Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique
Laboratoire de Commande des Processus

THESE

présentée au **L**aboratoire de **C**ommande des **P**rocessus
en vue de l'obtention du titre de

Docteur en Sciences

en Automatique
par

Cherrad BENBOUCHAMA

Magister en Automatique de l'ENP

Thème

Contribution à l'implémentation des réseaux de neurones artificiels sur hardware reconfigurable FPGA.

Soutenue publiquement le Samedi 14 Juin 2008 devant le jury composé de:

D. BOUKHETALA	Professeur à l'ENP	Président
M. TADJINE	Professeur à l'ENP	Rapporteur
A. BOURIDANE	Reader, Queen's University, Belfast, Royaume Uni	Rapporteur
F. BOUDJEMA	Professeur à l'ENP	Examinateur
L. BARAZANE	Maître de Conférences à l'USTHB	Examinateur
L. SI MOHAMED	Maître de Conférences à l'EMP	Examinateur
K. BENMANSOUR	Docteur, CUYF, Médéa	Invité

Thèse préparée au sein du **L**aboratoire de **C**ommande des **P**rocessus (LCP) de l'ENP
10, avenue Pasteur BP 182 El-Harrach Alger
en collaboration avec le Laboratoire des **S**ystèmes **N**umériques de l'EMP
BP 17, Bordj-El-Bahri Alger

ملخص :

الأعمال التي عرضت في هذه الاطروحه ترتكز أساسا على تطوير منصة برمجيات للمساعدة فى دمج برنامج الشبكات العصبية الاصطناعية على دارات من نوع FPGA.نهجان صمما للتصدي للتطوير وللتصميم. الأول يقوم على استخدام لغة البرمجة VHDL في بيئة ISE Xilinx، والثانية تقوم على استعمال Handel-C مع DK Celoxica .تم الحصول على نتائج لتبرير الحلول المختارة لحل مختلف المشاكل مثل: تمثيل البيانات، الحسابات المستخدمة الخ... لإثبات هذا العمل صنعنا نظام تحكم بواسطة شبكات عصبية اصطناعية لقيادة محرك كهربائي ذو تيار مستمر. أخيرا، تم تصميم واجهه بيانية مخصصة للاستعمال السريع و السهل لمنصة البرمجيات.

مفاتيح : الشبكات العصبية, الإدماج, FPGA, التحكم.

Résumé :

Les travaux présentés dans cette thèse portent, essentiellement, sur la réalisation d'une plateforme logicielle pour l'aide à l'implémentation des réseaux de neurones et la génération automatique de configurations pour FPGAs. Deux approches ont été développées pour aborder le développement et la conception. La première repose sur l'utilisation du langage VHDL sous l'environnement ISE de *Xilinx*, et la deuxième est basée sur le Handel-C avec l'outil DK de *Celoxica*. Différents résultats de synthèse ont été obtenus pour justifier les solutions choisies concernant les différents problèmes tels que : la représentation des données, l'arithmétique utilisée, la mise en œuvre de la fonction d'activation, etc…Nous avons effectués des validations pratiques basées sur la commande d'un moteur à courant continu par les réseaux de neurones. Enfin, nous avons réalisé une interface graphique à menus, dédiée à l'utilisation rapide et conviviale de la plateforme logicielle.

Mots clés : Réseaux de neurones, Implémentation, FPGA, Commande.

Abstract :

The work presented in this thesis deals, essentially, with the realization of a high level tool for the implementation of artificial neural networks and the automatic generation of configurations for FPGAs. Two technical methods have been developed to approach the development and the design. The first one is based on the use of the VHDL language in the ISE Xilinx environment, and the second one is based on the Handel-C used with the DK Celoxica tool. Different synthesis results were obtained to justify the solutions chosen for the various problems such as: the data representation, the arithmetic used, the implementation of the activation function, etc… We have made some practical validations based on the neural networks DC motor control. Finally, we realized a graphical unit interface dedicated to the fast and the user-friendly employment of the software platform.

Key words : Neural networks, Implementation, FPGA, Control.

Je dédie cette thèse

A mes regrettés grands parents

A mes parents qui ont tant fait pour moi

A ma femme qui m'a supporté et soutenu

A mes enfants.

REMERCIEMENTS

Avant tout, je remercie Dieu, le tout puissant, pour m'avoir assisté et armé de patience afin d'accomplir ce modeste travail.

Je tiens à exprimer ma profonde gratitude à mon directeur de thèse le Professeur Mohamed Tadjine pour m'avoir fait l'honneur de diriger cette thèse. Qu'il trouve ici mes respectueux remerciements pour l'intérêt et le soutien sans relâche qu'il m'a toujours accordé.

Mes remerciements s'adressent aussi à Djamel Boukhetala, Professeur à l'ENP, pour avoir accepté de présider le jury.

Mes remerciements vont à Fares Boudjema, Professeur à l'ENP, Linda Barazane, Maître de conférences à l'USTHB, Lotfi Mokhtar Si mohamed, Maître de conférences à l'EMP, qui ont bien voulus me faire l'honneur d'être membres du jury.

Je remercie aussi, Khelifa Benmansour, Docteur du CUYF de Médéa pour nous avoir fait l'honneur d'accepter l'invitation.

Je tiens à remercier, Ahmed Bouridane, Reader à Queen's University de Belfast pour m'avoir fait l'honneur de co-diriger cette thèse et m'avoir accueilli au sein de son laboratoire Image Vision Systems (IVS) durant mes stages de recherche.

Je remercie les collègues et membres du laboratoire des Systèmes Numériques de l'EMP qui ont contribué dans ce travail, et en particulier Samir Sakhi dont une partie de ce travail a fait l'objet de sa thèse de magister.

Enfin, je remercie tous les membres de ma famille pour leurs soutiens, leurs encouragements et leurs patiences durant toute la durée qu'a pris cette thèse.

LISTE DES FIGURES

LISTE DES TABLEAUX

SOMMAIRE

CHAPITRE 0 INTRODUCTION GENERALE

CHAPITRE I ETAT DE L'ART SUR L'IMPLEMENTATION DES R.N.As ET GENERALITES

Première partie : Etat de l'art sur l'implémentation des R.N.As.

Deuxième partie : Généralités.

CHAPITRE II DEVELOPPEMENT ET CONCEPTION EN LANGAGE VHDL

CHAPITRE III IMPLEMENTATION DES R.N.As EN LANGAGE HANDEL-C

CHAPITRE IV INTERFACE GRAPHIQUE POUR L'IMPLEMENTATION DES R.N.As SUR CIRCUITS FPGA

Chapitre 0

INTRODUCTION GENERALE

Jules RENARD / Journal / Robert Laffont - Bouquins 1990

« Au travail, le plus difficile, c'est d'allumer la petite lampe du cerveau. Après, ça brûle tout seul. »

< 29 novembre 1901 p.557 >

Cette thèse traite du problème de l'implémentation matérielle (hardware) des réseaux de neurones artificiels (RNA) sur un circuit FPGA. Les RNA sont utilisés dans une grande variété d'applications comme l'identification et la commande des systèmes, l'approximation des fonctions, le traitement d'images, la synthèse de la parole, la reconnaissance des formes, etc…[Benb00] [Skrb99]

Les RNA peuvent être mis en œuvre soit sur des circuits électroniques, soit sur des calculateurs numériques. L'inconvénient des réalisations sur calculateurs numériques réside dans la lenteur de l'apprentissage. En effet, contrairement aux circuits électroniques, les processeurs fonctionnent d'une manière séquentielle et n'exploitent pas le parallélisme des RNA. De leur coté, les réalisations matérielles sur des circuits électroniques présentent l'inconvénient d'être dépourvues de souplesse. En effet, l'un des challenges majeurs, lors de la conception d'architectures neuronales est le choix de la topologie adéquate. Par conséquence, la mise au point d'un RNA nécessite l'essai de plusieurs architectures avant de choisir la bonne configuration, ce qui peut s'avérer coûteux lors de la conception d'une réalisation matérielle [Sakh05].

Toutes ces raisons font qu'on préfère l'utilisation de circuits reconfigurables, tels les FPGA (*Field Programmable Gate Arrays*). Ces circuits sont programmables comme des processeurs et en raison de leur fonctionnement en parallèle, ils possèdent des performances proches de celles des circuits matériels comme les ASIC (*Application Specific Integrated Circuits*) [Cox92]. Le premier avantage apporté par les circuits FPGA est la souplesse de la programmation, ce qui permet de multiplier les essais, d'optimiser de diverses manières l'architecture développée et de vérifier à divers niveaux de la simulation la fonctionnalité de cette architecture. Le second avantage des FPGA est la possibilité de la reconfiguration dynamique partielle ou totale des circuits, ce qui permet, d'une part, une meilleure exploitation du composant et une réduction de la surface de silicium employé, d'autre part la programmation en temps réel (quelques microsecondes) tout ou une partie du circuit [Mcke01].

Néanmoins, le temps nécessaire pour l'implémentation d'une architecture neuronale sur FPGA est de l'ordre de mois [Coe04]. De plus, la programmation de ces circuits nécessite la maîtrise de langages spécifiques tels que : VHDL (*Very-high-speed-integrated-circuit Hardware Description Langage*), ABEL et Handel-C, etc…, ce qui réduit considérablement leur utilisation à une grande échelle. L'idée était donc, de réaliser une plateforme logicielle qui ferait abstraction de la programmation des circuits FPGA et qui serait dédié à des

utilisateurs pas nécessairement initiés à la programmation de ces circuits. Ce qui aurait pour conséquence de réduire le temps de mise au point d'une architecture neuronale.

Le travail proposé consiste à développer une interface de haut niveau permettant la génération automatique de configurations optimales sur des circuits reconfigurables (FPGA) pour l'implémentation des réseaux de neurones.

Cependant, pour aboutir à cela il était nécessaire, d'une part, d'aborder certains problèmes liés aux RNA, au circuit reconfigurable ciblé, au langage de conception et à l'environnement de développement, d'autre part, de procéder à des validations pratiques afin de prouver la fonctionnalité de chaque approche développée. D'où l'intérêt pour l'utilisation des RNA dans la commande des systèmes. De cela découle des étapes intermédiaires par lesquelles nous devons inévitablement passer pour parvenir à l'objectif final qui n'est autre que la plateforme logicielle. Ces différentes phases sont comme suit :

- Le type de RNA et l'algorithme d'apprentissage à implémenter.
- Les problèmes liés à la mise en œuvre des RNA.
- Le choix du circuit FPGA à cibler.
- La précision.
- L'arithmétique à utiliser.
- L'optimisation des ressources et de la fréquence de travail.
- La simulation.
- La conception de l'interface graphique.
- La communication avec la carte à FPGA.
- Les validations pratiques.

Ce rapport est organisé comme suit :

Le chapitre 0 est consacré aux motivations et objectives de ce travail de recherche.

Dans le chapitre I, nous exposons les différents travaux effectués dans le domaine de la mise œuvre des RNA sur circuits FPGA, avec les différents types d'implémentations. Nous insisterons, essentiellement, sur les différentes architectures de RNA implémentés, ainsi que les différentes représentations des données réelles utilisées. En effet, le choix de cette représentation est probablement l'une des plus importantes décisions à prendre lors de la conception d'une architecture neuronale.

Par la suite, nous présentons les RNA comme étant une structure de calcul hautement parallèle, capable d'apprendre et de s'adapter, et donc très efficace dans son utilisation pour la

commande des systèmes. Nous abordons le circuit ciblé par l'implémentation, le FPGA, avec son architecture et les différentes méthodes de sa programmation, ainsi que les différents langages et environnements utilisés.

Deux approches ont été développées pour aborder le développement et la conception. La première repose sur l'utilisation du langage VHDL sous l'environnement ISE de *Xilinx*, et la deuxième est basée sur le Handel-C avec l'outil DK de *Celoxica*.

Le chapitre II est consacré à la première approche dans laquelle sont cités et traités les différents problèmes qu'on rencontre, durant la conception d'une architecture neuronale, tels que la représentation des données, le bloc d'activation et le type d'implémentation. Les résultats obtenus sont présentés, ainsi qu'une application pour valider la démarche suivie.

Le chapitre III traite la deuxième approche qui consiste, dans un premier temps, à programmer l'algorithme de la rétropropagation en langage C, réaliser le « portage » du programme conçu en langage C vers le langage Handel-C, et enfin l'implémenter sur le circuit FPGA ciblé. Les résultats obtenus sont présentés, ainsi qu'une deuxième application pour valider la démarche suivie.

Enfin, dans le chapitre IV, nous présentons l'interface graphique à menus réalisée et dédiée à l'utilisation rapide et conviviale de la plateforme logicielle. La plateforme logicielle conçue a été utilisée pour quelques applications pratiques.

Chapitre I

ETAT DE L'ART SUR L'IMPLEMENTATION DES RESEAUX DE NEURONES ARTIFICIELS ET GENERALITES.

Première partie : Etat de l'art sur l'implémentation des réseaux de neurones artificiels.

Ernest RENAN /
L'Avenir de la science, Pensées de 1848 (1890) / GF 765 Flammarion 1995

« Tout est fécond excepté le bon sens. » < p.434 >

I.1 Introduction

Selon les performances requises par l'application, les réseaux de neurones artificiels (RNA) peuvent être implémentés soit en software sur un ordinateur conventionnel, soit à l'aide de processeurs DSP (*Digital signal Processors*), soit en VLSI (*Very Large Scale Integration*) sur circuits en silicium (circuits dédiés ASIC, cellules standards, FPGA), ou en combinant ces techniques [Ferr94].

Dans cette partie, nous présenterons les différents types d'implémentations de RNA, et détaillerons les différents travaux effectués dans le domaine de l'implémentation de RNA sur circuits FPGA. Nous insisterons, essentiellement, sur les différentes architectures de RNA implémentés, ainsi que les différentes représentations des données réelles utilisées.

I.2 Implémentation logicielle

L'implémentation logicielle est définie comme étant la programmation d'une méthode d'apprentissage d'un réseau de neurones, sur un ordinateur avec un langage évolué (C, Matlab, etc..).

Une bonne partie de l'effort d'implémentation logicielle est en fait consacrée à l'interface homme machine. En effet, différents types de présentations, numériques et graphiques permettent de donner à l'utilisateur un aperçu clair et rapide du comportement et des performances du RNA, ainsi que de l'algorithme d'apprentissage appliqué. Cependant ce type d'implémentation présente plusieurs inconvénients :

- Il n'exploite pas le parallélisme des RNA, ce qui induit des temps d'apprentissage relativement lents.
- La taille des RNA peut générer des contraintes dans des applications en temps réel.

Toutes ces raisons font que les implémentations logicielles des réseaux de neurones artificiels ne sont pas toujours bien adaptées aux applications exigeant une réponse rapide ou des contraintes en temps réel (contrôle, traitement de la parole, etc..), ce qui rend inévitable l'utilisation des implémentations matérielles.

Remarque :

En ce qui concerne la vitesse d'exécution, les ordinateurs digitaux actuels travaillent avec des cycles d'horloge de quelques nanosecondes, alors qu'un neurone biologique possède un temps de réponse de l'ordre de la milliseconde. Cet apparent manque de vitesse dans un

réseau de neurones biologiques est compensé par le très grand nombre de neurones le constituant ; à titre d'exemple, un cerveau humain est un réseau parallèle composé d'environ 10^{11} à 10^{12} neurones ayant chacun 10^{3} synapses.

I.3 Implémentation matérielle

La première implémentation matérielle d'un RNA est due aux travaux de Minsky et Edmonds qui assemblèrent en 1951 à l'aide de 300 tubes à vide et d'un pilote automatique de l'avion bombardier B24, une machine baptisée SNARC, qui simulait l'évolution d'un rat dans un labyrinthe [Ligo98]. Depuis, plusieurs mises en œuvre matérielles de RNA ont été réalisées par de grandes entreprises comme IBM et Intel.

Les implémentations matérielles permettent une meilleure prise en compte du parallélisme des réseaux de neurones et peuvent ainsi s'exécuter plus vite que les simulations logicielles [Lind95]. On distingue deux approches dans les implémentations matérielles des réseaux de neurones artificiels [Izeb99] :

- Les implémentations qui intègrent la phase d'apprentissage et la phase de test/généralisation dans un même circuit, d'où le terme anglais *"on-chip training circuits"*. Ce type d'implémentation permet une flexibilité et une adaptabilité du circuit à plusieurs applications.
- Les implémentations qui intègrent seulement la phase de généralisation. Dans cette approche, l'apprentissage est réalisé en software afin de générer les poids synaptiques. L'implémentation hardware du RNA consiste dans ce cas, à charger ses poids synaptiques dans des mémoires et à implémenter les fonctions de sommation et d'activation, d'où le terme anglais *"off-chip training circuits"*.

Selon la technologie de réalisation nous sommes amenés à distinguer trois (3) types principaux d'implémentations matérielles de RNA : L'implémentation analogique, l'implémentation digitale et l'implémentation hybride (analogique / digitale).

I.3.1 Implémentation analogique

Les opérations de base exécutées par un RNA et qui sont la multiplication, l'addition et la fonction d'activation non linéaire, peuvent être facilement réalisées par des circuits analogiques très simples, c'est pour cette raison qu'ils ont été utilisés pour l'implémentation de RNA.

Parmi les circuits neuronaux analogiques, nous pouvons citer le circuit ETANN (*Electrically Trainable Analog Neural Network*) d'Intel, qui fut le premier circuit analogique commercial. Il peut implémenter jusqu' à 64 neurones et 10 240 connexions [Sakh05]. Les poids sont directement chargés dans le circuit, l'apprentissage devant être effectué en *Off-chip*. Plusieurs applications qui ont utilisé ETANN avec succès, ont montré sa fiabilité.

L'inconvénient majeur de ce type d'implémentation réside dans le manque de précision ; de plus, les décalages de tension pouvant avoir lieu dans les implémentations analogiques peuvent corrompre l'apprentissage d'une façon significative. D'autres problèmes se posent aussi, tels que le stockage analogique des poids synaptiques des neurones, la nécessité de compenser les variations de température, le contrôle de la tension d'alimentation dont la résolution n'est pas toujours évidente et fait appel à des outils de conception et de fabrication sophistiqués. D'où la nécessité de recours aux implémentations digitales [Sakh05].

I.3.2 Implémentation digitale

Par rapport à la technologie analogique, la technologie digitale possède les avantages suivants :

- Facilité d'obtention d'une excellente précision.
- Stockage des poids dans des RAM.
- Possibilité d'implémenter un grand nombre de neurones identiques pouvant travailler en parallèle sur un seul circuit.

Parmi les implémentations digitales connues nous pouvons citer le circuit NISP *(Neural Instruction Set Processor)* et le circuit N64000 d'Inova, qui possèdent des architectures à base de processeurs. Parmi les implémentations à base de DSP, nous pouvons citer *System Professionnel BrainMaker* de *California Scientific Software*. A noter qu'il existe aussi des architectures à base de circuits ASIC comme le Système *MY-NEUPOWER* d'Hitachi.

I.3.3 Implémentation hybride

Les points forts des implémentations digitales résident surtout dans leur précision et leur flexibilité, tandis que les implémentations analogiques donnent lieu à des circuits très denses, permettant une architecture complètement parallèle pour des réseaux de grande taille, ce qui se manifeste par des vitesses de traitement élevées.

Pour cela, les chercheurs ont essayé de combiner ces deux approches, sous forme d'implémentations hybrides et ceci dans le but d'exploiter les avantages et d'annihiler les

inconvénients de chaque type d'implémentation. La structure du réseau est en analogique (entrées, sorties et traitements intermédiaires), tandis que le processus d'apprentissage, quant à lui, est implémenté en digital [Sakh05]. Parmi ces implémentations hybrides, nous pouvons citer le circuit ANNA *(Artificial Neural Network ALU*) d'AT & T.

I.4 Implémentation de RNA sur circuits FPGA

Les circuits FPGA présentent la caractéristique de reconfigurabilité qui est fort intéressante pour l'adaptation des poids et de l'architecture du réseau de neurones. Un autre avantage de ces circuits est qu'ils combinent la souplesse de la programmation avec la vitesse d'opération du matériel.

L'implémentation d'un réseau de neurones sur FPGA est une tâche qui a stimulé les recherches afin de trouver le meilleur compromis entre la taille des ressources et la qualité du réseau de neurones matériel. En effet, les RNA utilisent l'opération de multiplication, ce qui a pour effet de consommer les ressources des circuits digitaux en général et des FPGA en particulier [Zhu03]. Plusieurs stratégies on été développées afin d'implémenter des RNA sur FPGA d'une manière efficace, tout en optimisant les ressources du circuit. Le progrès continu de la microélectronique est la force motrice de ce développement. L'apparition de nouveaux circuits avec une plus grande capacité d'intégration permet d'élaborer des architectures toujours plus denses fonctionnant à des vitesses toujours plus élevées.

La première implémentation d'un réseau de neurones sur FPGA a été réalisée par Cox & al dans le cadre du projet GANGLION [Cox92]. C'est une architecture à base d'un RNA récurrent à connexion totale, implémenté sur une plateforme baptisée SUVME à base du circuit XC3090 de *Xilinx*. Elle a été appliquée dans le domaine de la segmentation d'images en temps réel et a été utilisée dans le domaine de la vision, notamment l'inspection industrielle. D'autres travaux ont suivi depuis, pour l'implémentation de différents types d'architectures de RNA et différentes représentations de données réelles.

I.5 Les différentes architectures de RNA implémentées sur FPGA

I.5.1 Les réseaux multicouches

L'algorithme le plus utilisé pour l'apprentissage de RNA multicouches est celui de la rétropropagation. Eldredge avait réussi, en 1994, la première implémentation de cet algorithme sur une plateforme baptisée RRANN (*Runtime Reconfiguration Artificial Neural Network*) construite autour du circuit FPGA XC3090. L'auteur avait démontré que

l'architecture développée pouvait apprendre à approximer les centroïdes des sous-ensembles flous [Eldr94].

Ferrucci et Martin avaient conçu une plateforme multi-FPGA baptisée *Adaptive Connectionist Model Émulateur* (**ACME**) qui se compose de quatorze (14) circuits FPGA type *Xilinx* XC4010. La plateforme ACME avait été validée avec succès par l'implémentation d'un réseau d'architecture [3, 3, 1] (3 entrées, 3 neurones cachés, et une sortie) qui peut apprendre la fonction non linéaire XOR [Ferr94], [Mart94].

H. Ossoing, en 1996, avait aussi implémenté l'algorithme de rétropropagation sur FPGA. Il avait discuté du parallélisme en nœud (*node parallélism)* qui requiert un multiplicateur par neurone et effectue les opérations de multiplication et d'accumulation en parallèle, pour tous les neurones. Il avait implémenté un RNA d'architecture [3, 3, 1] sur quatre (4) circuits Xilinx 4013 et un circuit Xilinx 4005 [Osso96].

J. Beuchat et Al, en 1998, ont été lourdement influencés par les travaux d'Eldredge. Ils ont mis au point une plateforme baptisée RENCO à base de quatre FPGA pilotés par microprocesseur. Les auteurs ont souligné la difficulté de développement d'une architecture totalement parallèle, ce qui induisait un gaspillage en ressources et en vitesse d'exécution, surtout que l'architecture nécessitait un multiplicateur pour chaque synapse. De là une approche temps-multiplexage a été développée et qui a donné un bon compromis entre la vitesse et l'optimisation des ressources. Elle a été appliquée avec succès à la reconnaissance des caractères écrits à la main [Beuc98].

V. Pandya, en 2005, avait implémenté l'algorithme de rétropropagation en langage Handel-C, et élaboré une architecture partiellement parallèle et une autre totalement parallèle, qui se sont avérés respectivement 2 fois et 4 fois plus rapides qu'une architecture séquentielle [Pand05].

I.5.2 Les réseaux ART

Le problème principal de la plupart des modèles de RNA est le manque de connaissances à priori pour déterminer le nombre de couches, le nombre de neurones par couche et comment ils seront reliés ensemble. Les réseaux neurologiques ontogéniques [Fies94] visent à surmonter ce problème, en offrant la possibilité de changer dynamiquement la topologie des RNA. Les approches ART (*Adaptive Resonance Theory*) et GAR (*Grow And Represent*) font partie de ce type de réseaux. La plateforme réalisée par A. Perez-Uribe, pour l'implémentation du réseau ART, été baptisée FAST (*Flexible Adaptable Size Topology*) [PU96].

L'auteur a conclu, à l'issue de ses travaux, que l'architecture FAST ne généralisait pas aussi bien que la rétropropagation mais pouvait apprendre plus rapidement et d'une manière plus efficace. L'architecture FAST a été la première de son genre à utiliser l'apprentissage non supervisé, néanmoins elle a montré ses limites, du temps qu'elle a été seulement appliquée pour résoudre des problèmes de jouets (*toy problems*).

I.5.3 Les réseaux de Hopfield

D. Abramson, en 1998, avait implémenté un réseau de Hopfield sur FPGA, dans le but de résoudre le problème classique des N reines. Le but du problème est de placer un nombre N de reines sur un échiquier de N x N, dans des positions de non attaque mutuellement. C'est un *benchmark* pour les problèmes d'ordonnancement.

La performance de cette implémentation avait été comparée à une implémentation logicielle en langage C. Le résultat obtenu indiquait qu'il était possible de gagner entre 2 à 3 ordres de grandeur la vitesse avec la réalisation matérielle [Abra98].

I.5.4 Les RNA probabilistes

M. Figueiredo, en 1998, avait implémenté sur FPGA, une architecture baptisée PNN (*Probabilistic Neural Network*), pour la classification des images multi spectrales. Le temps nécessaire à l'exécution de l'algorithme écrit en langage C, sur une machine DEC à 200MHz et une machine Pentium à 166MHz ont été 22 et 30 minutes respectivement. En employant la plateforme réalisée, baptisée X213, le temps d'exécution de l'application avait été réduit à 77 s [Figu98].

I.5.5 Les réseaux RBF

M. Skrbek, en 1999, avait utilisé la fonction XOR comme *benchmark* pour valider une plateforme baptisée ECX, sur laquelle il avait implémenté un réseau de neurones RBF (*Radial Basis Function*). Son travail avait été utilisé dans le domaine de la reconnaissance des formes et appliqué à l'identification des chiffres et du signal sonar [Skrb99].

I.5.6 Les RNA modulaires

Les architectures REAMP-α et REAMP-β développées par T. Norstrom ont été utilisées comme support pour l'implémentation de différents algorithmes d'apprentissage de RNA sur une plateforme multi-FPGA XC4025 [Nord95].

La plate forme REMAP a été utilisée en tant que simulateur de RNA matériels et aussi en tant qu'outil d'enseignement. La plateforme conçue est utilisée pour implémenter les RNA suivants [Nich03] :

- Réseaux Adaline et Madaline.
- Algorithme de rétropropagation.
- Réseaux de mémoires associatives bidirectionnelles.
- Réseaux de mémoires distribuées clairsemées.
- Mémoires associatives de Hopfield.
- SOM (*Self-organizing Maps*).
- ART (*Adaptive Resonance Theory*).

I.5.7 Les RNA évolutionnaires

Garis et al, en 2001, avaient implémenté un RNA basé sur les techniques évolutionnaires. Largement influencé par le projet CAM de MIT (*Massassuchests Institut of Technology*), Garis avait construit une plate forme baptisée CBM (*CAM-Brain machine*) sur laquelle, un algorithme génétique était utilisé pour faire évoluer un RNA. Le CBM a été appliqué avec succès dans des applications de prédiction et d'approximation de fonctions. Elle a été utilisée aussi dans les applications de commande [Gari01].

G. Earl, en 2004, avait élaboré une architecture basée sur les algorithmes génétiques, qui étaient utilisées pour faire évoluer l'architecture neuronale. Il avait réalisé une plateforme baptisée HEDANN (*Hardware-Evolved Digital Artificial Neural Networks*). C'est un outil d'aide à la recherche pour explorer les architectures complexes de RNA se rapprochant des « cerveaux artificiels » [Earl04]

I.6 La représentation des données

Les performances des RNA dépendent largement de la précision des données, en particulier l'algorithme de rétropropagation [Nord98]. En effet, une représentation trop limitée augmente l'erreur de quantification des poids du RNA et par conséquence, peut causer une déviation du gradient de sa trajectoire désirée. De ce fait, le choix de la représentation des données est probablement l'une des plus importantes décisions à prendre lors de la conception d'une architecture neuronale.

Certaines recherches avaient montré qu'on pouvait réaliser l'apprentissage des RNA avec des poids entiers. L'intérêt de l'emploi des poids en nombres entiers, provient du fait que les multiplicateurs entiers peuvent être mis en application plus efficacement que ceux en virgule flottante ou en virgule fixe. Le réseau de Hopfield implémenté par D. Abramson (cf§ II.5.3), présentait des poids représentés en nombres entiers. Ceci avait réduit d'une manière significative la taille des unités de calcul.

Il y a eu également l'emploi d'algorithmes d'apprentissage qui employaient des nombres entiers en puissance deux comme poids [Zhu03]. L'intérêt dans ce type de représentation est que l'utilisation de cette architecture ramène les opérations de multiplication à des opérations de décalage, ce qui réduit d'une manière significative les ressources employées. Néanmoins, les représentations des données réelles les plus répandues sont :

- La représentation en train d'impulsions (*Pulse Stream)*.
- La virgule flottante.
- La virgule fixe.

I.6.1 La représentation en train d'impulsions

Le principe de ce type de représentation est que les valeurs des données sont représentées par la fréquence des impulsions restreintes (*frequency of narrow constant width pulses*) [Lysa 94]. Les signaux codés par cette méthode peuvent être multipliés et sommés dans chaque nœud, en utilisant une simple porte logique. La figure (1.1) montre un exemple de représentation de données et d'opérations de multiplication et d'addition [Earl04].

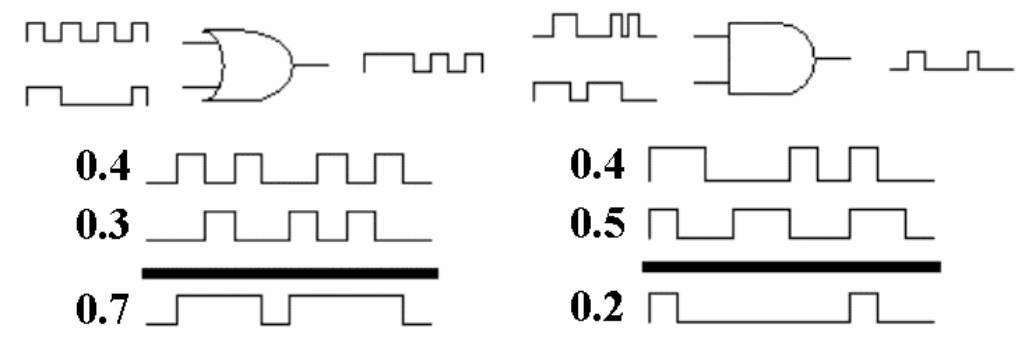


Figure 1.1 : *Exemples d'opération arithmétiques avec la représentation pulse stream*

Ce type de représentation avait été aussi utilisée dans la construction de CAM-Brain, il avait permis de réduire la consommation des ressources, bien que la précision obtenue n'était pas très bonne [Pand05]. Parmi les travaux effectués par cette méthode, nous pouvons aussi citer S. Bade [Bade94] et P. Lysaght [Lysa 94] en 1994.

L'inconvénient de ce type de représentation est son manque de précision, ce qui réduit d'une manière significative les capacités d'apprentissage du réseau de neurones [Pand05].

I.6.2 La représentation en virgule flottante

C'est un type de représentation répandu dans les implémentations sur processeurs, il procure une bonne précision dans le traitement et par conséquence, une grande souplesse pour la convergence. Certaines tentatives ont été faites, pour l'implémentation de RNA sur circuits FPGA en virgule flottante [Zhu03] [Nich03]; Cependant, aucun succès significatif n'a été rapporté.

L'étude menée par [Sakh05] a montré que pour une architecture de RNA avec un neurone dans les couches d'entrée et de sortie, et trois (3) neurones dans la couche cachée, implémentée sur le composant XC 2v1000-fg456-4 de *Xilinx*, on obtenait un large dépassement des ressources du circuit FPGA (tableau 1.1) :

Ressources	**Consommation**
Nombre I/O	96 / 324 (29%)
Nombre Slices	14,519 / 5,120 (**283%**)*
Nombre Mult 18x18s	84 / 40 **(210%)***
LUTs	28,955 / 10,240 (**282%**)*

Tableau 1.1 : *Ressources pour l'implémentation on-chip learning en format virgule flottante.*

Nous en concluons, qu'au stade de développement actuel des FPGA, la représentation en virgule flottante, bien que garantissant de meilleures performances concernant l'aspect de convergence, est gourmande en ressources que les circuits disponibles actuellement sur le marché ne peuvent pas fournir.

I.6.3 La représentation en virgule fixe

C'est le type de représentation le plus utilisé dans les travaux effectués sur l'implémentation de RNA sur circuits, car il représente le meilleur compromis entre la consommation des ressources, la fréquence et la précision [Nich03].

Pour les RNA multicouches dont l'apprentissage se fait par la rétropropagation de l'erreur, [Holt91] avait monté qu'une représentation sur 16 bits en virgule fixe était le minimum pour que l'apprentissage se fasse correctement, tout en supposant qu'on utilise des entrées normalisées (entre 0 et 1) et qu'on emploie la fonction sigmoïde pour l'activation.

K. Nichols avait montré aussi l'avantage indéniable d'une représentation en virgule fixe sur 32 bits face à une représentation en virgule flottante sur la génération de circuits *Xilinx* 4020E [Nich03].

J.G. Eldredge dans l'architecture qu'il a développée, avait utilisé plusieurs combinaisons de la taille des données en virgule fixe afin d'optimiser la rapidité de la convergence et la qualité de la généralisation, surtout avec les circuits FPGA disponibles à l'époque. Dans les conclusions de ses travaux, il avait recommandé, en fonction des composants disponibles, l'utilisation d'une représentation sur 32 bits qui procurerait une meilleure convergence [Eldr94]. C'était aussi les conclusions de M. Taveniku concernant les futures versions du REAMP destinées à être implémentées sur circuits ASIC [Tave95].

Les architectures développés par A. Perez-Uribe [PU96] ont utilisé une représentation en virgule fixe sur 8 bits, avec des données dans l'intervalle [0,1].

La plateforme ACME de A. Ferrucci et M. Martin [Ferr94], [Mart94], était basée sur une représentation en virgule fixe sur 8 bits, afin de réaliser la convergence de l'algorithme de rétropropagation pour le problème du XOR logique.

M. Skrbek avait aussi utilisé une architecture pour la résolution du problème du XOR, la plateforme ECX qu'il a construit était dédiée à l'implémentation de l'algorithme de la rétropropagation et l'algorithme RBF (*Radial Basis Function*), en utilisant une architecture sur 24 bits [Skrb99].

I.7 Conclusion

Dans cette partie, nous avons exposé les différents types de RNA implémentés sur circuits FPGA, depuis la première publication en 1992. Nous avons discuté aussi des différentes représentations des données réelles et nous avons mis en évidence l'avantage

d'une représentation en virgule fixe, qui présente un compromis entre la précision et la taille du RNA réalisé et son coût en ressources.

Les axes de recherches actuels dans ce domaine incluent [Pand05] :

- Les *benchmarks* qui devrait être créés pour comparer et analyser les performances des différentes implémentations de RNA.
- Les outils logiciels et librairies, qui sont nécessaires pour faciliter les différentes implémentations.
- Les différentes méthodes qui continuent à être développées afin d'optimiser le fonctionnement des algorithmes d'apprentissage des RNA, notamment en ce qui concerne la parallèlisation.
- Le développement d'architectures qui tirent profit des dispositifs des plus récents FPGA tels que les multiplicateurs et les unités spécialisés.

Dans la partie suivante, nous présenterons des généralités sur les RNA et leur implémentation sur les circuits FPGA.

Chapitre I

ETAT DE L'ART SUR L'IMPLEMENTATION DES RESEAUX DE NEURONES ARTIFICIELS ET GENERALITES.

Deuxième partie : Généralités.

Paul VALÉRY /
Mauvaises pensées et autres / OEuvres II / Bibliothèque de la Pléiade / nrf Gallimard 1960

« Qui veut faire de grandes choses doit penser profondément aux détails. »

< p.893 >

I.1 Introduction

Cette partie est consacrée à une présentation générale des RNA et leur utilisation dans la commande des systèmes, des circuits FPGA, des langages VHDL et Handel-C, ainsi que des outils de conception ISE de *Xilinx* et DK de *Celoxica*. Tout d'abord, nous présenterons le neurone formel qui est l'élément de base des RNA et nous en étudierons quelques caractéristiques. Nous exposerons, par la suite, les principaux types de RNA et nous donnerons un aperçu de quelques domaines d'application des réseaux de neurones. Nous présenterons, ensuite, les circuits FPGA avec leur architecture ainsi que les différentes méthodes de leur programmation, le langage VHDL et l'outil de conception ISE de *Xilinx*, le langage Handel-C et l'outil de conception DK de *Celoxica* et nous terminerons par une comparaison avec le langage VHDL.

I.2 Les réseaux de neurones artificiels

Les réseaux de neurones artificiels (RNA) sont des outils de calcul massivement parallèles, inspirés par l'étude du système nerveux. Leur fonctionnement imite celui des réseaux de neurones biologiques, en reproduisant leurs caractéristiques de base. Ils s'inscrivent dans cette vague de recherches, où l'effort de conception est mis sur le développement d'algorithmes d'apprentissage capables de doter les machines d'autonomie et de capacité d'adaptation. Parfois, ces systèmes, dits intelligents, arrivent même à « découvrir » de nouvelles solutions à des problèmes forts complexes et difficilement accessibles pour un cerveau humain.

I.2.1 Le neurone formel

Le neurone formel qui constitue l'élément de base d'un RNA est une unité de calcul, dont le modèle s'inspire de celui d'un neurone biologique. On peut le décrire par les éléments suivants [Touz92] :

- Son état appelé aussi activation.
- Ses connexions d'entrée auxquelles sont associés des poids.
- Sa fonction d'entrée réalisant un prétraitement (généralement une somme pondérée).
- Sa fonction d'activation (ou de transfert), qui calcule l'activation du neurone à partir du résultat de la fonction d'entrée.

Le modèle mathématique d'un neurone artificiel (figure 1.2) est donné par les équations suivantes:

$$u_k = \sum_{j=1}^{n} w_{kj} e_j \qquad \text{et} \qquad y_k = f(u_k - \theta_k) \qquad (1.1)$$

Où :

$e_1, e_2, ..., e_n$ sont les entrées,

$w_{k1}, w_{k2}, ..., w_{kn}$ sont les poids synaptiques du neurone k,

u_k est la sortie de l'unité de sommation,

θ_k est le seuil,

y_k est le signal de sortie du neurone k,

$f(.)$ est la fonction d'activation,

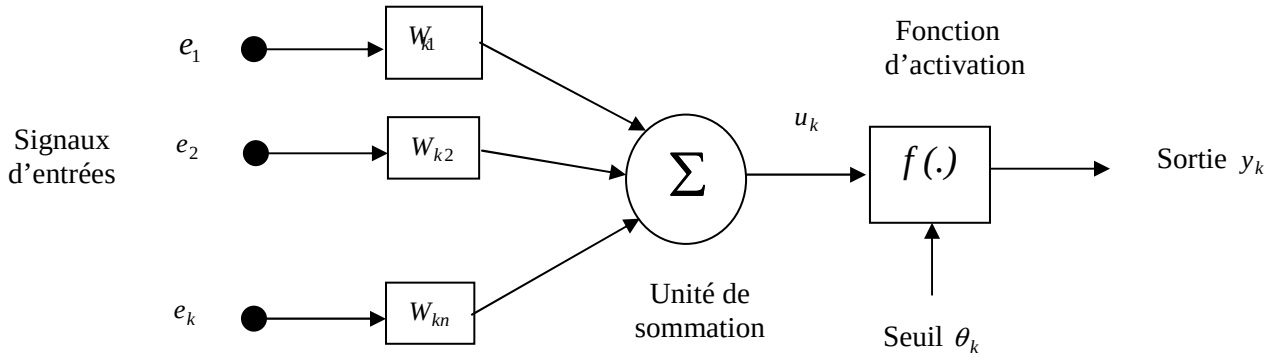


Figure 1.2: *Modèle Mathématique du neurone artificiel.*

L'utilisation d'une fonction d'activation non linéaire permet de modéliser des équations dont la sortie n'est pas une combinaison linéaire des entrées. Cette caractéristique confère au RNA de grandes capacités de modélisation fortement appréciées pour la résolution de problèmes non linéaires. Il existe plusieurs types de fonctions d'activation, elles sont illustrées par la figure 1.3 :

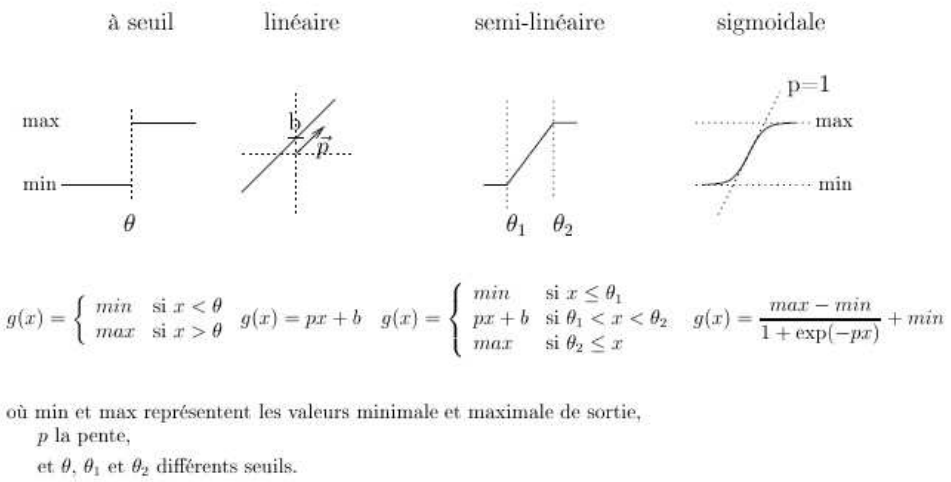


Figure 1.3*: Les différents types de fonctions d'activation.*

I.2.2 Caractéristiques des RNA

Les caractéristiques qui permettent aux réseaux de neurones artificiels d'être attractifs sont l'apprentissage, la généralisation et la capacité d'abréger [Outa04].

I.2.2.1 L'apprentissage

C'est la qualité la plus importante et remarquable des RNA. On entend par apprentissage, la modification automatique des poids des connexions ou du nombre et de l'organisation des neurones, afin d'adapter le traitement effectué par le réseau à une tâche particulière [Gaut99]. Un ensemble de règles bien définies, permettant de réaliser un tel processus d'adaptation des poids, constitue ce qu'on appelle l'algorithme d'apprentissage du réseau.

Plusieurs algorithmes ont été développés depuis la première règle d'apprentissage de Hebb en 1949. Ils sont divisés en trois classes: supervisé, non supervisé et par renforcement.

Parmi ces algorithmes d'apprentissage, le plus utilisé est l'algorithme de la rétropropagation du gradient.

Algorithme de la rétropropagation du gradient

L'algorithme de la rétropropagation s'applique en deux étapes. La première est le *forwad propagation*, durant laquelle l'excitation X_p est appliquée à la couche d'entrée et se propage, en avant, dans le réseau pour calculer la sortie O_p, et l'erreur $(Y_p - O_p)$ par rapport la sortie désirée Y_p. Durant la deuxième phase - *backward propagation* - cette erreur se propage

en arrière pour calculer l'erreur pour chaque neurone, et effectuer des changements appropriés des poids du réseau.

Cet algorithme est donné pour un réseau (N, L, M) à N neurones dans la couche d'entrée, une seule couche cachée à L neurones, et M neurones dans la couche de sortie. L'apprentissage se fait pour N_p exemples de couples Entrée/Sortie $\lfloor X_p, Y_p \rfloor$ avec un taux η.

L'algorithme est donné comme suit :

Forward propagation

1. Initialisation des poids w^c, w^s.
2. Application du vecteur d'entrée $X_p = (x_{p1}, x_{p2}, ..., x_{pN})$ à la couche d'entrée.
3. Calcul des états des neurones de la couche cachée $net^c_{pj} = \sum_{i=1}^{N} w^c_{ji} x_{pi}$ (1.2)
4. Calcul des sorties des neurones de la couche cachée par la fonction d'activation $f^c(.)$:

$$i_{pj} = f^c_j(net^c_{pj}) \quad (1.3)$$

5. Passer à la couche de sortie et calculer l'état de chaque neurone.

$$net^s_{pk} = \sum_{j=1}^{L} w^s_{kj} i_{pj} \quad (1.4)$$

6. Calcul des sorties des neurones de la couche de sortie par la fonction d'activation $f^s(.)$

$$o_{pk} = f^s_k(net^s_{pk}) \quad (1.5)$$

Backward propagation

7. Calcul de l'erreur de la couche de sortie $\delta^s_{pk} = (y_{pk} - o_{pk}) f'^s_k(net^s_{pk})$ (1.6)
8. Calcul de l'erreur de la couche cachée $\delta^c_{pj} = f'^c_j(net^c_{pj}) \sum_k \delta^s_{pk} w^s_{kj}$ (1.7)
9. Mise à jour des poids de la couche de sortie $w^s_{kj}(t+1) = w^s_{kj}(t) + \eta \sum_{p=1}^{Np} \delta^s_{pk} i_{pj}$ (1.8)
10. Mise à jour des poids de la couche cachée $w^c_{ji}(t+1) = w^c_{ji}(t) + \eta \sum_{p=1}^{Np} \delta^c_{pj} x_i$ (1.9)
11. Calcul de l'erreur pour l'exemple x_p $E_p = \frac{1}{2} \sum_{k=1}^{M} \delta^2_{pk}$ (1.10)
12. Calcul de l'erreur globale pour tous les N_p exemples

$$E_T = \sum_{p=1}^{N_p} E_p \qquad (1.11)$$

I.2.2.2 La généralisation

La généralisation est définie comme étant la capacité de produire des sorties non rencontrées avant, à partir d'entrées non rencontrées aussi.

I.2.2.3 La capacité d'abréger

La capacité d'abréger est la capacité d'un réseau de neurones d'être insensible aux variations des données d'entrée, c'est à dire son aptitude à s'adapter à un environnement non parfait.

I.2.3 Classification des RNA

Selon leur topologie, nous distinguons les réseaux de neurones statiques et les réseaux de neurones dynamiques [Touz92].

I.2.3.1 Les réseaux de neurones statiques

Ce sont des RNA dans lesquels l'information se propage de couche en couche sans retour en arrière possible. On distingue deux catégories de réseaux de neurones statiques, les perceptrons et les réseaux à fonction radiale ou RBF (*Radial Basis Function*). Dans la catégorie des perceptrons, on distingue les perceptrons monocouches et les perceptrons multicouches (figure 1.4).

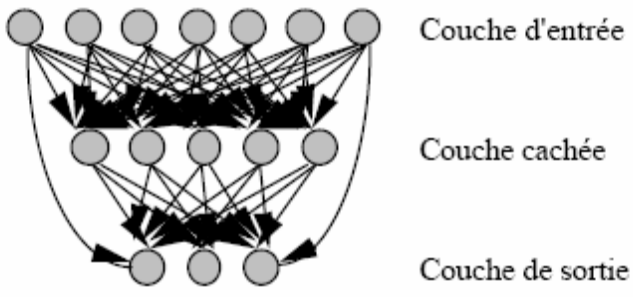


Figure 1.4: *Architecture d'un RNA multicouches.*

I.2.3.2 Les réseaux de neurones dynamiques

Les RNA dynamiques sont caractérisés par la présence, d'au moins une boucle de contre-réaction des nœuds de sorties vers les nœuds d'entrée ou vers une couche précédente (figure 1.5). Les connexions dynamiques ou récurrentes ramènent l'information en arrière par rapport au sens de propagation défini dans un réseau multicouches. Parmi les types de RNA dynamiques nous pouvons citer : les réseaux de Hopfield et les réseaux d'Elman.

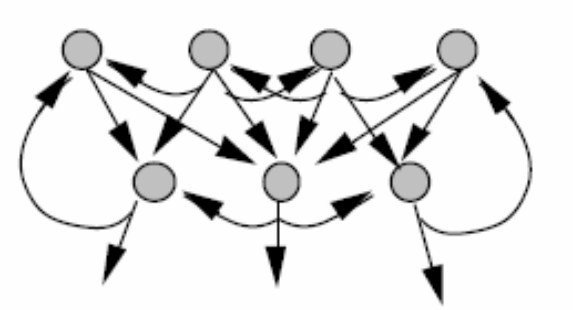


Figure 1.5*: Architecture d'un RNA dynamique.*

I.2.4 Domaines d'application des RNA

D'une manière générale, les applications des RNA sont variées, parmi lesquelles on trouve :

I.2.4.1 L'approximation des fonctions

Certains RNA, en particulier les multicouches, montrent des capacités d'approximation de fonctions très intéressantes. A partir de données expérimentales, les RNA peuvent approximer la fonction qui restitue pour chaque entrée la sortie correspondante.

I.2.4.2 La classification

Dans les applications de classification par RNA, on s'intéresse essentiellement à la classification d'objets pouvant être décrits par des vecteurs de caractéristiques numériques et booléennes. La reconnaissance de caractères est probablement, l'une des applications qui a fait le plus connaître la capacité de classification des RNA. Ainsi, chaque lettre manuscrite peut être classifiée par le réseau, de façon à regrouper et différentier efficacement le plus de caractères possibles.

I.2.4.3 La commande

Les RNA étant une structure de calcul hautement parallèle, capable d'apprendre et de s'adapter, sont souvent très efficaces pour la commande. Les sorties du système sont, alors,

contrôlées en fonction des entrées (ex: sorties de capteurs, commandes d'entrées, boucles de rétroaction, etc…), suite à un préapprentissage ou un apprentissage continu adéquat. Le projet ALVIN (*Autonomous Land Vehicle In a Neural Network*) [Pome93] est un exemple de RNA qui a appris à conduire une automobile à partir d'images vidéo de la route enregistrées lors de la conduite du véhicule par un utilisateur humain. Après la phase d'apprentissage, le RNA a pu conduire l'automobile sur plusieurs kilomètres sans quitter la route.

A noter qu'il existe bien d'autres domaines où les RNA sont appliqués : l'optimisation, la prédiction, le filtrage, la compression des données, etc...

Pour des raisons d'exigences pratiques pour certaines applications, les solutions logicielles ne sont pas suffisantes. Il faudra donc recourir à des RNA matériels (implémentation *Hardware*) sur des circuits dédiés. Un type de circuits qui est le plus utilisé dans l'implémentation des RNA, est le FPGA (*Field Programmable Gates Arrays).*

I.3 Les méthodes de commande par réseaux de neurones

Les données généralement disponibles pour la réalisation d'un système de commande par apprentissage sont les couples associant la sortie (ou l'état) y (t) du processus à la sortie (ou l'état) désirée $y^*(t)$, on peut ainsi mesurer l'erreur ε_y (différence entre la valeur de sortie désirée et la valeur mesurée) obtenue en sortie du système après l'application de la commande (Figure 3.1).

Il n'est, par contre, pas possible d'obtenir directement l'erreur $u_e(t)$ en sortie du contrôleur, c'est à dire la différence entre la commande idéale u(t) -qui n'est pas connue à priori- et la commande û(t) préconisée par le contrôleur (figure 1.6). C'est pourtant cette erreur qui est nécessaire pour réaliser l'adaptation du contrôleur par une méthode d'apprentissage. Ce problème est décrit sous le nom de l'apprentissage avec maître distant [Jord92].

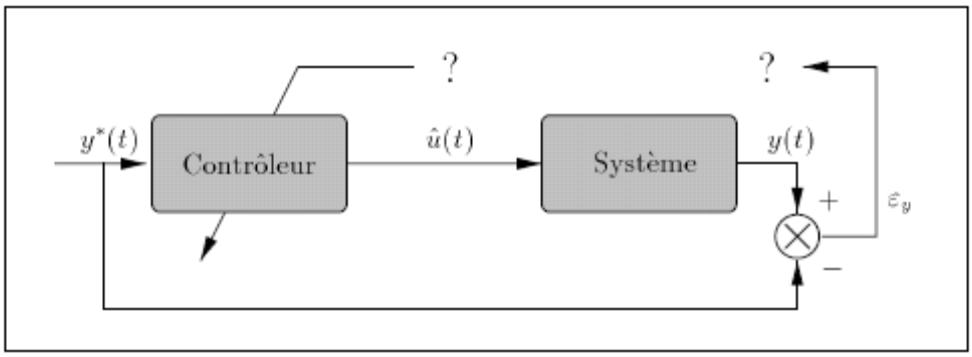


Figure 1.6: *Problème de l'apprentissage avec maître distant.*

Plusieurs méthodes ont été développées pour résoudre ce problème. Elles se divisent en deux classes principales, suivant qu'elles nécessitent ou non, l'identification préalable d'un modèle du processus commandé. Les méthodes de commande directes ne nécessitent pas l'emploi de modèle du système, contrairement aux méthodes indirectes.

I.3.1 Les méthodes directes

Parmi lesquelles nous pouvons citer :

- Reproduction d'un contrôleur existant [Widr64].

La première méthode utilisée pour la réalisation d'un système de commande neuronale, consiste à reproduire le fonctionnement d'un contrôleur existant (PID, Flou…). Même si cette approche semble, au premier abord, peu intéressante puisqu'elle nécessite l'existence d'un autre contrôleur, elle peut s'avérer très utile si ce dernier est trop complexe, trop lent pour être utilisé en temps réel, ou impraticable.

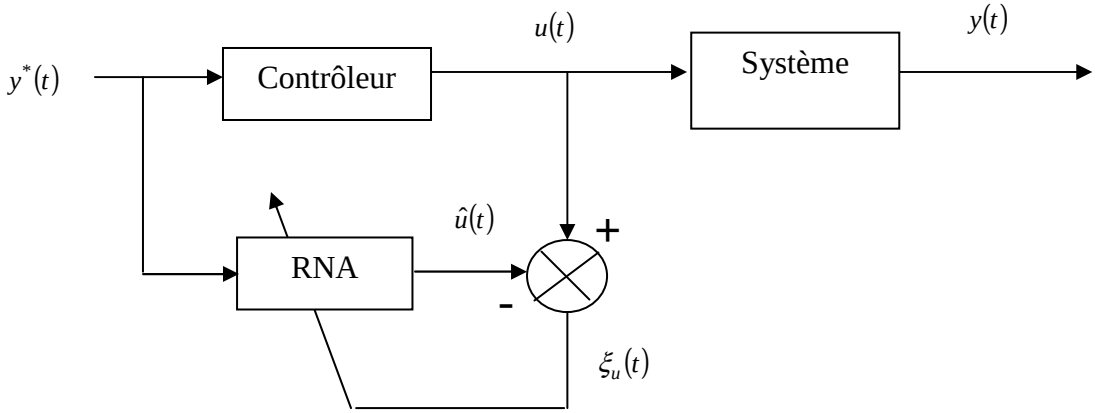


Figure 1.7: *Apprentissage d'un système de commande neuronal par reproduction d'un contrôleur existant.*

L'architecture générale est représentée sur la figure 1.7. Elle consiste à apprendre au réseau à reproduire la commande $\hat{u}(t)$ préconisée par le premier contrôleur à partir de la sortie désirée $y^*(t)$.

Un des premiers exemples de système de commande neuronal, qui a été proposé par *B. Widrow* et *F. W. Smith* en 1964, utilise cette technique pour résoudre le problème de la commande d'un pendule inversé. Il y'avait aussi des travaux de *D. A. Pomerleau* qui, en 1993, a utilisé également une architecture similaire pour apprendre à un véhicule à suivre une voie en observant les lignes blanches en bordure de la route.

Nous pouvons noter que cette approche nécessite de parcourir lors de la phase d'apprentissage, tous les modes de fonctionnement du système commandé. Il est donc nécessaire de posséder une bonne connaissance à priori des conditions d'utilisation du contrôleur.

- Commande par modèle inverse [Kuro94].

Cette approche nécessite deux phases séparées, la phase d'apprentissage puis la phase d'utilisation du réseau. Durant l'apprentissage, le réseau et le processus sont placés en parallèle (voir la figure 1.8).

Un échantillon de commandes u est fourni au processus. Nous utilisons alors les sorties y de ce dernier comme entrées du réseau qui est entraîné à retrouver en sortie les commandes u. Le réseau apprend, ainsi, un modèle inverse du processus, c'est-à-dire une fonction donnant la commande appliquée $u(t)$ à partir de la sortie actuelle du processus $y(t)$ et éventuellement de sa sortie passée $y(t-1)$.

A la fin de cette phase d'apprentissage le réseau est théoriquement capable de fournir à partir de la sortie $y^*(t)$, la commande $u(t)$ nécessaire pour obtenir une sortie du processus $y(t)$. Il est donc placé directement en série avec le système commandé.

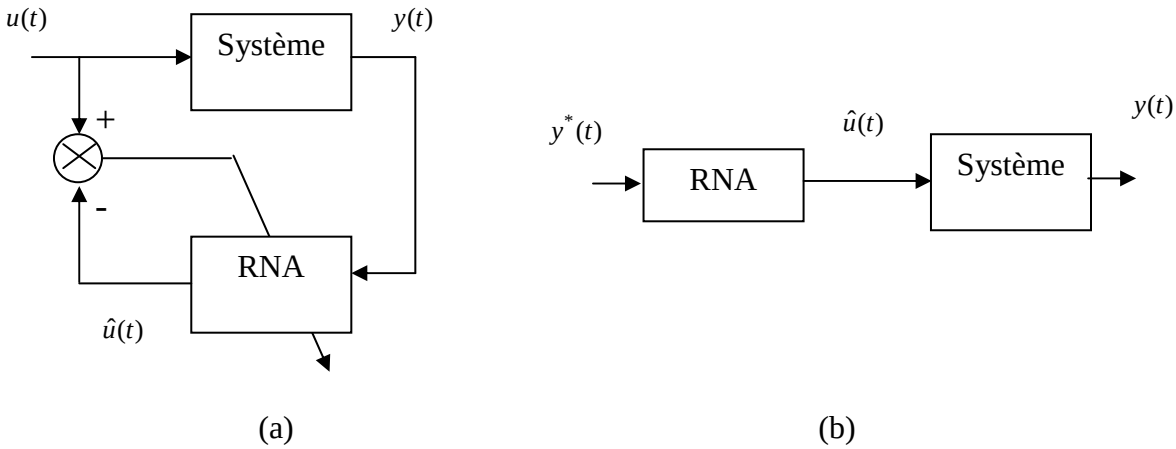


Figure 1.8: *La commande par modèle inverse : (a) phase d'apprentissage (b) phase d'utilisation.*

Cette approche a été utilisée avec succès par de nombreux chercheurs. Nous pouvons citer, par exemple, les travaux de Y. Kuroe et al. (en 1994) qui l'appliquèrent à la commande d'un bras manipulateur.

- Amélioration d'un système de commande linéaire [Kawa90].

I.3.2 Les méthodes indirectes

Parmi lesquelles nous pouvons citer :

- Apprentissage indirect du système de commande [Jord92].
- Réalisation d'un système de commande par modèle prédictif [Mill94].
- Apprentissage par renforcement [Werb92] et [Whit92].

I.3.3 La méthode d'amélioration d'un contrôleur linéaire

Une application typique des RNA dans la commande est d'améliorer un contrôleur conventionnel existant. Cette technique est connue aussi sous le nom de la méthode d'apprentissage par l'erreur de retour (*feed-back error learning*).

Le principe de cette méthode est d'employer conjointement un contrôleur linéaire classique et un contrôleur neuronal. L'idée principale est de réaliser une somme des commandes issues des deux contrôleurs, en augmentant progressivement l'importance donnée à la commande préconisée par le contrôleur neuronal, au fur et à mesure de l'apprentissage de ce dernier.

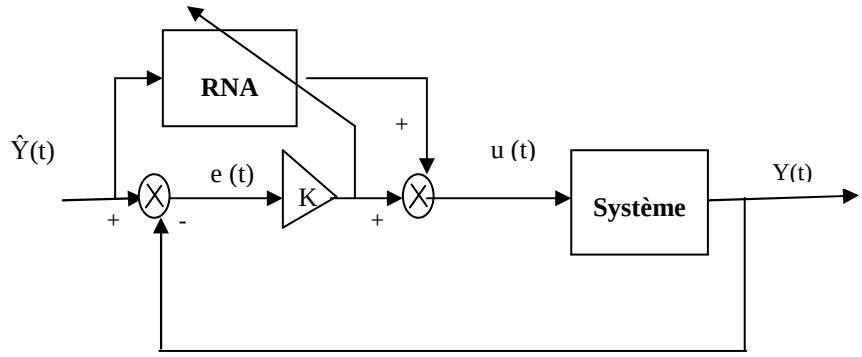


Figure 1.9: *Méthode d'amélioration d'un contrôleur linéaire.*

M. Kawato [Kawa90] propose d'utiliser un contrôleur proportionnel. On conçoit dans ce cas l'architecture décrite par la figure 1.9. Le réseau reçoit en entrée la consigne ŷ(t) et éventuellement la sortie précédente du système y(t-1) et on utilise comme erreur pour la mise à jour des poids, la sortie du contrôleur linéaire.

I.4 Les circuits FPGA

Le concept de logique programmable a été proposé par G. Estrin en 1963 [Derr02]. L'apparition des FPGA s'est d'abord fait au travers des circuits logiques programmables de type PAL (*Programmable Array Logic*) et avec les évolutions en électronique, différentes familles de circuits programmables ont commencé à apparaître : les CPLD (*Complex Logic Programmable Device*), puis les FPGA (*Field Programmable Gate Arrays*).

Les premiers circuits FPGA ont été commercialisés par la firme XILINX en 1985 et c'est un marché qui n'a pas cessé de croître depuis. Au fur et à mesure que la complexité des FPGA s'est développée, leurs possibilités d'emploi se sont accrues jusqu'à concurrencer sérieusement les circuits ASIC pour des petits volumes de production [Dute02]. Le recours aux ASIC impose en effet des temps de développement et de fabrication de l'ordre de plusieurs mois. Les FPGA, par contre, permettent une reconfiguration à volonté dans un temps très court (de l'ordre de quelques millisecondes) et donc une évolution des circuits au cours de leur période d'exploitation.

I.4.1 Caractéristiques des circuits FPGA

Le premier avantage apporté par les circuits FPGA est la souplesse de la programmation, ce qui permet de multiplier les essais, d'optimiser de diverses manières l'architecture développée et de vérifier à divers niveaux de la simulation la fonctionnalité de cette architecture [Zerr06].

Le second avantage des FPGA est la possibilité de la reconfiguration dynamique partielle ou totale des circuits, ce qui permet, d'une part, une meilleure exploitation du composant et une réduction de la surface de silicium employé, d'autre part la programmation en temps réel (quelques microsecondes) tout ou une partie du circuit.

Le composant FPGA réalise par conséquence un bon compromis Flexibilité / Performance (Figure 1.10).

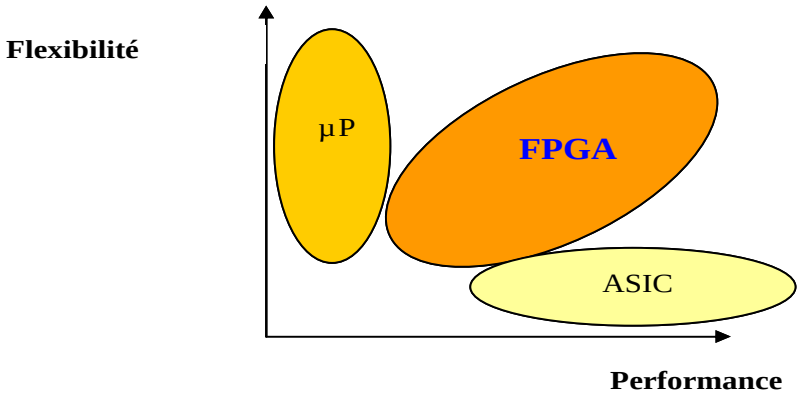


Figure 1.10: *Les circuits FPGA réalisent le compromis Flexibilité/performance*

L'inconvénient majeur des circuits FPGA est qu'ils ne sont pas très sécurisés sur le plan de la confidentialité, puisqu'il suffit d'analyser le contenu de la ROM associée pour remonter à la schématique imaginée [Sakh05].

I.4.2 Architecture des circuits FPGA

Pour décrire les FPGA, nous pouvons avoir recours à un partitionnement de leur architecture selon deux couches : la couche opérative et la couche de configuration [Groi01], [Derr02], [Dute02]. La couche opérative comprend les différents éléments assurant la réalisation de la fonction programmée et la couche de configuration regroupe tous les éléments nécessaires à la programmation du circuit.

I.4.2.1 La couche opérative

La couche opérative (figure 1.11) contient les éléments suivants:

- Les cellules logiques (CLB's).
- Les interconnexions.
- Les éléments de routage.
- Les entrées / sorties (I/O B's).

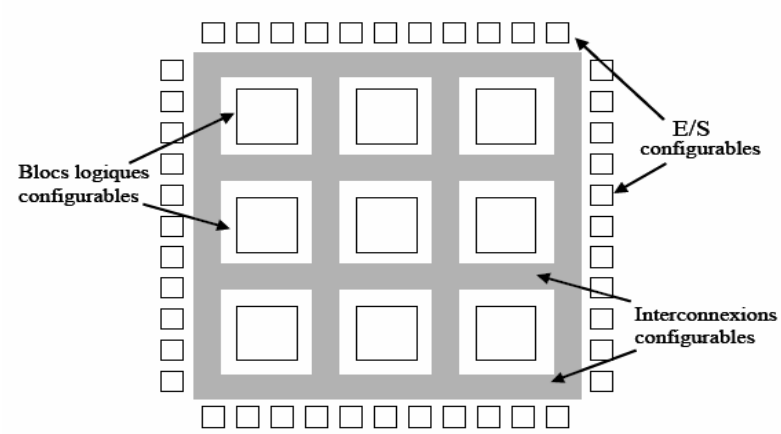


Figure 1.11: *La couche opérative d'un circuit FPGA.*

I.4.2.2 La couche de configuration

La couche de configuration d'un FPGA comporte des points mémoire, chargés de mémoriser leur configuration ainsi que les registres et la logique dédiée à la programmation. Les points mémoire retenus pour la configuration des FPGA sont appelés CSRAM (*Configuration Static Random Access Memory*) (figure 1.12). La configuration des FPGA est donc volatile et doit être chargée à chaque mise sous tension pour les familles qui adoptent la technologie SRAM.

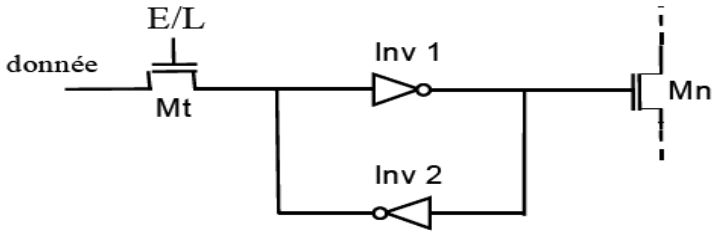


Figure 1.12: *Cellule mémoire de configuration (CSRAM)*

Chaque CSRAM permet de stocker un unique bit de configuration, un FPGA comporte donc jusqu'à plusieurs millions de points mémoires de configuration. C'est ce qui les distingue des autres familles de circuits intégrés.

I.4.3 Programmation des circuits FPGA

La programmation des circuits FPGA est l'ensemble des méthodes employées pour la génération du fichier de configuration. Le format des données de ce fichier est produit automatiquement par le logiciel de développement (ou outil de conception), sous forme d'un ensemble de bits (fichier *Bitstream*) organisés en champs de données.

Le flot de conception classique d'une architecture reconfigurable est composé de cinq étapes :

- La Spécification et synthèse logique.
- L'allocation des ressources.
- La simulation fonctionnelle.
- Le placement et routage.
- La génération de fichier de configuration.

On distingue trois grandes approches de la programmation des circuits FPGA: L'approche matérielle, l'approche impérative et l'approche objet [Groi01].

I.4.3.1 L'approche matérielle (*hardware*)

L'approche matérielle consiste à décrire l'architecture sous sa forme la plus élémentaire possible : on construit des composants par composition des portes, puis on forme des circuits par composition de composants. C'est une approche utilisée par des langages

comme le VHDL. L'avantage de cette méthode est qu'elle offre une très grande optimisation du code car ce dernier est dans ce cas très proche du matériel.

I.4.3.2 L'approche impérative

Cette deuxième méthode consiste à écrire des programmes dans un langage évolué, souvent proche du C (comme le Handel-C). Ceci a pour avantage d'être très proche des méthodes habituelles de développement. Le problème, avec ce type d'approche, se situe au niveau des compilateurs qui doivent être très performants.

I.4.3.3 L'approche objet

Les langages impératifs classiques ont montré leurs limites en ce qui concerne la modularité et la réutilisation des programmes. C'est pourquoi on développe maintenant des langages orientés objets pour les circuits à FPGA (langage Gamma). Cette approche permet de développer sur un « ordinateur hôte » et de faire une exécution à liaison dynamique sur les circuits FPGA.

I.5 Le langage VHDL

I.5.1 Historique

Le VHDL (*Very-high-speed-integrated-circuit Hardware Description Langage*) a été commandé par le DOD (Département de la défense américaine) pour décrire les circuits complexes, de manière à établir un langage commun avec ses fournisseurs. C'est un langage, standard IEEE1076 depuis 1987, qui aurait du assurer la portabilité du code pour les différents outils de travail (simulation, synthèse pour tous les circuits et tous les fabricants). La mise à jour du langage VHDL s'est faite en 1993 (IEEE 1164) et en 1996, la norme 1076.3 a permis de standardiser la synthèse VHDL [Guex98].

I.5.2 Utilité du VHDL

Le VHDL est un langage de spécification, de simulation et également de conception. Contrairement à d'autres langages (ABEL...) qui se trouvaient être en premier lieu des langages de conception, VHDL est d'abord un langage de spécification. En 1987, la normalisation a eu lieu pour la spécification et la simulation, par la suite pour la synthèse en 1993. Grâce à la normalisation, on peut être certain qu'un système décrit en VHDL standard est lisible quelque soit le fabricant du circuit. Par contre, cela demande un effort important

aux fabricants de circuits pour créer des compilateurs VHDL adaptés et autant que possible optimisés pour leurs propres circuits.

I.5.3 Spécification

Le VHDL est établi en premier lieu pour la spécification, et c'est dans ce domaine que la norme est actuellement la mieux établie. Il est tout à fait possible de décrire un circuit en un langage VHDL standard pour qu'il soit lisible. Certains fabricants ajoutent des macros offertes à l'utilisateur pour optimiser le code VHDL en fonction du circuit cible.

I.5.4 Simulation

Le VHDL est également un langage de simulation. Pour ce faire, la notion de temps, sous différentes formes, y a été introduite. Des modules, destinés uniquement à la simulation, peuvent ainsi être crées et utilisés pour valider un fonctionnement logique ou temporel du code VHDL.

La possibilité de simuler avec des programmes VHDL devrait considérablement faciliter l'écriture de tests avant la programmation du circuit et éviter, ainsi, de nombreux essais sur un prototype qui sont beaucoup plus coûteux et dont les erreurs sont plus difficiles à trouver. En plus, il est possible d'accéder aux signaux internes du circuit conçu.

I.5.5 Conception

Le VHDL permet la conception de circuits avec une grande quantité de portes logiques, en particulier les circuits FPGA. Nous pouvons résumer les avantages du langage VHDL par :

- Cycle de design plus court.
- Description plus compacte.
- Capacité d'exploration de l'espace de design, re-design.
- Maîtrise de la complexité.
- Description portable (indépendante de la technologie).

I.6 L'outil de conception ISE

La firme *Xilinx* a mis au point des logiciels performants de développement de circuits FPGA. Les séries de logiciels de *Xilinx* ont été conçues pour aider efficacement à l'étude des

projets sur FPGA. Le résultat final de tels projets est un dossier de train binaire qui peut être téléchargé dans un dispositif FPGA. Le logiciel de développement de projets sur circuits FPGA mis sur le marché est l'ISE (*Integrated Software Environement*).

L'ISE contrôle tous les aspects de déroulement de la conception. A l'aide de l'interface de gestion de projets, nous pouvons accéder aux divers outils d'exécution de la conception, et également aux dossiers et aux documents liés au développement du projet. L'outil ISE contient quatre environnements de développement d'un projet :

I.6.1 Environnement de conception en code VHDL

Cet environnement permet la description d'une architecture en langage VHDL ou VERILOG, soit en utilisant l'éditeur de schémas ou bien en utilisant les diagrammes d'états; le résultat final de l'architecture est toujours un fichier en code VHDL.

I.6.2 Environnement de simulation et de vérification (Model Sim)

La simulation permet de vérifier si le design est opérationnel par :

- Une simulation fonctionnelle après la description en code VHDL.
- Une simulation temporelle après l'implémentation de l'algorithme sur le circuit ciblé.

L'environnement de simulation « *Model Sim* » est séparé des autres environnements de conception pour une raison de souplesse lors de la vérification fonctionnelle et temporelle.

I.6.3 Environnement de synthèse

Cet environnement transforme la description VHDL en portes logiques et optimise l'architecture ciblée en surface et en temps.

I.6.4 Environnement de placement et routage

Dans cet environnement, on réalise l'implémentation et le routage sur le circuit FPGA ciblé. Les types de familles des circuits FPGA que l'outil ISE peut utiliser lors de l'implémentation sont : XC9600, Spartan-II, Virtex/-E/-II/-II Pro.

I.7 Le langage Handel-C

Le langage Handel-C a été conçu au sein de l'*Oxford University Compilation Group* et fait l'objet de la commercialisation par la société *Celoxica*. C'est un dérivé du langage C,

dont l'objectif est de permettre au concepteur un prototypage rapide d'applications sur FPGA, à partir de spécifications dans un langage de haut niveau. Les programmes écrits en Handel-C peuvent d'ailleurs être traduits en C-ANSI à l'aide d'un compilateur spécial et exécutés sur des machines séquentielles. C'est une caractéristique intéressante lors de la phase de spécification et synthèse.

La structure d'un programme en Handel-C est la même que celle d'un programme en C- standard. Elle est donnée par la figure 1.13 :

```
Déclaration d'horloge
Déclarations des variables globales
void main (void)
{
Déclarations des variables locales
Corps du programme
}
```

Figure 1.13 : *Structure d'un programme en Handel-C.*

Les principales additions par rapport au langage C sont :

- Le parallélisme.
- Canaux de communication entre les différents processus.
- Opérateurs pour la gestion du matériel: RAM, ROM, BUS, horloges….

Les principales instructions spécifiques au Handel-C et qui l'adaptent aux besoins de la programmation des circuits logiques sont :

Par {} : exécution des instructions en parallèle.

Chan : canal de communication entre deux processus.

\\n: écarte les n bits de poids faible d'une donnée.

<-n : écarte les n bits de poids fort d'une donnée.

a@b: concatène les variables a et b.

L'implémentation d'un algorithme en langage Handel-C passe généralement par les étapes suivantes [Celo06]:

- Programmation de l'algorithme en langage ANSI-C.
- Portage en Handel-C avec le minimum de changements.
- Introduction des spécifiés Handel-C : RAM, ROM, BUS….
- Introduction du *parallélisme* et des *pipelines.*

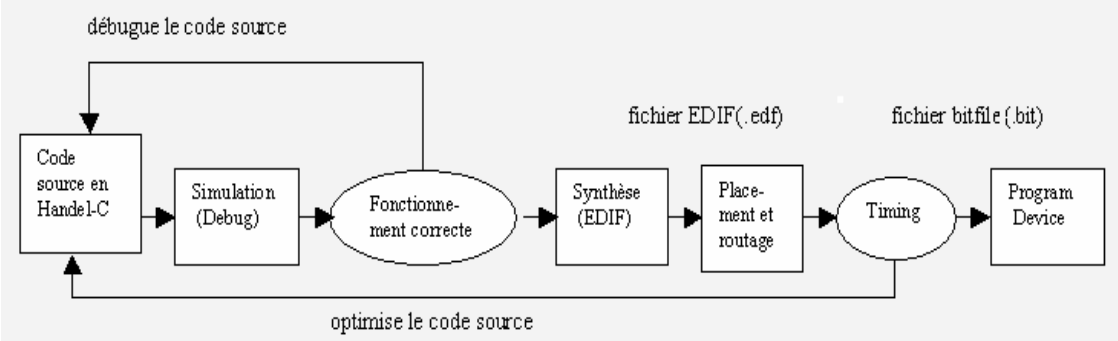


Figure 1.14 : *Etapes de développement en Handel-C.*

En compilant un code en Handel-C, on peut générer un code en VHDL pour une utilisation avec d'autres codes VHDL. On peut aussi générer un fichier de format EDIF (figure 1.14), qui à l'aide de logiciels spécifiques sera transformé en fichier de configuration des circuits FPGA (*bit file*) [Celo04].

I.7.1 Principe du langage Handel-C

Lors de la phase de compilation du programme à implanter sur le circuit reconfigurable, chaque instruction du programme source est transformée en une entité matérielle qui effectue le ou les traitement(s) associé(s) à cette instruction. Un ensemble de blocs matériels est ainsi généré représentant l'ensemble des instructions du programme. Celles-ci sont alors exécutées de façon totalement séquentielle à l'aide d'un contrôleur à jetons, qui à chaque cycle d'horloge active le bloc d'instructions courant.

Pour obtenir un circuit performant, il est nécessaire de le paralléliser. Les outils de la parallélisation automatique ne sont pas suffisamment performants, les concepteurs de Handel-

C ont donc choisi de laisser au programmeur le soin d'exprimer les différentes formes de parallélisme.

I.7.2 Comparaison Handel-C / VHDL

Le langage Handel-C, est intéressant pour les ingénieurs en Software - qui le plus souvent ne maîtrisent pas l'aspect matériel des circuits FPGA - ce qui leur permet de développer et d'implémenter des architectures sur ces circuits [Mart02]. Plusieurs études ont été réalisées afin de comparer les performances des langages VHDL et Handel-C, portant sur la consommation des ressources, les fréquences de travail des circuits et les temps de développement nécessaires pour la réalisation d'architectures matérielles.

Des travaux effectués en vue de l'implémentation d'un algorithme DES (*Data Encryption Standard*) et d'un algorithme DCT (*Discrete Cosine Transform*) sur FPGA [Loo02] avaient montré que les fréquences d'horloge générées par le VHDL et le Handel-C étaient relativement égales. L'implémentation de l'algorithme DES s'exécutaient 1.3 fois plus rapidement sous Handel-C. En revanche, l'algorithme DCT a généré une fréquence 0.75 fois celle développée sous VHDL.

En comparant la taille des ressources nécessaires pour les deux implémentations, les auteurs avaient constaté que le Handel-C consommait 2.5 fois plus de ressources. Cette différence réside dans le fait que le Handel-C utilise des librairies dont les fonctions ne sont pas toutes utilisées dans les algorithmes développés.

Une autre étude [Coe04] qui s'intéressait à l'implémentation d'algorithmes génétiques sur FPGA avait abouti aux résultats suivants :

- Le développement en VHDL avait pris 8 semaines de travail et avait nécessité l'utilisation de 8000 lignes de code. Le développement en Handel-C avait nécessité 10 jours et 1400 lignes de code
- La fréquence de fonctionnement pour chaque *Benchmark* utilisé est donnée par le tableau 1.2.

Il en résulte, que le langage VHDL optimise les ressources du circuit d'un facteur d'un demi (½) et travaille à des fréquences beaucoup plus élevées que le Handel-C de l'ordre de deux (2).

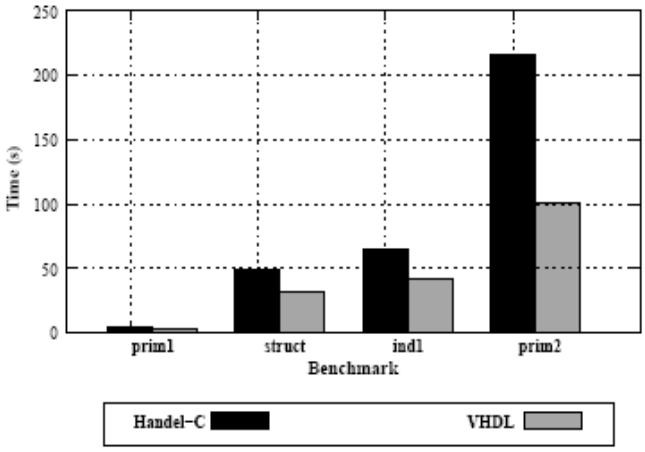


Tableau 1.2: Comparaison Handel-C VHDL [Coe04].

Les auteurs ont en conclu que le Handel-C présentait un certain intérêt pour les utilisateurs non initiés au matériel et aux circuits FPGA et nécessite beaucoup moins de temps pour le développement des implémentations, en revanche les architectures développées sous Handel-C consomment plus de ressources et travaillent à des fréquences moins rapides. Ces résultats ont été attribués par les auteurs aux limitations d'accès en mémoire externe de l'architecture conçue par le langage Handel-C.

A partir de toutes ces études, nous pouvons affirmer que le langage Handel-C peut être intéressant pour réaliser des applications qui nécessitent un temps de développement rapide et ne consomment pas beaucoup de ressources matérielles. Quant au langage VHDL, son utilisation est nécessaire pour développer des applications optimisées et travaillant à des fréquences élevées.

I.8 Conclusion

Dans cette partie j'ai exposé d'une manière succincte les RNA et étudié leurs caractéristiques principales, ainsi que les différentes méthodes de leur utilisation dans la commande des systèmes.

Par la suite, nous avons présenté un aperçu sur les circuits FPGA et montré que leurs caractéristiques de reconfigurabilité et de souplesse leurs offrent un avantage indéniable sur les autres composants.

Nous avons terminé par une présentation succincte des langages VHDL et Handel-C, et effectué une comparaison entre ces deux derniers. Nous avons montré que le VHDL

procure une meilleure fréquence de fonctionnement des architectures développées et une meilleure optimisation des ressources. En revanche, le Handel-C présente un avantage certain dans le temps de développement des applications et une plus grande souplesse dans la programmation.

Dans le chapitre suivant, nous exposerons l'approche adoptée pour l'implémentation d'un RNA, sur la carte RC200, en langage VHDL.

Chapitre II

DEVELOPPEMENT ET CONCEPTION EN LANGAGE VHDL.

André GIDE /
Journal 1889-1939 / Bibliothèque de la Pléiade / nrf Gallimard 1951

« Un chemin droit ne mène jamais qu'au but. »

< 28 octobre 1922 p.745 >

II.1 Introduction

Le chapitre précédent était consacré à un état de l'art sur l'implémentation de réseaux de neurones artificiels, ainsi qu'à des généralités concernant des notions dont nous aurons besoin par la suite.

Dans ce chapitre, nous présenterons l'approche adoptée pour l'implémentation d'un RNA, sur la carte RC200, en langage VHDL. Nous commencerons par aborder les différents problèmes qu'on rencontre, durant la conception d'une architecture neuronale, tels que la représentation des données, le bloc d'activation et le type d'implémentation. En parallèle à cela, les résultats d'implémentation obtenus seront présentés.

Par la suite, et dans le but de valider l'approche adoptée pour l'implémentation d'un RNA, sur la carte RC200, en langage VHDL, une application sera exposée avec les résultats obtenus.

II.2 Conception d'une architecture neuronale

On distingue deux types d'implémentation des réseaux de neurones multicouches, selon l'intégration ou non de la phase d'apprentissage [Izeb99] :

- **Implémentation *On-chip learning* :** Intègre la phase d'apprentissage et la généralisation dans un même circuit.
- **Implémentation *Off-chip learning* :** Intègre seulement la phase de généralisation. Dans ce genre de circuits, l'apprentissage est réalisé en simulation logicielle permettant de générer les paramètres du réseau. L'implémentation hardware consiste à charger ces paramètres dans le circuit ciblé.

II.2.1 Représentation des données

Dans les RNA, les valeurs des poids synaptiques, des entrées et des sorties sont considérées comme étant des données réelles. On distingue deux types de représentation de données :

- La virgule fixe,
- La virgule flottante.

Notre travail consiste à tester les deux types d'implémentation (*Off-chip* et *On-chip learning*) du réseau de neurones multicouches selon les deux représentations possibles (la virgule fixe et flottante).

II.2.2 Les différentes représentations du bloc d'activation

La fonction d'activation utilisée est la fonction logistique, à qui on associe un paramètre réel α . Elle est donnée par l'équation suivante :

$$f(x) = \frac{1}{1 + \exp(-\alpha x)} \qquad (2.1)$$

Rappelons que la forme de la fonction sigmoïde pour $\alpha = 1$ est donnée par la figure 2.1 :

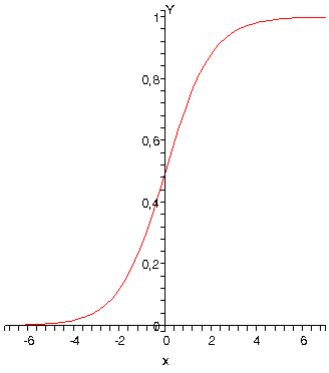


Figure 2.1: *La fonction Sigmoïde.*

Ce bloc représente la fonction d'activation, son rôle est de prendre la valeur de la somme pondérée calculée par le neurone et lui appliquer la fonction sigmoïde pour transmettre une valeur d'activation à la sortie du neurone.

Le problème de la fonction sigmoïde est qu'elle est difficile à implémenter. Pour remédier à ce problème, trois méthodes sont utilisées (figure 2.2).

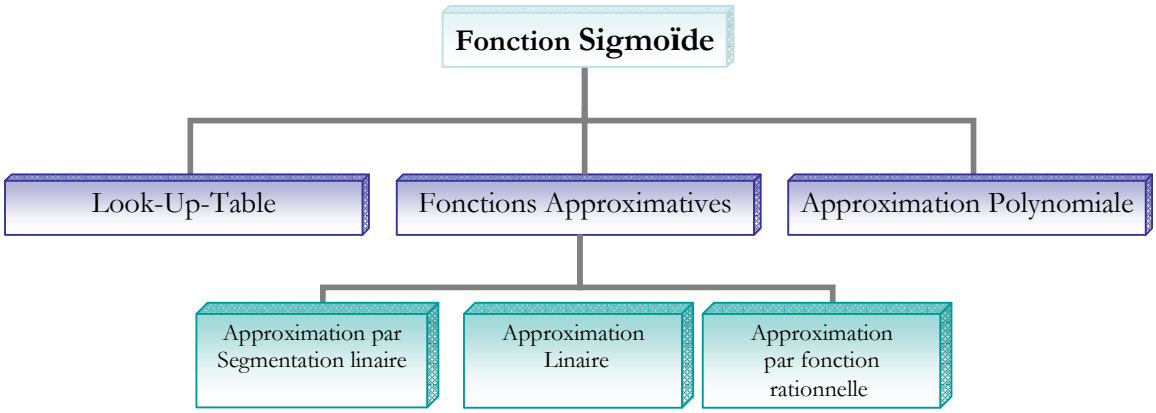


Figure 2.2: *Les différentes représentations de la fonction d'activation.*

II.2.2.1 Look-Up-Tables (LUTs)

La fonction sigmoïde est définie sous forme adresse/valeur. L'adresse représente la valeur de l'état du neurone qui sera affectée à la fonction sigmoïde pour donner la sortie de la LUT qui représente la sortie du neurone (figure 2.3).

Ainsi, la fonction sigmoïde est facile à implémenter, mais elle nécessite un nombre considérable de données représentatives.

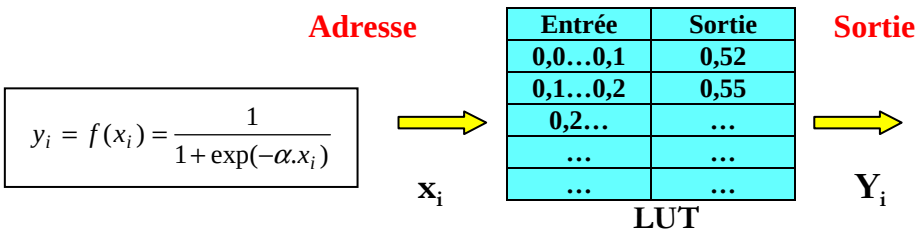


Figure 2.3: *Architecture par LUT du bloc.*

II.2.2.2 Fonctions approximatives

On peut représenter le bloc d'activation par des fonctions spécifiques approximant la sigmoïde et pouvant être implémentées facilement sur circuit FPGA. On peut citer les exemples suivants:

a- Approximation linéaire

Cette approximation est simple à implémenter et donnée par l'équation suivante :

$$f(x) = \begin{cases} 1 & \text{si } x \geq h \\ \frac{1}{2}.\left(1 + \frac{x}{h}\right) & \text{si } |x| \leq h \\ 0 & \text{si } x \leq -h \end{cases} \qquad (2.2)$$

Cette fonction (figure 2.4) est linéaire dans l'intervalle $[-h,+h]$, avec une pente de $1/2h$, et h est un paramètre réel à ajuster.

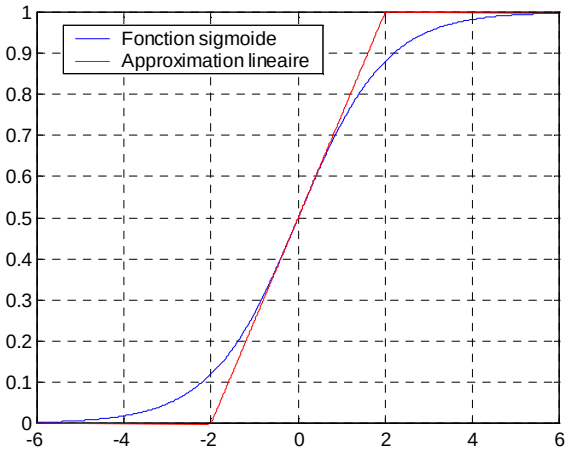


Figure 2.4: *Comparaison entre la fonction sigmoïde et l'approximation linéaire (h=2).*

b- Approximation par segmentation linéaire

Une autre alternative consiste à utiliser une approximation linéaire par morceaux de la fonction sigmoïde [Amin97], dans ce cas il s'agit de diviser la fonction sigmoïde en huit (8) segments, avec X compris dans l'intervalle [-7, 7] et Y dans l'intervalle [0, 1]. On donne les équations (2.3) de cette approximation :

$$\begin{cases} y = y_1 = 0{,}25.|x| + 0{,}5 & 0 \leq |x| < 1 \\ y = y_2 = 0.125|x| + 0.625 & 1 \leq |x| < 2.375 \\ y = y_3 = 0.03125|x| + 0.84375 & 2.375 \leq |x| < 5 \\ y = y_4 = 1 & |x| \geq 5 \end{cases} \qquad (2.3)$$

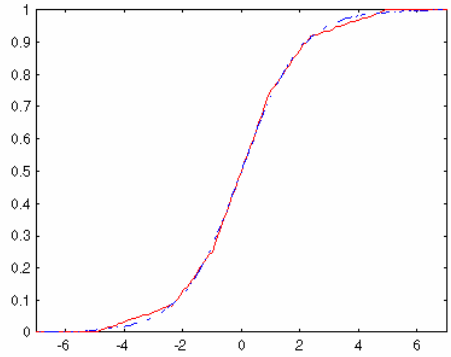


Figure 2.5: *L'approximation par segmentation linéaire de la fonction sigmoïde.*

Les pentes des différentes droites (figure 2.5) sont choisies de telle manière que la multiplication puisse être remplacée par des opérations de décalage, ce qui permet de réduire la complexité de l'implémentation de la fonction d'activation.

c- Approximation par fonction rationnelle

Donnée par *Nordstrom et Svensson* en 1992 (figure 2.6) selon l'expression suivante :

$$f(x) = \frac{1}{2}\left(\frac{x}{1+|x|}+1\right) \qquad (2.4)$$

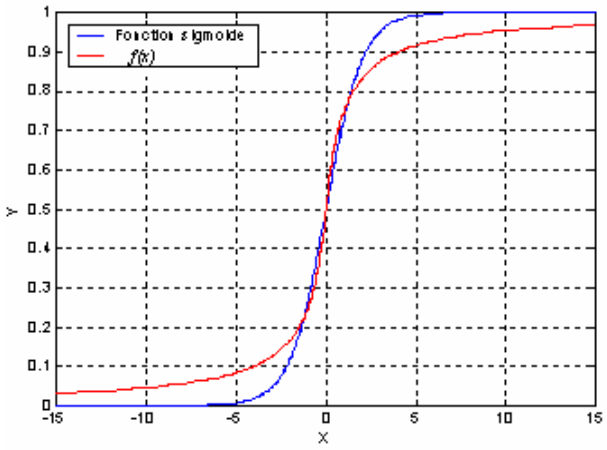


Figure 2.6: *Approximation par fonction rationnelle.*

Cette fonction est utilisée généralement pour implémenter les réseaux de neurones qui réalisent la fonction non-linéaire XOR [Gucc93].

d- Polynômes approximatifs

Cette méthode consiste à écrire le polynôme P d'ordre n donné qui approxime la fonction sigmoïde sur l'intervalle $I = [a, b]$ déterminé [Beuc02]. Il existe différentes manières de procéder, un développement de Taylor autour du centre de l'intervalle, mais il n'existe de meilleure approximation qu'autour du point considéré et pas globalement sur l'intervalle complet.

Une bien meilleure solution consiste à utiliser une approximation au sens de la norme :

$$\left\| f(x) - P(x) \right\|_{\infty} = Sup_I \left| w(x)(f(x) - P(x)) \right| \qquad (2.5)$$

Cette approximation se calcule numériquement avec un algorithme de Remez [Beuc02]. La fonction *w(x)* est une fonction de poids non nul, on prend par exemple *w(x)=1/f(x)* pour minimiser l'erreur relative.

Le polynôme approximatif d'ordre 5 dans l'intervalle I=[0 ,5] de la fonction sigmoïde est donné par l'équation suivante :

$$P(x)=0.49999351+(0.25276133+(-0.00468879 +(-0.02246096+ (0.00541780- 0.00039438.x).x).x).x).x \qquad (2.6)$$

Avec une erreur relative de **0.689 e-3.**

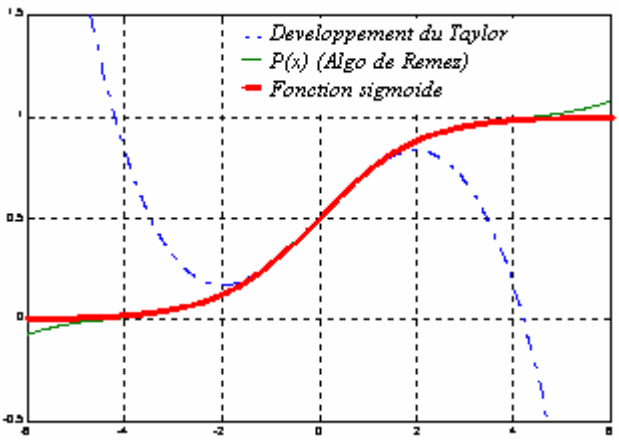


Figure 2.7: *Le graphe de P(x), en comparaison avec la fonction sigmoïde et le développement de Taylor.*

II.2.3 Implémentation avec apprentissage hors ligne

La description structurelle du neurone en VHDL est constituée des éléments suivants :

- Un multiplieur.
- Un additionneur.
- Une fonction d'activation.

Le neurone effectue d'abord le produit des données en entrée avec les poids synaptiques, il les additionne ensuite. Le résultat de ces opérations est enfin soumis à la fonction d'activation.

Après la détermination des paramètres de l'architecture par l'apprentissage logicielle (par exemple Matlab), nous implémenterons ces mêmes paramètres sur le circuit FPGA.

II.2.4 Implémentation avec apprentissage en ligne

L'architecture proposée permet de faire une implémentation qui intègre la phase d'apprentissage et la généralisation sur le même composant, elle contient trois modules (figure 2.8) :

- Forward-propagation.
- Back-propagation.
- Unité de contrôle.

a- Forward-propagation

Ce module représente la première étape de fonctionnement du réseau de neurones, dont l'architecture (multicouche) dépend de la structure fixée. Il contient une couche d'entrée, une couche cachée et une couche de sortie. Il calcule tous les états des neurones.

b- Back-propagation

Le module back-propagation représente la deuxième phase fonctionnelle, il est structuré de la même manière que le module précédent, mais dans le sens de propagation de l'erreur (en arrière). Dans ce module, l'erreur globale E_T est calculée pour tous les exemples d'apprentissage, et les paramètres du réseau de neurones sont ajustés.

c- Unité de contrôle

Ce module contrôle le processus d'apprentissage, il arrête l'apprentissage en ligne (*on-chip learning*) si le nombre maximal d'itérations ou l'erreur globale sont atteints.

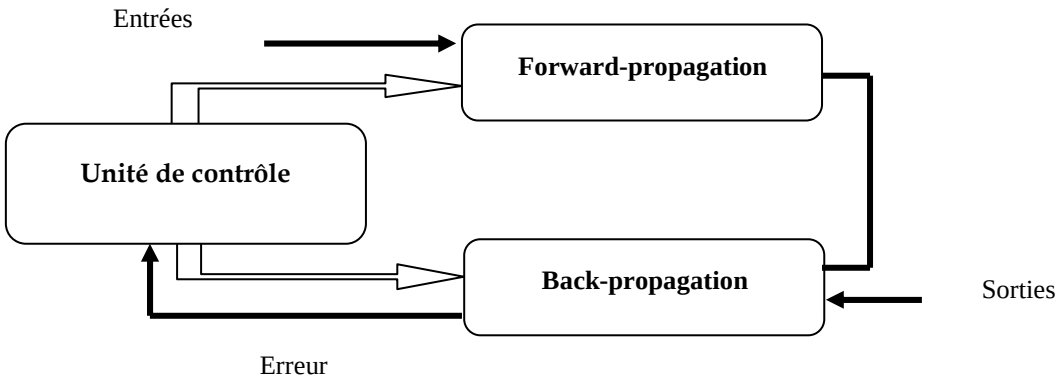


Figure 2.8: *Schéma global de l'implémentation on-chip learning des RNA.*

II.3 Représentation en virgule fixe

Cette forme de représentation s'appelle le Codage en virgule fixe complément à deux [Pari04].

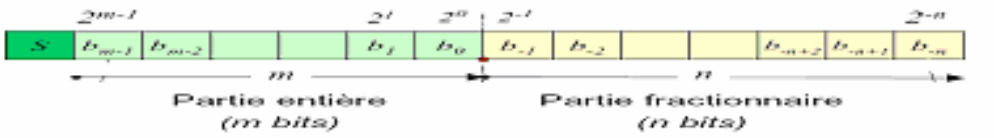


Figure 2.9: *Représentation des données en virgule fixe.*

La valeur du nombre x est donnée par l'équation suivante :

$$x = -2^m S + \sum_{i=-n}^{m-1} b_i 2^i \qquad (2.7)$$

La précision de la virgule est définie selon la précision exigée. Pour notre cas, les données sont sur 16 bit en format virgule fixe (m=7et n=8), donc le domaine de définition de x est $[-2^7 , 2^7 - 2^{-8}]$ avec une précision de 2^{-8}.

II.3.1 Implémentation avec apprentissage hors ligne

Résultats de la simulation et de la synthèse

La conception de l'implémentation a été effectuée pour deux architectures de réseau de neurones : [1,5,1] et [1,3,1]. Les paramètres du réseau sont calculés hors ligne (*off line*) par Matlab.

Après avoir représenté les paramètres du réseau dans le format virgule fixe (sur 16 bits), l'implémentation a été testée pour la généralisation de différentes fonctions : la fonction linéaire, l'exponentielle et la sinusoïde. Ceci a donné les résultats suivants :

1- fonction linéaire

$$f(x) = x \quad \text{pour} \quad x \in [0,1]$$

L'erreur globale atteinte est 0.006 et la sortie du réseau [1,3,1] est illustrée par la figure 2.10.

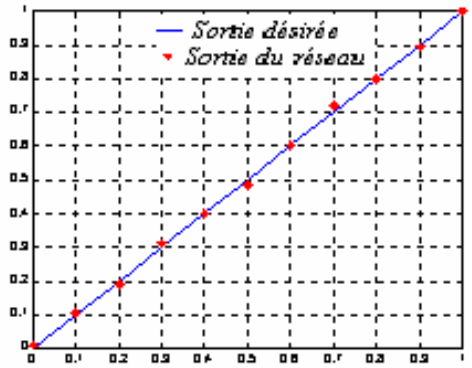


Figure 2.10: *Approximation de la fonction f(x) = x.*

2- Fonction exponentielle

$$f(x) = 1 - \exp(-5x) \quad \text{pour} \quad x \in [0,1]$$

L'erreur globale atteinte est 0.006 et la sortie du réseau [1,3,1] est illustrée par la figure 2.11.

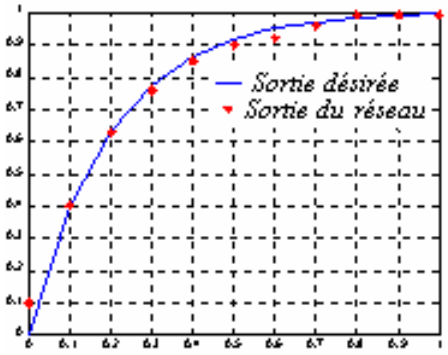


Figure 2.11: *Approximation de la fonction f(x) = 1 - exp (-5 x).*

3- fonction sinusoïdale

$$f(x) = 0.5\sin(2\pi.x) \quad \text{pour} \quad x \in [0,1]$$

L'erreur globale atteinte est 0.0094, et la sortie du réseau [1,3,1] est illustrée par la figure 2.12.

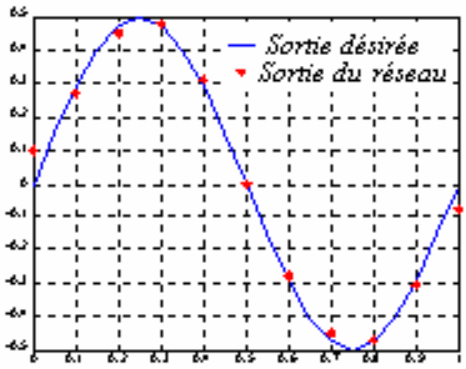


Figure 2.12: *Approximation de la fonction f(x) = 0.5 sin (2πx).*

Les résultats de la synthèse

Les résultats de la synthèse pour l'implémentation avec apprentissage hors ligne (*off-chip learning*) pour les deux architectures de réseau de neurones [1,5,1] et [1,3,1], sur le composant XC2V100fg456 de la famille Virtex II, sont indiqués sur le tableau 2.1.

RESEAU / Ressources	[1, 5, 1]	[1, 3, 1]
Entrées/Sorties I/O	33 / 324 (10%)	33 /324 (10%)
CLB Slice	1112 / 5120 (21%)	683 /5120 (13%)
Mult 18x18s	34 / 40 (85%)	21 / 40 (52%)
LUTs	1938 / 10240 (18%)	1193 / 10240 (11%)
Fréquence maximale	16.55 Mhz	17.21 Mhz

Tableau 2.1: *Résultats de la synthèse pour l'implémentation off-chip learning.*

Nous constatons que l'implémentation avec apprentissage hors ligne (*off-chip learning*) n'est pas coûteuse en ressources, et on peut atteindre des erreurs globales faibles pour des architectures très réduites [1,3,1].

II.3.2 Implémentation avec apprentissage en ligne

Résultats de la simulation et de la synthèse

	RESEAU [1,5,1]	
Multiplication / Ressources	**Bloc multiplieur (Mult18x18s)**	**Algorithme de Multiplication**
Entrées/Sorties (I/O)	64 / 556 (11%)	64 / 556 (11%)
CLBs Slice	5052 / 9792 (51%)	15969 / 9792 **163% (*)**
Mult 18x18s	116 / 88 **131% (*)**	0 / 88 (0%)
Luts	9288 / 19584 (47%)	30275 / 19584 **154% (*)**
Fréquence maximale	16.55 Mhz	17.21 Mhz

Tableau 2.2: *Résultats de synthèse pour l'implémentation on-chip learning.*

Le tableau 2.2 montre les résultats de la synthèse de l'implémentation avec apprentissage en ligne (*on-chip learning*), de l'architecture [1,5,1] d'un réseau de neurones, selon différentes manières d'implémentation de la multiplication, soit par le bloc multiplieur 18x18s de la famille Virtex-IIPro, ou bien par un algorithme de multiplication [Aumiau]. La fonction d'activation est de type Look-Up-Table.

Dans les deux cas, il y a un dépassement de capacité. Pour remédier à ce problème, la multiplication sera faite en combinant les deux méthodes (tableau 2.3).

	RESEAU [1 ,5 ,1]		
Fonction d'activation / Fonction à approximer	**Look-Up-Table**	**Approximation linaire**	**Polynôme approximatif**
Fonction linéaire F(x)=x	0.0379	0.00221	4.68 e-4
Fonction exponentielle 1-exp (5x)	0.0601	0.0250	0.0077
Fonction sinusoïdale 0.5 sin (2πx)	0.1034	0.0867	0.0235

Tableau 2.3: *Erreur globale pour dix (10) exemples.*

Le tableau 2.3 montre les résultats de la simulation de l'implémentation avec apprentissage en ligne de l'architecture [1,5,1] de réseaux de neurones, selon la conception de la fonction d'activation: Look-Up-Table, approximation linéaire, ou polynôme approximatif. La multiplication est réalisée en combinant les deux méthodes déjà citées. L'erreur globale E_T est donnée pour les différentes fonctions approximées, et avec dix (10) exemples d'apprentissage.

	RESEAU [1,5,1]		
Fonction d'activation / Ressources	**Look-Up-Table**	**Approximation linaire**	**Polynôme approximatif (Algo. de Remez)**
Entrées/Sorties (I/O)	64 / 556 (11%)	64 / 556 (11%)	64 / 556 (11%)
CLBs Slice	6728 / 9792 (68%)	6238 / 9792 (63%)	9470 / 9792 (96%)
Mult 18x18s	86 / 88 (97%)	86 / 88 (97%)	86 / 88 (97%)
Luts	11997 / 19584 (61%)	11189 / 19584 (57%)	17392 / 19584 (88%)
Fréquence maximale	3.940 MHz	4.169 MHz	1.761 MHz

Tableau 2.4: *Résultats de la synthèse pour l'implémentation on-chip learning.*

Le tableau 2.4 montre les résultats de la synthèse des différents cas du tableau 2.3. A partir de ces derniers résultats, nous constatons que la conception du bloc d'activation par un polynôme approximatif réalise des erreurs globales relativement petites, mais nécessite plus de ressources.

II.4 Représentation en virgule flottante

Le codage en virgule fixe sur *n* bits (partie entière) ne permet de représenter qu'un intervalle de 2^n valeurs. Pour un grand nombre d'applications, cet intervalle est trop restreint. La représentation en virgule flottante (*floating-point*) a été introduite pour répondre à ce besoin. Pour des données de 32 bits, cette représentation en virgule fixe permet de coder un intervalle de 2^{32} valeurs tandis que la représentation en virgule flottante permet de coder un intervalle d'environ 2^{255} valeurs.

Les nombres à virgule flottante sont de la forme :

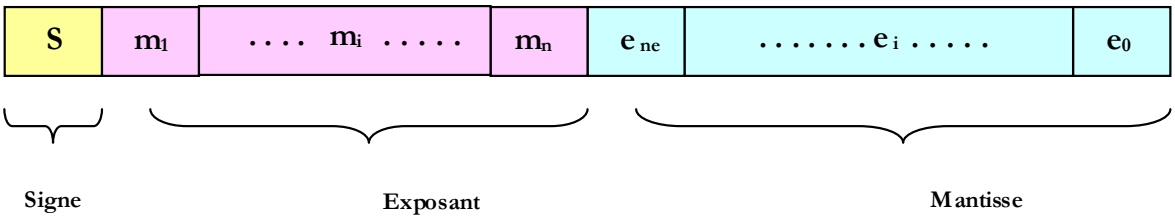


Figure 2.13: *Représentation des nombres réels à virgule flottante.*

La valeur d'un nombre x écrit en virgule flottante est donnée par l'équation suivante :

$$x = (-1)^s .M.\beta^e \qquad (2.8)$$

Où

β est la base de représentation (usuellement 2 ou 10).

$S \in \{0,1\}$ est le bit qui code le signe.

M est la mantisse, un nombre rationnel à virgule fixe codé sur n_m chiffres en base β, soit

$M = d_0, d_1 d_2 ... d_{n_m - 1}$

E est l'exposant, un entier relatif codé en binaire sur n_e.

II.4.1 Nombres normalisés et dénormalisés

Même en fixant la place de la virgule dans la mantisse, le même nombre peut avoir plusieurs représentations. Ainsi en base $\beta = 10$, avec une précision $p = 4$, les deux flottants $3{,}140$ et $0{,}314.10^1$ sont identiques. On parle de nombre normalisé lorsque le chiffre de poids fort de la mantisse est non nul, et pour tout réel représentable non nul, la représentation devient alors unique [Guyot].

Une conséquence intéressante en base $\beta = 2$ est que pour un nombre normalisé, le premier chiffre (bit) de la mantisse est toujours 1. On peut, alors, décider de ne pas le stocker physiquement en mémoire, et on parle alors de bit de poids fort implicite (*implicit leading*).

II.4.2 Représentation IEEE 754

La représentation en virgule flottante n'est pas unique pour un nombre x donné. Pour cette raison elle a été normalisée (norme IEEE 754). Cette norme définit quatre (4) formats de flottants en base 2 [Guyot]:

- simple précision (correspondant généralement au type float en C).
- simple précision étendue (obsolète).
- double précision (correspondant généralement au type double en C).
- double précision étendue (la mantisse est normalisée).

Dans cette représentation, la valeur d'un nombre s est donnée par l'expression :

$$x = (-1)^s.(1+\sum_{i=1}^{n} m_i.2^{-i}).2^{E-biais} \quad \text{tel que} \quad E = \sum_{i=1}^{ne} e_i.2^i \qquad (2.9)$$

La valeur du biais est fixée selon le type de la précision (simple ou double).

	Simple précision (32 bits)	**Double précision (64 bits)**
Longueur totale	32 bits	64 bits
Mantisse + bit implicite	23 +1 bits	52 +1 bits
Exposant	8 bits	11 bits
Biais, max, min	127	1023
Domaine approximatif	$2^{128} \approx 3.8\ 10^{38}$	$2^{1024} \approx 9\ \text{x}\ 10^{307}$
Précision approximative	$2^{-23} \approx 10^{-7}$	$2^{-52} \approx 10^{-15}$
Plus petit nombre normalisé	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$

Tableau 2.5: *La représentation simple et double précision en virgule flottante.*

En résumé, n_m spécifie la précision du format, et n_e spécifie sa dynamique.

II.4.3 Arithmétique flottante

II.4.3.1 Addition flottante

L'addition de deux nombres flottants se fait en quatre étapes [Guyot]:

1. Tri du plus grand et du plus petit, et alignement de la mantisse du plus petit.
2. Exécution de l'opération (addition ou soustraction).
3. Normalisation (si c'est possible).
4. Arrondi du résultat.

II.4.3.2 Multiplication flottante

La multiplication de deux nombres flottants se fait en quatre étapes [Guyot]:

1. Multiplication des mantisses.
2. Ajout des exposants
3. Normalisation (si c'est possible).
4. Arrondi du résultat.

II.4.3.3 Résultat de la synthèse

Les tableaux suivants comparent les tailles des ressources consommées des opérateurs de base, selon la représentation des données utilisée (virgule fixe ou flottante). Le composant FPGA ciblé est de la famille Xilinx Virtex-II type **XC 2v1000-fg456-4.**

a- Réalisation de l'addition

L'architecture de l'additionneur flottant (simple précision-32 bits) est assez complexe et nécessite plusieurs opérations de traitement, qui ne peuvent pas être réalisées en parallèle.

REPRESENTATION DES DONNEES / RESSOURCES	ADDITIONNEUR (SIMPLE PRECISION 32 BITS)	ADDITIONNEUR (VIRGULE FIXE 16 BITS)
Nombre I/O	96 / 324 (29%)	48 / 324 (14%)
Nombre Slice	916 / 5120 (17%)	8 / 5120 (0.16 %)
Nombre Mult 18x18s	0 / 40 (0 %)	0 / 40 (0 %)
LUTs	1668 / 10240 (16%)	16 / 10240 (0.16 %)
Fréquence Max	11.62 Mhz	111.43 Mhz

Tableau 2.6: *Comparaison en termes de ressources de l'addition selon le format des données.*

b- Réalisation de la multiplication

L'architecture du multiplieur est plus simple que celle de l'additionneur, il suffit d'effectuer le produit des mantisses des deux opérandes, et l'exposant du résultat est obtenu en prenant la somme.

REPRESENTATION DES DONNEES / RESSOURCES	MULTIPLIEUR (VIRGULE FIXE 16 BITS)	MULTIPLIEUR (SIMPLE PRECISION 32 BITS)
Nombre I/O	48 / 324 (14%)	96 / 324 (29%)
Nombre Slice	56 / 5120 (1%)	93 / 5120 (1%)
Nombre mult 18x18s	1 / 40 (2%)	4 / 40 (10%)
LUTs	101 / 10240 (1%)	166 / 10240 (1%)
Fréquence Max	45.63 Mhz	37.76 Mhz

Tableau 2.7: *Comparaison en termes de ressources de la multiplication selon le format des données.*

c- Réalisation de l'opération MAC (Multiplication et accumulation)

L'opération MAC (donnée par Y = A.B + C) contenant les deux principales opérations de calcul (l'addition et la multiplication), est la base du traitement numérique.

REPRESENTATION DES DONNEES / RESSOURCES	MAC (VIRGULE FIXE 16 BITS)	MAC (SIMPLE PRECISION)
Nombre I/O	64 / 324 (19%)	128 / 324 (39%)
Nombre Slice	55 / 5120 (1%)	725 / 5120 (14%)
Nombre mult 18x18s	1 / 40 (2%)	4 / 40 (10%)
LUTs	102 / 10240 (1%)	1311 / 10240 (12%)
Fréquence Max	42.96 Mhz	9.17 Mhz

Tableau 2.8: *Comparaison en termes de ressources du MAC selon le format des données.*

Nous remarquons que la taille des ressources consommées pour réaliser des opérations en virgule flottante (simple précision) est nettement supérieure par rapport aux opérateurs en virgule fixe.

II.4.4 Implémentation avec apprentissage hors ligne

Le tableau suivant indique les ressources consommées par l'implémentation avec apprentissage hors ligne (*off-chip learning*) de l'architecture [1,3,1], selon la représentation

des données utilisées (virgule fixe ou flottante). Le composant FPGA ciblé est de la famille Xilinx Virtex-II type **XC 2v1000-fg456-4**.

REPRESENTATION DES DONNEES / RESSOURCES	FORMAT VIRGULE FLOTTANTE (SIMPLE PRECISION 32 BITS) RESEAU DE [1,3,1]		FORMAT VIRGULE FIXE (16 BITS) RESEAU DE [1,3,1]
	XC 2v1000-fg456-4	**XC 2v8000-ff1152-4**	**XC 2v1000-fg456-4**
Nombre I/O	96 / 324 29%	64 / 824 7%	33 / 324 (10%)
Nombre Slice	14519 / 5120 **283%(*)**	15803 / 46592 33%	683 /5120 (13%)
Nombre Mult 18x18s	84 / 40 **210%(*)**	84 / 168 50%	21 / 40 (52%)
LUTs	28955 / 10240 **282%(*)**	28634 / 93184 30%	1193 / 10240 (11%)

Tableau 2.9: *Résultats de la synthèse de l'implémentation off-chip learning selon le type de format utilisé.*

Les (*) indiquent qu'il y a des dépassements de ressources du circuit **XC 2v1000-fg456-4.** Pour cela, la synthèse a été refaite sur le circuit Virtex-II type **XC 2v8000-ff1152-4** afin d'intégrer toute l'architecture du réseau [1,3,1] sur le composant (tableau 2.9).

II.4.5 Implémentation avec apprentissage en ligne

a- Résultats de la simulation

Le tableau 2.10 montre les résultats de simulation de l'implémentation avec apprentissage en ligne (*on-chip learning*) de l'architecture [1,5,1] d'un réseau de neurones, pour l'approximation de différentes fonctions.

La fonction d'activation est représentée par un polynôme d'ordre 7 afin d'améliorer l'approximation de la fonction sigmoïde.

Au cours du processus d'apprentissage du réseau, la valeur du taux d'apprentissage est adaptée selon la variation de l'erreur globale. Avec dix (10) exemples d'apprentissage et après un nombre considérable d'itération (10^6), l'erreur globale E_T est donnée.

FONCTION A APPROXIMER	RESEAU [1,5,1] FONCTION D'ACTIVATION POLYNOME APPROXIMATIF
Fonction linéaire F(x) = x	1.894 e-4
Fonction exponentielle 1-exp (5x)	4.980 e-3
Fonction sinusoïdale 0.5 sin(2πx)	1.081 e-2

Tableau 2.10: *Evaluation de l'erreur globale.*

Par conséquent, la représentation en virgule flottante m'a permis d'améliorer l'erreur globale obtenue par le format en virgule fixe.

b- Résultats de la synthèse

Le tableau 2.11 montre que le circuit **XC 2vp100-6 ff1696** de la famille Virtex II Pro peut supporter l'implémentation avec apprentissage en ligne pour l'architecture [1,3,1] avec une fonction d'activation représentée sous la forme d'un polynôme approximatif.

RESEAUX / RESSOURCES	RESEAU [1,3,1]
Nombre I/O	98 / 1164 (8%)
Nombre Slice	29359 / 44096 (66%)
Nombre Mult 18x18s	160 / 444 (36%)
Fréquence max	1.21 *Mhz*

Tableau 2.11: *Résultats de la synthèse de l'implémentation on-chip learning.*

II.5 Application: Approximation d'un PID pour la commande en vitesse d'un moteur à courant continu

Dans cette application, nous implémentons un réseau de neurones qui approxime un contrôleur PID sur la carte RC200 (Virtex II) de Celoxica [Celo06], et cela pour la commande en vitesse d'un moteur à courant continu [Benb06].

Pour cela, nous suivrons les étapes suivantes :

- Identification du modèle du système à commander (Moteur à courant continu).

- Détermination des paramètres du contrôleur PID par une simulation logicielle (Matlab).
- Conception du réseau de neurones qui approxime le contrôleur PID par un apprentissage logiciel (*Off chip*).
- Implémentation de cette architecture sur la carte RC200.
- Test du système global.

II.5.1 Identification du modèle du Moteur

Le Moteur est un système qui a comme entrée la tension $u(t)$ qui représente la valeur moyenne du signal MLI (Modulation Largeur Impulsion), et dont la sortie est une tension analogique prélevée du capteur tachymétrique, monté sur l'axe du moteur, avec une constante de 7mv/trpm.

Le moteur à courant continu est représenté dans le domaine fréquentiel par une fonction de transfert du premier ordre (dans le cas d'un asservissement en vitesse).

$$F(p) = \frac{G_0}{1 + \tau . p} \tag{2.10}$$

Tel que G_0 : est le gain statique,

τ : est la constante du temps.

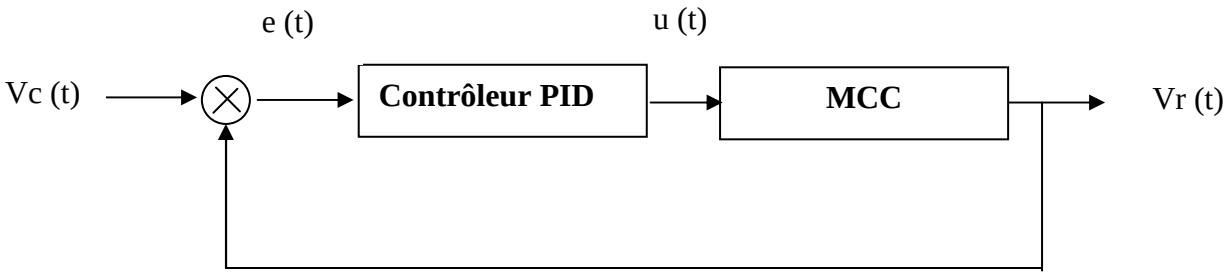
L'identification du moteur est effectuée par la méthode de SBPA (séquence binaire pseudo aléatoire) sous Matlab, et l'acquisition est faite par une carte Dspace.

Les valeurs des paramètres identifiés sont les suivantes :

$$\begin{cases} G_0 = 0.4 \\ \tau = 0.034 \end{cases} \tag{2.11}$$

II.5.2 Détermination des paramètres du PID

Le but est de trouver les paramètres du contrôleur PID, adéquats pour l'asservissement en vitesse pour une consigne type échelon unitaire.



$V_c(t)$: La consigne vitesse.
$V_r(t)$: La vitesse réelle
$e(t) = V_c(t) - V_r(t)$: L'erreur entre la consigne vitesse et la vitesse réelle
$u(t)$: La commande

Figure 2.14: A*sservissement du MCC par un contrôleur PID.*

Après la simulation par Matlab, les gains du PID sont $K_p = 10$, $K_I = 220$, $K_D = 0$.

La figure suivante montre l'allure de la consigne et de la sortie du système.

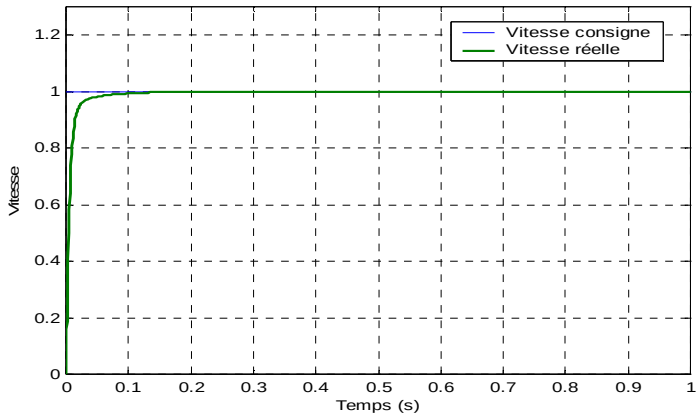


Figure 2.15: *La réponse du système commandé par le PID.*

II.5.3 Conception du réseau de neurones approximant le PID

A partir du vecteur entrée/sortie du contrôleur PID, un réseau de neurones d'architecture [1,3,1] nous a permis de l'approximer.

La base des exemples est constituée de dix (10) échantillons, l'apprentissage est basé sur l'algorithme de la rétropropagation du gradient (Levenberg-Marquardt), et l'erreur atteinte est de 0.0035.

Les paramètres du réseau obtenus sont comme suit :

Les poids de la couche cachée :

Wc = {22.811, 8.201, -17.197}

bc = {-7.414, -6.005, -1.940}

Les poids de la couche de sortie :

Ws = {2.057, 2.472, 0.171}

bs = {0.717}

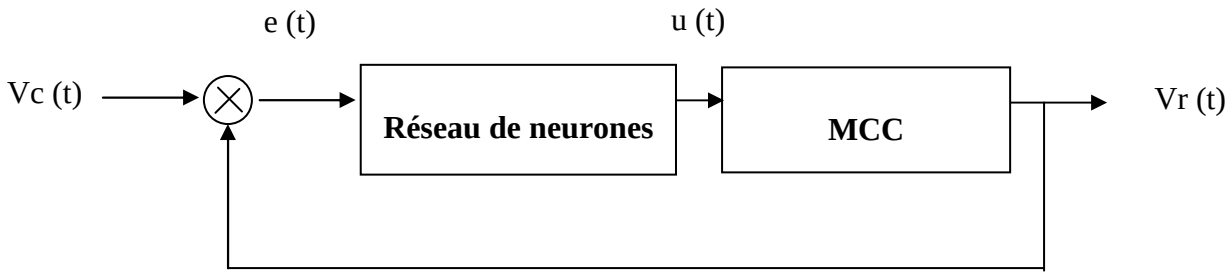


Figure 2.16: *La commande du MCC par un réseau de neurones pour un échelon unitaire.*

La sortie du système est donnée par la figure 2.17.

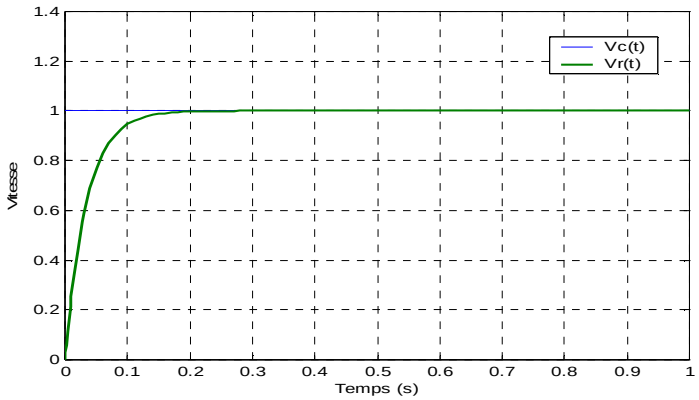


Figure 2.17: *La réponse du système, commandé par un réseau de neurone pour un échelon unitaire.*

Le système de commande illustré par la figure 2.16 fonctionne pour une vitesse consigne en échelon unitaire $\mu(t)$, car le réseau de neurones utilisé approxime le contrôleur

PID pour Vc(t)= $\mu(t)$. Pour que l'asservissement en vitesse du moteur soit possible pour d'autres vitesses Vc(t), nous avons ajouté deux (2) gains dans la boucle de commande.

Donc le système de commande sera de la forme suivante :

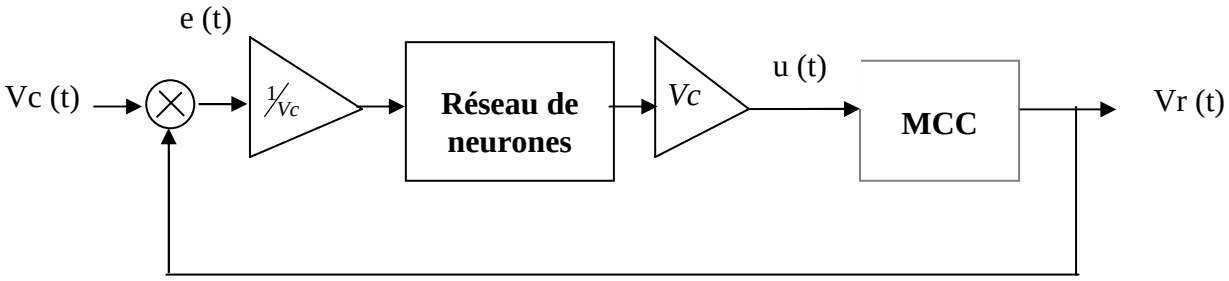


Figure 2.18: *La commande du MCC par un réseau de neurones.*

II.5.4 Description matérielle du système de commande

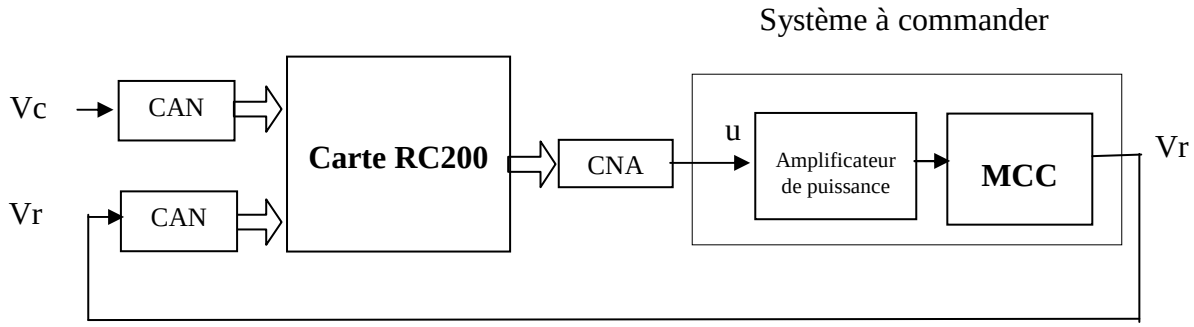


Figure 2.19: *Description matérielle du système de commande.*

Le système de commande est constitué de (annexe 1) :

- **Le système à commander** : comporte l'amplificateur de puissance et le moteur à courant continu.
- **Carte RC200** : comporte le circuit FPGA type Virtex II de Xilinx.
- **Carte de conversion A/N :** son rôle est de conditionner les grandeurs analogiques Vc(t) et Vr(t) afin de les convertir en données numériques, et est composée de :
  - Un amplificateur.
  - Un montage de convertisseur A/N type ICL7109 à 12 bits (annexe 2).

- **Carte de conversion N/A** : son rôle est de convertir les données numériques issues de la carte RC200 en grandeurs analogiques (annexe 3). Ce signal de commande U(t), sera amplifié pour attaquer le moteur à courant continu.

II.5.5 Description en VHDL du système de commande

Le système de commande est définit par des entités en VHDL, il comporte trois (03) entrées : les deux vitesses consigne Vc12 et réelle Vr12 sur 12 bits, le signal d'horloge Clk pour la synchronisation, et deux sorties : la commande U8 (sur 8 bits) et le signal WR pour commander le convertisseur N/A.

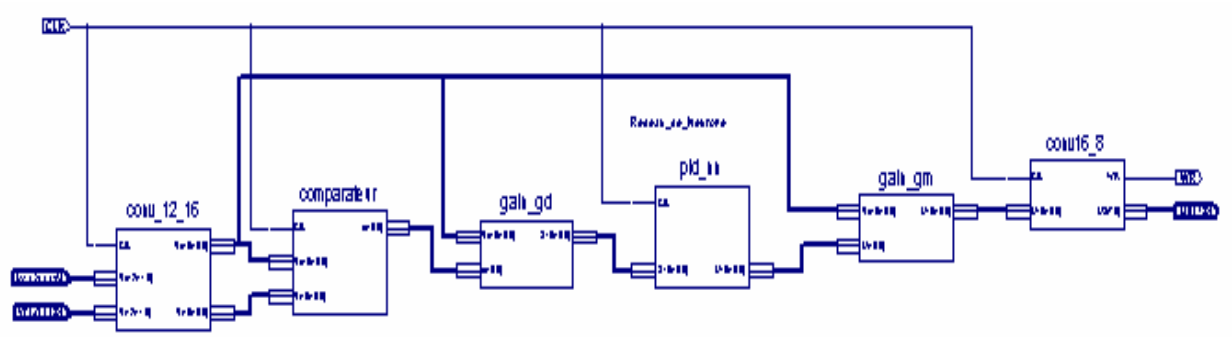


Figure 2.20: *Description schématique du système de commande.*

Le système de commande contient :

- **Conv12_16** : effectue la conversion des données numériques sur 12 bits issues du convertisseur en format virgule fixe sur 16 bits.
- **Comparateur** : calcule l'erreur entre la vitesse consigne et la vitesse réelle.
- **Gain_Gd** : normalise l'erreur en la divisant par la vitesse consigne Vc16.
- **PID_NNET** : représente le réseau de neurones d'architecture [1,3,1] définie par la simulation logicielle, dont les données sont représentées en virgule fixe sur 16 bits. La fonction d'activation est approximée par un polynôme d'ordre trois (3).
- **Gain_Gm** : calcule la commande nécessaire en multipliant la sortie du réseau de neurones par la vitesse consigne VC16.
- **Conv16_8** : réalise la conversion de la commande U16, qui est représentée en virgule fixe, en donnée sur 8 bits pour attaquer le convertisseur N/A.

II.5.6 Résultats de la synthèse

Les résultats de la synthèse pour l'implémentation du système de commande sont donnés par la table 2.12.

APPLICATION / RESSOURCES	SYSTEME DE COMMANDE
Entrées/Sorties I/O	33 / 324 (10%)
CLB Slice	2453 / 5120 (47%)
Mult 18x18s	34 / 40 (85%)
LUTs	4583 / 10240 (44%)
Fréquence max	3.285 Mhz

Table 2.12: *Résultats de synthèse pour l'implémentation du système de commande.*

II.5.7 Résultats expérimentaux

Les courbes suivantes représentent les résultats expérimentaux obtenus, avec Vc et Vr représentant, respectivement, les deux vitesses consigne et réelle. L'acquisition de ces résultats est faite par une carte DSPACE montée sur PC. Les signaux de consigne sont délivrés par un GBF. Nous avons commencé par prélever la réponse du système en boucle ouverte, puis nous avons inséré le système global de commande décrit auparavant.

Dans tout ce qui suit, l'allure des vitesses est exprimée en volt, ce qui correspond à la tension prélevée au niveau du tachymètre. Pour les transformer en tr/min, il suffit de faire la division par le coefficient 0,007.

II.5.7.1 La réponse du système en boucle ouverte

La figure 2.21 montre la réponse du système en boucle ouverte, en utilisant un signal carré d'amplitude variable [2,9v – 4,7v] comme consigne de la vitesse désirée [415tr/min - 670tr/min].

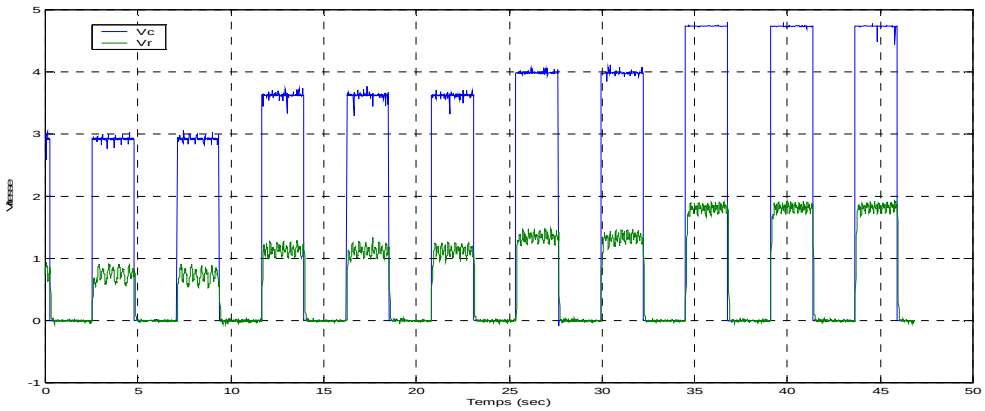


Figure 2.21: *Réponse du MCC en boucle ouverte pour un signal carré à amplitude variable.*

Il en ressort que le système en boucle ouverte est stable mais présente une erreur statique élevée.

II.5.7.2 La réponse du système en boucle fermée

Les résultats suivants représentent les réponses du MCC en utilisant le système de commande. Premièrement, nous avons prélevé la réponse du MCC pour une entrée échelon, puis pour des signaux carrés pour différents points de fonctionnement.

a- Entrée échelon

La réponse indicielle du MCC est donnée par la figure suivante pour un échelon de vitesse de 428tr/min qui correspond à une tension de 3v.

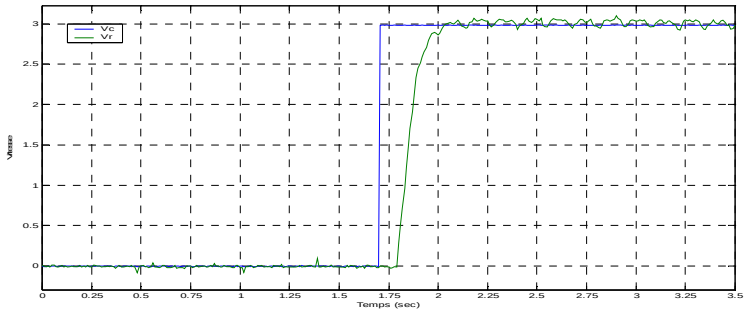


Figure 2.22: *Réponse indicielle du MCC en boucle fermée pour un échelon de vitesse 428 tr/min (≅3 v).*

b- Entrée signal carré

Les figures 2.23, 2.24, 2.25 et 2.26 illustrent la réponse du MCC en utilisant un signal carré de fréquence 0.22 Hz. Nous avons variés l'amplitude du signal d'entrée de 2v à 4v, ce qui correspond à une plage de vitesse de 280 tr/min à 560 tr/min.

1- Vitesse 280 tr/min (1.96v)

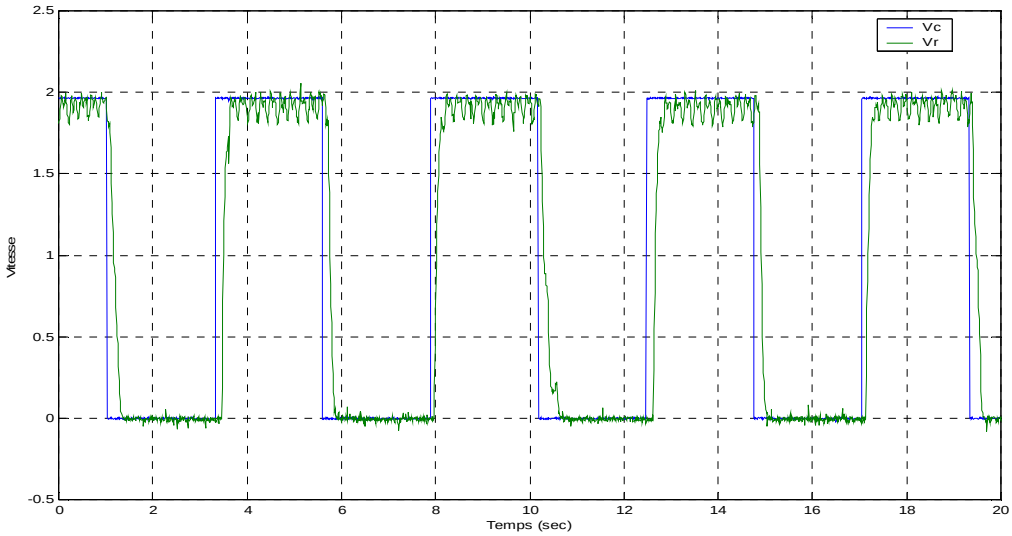


Figure 2.23: *Réponse du MCC en boucle fermée pour un signal carré de vitesse 280 tr/min (1.96 v).*

2- Vitesse 428 tr/min (≅3 v)

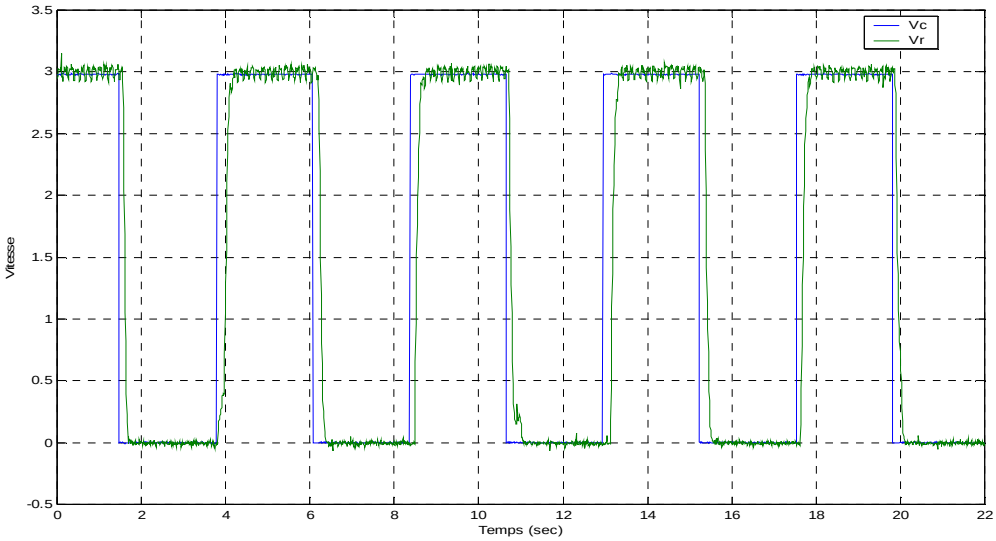


Figure 2.24: *Réponse du MCC en boucle fermée pour un signal* carré de vitesse 428 tr/min (≅3 v).

3- Vitesse 480 tr/min (≅ 3.35 v)

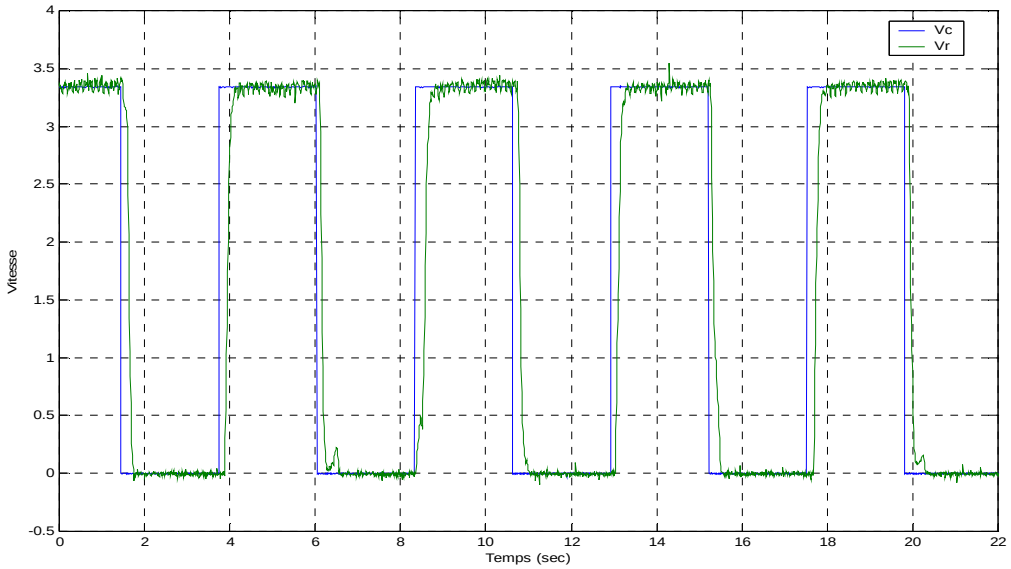


Figure 2.25: *Réponse du MCC en boucle fermée pour un signal carré de vitesse 480 tr/min (≅ 3.35 v).*

4- Vitesse 560 tr/min (≅3.92 v)

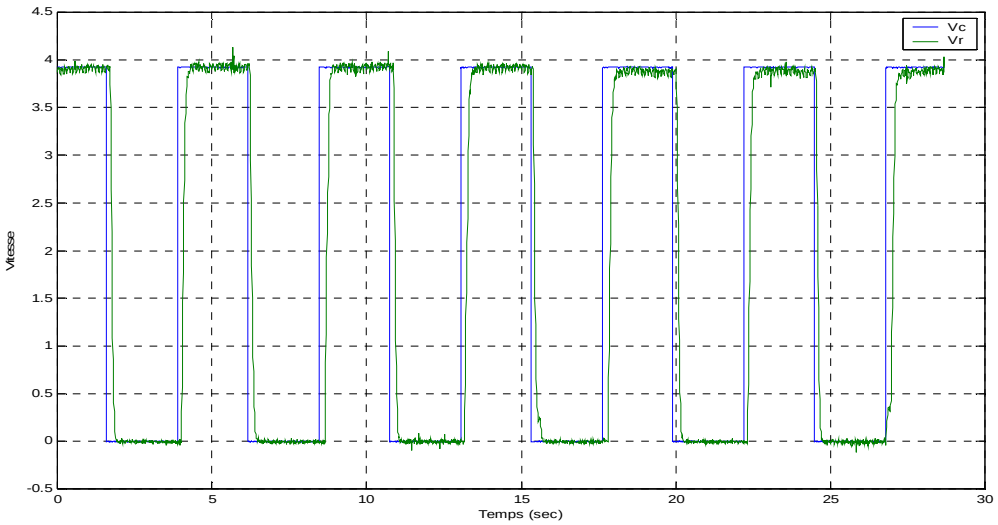


Figure 2.26: *Réponse du MCC en boucle fermée pour un signal carré de vitesse 560 tr/min (≅3.92 v).*

Pour des vitesses consignes inférieures à 280 tr/min, le système de commande ne peut pas compenser l'erreur statique car, dans ce cas, le modèle du Moteur n'est plus valable.

II.9 Conclusion

Les résultats de synthèses ont montré une grande précision atteinte par la représentation en virgule flottante, qui devient une performance très intéressante surtout pour une implémentation « *On-chip learning* », mais nécessitant une quantité de ressources assez importante.

Pour résoudre ce problème, la représentation en virgule fixe peut réaliser un bon compromis précision/ressources. C'est une représentation qui permet de concevoir des architectures intégrant la phase d'apprentissage et de généralisation sur le même circuit.

Concernant l'implémentation de la fonction d'activation, nous avons constaté que l'approximation linéaire est, à la fois, la moins onéreuse en ressources et la plus rapide. Alors que celle basée sur le polynôme approximatif, peut garantir une meilleure précision.

Le développement et la conception de l'implémentation ont été validé par une application basée sur l'approximation d'un PID pour la commande en vitesse d'un moteur à courant continu.

Les résultats obtenus montrent la capacité de régulation en vitesse du système de commande pour différents points de fonctionnement.

La souplesse de programmation et la reconfiguration totale du circuit FPGA (carte RC200) offre des possibilités d'amélioration, éventuelle, de l'architecture du réseau de neurone implémenté.

Le système de commande peut être utilisé pour implémenter d'autres types de contrôleur.

Dans le chapitre suivant, nous exposerons l'approche adoptée pour l'implémentation des RNAs, sur la carte RC200, en langage Handel-C.

Chapitre III

IMPLEMENTATION DES RESEAUX DE NEURONES ARTIFICIELS EN LANGAGE HANDEL-C

Victor HUGO /
Philosophie prose / Océan / OEuvres complètes / Robert Laffont - Bouquins 1989

« Le progrès rapetisse la terre et grandit l'homme. »

< 1855-56 p.64 >

III.1 Introduction

Dans ce chapitre, nous présenterons l'approche adoptée pour l'implémentation des RNAs, sur la carte RC200, en langage Handel-C. La stratégie adoptée consiste dans un premier temps, à programmer l'algorithme de la rétropropagation en langage C sur calculateur. L'étape suivante est la réalisation du « portage » du programme conçu en langage C, vers le langage Handel-C. La dernière étape enfin, est l'implémentation de l'architecture neuronale conçue, sur la carte RC200 contenant le circuit FPGA XC2V100fg456 de *Xilinx*. En parallèle à cela, les résultats d'implémentation obtenus seront présentés.

Par la suite, et dans le but de valider l'approche adoptée pour l'implémentation des RNAs, sur la carte RC200, en langage Handel-C, une application sera exposée avec les résultats obtenus.

III.2 Conception de l'architecture neuronale

Les spécifications principales d'une architecture neuronale sont [Sakh05] :

- Le type du réseau (multicouche, récurrent, etc.).
- L'architecture (nombre d'entrées et de sorties, nombre de neurones, nombre de connexions…).
- Le type d'apprentissage: en ligne sur le circuit (o*n-chip learning)* ou hors du circuit (*off-chip learning*).
- Le nombre maximal de neurones et de connexions supporté par le circuit.
- La représentation des données réelles.
- L'approximation de la fonction d'activation.

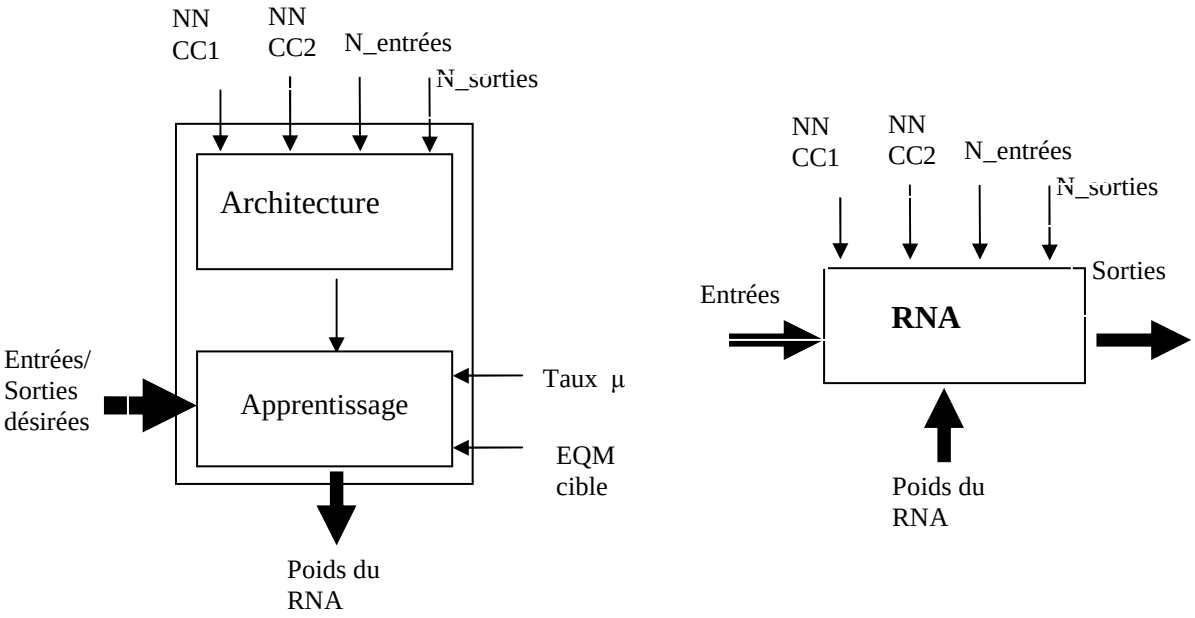
Dans ce travail, nous avons implémenté un RNA de type multicouche. Il est caractérisé par :

- Deux entrées au maximum.
- Deux sorties au maximum.
- Deux couches cachées.
- Quinze neurones au maximum par couche cachée.
- Fonction d'activation sigmoïde pour les couches cachées.
- Fonction d'activation linéaire pour la couche de sortie.

- L'algorithme d'apprentissage adopté est celui de la rétropropagation.

A noter que nous allons implémenter les deux types d'apprentissage. L'apprentissage en ligne et l'apprentissage hors ligne.

L'approche adoptée dans notre travail consiste à concevoir une architecture paramétrée (figure 3.1), c'est à dire qu'à chaque fois l'utilisateur aurait la possibilité de changer les paramètres du réseau de neurones (nombre de couches cachées, nombre d'entrées...), sans avoir à reprogrammer l'architecture matérielle.



a- Apprentissage en ligne. *b- Apprentissage hors ligne.*

Figure 3.1 : *L'architecture neuronale conçue.*

Nous avons choisi le langage Handel-C pour l'implémentation de l'architecture neuronale pour les raisons suivantes :

- Le développement d'une architecture matérielle avec le langage Handel-C nécessite un temps relativement faible, comparé aux langages de description des circuits comme VHDL ou Verilog.
- Le langage Handel-C possède une syntaxe similaire à celle d'un outil de programmation usuel en l'occurrence ANSI-C.

- Disponibilité de librairies écrites en Handel-C, livrées avec le logiciel DK de *Celoxica* pour l'utilisation des différents périphériques de la carte RC200.
- Le langage Handel-C contrairement au VHDL, permet de concevoir des architectures complètement paramétrables.

L'étape suivante dans la conception de l'architecture neuronale est le choix de la représentation des données réelles et de la méthode d'approximation de la fonction d'activation.

III.2.1 Représentation des données

Comme nous l'avons vu précédemment, l'un des problèmes qu'on rencontre lors de l'implémentation d'une architecture neuronale sur FPGA est le choix de la représentation des données réelles. Dans notre travail, nous avons essayé de tester la possibilité de l'utilisation d'une représentation en virgule fixe sur 32 bits, comme l'avaient recommandé certains auteurs [Eldr94], et de comparer ses performances avec une architecture sur 16 bits.

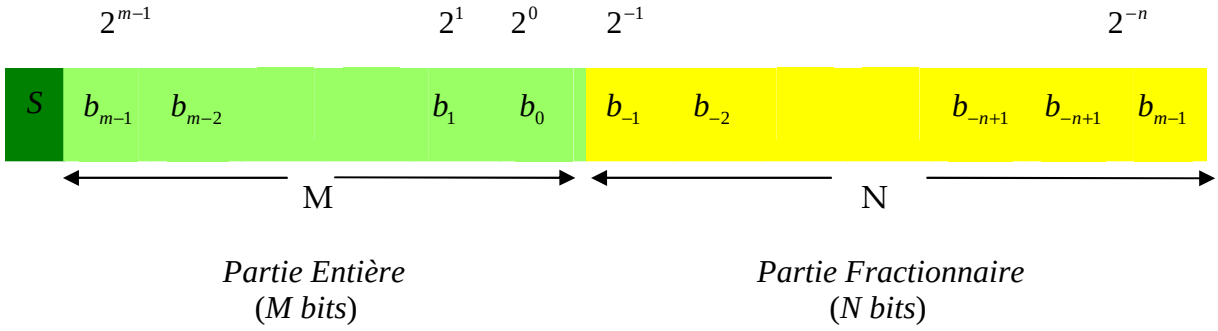


Figure 3.2 : *Représentation des données en virgule fixe.*

Nous avons adopté les représentations suivantes :

- Virgule fixe sur 16 bits 5Q11, c'est à dire 1 bit de signe, 4 bits pour la partie entière et 11 bits pour la partie réelle. L'erreur est 2^{-11}, ce qui est de l'ordre de 10^{-4}, la partie entière est dans un rang de -15 à +15. Ce choix permettrait de réduire les débordements tout en ayant une bonne précision.
- Virgule fixe sur 32 bits 8Q24, c'est à dire 1 bit de signe, 7 bits pour la partie entière et 24 bits pour la partie réelle. L'erreur est 2^{-24}, ce qui est de l'ordre de 10^{-7}, la partie

entière est dans un rang de -128 à +128 ce qui devrait apporter une grande souplesse par rapport à une représentation sur 16 bits, en réduisant encore plus les risques de débordement.

III.2.2 Approximation de la fonction d'activation

La fonction d'activation adoptée dans notre architecture est du type sigmoïde (figure 3.3), elle est donnée par :

$$f(x) = \frac{1}{1 + \exp(-x)} \qquad (3.1)$$

Comme la fonction exp(.) est absente des bibliothèques du langage Handel-C, il s'agit pour nous de choisir la meilleure approximation de la fonction sigmoïde en terme de précision / ressources.

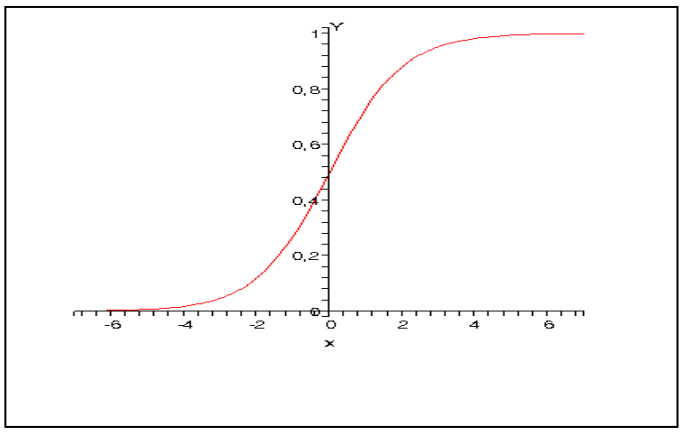


Figure 3.3 : *La fonction Sigmoïde.*

Plusieurs méthodes ont été développées pour l'approximation de la fonction sigmoïde [Nich03], [Pand05]. Dans cette partie de notre travail, nous avons adopté la méthode de développement en polynôme d'ordre n, qui présente le meilleur compromis entre la précision et la consommation des ressources (chapitre 2). Le polynôme approximatif d'ordre 5 dans l'intervalle I= [0 ,5] de la fonction sigmoïde s'écrit alors:

$$P(x)=0.4999935+ (0.2527613+ (-0.0046887 + (-0.0224609+$$

$$(0.0054178- 0.0003943.x).x).x).x).x \qquad (3.2)$$

L'estimation de la fonction se fait avec une erreur relative **de l'ordre de 0.689 e-3** (chapitre 2).

III.3 Etapes de l'implémentation en langage Handel-C

Nous avons d'abord implémenté le RNA en langage C sur P.C. Les simulations effectuées avaient permis de vérifier le bon fonctionnement du programme. L'étape suivante est le portage vers le langage Handel-C.

Le portage du langage C vers le Handel-C passe par les étapes suivantes :

- Le portage direct qui est la transcription du programme élaboré en C vers le Handel-C avec le minimum de changements.
- L'introduction des opérateurs spécifiques du Handel-C.
- L'introduction du parallélisme.

Le portage est mieux indiqué que le développement direct en Handel-C, à cause de la difficulté du « débogage » des réels avec le logiciel de développement DK de *celoxica* [Pand05].

III.3.1 Le portage direct

Dans cette étape, le programme est directement transcrit à partir du langage C en Handel-C avec le minimum de modifications. Le programme reste séquentiel mais doit être compilable en Handel-C. Le "fond" (l'algorithmique) reste inchangé, seule la "forme" (la syntaxe) est modifiée. Afin de vérifier que le programme fonctionne correctement, des simulations sont nécessaires.

III.3.2 Les opérateurs supplémentaires

Cette étape consiste à utiliser au mieux les différents opérateurs spécifiques au langage Handel-C, la réécriture de certaines expressions en utilisant les opérateurs supplémentaires doit permettre un gain de temps et des ressources du circuit ciblé. Le code ainsi modifié doit être plus "proche des portes" : en reflétant plus précisément l'implémentation matérielle (c'est à dire les circuits utilisées sur la carte FPGA). Par exemple l'utilisation des RAM au lieu des tableaux etc...

III.3.3 Le parallélisme

Contrairement au langage C, dont les instructions s'exécutent d'une manière séquentielle, le langage Handel-C dispose du parallélisme. Le temps d'exécution de deux instructions en parallèle est exactement égal au temps nécessaire à leur exécution par deux circuits totalement dissociés. La figure 3.4 illustre une architecture parallèle en Handel-C. Le

parallélisme n'est pas effectué d'une manière automatique par le logiciel, c'est le programmeur qui doit paralléliser l'architecture en fonction des contraintes matérielles (accès au RAM's....).

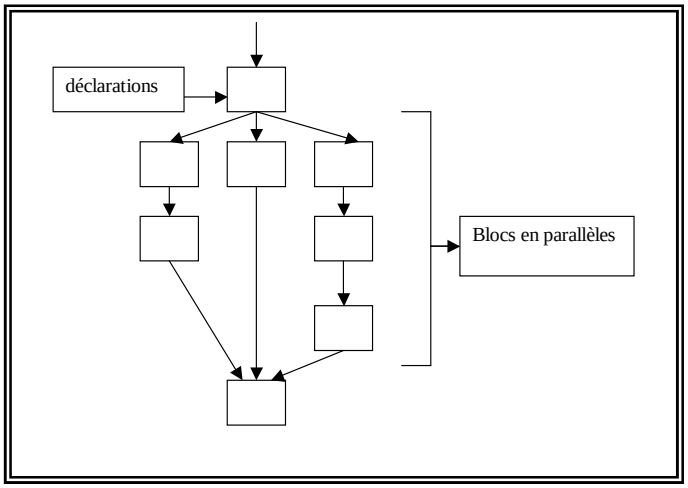


Figure 3.4 : *Structure du parallélisme en Handel-C.*

III.4 Validation de l'architecture neuronale

A l'issue de la phase du portage et avant de passer à l'implémentation du RNA sur la carte RC200, nous avons effectué des simulations afin de valider l'architecture conçue. Nous avons appliqué le RNA à l'approximation de fonction sin(x) et par la suite à un *benchmark* très utilisé qui est le XOR logique.

III.4.1 Approximation de la fonction sin(x)

L'architecture développée dans notre travail a été appliquée à l'apprentissage en ligne de la fonction sin(x). L'ensemble d'apprentissage est constitué de 20 éléments dans l'intervalle $[0,2\pi]$. Nous avons élaboré une architecture [1,5,1] (1 neurone dans la couche d'entrée, une seule couche cachée avec 5 neurones et 1 neurone dans la couche de sortie).

Pour la représentation en virgule fixe 32 bits les paramètres de l'apprentissage sont :

EQM cible =1e-4.

Taux d'apprentissage μ=0.6.

Le processus d'apprentissage a nécessité 8366 itérations pour converger.

Pour la représentation en virgule fixe 16 bits les paramètres de l'apprentissage sont :

EQM cible =0.002 .

Taux d'apprentissage μ=0.6 .

Le processus d'apprentissage a nécessité 698 itérations pour converger.

Les valeurs de la fonction sin(x) pour différentes entrées sont données par le tableau 3.1 :

x / sin(x)	0	π/2	π	3π/2	2 π
Virgule fixe 16 bits	-0.0004	0.9293	0.0581	-0.9750	-0.0004
Virgule fixe 32 bits	0.0009	0.9868	0.0033	-0.9867	0.0009

Tableau 3.1 : *Approximation de la fonction sin(x).*

Nous constatons que nous avons obtenu une bonne approximation de la fonction sin(x), bien que pour la représentation en virgule fixe l'erreur obtenue est de l'ordre de 1e-3, elle pourrait constituer néanmoins une bonne approximation pour la résolution des problèmes réels et c'est ce que nous verrons plus tard.

III.4.2 Application au problème du XOR logique

Le XOR logique a été utilisé dans beaucoup de travaux comme *benchmark* pour valider des architectures neuronales [Nich04]. C'est une fonction qui illustre parfaitement un problème de non linéarités séparables, elle permet de tester la capacité de classification d'une architecture neuronale donnée (Figure 3.5).

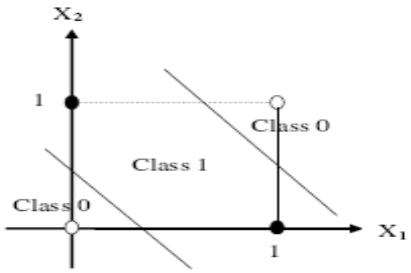


Figure 3.5 : *Position des ensembles dans un problème de XOR logique.*

Comme il a été montré [Ferr94], l'approximation du XOR nécessite, au moins, une couche cachée. Nous avons, alors, adopté une architecture [2,3,3,1] : 2 neurones dans la

couche d'entrée, deux couches cachées avec trois neurones chacune et un neurone dans la couche de sortie.

Pour la représentation en virgule fixe 32 bits les paramètres de l'apprentissage sont :

EQM cible =1e-5.

Taux d'apprentissage μ=0.6.

Le processus d'apprentissage a nécessité 1618 itérations pour converger.

Pour la représentation en virgule fixe 16 bits les paramètres de l'apprentissage sont :

EQM cible =0.0005.

Taux d'apprentissage μ=0.6.

Le processus d'apprentissage a nécessité 323 itérations pour converger.

Le tableau suivant présente les sorties estimées de la fonction pour chaque type de représentation :

Entrées		**Sortie désirée**	**Sortie estimée (32bits)**	**Sortie estimée (16bits)**
X0	X1	Y	$\hat{Y}$	$\hat{Y}$
0	0	0	0.0014	0.0151
0	1	1	1.0007	1.0141
1	0	1	0.9989	0.9763
1	1	0	-0.0000	0.0083

Tableau 3.2 : *Résultats de l'approximation du XOR logique.*

Nous constatons que nous avons obtenu une bonne approximation de la fonction XOR avec les deux types de représentation. Toutefois, la représentation sur 32 bits permet d'obtenir de meilleurs résultats. Ceci est du au fait que ce type de représentation des données réelles permet d'obtenir des erreurs cibles de l'ordre 1e-7, tout en évitant les débordements pendant le processus d'apprentissage.

Une fois l'architecture validée, l'étape suivante est la génération du fichier de configuration du circuit FPGA.

III.5 Génération du fichier de configuration

La génération du fichier de configuration passe, tout d'abord, par les étapes de synthèse. Il en résulte un fichier EDIF, qui va être utilisé pour générer le fichier de configuration BIT (*bitfile*) du circuit FPGA, à l'issue des étapes de placement, de routage et d'optimisation, à l'aide du logiciel ISE de *xilinx* [Xili06]. Le fichier BIT va être chargé dans la carte RC200 [Annexe 1], afin de configurer le circuit FPGA.

L'emploi de librairies spécialisées du logiciel DK-3 de *Celoxica*, en l'occurrence les librairies : PAL (*Platform Abstraction Layer*), RC200PSL (*Platform Support Library*) avait facilité l'implémentation de l'architecture neuronale sur la carte RC200 car ces librairies possèdent des fonctions pour le transfert automatique des données entre « l'ordinateur hôte » et la carte, via le port parallèle.

III.5.1 Implémentation d'une architecture avec apprentissage hors ligne

La figure 3.6 détaille l'algorithme implémenté en Handel-C avec un apprentissage hors ligne (*off chip*). Ce type d'implémentation intègre seulement la phase de généralisation. L'apprentissage est réalisé, antérieurement, en simulation logicielle afin de générer les paramètres du RNA.

- *Signal de synchronisation présent ?*
- *Lecture des données de la mémoire tampon du port parallèle*
  - *Caractéristiques du RNA*
  - *Entrées du RNA*
  - *Poids du RNA*
- *Propagation des données dans le RNA*
- *Ecriture des sorties du RNA dans la mémoire tampon du port parallèle*

Figure 3.6 : *Algorithme de l'apprentissage hors ligne.*

A l'issue des étapes de placement, de routage et d'optimisation et à partir des résultats obtenus, nous remarquons que la représentation des données sur 32 bits a donné lieu à une grande consommation des ressources du circuit comparée avec l'architecture sur 16 bits (tableau 3.3). Ceci laisse prévoir l'impossibilité de réaliser une implémentation en ligne avec des données réelles représentées sur 32 bits.

Représentation / Ressources	Virgule fixe (16bits)	Virgule fixe (32 bits)
IOBs	20 / 324 (6%)	20 / 324 (6%)
CLBs	2024 / 5120 (39%)	4633 / 5120 (90%)
MULT 18X18s	13 / 40 (32%)	9 / 40 (22%)
LUTs	3656 / 10240 (35%)	8765/10240 (85%)
Fréquence	38,62Mhz	15,35Mhz

Tableau 3.3 : *Résultats de la synthèse pour l'apprentissage off chip.*

III.5.2 Implémentation d'une architecture avec apprentissage en ligne

La figure 3.7 détaille l'algorithme implémenté pour un apprentissage en ligne (*on chip*), en Handel-C. Ce type d'apprentissage intègre les phases d'apprentissage et de généralisation sur un même circuit.

- *Signal de synchronisation présent ?*
- *Lecture des données dans la mémoire tampon du port parallèle*
  - *Caractéristiques du RNA*
  - *Paramètres de l'apprentissage*
  - *Entrée et Sortie désirées*
  - *Poids initiaux*
- *Apprentissage*
- *Ecriture des résultats dans la mémoire tampon du port parallèle*
  - *Résultats de l'apprentissage*
  - *Sorties du RNA*
  - *Poids du RNA*

Figure 3.7 : *Algorithme de l'apprentissage On-chip.*

A l'issue des étapes de placement, routage et optimisation, nous constatons un dépassement des ressources pour la représentation des données sur 32 bits en virgule fixe (tableau 3.4).

Représentation / Ressources	Virgule fixe (16bits)	Virgule fixe (32 bits)
Entrées/Sorties (I/O)	20 / 324 (6%)	20 / 324 (6%)
CLBs Slice	2748 / 5120 (53%)	13026 / 5120 (254%)*
Mult 18x18s	32 / 40 (80%)	15 / 40 (37%)
LUTs	4926 / 10240 (48%)	25973 / 10240 (253%)*
Fréquence max	26,1 Mhz	14,63 Mhz

Tableau 3.4 : *Résultats de synthèse pour l'implémentation On-chip learning.*

A partir de ces résultats, le choix de la représentation des données va finalement être porté sur la virgule fixe sur 16 bits, pour l'architecture incluant la phase d'apprentissage (*on chip learning)*. Le circuit disponible ne permet pas en effet, de réaliser un RNA avec des données réelles représentées sur 32 bits.

Cependant, pour l'architecture n'incluant pas la phase d'apprentissage (*off chip learning)*, nous allons utiliser une représentation sur 32 bits car elle procure une meilleure précision.

III.6 Application: Commande en position d'un moteur à courant continu

La mise en œuvre pratique de la commande pour l'asservissement d'un moteur à courant continu est donnée par la figure 3.8. Nous utilisons un RNA à deux entrées : la position désirée Yc(t) et la sortie du système à l'instant d'échantillonnage précédent Y(t-1), pondérée par un coefficient α. L'apprentissage du réseau se fait par la différence e(t), entre la consigne Yc(t) et la position actuelle du moteur Y(t).

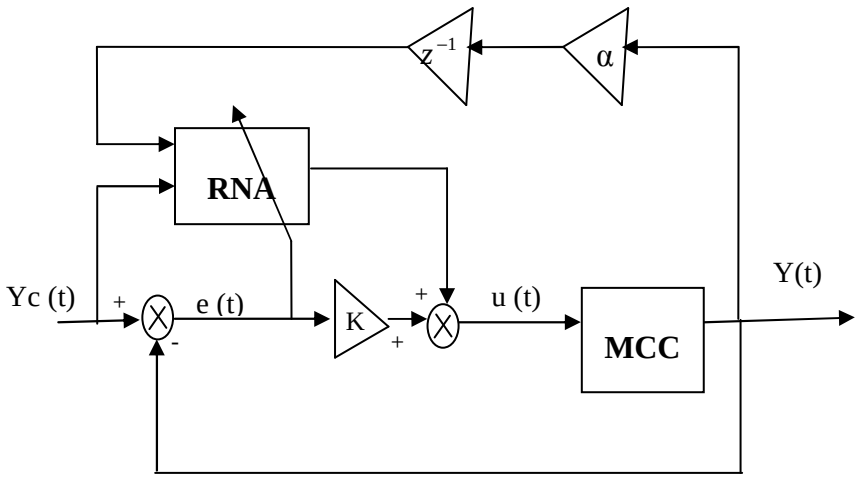


Figure 3.8 : *Commande en position d'un moteur à courant continu*

III.6.1 Description du système de commande

Le système de commande (figure 3.9) est composé de [Benb17] :

- L'amplificateur de puissance et le moteur à courant continu.
- La carte RC200
- La carte d'interface (annexe 4).

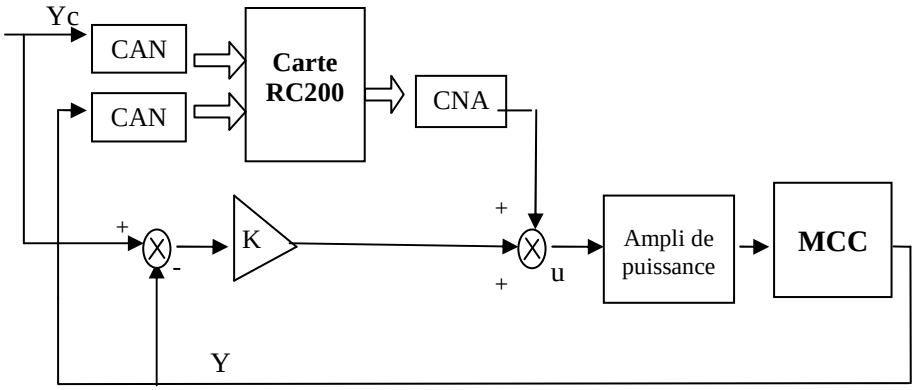


Figure 3.9 : *Description matérielle du système de commande*

III.6.1.1 Le moteur à courant continu

Le Moteur à courant continu est un système qui possède comme entrée la tension $u(t)$ qui représente la valeur moyenne du signal MLI (Modulation en Largeur d'Impulsion) et dont la sortie est une tension analogique prélevée du potentiomètre.

Il est représenté dans le domaine fréquentiel par une fonction de transfert du second ordre (dans le cas d'un asservissement de position).

$$F(p) = \frac{G_0}{p(1+\tau.p)} \tag{3.1}$$

Tel que G_0 : est le gain statique,

τ : est la constante du temps.

Les valeurs des paramètres identifiés sont les suivantes (chapitre 4) :

$$\begin{cases} G_0 = 0.4 \\ \tau = 0.034\ s \end{cases} \tag{3.2}$$

III.6.1.2 La carte d'interface

La carte d'interface (annexe 4) est composée essentiellement de CAN et de CNA. Son rôle est de conditionner les grandeurs analogiques Yc(t) et Y(t), afin de les convertir en données numériques et de convertir les données numériques, de la commande U(t), issues de la carte RC200 en grandeurs analogiques. Ce signal de commande sera amplifié pour attaquer le moteur à courant continu.

III.6.1.3 Description de l'architecture neuronale

Nous utilisons comme entrées pour le RNA, la consigne Yc(t) et la sortie précédente Y(t-1), pondérée par un facteur α. L'architecture conçue est différente de la précédente par le fait que l'erreur utilisée pour la mise à jour des poids lors de la phase d'apprentissage, n'est pas la différence entre la sortie du RNA et la sortie désirée, car celle-ci est inconnue, mais c'est la différence entre la sortie désirée du système et la sortie mesurée.

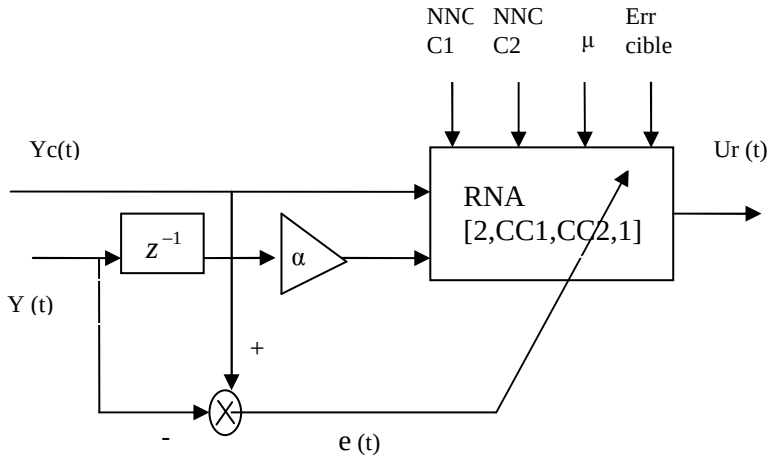


Figure 3.10 : *Description de l'architecture neuronale pour la commande*

La figure 3.11 explicite le fonctionnement de la commande avec la phase d'apprentissage et la phase de fonctionnement (généralisation).

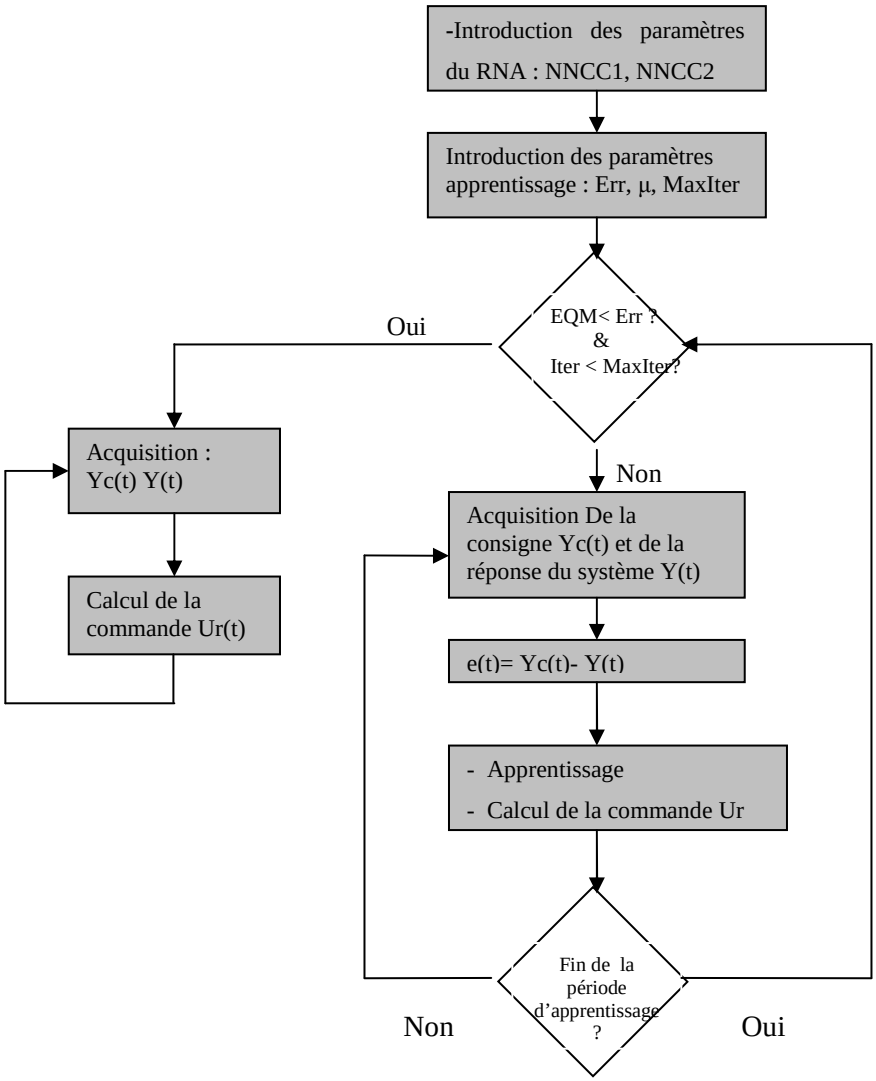


Figure 3.11 : *Principe de l'implémentation de la commande*

III.6.2 Evolution du système durant la phase d'apprentissage

Afin d'observer l'évolution du système de commande au cours de l'apprentissage, nous l'avons réglé avec les paramètres suivants :

- Nombre de couches cachées = 1.
- Nombre de neurones dans la couche cachée =3.
- Taux d'apprentissage = 0.08.
- Erreur Quadratique moyenne désirée = 0.0375.
- Gain du correcteur linéaire K=1.
- le facteur de pondération de Y(t-1), α=1.
- La consigne est un signal carré de période 2s et d'amplitude : 57° (4v).

La figure 3.12 permet de visualiser la réponse du système lors de la phase d'apprentissage.

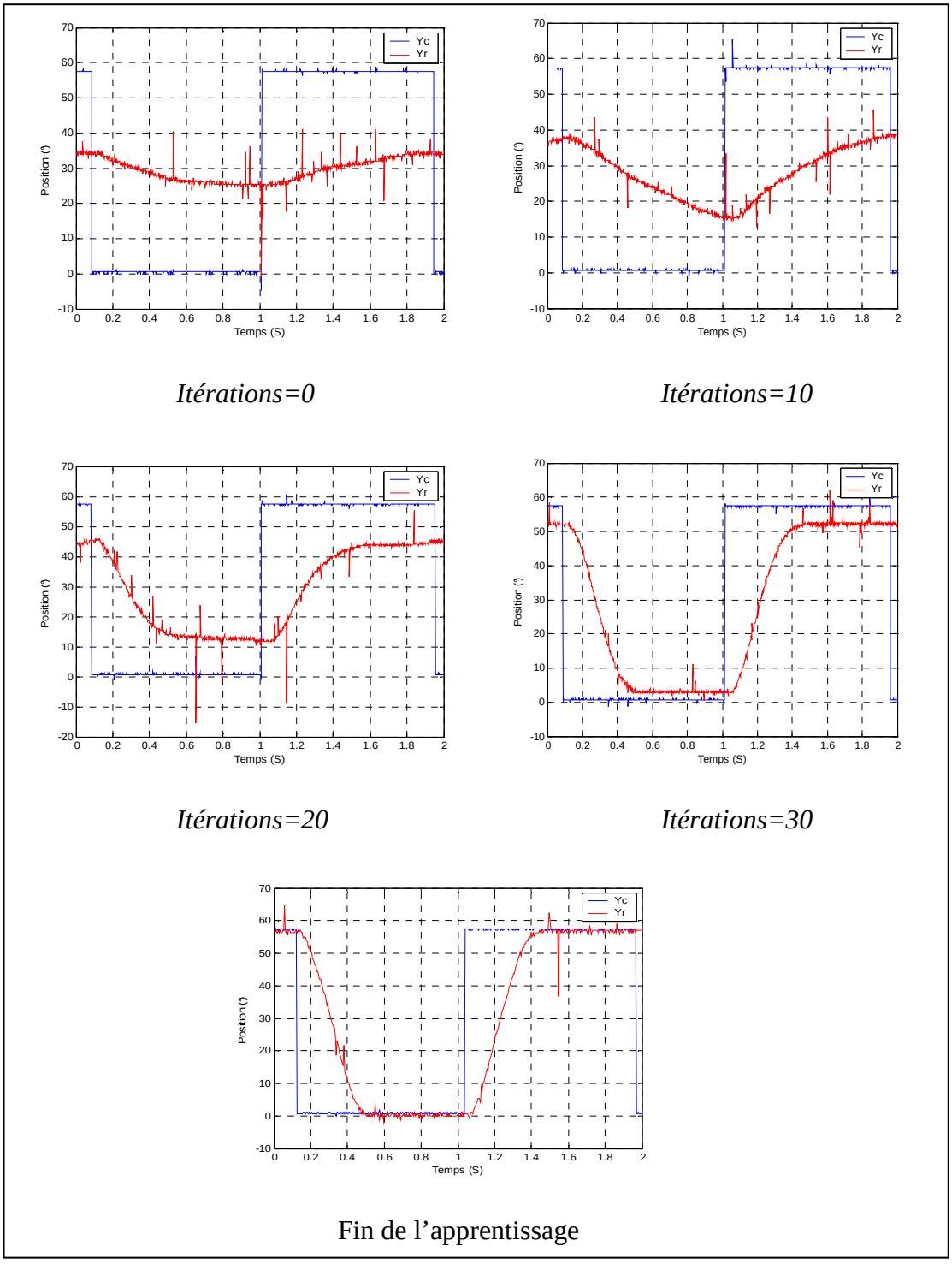


Figure 3.12 : *Evolution de la sortie du système durant l'apprentissage*

Le système de commande a nécessité 42 itérations pour atteindre l'EQM cible. Nous constatons que l'erreur reste bornée et décroissante durant l'apprentissage, ce qui caractérise la stabilité du système de commande.

III.6.3 Test de la capacité de généralisation du système de commande

Une fois la phase d'apprentissage complétée, le réseau pourra être utilisé pour obtenir la réponse à des vecteurs d'entrée inconnus, c'est à dire non rencontrés durant l'apprentissage, ce qui constitue la phase dite de généralisation. Pour ce faire, nous faisons varier l'amplitude de la consigne pour le système auquel nous avons réalisé l'apprentissage à l'étape précédente. Ceci nous permet de prélever les sorties correspondant aux différentes valeurs de l'entrée (Figure 3.13).

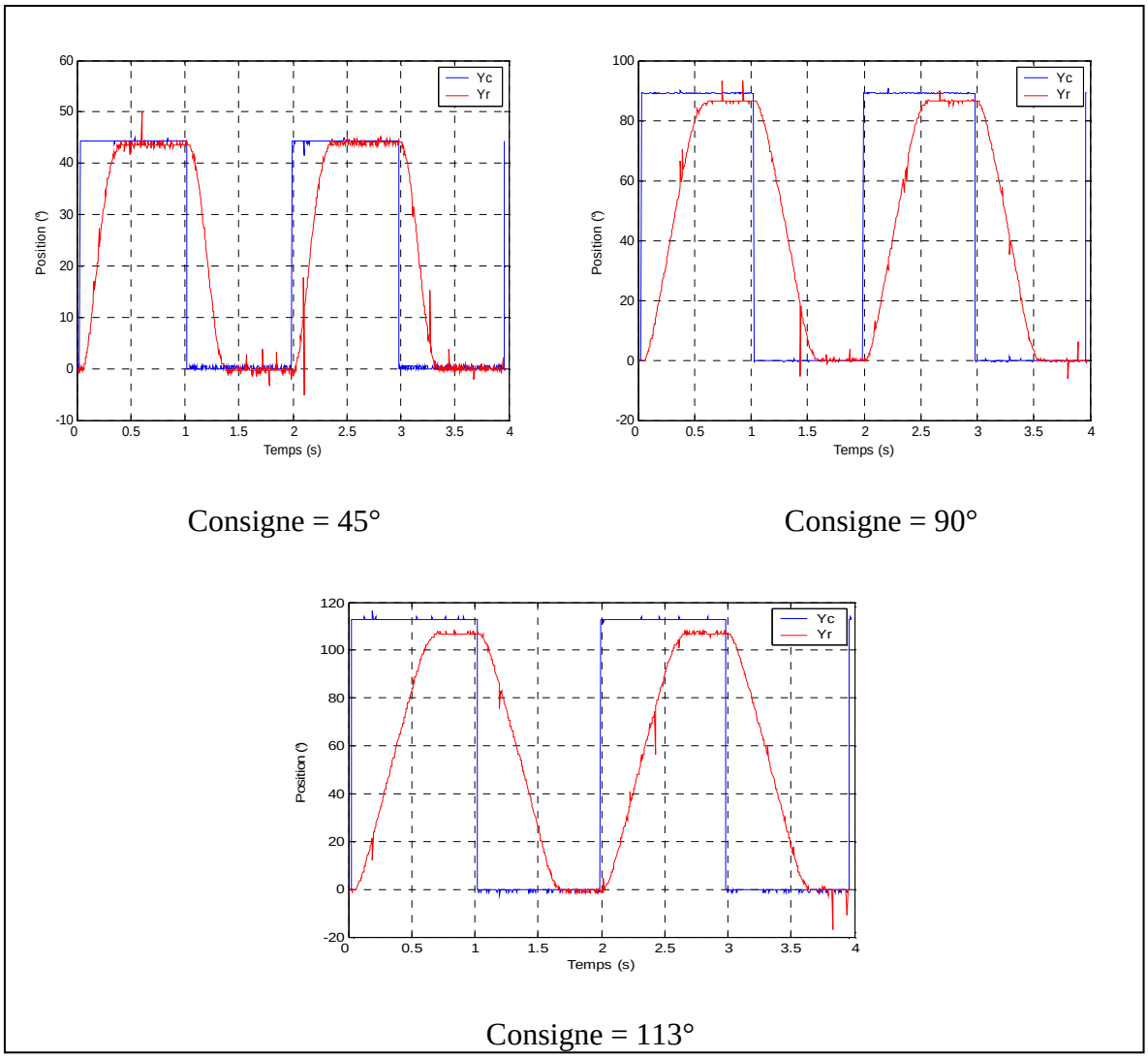


Figure 3.13 *: Réponse du système de commande pour différentes valeurs de la consigne*

Nous remarquons l'apparition d'une erreur statique, si on augmente l'amplitude de la consigne. Cette erreur statique reste néanmoins négligeable (de l'ordre de 5%).

Cette expérience nous a permis de mettre en valeur la capacité de généralisation du système de commande, qui a « appris » avec une certaine valeur de la consigne et qui arrive à suivre des consignes « inconnues ».

III.6.4 Test de la robustesse de la commande

Le but de cette expérience est de garantir la stabilité de toute la boucle de commande. On cherche dans ce cas à garantir que quelle que soient les perturbations que subit le processus, la commande est apte à le faire converger vers son état stable.

Nous avons procédé par l'application de perturbations sur le système de commande en injectant une impulsion avec le signal de commande (figure 3.14).

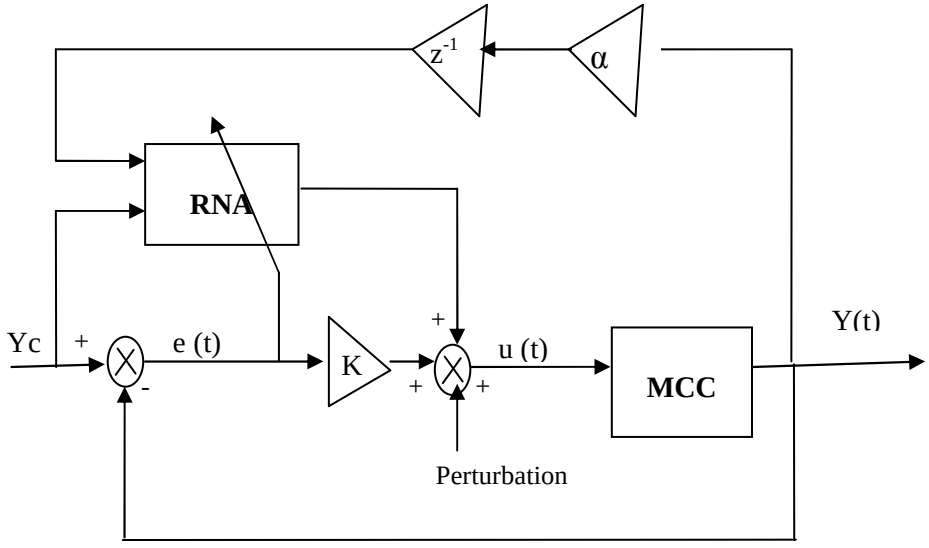


Figure 3.14 : *Principe de l'injection de perturbation dans le système*

Soit le système pour lequel nous avons réalisé l'apprentissage lors de la phase précédente. Nous le réglons ensuite avec une consigne continue Yc=60° et nous lui appliquons des perturbations sous forme d'impulsions.

Nous avons injecté une impulsion à t=0.3s. Nous constatons que le système reprend sont état initial sans présenter d'oscillations (figure 3.15).

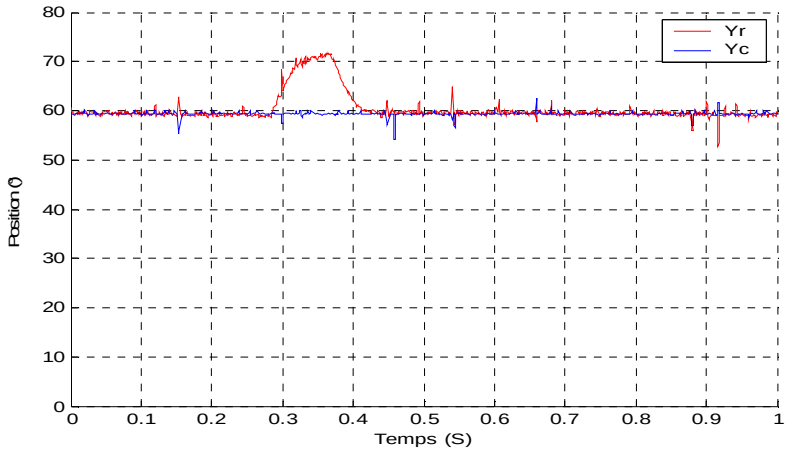


Figure 3.15 *: Réponse du système de commande aux perturbations*

Les résultats obtenus illustrent parfaitement la capacité d'abréger des RNAs, ce qui leur permet de résister aux bruits et aux perturbations et de garantir ainsi la robustesse du système de commande.

III.6.5 Robustesse par rapport à une panne du contrôleur linéaire

Sur le même système et après la phase d'apprentissage, nous coupons la sortie du contrôleur proportionnel, le système a l'allure de la figure 3.16 :

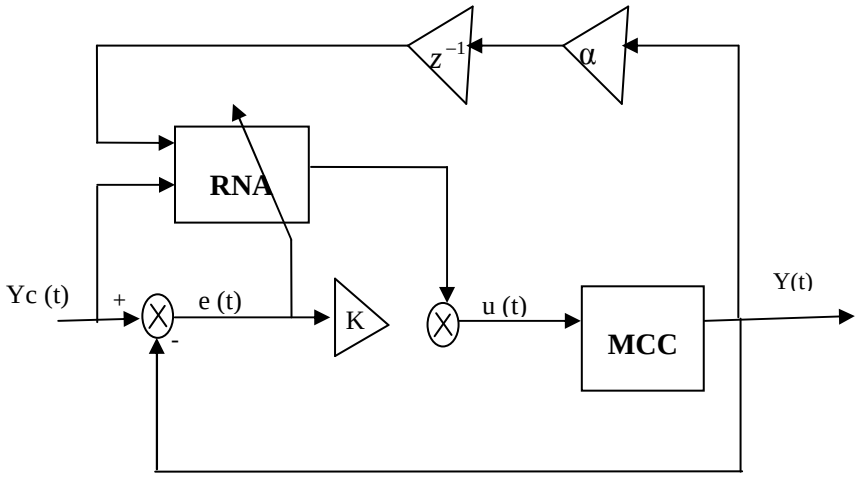


Figure 3.16 : *Le système de commande avec une panne du contrôleur linéaire*

Nous remarquons (figure 3.17) la présence d'une erreur statique, due à l'absence du signal de commande fourni par le contrôleur linéaire. Néanmoins cette erreur est négligeable, étant inférieure à 5% du régime statique. Ce résultat confirme l'hypothèse selon laquelle

l'importance est donnée à la commande préconisée par le contrôleur neuronal au fur et à mesure de l'apprentissage.

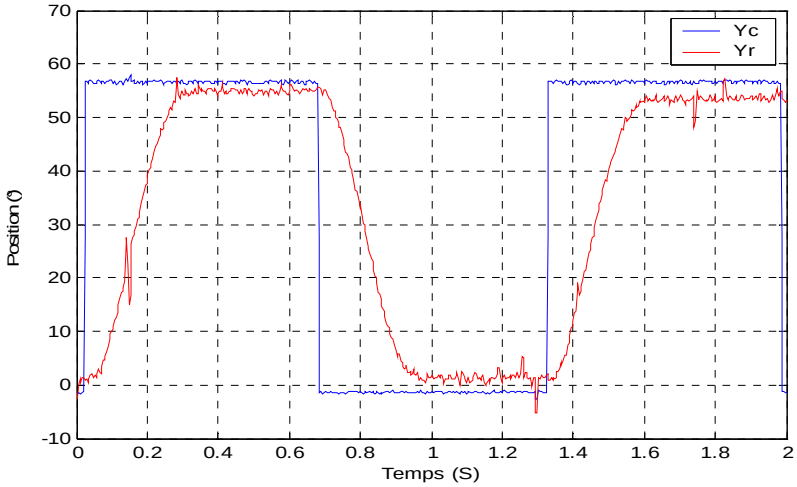


Figure 3.17 : *Réponse du système à une panne du contrôleur linéaire*

III.6.6 Effet du facteur de pondération de la sortie précédente

Afin d'observer l'effet du facteur de pondération de la sortie à l'instant d'échantillonnage précédent, sur l'apprentissage, nous avons réglé le système avec les paramètres suivants :

- Nombre de couches cachées = 1.
- Nombre de neurones dans la couche cachée = 3.
- Taux d'apprentissage = 0.08.
- Erreur Quadratique moyenne désirée = 0.0375.
- Gain du correcteur linéaire K = 1.
- La consigne est un signal carré de période 2 s et d'amplitude : 60° (4 v).

Pour différentes valeurs du facteur de pondération α, nous avons mesuré le nombre d'itérations nécessaires pour atteindre l'EQM cible. Ceci nous a permis de dresser le tableau suivant :

Facteur de pondération α	Nombre d'itérations
0	Pas de convergence
0.5	Pas de convergence
0.7	26
0.8	28
1	32
1.2	Pas de convergence

Tableau 3.5 : *Effet du facteur de pondération α sur la convergence du système.*

Nous remarquons qu'il n'y a pas de convergence de la commande pour une valeur du facteur α inférieure à 0.7, c'est à dire que l'EQM désirée n'est pas atteinte dans ce cas. Les réponses obtenues présentent souvent l'allure suivante (figure 3.18):

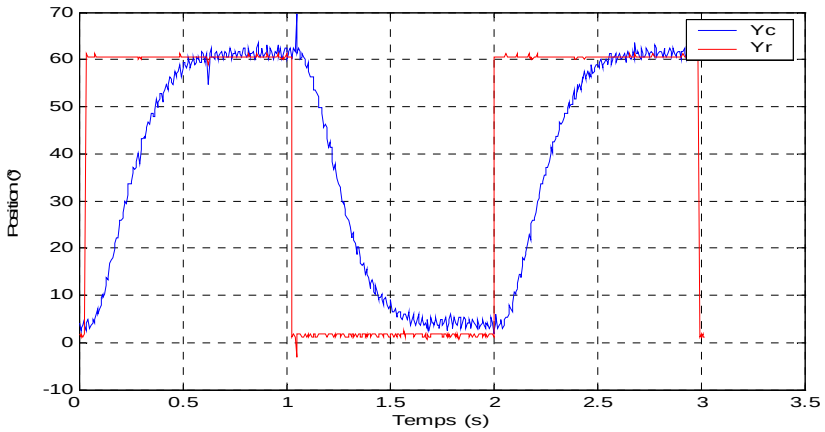


Figure 3.18 : *Réponse du système pour α =0.5 et un nombre d'itérations = 22.*

Pour une valeur du facteur de pondération supérieure à un, nous remarquons qu'il n'y a pas de convergence de la commande et qu'à partir d'un certain nombre d'itérations, nous constatons l'apparition d'oscillations sur la réponse du système (figure 3.19) :

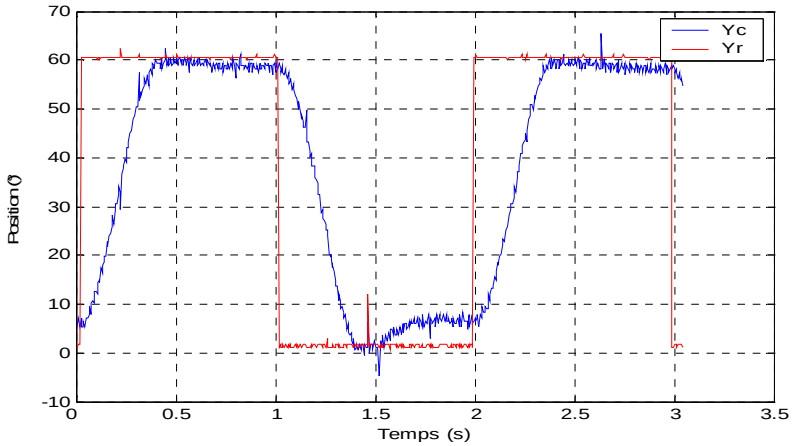


Figure 3.19 : *Réponse du système pour α =1.2 et un nombre d'itérations = 32.*

Nous en concluons que la convergence du système de commande nécessite d'effectuer un bon choix du facteur de pondération de la sortie précédente.

III.6.7 Effet de l'architecture du réseau de neurones

Afin d'étudier l'effet de l'architecture du RNA (nombre de couches cachées et de neurones par couche cachée) sur la commande, nous avons réglé le système avec les paramètres suivants :

- Taux d'apprentissage = 0.08.
- Erreur Quadratique moyenne désirée = 0.0375.
- Gain du correcteur linéaire K=1.
- La consigne est un signal carré de période 2s et d'amplitude : 60° (4v).
- La facteur de pondération de Y(t-1), $\alpha = 0.8$.

Ensuite, pour une architecture à une couche cachée, nous avons varié le nombre de neurones et nous avons mesuré le nombre d'itérations nécessaires à chaque fois, pour atteindre l'EQM cible. Ceci nous a permis de dresser le tableau suivant :

Nombre de neurones dans la couche cachée	**Nombre d'itérations**
3	28
4	44
5	Pas de convergence
10	Pas de convergence

Tableau 3.6 *: Effet du nombre de neurones dans la couche cachée sur la convergence du système*

Nous constatons que le système se déstabilise au bout d'un certain nombre d'itérations, pour une architecture dépassant quatre neurones dans la couche cachée. La réponse du système prend souvent la forme suivante (figure 3.20) :

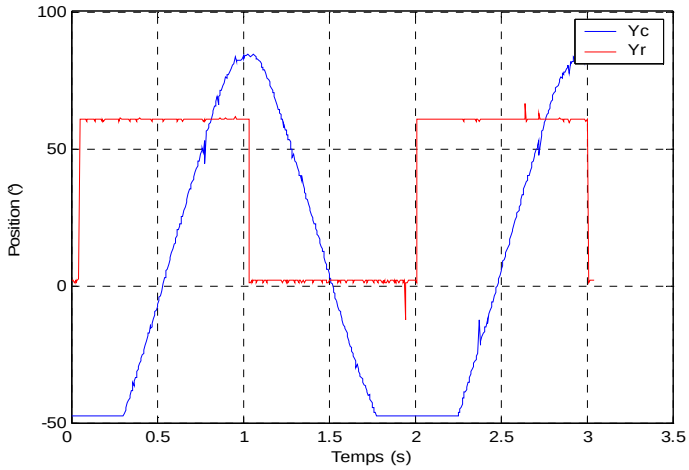


Figure 3.20 : *Réponse du système pour 5 neurones dans la couche cachée au bout de 30 itérations.*

A noter que sur les architectures à deux couches cachées, aucune configuration n'avait permis la convergence du système de commande.

III.7 Conclusion

Dans ce chapitre, nous avons exposé l'approche utilisée pour l'implémentation d'un réseau de neurones multicouches, sur la carte RC200, en langage Handel-C. Nous avons conçu une architecture à deux couches cachées et nous avons opté pour une représentation des données en virgule fixe. Nous avons utilisé l'algorithme de Remez pour l'approximation de la fonction d'activation sigmoïde.

La stratégie adoptée a consisté dans un premier temps, à programmer l'algorithme de rétropropagation en langage C sur calculateur. L'étape suivante a été la réalisation du « portage » du programme conçu en langage C, vers le langage Handel-C. La dernière étape enfin, a été l'implémentation de l'architecture conçue sur la carte RC200.

Pour une implémentation avec un apprentissage en ligne, nous avons adopté une représentation des données en virgule fixe sur 16 bits. Pour une implémentation avec un apprentissage hors ligne, nous avons adopté une représentation en virgule fixe sur 32 bits.

Le développement et la conception de l'implémentation ont été validé par une application basée la technique de l'amélioration d'un contrôleur linéaire (*feed-back error learning*) pour la commande en position d'un moteur à courant continu.

L'architecture neuronale développée pour la commande, inclut la phase d'apprentissage en ligne. Nous avons constaté que l'erreur reste bornée durant l'apprentissage, moyennant un choix précis de l'architecture neuronale et des paramètres de l'apprentissage. Nous avons aussi pu mettre en pratique les capacités de généraliser et d'abréger des RNAs, qui sont intéressantes pour les systèmes de commande.

Nous avons étudié aussi le comportement de la commande pour différentes valeurs du facteur de pondération de la sortie précédente du système et aussi pour différentes architectures neuronales.

Chapitre IV

INTERFACE GRAPHIQUE POUR L'IMPLEMENTATION DES RESEAUX DE NEURONES ARTIFICIELS SUR LES CIRCUITS FPGA

Benjamin FRANKLIN /
Mélanges de Morale, d'Économie et de Politique (t.1) / Paris, J.Renouard 1826 [BnF]

« Souvenez-vous que le temps est de l'argent. »

< *Avis à un jeune ouvrier*, 1748 p.111 >

IV.1 Introduction

Dans le chapitre précédent, nous avons exposé l'approche adoptée pour la conception et l'implémentation d'une architecture neuronale paramétrable. Dans ce chapitre, nous allons expliquer l'approche utilisée pour la réalisation du programme de téléchargement automatique du fichier de configuration sur la carte RC200, via le port parallèle, ensuite nous allons présenter l'interface graphique réalisée et expliquer ses fonctionnalités et enfin nous terminerons par quelques exemples d'applications pratiques.

L'implémentation des RNAs sur circuits FPGA, comme nous l'avons si bien constaté, peut nécessiter des mois de travail. Ceci réduit considérablement l'intérêt de l'emploi de ces circuits sur une large échelle. L'idée était donc, de réaliser une plateforme logicielle qui ferait abstraction de la programmation des circuits FPGA, ce qui aurait pour conséquence de réduire le temps de mise au point d'une architecture neuronale à quelques secondes seulement. Le principe de l'interface graphique est donné par la figure 6.1 :

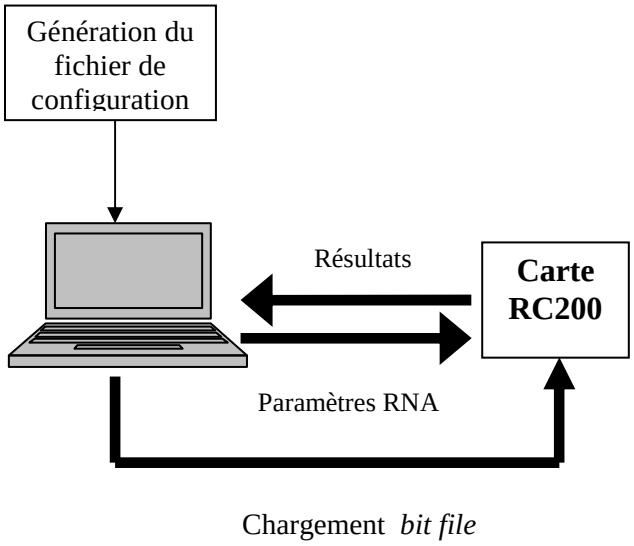


Figure 4.1 : *Principe de l'interface pour l'implémentation des RNAs sur FPGA.*

La stratégie adoptée pour la réalisation d'un environnement graphique pour la génération automatique d'un RNA sur la carte RC200 est la suivante [Benb05] et [Benb37]:

- L'implémentation d'une architecture neuronale paramétrable.

- La mise au point d'un programme de téléchargement automatique du fichier de configuration (*bitfile*) sur la carte RC200, de la transmission des paramètres de l'apprentissage et de la lecture des résultats du traitement.
- La réalisation d'une interface graphique pour l'utilisateur.

IV.2 Le programme de téléchargement automatique

En utilisant une librairie spécialisée, nous avons pu mettre au point un programme de chargement automatique à partir de l'ordinateur hôte (*host computer*), vers la carte RC200 via le port parallèle.

Il assure les tâches suivantes :

- La synchronisation entre le l'ordinateur hôte et la carte RC200.
- La configuration du circuit FPGA par le chargement du *bit file.*
- Le transfert des données entre la carte RC200 et l'ordinateur hôte.

La figure 6.2 détaille l'algorithme implémenté en langage C++ sur l'ordinateur hôte. La première étape est l'initialisation de la carte RC200, ensuite vient l'étape de configuration du circuit FPGA avec le bit file. Vient par la suite, l'étape de chargement dans la mémoire tampon du port parallèle, des paramètres de l'apprentissage (taux, EQM cible) et de la base d'apprentissage (entrées/ sorties).

Le début de l'apprentissage nécessite l'envoi d'un signal de synchronisation au circuit FPGA. La dernière étape, à la fin de l'apprentissage, est la lecture des résultats à partir de la mémoire tampon du port parallèle. Les données transmises ou celles lues à partir de la carte RC200, sont sauvegardées dans des fichiers spécifiques qui seront utilisés par le menu graphique.

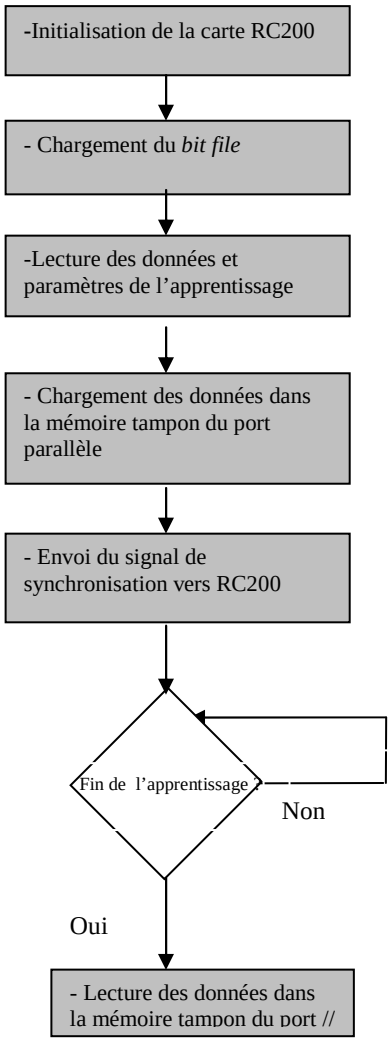


Figure 4.2 : *Algorithme de communication automatique P.C / carte RC200*

IV.3 L'interface graphique

L'interface graphique à menus est dédiée à l'utilisation rapide et conviviale de l'architecture développée. Elle propose une implémentation du modèle conçu sur le circuit FPGA, l'avantage qu'elle offre réside dans la facilité de l'approche : pas de langage informatique, elle nécessite seulement l'utilisation de menus graphiques (boutons de commande, zones d'édition...).

A partir de ces menus graphiques, l'utilisateur est invité à construire son application. Pour ce faire, il peut spécifier les différents paramètres du RNA (nombre de couches, nombre de neurones, etc...) ainsi que celles de l'apprentissage (taux de mise à jour des poids, nombre d'itérations maximales, etc…). Ce système apporte de nombreux outils pour construire le réseau de neurones et aussi pour contrôler l'exécution de l'apprentissage.

L'environnement graphique que nous avons réalisé avec *Microsoft visual C++* se compose de trois fenêtres : La fenêtre du menu principal, la fenêtre du menu d'apprentissage en ligne (on chip) et la fenêtre du menu d'apprentissage hors ligne (*off chip*).

IV.3.1 Le menu principal

La fenêtre du menu principal (figure 4.3) permet de spécifier l'architecture à implémenter, à savoir :

- Nombre d'entrées.
- Nombre de sorties.
- Nombre de couches cachées.
- Nombre de neurones dans chaque couche cachée.

L'utilisation de listes déroulantes permet d'éviter d'entrer des valeurs en dehors la taille maximale du RNA.

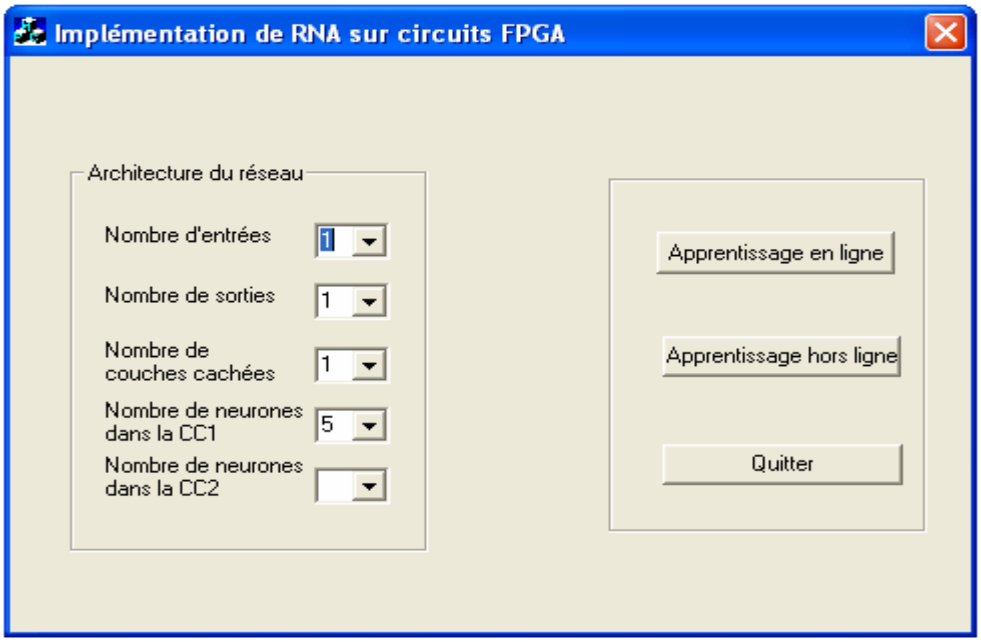


Figure 4.3 : *Le menu principal*

Dans ce menu principal nous disposons aussi de trois boutons de commande :

- **Le bouton « *Apprentissage en ligne* » :** Permet d'ouvrir la fenêtre de l'apprentissage en ligne.

- **Le bouton « Apprentissage hors ligne »** : Permet d'ouvrir la fenêtre de l'apprentissage hors ligne.
- **Le bouton « *Quitter* » :** Permet de quitter l'application**.**

IV.3.2 Le menu de l'apprentissage en ligne

La fenêtre *on chip learning* (figure 4.4) contient des zones d'édition qui permettent de sélectionner les paramètres de l'apprentissage en ligne, à savoir :

- Le taux d'apprentissage.
- L'erreur quadratique moyenne ciblée.
- Les fichiers des entrées et des sorties désirées.

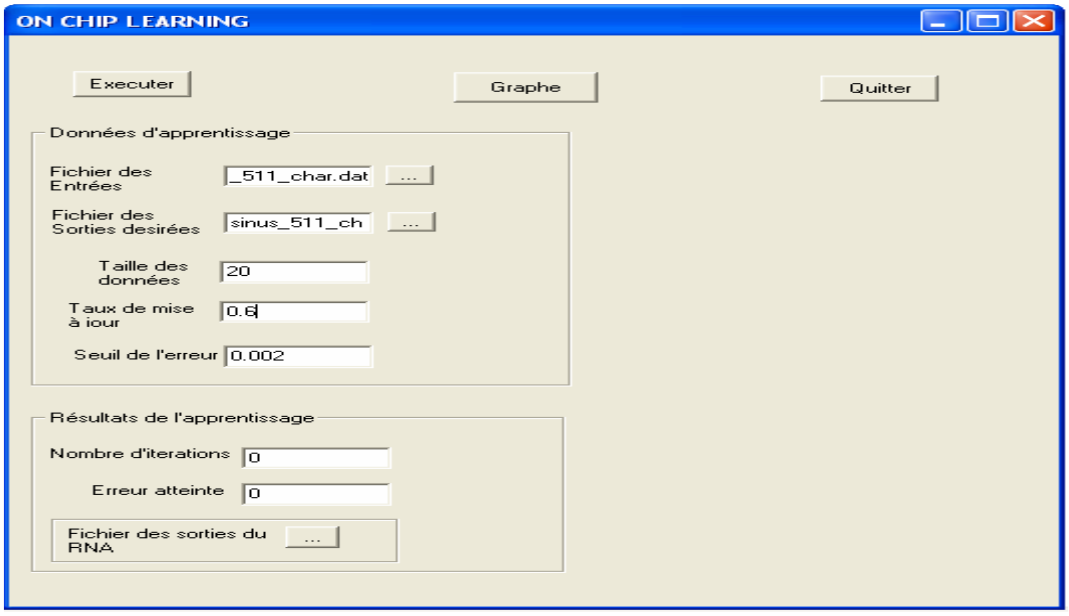


Figure 4.4 : *Le menu on chip*

Nous disposons aussi, de zones de texte où sont affichés, une fois l'apprentissage terminé, le nombre d'itérations atteint ainsi que EQM obtenue.

Ce menu contient aussi des boutons de commande qui permettent de :

- Spécifier les données de la base d'apprentissage (fichiers des entrées / sorties désirées).
- Exécuter le processus d'apprentissage.
- Afficher les graphes des fonctions estimées.

- Ouvrir le fichier des sorties estimées du RNA.
- Quitter le menu.

IV.3.2.1 Le bouton « Exécuter »

Le bouton de commande « Exécuter » permet de lancer la configuration de la carte RC200 et de commencer le processus d'apprentissage. A la fin de l'apprentissage, les poids résultants ainsi que les sorties estimées du RNA sont sauvegardés dans des fichiers. L'erreur quadratique moyenne obtenue ainsi que le nombre d'itérations atteint sont affichés dans des zones de texte. La figure 6.5 détaille le processus provoqué par le bouton de commande « Exécuter » .

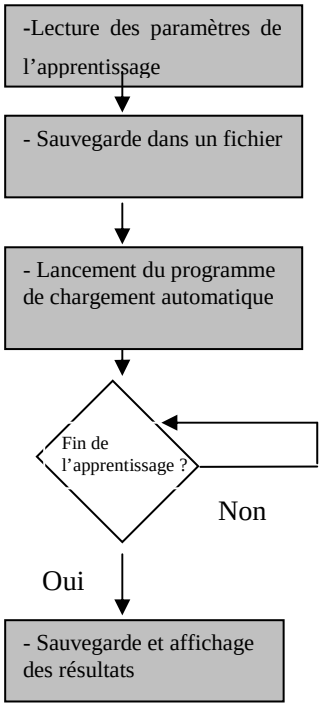


Figure 4.5 : *Processus déclenché par le bouton « exécuter »*

IV.3.2.2 Le bouton « fichier des sorties du RNA »

En appuyant sur le bouton de commande « fichier des sorties du RNA », nous ouvrons d'une manière automatique un fichier de données (.dat), qui contient les sorties correspondant aux entrées utilisées lors de l'apprentissage de la fonction estimée par le RNA (figure 4.6). Ces données représentées sous forme de réels, permettront un meilleur « débogage » des

paramètres du RNA, car elles sont plus lisibles que celles en virgule fixe. Ces données pourront aussi être exportées vers un autre logiciel (Matlab par exemple).

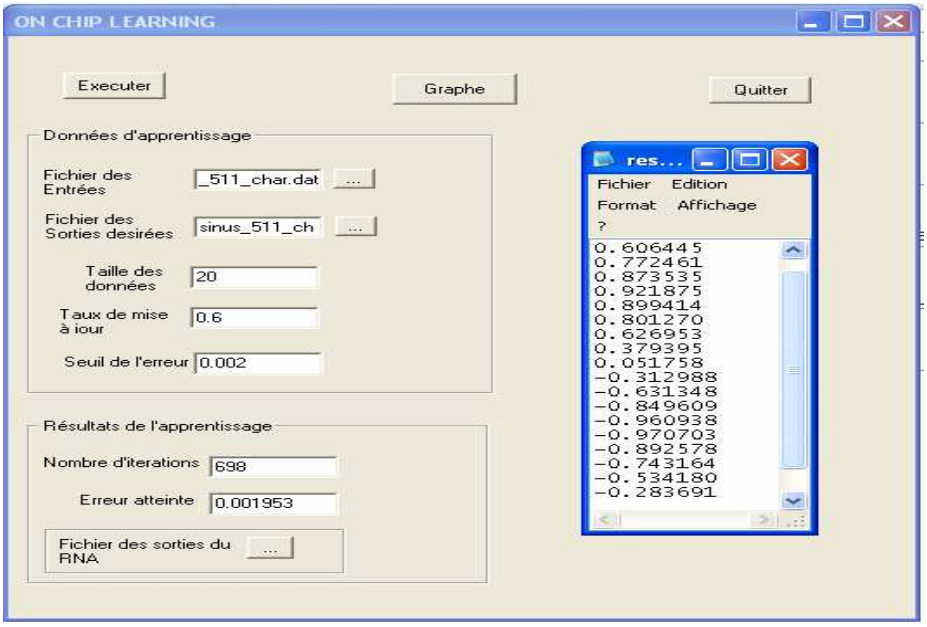


Figure 4.6 : *Sorties du RNA obtenues à la fin de l'apprentissage*

IV.3.2.3 Le bouton « graphe »

En appuyant sur le bouton de commande « graphe », nous affichons dans une zone du menu les graphes de la fonction désirée et de la fonction estimée (figure 4.7). Ceci constitue un excellent élément d'aide au « débogage » des paramètres du RNA.

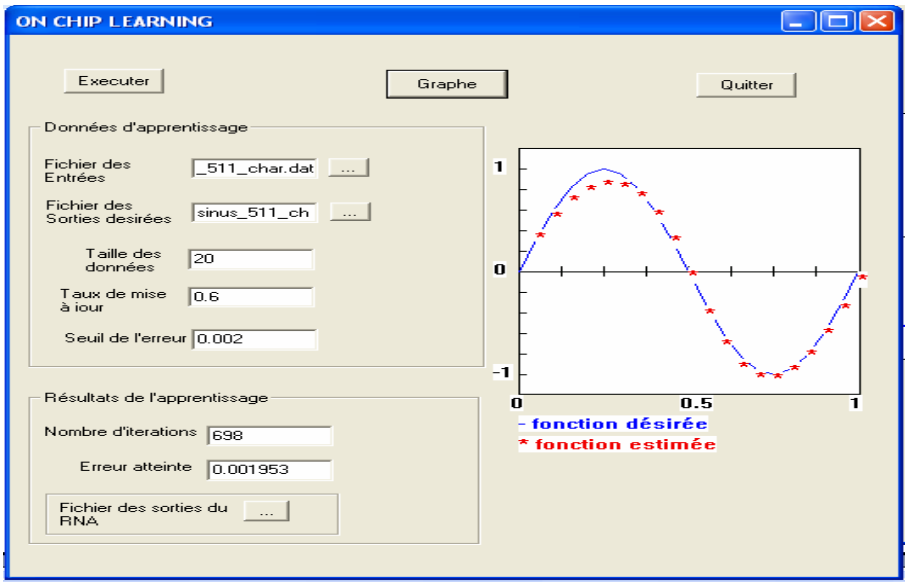


Figure 4.7 : *Visualisation de la sortie du RNA*

IV.3.3 Le menu de l'apprentissage hors ligne

La menu de l'apprentissage *off chip* (figure 4.8), contient des boutons de commande qui permettent de :

- Sélectionner le fichier des poids de l'architecture adoptée.
- Sélectionner le fichier des entrées.
- Exécuter le processus de configuration de la carte, propagation des entrées, transfert des résultats.
- Dessiner les graphes de la fonction estimée et de la fonction désirée.
- Visualiser le fichier des sorties du RNA.
- Quitter le menu.

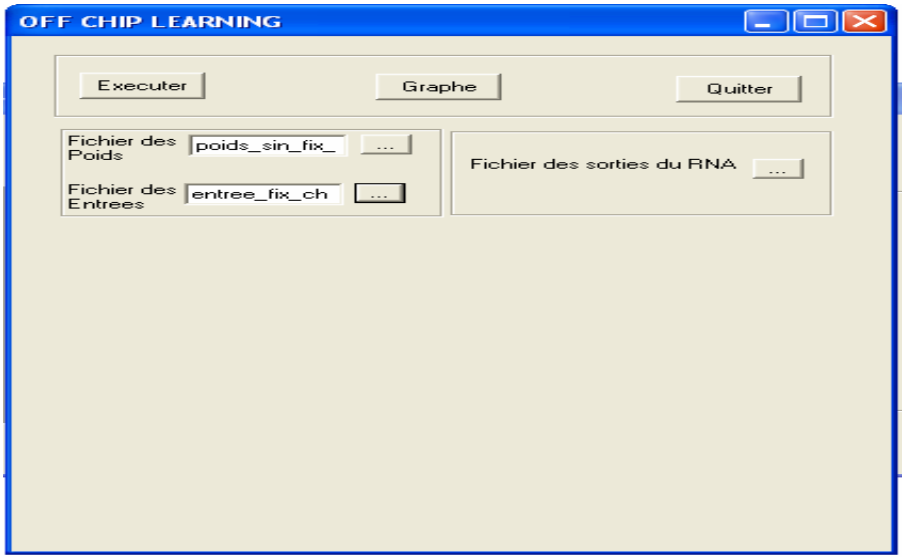


Figure 4.8 : *Le menu off chip*

IV.3.3.1 Le bouton « Exécuter »

Le bouton « exécuter » permet de configurer la carte RC200, d'effectuer la propagation des données d'entrée dans le RNA et de sauvegarder les résultats dans un fichier de données. La figure 4.9 explicite le processus déclenché par l'appui sur le bouton « exécuter » :

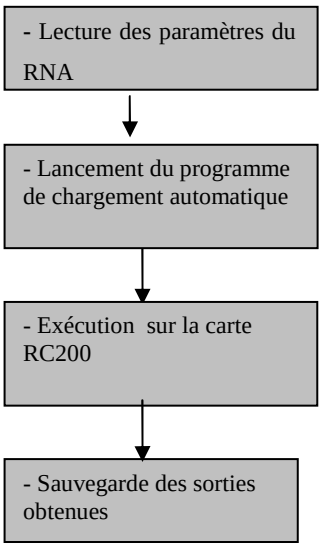


Figure 4.9 : *Processus déclenché par le bouton « exécuter »*

IV.3.3.2 Le bouton « fichier des résultats »

Ce bouton de commande permet d'afficher les sorties obtenues à l'issue de la propagation des entrées dans le RNA construit (figure 4.10).

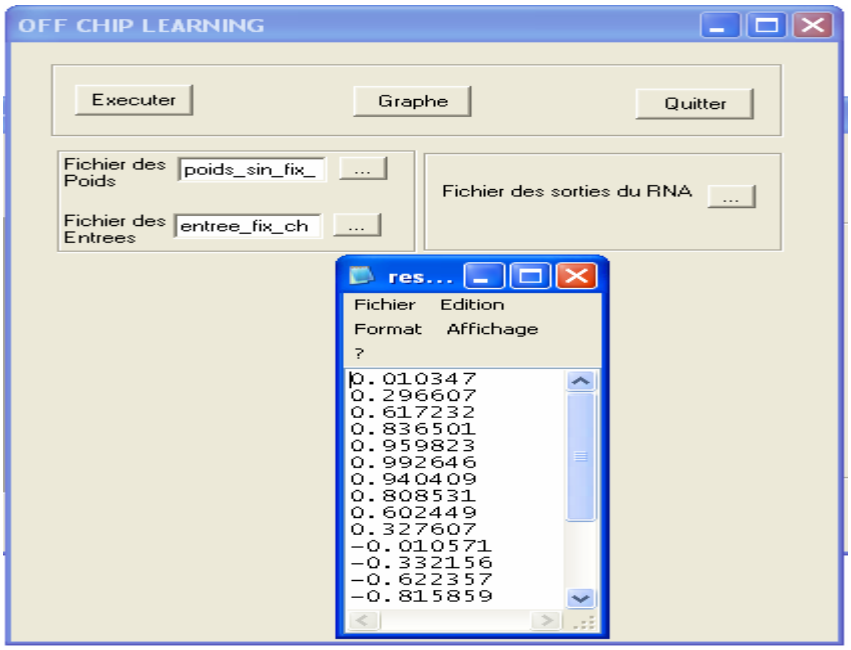


Figure 4.10 : *Sorties du RNA*

IV.3.3.3 Le bouton « Graphe »

Ce bouton de commande permet de dessiner le graphe de la fonction désirée et celui de la fonction estimée (figure 4.11).

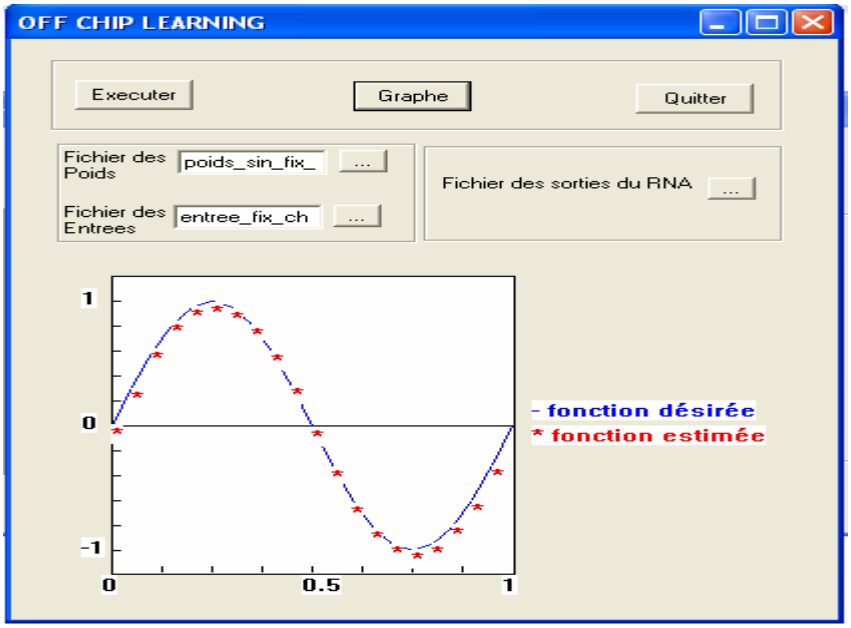


Figure 4.11 : *Visualisation de la sortie du RNA*

IV.4 Exemples d'application

Nous avons utilisé l'interface conçue pour quelques applications pratiques de réseaux de neurones artificiels. La première application est l'estimation de la fonction XOR logique qui constitue un exemple typique de classification des données. La seconde application est l'approximation de la réponse indicielle d'un système d'ordre un. La troisième application est la technique de l'amélioration d'un contrôleur linéaire (*feed-back error learning*) pour la commande en position d'un moteur à courant continu (chapitre III).

IV.4.1 Implémentation d'un RNA pour le problème du XOR logique

Nous avons construit un RNA [2,3,1]: deux neurones dans la couche d'entrée, une couche cachée avec trois neurones et une couche de sortie. La sélection du menu « *On Chip learning* » va permettre de sélectionner les paramètres de l'apprentissage suivants :

L'EQM cible = 0.001

Le taux d'apprentissage μ=0.3

Nous appuyons ensuite sur le bouton « exécuter » pour lancer l'apprentissage.

Une fois l'apprentissage terminé, nous appuyons sur le bouton « fichier des sorties du RNA» afin de visualiser les résultats (figure 4.12).

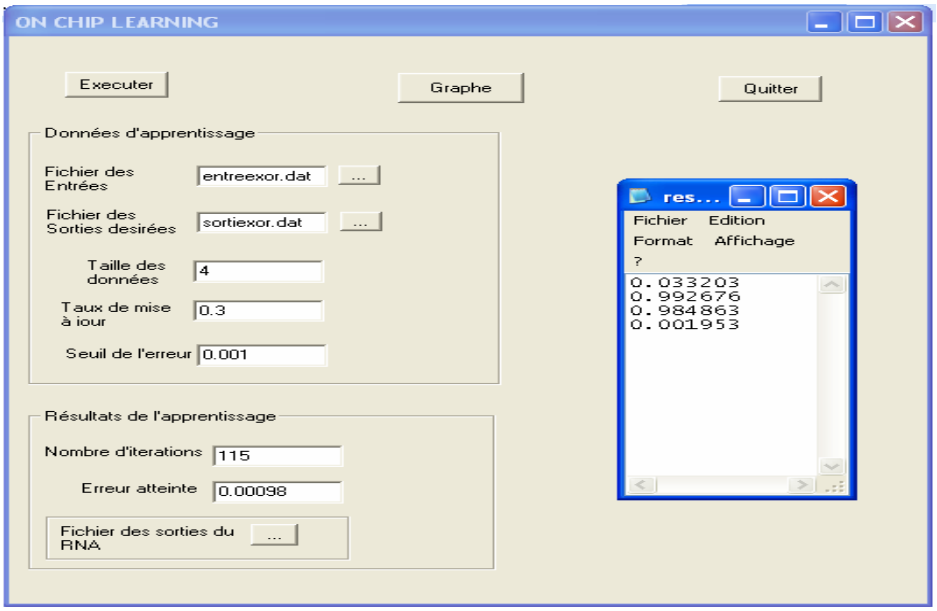


Figure 4.12 : *Résultats de l'approximation du XOR logique.*

Nous constatons que l'EQM désirée a été atteinte au bout de 115 itérations et que la fonction est estimée avec une assez bonne précision, si l'on considère la représentation des données réelles (16 bits en virgule fixe).

Nous pouvons en conclure, que l'application développée donne de bonnes solutions pour un problème de classification par RNA, dont le XOR logique constitue un *benchmark.*

IV.4.2 Implémentation d'un RNA pour l'approximation de la réponse indicielle d'un système du premier ordre

Comme nous l'avons vu, parmi les domaines d'application des RNAs, on retrouve celui l'approximation des fonctions à partir de données expérimentales. Afin d'illustrer l'utilisation de l'environnement graphique que nous avons réalisé, pour ce type d'applications, nous avons pris comme exemple l'approximation de la réponse indicielle d'un système du premier ordre.

Sa fonction de transfert est donnée par :

$$F(p) = \frac{1}{1+0.2p} \qquad (4.1)$$

La réponse indicielle d'un tel système est donnée par l'équation :

$$y(t) = 1 - e^{-5t} \qquad (4.2)$$

Nous avons utilisé l'application développée, pour l'approximation de la réponse du système précédent par un RNA. Nous avons élaboré une architecture [1,3,1]. Comme pour l'exemple précédent, la sélection du menu « *On Chip learning* » va permettre de sélectionner les paramètres de l'apprentissage.

Le taux d'apprentissage μ=0.3

L'EQM cible = 0.001

Nous appuyons ensuite sur le bouton « exécuter » pour lancer l'apprentissage et une fois le processus terminé, nous appuyons sur le bouton « graphe » afin de visualiser la fonction désirée et celle estimée (figure 4.13).

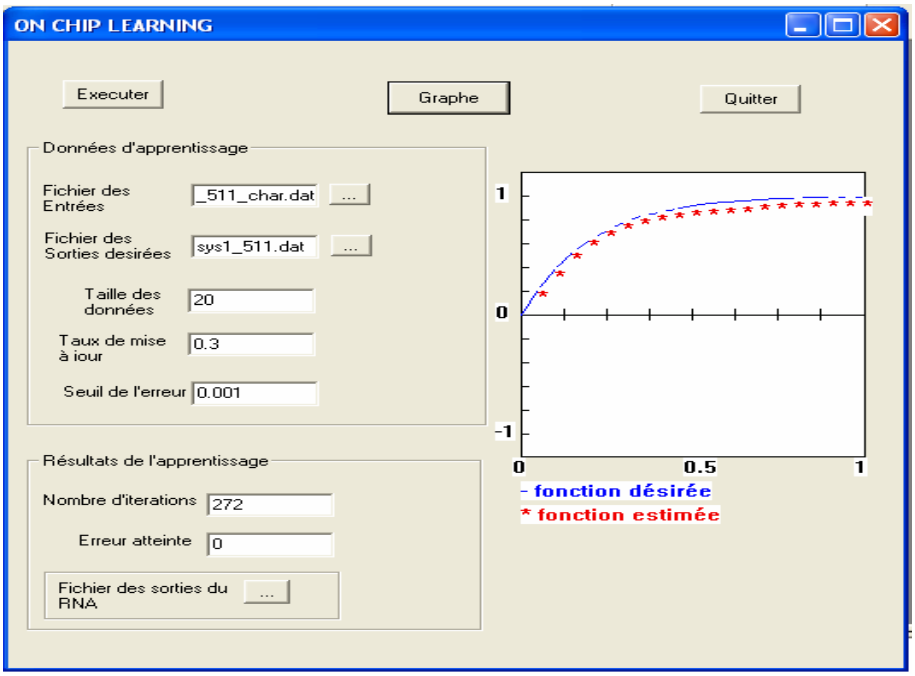


Figure 4.13 : *Approximation d'un système d'ordre un*

A partir des résultats obtenus, nous pouvons affirmer que l'architecture neuronale implémentée restitue une bonne approximation des fonctions et que l'interface graphique développée constitue un environnement pratique et convivial pour ce type d'applications.

IV.4.3 Implémentation de la commande en position d'un moteur à courant continu

Nous avons conçu une interface graphique, pour l'implémentation de la technique de l'amélioration d'un contrôleur linéaire (*feed-back error learning*) pour la commande en position d'un moteur à courant continu (chapitre III) (figure 4.14).

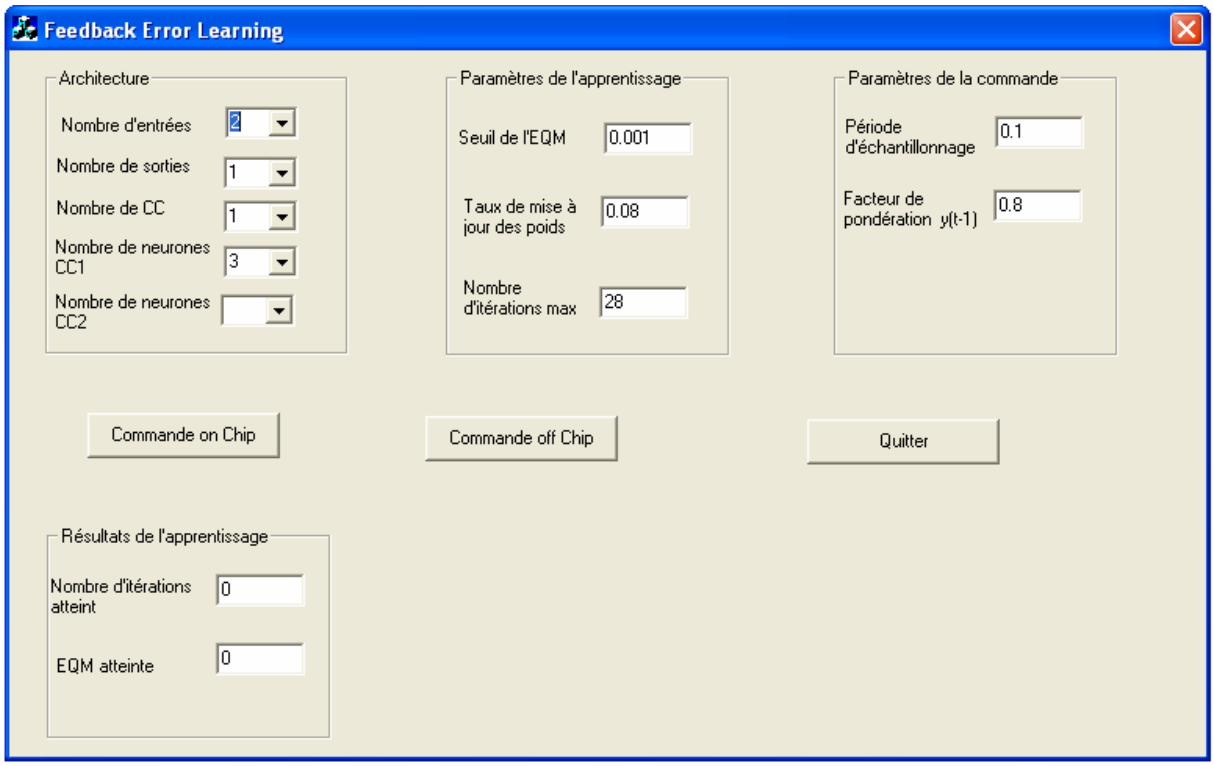


Figure 4.14 : *Interface graphique pour l'implémentation de la commande sur la carte RC200*

L'interface graphique comporte quatre groupes :

- Le groupe « Architecture » : contient des listes déroulantes, à partir desquelles nous pouvons spécifier l'architecture du réseau de neurones.
- Le groupe « Paramètres de l'apprentissage » : contient des zones d'édition qui permettent de spécifier l'EQM cible, le taux de mise à jour des poids et le nombre d'itérations maximales.
- Le groupe « Paramètres de la commande » : contient des zones d'édition qui permettent de spécifier la période d'échantillonnage ainsi que le facteur de pondération de la sortie à l'instant précédent.
- Le groupe « résultats de l'apprentissage » : contient des zones de texte dans lesquelles nous pouvons lire les résultats de l'apprentissage.

Dans cet interface graphique nous trouvons aussi, trois boutons de commande : Commande on chip, Commande off chip et Quitter.

- Le bouton « Commande on chip » **:** Permet de lancer le processus d'apprentissage en ligne de la commande.
- Le bouton « Commande off chip » : Permet d'implémenter le contrôleur neuronal avec des poids calculés lors de la précédente phase d'apprentissage.
- Le bouton « Quitter » : Permet de quitter l'application.

IV.5 Conclusion

Dans ce chapitre, nous avons présenté l'environnement graphique conçu. Cette application permet, à un utilisateur non initié dans le domaine de la programmation des circuits FPGA, de réaliser l'implémentation d'un RNA sur la carte de développement RC200 de *Celoxica*, ce qui aurait pour conséquence de faciliter et de réduire le temps de mise au point d'une architecture neuronale à quelques secondes seulement.

Cette interface graphique permet à partir de différents menus, de construire l'application désirée. Pour ce faire, l'utilisateur peut spécifier les différents paramètres du RNA (nombre de couches, nombre de neurones, etc...) ainsi que celles de l'apprentissage (taux de mise à jour des poids, nombre d'itérations maximales, etc…). L'interface graphique apporte aussi des outils pour contrôler l'exécution de l'apprentissage.

Nous l'avons utilisée pour une première application d'estimation du XOR logique, qui illustre l'utilisation des RNA dans le domaine de la classification. Une deuxième application à l'identification de la réponse indicielle d'un système du premier ordre, illustre l'utilisation des RNA dans le domaine d'approximation des fonctions. Enfin, une troisième application de commande en position d'un moteur à courant continu, déjà abordée en détail dans le chapitre précédent (chapitre III).

CONCLUSION GENERALE

LA ROCHEFOUCAULD / Maximes / Garnier 1967

« Tout le monde se plaint de sa mémoire, et personne ne se plaint de son jugement. »

< M 89 p.27 >

Les travaux développés dans cette thèse ont porté sur la réalisation d'une interface de haut niveau permettant la génération automatique de configurations optimales sur des circuits reconfigurables (FPGA) pour l'implémentation des réseaux de neurones. Deux approches ont été développées pour aborder, en premier lieu, le développement et la conception. La première repose sur l'utilisation du langage VHDL sous l'environnement ISE de *Xilinx,* et la deuxième est basée sur le Handel-C avec l'outil DK de *Celoxica*. Le premier choix est motivé par le souci de réaliser des applications optimisées et travaillant à des fréquences élevées, alors que le second est justifié par le désir de réduire au maximum le temps de développement et la nécessité de concevoir des configurations complètement paramétrables.

Pour atteindre l'objectif souhaité, il était indispensable d'aborder certaines difficultés liés au type de RNA et l'algorithme d'apprentissage à implémenter, à leurs mises en œuvre, au choix du circuit FPGA à cibler, à la précision des données à manipuler, à l'arithmétique à utiliser, à l'optimisation des ressources et de la fréquence de travail, à la simulation, à la conception de l'interface graphique et sa communication avec la carte à FPGA. Pour résoudre tous ces problèmes, des solutions ont été proposés et les résultats obtenus, suite aux différents tests effectués, attestent de la conformité de ces solutions.

Avec le langage VHDL, les résultats de synthèses ont montré une grande précision atteinte par la représentation en virgule flottante, qui devient une performance très intéressante surtout pour une implémentation « *On-chip learning* », mais nécessitant une quantité de ressources assez importante. Pour résoudre ce problème, la représentation en virgule fixe peut réaliser un bon compromis précision/ressources. C'est une représentation qui permet de concevoir des architectures intégrant la phase d'apprentissage et de généralisation sur le même circuit. Concernant l'implémentation de la fonction d'activation, nous avons constaté que l'approximation linéaire est, à la fois, la moins onéreuse en ressources et la plus rapide. Alors que celle basée sur le polynôme approximatif, peut garantir une meilleure précision.

Avec le Handel-C , nous avons conçu une architecture à deux couches cachées et nous avons opté pour une représentation des données en virgule fixe. Nous avons utilisé l'algorithme de Remez pour l'approximation de la fonction d'activation sigmoïde. La stratégie adoptée a consisté dans un premier temps, à programmer l'algorithme de la rétropropagation en langage C sur calculateur. L'étape suivante a été la réalisation du « portage » du programme conçu en langage C, vers le langage Handel-C. La dernière étape enfin, a été l'implémentation de l'architecture conçue sur la carte RC200. Pour une

implémentation avec un apprentissage en ligne, nous avons adopté une représentation des données en virgule fixe sur 16 bits. Pour une implémentation avec un apprentissage hors ligne, nous avons adopté une représentation en virgule fixe sur 32 bits.

Pour prouver la fonctionnalité de chaque approche développée, nous avons élaboré des applications pratiques qui consistaient, en premier lieu, à commander en vitesse un moteur à courant continu à travers un PID approximé par un RNA, et en second lieu, à améliorer un contrôleur linéaire dans le cas de la commande en position d'un MCC. Les résultats obtenus ont été satisfaisants.

En effet, dans la première validation, les résultats obtenus montrent la capacité de régulation en vitesse du système de commande pour différents points de fonctionnement. La souplesse de programmation et la reconfiguration totale du circuit FPGA (carte RC200) offre des possibilités d'amélioration, éventuelle, de l'architecture du réseau de neurone implémenté. Le système de commande peut être utilisé pour implémenter d'autres types de contrôleur.

Dans la deuxième validation, l'architecture neuronale développée pour la commande, inclut la phase d'apprentissage en ligne. Nous avons constaté que l'erreur reste bornée durant l'apprentissage, moyennant un choix précis de l'architecture neuronale et des paramètres de l'apprentissage. Nous avons aussi pu mettre en pratique les capacités de généraliser et d'abréger des RNAs, qui sont intéressantes pour les systèmes de commande.

Enfin, nous avons réalisés une plateforme logicielle permettant le téléchargement automatique du fichier de configuration (*bitfile*) sur la carte à FPGA, la transmission des données et des paramètres de l'apprentissage, ainsi que la lecture des résultats du traitement. Avec *Microsoft visual C++*, nous avons conçus une interface graphique à menus dédiée à l'utilisation rapide et conviviale de l'architecture développée. L'avantage qu'elle offre réside dans la facilité de l'approche : pas de langage informatique, elle nécessite seulement l'utilisation de menus graphiques (boutons de commande, zones d'édition...). A partir de ces menus graphiques, l'utilisateur est invité à construire son application. Pour ce faire, il peut spécifier les différents paramètres du RNA (nombre de couches, nombre de neurones, topologie du réseau, etc.). Ce système apporte de nombreux outils pour construire le réseau de neurones et aussi pour contrôler l'exécution de l'apprentissage. Dans le but de la tester, cette plateforme logicielle a été utilisée dans trois différentes applications pratiques : l'estimation du XOR logique qui illustre l'utilisation des RNAs dans le domaine de la classification, l'identification de la réponse indicielle d'un système du premier ordre illustrant l'utilisation

des RNAs dans le domaine de l'approximation des fonctions, et la commande en position d'un moteur à courant continu.

Comme perspectives des travaux réalisés, il serait intéressant d'optimiser, encore plus, la fréquence de travail et la consommation des ressources, d'utiliser d'autres circuits FPGA plus performants afin de pouvoir profiter d'une représentation de données de plus grande taille en virgule fixe ou en virgule flottante, d'étudier l'implémentation d'autres types de RNA et d'autres algorithmes d'apprentissage, d'améliorer l'interface graphique par l'ajout d'autres fonctionnalités (menu pour d'autres types de RNA, menu pour d'autres cartes de développement, apprentissage par topologie, etc…), et enfin, utiliser cette interface pour l'apprentissage par la topologie des RNAs et la recherche de l'architecture adéquate pour une application quelconque.

BIBLIOGRAPHIE

[Abra98] D.Abramson, K.Smith, P.Logothetis, and D. Duke, " FPGA based implementation of a Hopeld neural network for solving constraint satisfaction problem ",In Proceedings of 24th euromicro workshop on computational intelligence, pp. 688693, Sweden, August 1998.

[Aumiau] M. Aumiaux. Logique binaire, " Fonctions Logiques et arithmétique binaire ", MASSON.

[Amin97] H. Amin & al., **"** Piecewise linear approximation applied to non linear function of neural network ", IEE Proc- Circuits Devices Syst., vol 144(6), pp 313-317, 1997.

[Bade94] S. Bade and B. L. Hutchings, " FPGA based stochastic neural network implementation ", Proc. IEEE workshop on fpgas for custom computing machines, 1994, pp.189-198.

[Benb00] **C. Benbouchama**, C.Larbes and N.Louam, "A neuro-linguistic controller for vehicle active suspension systems ", Ninth International Conference On Aerospace Sciences and Aviation Technology, ASAT-9, Cairo, Egypt.

[Benb04] **C. Benbouchama,** " Deux contrôleurs neuro-linguistiques pour la commande de la suspension active d'un véhicule roulant ", C.G.E.03- E.M.P. Bordj El Bahri du 15 au 16 Février 2004.

[Benb14] **C.Benbouchama**, C.Larbes, " A Neuro-Linguistic Controller for Vehicle Active Suspension Systems ", Proc. of the 7th International Symposium on Advanced Vehicle Control, AVEC'04, Arnhem, the Netherlands, 23-27 August, 2004, pp.567-572, 2004.

[Benb05] **C. Benbouchama,** M. Tadjine et A. Bouridane, " Interface pour la génération automatique de configurations sur FPGAs pour l'implémentation des réseaux de neurones ", revue EL MIR'AT de l'EMP, N°2.

[Benb06] **C. Benbouchama**, S. Sakhi, M. Tadjine et A. Bouridane, " On the Implementation of Neural Networks on FPGA Circuits: Application to Real Time Speed Control of DC Motor ", International Review of Electrical Engineering (I.R.E.E.), Vol.1, n. 4, pp.525-534, 2006.

[Benb07] **C. Benbouchama**, T. Souanef, M. Tadjine et A. Bouridane, " Design of an FPGA based Neural Controller ", 4th international conference on informatics in control, Automation and Robotics (ICINCO'07), Angers, France, 9-12 Mai 2007.

[Benb17] **C. Benbouchama**, S. Sakhi, M. Tadjine et A. Bouridane, " The FPGA Neural Networks Implementation for a Real Time Control ", Archives of Control Sciences (A.C.S.), Vol. 17(LIII), n. 1, pp.5-27, 2007.

[Benb27] **C. Benbouchama**, T. Souanef, M. Tadjine et A. Bouridane, " Experimental Evaluation of FPGA Neural Networks Implementation for position control of DC Motor ", accepted in WSEAS Transactions on Systems and Control, 2007.

[Benb37] **C. Benbouchama**, " Plateforme logicielle pour l'aide à la configuration des circuits FPGA ", revue EL MIR'AT de l'EMP, N°7.

[Beuc98] J. Beuchat, J. Haenni, and E.Sanchez, " Hardware Reconfigurable Neural Networks, In Parallel and Distributed Processing ", IPPS/SPDP, pp. 91 98, Springer-Verlag, 1998.

[Beuc02] Jean-Luc Beuchat et Arnaud Tisserand, " Opérateur en-ligne sur FPGA pour l'implémentation de quelques fonctions élémentaire ", RENPAR'14 /ASF/ SYMBA Tunisie 2002.

[Boni00] Y. Boniface, " Etude et développement d'une bibliothèque d'adaptation du parallélisme neuromimétique au parallélisme MIMD " , thèse de doctorat de l'université de Nancy1, octobre2000.

[Cant98] M.A Cantin, " Implantations du réseau de neurones Fuzzy Art ", Université du

Québec, Montréal 1998.
[Celo04] Celoxica Inc, “ Handel-C language reference manual for DK 3.0 ”, 2004.
[Celo06] Celoxica Inc, “ http://www.celoxica.com ”.
[Coe04] S.Coe, “ A Memetic Algorithm Implementation on an FPGA for VLSI Circuit Partitioning ”, Masters thesis, School of Engineering, University of Guelph, August 2004.
[Cox92] Cox, C.E. and E. Blanz, GangLion, “ a fast field -programmable gate array implementation of a connectionist classifier ”, IEEE Journal of Solid-State Circuits, 1992. 28(3): p. 288-299.
[Derr02] S. Derrien, “ Etude quantitative de partitionnement de réseaux de processeurs pour l’implantation sur circuits FPGA ”, thèse de doctorat de l’université de Rennes, 2002.
[Dute02] J.M Dutertre, “ Circuits reconfigurables robustes ”, thèse de doctorat de l’université de Montpellier II, 2002.
[Earl04] Dennis Duncan Earl, “ Developpement of an FPGA-based hardware evaluation system for use with GA-designed Artificial neural ”, A Dissertation Presented for the Doctor of Philosophy Degree ,The University of Tennessee, Knoxville May 2004.
[Eldr94] J. G. Eldredge and B. L. Hutchings, “ RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs ”, In IEEE World Conference on Computational Intelligence, Orlando, FL, 1994.
[Ferr94] A.T.Ferrucci, “ A Field-Programmable Gate Array Implementation of Self-Adapting and Scalable Connectionist Network ”, Mars 1994.
[Fies94] E.Fiesler, “ Comparative Bibliography of Ontogenic Neural Networks ”, Proceedings of the International Conference on Artificial Neural Networks (ICANN94), 1994.
[Figu98] M. Figueiredo and C. Gloster, “ Implementation of a probabilistic neural network for multispectral image classification on an FPGA based custom computing machine ”, In 5th Brazilian Symposium on Neural Networks, Brazil, December 1998.
[Gari01] De Garis, H., & al., “ Initial evolvability experiments on the CAM-brain machines (CBMs) ”, In Proceedings of the 2001 Congress on Evolutionary Computation, 2001, pp 635-641, vol. 1.
[Gaut99] E.Gauthier, “ Utilisation des réseaux de neurones artificiels pour la commande d’un véhicule autonome ”, thèse de doctorat de l’INP Grenoble 1999.
[Groi01] T.Groisil, J.Coyne, O.Oliny, “ Etude d’un algorithme de détection d’objets Méthodologie de parallélisation sur FPGA ”, ENSP Stratbourg 2001.
[Gucc93] Steven A. Guccione and Mario J. Gonzalez, “ A Neural Network Implementation Using Reconfigurable Architectures ”, Abingdon EE&CS Books, Abingdon, England, 1993, 443-451.
[Gucc95] Guccione, S.A. and M. Gonzalez, “ Classification and performance of reconfigurable architectures ”, in Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications. 1995, pp 439-448, Springer -Verlag, Berlin.
[Guex98] C. Guex, “ Introduction au VHDL ”, Ecole d’ingénieurs du Canton de Vaud 1998.
[Guyot] Alain GUYOT, “ Virgule flottante ”, Techniques de l'Informatique et de la Microélectronique pour l'Architecture. Unité associée au C.N.R.S.
[Holt91] J. Holt and T. Baker, “ Backpropagation simulations using limited precision calculations ”, In International joint conference on Neural Networks, pp. 121126, Seattle, WA, July 1991.
[Izeb99] N. Izeboudjen, “ Conception et implémentation en FPGA d’un classifieur neuronal

des arythmies cardiaques ", 1999.

[Jame00] James-Roxby, P. and B.A. Blodget, " Adapting constant multipliers in a neural network implementation ", in Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines, 2000. pp 335-336.

[Jord92] M. I. Jordan and D. E. Rumelhart, " Forward models: Supervised learning with a distal teacher ", Cognitive Science, 16:307-354, 1992.

[Kawa90] M. Kawato, " Schemes and models for control of arm trajectory ", In W. T. Miller, R.S. Sutton, and P. J. Werbos, editors, Neural Networks for Control, pages 197-228. MIT Press, 1990.

[Koth04] Sampath Kumar Kothandaraman, " Implementation of block-based neural networks on reconfigurable computing platforms ", A Thesis Presented for the Master of Science Degree of The University of Tennessee, Knoxville, August 2004

[Kuro94] Y. Kuroe, Y. Nakai, and T. Mori, " A new neural network learning of inverse kinematics of robot manipulator ", In Proc. of the Int. Conf. on Neural Networks, Orlando (FL), 1994.

[Lind95] Lindey C.S & Lindblad T, " Survey of Neural Networks Hardware ", Proc. of Applications and Science of Artificial Neural Networks Conference, pp. 1194-1205. Orlando, Fla., USA, 1995.

[Ligo98] Ligon W.B & al., " A re-evaluation of the practicality of floating point operations on FPGAs ", In Kenneth L. Pocek & Jeffrey Arnold , editors, IEEE Symposium on FPGAs for Custom Computing Machines, pages 206-215, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[Loo02] S.M Loo, B.Earl Wells, N Freije, and J.Kulick, " Handel-C for Rapid Prototyping of VLSI Coprocessors for Real Time Systems ", In South-eastern Symposium on System Theory, 2002.

[Lysa94] P. Lysaght, J. Stockwood, J. Law, and D. Girma, " Artificial Neural Network Implementation on a Fine-Grained F P G A ", In R. Hartenstein and M. Z. Servit, editors, Field-Programmable Logic: Architectures, Synthesis and Applications. 4th International Workshop on Field-Programmable Logic and Applications, pp. 421-431, Springer-Verlag, Prague, Czech Republic, 1994.

[Marc93] Marchesi, M., et al.. Fast neural networks without multipliers. IEEE Transactions on Neural Networks, 1993. 4(1): p. 53-62.

[Mart02] P. Martin, " A Pipelined Hardware Implementation of Genetic Programming Using FPGAs and Handel-C ", In EuroGP '02: Proceedings of the 5th European Conference on Genetic Programming, pp. 1{12, Springer-Verlag, London, UK, 2002.

[Mart94] M. Martin, " A reconfigurable hardware accelerator for back-propagation connectionist classifiers ", Masters thesis, University of California, Santa Cruz, 1994.

[Mcke01] M. McKenna and B. Wilamowski, " Implementing a fuzzy system on a field programmable gate array ", International Joint Conference on Neural Networks (IJCNN'01), Washington DC, July 2001, pp. 189-194.

[Mill94] P. M. Mills, A. Y. Zomaya, and M. O. Tadife, " Adaptive model-based control using neural networks ", International Journal of Control, 60(6):1163-1192, 1994.

[Nich03] Nichols, K., M. Moussa, and S. Areibi, " Feasibility of Floating-Point Arithmetic in FPGA based Artificial Neural Networks ", in Proceedings of the 15th International Conference on Computer Applications in Industry, 2003

[Nich04] K. Nichols, " A Reconfigurable Computing architectures for Implementing Artificial Neural Networks on FPGA ", Masters thesis, School of Engineering, University of

Guelph, December 2003.
[Nord92] T. Nordstrom, " Designing parallel computers for self organizing maps ", In Proceedings of the 4th Swedish Workshop on Computer System Architecture (DSA-92), Linkoping, Sweden, January 13-15 1992.
[Nord95] T.Nordstrom, " Highly Parallel Computers for Artificial Neural Networks ", Ph.d. thesis (1995:162 f), Division of Computer Science and Engineering, Lulea University of Technology, Sweden, March 1995.
[Nord98] T. Nordstrom and B. Svensson, " Using and designing massively parallel computers for artificial neural networks ", Journal of Parallel and Distributed Computing, Vol. 14,1992.
[Outa04] S.Outamzabet & M.Tafer, " Contribution à l'implémentation d'u réseau de neurones sur circuit FPGA ", Mémoire de projet de fin d'études, EMP 2004.
[Osso96] H. Ossoinig, E. Reisinger, C. Steger, and R. Weiss, " Design and FPGA implmentation of a neural network ", In Proc. 7th Int. Conf. on Signal Processing Applications and Technology, pp. 939943, Orlando, Florida,1996.
[Pand05] V.Pandya, " A Handel-C implementation of the backpropagation algorithm on field programmable gate arrays ", Master thesis. Faculty of Graduate Studies of the University of Guelph, Canada, December 2005.
[Pari04] Marc Parizeau, " Les Réseaux de Neurones ", Université LAVAL, 2004.
[Poor02] M. Poormann, U. Witkowski, H. Kalte, and U. Ruckert, " Implementation of ANN on a reconfigurable hardware accelerator ", In Euromicro workshop on parallel, distributed and network based processing, pp. 243250, Spain, January 2002.
[Pome93] D. A. Pomerleau, " Neural Network Perception for Mobile Robot Guidance ", Kluwer Academic Press, 1993.
[PU96] Perez-Uribe A. and E. Sanchez, " FPGA Implementation of an Adaptable-Size Neural Network ", in Proceedings of the Sixth International Conference on Artificial Neural Networks. 1996. pp 382-388, Springer-Verlog.
[Sakh05] S.Sakhi, " Implémentation de réseaux de neurones sur carte à FPGA RC200 ", Thèse de magistère, EMP 2005.
[Sakh15] S.Sakhi, **C. Benbouchama**, " Contribution à l'implémentation des réseaux de neurones sur circuits FPGA ", CGE'04, EMP, 12 - 13 Avril 2005.
[Sakh06] S. Sakhi, **C. Benbouchama**, M. Tadjine, " Implémentation des réseaux de neurones sur carte à FPGA RC200 : application à la commande temps réel d'un moteur à courant continu ", conférence internationale sur l'ingénierie de l'électronique, 28-29 mai 2006, U.S.T.Oran.
[Skrb99] M. Skrbek, " Fast neural network implementation ", Neural Network World, Vol. 9(No.5), 375-391, 1999.
[Tave95] Mikael Taveniku and Arne Linde, " A reconfigurable simd computer for artificial neural networks ", Licentiate Thesis No. 189L, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 1995.
[Touz92] Claude TOUZET, " Les Réseaux de Neurones Artificiels ", Juillet 1992.
[Werb92] P. J. Werbos. Approximate dynamic programming for real-time control and neural modelling ", In D. A. White and D. A. Sofge, editors, Handbook of Intelligent Control, pages 493-526. Van Nostrand Rheinold, 1992.
[Whit92] D. A. White and D. A. Sofge, " Applied learning optimal control for manufacturing ", In D. A. White and D. A. Sofge, editors, Handbook of Intelligent Control, pages 259-282. Van Nostrand Rheinold, 1992.
[Widr64] B. Widrow and F. W. Smith, " Pattern-recognizing control systems ", In Proceeding of Computer and Information Sciences, 1964.
[Xili06] " http://www.xilinx.com "

[Zerr06] M. Zerrouki, F. Kobzili, “ Environnement d’aide à l’implémentation des controleurs classiques sur une carte à FPGA ”, Mémoire de fin d’études ingénieurs. EMP 2006.

[Zhu99] Zhu, J.M., G.J., Gunther B.K. , “ Towards an FPGA based reconfigurable computing environment for neural network implementations ”, in Proceedings of Ninth International Conference on Artificial Neural Networks, 1999. pp. 661-666, vol.2.

[Zhu03] Jihan Zhu and Peter Sutton, “ FPGA Implementations of Neural Networks - a Survey of a Decade of Progress ”, Proceedings of 13th International Conference on Field Programmable Logic and Applications (FPL 2003), Lisbon, Sept.2003.

ANNEXES

ANNEXE 1

Système de commande du Moteur à courant continu

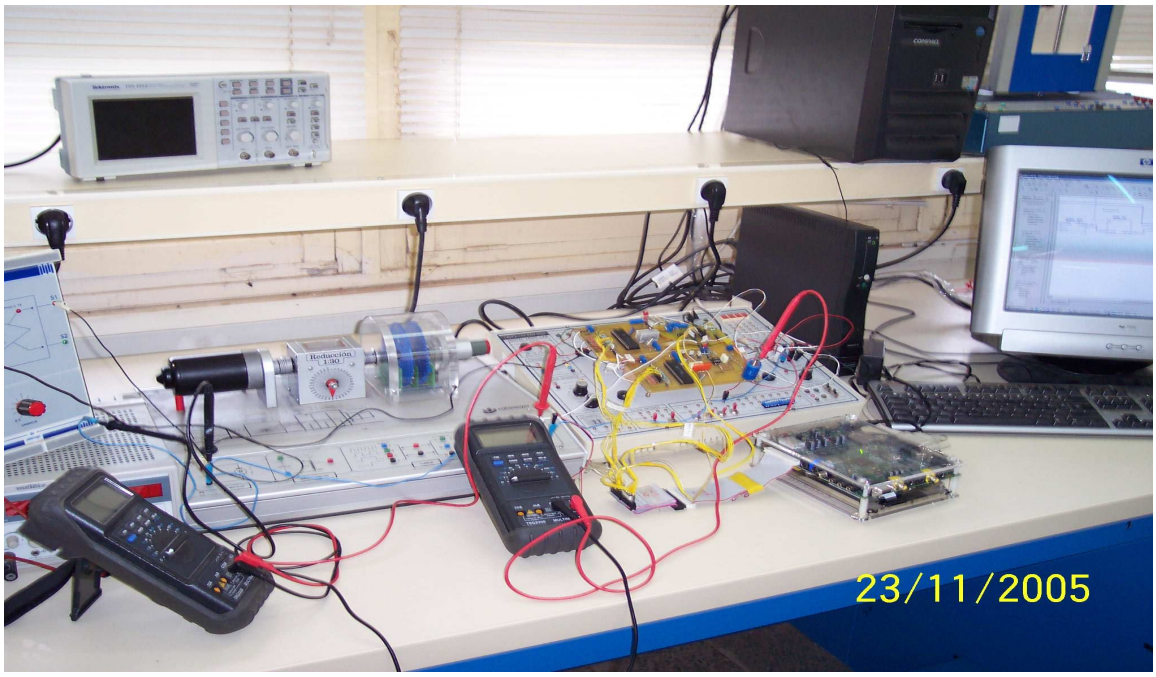


Figure 1: *Système de commande du Moteur à courant continu.*

La carte de développement RC200

La carte RC200 est une plate-forme d'évaluation et de développement à base d'un circuit FPGA. Elle inclut le circuit Virtex-II de type XC2V1000 FG456-4, une mémoire externe, des horloges programmables, un Ethernet, un support Audio, un support Vidéo, une Smart Media, un port parallèle, un port série RS-232 et un port PS/2 pour le clavier et la souris. Le logiciel qui l'accompagne comprend plusieurs plates formes telles le PAL (Platform Abstraction Layer), le DSM (Data Stream Manager), le RC200 PSL (Platform Support Library), et le FTU2 (File Tansfert Utility).

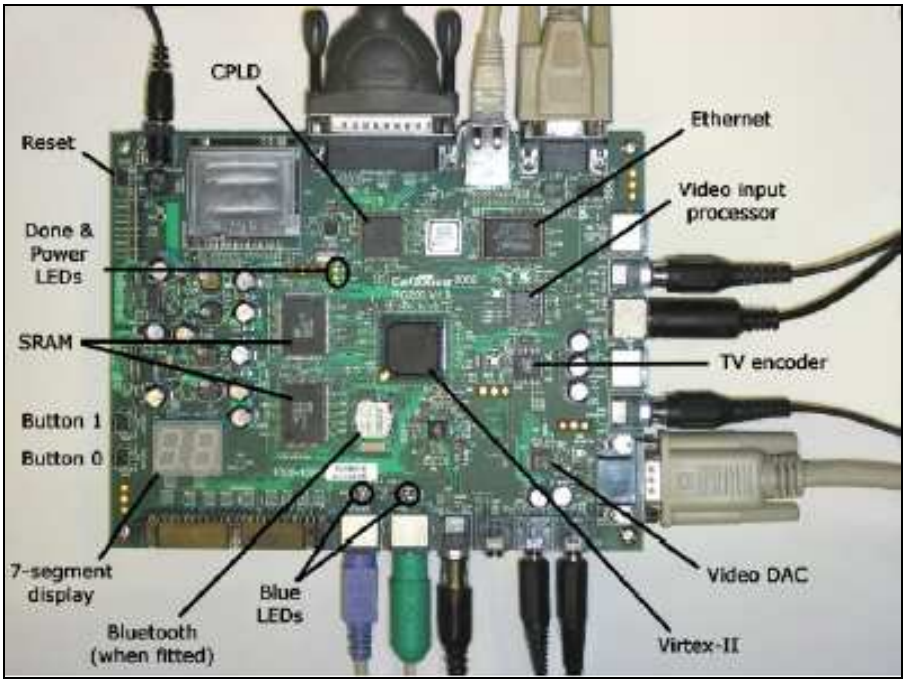


Figure 2: *Les composants de la carte RC200.*

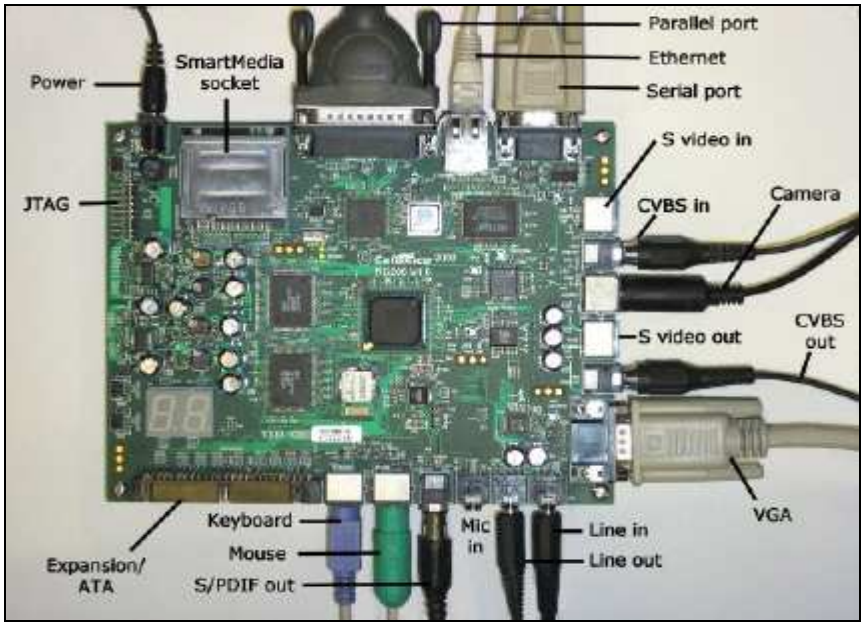


Figure 3: *Les connecteurs de la carte RC200.*

1- L'équipement standard

- Le circuit FPGA VIRTEX-II X2V1000-4 FPGA.
- L'Ethernet MAC/PHY.
- Les deux blocs de ZBT SRAM qui fournit un total de 4-MB.
- Le support Vidéo.
- Le support Audio.

- Le port parallèle qui permet la configuration du FPGA via le CPLD.
- Le RS-232.
- Les connecteurs du clavier et de la souris PS/2.
- Les deux afficheurs sept segments.
- Les deux LEDs bleus.
- Les deux commutateurs de contacts.

Equipement de la plate forme Celoxica renferme:

- La Platform Support Library (PSL).
- La Platform Abstraction Layer (PAL).
- La Data Stream Manager (DSM).
- Le FTU2 BIT utilisé pour le transfert des fichiers.

2- Description du matériel

Cette section décrit les composants dont dispose la RC200, ainsi que les moyens mis en œuvres pour programmer le FPGA via le transfert des données entre l'host et l' FPGA.

2.1 Le circuit FPGA

La carte RC200 comporte un FPGA Xilinx Virtex-II (XC2V1000-4FG456C). Ce dernier détient des rapports directs avec les composants cités précédemment dans la description de l'équipement standard.

2.2 Le port parallèle

Le RC200 est munit d'un port parallèle compatible que nous pouvons utiliser pour:

- Programmer le FPGA.
- Lire et écrire des données dans le FPGA.

2.3 ZBT SRAM

La RC200 est composé de deux ZBT RAMs, qui sont capables de fonctionner jusqu'à une fréquence de 100 MHz. Les RC200 Standard et Professionnel ont deux blocs de 2 MB, la RC200 Expert possède deux blocs de 4 MB. Les RAMs sont des composants à 512K ou 1024K mots de 36 bits.

2.4 L'Ethernet

La plate forme RC200 contient le composant Ethernet Standard Microsystems Corporation. Elle supporte un accès au FPGA de 8 bit et 16 bit. L'Ethernet comprend une entrée horloge à 25 MHz.

2.5 Les ports souris et clavier PS/2

La carte RC200 comprend deux ports PS/2, un pour la souris et l'autre pour le clavier.

2.6 Les afficheurs sept segments

Il existe deux afficheurs sept segments sur la RC200. Les segments sont numérotés comme suit:

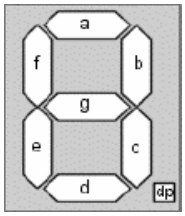


Figure 4: *Afficheur 7 segments.*

2.7 Les LEDs

La carte RC200 met à notre disposition deux LEDs bleus qui peuvent être contrôlées à partir du FPGA. Elle comporte aussi deux autres LEDs qui s'illuminent lorsque l'alimentation de la carte est en marche (LED D2) et quand le FPGA est programmé (LED D1). Celles-ci sont localisées à gauche de la marque Celoxica Copyright sur la carte.

2.8 Les commutateurs de contact

Il existe deux boutons dans le coin gauche inférieur de la carte (Bouton 0 et Bouton 1). Quand ils sont pressés, ceux-ci agissent comme des entrées à l'état haut.

2.9 Le bouton Reset

Le bouton reset sur la RC200 est à côté de l'entrée de l'alimentation. Il efface les programmes du FPGA, et le réinitialise.

3- Caractéristiques du composant Virtex-II XC2V100fg456

Le composant XC2V100fg456 comporte :

- 40 x 32 CLB.
- 40 Multiplieurs de 18x18s.
- 40 Mémoires de 18 Koctet.
- 12 DCM (Digital Clock Manager).
- 324 I/O.

ANNEXE 2

Le Convertisseur A/N : ICL 7109

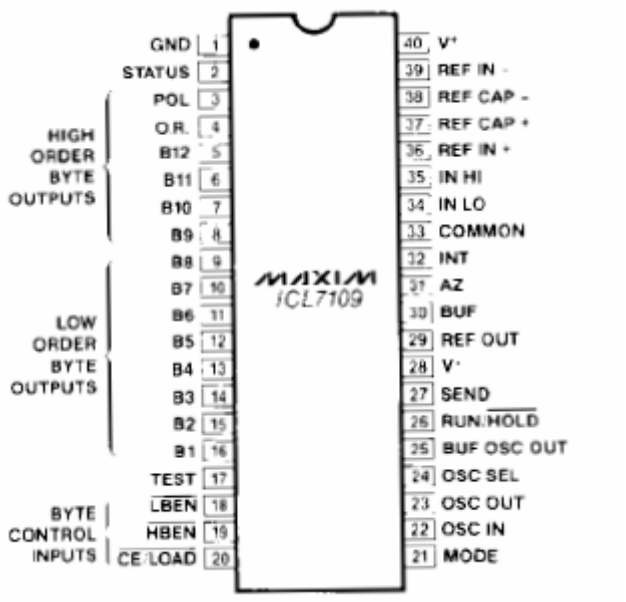


Figure 5: *Le brochage du circuit ICL 7109.*

PIN	Function	Description
1	GND	Ground
2	Status	Converter integrate/deintegrate phase
3	POL	Polarity =HI : Positive input
4	OR	Overrange
5…16	Bit11…bit1	Data input
17	Test	Test = HI : normal operation Test = LO : all output bits high
18, 19	$\overline{LBEN}$, $\overline{HBEN}$	Low Byte Enable, High Byte Enable
20	$\overline{CE}$ / $\overline{LOAD}$	Chip Enable/Load
21	MODE	Convertion Mode
22,23	OSC IN ,OSC OUT	Oscillator input / Oscillator Output
24	OSC SEL	Oscillator Select
25	BUF OSC OUT	Buffered Oscillator Output
26	RUN / $\overline{HOLD}$	HI : Run , LO : Hold
27	SEND	Indicate ability of external device to accept Data
28	V^-	Negative supply
29	REF OUT	Reference voltage Output
30	BUFFER	Buffer Amplifier Output
31,32	AUTO-ZERO, INTEGRATOR	Connected to Capacity
33	COMMON	Analog Common
34,35	INPUT LO, INPUT HI	Low side , High side of differential input
36,39	REF IN^+ , REF IN^-	Positive input, Negative input of differential reference
37,38	REF CAP^+ , REF CAP^-	Positive side, Negative side of reference capacitor
40	V^+	Positive supply (+5volt)

Table 1: *Description des broches du convertisseur « ICL 7109 ».*

ANNEXE 3

Le Convertisseur N/A : AD 7569

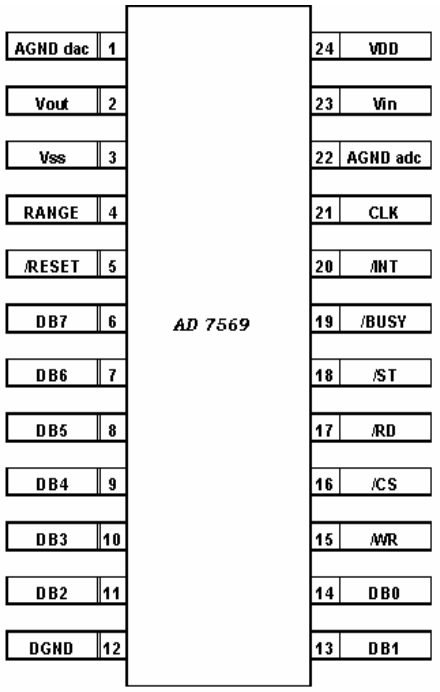


Figure 6: *Brochage du circuit AD 7569.*

PIN	**Fonction**
AGND dac	Masse analogique pour le CNA
Vout	Tension de sortie analogique
Vss	Tension d'alimentation (-5 ou 0 V) en conjonction avec RANGE
RANGE	Entrée de sélection de la plage de tension d'entrée sortie avec Vss
RESET	Entrée asynchrone de remise à zéro (active à l'état bas)
DB7-DB0	Bus de donnée
DGND	Masse digitale
WR	Entrée d'écriture (sur front)
CS	Entrée de sélection de boîtier (active au niveau bas)
RD	Entrée de lecture (active au niveau bas)
ST	Départ de conversion (sur front)
BUSY	Sortie (active au niveau bas), indique une conversion A/D
INT	Sortie interruption (active au niveau bas)
CLK	Un signal d'horloge
AGND adc	Masse analogique de l'ADC
Vin	Entrée analogique
VDD	Tension d'alimentation 5V

Table 2: *Description des broches du convertisseur « AD7569 ».*

ANNEXE 4
Schémas de principe de la carte d'interface