heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Hes·so
Haute Ecole Spécialisée
de Suisse Occidentale

Auto-calibration d'un réseau de capteurs de pollution

Projet de diplôme 2006
Rapport final

Auteur : Angélique Byrde

Professeur : Stephan Robert

Assistant : Karl Baumgartner

Table des matières

1. RESUME **3**

2. INTRODUCTION **4**

3. LES RESEAUX NEURONAUX **5**

3.1. INTRODUCTION AUX RESEAUX NEURONAUX 5
3.2. HISTORIQUE DES RESEAUX NEURONAUX 7
3.3. DU LABORATOIRE A L'INDUSTRIE 9
3.4. PRINCIPES FONDAMENTAUX 10
3.4.1. Les termes utilisés 10
3.4.1.1 Les unités de traitement 11
3.4.1.2 Les liaisons 11
3.4.1.3 La fonction d'activation 11
3.5. LES DIFFERENTES TOPOLOGIES 12
3.5.1. Réseau non bouclé (feed-forward network) 12
3.5.2. Réseau bouclé (recurrent network with feedback connections) 12
3.6. L'APPRENTISSAGE 13
3.6.1. L'apprentissage supervisé 13
3.6.2. L'apprentissage non supervisé 13
3.6.3. Le sur apprentissage 14
3.6.4. Les modèles de modification de la connectivité 14
3.6.5. L'évaluation du modèle 14
3.7. RESEAU DE NEURONES « SIMPLE COUCHE » 15
3.7.1. Les réseaux à fonctions d'activation 15
3.7.2. L'apprentissage Adaline 16
3.7.3. L'apprentissage du Perceptron 17
3.7.3.1 Exemple 17
3.8. RESEAU DE NEURONES « MULTI COUCHES » 19
3.8.1. Le problème du XOR 19
3.9. LA RETRO PROPAGATION 21
3.9.1. L'algorithme de rétro propagation 21
3.9.2. L'ajout d'inertie 22

4. LE PROJET TERNOPIL **23**

4.1. INTRODUCTION AU PROJET TERNOPIL 23
4.1.1. L'algorithme d'entraînement pour un réseau simple couche 24
4.1.2. L'algorithme d'entraînement pour un réseau multicouche (Backpropagation) 24
4.2. LA PREDICTION 25
4.2.1. La prédiction avec un Réseau de neurones 25
4.2.1.1 Les 3 types de données d'entraînement [8] 26
4.2.1.2 Les méthode d'augmentation du volume des données d'entraînement [10] [12] [17] 27
4.2.1.2.1 Le réseau de neurones approximatif (ANN) 27
4.2.1.2.2 L'intégration de données historiques (IHDNN) 28
4.2.1.2.3 Solution finale adoptée 31

5. NOTRE SOLUTION : PREDICTION POUR UN SEUL SENSEUR 32
5.1. LA GENERATION DE COURBES (DEVELOPPEMENT PRATIQUE PREMIERE PARTIE) 34
5.2. L'ENTRAINEMENT 36
5.2.1. Pseudo code 37
5.2.2. Développement pratique seconde partie (l'entraînement) 38
5.3. LA SIMULATION 41
5.3.1. Pseudo code 42
5.3.2. Développement pratique troisième partie (la simulation) 43
5.4. TROIS SORTES DES COURBES 45
5.5. OPTIMISATION DU RESEAU DE NEURONES 46
5.5.1. Benchmark 53
5.5.2. Réseau simple couche ou multi couches ? 54
5.6. ESTIMATION DE LA COURBE 56
6. NOTRE SOLUTION : PREDICTION POUR UN RESEAU DE SENSEURS 60
6.1. PREMIERE PHASE DE L'ALGORITHME 61
6.1.1. Développement pratique, première partie de l'algorithme 63
6.2. SECONDE PHASE DE L'ALGORITHME 66
7. CONCLUSION 67
8. REFERENCES BIBLIOGRAPHIQUES ET WEBOGRAPHIQUES 68

Je remercie mon chef de projet M. Stephan Robert de m'avoir fait confiance en me permettant de réaliser ce projet dans le domaine des télécommunications et d'avoir été ensuite très disponible lorsque j'en avais besoin.
Je remercie aussi M. Karl Baumgartner de m'avoir suivie, aiguillée et aidée lors de la réalisation de ce projet de diplôme. Il a consacré énormément de temps pour répondre à mes questions.

1. Résumé

Dans le cadre d'un projet de recherche en liaison avec l'EPFL, un réseau auto-organisé de senseurs (capteurs de pollution) a été mis sur pied. Un des plus problèmes à l'utilisation de capteurs de pollution concerne la calibration. Chaque capteur doit être individuellement calibré, ce qui pose de réels problèmes pratiques. De plus, il faut régulièrement les re-calibrer au cours du temps. C'est pour cela qu'un système d'auto calibration est souhaitable. Le travail consiste à essayer de résoudre ce problème avec les moyens que nous avons à disposition (laboratoire de test, plateforme réelle, senseurs, ...).

2. Introduction

Dans le cadre d'un projet de recherche en liaison avec l'EPFL, un réseau auto organisé de senseurs (capteurs de pollution) a été mis sur pied. L'un des plus gros problèmes à l'utilisation de capteurs de pollution concerne la calibration. Chaque capteur doit être individuellement calibré, ce qui pose de réels problèmes pratiques. De plus, il faut régulièrement les re-calibrer au cours du temps. C'est pour cela qu'un système d'auto calibration est souhaitable.

Mon travail de diplôme consiste donc à résoudre ce problème en développant une méthode permettant de compenser le drift des capteurs (senseurs) de pollution dans un réseau de capteur sans-fil.

Un groupe de travail en Ukraine (L'ICIT de Ternopil) a proposé des solutions pour augmenter la précision des mesures des capteurs. Leurs recherches se basent sur une solution avec des réseaux de neurones (RN) pour prédire la déviation des capteurs.

Ma principale tâche est donc de (plus ou moins) recréer ce que ces chercheurs ont réalisé, le tester, créer des variantes, les documenter et peut être même améliorer leur solution.

Un second groupe de travail (de UCLA) a réalisé une étude sur l'auto calibration par la collaboration de senseurs placés au même endroit [19]. En maximisant la corrélation des valeurs de nos senseurs deux à deux, ils réussissent à créer une matrice de corrélation permettant de calculer les valeurs d'un senseur x en partant des valeurs d'un senseurs y (avec ou sans intermédiaires).

Ma seconde tâche est donc d'essayer d'auto calibrer non plus un senseur mais tout un réseau de senseurs en s'aidant de ce que ce groupe de travail à publié (avec ou sans RN).

3. Les réseaux neuronaux

Mon pré projet de diplôme concernait principalement l'étude de l'auto-calibration des réseaux de senseurs dans le domaine de la pollution. Mon projet de diplôme lui intègre une nouvelle notion très importante qui est les réseaux neuronaux.
Au début de mon projet cette notion m'était quasiment inconnue. Cette première partie à donc pour but de synthétiser l'état de mes connaissances sur ce sujet après lecture de différents documents et livres empruntés.

3.1. Introduction aux réseaux neuronaux

Un réseau de neurones (ou Artificial Neural Network en anglais) est un modèle de calcul dont la conception est très schématiquement inspirée du fonctionnement de vrais neurones (humains ou non). C'est une interconnexion de neurones, comportant chacun plusieurs entrées dont les valeurs déterminent la sortie (fonction d'activation). C'est en variant le poids des entrées, appelé « poids synaptique », que la valeur de sortie pourra être modifiée. Il existe différentes topologies de réseaux mais la plus courante est celle représentée par la Figure 1.

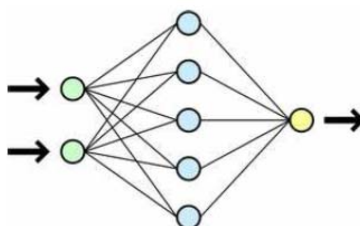

Figure 1 : représentation d'un réseau de neurones

Le réseau, au départ ignorant, va s'auto instruire à l'aide de valeurs d'entrées pour lesquelles la solution est déjà connue. Pour se faire, il va adapter ses poids synaptiques jusqu'à ce qu'il obtienne le résultat correct pour toutes les valeurs d'entraînement qui lui ont été fournies.
Une fois ce dernier terminé le réseau sera apte à construire un modèle de prédiction lui permettant de trouver des solutions jusqu'alors inconnues. Ce mécanisme, appelé « apprentissage supervisé » est entièrement automatique, ne demande aucune intervention externe et présente l'avantage de ne pas être influencé par des valeurs corrompues, car le réseau les détectera et les éliminera de lui-même et ne tiendra compte que des valeurs préalablement fournies. C'est pourquoi le jeu d'entraînement est le point critique d'un réseau de neurones artificiel, car c'est lui et lui seul qui déterminera les performances finales du système.

Un réseau de neurones (RN) (plus ou moins complexe) trouvera inévitablement la solution d'un problème pour autant qu'une relation existe entre les données d'entrée et cette solution. Il ne va pas se contenter de chercher un modèle similaire existant, ni s'inspirer du passé. Le modèle résultat de son apprentissage sera donc totalement sur mesure.

L'utilisation de RN prend tout son intérêt dans la résolution ou la prédiction de problèmes dont les relations de cause à effet entre les entrées et la sortie ne sont pas connues. Un jeu d'entrée est très fréquemment basé sur des statistiques que nous ne sommes mathématiquement pas en mesure de résoudre.

Les réseaux de neurones font partie de la famille des « Machine Learning ». Cette famille regroupe les méthodes permettant comme les réseaux de neurones d'apprendre par l'exemple. Une question se pose : Aurions nous pu utiliser une autre technique pour résoudre ce projet de diplôme ? La réponse la plus probable est non ! et l'explication se trouve dans le document [18] (annexe du rapport) qui regroupe et explique ces différentes techniques.

3.2. Historique des réseaux neuronaux

Les neurologues Warren Sturgis 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.

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 américain 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 1957 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 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 une impasse, la recherche sur les réseaux de neurones perdit une grande partie de ses financements publics.

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 eût 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.

En 1984 , c'est le système de *rétro propagation du gradient de l'erreur* qui est 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 Minsky. Proposé pour la première fois par Werbos, le Perceptron multi couche apparaît en 1986 introduit par Rumelhart, et, simultanément, sous une appellation voisine, chez Yann le Cun.

Les réseaux de neurones ont par la suite connu un essore considérable.

3.3. Du laboratoire à l'industrie

Un réseau de neurones artificiel n'est pas un produit à proprement parler mais une méthode. Avant toute utilisation de celui-ci dans une application telle qu'elle soit, quelques étapes de mise en œuvre sont donc nécessaires.

1. Il s'agit de déterminer si l'utilisation d'un réseau neuronal est adaptée et quelle est la raison. On distingue deux raisons principales : technique et/ou marketing. La raison est dite « technique » lorsque une amélioration de performance est attendue ou que l'utilisation d'autres méthodes soit impossible.
   Il est fréquent qu'une entreprise désire l'implication d'un réseau de neurones uniquement pour être en droit d'utiliser l'appellation « réseau neuronal » dans la vente du produit, On appellera cette raison « marketing ».

2. La seconde étape, déterminante, consiste à analyser les données qui seront utilisées pour l'apprentissage du réseau. Il s'agit d'identifier lesquelles sont les plus pertinentes, de les normaliser et éventuellement de les compléter.

3. Une fois cette analyse terminée, on va devoir déterminer la structure proprement dite du réseau. C'est-à-dire le nombre d'entrées et de sorties. Le nombre d'entrées nécessaire sera déterminé par la quantité de données pertinentes que l'on souhaite modéliser et le nombre de sorties par le type de sortie attendu.

4. Il est très fréquent d'inclure dans le réseau des règles d'apprentissage. Celles-ci sont des conditions, généralement très simples, destinées à aider le réseau de neurones et donc d'en améliorer les performances.

5. Enfin, la dernière étape est une étape de validation. Elle consiste à produire un certain nombre de résultats afin de les comparer aux autres méthodes qui étaient utilisées avant la mise en œuvre du réseau neuronal.

3.4. Principes fondamentaux

Dans les grandes lignes, un réseau de senseur artificiel est un mélange de simples unités de traitement qui communiquent les une aux autres en envoyant un signal à travers un grand nombre de liaisons pondérées unidirectionnelles. Un réseau de neurones peut donc se représenter par un réseau ou graphe orienté dont les nœuds sont les neurones artificiels.

3.4.1. Les termes utilisés

- Les unités de traitement sont les neurones
- La/les sortie(s) représentées par : Y_k
- La/les entrées(s) représentées par : Y_j
- Les connections entre deux neurones sont définies par un poids W_{jk} . Cette dénomination W_{jk} détermine l'effet qu'aura le signal entre le neurone j et le neurone k .
- S_k représente la somme de toutes les entrées ainsi que le biais (voir Équation 1).
- La fonction d'activation F_k détermine le nouveau niveau d'activation en se basant sur l'entrée effective S_k et le niveau d'activation courante (update).
- Le biais représenté par : θ_k

$$s_k(t) = \sum_j w_{jk}(t)\, y_j(t) + \theta_k(t).$$

Équation 1 : Equation de la somme des entrées avec le biais

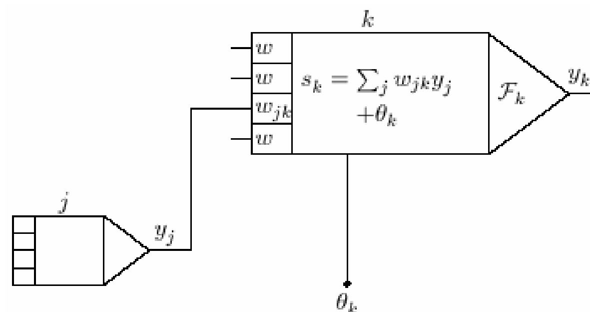

Figure 2 : Réseau de neurones avec la représentation des symboles usuels

3.4.1.1 Les unités de traitement

Chaque neurone réalise un travail assez simple. Elles reçoivent un signal d'entrée par leur voisin ou une source externe et le renvoient à leur sortie pour continuer la propagation. Leur seconde tâche est l'ajustement des poids des liaisons.

Dans un réseau de senseurs, il faut distinguer 3 sortes de neurones :

- Les neurones d'entrée : i qui reçoivent un signal de l'extérieur
- Les neurones de sortie : o qui envoient un signal dehors du réseau
- Les neurones cachés : h qui reçoivent un signal d'un neurone et le renvoie à un autre neurone.

3.4.1.2 Les liaisons

Comme expliqué plus haut, les liaisons sont pondérées. Ce Poids W_{jk} est multiplié à la valeur de sortie du neurone auquel il est connecté Y_k $(W_{jk} * Y_k)$.

Un poids positif est considéré comme une excitation et un poids négatif, comme une inhibition.

3.4.1.3 La fonction d'activation

Cette fonction d'activation sert à introduire une non linéarité dans le fonctionnement du neurone. Ces fonctions d'activation (par exemple sigmoïde) présentent généralement trois intervalles :

1. En dessous du seuil : neurone non actif (sortie = 0 ou -1)
2. Aux alentours du seuil : phase de transition
3. Au dessus du seuil : neurone actif (sortie = 1)

Cette fonction d'activation s'applique sur la totalité des entrées (voir Équation 2).

$$y_k(t+1) = \mathcal{F}_k(s_k(t)) = \mathcal{F}_k\left(\sum_j w_{jk}(t)\, y_j(t) + \theta_k(t)\right)$$

Équation 2 : Calcul de la sortie du réseau avec une fonction d'activation

Il existe plusieurs sortes de fonctions d'activations. Les 3 plus courantes sont la fonction "sgn" la fonction "semi linéaire" et la fonction "sigmoïde".

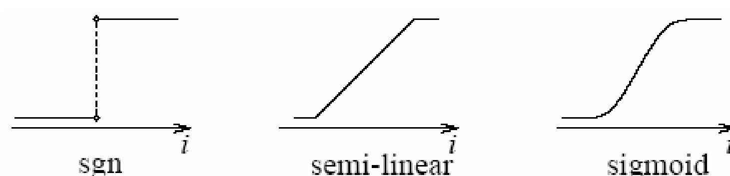

Figure 3 : Les 3 fonctions d'activation les plus courantes

3.5. Les différentes topologies

3.5.1. Réseau non bouclé (feed-forward network)

Ce réseau réalise une (ou plusieurs) fonctions de ses entrées, par composition des fonctions réalisées par chacun de ses neurones.
Si l'on se déplace dans le réseau à partir d'un neurone quelconque, en suivant les connexions, on ne peut pas revenir au neurone de départ.
Le réseau peut être étendu à un réseau multicouches (Multi-layer Perceptron ou MLP).

3.5.2. Réseau bouclé (recurrent network with feedback connections)

Lorsque on se déplace dans le réseau en suivant le sens des connexions, il est possible de trouver au moins un chemin qui revient à son point de départ (cycle). La sortie du réseau peut donc être une fonction d'elle-même. Il faut par contre faire attention aux retards, c'est pourquoi, tous cycles du graphe des connexions d'un réseau de neurones bouclés doit comprendre au moins une connexion de retard non nul.
Deux exemples classiques de réseau bouclé sont le Perceptron et l'Adaline.

Remarque : Tout réseau bouclé, aussi complexe soit-il, peut être mis sous forme canonique comportant un RN non bouclé dont certaines sorties sont ramenées aux entrées par des bouclages de retard.

3.6. L'apprentissage

On appelle apprentissage la procédure qui consiste à estimer les paramètres des neurones du réseau, afin que celui-ci remplisse au mieux la tâche qui lui est affectée.

Pour pouvoir apprendre, nous devons créer un modèle (une représentation de la réalité visible ou observable).

Le modèle boîte noire constitue la forme la plus primitive de modèle mathématique : il est réalisé uniquement à partir des données expérimentales (ou observations). Il peut avoir une valeur prédictive, dans un certain domaine de validité, mais il n'a aucune valeur explicative.
L'objectif de cette modélisation est de trouver à partir de mesures disponibles, une relation qui permette de prévoir ce que serait le résultat de la mesure s'il n'y avait pas de bruit.

A l'opposé, un modèle de connaissance, que l'on pourrait qualifier de boîte blanche est issu de l'analyse des phénomènes physiques, chimiques, biologiques,... dont résulte la grandeur que l'on cherche à modéliser. Ces phénomènes sont mis en équation à l'aide des connaissances théoriques disponibles au moment de l'élaboration du modèle. Ce dernier a donc une valeur à la fois prédictive et explicative.

Entre la boîte noire et le modèle de connaissance se situe le modèle semi physique, ou modèle boite grise qui contient à la fois des équations résultant d'une théorie, et des équations purement empiriques résultant d'une modélisation de type boîte noire.

Les RN constituent le plus souvent des modèles boîtes noires.

3.6.1. L'apprentissage supervisé

On connaît les valeurs que doit avoir la sortie du RN en fonction des entrées correspondantes. On va donc fournir au réseau des exemples de ce qu'il doit faire.

3.6.2. L'apprentissage non supervisé

Il n'y a là pas d'exemples donnés. C'est au réseau de découvrir les ressemblances entre les éléments, d'une DB par exemple, et les traduire par une proximité.

3.6.3. Le sur apprentissage

Il arrive qu'à force de faire apprendre à un RN toujours le même échantillon, celui-ci devienne inapte à reconnaître autre chose que les éléments présents dans cet l'échantillon. Le réseau est devenu trop spécialisé et ne généralise plus correctement.

3.6.4. Les modèles de modification de la connectivité

Les deux modèles d'apprentissages vus plus haut sont le résultat d'un ajustement du poids des liaisons entre les neurones en respectant certaines règles de modifications.
L'idée générale (Hebb 1949) est que si deux neurones j et k sont actives simultanément, leur interconnexion doit être renforcée.
Si j reçoit un signal d'entrée de la part de k, la règle de modification "Hebbian" proscrit la modification du poids W_{jk} en utilisant la formule : $\Delta w_{jk} = \gamma y_j y_k$: ($\gamma$ est une constante positive représentant le taux d'apprentissage).
Plusieurs autres règles de modification ont été publiées ces dernières années. Une d'elle utilisable avec un RN à apprentissage supervisé utilise la différence entre l'activation désirée d_k (proscrite par un professeur) et l'activation actuelle Y_k pour modifier le poids W_{jk}.

3.6.5. L'évaluation du modèle

L'évaluation du modèle est en fait la vérification que notre modèle fait bien ce que nous voulons (qu'il classe correctement nos éléments). Pour évaluer un modèle nous utilisons plusieurs ensembles de données (normalement trois).

Le premier ensemble (l'ensemble d'entraînement qui recouvre 50% des données totales) sert à construire le modèle initial. C'est-à-dire qu'on va soumettre ces données au réseau de neurones et celui-ci va commencer le classement des données tout en ajustant les poids des liaisons entre neurones.
Le second ensemble (l'ensemble de validation qui recouvre 25% des données totales) sert à mesurer l'efficacité du modèle.
Le troisième ensemble (ensemble de test qui recouvre lui aussi 25% des données totales) sert à ajuster le modèle initial.

Il faut faire très attention à la taille de nos ensembles. En effet, supposons que nous n'ayons qu'un petit nombre d'exemples et que le réseau s'entraîne avec ce petit ensemble. Lorsque l'on utilisera notre ensemble de test, les exemples de l'ensemble d'entraînement seront très bien classés mais pas ceux de l'ensemble de test car nous n'aurons pas assez entraîné notre réseau.

3.7. Réseau de neurones « Simple couche »

Il existe deux modèles "classiques" de RN simple couche. Le Perceptron (simple) de Rosenblatt (Rosenblatt, 1959) et l'Adaline de Widrow et Hoff (Widrow et Hoff, 1960).
Un réseau de neurone simple couche veut simplement dire que nous avons une couche d'entrée, une couche de sortie et les liaisons pondérées entre-elles.

3.7.1. Les réseaux à fonctions d'activation

Un réseau simple couche feed-forward contient une ou plusieurs sorties y chacune connectées part une liaison pondérée à chaque entrées x.

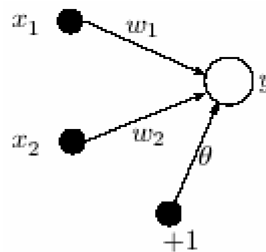

Figure 4 : Exemple de réseau simple couche

La sortie du réseau est le résultat d'une fonction d'activation appliquée sur la somme des entrées multiplié par le poids des liaisons et additionné au biais.

$$y = \mathcal{F}\left(\sum_{i=1}^{2} w_i x_i + \theta\right)$$

Équation 3 : Equation de la sortie d'un réseau de neurones

La fonction d'activation peut-être linéaire ou non linéaire. Cela dépend de l'application de notre réseau. Nous allons approfondir la fonctions linéaire threshold ou à seuil (voir Équation 4).

$$\mathcal{F}(s) = \begin{cases} 1 & \text{if } s > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Équation 4 : Equation de la fonction linéaire threshold

La sortie dépend donc de l'entrée et peut prendre les valeurs 1 ou -1. Notre RN peut donc maintenant être utilisé pour une tâche de classification. En effet suivant l'entrée, notre réseau va choisir la sortie entre deux classes possibles. Si toutes les entrées sont positives, le modèle va être assigné à la classe +1 et à contrario, si toutes les entrées sont négatives, le modèle va être assigné à la classe -1.

La séparation entre les deux classes est définie par une droite (voir Équation 5) où le poids des liaisons équivaut à la pente et le biais équivaut au déplacement.

$$w_1x_1 + w_2x_2 + \theta = 0$$

Équation 5 : Equation de la droite de séparation des classes

Une question importante serait maintenant : comment le réseau peut il apprendre à modifier les poids et le biais pour résoudre un problème de manière juste et optimale?
Il y a deux méthodes courantes pour répondre à ce problème. La première est "la règle d'apprentissage du Perceptron" et la seconde, "la règle d'apprentissage du delta" ou LMS.
Ces deux méthodes ajustent itérativement le poids des liaisons. A chaque itération les poids sont recalculés en ajoutant une correction à l'ancien poids et le seuil est lui aussi réajusté (voir Équation 6).

$$\begin{aligned} w_i(t+1) &= w_i(t) + \Delta w_i(t), \\ \theta(t+1) &= \theta(t) + \Delta\theta(t). \end{aligned}$$

Équation 6 : Ajustement des poids et du seuil

3.7.2. L'apprentissage Adaline

ADALINE est un acronyme pour ADAptive LINear Element (ou ADAptive LInear NEuron). Il fût développé par Bernard Widrow et Marcian Hoff (1960).
La règle d'apprentissage d'Adaline (aussi connue sous le nom least-mean-squares rule, delta rule, ou Widrow-Hoff rule) est une règle d'apprentissage qui minimise l'erreur de sortie en utilisant une descente de gradient approximée. Après chaque motif d'apprentissage I^p présenté, la correction s'applique au poids proportionnellement à l'erreur. La correction est calculée avant le seuillage (fonction d'activation) en utilisant $err_{ij}\ (p) = T^p - W_{ij}\ I^p$ (voir Figure 5). Les poids sont eux ajustés par l'Équation 7.

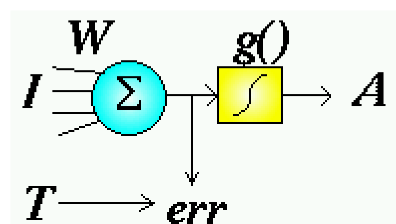

Figure 5 : Représentation d'un réseau de neurones Adaline

$$W_{ij}\ (t+1) = W_{ij}\ (t) + eta\ (T^p - W_{ij}\ I^p)\ (I^p)$$

Équation 7 : Ajustement des poids

3.7.3. L'apprentissage du Perceptron

La différence avec l'apprentissage de l'Adaline c'est qu'ici, les poids sont ajustés seulement si un motifs est mal classé.
Au départ nous avons un vecteur d'entrée x et une sortie désirée $d(x)$. Pour une tâche de classification (deux classes) la $d(x)$ est normalement +1 ou -1.

L'algorithme d'apprentissage du Perceptron peut être modélisé ainsi :

1. On commence avec des poids aléatoires.
2. On sélectionne un vecteur d'entrée x parmi les exemples d'apprentissage.
3. Si $y \neq d(x)$ (le perceptron rend une valeur erronée) nous devons
   a. modifier tous les poids w_i en utilisant la formule $\Delta w_i = d(\mathbf{x})x_i$;
   b. modifier θ en lui additionnant $\Delta\theta$ (voir Équation 8).
4. Retour au point 2 tant que $y \neq d(x)$.

$$\Delta\theta = \begin{cases} 0 & \text{if the perceptron responds correctly} \\ d(\mathbf{x}) & \text{otherwise.} \end{cases}$$

Équation 8 : Calcul du delta du seuil

3.7.3.1 Exemple

Notre Perceptron est initialisé avec les valeurs $w_1 = 1$, $w_2 = 2$ et $\theta = -2$.
Notre premier exemple A : le vecteur $x = (0.5, 1.5)$ et $d(x) = +1$.
Notre second exemple B : le vecteur $x = (-0.5, 0.5)$ et $d(x) = -1$.
Notre dernier exemple C : le vecteur $x = (0.5, 0.5)$ et $d(x) = +1$.

Pour le premier exemple, aucun problèmes :

$$(w_1 * x_1) + (w_2 * x_2) + \theta = (1 * 0.5) + (2 * 1.5) - 2 = 1.5$$

$$s > 0 \Rightarrow y = 1$$

Pour le second problème il n'y a non plus pas de problèmes.

Pour le dernier par contre :
$(w_1 * x_1) + (w_2 * x_2) + \theta = (1 * 0.5) + (2 * 0.5) - 2 = -0.5$
$s < 0 \Rightarrow y = -1$
Nous devons donc mettre à jour les poids et θ :
$\Delta\theta = d(x) = 1$
$\Delta w_1 = 0.5, \Delta w_2 = 0.5$
$w_1 = 1.5$, $w_2 = 2.5$ et -1
$(w_1 * x_1) + (w_2 * x_2) + \theta = (1.5 * 0.5) + (2.5 * 0.5) - 1 = -1$
$s > 0 \Rightarrow y = 1$
L'exemple C est maintenant classé correctement (voir Figure 6).

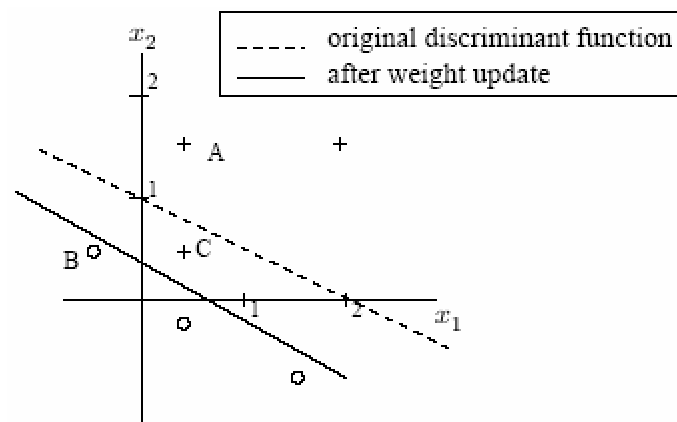

Figure 6 : Représentation graphique de la classification de l'exemple ci-dessus

3.8. Réseau de neurones « Multi couches »

3.8.1. Le problème du XOR

Les Réseaux utilisant l'un des deux algorithmes d'apprentissage pour les réseaux simple couche ont des limitations bien connues.
Minsky et Papert's ont édités une liste de ces limitations, ce qui a malheureusement eu un mauvais effet sur la recherche des RN. L'une des limitations les plus décourageantes est l'impossible représentation d'un XOR (voir Figure 7).

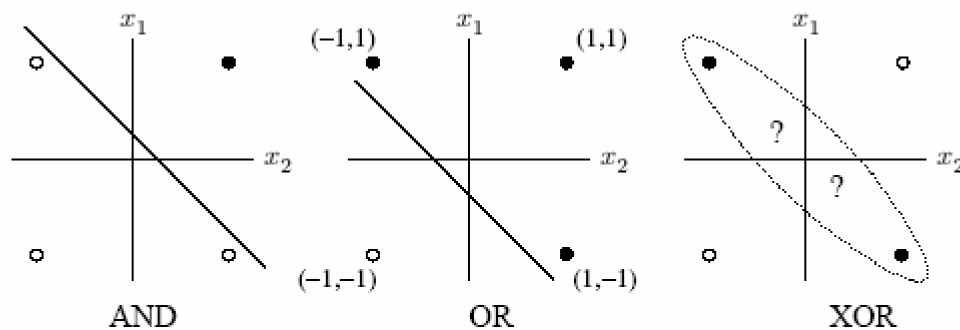

Figure 7 : Représentation graphique du problème de la classification XOR

La figure ci-dessus nous montre que l'on ne peut pas partager en deux classes ces valeurs d'entrées. Aucune ligne ne peut séparer les points (1,-1) et (-1,1) des points (-1,-1) et (1,1).

Cette limitation à une solution. En effet, en ajoutant une couche d'attribut reliée à toutes les entrées (RN multi couches). Nous ajoutons donc une couche de neurones cachés ce qui étend notre RN à un Perceptron multicouche (voir Figure 8).

Nos quatre entrées sont maintenant représentées dans un espace défini par les deux entrées et le neurone caché. Ces quatre points peuvent maintenant facilement être séparés par un plan (voir Figure 9).

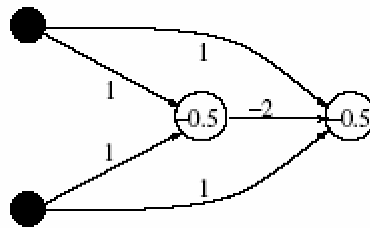

Figure 8 : réseau multi couches

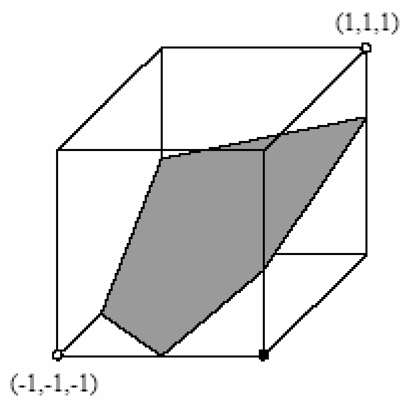

Figure 9 : classification avec un réseau multi couches

3.9. La rétro propagation

Minsky et Papert ont démontré dans leur publication de 1969 qu'un RN double couche pouvait résoudre beaucoup de problèmes qu'un RN simple couche ne pouvait solutionner mais ne pouvait toujours pas présenter une solution concernant l'ajustement des poids entre les entrées et la couche de neurones cachées.

Une réponse à la question fut présentée par Hinton et Williams en 1986. L'idée centrale de cette solution est que l'erreur des neurones de la couche cachée est déterminée en rétro propageant l'erreur des neurones de sortie.
Les différences entre ces sorties et les sorties désirées forment des erreurs qui sont corrigées via la rétro propagation, les poids du réseau de neurones sont alors changés. On corrige ces dernières selon l'importance des éléments qui ont justement participé à la réalisation de ces erreurs. Les poids synaptiques qui contribuent à engendrer une erreur importante se verront modifiés de manière plus significative que les poids qui ont engendré une erreur marginale.
En appliquant cette étape plusieurs fois, l'erreur tend à diminuer et le réseau offre une meilleure prédiction. Il se peut toutefois qu'il ne parvienne pas à échapper à un minimum local, c'est pourquoi on ajoute en général un terme d'inertie (momentum) à la formule de la rétro propagation pour aider la descente de gradient à sortir de ces minimums locaux.

3.9.1. L'algorithme de rétro propagation

Les poids sont au préalable initialisés avec des valeurs aléatoires. On considère ensuite un ensemble de données qui vont servir à l'apprentissage. Chaque échantillon possède ses valeurs cibles qui sont celles que le réseau de neurones doit à terme prédire lorsque on lui présente le même échantillon. L'algorithme se présente comme ceci :

- Soit un échantillon $\vec{x}$ que l'on met à l'entrée du réseau de neurones et la sortie recherchée pour cet échantillon $\vec{t}$
- On propage le signal en avant dans les couches du réseau de neurones : $x_k^{(n-1)} \mapsto x_j^{(n)}$
- La propagation vers l'avant se calcule à l'aide la fonction d'activation g, de la fonction d'agrégation h (souvent un produit scalaire entre les poids et les entrées du neurone) et des poids synaptiques $\vec{w}_{jk}$ entre le neurone $x_k^{(n-1)}$ et le neurone $x_j^{(n)}$. Attention au passage à cette notation qui est inversée, $\vec{w}_{jk}$ indique bien un poids de k vers j.

$$x_j^{(n)} = g^{(n)}(h_j^{(n)}) = g^{(n)}(\sum_k w_{jk}^{(n)} x_k^{(n-1)})$$

Équation 9 : Propagation avant

- Lorsque la propagation vers l'avant est terminée, on obtient à la sortie le résultat $\vec{y}$
- On calcule alors l'erreur entre la sortie donnée par le réseau $\vec{y}$ et le vecteur $\vec{t}$ désiré à la sortie pour cet échantillon. Pour chaque neurone i dans la couche de sortie (voir Équation 10).

$$e_i^{sortie} = g'(h_i^{sortie})[t_i - y_i]$$

Équation 10 : Calcul de l'erreur entre la sortie et la sortie désirée

- On propage l'erreur vers l'arrière $e_i^{(n)} \mapsto e_j^{(n-1)}$ (voir Équation 11).

$$e_j^{(n-1)} = g'^{(n-1)}(h^{(n-1)}) \sum_i w_{ij} e_i^{(n)}$$

Équation 11 : Rétro propagation arrière de l'erreure

- On met à jour les poids dans toutes les couches (voir Équation 12).

$$\Delta w_{ij}^{(n)} = \lambda e_i^{(n)} x_j^{(n-1)}$$

Équation 12 : Ajustement des poids,
avec λ représentant le taux d'apprentissage (inférieur à 1.0)

3.9.2. L'ajout d'inertie

Pour éviter les problèmes liés à une stabilisation dans un minimum local, on ajoute un terme d'inertie (momentum). Celui-ci permet de sortir des minimums locaux dans la mesure du possible et de poursuivre la descente de la fonction d'erreur. À chaque itération, le changement de poids conserve les informations des changements précédents. Cet effet de mémoire permet d'éviter les oscillations et accélère l'optimisation du réseau (voir Équation 13).

$$\Delta w_{ij}^{(n)}(t) = \lambda e_i^{(n)} x_j^{(n-1)} + \alpha \Delta w_{ij}^{(n)}(t-1)$$

Équation 13 : Ajustement des poids avec inertie au temps t,
avec α un paramètre compris entre 0 et 1.0

4. Le projet Ternopil

Maintenant que les notions de réseaux de neurones sont approfondies nous pouvons donc commencer à rechercher une solution pour auto calibrer notre réseau de neurones grâce à cette technique.
Comme décrit dans le cahier des charges, un groupe de travail de l'ICIT de Ternopil s'est déjà penché sur le sujet. Un bon départ serait donc de s'attarder à comprendre leur travail avant de développer une autre solution et c'est précisément le but de ce chapitre.

4.1. Introduction au projet Ternopil

Comme nous l'avions vu dans le pré projet de diplôme, la recherche sur les réseaux de senseurs ne cesse de s'amplifier, le nombre d'applications croit de jours en jours mais cependant un problème très important reste sans solution efficace pour le moment : La calibration (et si possible l'auto-calibration) des réseaux de senseurs.

En effet les réseaux de senseurs sont très pratiques mais leur précision n'est pas à la hauteur de nos attentes.
Les senseurs sont souvent très fragiles et placés à des endroits où ils ne subissent pas les conditions les plus favorables. C'est pourquoi après quelques temps, les réseaux de senseurs ne sont plus autant précis et que leurs mesures comportent des erreurs (biais par exemple) en raison du manque d'énergie, des mauvaises conditions,...
C'est à ce moment qu'il devient important de les re-calibrer. Il existe deux moyens de le faire :

1. La solution la plus simple mais la moins pratique est de se rendre sur chaque senseur et de les re-calibrer un à un à la main. Ceci devient bien évidemment épique si nous ne connaissons pas leur emplacement précis où s'ils sont placés à un endroit inaccessible pour l'homme. De plus un tel procédé nécessite une déconnexion des senseurs durant la calibration, ce qui peut s'avérer impossible.
2. La solution la plus compliquée (car elle nécessite une recherche importante) mais la plus pratique : l'auto-calibration (prédiction). Cette méthode consiste en une correction de l'erreur systématique introduite lors de la prise de mesure (sans interruption du réseau), sur la base de modèles mathématiques précédemment développés sur des groupes de sondes de même type.

C'est cette direction qu'ont prise les chercheurs de l'ICIT de Ternopil et c'est donc cette prédiction que nous allons traiter. Ils on en effet axés leurs études sur les avantages à calculer cette erreur à l'aide d'un réseau de neurones.

4.1.1. L'algorithme d'entraînement pour un réseau simple couche

1. Initialisation des poids et le seuil (petit nombre aléatoire).
2. Passer les valeurs en entrée et calculer la sortie en utilisant les équations Équation 14,Équation 15, Équation 16.
3. Si la sortie trouvée est identique à la sortie désirée aller au point 4. Si ce n'est pas le cas, calculer la valeur entre la sortie obtenue et la sortie désirée $\beta = y - d(x)$. Ensuite, mettre ensuite à jour les poids $w_i(t+1) = w_i(t) - \alpha(t) * \beta * x_i$ où t et $t+1$ sont l'itération courante et la suivante, i est le nombre d'entrés du Perceptron et $\alpha(t)$ est le taux d'apprentissage.
4. Répéter depuis le point 2 tant que l'erreur dépasse la valeur demandée (sortie désirée).

4.1.2. L'algorithme d'entraînement pour un réseau multicouche (Backpropagation)

1. Initialisation des poids et le seuil (petit nombre aléatoire).
2. Passer les valeurs en entrée et calculer la sortie en utilisant les équations Équation 14Équation 15, Équation 17 .
3. Si la sortie trouvée est identique à la sortie désirée aller au point 4. Si ce n'est pas le cas, calculer l'erreur de la dernière couche : $\beta_j = (y_j - d_j) \cdot \frac{dy_j}{ds_j}$, où $\frac{dy_j}{ds_j}$ est la fonction d'activation sigmoïde du neurone j. Ensuite, mettre ensuite à jour les poids $w_{ij}(t) = w_{ij}(t-1) - \alpha(t) * \beta_j * y_i$ où β_j est l'erreur du j-ème neurone de la couche actuelle, y_i la sortie de la couche précédente $\alpha(t)$ est le taux d'apprentissage.
4. Répéter depuis le point 2 tant que l'erreur dépasse la valeur demandée (sortie désirée).

$$s = \sum_{i=1}^{N} x_i \cdot w_i - T$$

Équation 14 : L'état du neurone,
où x_i est la valeur synaptique, w_i est le poids synaptique est T est la valeur du seuil.

$$y = f(s)$$

Équation 15 : La sortie du réseau de neurone

$$f(x) = K \cdot x, K = 1$$

Équation 16 : fonction d'activation linéaire pour un réseau simple couche

$$f(x) = \frac{1}{1 + e^{-x}}$$

Équation 17 : fonction d'activation non linéaire sigmoïde pour un réseau multi couches

4.2. La prédiction

Comme expliqué ci-dessus, la calibration sur chacun des senseurs présente bien des désavantages c'est pourquoi énormément de chercheurs ont axé leurs études sur la prédiction (dont l'ICIT de Ternopil).

L'algorithme de prédiction de L'ICIT [7] (dans les grandes lignes) se présente ainsi :

1. Test préliminaire effectué sur le même type de sondes et dans des conditions simulant les conditions d'exploitation. Cette première phase nous rend un modèle mathématique sur la dérive des mesures des senseurs.
2. Mesure des facteurs d'influence en conditions réelles d'exploitation et compensation du modèle sur la base de ces mesures.
3. Calcul des prévisions de dérive sur la base du modèle de dérive et application de cette compensation sur les mesures réelles.

Il est nécessaire de trouver une compensation individuelle à chaque réseau (type de réseau, conditions,...) car si les conditions ne coïncident pas alors l'algorithme de prévision de dérive devient très compliqué.

4.2.1. La prédiction avec un Réseau de neurones

L'ICIT de Ternopil a eu l'idée d'utiliser les réseaux de neurones (Perceptron simple ou multicouche) pour la prévision des mesures des réseaux de senseurs

Les réseaux de neurones sont une méthode de traitement de l'information souvent utilisée pour la prédiction, la classification,... Un réseau de neurones est un procédé « intelligent » qui apprend par l'exemple. En effet on lui donne des informations au départ (entraînement), on lui dit par exemple ce qui est bien et ce qui n'est pas bien et après un certain nombre d'informations d'entraînement il est capable (moyennant un faible taux d'erreur) de classer nos nouvelles informations ou alors de prédire un résultat

4.2.1.1 Les 3 types de données d'entraînement [8]

L'augmentation de la précision des senseurs vient des tests périodiques effectués en exploitation (calibration). Malheureusement, comme abordé plus haut, ces tests sont souvent impossibles car ils nécessitent des accès physiques ainsi qu'une déconnexion du senseur.
Pour notre prédiction, nous utilisons un réseau de neurones et ce dernier à besoin de données pour s'entraîner. Il existe trois sortes de données pouvant servir à cet entraînement :

- Les données réelles : Ces données sont obtenue en effectuant de vraies opérations sur le réseau de senseur. Elles fournissent l'exactitude la plus élevée de l'évaluation des erreurs de prévision.
- Les données historiques : Ces données sont obtenues en utilisant les résultats des opérations faites sur un même type de réseau de senseur que celui que l'on doit prédire et dans les mêmes conditions.
- Les données hypothétiques : Ces données sont obtenues par source littéraires, recherche scientifique et technologique, information des constructeurs,...

L'ICIT de Ternopil utilise pour son système de prédiction, des données historiques et des données réelles. En effet, nous verrons dans le chapitre sur l'IHDNN comment ont peut faire pour intégrer ces données historiques tout en restant efficace dans nos prévisions.

4.2.1.2 Les méthode d'augmentation du volume des données d'entraînement [10] [12] [17]

Comme nous abordé ci-dessus, la qualité de l'entraînement d'un réseau de neurones dépend du volume de données utilisées pour l'entraîner.
Il existe deux moyens d'augmenter le volume des données :

- L'utilisation d'un réseau de neurones additionnel d'approximation (ANN).
- l'intégration de données historiques.

Ces solutions augmentent donc artificiellement le nombre de données d'entraînement pour permettre ce dernier de bonne qualité.

4.2.1.2.1 Le réseau de neurones approximatif (ANN)

La première méthode d'augmentation du volume de données est l'utilisation d'un réseau de neurones approximatif (ANN).

Voici un exemple pour mieux comprendre le fonctionnement :
Imaginons un ensemble de points de calibration $x(t)$ où $t = 1..n$ (voir Figure 10).
La courbe de la dérive du senseur qui passe par ces points est divisée en $n-1$ intervalles où n est le nombre de points de calibration.

La tâche de l'ANN est de trouver q points approximatifs dans chacun des intervalles. Nous aurons donc $m = (n-1) * q$ points d'approximation.

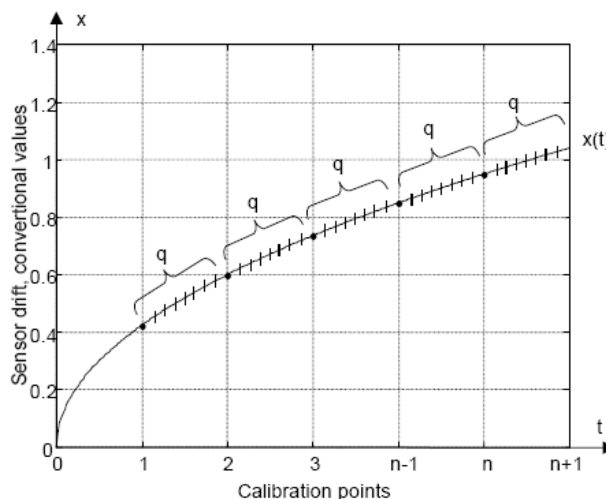

Figure 10 : Représentation graphique de la tâche de l'ANN

L'inconvénient de cette méthode est que pour entraîner notre ANN nous avons aussi besoin de données et comme nous l'avons vu, ces données ne sont pas disponibles au début de l'exploitation des senseurs.

4.2.1.2.2 L'intégration de données historiques (IHDNN)

L'utilisation d'un ANN présentant des inconvénients, la seconde méthode d'augmentation du volume de données peut devenir intéressante. Cette seconde méthode est l'intégration de données historiques

Nous obtenons une meilleure qualité de prédiction en utilisant de vraies données mais le volume de ces dernières est trop faible pour pouvoir obtenir un entraînement de bonne qualité. De plus, ces données réelles ne sont pas encore disponibles lors de la mis en place du réseau de senseur.
Pour optimiser au maximum cette prédiction, L'ICIT de Ternopil a décidé d'intégrer des données historiques parmi les données réelles pour permettre un meilleur entraînement.

Imaginons que les données historiques sont représentées par les courbes $x_1 ... x_n$
(voir Figure 11), qui aux instants de calibrations a, b, c sont égales aux points xan, xbn, xcn , n est le nombre de senseurs calibrés auparavant (le nombre de courbes).
Les données réelles sont représentées par la courbe xai, xbi, xci .
La courbe à prédire est représentée par les points xak, xbk, xck .

Au point de calibration 0, il n'y a pas d'erreur car les senseurs sont calibrés.
Le premier point de calibration a est quand à lui déjà connu car nous en avons besoin pour calculer la dérive de ce senseur au point suivant.
Pour ce qui est des points de calibration suivants, ils nous donnent les dérives $xbk..xnk$ de notre courbe à prédire.

La prédiction du point xbk est donc réalisée sur la base des points xak et xai , la prédiction du point xck , sur la base des points xbk et xbi , etc.

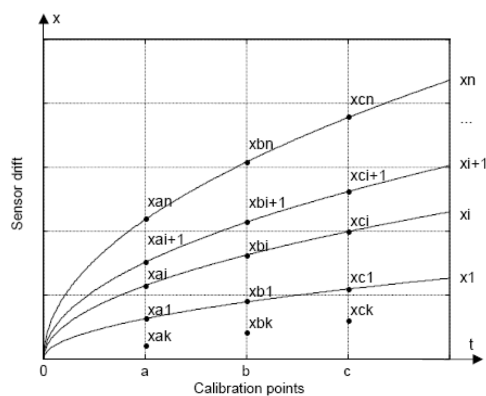

Figure 11 : représentation des courbes réelles et historiques et des points à prédire
Remarque : Le nombre de courbes historiques disponibles (n) détermine la structure de la couche d'entrée du réseau de neurones.

4.2.1.2.2.1 L'algorithme d'entraînement de l'integrating historical data neural network (IHDNN)[17]

- La courbe xi est considérée comme une courbe de valeurs réelles. Les autres courbes xj où $j = 1..i-1$ et $j = i+1..n$ sont considérées comme des courbes de valeurs historiques. Ainsi, la valeur xai de la dérive réelle du senseur et obtenue au point de calibration a et la valeur xbi de la dérive réelle du senseur et obtenue au point de calibration b.
- Calculer la valeur absolue de la distance entre le point xai et tous les autres points xaj : $\Delta ij = |xai - xaj|$
- Trier dans l'ordre décroissant toutes les distances calculées au point précédant et garder le maximum $\Delta ij^{\max} = \max \Delta ij$ et le minimum $\Delta ij^{\min} = \min \Delta ij$.
- Générer tous les vecteurs d'entraînement selon la structure établie (voir Figure 12).
- Répéter les points 1 à 4 pour les valeurs de i suivantes : $i = 1..n$.

Max. value	Interm. values	Min. value	Drift in a	Drift in b
xaj, $\Delta_{ij} = \Delta_{ij}^{\max}$	. . .	xaj, $\Delta_{ij} = \Delta_{ij}^{\min}$	xai	xbi

Figure 12 : Structure du vecteur d'entrée du réseau de neurones

4.2.1.2.2.2 La topologie du réseau IHDNN

Le réseau IHDNN est défini par un Perceptron multicouche avec une fonction d'activation log pour la couche des neurones cachés et une fonction linéaire pour la couche de sortie (voir Équation 18).

$$y_O = F_O\left(\sum_{j=1}^{M} w_{jO} \cdot F_h\left(\sum_{i=1}^{N} x_i w_{ij} - T_{hj}\right) - T_O\right)$$

Équation 18 : Valeur de sortie du Perceptron,

Avec $F_o(x)$ et $F_h(x)$: fonctions d'activation de la couche de sortie et respectivement de la couche des neurones cachés,

w_{jo} et w_{ij} : cœfficients de poids des neurones de sortie et respectivement des neurones cachés,

T_o et T_{hj} : seuils de la couches de sortie et respectivement de la couches des neurones cachés.

L'algorithme de back propagation est utilisé pour entraîner l'IHDNN (voir Équation 19).

$$\alpha(t) = \frac{4}{\left(1+(x_i^p)^2\right)} \times \frac{\sum_{j=1}^{M} (\gamma_j^p(t))^2 h_j^p(t)(1-h_j^p(t))}{\left(\sum_{j=1}^{M} (\gamma_j^p(t))^2 (h_j^p(t))^2 (1-h_j^p(t))^2\right)},$$

Équation 19 : Algorithme de Backpropagation

$$\gamma_j^p(t) = \sum_{j=1}^{M} \gamma_0^p(t) w_{j0}(t) h_j^p(t)(1-h_j^p(t))$$

Équation 20 : l'erreur du j-ème neurone de la couche des neurones cachés

Après avoir entraîné l'IHDNN avec des données historiques, le vecteur $xai+1, xai, xa1, xak$ est formé et la valeur xbk est retournée.

Après cela la « fenêtre » est déplacée sur la droite est ainsi de suite. C'est pourquoi le nombre d'IHDNN est égal au nombre de points de calibration.

4.2.1.2.3 Solution finale adoptée

L'ICIT de Ternopil a décidé de coupler ces deux solutions. En effet un premier groupe de réseaux s'occupant de l'intégration des données historiques puis de la prédiction de la dérive de notre senseur est utilisé (un réseau par point de calibration). Ces prédictions faites, un second réseau (ANN) à pour tâche de calculer un nombre x de points entre chaque prédiction faite pour pouvoir obtenir une courbe. Cette tâche réalisée, le tout est envoyé au PNN. Malheureusement ses tâches n'ont pas été détaillées par l'ICIT de Ternopil. Malgré cela, nous avons déjà, à ce niveau, une très bonne prédiction de la dérive d'un senseur.

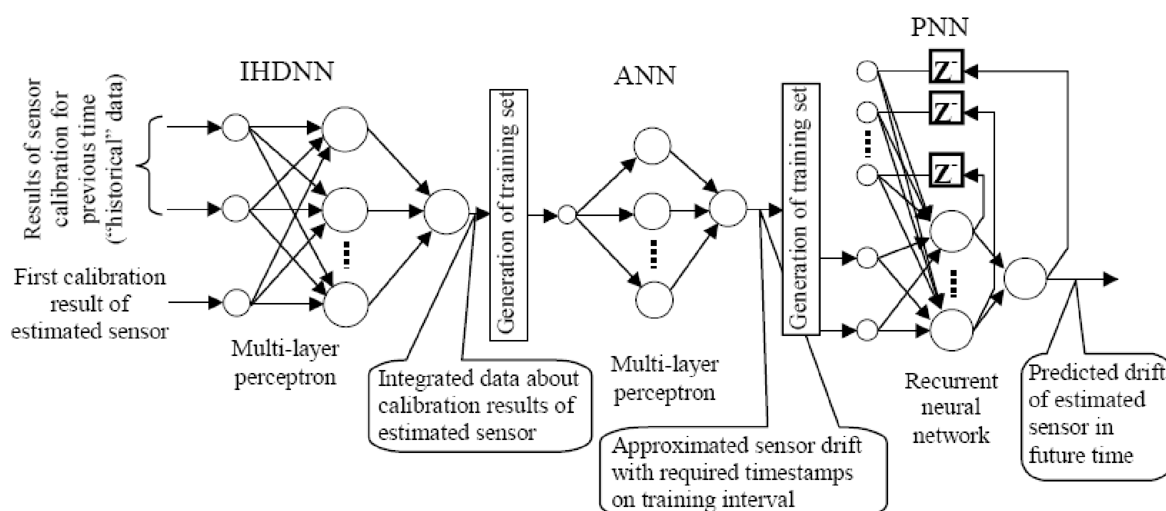

Figure 13 : Représentation graphique de la solution adoptée par l'ICIT de Ternopil

5. Notre solution : Prédiction pour un seul senseur

Maintenant que nous avons pu étudier de bons exemples sur l sujet, il est temps de développer notre propre solution de prédiction de dérive.

Nous mettons en service un senseur, nous prenons sa première valeur qui est une valeur sans erreur car le senseur ne dérive pas encore (il est calibré donc la dérive vaut 0). Nous prenons ensuite sa valeur au premier point de calibration (x temps après) et nous calculons la dérive.

L'idée serait maintenant avec ces deux premières valeurs de pouvoir prédire la dérive au point de calibration suivant et ainsi de suite pour pouvoir à chaque fois corriger cette erreur (trouver la courbe de cette dérive).

Comme nous l'avons expliqué plus haut, nous devons trouver une solution utilisant les réseaux de neurones. Nous aimerions qu'après avoir été entraîné, notre réseau puisse nous prédire une courbe en lui entrant les points de calibration 0 et 1 ainsi que les courbes d'autres réseaux de capteurs (courbes réelles et historiques).
Pour ce faire, nous avons donc besoin d'un certain nombre d'autres courbes (par exemple 1 réelle, 9 historiques). L'une des courbes ne sera pas utilisée pour l'entraînement mais pour tester notre réseau de neurones après l'avoir entraîné.
Pour ce qui est de la taille du vecteur d'entrée de nos réseaux de neurones, nous en aurons une entrée de moins que le nombre de signaux total.
Nous devons définir un certain nombre de points de calibration (10 pour notre exemple) tout en sachant que nous aurons un réseau de neurones par points de calibration non connus (9 dans notre cas). Nous pourrions avoir un seul réseau de neurones mais nous montrerons plus tard que cette solution est peu performante.

Pour plus de précision, un bon système ne doit pas se contenter de nous donner les dérives aux points de calibration mais il doit nous donner une courbe. C'est pourquoi après cette première couche de réseaux de neurones, nous aurons besoin d'un procédé (une fonction mathématique, un second réseau de neurones,…) pour calculer une courbe entre les points calculés.

Comme nous l'avons vu plus haut, nous avons besoin d'un certain nombre de courbes pour entraîner notre réseau. Ces courbes doivent représenter un senseur de même type dans les mêmes conditions que celui que nous voulons mettre en service. Ces courbes sont donc bien difficiles à obtenir, c'est pourquoi nous allons utiliser des courbes issues de simulations.

Nous devons donc créer un certain nombre de courbes (10 pour notre exemple) représentant +/- la réalité. La courbe d'un capteur à tendance à suivre une progression en $y = \sqrt{x}$, en $y = x^2$ ou en $y = x * n$.

L'algorithme d'entraînement de l'ICIT de Ternopil intègre une notion de courbe réelle et de courbes historiques.
C'est assez simple à comprendre : Lorsque nous entraînons notre réseau, nous avons besoin d'une courbe qui joue le rôle de la courbe à prédire. En effet, quand nous simulerons notre réseau, nous aurons toujours 9 entrées à notre réseau de neurones et une de ces entrées (xai) sera la valeur du point de calibration actuel de notre courbe à prédire. Nous utilisons donc à la place de cette courbe à prédire une courbe simulée dont nous connaissons déjà la sortie attendue (point de calibration suivant). Cette courbe s'appelle courbe réelle. Ainsi nous entrons dans notre réseau les valeurs de toutes nos courbes (historiques et réelle) au point de calibration x, nous lui donnons la sortie attendue pour notre courbe réelle et nous l'entraînons.

Lors de l'entraînement, l'algorithme change tour à tour la courbe réelle. Ce procédé permet premièrement au réseau de ne pas s'habituer à donner toujours les même résultats et deuxièmement à augmenter le nombre de valeurs d'entraînement.

5.1. La génération de courbes (développement pratique première partie)

Pour mieux comprendre la théorie vue ci-dessus, les contraintes, les problèmes,... un exemple serait de rigueur pour illustrer les explications.
Cet exemple complet explique l'ensemble du travail à réaliser pour imaginer, créer, tester,... nos réseaux de neurones.

La première étape est la simulation des courbes que nous avons besoin pour entraîner nos réseaux. Pour ce faire, nous créons simplement 10 courbes en $y = \sqrt{x}$, en $y = x^2$ ou en $y = x * n$ et nous y ajoutons un bruit aléatoire :

```
y(i,j)= (sqrt(j))*aleatoire(i,1);
y(i,j)= (j/25)*aleatoire(i,1);
y(i,j)= ((j^2)/14000)*aleatoire(i,1);
```

avec i=1..nbre de signaux.

Ensuite nous devons simuler dix dérives de senseurs (voir Tableau 1).

Courbes										
0	1	2	3	4	5	6	7	8	9	10
0.000	1.900	3.801	5.701	7.601	9.501	11.402	13.302	15.202	17.102	19.003
0.000	0.462	0.925	1.387	1.849	2.311	2.774	3.236	3.698	4.160	4.623
0.000	1.214	2.427	3.641	4.855	6.068	7.282	8.496	9.709	10.923	12.137
0.000	0.972	1.944	2.916	3.888	4.860	5.832	6.804	7.776	8.748	9.720
0.000	1.783	3.565	5.348	7.130	8.913	10.696	12.478	14.261	16.043	17.826
0.000	1.524	3.048	4.573	6.097	7.621	9.145	10.669	12.194	13.718	15.242
0.000	0.913	1.826	2.739	3.652	4.565	5.478	6.391	7.303	8.216	9.129
0.000	0.037	0.074	0.111	0.148	0.185	0.222	0.259	0.296	0.333	0.370
0.000	1.643	3.286	4.928	6.571	8.214	9.857	11.500	13.143	14.785	16.428
0.000	0.889	1.779	2.668	3.558	4.447	5.336	6.226	7.115	8.005	8.894
0.000	2.000									

Tableau 1 : Coordonnées des 10 courbes historiques et réelles pour chaque points de calibration et premier point de notre courbe à prédire.

En bleu nous avons notre première courbe réelle et en rouge le début de notre courbe à prédire (voir Tableau 1 et Figure 14).

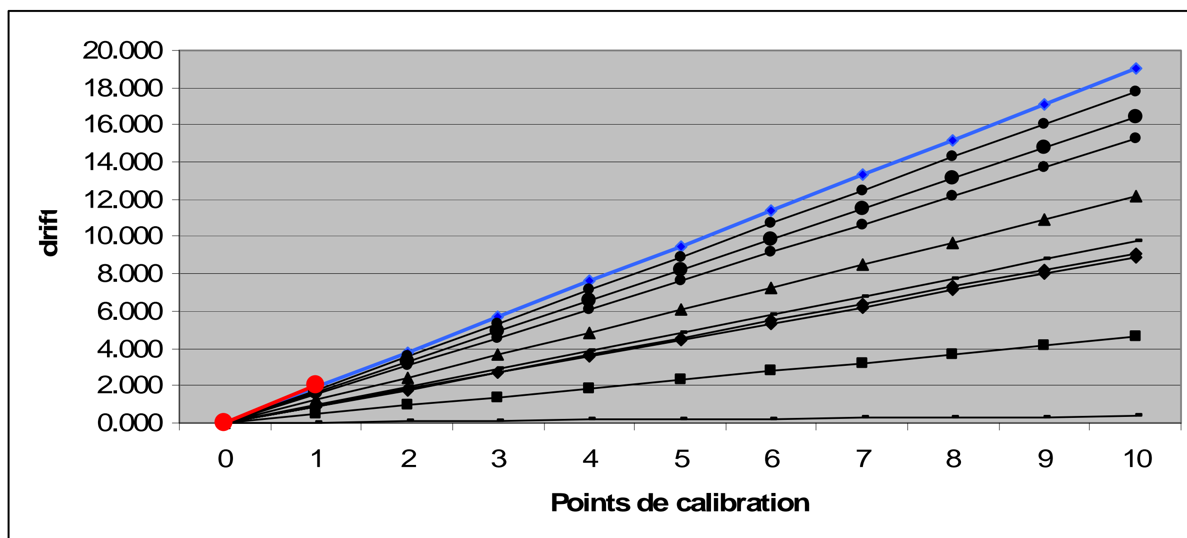

Figure 14 : Représentation graphiques des courbes simulées

5.2. L'entraînement

Comme nous l'avons vu précédemment, pour calculer la dérive de notre senseur, nous utilisons un réseau de neurones par point de calibration.
Pour créer nos réseaux de neurones, nous utilisons Matlab qui est un logiciel mathématique très puissant et qui intègre les réseaux de neurones.
La structure exacte de ces réseaux de neurones sera détaillée plus loin (nombre de couches, fonction de transfert,...).

Le point qui nous intéresse actuellement est la réalisation de l'algorithme d'entraînement de ces réseaux et l'intégration des données historiques.
Avant de pouvoir utiliser les réseaux de neurones que nous avons créé, nous devons les entraîner. En effet, ces derniers ne deviennent « intelligents » (ils peuvent prédire ou classifier) qu'après s'être entraînés avec un certain nombre d'exemples (c'est pourquoi nous avons besoin d'autant de courbes). Après cet apprentissage, notre réseau aura donc la possibilité de prédire une dérive. La précision étant donc en rapport avec l'efficacité de l'apprentissage.

Pour commencer nous avons besoin d'une matrice y contenant les valeurs de y pour les valeurs de 1..maxx correspondantes et ceci pour tous les signaux. Le nombre de lignes correspond donc au nombre de signaux et le nombre de colonnes à maxx.
Ensuite nous devons calculer $y = \sqrt{x} * nbreRandom$ pour chaque entrée de la matrice.

Nous avons ensuite besoin d'une matrice input où nous allons mettre pour chaque points de calibration et chaque signaux la valeur de y correspondante. Nous avons encore besoin d'une dimension pour pouvoir changer le type des courbes (réel ou historique).
Notre matrice input aura donc une taille de (nbSignaux-1, nbPntCalibration-1, NbSignaux-1).
Nous avons aussi besoin d'une matrice inputTriee de même dimension pour trier la matrice input.

Nous avons ensuite besoin d'une matrice output de dimension (nbSignaux-1, nbPntCalibration-1) qui contiendra pour chaque point de calibration les valeurs des sorties attendues pour les courbes réelles (xbi).

Pour finir, nous avons besoin de deux matrices, différence et diffSort, de dimension (nbSignaux-2,1) qui contiendra les valeurs des $\Delta ij = |xai - xaj|$ non triées pour différence puis triées pour diffSort. Cette matrice sera modifiée pour chaque point de calibration et pour chaque nouveaux i.

5.2.1. Pseudo code

1	Choisir les valeurs de départ			
2	Initialiser les matrices			
3	Calculer la matrice y			
4	Créer les réseaux			
5	Choisir un premier point de calibration			
6		Choisir une première courbe réelle		
7			Mettre sa valeur de y en dernière position dans inputTrie	
8			Garder cette valeur en mémoire pour la suite	
9			Calculer la sortie attendue pour cet y (donc le y suivant pour ce signal)	
10			et la mettre dans la matrice output.	
11			Pour tous les autres signaux	
12				Calculer sa valeur en y et l'insérer dans la matrice input
13				Calculer le delta et le mettre dans la matrice différence
14			Trier la matrice différence pour la mettre dans un ordre décroissant	
15			Pour chaque entrée du vecteur d'entrée (nbInput)	
16				Remplir la matrice inputTrie dans l'ordre de décroissance
17	Entraîner le réseau			

5.2.2. Développement pratique seconde partie (l'entraînement)

Le but de notre premier réseau de neurones est de prédire la dérive du senseur au point de calibration deux en utilisant les valeurs du point de calibration un (toutes connues au départ). Ce réseau va nous rendre une valeur de sortie qui est la valeur prédite pour le prochain point de calibration (xbi).
Le but de notre second réseau de neurones est de prédire la dérive du senseur au point de calibration trois en utilisant les valeurs du point de calibration deux et la sortie du réseau de neurones précédant pour l'entrée de la courbe à prédire et ainsi de suite. Nos réseaux de neurones ne peuvent accomplir leur tâche de manière optimale que si ils ont été préalablement bien entraînés.

Nous allons pour le moment nous inspirer de l'algorithme d'entraînement de l'IHDNN (réseau de prédiction et intégration des donnés historiques) de l'ICIT de Ternopil :

1. La courbe xi est considérée comme une courbe de valeurs réelles. Les autres courbes xj où $j = 1..i-1$ et $j = i+1..n$ sont considérées comme des courbes de valeurs historiques. Ainsi, la valeur xai de la dérive réelle du senseur et obtenue au point de calibration a et la valeur xbi de la dérive réelle du senseur et obtenue au point de calibration b .
2. Calculer la valeur absolue de la distance entre le point xai et tous les autres points xaj : $\Delta ij = |xai - xaj|$
3. Trier dans l'ordre décroissant toutes les distances calculées au point précédant et garder le maximum $\Delta ij^{\max} = \max \Delta ij$ et le minimum $\Delta ij^{\min} = \min \Delta ij$.
4. Générer tous les vecteurs d'entraînement selon la structure établie (voir Tableau 2).
5. Répéter les points 1 à 4 pour les valeurs de i suivantes : $i = 1..n$.

Max. value	Interm. values	Min. value	Drift in a	Drift in b
xaj , $\Delta_{ij} = \Delta_{ij}^{\max}$	. . .	xaj , $\Delta_{ij} = \Delta_{ij}^{\min}$	xai	xbi

Tableau 2 : Structure du vecteur d'entrée du réseau de neurones

Voici pour notre exemple pratique, l'application de cet algorithme d'entraînement :

1. Initialisation des valeurs.
   Xi=X4
   Xj= X1, X2, X3, X5, X6, X7, X8, X9, X10

2. Calcule de la valeur absolue de la distance entre le point xai et tous les autres points.

Deltas										
	1	2	3	4	5	6	7	8	9	10
abs(xa1-xa2)	1.438	2.876	4.314	5.752	7.190	8.628	10.066	11.504	12.942	14.380
abs(xa1-xa3)	0.687	1.373	2.060	2.746	3.433	4.119	4.806	5.493	6.179	6.866
abs(xa1-xa4)	0.928	1.857	2.785	3.713	4.641	5.570	6.498	7.426	8.355	9.283
abs(xa1-xa5)	0.118	0.235	0.353	0.471	0.588	0.706	0.824	0.941	1.059	1.177
abs(xa1-xa6)	0.376	0.752	1.128	1.504	1.880	2.256	2.632	3.009	3.385	3.761
abs(xa1-xa7)	0.987	1.975	2.962	3.949	4.937	5.924	6.911	7.899	8.886	9.873
abs(xa1-xa8)	1.863	3.727	5.590	7.453	9.316	11.180	13.043	14.906	16.769	18.633
abs(xa1-xa9)	0.257	0.515	0.772	1.030	1.287	1.545	1.802	2.060	2.317	2.574

Tableau 3 : Tableau complet des $\Delta ij = |xai - xaj|$ **pour i=4 :**

3. Tri de Tableau 3.

Deltas Tri Descendant									
1	2	3	4	5	6	7	8	9	10
1.863	3.727	5.590	7.453	9.316	11.180	13.043	14.906	16.769	18.633
1.438	2.876	4.314	5.752	7.190	8.628	10.066	11.504	12.942	14.380
0.987	1.975	2.962	3.949	4.937	5.924	6.911	7.899	8.886	9.873
0.928	1.857	2.785	3.713	4.641	5.570	6.498	7.426	8.355	9.283
0.687	1.373	2.060	2.746	3.433	4.119	4.806	5.493	6.179	6.866
0.376	0.752	1.128	1.504	1.880	2.256	2.632	3.009	3.385	3.761
0.257	0.515	0.772	1.030	1.287	1.545	1.802	2.060	2.317	2.574
0.118	0.235	0.353	0.471	0.588	0.706	0.824	0.941	1.059	1.177

Tableau 4 : Tableau trié des $\Delta ij = |xai - xaj|$ **pour chaque points de calibration et pour i=1.**

4. Construction du vecteur d'entrée du réseau de neurones.

Vecteur d'entrée 1
0.037
0.462
0.913
0.972
1.214
1.524
1.643
1.783
1.900

Tableau 5 : Vecteur d'entraînement (voir Tableau 2)

5. recommencer pour tous $xi = \overline{1,n}$, ainsi tous les signaux deviendront réels à leur tour.

5.3. La simulation

Après avoir été entraîné, notre réseau est prêt à débuter sa tâche de prédiction. Pour ce faire nous devons utiliser une fonction Matlab qui s'appelle « sim » et qui sert à simuler les résultats de notre réseau de neurones. Comme pour l'entraînement, nous devons lui donner un vecteur d'entrée et lui nous rend en sortie la prédiction du point suivant, similaire à ce que l'on appelait sortie prévue lors de l'entraînement.

Lors de l'entraînement, notre vecteur contenait neuf entrées. Il y avait huit entrées historiques et une entrée réelle. Nous lui donnions aussi une valeur pour la sortie prévue. Ainsi il entraînait le réseau pour trouver la sortie attendue correspondant aux entrées passées en paramètres.
Pour la simulation, nous gardons les mêmes contraintes de taille. Nous avons toujours un vecteur contenant neuf entrées. Huit des 10 signaux que nous avons générés jouerons le rôle des courbes historiques et la courbe à prédire quand à elle jouera le rôle de la courbe réelle. De cette manière la sortie calculée par notre réseau de neurones correspondra au point de calibration suivant de notre courbe à prédire.

Nous avons donc toujours besoin de la matrice y contenant les valeurs de y pour les valeurs de 1..maxx correspondantes et ceci pour tous les signaux.

Tout comme pour l'entraînement, nous devons continuer à trier, à chaque simulation, nos valeurs d'entrée. En effet, lors de l'entraînement, nous devions respecter un classement décroissant par rapport à la courbe réelle (voir Tableau 2) puis placer cette dernière à la fin du vecteur. La seule différence est que la courbe réelle est maintenant la courbe à prédire.
Pour préparer ce vecteur d'entrée, nous avons donc besoin de deux nouvelles matrices à deux dimensions (inputSim et inputSimTriee pour stocker nos entrées puis nos entrées triées). Ces deux matrices ont une taille de (nbInput-1, nbPntsCal) pour la première et (nbInput, nbPntsCal) pour la suivante. Cette différence de taille car la première ne contient que les courbes à classer par ordre décroissant et que la courbe à prédire n'en fait pas partie. Nous la rajoutons donc tout simplement à la fin de la seconde matrice après avoir effectué le classement.
Comme au départ nous ne connaissons que le premier point de notre courbe à prédire et que les suivants sont calculés au fur et a mesure des simulations, nous devons à chaque nouveau point de calibration remplir et trier le vecteur (les matrices inputSim et inputSimTriee au bon point de calibration).

Pour finir, nous avons besoin d'une variable simOut contenant, pour commencer, la première valeur de la courbe à prédire puis les sorties des réseaux de neurones.

Comme nous l'avons déjà vu précédemment, nous avons plusieurs réseaux de neurones en série. Chacun de ces réseaux de neurones s'occupent d'un point de calibration. Pour réaliser cette structure, nous devons donc à chaque point de calibration transférer la sortie du réseau précédant vers l'entrée du réseau suivant.
Pour commencer, nous avons donc la valeur du premier point de calibration de notre courbe à prédire (SimOut). Nous allons comparer cette valeur avec les autres signaux pour le classement puis l'insérer à son emplacement dans le vecteur. Nous allons, ensuite, passer en paramètres ce vecteur trié au réseau de neurones. Ce dernier va nous calculer une sortie et cette sortie va devenir la valeur de notre second point de calibration de notre courbe à prédire. Nous pouvons donc recommencer les manipulations avec cette nouvelle valeur et ainsi de suite.

5.3.1. Pseudo code

1	Choisir une valeur de départ pour notre courbe à prédire		
2	Initialiser les nouvelles matrices		
3	Choisir un premier point de calibration		
4		Choisir une première courbe	
5			Mettre sa valeur de y dans inputSim
6	Choisir un premier point de calibration		
7		Choisir une première courbe	
8			Calculer le delta et le mettre dans la matrice différence
9		Trier la matrice différence pour la mettre dans un ordre décroissant	
10		Pour chaque ligne du vecteur d'entrée (sans la courbe à prédire)	
11			Remplir la matrice inputSimTrie dans l'ordre de décroissance
12		Insérer SimOut à la dernière ligne de la matrice inputSimTrie	
13		Simuler les réseaux entraînés avec inputSimTrie et garder la valeur de sortie	

5.3.2. Développement pratique troisième partie (la simulation)

Pour commencer notre simulation, nous devons choisir une valeur de départ pour notre courbe à prédire.
Pour notre exemple, nous prendrons la valeur simOut=2.

Nous devons réutiliser les valeurs des huit premières courbes générées et les stocker dans une matrice (voir Tableau 6).

inputSim										
0	1	2	3	4	5	6	7	8	9	10
0.000	1.900	3.801	5.701	7.601	9.501	11.402	13.302	15.202	17.102	19.003
0.000	0.462	0.925	1.387	1.849	2.311	2.774	3.236	3.698	4.160	4.623
0.000	1.214	2.427	3.641	4.855	6.068	7.282	8.496	9.709	10.923	12.137
0.000	0.972	1.944	2.916	3.888	4.860	5.832	6.804	7.776	8.748	9.720
0.000	1.783	3.565	5.348	7.130	8.913	10.696	12.478	14.261	16.043	17.826
0.000	1.524	3.048	4.573	6.097	7.621	9.145	10.669	12.194	13.718	15.242
0.000	0.913	1.826	2.739	3.652	4.565	5.478	6.391	7.303	8.216	9.129
0.000	0.037	0.074	0.111	0.148	0.185	0.222	0.259	0.296	0.333	0.370

Tableau 6 : Matrice des coordonnées des 8 premières courbes historiques pour chaque points de calibration

Ensuite nous devons calculer le delta $\Delta ij = |xai - xaj|$ entre la valeur de simOut (qui représente la valeur de la courbe à prédire au point de calibration actuel et nos autres signaux.

Nous ne connaissons pour le moment que la première valeur de notre courbe à prédire (premier point de calibration). Nous allons donc trier les deltas de ce premier point de calibration, par ordre décroissant, puis créer le vecteur d'entrée du réseau de neurones. Nous devons insérer dans l'ordre des deltas leurs valeurs des signaux correspondants. On peut ensuite commencer la simulation qui va nous donner (simOut) la seconde valeur de notre courbe (second point de calibration) que nous pouvons insérer à la dernière ligne, au prochain point de calibration.
Nous pouvons maintenant recommencer le tri, la création du vecteur, la simulation,... et ceci pour chaque points de calibration.

Pour finir, nous obtenons une matrice où chaque colonne représente un vecteur d'entrée et la dernière ligne, notre courbe prédite (voir Tableau 7).

inputSimTrie										
0	1	2	3	4	5	6	7	8	9	10
0.000	0.037	0.074	0.111	0.148	0.185	0.222	0.259	0.296	0.333	0.370
0.000	0.462	0.925	1.387	1.849	2.311	2.774	3.236	3.698	4.160	4.623
0.000	0.913	1.826	2.739	3.652	4.565	5.478	6.391	7.303	8.216	9.129
0.000	0.972	1.944	2.916	3.888	4.860	5.832	6.804	7.776	8.748	9.720
0.000	1.214	2.427	3.641	4.855	6.068	7.282	8.496	9.709	10.923	12.137
0.000	1.524	3.048	4.573	6.097	7.621	9.145	10.669	12.194	13.718	15.242
0.000	1.783	3.565	5.348	7.601	9.501	11.402	13.302	14.261	16.043	19.003
0.000	1.900	3.801	5.701	7.130	8.913	10.696	12.478	15.202	17.102	17.826
0.000	2.000	3.873	5.643	7.283	8.960	10.674	12.473	15.324	16.604	17.890

Tableau 7 : Matrice Tableau 6 triée selon la structure de Tableau 2

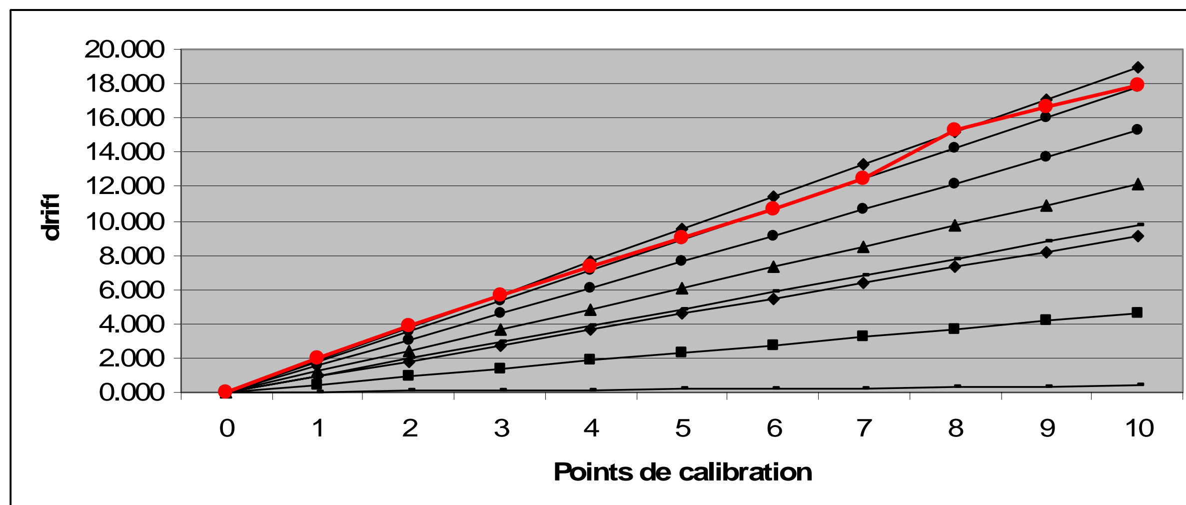

Figure 15 : Représentation graphique de la courbe prédite (rouge) par rapport aux courbes simulées (voir Tableau 6)

5.4. Trois sortes des courbes

Comme nous l'avons vu précédemment, la courbe d'un capteur à tendance à suivre une progression en $y = \sqrt{x}$, en $y = x^2$ ou en $y = x * n$ avec bien évidemment un peu de bruit aléatoire.

Notre exemple pratique utilisait des courbes linéaires mais notre algorithme de prédiction est aussi capable de prédire une courbe d'une autre progression (voir Tableau 8). Pour ce faire, il suffit simplement de modifier les courbes historiques et réelles de départ.

	0	1	2	3	4	5	6	7	8	9	10
Sqrt	0.000	3.430	4.711	5.585	6.377	7.025	7.639	8.181	8.668	9.223	9.557
Carré	0.000	0.100	0.515	1.102	1.876	2.750	5.318	8.628	11.315	14.282	17.154
Lin	0.000	2.000	3.873	5.643	7.283	8.960	10.674	12.473	15.324	16.604	17.890

Tableau 8 : exemple de coordonnées prédites par le RN pour chacune des progressions

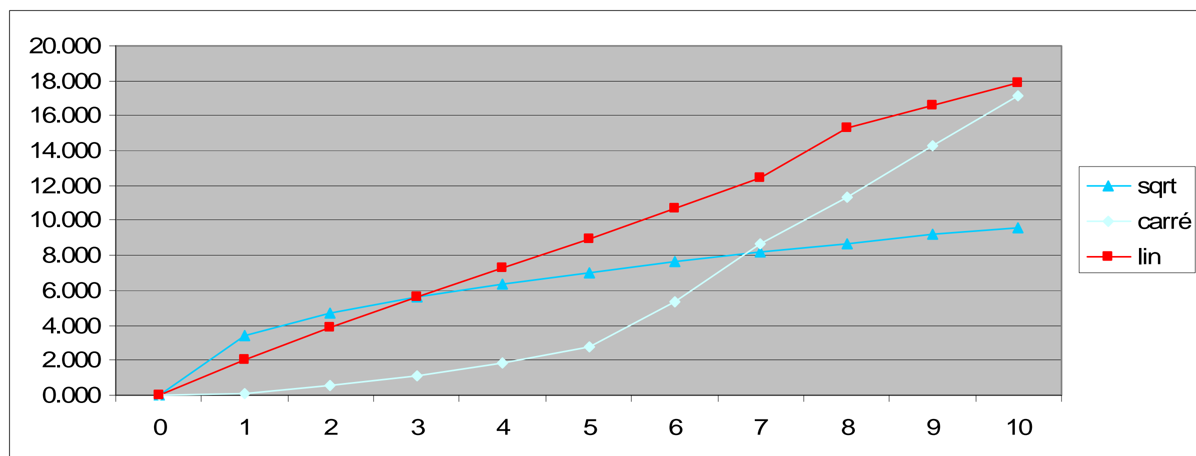

Figure 16 : Représentation graphique des progressions du Tableau 8

5.5. Optimisation du réseau de neurones

Il existe plusieurs algorithmes de Backpropagation. Certain sont reconnus pour être plus rapides, certains sont plus performants, certains utilisent moins de mémoire,...
Jusqu'à maintenant nous avions utilisé l'algorithme traingda. Cet algorithme est celui de base utilisé dans Matlab. Il est performant mais pas très rapide. Nous allons maintenant tester les performances de ces différents algorithmes ainsi que leur temps de calcul.

Ces tests ont étés réalisés avec les même courbes et les mêmes paramètres d'entraînement.

Traingda

	0	1	2	3	4	5	6	7	8	9	10
multi	0.000	0.100	0.431	1.004	1.842	2.914	3.917	5.523	7.332	9.025	11.362
Simple	0.000	0.100	0.456	1.082	1.859	2.749	4.053	5.501	7.200	8.920	11.342

Tableau 9 : prédiction avec l'algorithme traingda pour un réseau multi couche (bleu foncé) et pour un réseau simple couche (bleu clair)

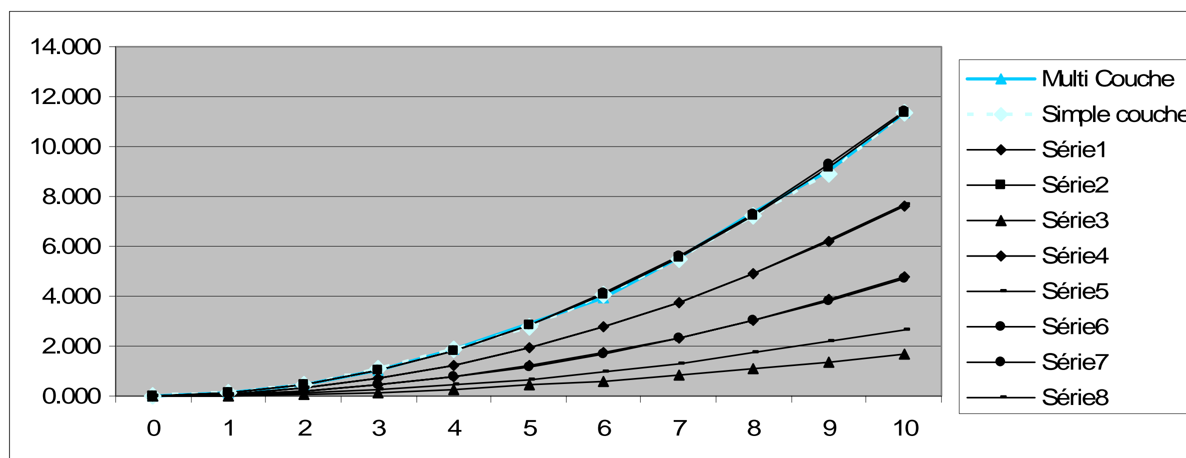

Figure 17 : représentation graphique des deux prédictions avec l'algorithme traingda par rapport aux 8 courbes de test.

Traingdx

	0	1	2	3	4	5	6	7	8	9	10
multi	0.000	0.100	0.419	1.014	1.900	2.898	4.085	5.677	7.504	9.244	11.382
Simple	0.000	0.100	0.435	0.965	1.714	2.815	4.056	5.456	7.229	9.177	10.616

Tableau 10 : prédiction avec l'algorithme traingdx pour un réseau multi couche (bleu foncé) et pour un réseau simple couche (bleu clair)

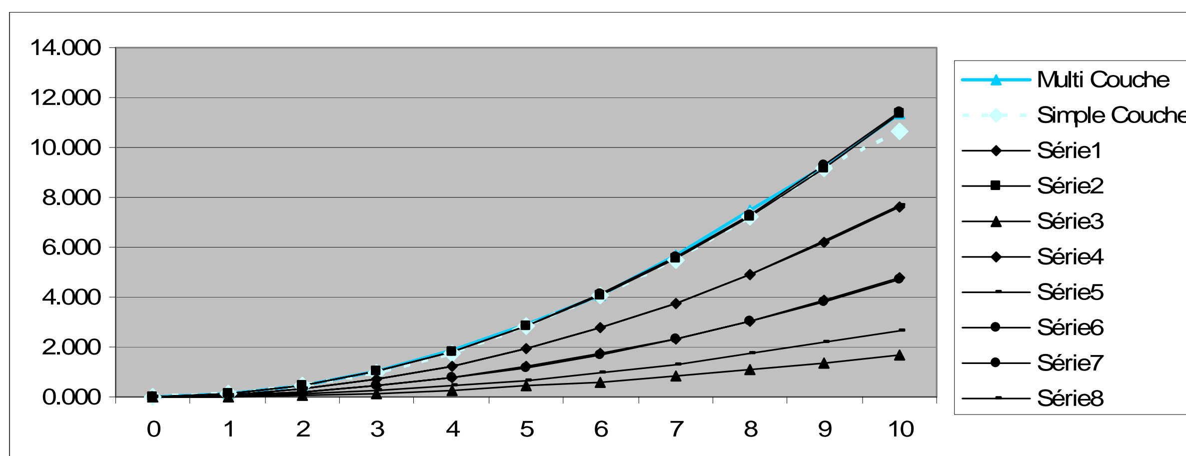

Figure 18: représentation graphique des deux prédictions avec l'algorithme traingdx par rapport aux 8 courbes de test.

Trainrp

	0	1	2	3	4	5	6	7	8	9	10
multi	0.000	0.100	0.436	0.995	1.819	2.852	4.125	5.703	7.290	9.262	10.180
Simple	0.000	0.100	0.434	0.973	1.802	2.834	4.303	5.911	7.640	9.526	11.920

Tableau 11 : prédiction avec l'algorithme trainrp pour un réseau multi couche (bleu foncé) et pour un réseau simple couche (bleu clair)

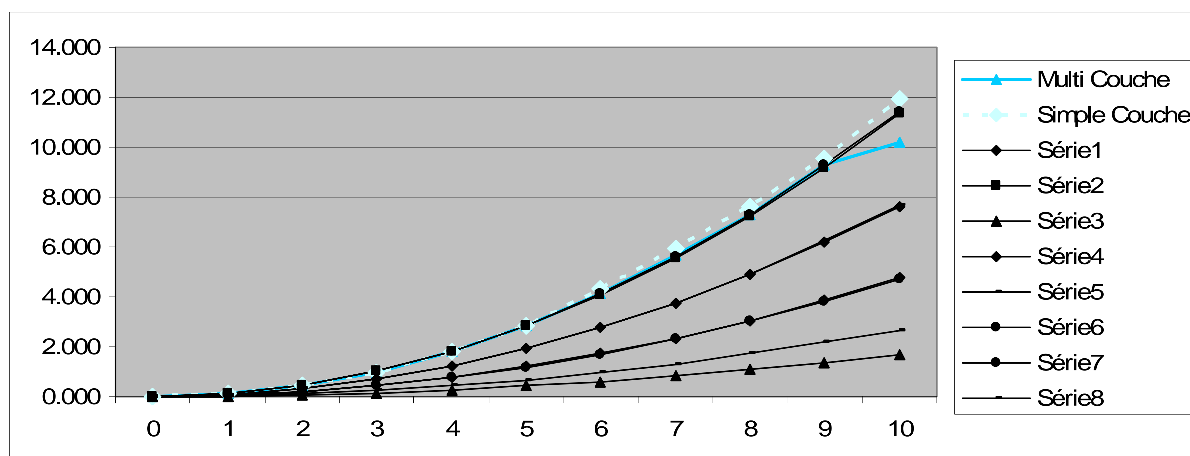

Figure 19: représentation graphique des deux prédictions avec l'algorithme trainrp par rapport aux 8 courbes de test.

Traincgf

	0	1	2	3	4	5	6	7	8	9	10
multi	0.000	0.100	0.449	1.064	1.800	2.964	4.179	5.615	7.081	9.179	11.393
Simple	0.000	0.100	0.449	0.943	1.702	2.842	3.765	5.201	7.349	9.067	11.705

Tableau 12 : prédiction avec l'algorithme traincgf pour un réseau multi couche (bleu foncé) et pour un réseau simple couche (bleu clair)

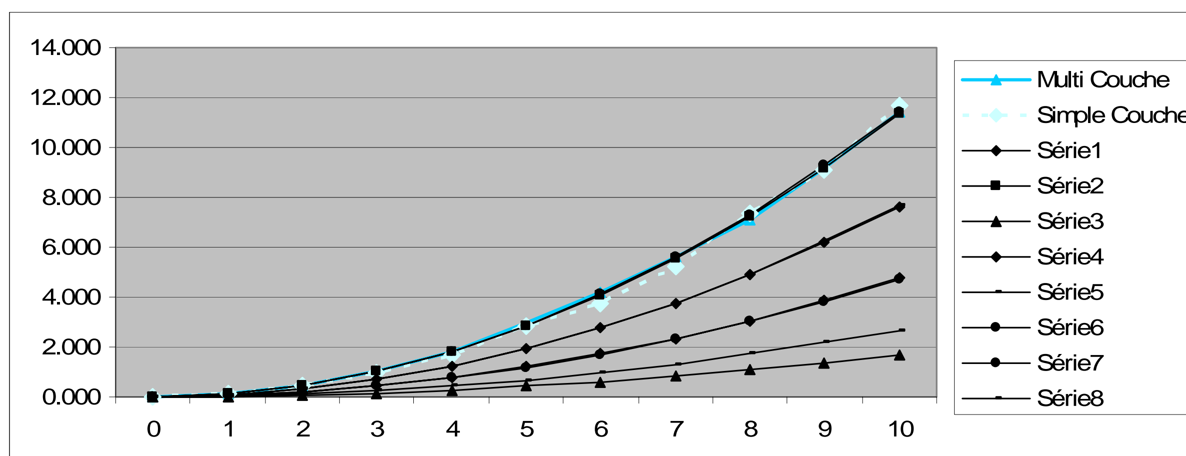

Figure 20: représentation graphique des deux prédictions avec l'algorithme traincgf par rapport aux 8 courbes de test.

Traincgb

	0	1	2	3	4	5	6	7	8	9	10
multi	0.000	0.100	0.424	1.000	1.892	2.930	4.190	5.707	7.376	9.265	12.220
Simple	0.000	0.100	0.438	1.025	1.753	2.955	4.024	5.636	6.993	9.144	10.764

Tableau 13 : prédiction avec l'algorithme traincgb pour un réseau multi couche (bleu foncé) et pour un réseau simple couche (bleu clair)

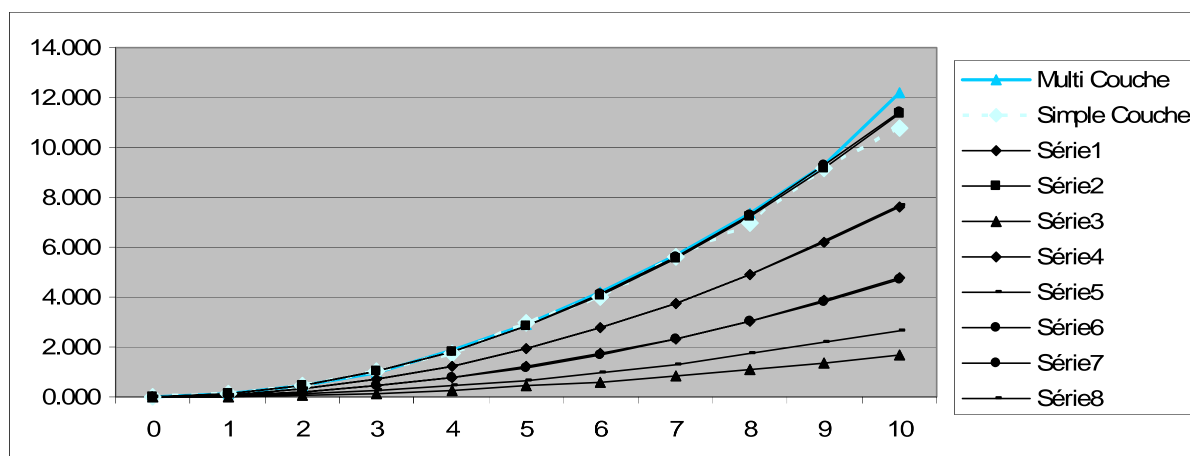

Figure 21: représentation graphique des deux prédictions avec l'algorithme traincgb par rapport aux 8 courbes de test.

Trainscg

	0	1	2	3	4	5	6	7	8	9	10
multi	0.000	0.100	0.431	1.016	1.875	2.859	4.277	5.572	7.250	9.080	11.921
Simple	0.000	0.100	0.453	1.067	1.984	2.977	4.290	5.755	7.562	9.614	12.445

Tableau 14 : prédiction avec l'algorithme trainscg pour un réseau multi couche (bleu foncé) et pour un réseau simple couche (bleu clair)

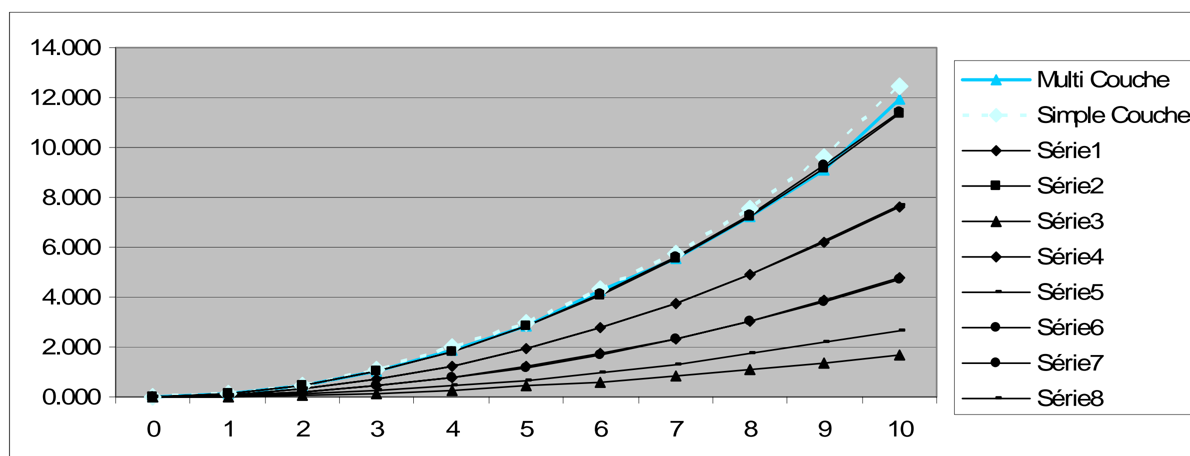

Figure 22: représentation graphique des deux prédictions avec l'algorithme trainscg par rapport aux 8 courbes de test.

Trainlm

	0	1	2	3	4	5	6	7	8	9	10
multi	0.000	0.100	0.435	0.996	1.814	2.988	4.124	5.349	7.346	9.350	11.124
Simple	0.000	0.100	0.445	1.030	1.899	3.093	4.378	5.608	7.543	9.183	11.418

Tableau 15 : prédiction avec l'algorithme trainlm pour un réseau multi couche (bleu foncé) et pour un réseau simple couche (bleu clair)

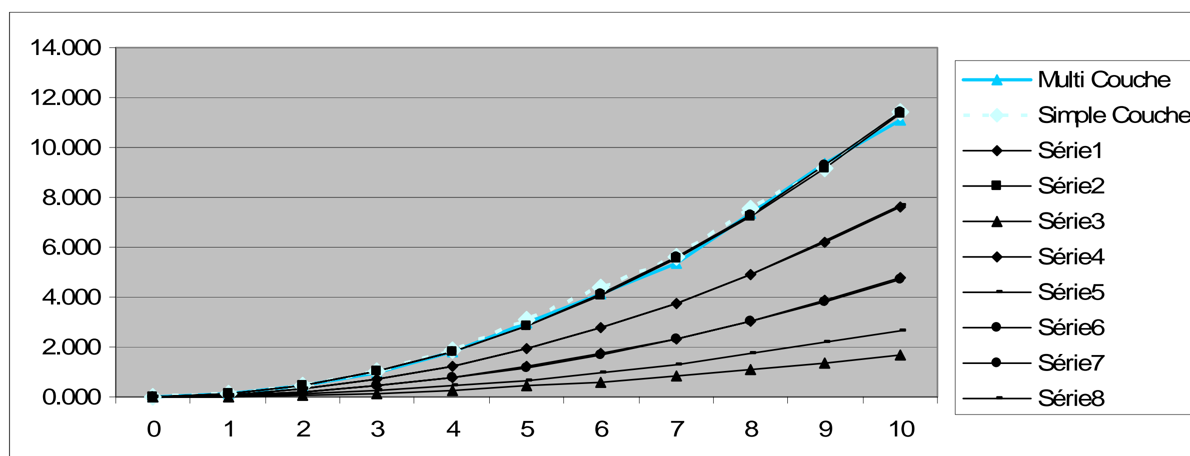

Figure 23: représentation graphique des deux prédictions avec l'algorithme trainlm par rapport aux 8 courbes de test.

Comme le montre ces tests, les courbes prédites sont très similaires. Il n'y a donc apparemment pas d'algorithme à écarter pour un manque de précision.
Nous devons donc trouver un autre critère pour tester les performances de nos algorithmes. Le temps ainsi que la mémoire utilisée peuvent être de bons critères de comparaison (voir Tableau 16).

Comparaison des temps (s)

Trainlm	6.5
Traincgb	8.1
Traincgf	12.3
Trainscg	18.1
Trainrp	21.5
Traingdx	101.1
Traingda	112.7

Comparaison de la mémoire (Ko)

Trainscg	1000
Traincgf	2000
Traingdx	2000
Trainrp	2000
Traingda	2000
Trainlm	3000
Traincgb	27000

Tableau 16 : Comparaison des temps et de la mémoire pour les différents algorithmes de Backpropagation

On peut maintenant écarter facilement certains algorithmes. Par exemple Traingda et Traingdx mettent beaucoup plus de temps que les autres algorithmes à trouver une solution à notre problème de prédiction. Traincgb quand a lui utilise beaucoup plus de mémoire que les autres algorithmes pour cette même tâche.

Trainlm et Trainscg sont apparemment les algorithmes les plus performants pour cette tâche. Par contre Trainlm converge très rapidement et apparemment ça pose des problèmes de cohérence dans les résultats.

5.5.1. Benchmark

Un benchmark, en anglais, est un point de référence servant à effectuer une mesure.
En informatique, un benchmark est un banc d'essai permettant de mesurer les performances d'un système pour le comparer à d'autres.

Pour en revenir à nos réseaux de neurones, le benchmark a testé ces différents algorithmes et voici ce qu'il en est sorti :

Trainlm converge très rapidement, cet avantage est spécialement appréciable si un entraînement très précis est attendu.
La plupart du temps, trainlm est capable d'obtenir une plus petite erreur de moindre carré que tous les autre algorithmes. Par contre, plus le poids croisse, plus la précision décroisse. De plus les performances de cet algorithme en identification de modèles sont très basses.

Trainrp est l'algorithme le plus rapide pour les identifications de modèles. Par contre il n'est pas excellent pour des problèmes d'approximation. Ses performances se dégradent rapidement quand l'erreur est réduite. Par contre ses besoins en mémoire sont très basses comparé aux autres algorithmes.

Trainscg semble performant pour une grande variété de problèmes. Particulièrement quand on a besoin d'utiliser un réseau de neurones avec un grand nombre de poids. Cet algorithme est beaucoup plus rapide que trainlm pour des problèmes d'approximation et que trainrp pour des problèmes d'identification de modèle si on utilise un grand réseau. De plus, ses performances ne se dégradent par si l'erreur se réduit et ne demande pas énormément de mémoire pour s'exécuter.

Traingdx et traingda quand à eux, sont beaucoup plus lent que les autres algorithmes. Par contre, tout comme trainrp, ils ont besoin de peu de capacité mémoire. Ces algorithmes peuvent être utiles pour certains problèmes où il est préférable de converger lentement.

5.5.2. Réseau simple couche ou multi couches ?

Toujours dans le sujet de l'optimisation de senseurs, une grande question se pose. Devons nous utiliser un réseau simple couche ou un réseau multi couches ?
Comme nous l'avons vu l'ors des tests de performance, une réseau simple couche rend d'aussi bonnes réponses qu'un réseau multi couches pourtant, il y a quand même lieu de se poser la question. En effet, l'ICIT de Ternopil n'a pas seulement fait des tests avec une seule progression de courbes mais aussi en mélangeant les trois différentes progressions que nous avons vues. C'est en faisant ce mélange qu'ils ce sont rendus compte que le réseau multi couches restait fiable mais que par contre, le réseau simple couche rendait des résultats moins cohérents.

Nous avons nous aussi réalisé ces tests avec différentes progressions de signaux (voir Figure 24). Nous avons pu constater les mêmes phénomènes que l'ICIT de Ternopil. En effet, le réseau simple couche nous a rendu de très mauvais résultats (voir Figure 25) alors que le réseau multi couche, à lui adopté une des progressions connues (voir Figure 26).

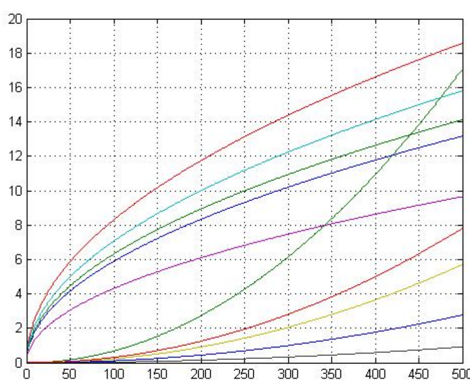

Figure 24 : Mélange de deux progressions de courbes

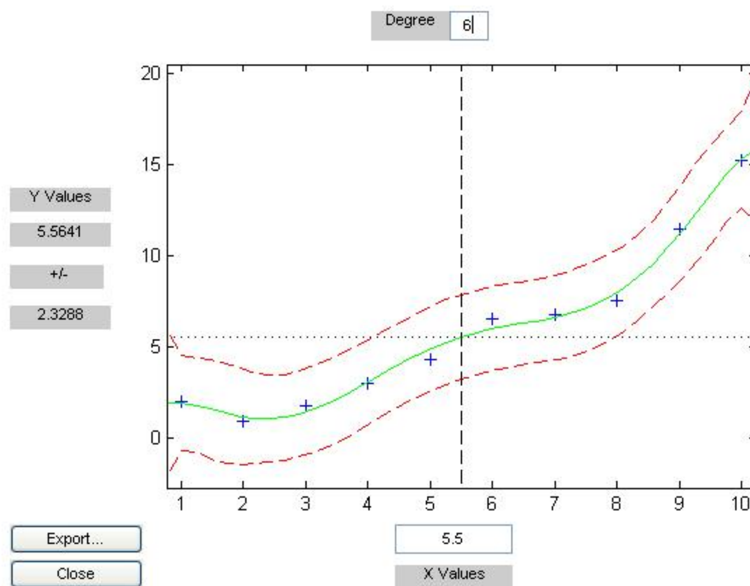

Figure 25 : Sortie d'un réseau simple couche entraîné avec des courbes de différentes progressions

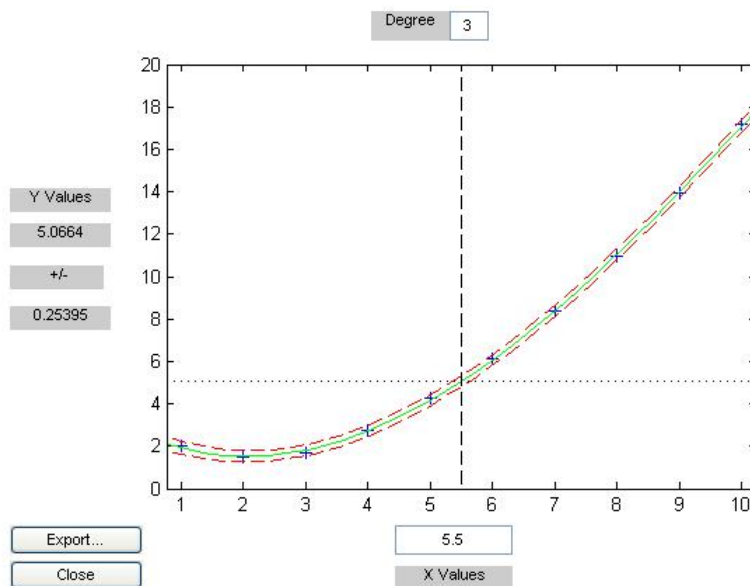

Figure 26 : Sortie d'un multi couches entraîné avec des courbes de différentes progressions

5.6. Estimation de la courbe

Comme nous l'avions vu lors de l'étude du projet mené par L'ICIT de Ternopil, ces derniers couplaient le premier groupe de réseaux s'occupant de l'intégration des données historiques et de la prédiction de la dérive avec un second réseau (ANN) ayant pour tâche de calculer un nombre x de points entre chaque prédiction faite pour pouvoir obtenir une courbe.

Cette solution n'a pas été intégrée à ce projet de diplôme car, après réflexion une autre solution mathématique et statistique pourrait être moins gourmande et performante.

La méthode des moindres carrés permet de comparer des données à un modèle mathématique censé décrire ces données. Elle permet de minimiser l'impact des erreurs en ajoutant de l'information dans le processus de mesure. C'est cette méthode qui a été utilisée pour remplacer le second réseau de neurones (ANN).
En effet, après avoir prédit un certain nombre de points de la dérive, nous utilisons la méthode des moindres carrés pour calculer la courbe qui se rapproche le plus de ces points. De cette manière nous obtenons pour tout x dans le temps une prédiction de la dérive de notre senseur.

Matlab intègre une fonction qui calcule la droite de régression d'un certain nombre de points par la méthode des moindres carrés (polyfit), ainsi qu'une fonction qui illustre graphiquement cette courbe ainsi que les points utilisés pour la calculer (polytool). Nous avons donc utilisé ces fonctions pour résoudre notre problème :

```
polytool(ordonnees,abcisses,3)
courbeRegression = polyfit(ordonnees,abcisses,3)
```

Voici ce que Matlab nous a retourné pour des progression de courbes historiques et réelles en $y = x^2$:

```
abcisses = (0.1000, 0.4269, 0.9735, 1.8863, 2.8404, 3.9559, 5.6084,
7.5001,  8.8977, 12.1481)
ordonnees = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
courbeRegression = (0.0059, 0.0319, 0.3191, -0.3253)
```

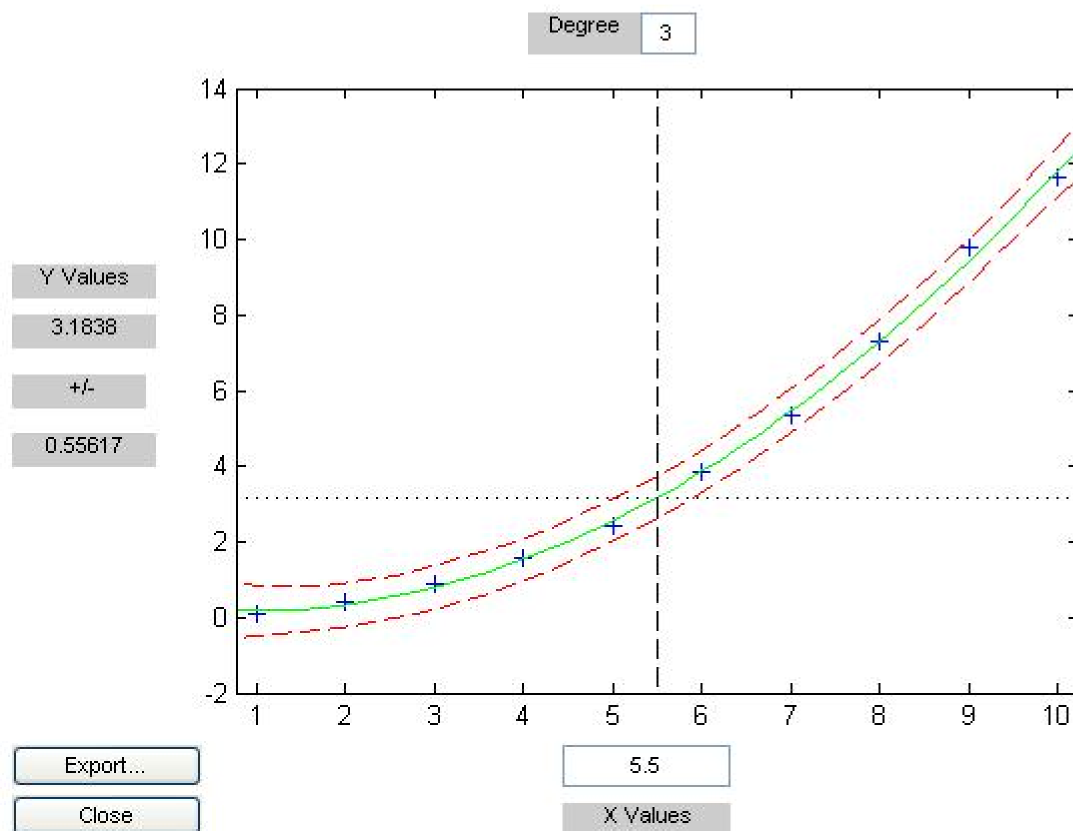

Figure 27 : représentation graphique de la courbe rendue par matlab par rapport aux points prédits pour une progression en $y = x^2$

Les variables abscisses et ordonnees sont les points prédits par notre réseau de neurones. La variable courbeRegression est la courbe de régression d'ordre trois (f(x)=0.0059 x^3 + 0.0319 x^2 + 0.3191 x - 0.3253) minimisant l'erreur des points prédits.

Voici ce que Matlab nous a retourné pour des progression de courbes historiques et réelles en $y = \sqrt{x}$:

```
abcisses = (3.4300, 4.7628, 6.8774, 7.9378, 8.8395, 9.7754, 10.5734,
11.4321, 12.0160, 12.5803)
ordonnees = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
courbeRegression = (0.0059, -0.1683, 2.2265, 1.2672)
```

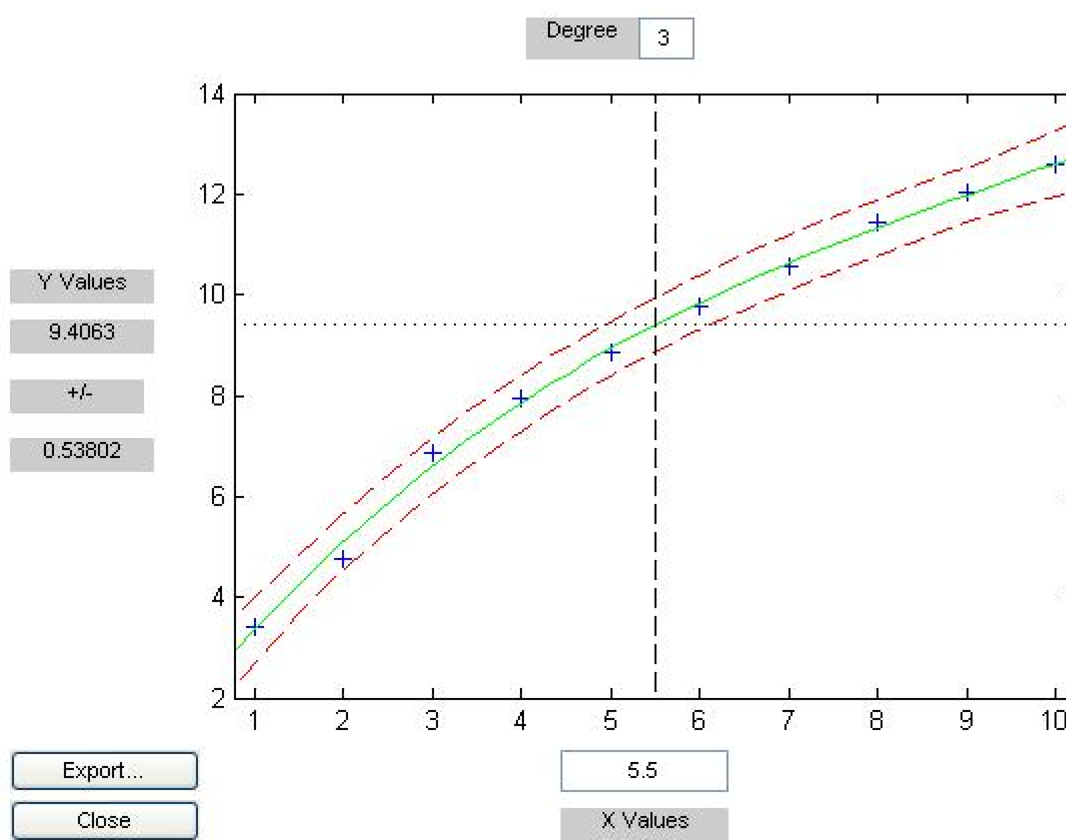

Figure 28 : représentation graphique de la courbe rendue par matlab par rapport aux points prédits pour une progression en $y = \sqrt{x}$

Voici ce que Matlab nous a retourné pour des progression de courbes historiques et réelles en $y = x * n$:

```
abcisses = (2.0000, 3.3656, 5.1556, 6.1330, 7.0226, 8.3498, 11.1011,
11.9265, 13.6450, 15.0775)
ordonnees = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
courbeRegression = (-0.0003, 0.0286, 1.1708, 0.9303)
```

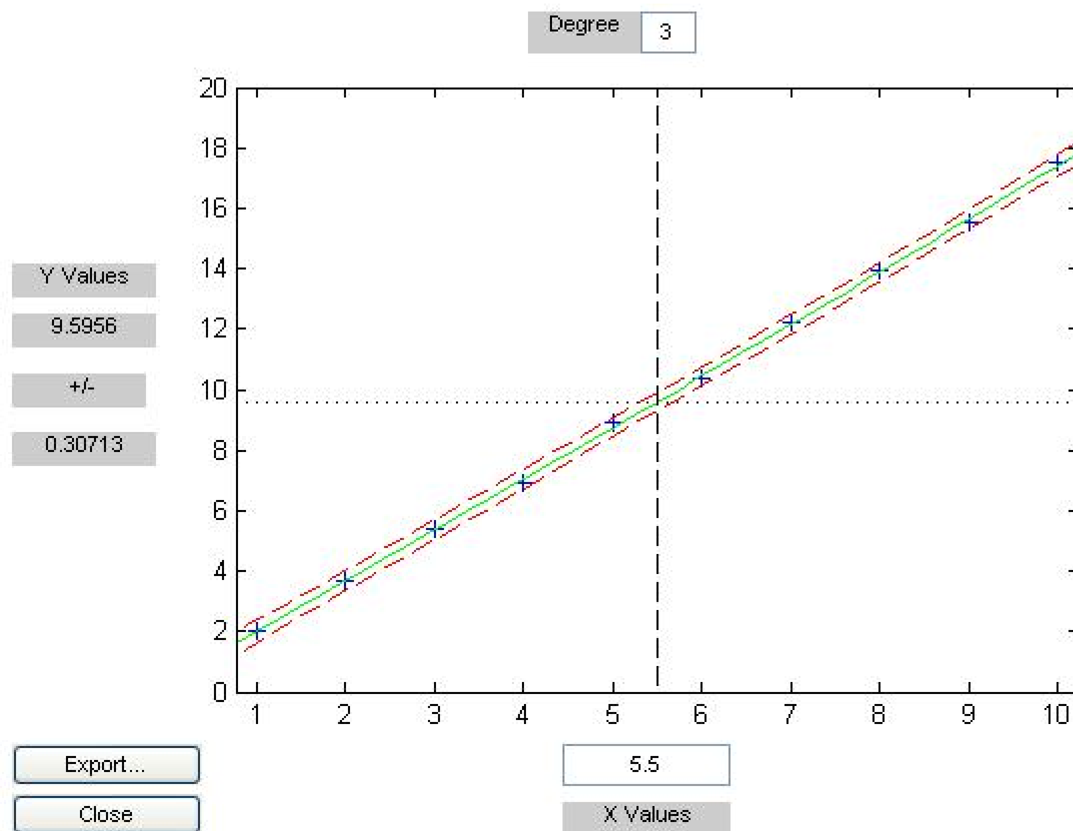

Figure 29 : représentation graphique de la courbe rendue par matlab par rapport aux points prédits pour une progression en $y = x * n$

Comme nous avons pu les remarquer dans ces trois différents exemples, la courbe épouse très bien les points et nous obtenons un résultat beaucoup plus précis que si nous avions juste tiré un trait entre chaque points (plus ou moins le rôle de l'ANN). De plus, cela nous demande très peu de ressources systèmes. C'est donc cette solution qui est retenue pour estimer les courbes dans ce projet de diplôme.

6. Notre solution : Prédiction pour un réseau de senseurs

Comme nous l'avions vu lors du pré projet de diplôme [7], l'UCLA (university of California, Los Angeles) a publié un article sur l'auto calibration de réseau de senseurs [19]. Cet article explique que le gros problème dans la détection automatique d'erreurs est le manque de connaissances sur les différents stimuli contre lesquels les sondes devraient être calibrées.
L'algorithme développé présente donc une sorte de solution à ce problème. Cette solution exploite la notion de « redondance de mesures » (on calibre la sorties d'une sonde avec celle d'une autre et ainsi de suite).

En ce moment notre réseau de neurone nous permet de calculer la dérive de chaque senseur séparément mais pas d'auto calibrer tout le réseau.
Les chercheurs de UCLA ont donc développé un algorithme en deux phases que nous allons essayer d'intégrer à ce projet de diplôme pour permettre cette auto calibration générale.

Voici comment cet algorithme se découpe:

- La première phase emploie la corrélation des signaux voisins (c.-à-d. les sondes observant le même phénomène) pour dériver la fonction.
- La deuxième phase est un problème d'optimisation de la première mais cette fois non plus pour des couples de senseurs mais pour des groupes.

6.1. Première phase de l'algorithme

Voici le déroulement de l'algorithme :

1. Rassembler les données temporelles de façon synchronisée.
2. Peser chaque point de repères potentiel.
3. Filtrer les points de repères non pertinents.
4. Adapter une fonction de calibrage au jeu de données filtrées.

En traçant un graphique (Figure 30, image 1) représentant les signaux de nos deux senseurs, pendant un temps T, nous pouvons nous rendre compte de leur corrélation (calibration similaire).
Dans cet exemple, la tendance de la courbe est relativement proche mais les valeurs calculées ne sont pas identiques.

En traçant un nuage de point (Figure 30, image 2) représentant ces mêmes signaux et en dessinant la ligne de tendance, on peut se rendre compte de la corrélation ou la non corrélation des mesures de ces senseurs.

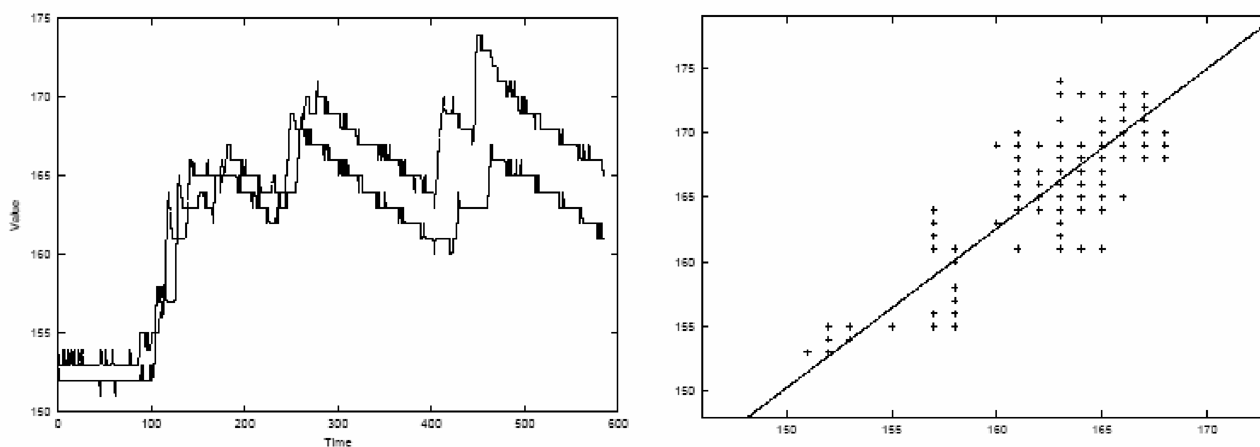

Figure 30 : La première image représente les signaux envoyés par deux senseurs placés au même endroit. La seconde image est une représentation en nuage de points de ces mêmes signaux.

Des mesures non corrélées rendraient impossible l'établissement d'une relation directe. Pour cette raison l'algorithme filtre les mesures correspondant aux périodes où les sondes ont observé des phénomènes différents (point 3 de l'algorithme). Pour ce faire, nous allons calculer pour chaque point du scatter plot la distance à la courbe de régression. Nous allons ensuite trier dans un ordre croissant ces distances et éliminer les x moins corrélées (donc les x plus grandes distances). Nous avons ainsi enlevé les points les moins pertinents et nous obtenons ainsi un ensemble de poids maximum.

Si maintenant nous retraçons un nuage de points avec l'ensemble de valeurs fiables que nous avons gardées, (Figure 31), nous pouvons nous rendre compte de l'amélioration apportée par cette première phase.

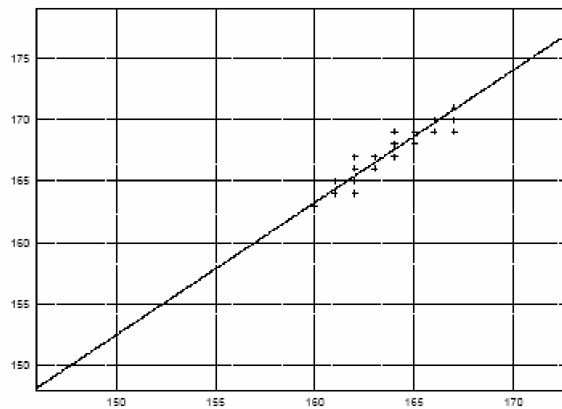

Figure 31 : Nuage de points représentant les valeurs fiables restantes

Il nous reste maintenant à procéder à l'adaptation de la fonction de calibration Fij. Ce procédé est semblable à la calibration d'usine, mais ici, les stimuli sont inconnus.

Remarque : La nature de cette fonction de calibration dépend du type de sondes utilisé.

6.1.1. Développement pratique, première partie de l'algorithme

a) Rassembler les données temporelles de façon synchronisée

Cet algorithme test la corrélation entre deux signaux voisins, il traite donc les signaux deux à deux.
Pour commencer nous allons à nouveau séparer notre matrice y en un certain nombre de courbes données (de x points chacune) puis la remplir avec des valeurs de progression et de bruit aléatoire (voir Figure 32).

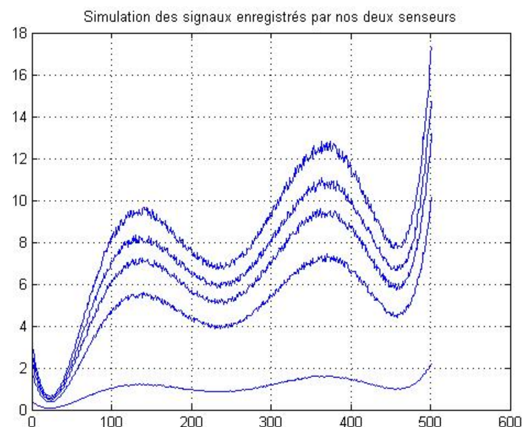

Figure 32 : Séparation de nos deux courbes en x points

Nous devons maintenant prendre deux à deux chaque signaux voisins.
Par exemple pour 3 senseurs :

- 1 avec 2
- 1 avec 3
- 2 avec 3

Nous devons veiller à ne pas parcourir les liens dans les deux sens (prendre 1 avec 2 mais pas 2 avec 1) et ne pas prendre deux fois le même senseur (ne pas prendre 1 avec 1).
Si nous sommes dans un de ces cas de figure, nous remplissons notre matrice Fij avec des 0

```
if (i==j)
      matriceFijA(i,j)=0;
      matriceFijB(i,j)=0;
%Nous calculons Fij dans un seul sense de déplacement
elseif (i>j)
      matriceFijA(i,j)=0;
      matriceFijB(i,j)=0;
```

A chaque tour de boucle nous allons donc obtenir deux vecteurs : X et Y.

X = (0.042, 0.169, 0.381, 0.678, 1.060, ..., 10.858, 12.258, 13.742, 15.312, 16.966)
Y= (0.010, 0.041, 0.092, 0.165, 0.257, ..., 2.641, 2.982, 3.343, 3.725, 4.127)

Avec ces deux vecteurs, nous pouvons afficher une représentation en scatter plot de ces deux courbes ainsi que la droite de régression ou droite de tendance
(voir Figure 33).

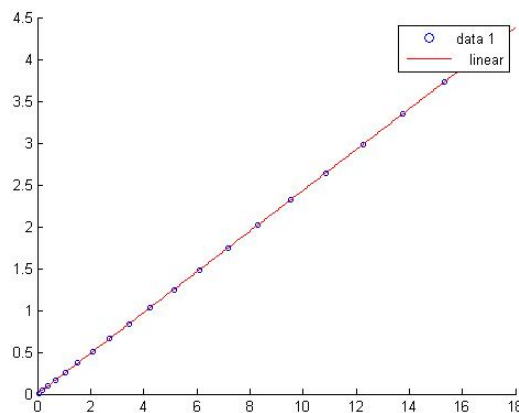

Figure 33 : Représentation en nuage de points de nos deux courbe

b) Peser chaque point de repères potentiel

Maintenant que nous avons nos deux vecteurs, nous pouvons calculer cette droite de régression, puis pour chaque points du vecteur X, prendre son équivalent au vecteur Y et calculer son éloignement à la courbe de régression.

```
%Affichage de la droite de régression linéaire
dtReg=polyfit(X,Y,1)

%Calcul de l'éloignement à la droite de régression
for c=1:nbPntsCal;
   pntY=Y(c);
   pntX=X(c);
   pntCourbe=(dtReg(1)*pntX);
   distance(c)=abs(pntCourbe-pntY);
end
```

c) Filtrer les points de repères non pertinents

Ces distances sont stockées dans un vecteur des distances qui est ensuite trié dans un ordre croissant.
Nous pouvons ensuite ne garder que les x plus corrélés, ce qui veut dire les x premier de notre vecteur des distances triées et pour finir tracer à nouveau une représentation scatter plot de ces points (voir Figure 34).

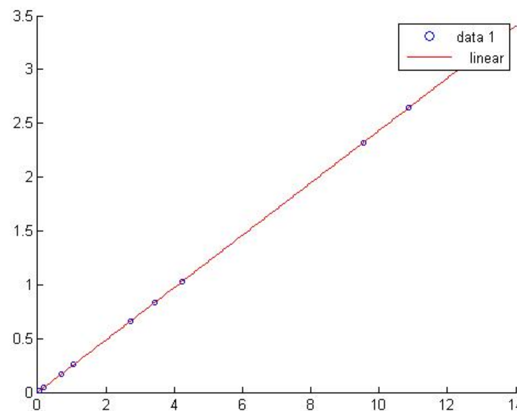

Figure 34 : Représentation en nuage de points les x valeurs les plus corrélées de nos deux courbes

d) Adapter une fonction de calibrage au jeu de données filtrées

Il ne nous reste plus qu'à calculer la nouvelle droite de régression qui est la fonction de calibration Fij et remplir nos deux matrices.

```
Fij=polyfit(newX,newY,1)
%Première partie de l'équation de la droite Fij (a de a*x+b)
matriceFijA(i,j)=Fij(1);
Première partie de l'équation de la droite Fij (a de a*x+b)
matriceFijB(i,j)=Fij(2);
```

Au final nous obtenons deux matrices Fij contenant pour la première la partie a de a*x+b et pour la seconde la partie b. Ces matrices Fij contiennent donc les fonctions de calibration pour passer d'un senseur à l'autre.
Si nous avons 3 senseurs, nous pouvons donc trouver les valeurs de la courbe du troisième senseur en partant du premier et passant par le deuxième senseur :
$F23(F12(x)) = a23 * (a12 * x + b12) + b23$

6.2. Seconde phase de l'algorithme

Cette seconde phase est en fait une optimisation de la première (maximisation de la consistance).

En raison des erreurs, le parcourt des fonctions de calibration (mappe une sortie x d'un senseur à un second senseur) par différents chemins peut donner plusieurs résultats différents pour un noeud donné.
Afin d'illustrer ce problème, on peut étudier le graphique de calibration (CG) représenté à la Figure 35. Un CG est une représentation graphique d'une matrice de calibration. Chaque sommet dans le CG représente un senseur et chaque lien représente les fonctions de calibration (CF) correspondantes.

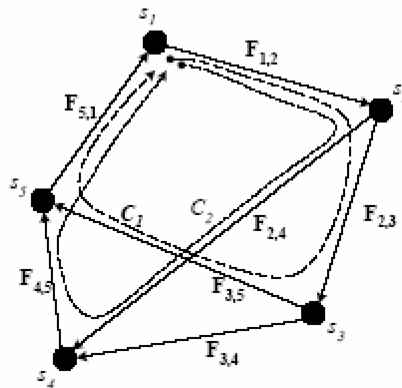

Figure 35 : représente un graphique de calibration (CG).
rem: on peut remarquer 2 cycles de calibration (C1 et C2) pour le nœud S1

Notre but ici est de calculer une nouvelle matrice de calibration Fij' (en maximisant l'uniformité) avec comme base la matrice de calibration Fij donnée.

Cette seconde phase n'a malheureusement pas été intégrée à ce projet de diplôme mais fera sûrement l'objet de prochaines investigations plus poussées.

7. Conclusion

Enormément de recherches ont été et sont réalisées dans le domaine des réseaux de senseurs et notamment sur l'auto-calibration. Malheureusement, personne n'a trouvé de solution miracle pour pouvoir limiter les erreurs engendrées par les senseurs et causées par le temps, les mauvaises conditions ou tout autres stimuli.

Il était très intéressant de pouvoir coupler deux études réalisées sur l'auto-calibration (L'ICIT de Ternopil et UCLA). La première concernait la prédiction du drift d'erreur d'un senseur et la seconde l'auto-calibration en utilisant la corrélation des signaux voisins.
En effet, si aucune solution miracle n'existe pour le moment, le couplage de deux solutions proposées pourrait déjà améliorer sensiblement les résultats et peut être même ouvrir une nouvelle direction de recherche.

A la fin de ce projet de diplôme, les deux solutions sont développées séparément. Il aurait fallu plus de temps pour pouvoir les coupler et peut être constater de meilleurs résultats. Mais nous pourrions très bien imaginer corriger les erreurs dues à des stimuli naturels ou temporels grâce à la prédiction de drift et ensuite les erreurs plus importantes dues à des dérèglements par la corrélation des signaux.

Une multitude de solutions pourraient être développées mais je pense qu'en vue des résultats actuels, ce couplage est très prometteur. En effet, autant le réseau de neurone développé pour la reproduction de la première solution (correction du drift d'un senseur) que l'algorithme permettant de trouver la matrice de calibration pour l'étude de UCLA a rendu de très bons résultats, rapides et réels.

Angélique Byrde

8. Références Bibliographiques et Webographiques

[1] « Réseaux de neurones, méthodologie et applications », Eyrolles, 2ème édition [2004] de Gérard Dreyfus, Jean-Marc Martinez, Mannuel Samuelides, Mirta Gordon, Fouad Badran, Sylvie Thiria, Laurent Hérault.
Cet Ouvrage de base qui fournit les éléments indispensables à la compréhension et la mise en œuvre des réseaux de neurones.

[2] « Neural Networks : A Comprehensive Foundation (2nd Edition) », Prentice Hall [1998] de Simon Haykin
L'une des meilleures références disponibles, couvrant de manière précise la plupart des concepts nécessaires à l'utilisation et à la compréhension des réseaux de neurones.

[3] « An introduction to neural networks », Eighth edition [1996] de Ben Kröse et Patrick Van Der Smagt.
Une introduction très complète sur les réseaux de neurones. http://neuron.tuke.sk/math.chtf.stuba.sk/pub/vlado/NN_books_texts/Krose_Smagt_neuro-intro.pdf

[4] « Réseaux de neurones artificiels », de Andrés Pérez Uribe (Heig-VD). Cours sur les réseaux de neurones, donné en option aux 3ème année de l'Heig d'Yverdon les bains.

[5] http://fr.wikipedia.org/wiki/R%C3%A9seaux_de_neurones
Page de l'encyclopédie en ligne « Wikipedia » sur le sujet des réseaux neuronaux artificiels.

[6] http://www.becoz.org/these/memoirehtml/ch06s04.html
Site interne expliquant clairement le Perceptron et l'Adaline.

[7] « Intelligent distributed sensor network », [1998] de A.Sachenko, V.Kochan, V.Tuchenko (ICIT de Ternopil).

[8] « Sensors signal processing using neural networks », [1999] de V. Golovko, L.Grandinetti, A.Sachenko, V.Kochan, V.Tuchenko (ICIT de Ternopil).

[9] « Intelligent nodes for distributed sensor network », [1999] de A.Sachenko, V.Kochan, V.Turchenko, V.Tymchyschyn, N.Vasylkiv (ICIT de Ternopil).

[10] « Sensor errors prediction using neural networks », [2000] de A.Sachenko, V. Turchenko, V. Kochan, V.Golovko, J.Savitsky, A.Dunets, T.laopoulos (ICIT de Ternopil).

[11] « Approach of an intelligent sensing instrumentation structure Development », [1999] de V. Golovko, L.Grandinetti, A.Sachenko, V.Kochan, V.Turchenko, V.Tymchyschyn, T. Laopoulos (ICIT de Ternopil).

[12] « Sensor drift prediction using neural networks », [2000] de A.Sachenko, V. Turchenko, V. Kochan (ICIT de Ternopil).

[13] « Technique of learning rate estimation for efficient training of MLP », [2000] de V. Golovko, L.Grandinetti, A.Sachenko, Y. Savitsky, T. Laopoulos (ICIT de Ternopil).

[14] « Estimation of computational complexity of sensor accuracy improvement algorithm based on neural networks », [2001] de A.Sachenko, V. Turchenko, V. Kochan (ICIT de Ternopil).

[15] « Error compensation in an intelligent sensing instrumentation system », [2001] de A.Sachenko, V. Turchenko, V. Kochan, R. Kochan, K. Tsahouridis, T.Laopoulos (ICIT de Ternopil).

[16] « The new method of historical sensor data integration using neural networks », [2001] de A.Sachenko, V. Turchenko, V. Kochan, T. Laopoulos (ICIT de Ternopil).

[17] « Approach to parallel training of integration historical data neural networks», [2001] de A.Sachenko, C. Triki, V.Turchenko (ICIT de Ternopil).

[18] « Machine Learning », [2006] de A.Byrde.
Rapport décrivant les différents « machine learning » et leurs applications.

[19] « A cllaborative approach to in-place Sensor calibration », [2002] de V.Bychkovskiy, S.Megerian, D.Estrin, M. Potkonjak (UCLA).