. N° 1/2008.*

[19] L, Personnaz, I. Rivals, *Réseaux de neurones formels pour la modélisation, la commande et la classification*, CNRS Éditions, 9/2003, 464 p.

[20] H. Demuth, M. Beale, M. Hogan, *Neural Network Toolbox 6: User's Guide*, The MathWorks, Inc, 2010, 900 p.

[21] P. Borne, M. Benrejeb, J. Haggège, *Les réseaux de neurones : présentation et application*, Edition Technip, Paris, 2009, 152 p.

[22] M. Avriel, *Nonlinear Programming: Analysis and Methods*, Dover Publications, 9/2003, 512 p.

[23] J. C. Príncipe, N. R. Euliano, W. C. Lefebvre, *Neural and adaptive systems: fundamentals through simulations*, Wiley, 2000, 656 p.

[24] M. Parizeau, *Réseau de neurones GIF-21140 et GIF-64326*, Université Laval, 2004, 116 p.

[25] A. Vogt, J. G. Bared, *Accident Models for Two Lane Rural Roads segments and intersections, No. FHWA-RD-98-133: Artificial Neural Networks, 2. Literature Review*, http://www.tfhrc.gov/safety/98133/ch02/ch02_05.html, 10/1998.

[26] S. Nadi, *Multi-Dimensional Interpolation Using Artificial Neural Networks: An Urban Air Pollution Case Study*, Department of Surveying and Geomatics Eng, Islamic Azad University,Yazd,Iran,http://beta.gisdevelopment.net/index.php?option=com_content&view=article&id=15496&catid=130%3Aenvironment-air-pollution&Itemid=41, 2009.

[27] H. Lohninger, *Neural Networks Extrapolation*, Teach/Me Data Analysis http://www.vias.org/tmdatanaleng/cc_ann_extrapolation.html, 1999.

[28] S. Canu, "Réseaux de neurones artificiels: la rétropropagation du gradient", *labo PSI, INSA de Rouen, équipe : système d'information pour l'environnement.*

[29] C-W.Chua, *A stochastic pool-based electricity market simulator*, McGill University, Montreal, 11/2000, 106 p.

[30] FERC Staff, "Economic Dispatch: Concepts, Practices and Issues*", Palm Springs California, 11/ 2005.*

[31] ISO New York Independent System Operator, "Security Constrained Unit Commitment and Real-Time Commitment Rules", *Technical Bulletin 51, 11/10/2004; revised 9/9/09.*

[32] F. Denis, *cours d'intelligence artificielle introduction*, http://www.grappa.univ-lille3.fr/polys/intro/sortie003.html

[33] J. D. Glover, M. S. Sarma, *Power System Analysis & design*, Pws Pub Co, 2nd edition 01/ 1994, 583 p.

[34] Dr. D. Labed, TEC 429: fonctionnement et exploitation des réseaux électriques, *Cours de l'écoulement de puissance*, 5ème année, département d'électrotechnique, faculté des sciences de l'ingénieur, Université de Constantine, 1995-2010.

Résumé

Le développement de l'informatique et des logiciels de commande a contribué à l'innovation des réseaux électriques, on parle aujourd'hui des 'smart grids' ou réseaux intelligents. Ce développement est forcément lié à plusieurs préoccupations : énergétiques, économiques, environnementales,…etc. L'introduction des techniques de l'intelligence artificielle dans les logiciels de commande et de décision, est un élément essentiel dans la recherche et dans le développement des réseaux de demain. Les réseaux de neurones figurent parmi les techniques les plus répondues dans le domaine de l'intelligence artificielle.

La répartition économique d'énergie électrique, ou le dispatching économique est un secteur essentiel dans les réseaux électriques, où on doit générer moins d'énergie pour la même demande en diminuant les pertes linéiques, avec une bonne gestion économique pour avoir le moindre coût du kWh possible.

Dans ce mémoire, on va opter pour un dispatching plus rapide. Commençant par programmer un dispatching économique sans et avec pertes d'un réseau radial avec un programme classique (sous Matlab) de dispatching économique et avec un programme de réseau de neurones d'ont l'apprentissage est de type rétropropagation de l'erreur, ensuite on va programmer un réseau maillé de 9 jeux de barres dont 3 unités de production avec un programme classique et avec un programme de réseau de neurones à rétropropagation de l'erreur, enfin, on compare les deux méthodes utilisées en terme de vitesse d'exécution et fiabilité.

Abstract

The development of computer software has contributed to the innovation of power systems to become smart grids. This development is necessarily linked to several concerns: energetic, economic, environmental, etc. The introduction of techniques of artificial intelligence in software of control and decision is an essential element in research and development of tomorrow's networks. Neural networks are among the techniques most used in the field of artificial intelligence.

The economic dispatch is a key sector in electricity network, where it must generate less energy for the same demand with good economic operation reducing repartition grid losses to have the least cost of kWh possible.

In this thesis, we will opt for a quicker economic dispatch. Beginning with programming radial network economic dispatch with and without losses using traditional program and error backpropagation learning neural network program, then, we will program a mesh

network of 9 busses including 3 production units using traditional program and error backpropagation neural network program, finally; we will compare the two programs in terms of speed and reliability.

الملخّص

لقد ساهم تطوّر المعلوماتية و أنظمة التحكّم في تطور الشبكات الكهربائيّة, صرنا نتحدّث اليوم عن الشبكات الذكيّة. هذا التّطور ناجم عن عدّة انشغالات: طاقويّة, اقتصادية, بيئية,...الخ. يعدّ استخدام تقنيات الذّكاء الاصطناعي في هذه الأنظمة عنصر أساسي في مجال البحث و تطوير شبكات الغد حيث تعتبر الشبكات العصبونية من بين التقنيات الأكثر انتشارا في مجال الذّكاء الاصطناعي.

يمثّل التوزيع المقتصد للطاقة جزءا مهمًّا في منظومة الشبكات الكهربائية, حيث يتوجب إنتاج أقلّ قدر ممكن من الطاقة من أجل نفس الطلب, وهذا يكون بالتقليل من الضياع في الخطوط الكهربائيّة عن طريق التسيير الاقتصادي المحكم وبالتالي إنزال كلفة الكيلو وات الواحد إلى أقلّ قدر ممكن.

سنعمل في هذه المذكرة من أجل الحصول على توزيع اقتصادي للطاقة بأسرع طريقة ممكنة. نبدأ ببرمجة التوزيع الكهربائي لشبكة خطية دون ضياع, ثم بوجود ضياع للطاقة في خطوط النقل. البرمجة تكون بواسطة برنامج تقليدي ثمّ بواسطة برنامج شبكات العصبونات, حيث يكون تدريب الشبكة من نوع عودة انتشار الخطأ. نقوم بعدها ببرمجة شبكة كهربائية متشعّبة تتكون من 9 عقد منها ثلاثة عقد توليد. البرمجة كذلك تكون ببرنامج تقليدي ثم ببرنامج شبكات العصبونات من نوع عودة انتشار الخطأ, أخيرا, نقوم بمقارنة البرنامجين, التقليدي والعصبوني من حيث سرعة التطبيق و الفعالية.