RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique

Département d'Electronique

Mémoire de projet de fin d'études

pour l'obtention du diplôme d'Ingénieur d'Etat en Electronique

---

Utilisation de l'apprentissage profond pour la classification : intégration de la solution sur Pynq et développement d'accélérateurs matériels

---

Imene DJELLAD

Sous la direction de
Dr. Mourad ADNANE

Présenté et soutenu publiquement le 21/06/2017

Composition du Jury :

Promoteur	M. Nicolas FARRUGIA	MCF	
Co-promoteur	M. Mourad ADNANE	Dr.	ENP
Président	M. Adel BELOUCHRANI	Prof.	ENP
Examinateur	M. Rabah SADOUN	Dr.	ENP

ENP 2017

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique

Département d'Electronique

Mémoire de projet de fin d'études

pour l'obtention du diplôme d'Ingénieur d'Etat en Electronique

---

Utilisation de l'apprentissage profond pour la classification : intégration de la solution sur Pynq et développement d'accélérateurs matériels

---

Imene DJELLAD

Sous la direction de
Dr. Mourad ADNANE

Présenté et soutenu publiquement le 21/06/2017

Composition du Jury :

Promoteur	M. Nicolas FARRUGIA	MCF	
Co-promoteur	M. Mourad ADNANE	Dr.	ENP
Président	M. Adel BELOUCHRANI	Prof.	ENP
Examinateur	M. Rabah SADOUN	Dr.	ENP

ENP 2017

Remerciements

Ce travail n'aurait pas pu être possible sans l'aide et les conseils de nombreuses personnes.

Tout d'abord, je voudrai remercier ma famille et toutes les personnes qui m'ont aidé durant cette période et qui sauront se reconnaitre.

Ensuite, je souhaiterai remercier mon encadrant, enseignant et superviseur Dr. Mourad ADNANE pour son soutien.

Je voudrai aussi remercier mes encadrants Pr. Nicolas FARRUGIA et Ghouti BOUCLI HACENE sans lesquels ce stage n'aurait pas pu avoir lieu.

Je suis très reconnaissante envers tous les membres constituants le jury : M. Adel BELOUCHRANI , M. Rabah SADOUN et M. Mourad ADNANE de m'avoir fait l'honneur d'examiner mon mémoire.

ملخص

في هذا الموجز سنرى مشكلة تصنيف الأرقام المكتوبة بخط اليد و الطريقة التي تستخدمها الشبكات العصبية الاصطناعية لمعالجة هاته المشكلة و ونقترح نماذج مختلفة من الشبكات العصبية التي سننفذها على
FPGA . ثم سنبحث على إمكانيات تسريع الأجهزة لتحسين الحسابات التي أجريت من قبل هذه الشبكات.

الكلمات الدالة : الشبكة العصبية، تسريع الأجهزة ،تصنيف، FPGA ، مستشعر

Abstract

In this thesis, we will discuss the problem of handwritten classification, we will see how artificial neural networks approaches this problem and we will propose different modals that we will implement on FPGA. Then we will explore the possibilities of hardware acceleration in order to optimize the calculations made in the FPGAs by these networks.

Key words : Neural Network, classification, FPGA, implementation , acceleration.

Résumé

Dans ce mémoire, nous aborderons le problème de la classification des chiffres manuscrits, nous verrons comment les réseaux de neurones artificiels traitent ce problème et nous en proposerons des modèles que nous implémenterons sur FPGA. Puis nous explorerons les possibilités d'accélération matérielle afin d'optimiser les calculs réalisés dans les FPGA par ces réseaux.

Mots clés : Réseaux de neurones, classification, FPGA, implantation, accélération.

Table des matières

Liste des tableaux

Liste des figures

Liste des abréviations

ABI Application Binary Interface
AMBA Advanced Microcontroller Bus Architecture
API Application Programming Interface
AXI Advanced eXtensible Interface
BRAM Block Ram
CBL Configurable Logic Blocks
CFFI C Foreign Function Interface
CIFAR Canadian Institute for Advanced Research
Conv Convolutional
DDR Double Data Rate
DMA Direct Access Memory
FF Flip Flop
FPGA Field-Programmable Gate Array
GPIO General Purpose Input/Output
HDMI High Definition Multimedia Interface
IIC Inter-Integrated Circuit
IMT Institut Mines-Télécom
IOB Input/Output Buffers
LED Light-Emitting Diode
MMCM Mixed-Mode Clock Manager
MMIO Memory-mapped Input/Output
MNIST Modified ou Mixed National Institute of Standards and Technology
MLP MultiLayer Perceptron
PL Programmable Logic
PYNQ Python productivity for Zynq
PS Processing System
QSPI Quad Serial Peripheral Interface
RAM Random Access Memory
ReLU Rectified Linear Unit
SD Secure Digital
SDRAM Synchronous Dynamic Random Access Memory
SOC System On Chip
SPI Serial Peripheral Interface
TCL Tool Command Language

UART Universal Asynchronous Receiver Transmitter
USB Universal Serial Bus
VHDL VHSIC Hardware Description Language
VHSIC Very High Speed Integrated Circuit

Contexte et introduction

Contexte

IMT Atlantique (fusion de Telecom Bretagne (sites à Brest, Rennes et Toulouse) et Mines Nantes) est une grande école française d'ingénieurs dont le niveau national est élevé et reconnu. C'est une grande école associant numérique, énergie et environnement, elle est également un centre de recherche international dans les sciences et technologies de l'information. De plus, elle est partenaire avec plus de 50 établissements d'enseignement supérieur et de recherche en Europe et dans le monde entier et elle collabore avec de nombreux instituts réputés tel que le MIT (Massachusetts Institute of Technology).

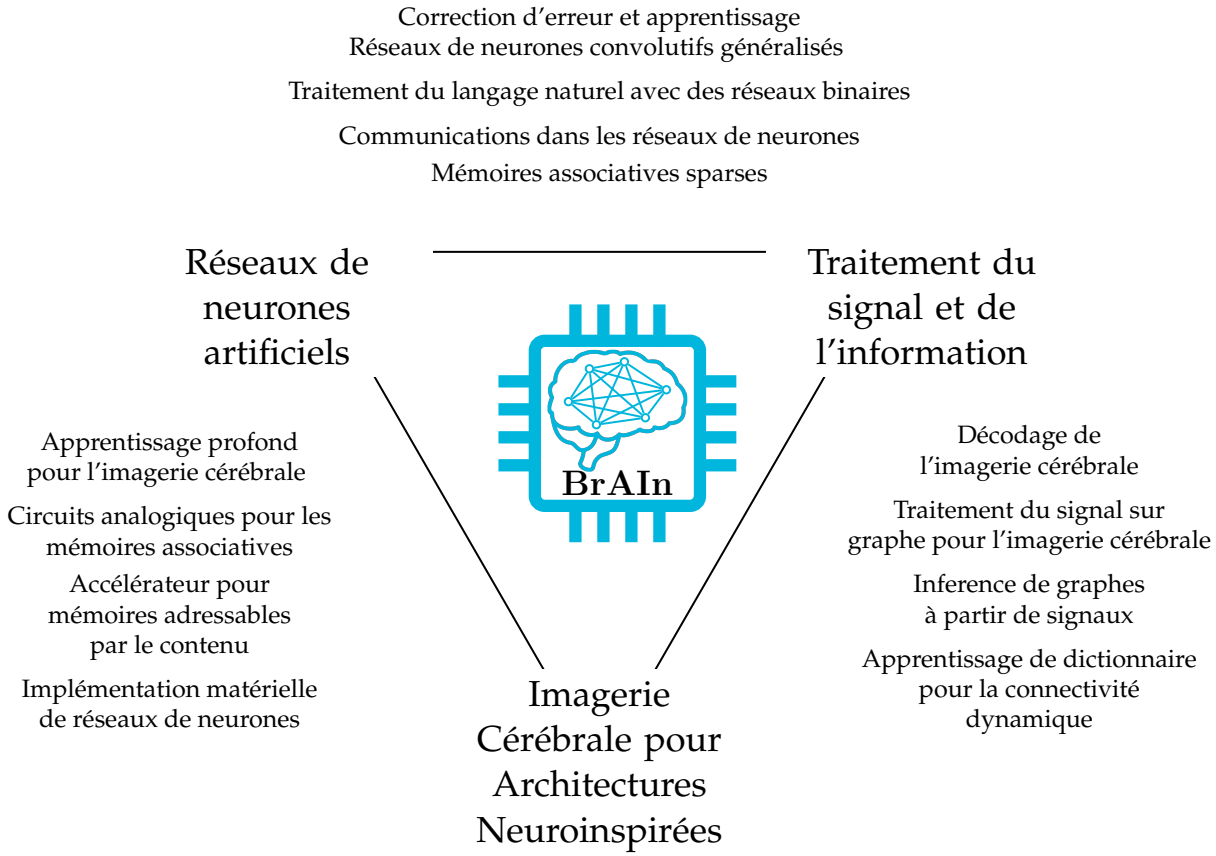
Grâce à son niveau international, l'école accueille des milliers d'étudiants venant de 40 pays désireux de poursuivre des formations d'ingénieur, de master recherche, de master spécialisé et de doctorat.

IMT Atlantique dispose de neuf départements d'enseignement et de recherche :
- Électronique où j'ai effectué mon stage.
- Informatique
- Traitement d'image et traitement de l'information.
- Langues et culture internationale.
- Logique des usages, sciences sociales et sciences de l'information.
- Micro-ondes.
- Optique.
- Réseaux, sécurité et multimédia.
- Signal et communications.

Le département électronique met en œuvre des moyens modernes de conception des composants de l'électronique. Il dispose des outils informatiques et de matériels de mesure très performants pour donner aux étudiants et aux chercheurs une formation adaptée aux réalités de la micro-électronique.

L'équipe BrAIn (Brain Inspired Artificial Inteligence) du département d'électronique de l'IMT Atlantique travaille à la frontière de plusieurs domaines, tels que l'informatique, l'électronique, la psychologie et les neurosciences, pour établir des

modèles à la fois efficaces et biologiquement plausibles.
Le laboratoire a fondé le projet NEUCOD (pour Neural Coding) qui vise à identifier et exploiter les analogies observées entre les propriétés du cortex cérébral, et celles des décodeurs correcteurs d'erreurs modernes. Les sujets les plus répandus en informatique dans l'équipe sont : les réseaux à clique dits Gripon-Berrou, le traitement de signal sur graphe, l'analyse des EEG et les modèles neuronaux en général.



Introduction générale

Nous utilisons l'intelligence artificielle chaque jour à notre insu. Les exemples sont nombreux, celui qui nous intéressera le plus pour ce projet sera la reconnaissance d'images et plus précisément la reconnaissance de l'écriture manuscrite.

Cette tâche pourrait paraitre anodine pour n'importe quel être humain car cette reconnaissance est effectuée inconsciemment, or pour une machine cela n'est pas aussi simple.
La plupart des gens reconnaissent sans effort les séquences de chiffres manuscrits. Dans chaque hémisphère du cerveau humain, on trouve un cortex visuel primaire contenant 140 millions de neurones, avec des dizaines de milliards de connexions entre eux. Et pourtant, la vision humaine implique non seulement ce cortex, mais une série complète de cortex visuels faisant progressivement un traitement d'images plus complexe. Nous portons dans nos têtes un superordinateur, adapté par l'évolution sur des centaines de millions d'années pour comprendre le monde visuel. La reconnaissance des chiffres manuscrits n'est pas simple.

La difficulté devient évidente quand nous essayons d'écrire un programme informatique pour reconnaître une séquence de chiffres manuscrits. Ce qui semble facile lorsque nous le faisons nous-mêmes devient soudainement extrêmement difficile. Des intuitions simples sur la façon dont nous reconnaissons les formes ne sont pas si simples à exprimer de manière algorithmique. Lorsque nous essayons de rendre ces règles précises, nous nous perdons rapidement dans un marasme d'exceptions, de mises en garde et de cas spéciaux.

Les réseaux de neurones abordent le problème d'une manière différente. L'idée est de prendre un grand nombre de chiffres manuscrits, connus sous le nom d'exemples de formation et de lui donner la réponse associée. Et ensuite développer un système qui peut apprendre de ces exemples de formation. En d'autres termes, le réseau de neurones utilise les exemples pour inférer automatiquement des règles pour la reconnaissance des chiffres manuscrits, il calculera son degré de précision et à chaque fois ajustera ses paramètres pour se rapprocher de la bonne réponse. Le système apprend tout seul en utilisant la base de données avec laquelle on l'alimente. De plus, en augmentant le nombre d'exemples de formation, le réseau peut en apprendre davantage sur l'écriture manuscrite et ainsi améliorer sa précision.

La reconnaissance de l'écriture manuscrite est utilisée quotidiennement pour le tri automatique du courrier, le traitement informatisé de certains dossiers administratifs, le traitement automatique des chèques bancaires,...

Dans ce projet, plusieurs réseaux de neurones sont proposés, ils ont été entrainés grâce à la base de données MNIST qui est une base de données de chiffres écrits à la main en utilisant le langage de programmation PYTHON et en utilisant les biblio-

thèques de base.
Cela pour permettre l'utilisation de l'apprentissage profond avec des réseaux de neurones afin d'apprendre à classifier des données.

Parmi ces réseaux de neurones, un réseau a été intégré sur la carte de développement PYNQ (Python Productivity for ZYNQ) et un travail d'accélération des calculs dans ce réseau a été entrepris.

Ce travail comporte 5 chapitre en comptant la conclusion.
Le chapitre 1 introduit le concept de réseaux de neurones et les bases requises afin de comprendre les étapes de création d'un réseau de neurones, son entrainement et validation sur une base de données bien défini et son intégration dans la carte de développement.
Le chapitre 2 nous présente la carte de développement PYNQ (Python Productivity for ZYNQ) qui servira par la suite à implémenter le réseau de neurones. Nous comprendrons alors pourquoi notre choix s'est porté sur cette carte, nous verrons ces divers constituants, comment travailler sur cette carte mais aussi les différentes interfaces et bus de communication qui ont facilité de communication avec la carte même après modification de l'architecture de la partie FPGA.
Dans le chapitre 3, nous allons voir comment entrainer, valider, tester et implémenter un réseau de neurones dans la PYNQ avant de voir dans le chapitre 4, comment accélérer matériellement les calculs dans ce même réseau pour permettre des calculs parallèles plus rapides et moins couteux.
Pour finir, la conclusion représente une synthèse du travail effectué et des perspectives futures.

Chapitre 1

Réseaux de neurones

1.1 Introduction

Les réseaux de neurones formels sont à l'origine inspirés des réseaux de neurones biologiques, ce sont des réseaux complexes d'unités de calcul élémentaires interconnectées
Il existe différentes structures de réseaux de neurones selon le nombre de couches du réseau et le nombre de neurones constituants chaque couche.
Les programmes apprennent automatiquement à reconnaître des modèles complexes et à prendre des décisions intelligentes fondées sur une expérience générée par l'apprentissage.
Pour permettre une grande précision, le réseau de neurones doit être formé, testé et calibré pour détecter les modèles en utilisant les expériences précédentes.
Dans ce chapitre, nous allons plonger dans le monde de l'intelligence artificielle à travers une brève introduction aux réseaux de neurones et ceci dans le but de mieux comprendre les notions plus avancées qui seront abordés plus loin sur les réseaux de neurones.

1.2 Qu'est-ce qu'un réseau de neurones ?

Un réseau de neurones est l'association en graphe d'un certain nombre de neurones formels. Il existe différentes structures de réseaux de neurones selon le nombre de couches du réseau et le nombre de neurones constituants chaque couche.
Les réseaux de neurones se distinguent sur différents autres critères.
Ces réseaux sont à l'origine inspirés des réseaux de neurones biologiques.

1.3 Qu'est-ce qu'un neurone formel ?

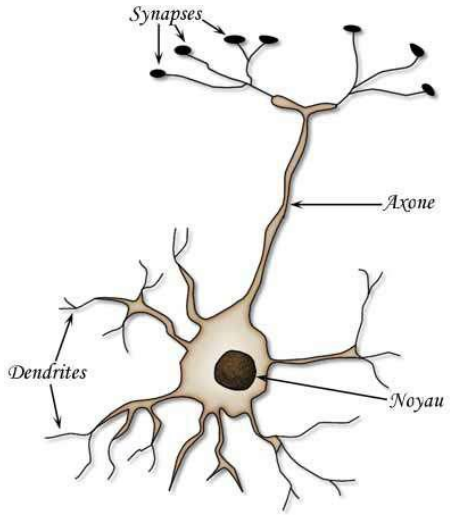
1.3.1 Définition

Le neurone formel est l'élément élémentaire constituant un réseau de neurones. Il est connecté à des données en entrées. Dans le cas des réseaux de neurones séquentiels, ces entrées peuvent êtres les neurones d'une couche inférieur correspondant à une source d'informations pour la couche de neurones qui suit. En sortie, ce neurone renvoie à son tour une information qui peut aussi être destinée à la couche suivante de neurones.

1.3.2 Neurone biologique

Le neurone artificiel étant inspiré du neurone biologique nous devons voir les constituants du neurone biologique. Le neurone biologique se compose de différentes parties essentielles illustrés sur la figure 1.1 et qui sont :

- les synapses : ce sont les différentes connexions qui lient un neurone avec les autres neurones. Pour le neurone biologique cela représente les fibres nerveuses ou musculaires ;
- l'axone : c'est la sortie du neurone qui peut être liée à d'autres neurones ;
- le noyau : c'est la partie qui active la sortie en fonction des stimulations à l'entrée du neurone ;
- les dendrites : ce sont des extensions du neurone qui reçoivent les informations provenant d'autres neurones et transmettent la stimulation électrique.



[1]

FIGURE 1.1: Neurone biologique

1.3.3 Analogie entre le neurone biologique et le neurone artificiel ou formel

- Les dendrites du neurone biologique représentent les entrées du neurone artificiel.
- Les synapses du neurone biologique représentent les poids du neurone artificiel.
- Le noyau du neurone biologique représente la fonction d'activation du neurone artificiel.
- L'axone du neurone biologique représente la sortie du neurone artificiel.

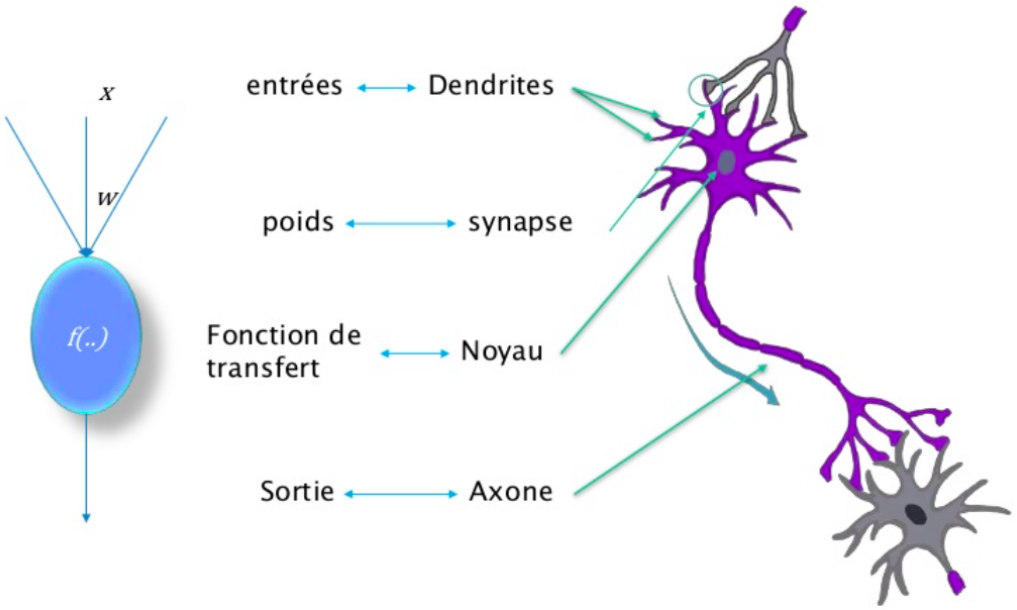
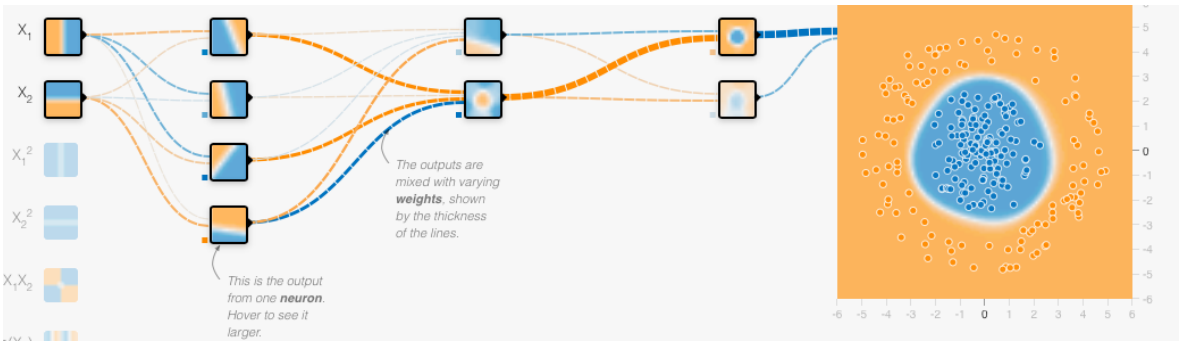


FIGURE 1.2: Analogie entre neurone biologique et neurone formel [2]

1.3.4 Neurone formel

Comme nous l'avons dit un neurone est l'élément élémentaire de traitement d'un réseau, comme le montre la figure 1.2, il est caractérisé par un poids souvent noté "w". Ce poids correspond à une valeur réelle indiquant la force de l'information du neurone dans le réseau.
Un neurone dont l'information est très pertinente et qui affecte beaucoup la sortie du réseau aura un poids de valeur très grande comparé à un poids qui n'apporte aucune information ou qui apporte une information n'ayant aucun effet sur la sortie du réseau de neurones.
Dans la figure présentée ci-dessous 1.3, la force des poids est illustrée par la largeur des connexions. On peut bien voir que dans le même réseau, différents neurones sont caractérisés par des poids de différentes valeurs. Les liaisons épaisses signifient

des valeurs plus grandes que les liaisons fines et les deux couleurs (bleu et orange) illustrent les deux classes de données présentes.



[3]

FIGURE 1.3: Exemple de réseau de neurones

Un neurone peut recevoir à son entrée plusieurs informations qu'il somme et sur lesquelles il applique une fonction d'activation avant de renvoyer le résultat en sortie, cette somme est pondérée par un coefficient multiplicatif associé à chaque entrée, c'est le poids.
La figure 1.4 illustre la force des poids.

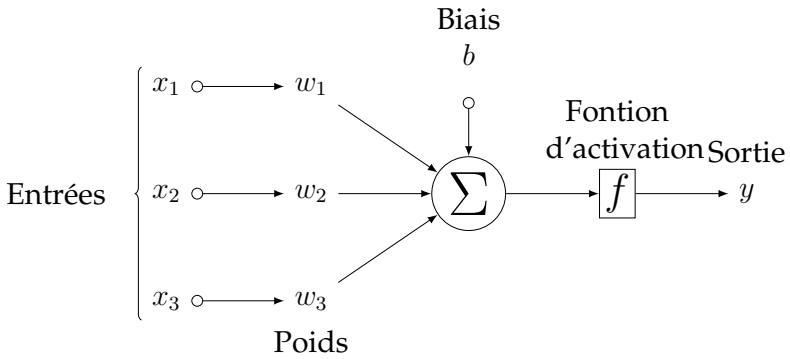


FIGURE 1.4: Traitement au niveau du neurone artificiel

1.4 Fonction d'activation

La fonction d'activation est avant tout une fonction mathématique dont l'entrée est l'entrée du neurone ou la somme pondérée des entrées du neurone et la sortie représente la sortie du neurone.

La fonction d'activation, ou fonction de transfert, est une fonction qui doit renvoyer un réel proche de 1 quand les "bonnes" informations d'entrée sont données et un réel proche de 0 quand elles sont "mauvaises". On utilise généralement des fonctions à valeurs dans l'intervalle réel [0,1]. Quand le réel est proche de 1, on dit que le neurone est actif alors que quand le réel est proche de 0, on dit que le neurone est inactif. Le réel en question est appelé la sortie du neurone. Si la fonction d'activation est linéaire, le réseau de neurones se réduirait à une simple fonction linéaire.

En effet, si les fonctions d'activation sont linéaires, alors le réseau est l'équivalent d'une régression multi-linéaire (méthode utilisée en statistiques) . L'utilisation du réseau de neurone est toutefois bien plus intéressante lorsque l'on utilise des fonctions d'activation non linéaires.

Il existe différentes fonctions d'activation linéaires ou non linéaire.
Pour les problèmes de classification nous avons : la fonction d'activation logistique ou sigmoïd et la fonction d'activation tangente hyperbolique qui sont très utilisées.

1.5 Apprentissage

L'apprentissage est l'estimation des paramètres d'un réseau de neurones soit les poids par minimisation de l'erreur commise sur la prédiction en sortie.
Il existe différents algorithmes pour l'apprentissage des réseaux de neurones.

L'idée de l'apprentissage consiste à forcer le réseau de neurones à se généraliser. Un temps énorme est consacré à l'analyse de l'ensemble de données servant à l'entrainement du réseau de neurones, mais dans le monde réel, la performance du réseau de neurones sur cet ensemble n'a absolument aucun intérêt. Presque tout réseau de neurones peut être entrainé de sorte à ne pas rencontrer d'erreurs sur l'ensemble d'entrainement. La vraie question est, à quel point le réseau de neurones fonctionnera-t-il dans le monde réel sur des modèles qu'il n'a jamais vus auparavant ?

La capacité d'un réseau de neurones à bien fonctionner sur des modèles qu'il n'a jamais vus auparavant s'appelle «généralisation». L'effort déployé dans l'apprentissage consiste à forcer le réseau de neurones à voir des caractéristiques intrinsèques dans l'ensemble d'apprentissage, ce qui, dans le meilleur des cas, sera également présent dans le monde réel.

Quelques techniques sont utilisées pendant l'apprentissage dans le but d'améliorer la généralisation du réseau de neurones. À titre d'exemple, l'ensemble d'entrainement est fourni à l'entrée du réseau de neurones dans un ordre aléatoire à chaque époque. De manière conceptuelle, ces techniques font que les poids "rebondissent" un peu, empêchant ainsi le réseau de neurones de garder des poids qui

mettent l'accent sur les faux attributs des motifs, c'est-à-dire des attributs propres aux modèles dans l'ensemble d'entrainement mais qui ne sont pas des caractéristiques intrinsèques des modèles rencontrés dans le monde réel.

Pour ce faire, des algorithmes d'apprentissage itératifs sont utilisés, ils permettent d'ajuster graduellement les poids pour atteindre le maximum de précision possible. Il faut d'abord, initialiser les poids, ceci se fait souvent en utilisant une fonction permettant d'assigner des valeurs aléatoires suivant les distributions de probabilités Gaussiennes ou Normales.
Durant cette période, un ensemble d'exemples est présenté au réseau, il doit alors faire des suppositions sur les sorties suivant les valeurs actuelles de ses poids puis tester ses performances. Les paramètres sont alors ajustés afin de produire une réponse exacte.
L'algorithme le plus souvent utilisé est le gradient descendant, cet algorithme procède par améliorations successives de telle sorte à minimiser l'erreur commise sur la précision à chaque itération

1.6 Exemple de réseau de neurones : le perceptron

Ce réseau est constitué d'un seul neurone formel. Dans l'exemple présenté cidessous 1.5, le perceptron a trois entrées x1, x2, x3. Dans le cas général, ce nombre pourrait augmenter ou diminuer selon le contexte.

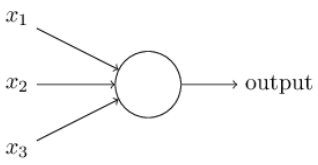


FIGURE 1.5: Perceptron

La sortie est calculée de façon très simple, en introduisant des poids notés : w1, w2, .., qui représentent des nombres réels exprimant l'importance de chaque entrée à laquelle ils sont liés. Le neurone renvoie en sortie 0 ou 1 selon que la somme des entrées pondérées par les sorties soit inférieure ou supérieure à un seuil donné.

$$sortie = \begin{cases} 0 & \text{si } \sum_j w_j x_j \leq seuil; \\ 1 & \text{si } \sum_j w_j x_j > seuil. \end{cases}$$

Ceci est donc un modèle mathématique basique qui peut être vu comme un outil de prise de décision en donnant un degré d'importance à chaque entrée et qui en sortie essaie de satisfaire au maximum les exigences.

Un réseau de neurones plus complexe peut être constitué en combinant plusieurs perceptrons sur plusieurs couches afin de pouvoir prendre des décisions plus complexes, comme on peut le voir sur la figure ci-dessous 1.6

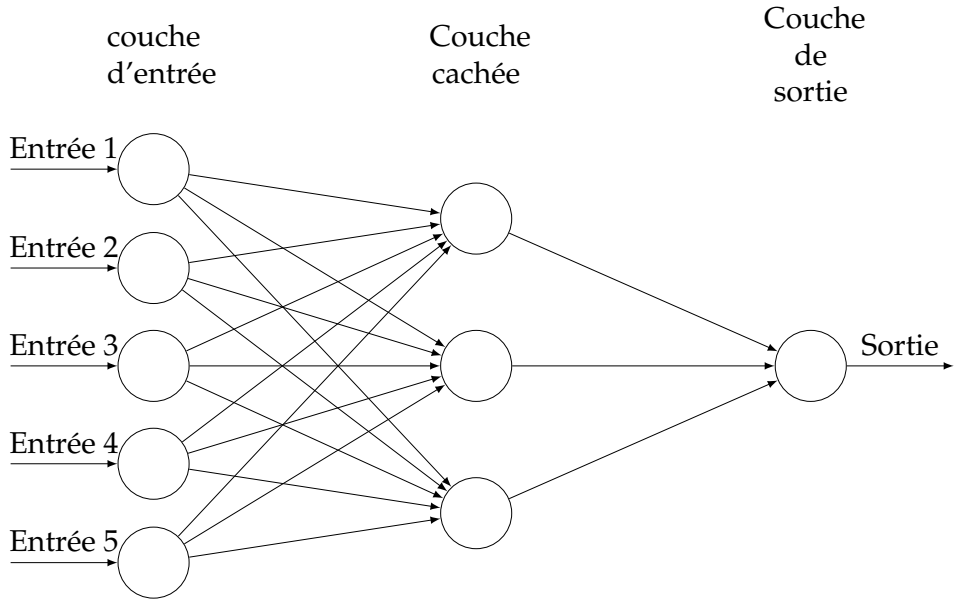


FIGURE 1.6: MLP

Chaque perceptron de la couche cachée prend une décision par la sommation pondérée par les poids de ses entrées, soit les sorties des perceptrons de la couche précédente.
Dans le cas général, on ne parle pas souvent du seuil d'activation, on parle plutôt du biais. La valeur de ce biais correspond à la valeur du seuil, on l'illustre souvent par un neurone qu'on rajoute à chaque couche tel que b=-seuil. On aura alors la sortie du neurone recevant les valeurs ci-dessous :

$$sortie = \begin{cases} 0 & \text{si } wx + b \leq 0; \\ 1 & \text{si } wx + b > 0. \end{cases}$$

Les poids et les biais sont les paramètres d'un réseau de neurones, leur valeur changent durant la phase d'apprentissage pour se fixer à des valeurs précises à la fin de cette phase. Ces valeurs fixées seront par la suite utilisées pour prédire des résultats sur de nouvelles données. Leur précision estime donc leur capacité à généraliser leur prédiction d'après leur expérience.
Le problème, avec ces perceptrons, est que durant la phase d'apprentissage, un petit changement dans la valeur des paramètres peut impliquer un grand changement

dans la sortie, or nous voudrions qu'un petit changement dans les paramètres implique un petit changement dans la sortie et c'est pour cela qu'on utilise une fonction d'activation dont l'entrée est la somme pondérée par les poids dont on a parlé précédemment. La sortie n'est donc plus 0 ou 1 mais peut prendre n'importe quelle valeur entre 0 et 1.

1.7 Perceptron multicouches (réseau de neurones complètement connecté)

Un réseau de neurones reçoit une entrée (un seul vecteur) et le transforme en une série de couches cachées. Chaque couche cachée est constituée d'un ensemble de neurones, où chaque neurone est entièrement connecté à tous les neurones de la couche précédente et où les neurones d'une seule couche fonctionnent de manière totalement indépendante et ne partagent aucune connexion. La dernière couche entièrement connectée s'appelle «couche de sortie» et dans les paramètres de classification, elle représente les scores de classe.

Les réseaux de neurones réguliers ne s'adaptent pas bien aux images complètes. Dans certaines bases de données comme CIFAR-10, les images ne sont que de taille 32 X 32 X 3 (32 largeur, 32 haute, 3 canaux de couleur), de sorte qu'un seul neurone entièrement connecté dans une première couche cachée d'un réseau de neurones régulier aurait 32 X 32 X 3 = 3072 poids . Ce montant semble toujours gérable, mais clairement, cette structure entièrement connectée ne s'adapte pas aux grandes images. Par exemple, une image de taille plus respectable, par exemple 200x200x3, conduirait à des neurones qui ont 200 X 200 X 3 = 120 000 poids. En outre, nous voudrions presque certainement avoir plusieurs de ces neurones, de sorte que les paramètres s'ajoutent rapidement! De toute évidence, cette connectivité complète est un gaspillage et le nombre énorme de paramètres entraînerait rapidement une surutilisation.

1.8 Réseaux de neurones convolutifs

Les réseaux de neurones convolutifs profitent du fait que l'entrée soit des images et qu'elles limitent l'architecture de manière plus judicieuse. Contrairement à un réseau de neurones régulier, les couches d'un ConvNet ont des neurones disposés en 3 dimensions : largeur, hauteur, profondeur. (Notez que la profondeur du mot se réfère ici à la troisième dimension d'un volume d'activation, et non à la profondeur d'un réseau de neurones complet, qui peut se référer au nombre total de couches dans un réseau.) Par exemple, les images d'entrée dans CIFAR- 10 sont un volume d'entrée des activations, et le volume a des dimensions 32x32x3 (largeur, hauteur, profondeur, respectivement). Comme nous le verrons bientôt, les neurones dans une couche ne seront connectés qu'à une petite région de la couche avant, au lieu de

tous les neurones d'une manière entièrement connectée. En outre, la couche de sortie finale aurait pour CIFAR-10 des dimensions 1x1x10 car, à la fin de l'architecture ConvNet, nous réduirons l'image complète en un seul vecteur de scores de classe, disposé selon la dimension de profondeur.

A l'entrée de ces réseaux nous avons comme dans le cas étudié la base de données MNIST mais dans le cas général cela peut être différentes images avec un contenu très varié.
Ces réseaux de neurones contiennent une suite de couches convolutives, MAX POOLING, RELU et à la fin des couches connectées (FC layer : Full Connected layer). Pour augmenter la précision on joue souvent sur le nombre de couches car on peut mettre plusieurs couches convolutives ainsi que plusieurs couches MAX POOLING et RELU. Un exemple est illustré ci-dessous 1.7 et 1.8.

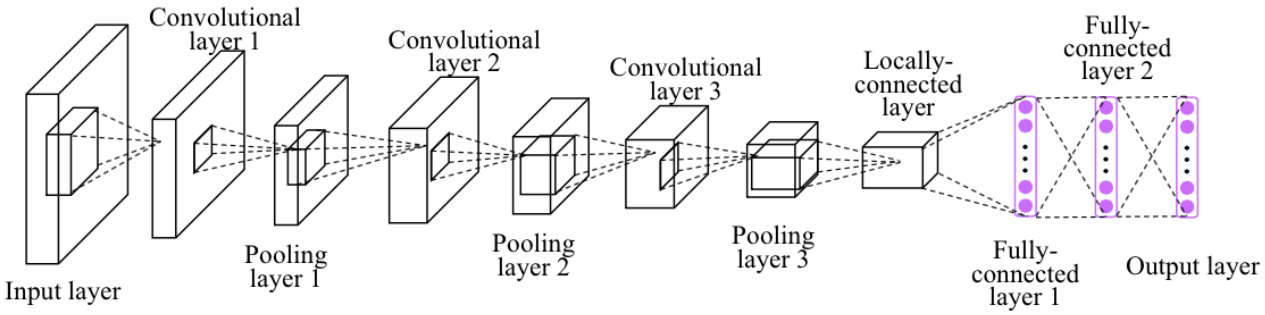


FIGURE 1.7: Exemple de réseau de neurones convolutifs
[4]

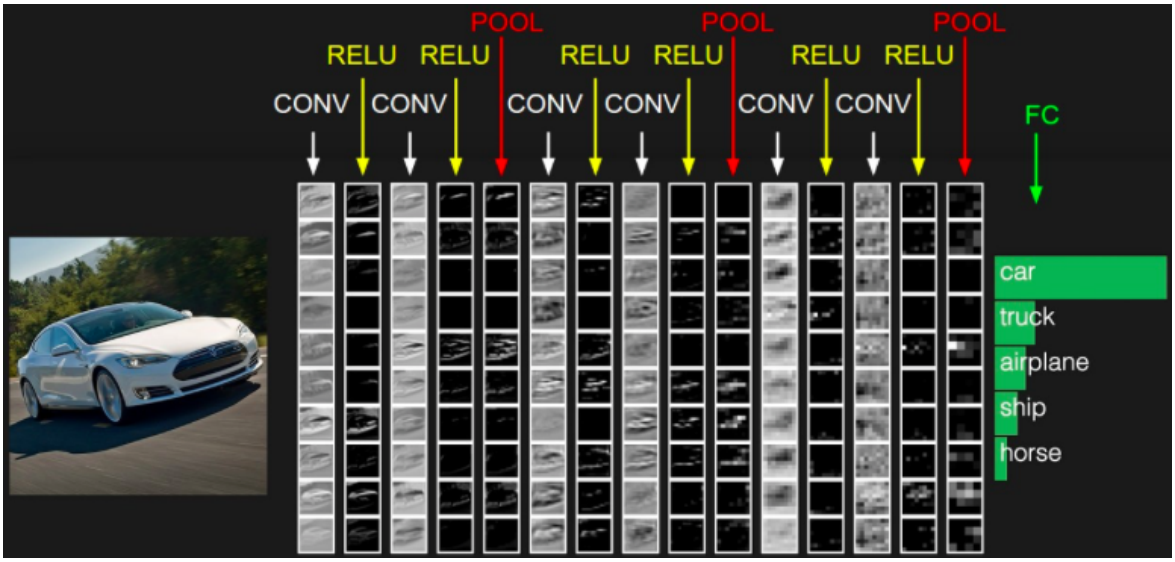


FIGURE 1.8: Exemple de traitement du réseau de neurones convolutifs [5]

Les réseaux de neurones convolutifs sont très semblables aux réseaux de neurones ordinaires qu'on a vus précédemment : ils sont composés de neurones qui ont des poids et des biais appréciables. Chaque neurone reçoit certaines entrées, calcul un produit suivi éventuellement d'une non-linéarité. Tout le réseau exprime encore une seule fonction de score différenciable : des pixels d'image d'un coté au score à l'autre coté. Et ils ont toujours une fonction de perte (par exemple Softmax) sur la dernière couche (entièrement connectée) et toutes les méthodes d'apprentissages des réseaux de neurones ordinaires s'appliquent toujours.

Alors, qu'est-ce qui change ?
Ce qui change, c'est que les architectures ConvNet font l'hypothèse explicite que les entrées sont des images, ce qui nous permet d'encoder certaines propriétés dans l'architecture. Ceux-ci rendent la fonction avant plus efficace pour la mise en œuvre et réduisent considérablement la quantité de paramètres dans le réseau.

1.8.1 Couche CONV

La couche CONV est le bloc de base d'un réseau de neurones convolutifs, ce bloc effectue la partie la plus lourde des calculs dans un réseau convolutif.
La vue du cerveau : en faisant une petite analogie cerveau / neuron, chaque entrée dans le volume de sortie 3D peut également être interprétée comme une sortie d'un neurone qui ne regarde qu'une petite région dans l'entrée et partage les paramètres avec tous les neurones à gauche et Dans le sens spatial (puisque ces nombres résultent tous de l'application du même filtre).

En ce qui concerne les entrées haute dimension telles que les images, comme

nous l'avons vu plus haut, il est peu pratique de connecter les neurones à tous les neurones du volume précédent. Au lieu de cela, nous allons connecter chaque neurone à une région locale du volume d'entrée. L'étendue spatiale de cette connectivité est un hyper paramètre appelé le champ réceptif du neurone (en équivalent, c'est la taille du filtre). L'étendue de la connectivité selon l'axe de la profondeur est toujours égale à la profondeur du volume d'entrée. Il est important de souligner cette asymétrie dans la façon dont nous traitons les dimensions spatiales (largeur et hauteur) et la dimension de profondeur : les connexions sont locales dans l'espace (le long de la largeur et de la hauteur), mais toujours complètes sur toute la profondeur du volume d'entrée.

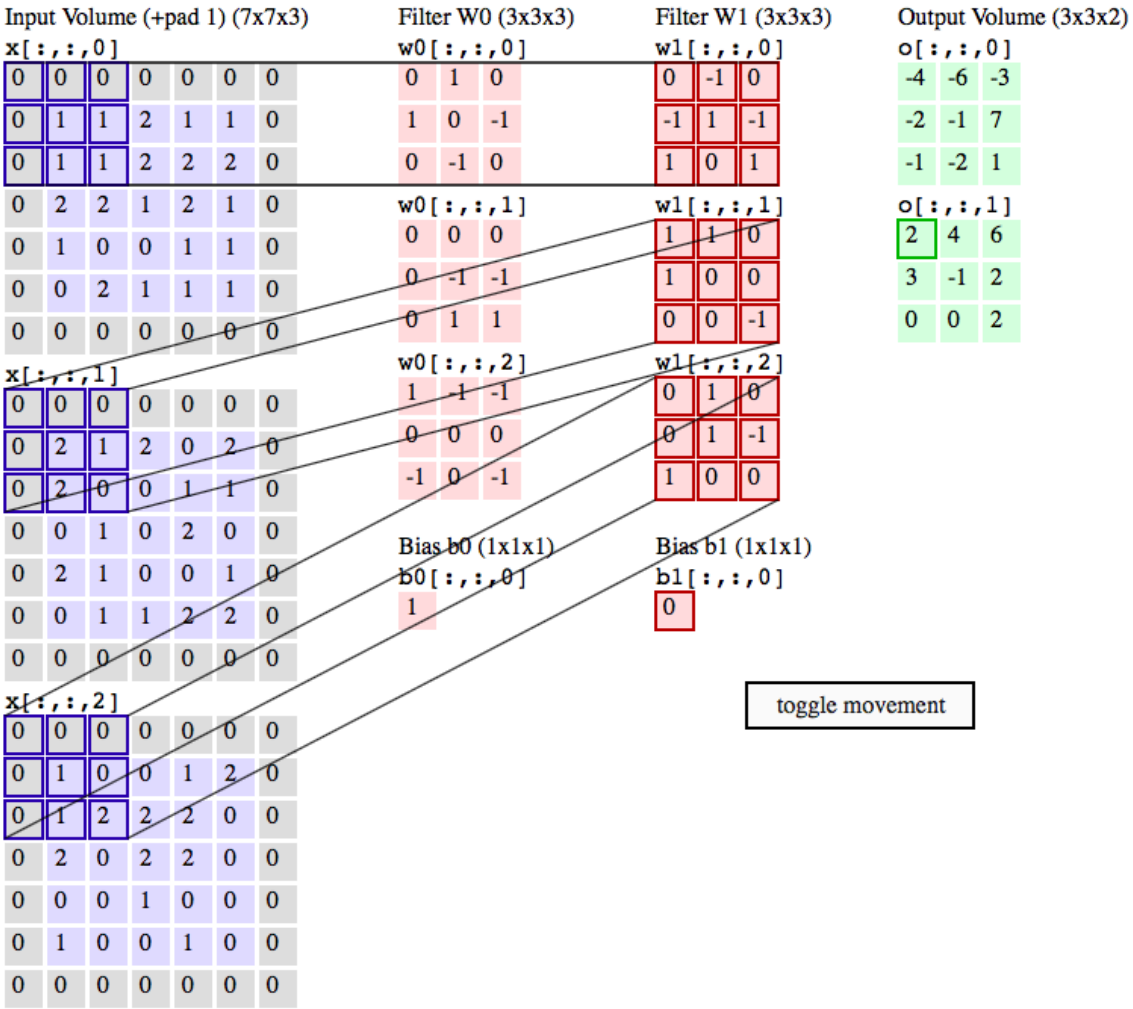


FIGURE 1.9: Exemple de filtrage sur une image de profondeur 3 [5]

Exemple 1 : supposons que le volume d'entrée ait une taille [32 X 32 X 3] (par

exemple une image RGB CIFAR-10). Si le champ réceptif (ou la taille du filtre) est 5x5, chaque neurone de la couche CONV aura des poids à une région [5x5x3] dans le volume d'entrée, pour un total de 5 X 5 X 3 = 75 poids (et +1 Paramètre de polarisation). Notez que l'étendue de la connectivité le long de l'axe de profondeur doit être de 3, puisqu'il s'agit de la profondeur du volume d'entrée.

Exemple 2 : supposons qu'un volume d'entrée ait une taille [16x16x20]. Ensuite, en utilisant un exemple de taille de champ réceptif de 3 X 3, chaque neurone de la couche CONV maintenant aurait un total de 3 X 3 X 20 = 180 connexions au volume d'entrée. Notez que, encore une fois, la connectivité est locale dans l'espace (par exemple 3 X 3), mais pleine le long de la profondeur d'entrée (20).

Les paramètres de la couche CONV consistent en un ensemble de filtres. Chaque filtre est de petite taille spatiale (le long de la largeur et de la hauteur), mais s'étend à travers la profondeur totale du volume d'entrée. Par exemple, un filtre typique sur une première couche d'un ConvNet peut avoir une taille 5x5x3 (c'est-à-dire 5 pixels de largeur et de hauteur et une profondeur de 3, car les images ont une profondeur 3 correspondant aux canaux de couleurs). Pendant le passage en avant, chaque filtre glisse sur la largeur et la hauteur du volume d'entrée. Les produits s'effectuent entre les entrées du filtre et l'entrée à n'importe quelle position.
Lorsque le filtre glisse sur la largeur et la hauteur du volume d'entrée, il produit une carte d'activation bidimensionnelle qui donne les réponses de ce filtre à toutes les positions spatiales. Intuitivement, le réseau va apprendre les filtres qui s'activent lorsqu'il voit un certain type de fonctionnalité visuelle, comme un bord d'orientation ou une tache de couleur sur la première couche, ou éventuellement des modèles en forme de nids d'abeille ou de roue sur des couches supérieures du réseau .
Notre réseau finit par produire un ensemble complet de filtres dans chaque couche CONV (par exemple, 12 filtres) et chacun d'eux produira une carte d'activation bidimensionnelle distincte. Nous allons empiler ces cartes d'activation le long de la dimension de profondeur et produire le volume de sortie.
Un exemple de filtrage est illustré sur la figure 1.9.

1.8.2 Couche MAX POOLING

Il est fréquent d'insérer périodiquement une couche POOLING entre les couches conv successives dans une architecture ConvNet. Sa fonction est de réduire progressivement la taille spatiale de la représentation afin de réduire la quantité de paramètres et de calcul dans le réseau et, par conséquent, de contrôler également l'ajustement excessif. Cette couche fonctionne indépendamment sur chaque tranche de profondeur de l'entrée et redimensionne spatialement, en utilisant l'opération MAX. La forme la plus courante est une couche avec des filtres de taille 2 X 2 qu'on fait passer progressivement sur des régions de taille 2x2, telle que sur chaque région de taille 2x2, soit une région contenant 4 nombres, on choisit le nombre le plus grand (le MAX), on élimine alors 75% des paramètres. Chaque opération MAX POOLING

devrait dans ce cas prendre le maximum des 4 nombres (petite région 2x2 dans une tranche en profondeur). Un exemple est illustré sur la figure 1.10

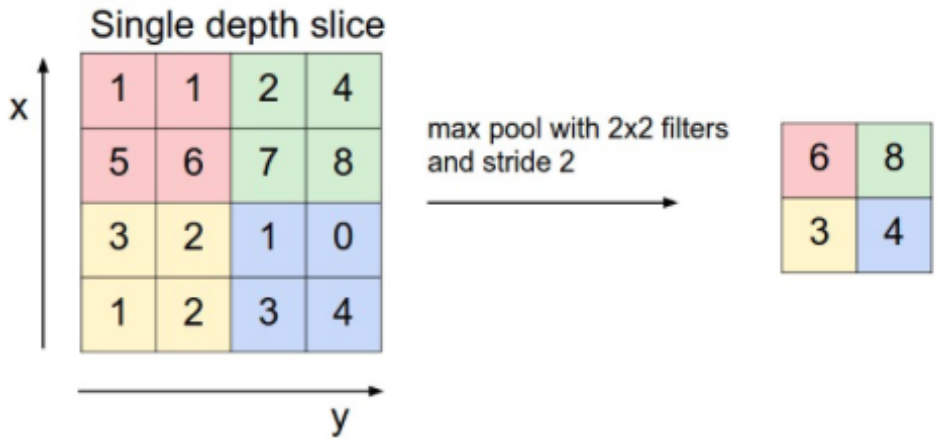


FIGURE 1.10: Exemple d'un MAX POOLING de taille 2 X 2
[5]

1.8.3 Couche RELU

La couche RELU applique une fonction d'activation élémentaire, telle que la max (0, x) seuil à zéro. Cela laisse la taille du volume inchangée avant et après passage de la couche RELU.

1.8.4 Couches connectées

Les neurones dans une couche entièrement connectée sont connectés à toutes les activations de la couche précédente, comme on l'a vu dans les réseaux de neurones précédents 1.6.

1.9 Bases de données

Il existe différentes bases de données utilisées pour entrainer les réseaux de neurones jusqu'à obtention d'un certain degré de précision permettant de comparer les multiples réseaux de neurones proposés pour la résolution d'un même problème. Dans ce qui suit nous allons présenter les bases de données MNIST et CIFAR utilisées dans la reconnaissance d'images.

1.9.1 La base de données MNIST

MNIST est une base de données de chiffres écrits à la main et d'étiquettes pour chaque image spécifiant le chiffre inscrit dans chaque image. Sur la figure 1.11, on peut voir un ensemble de chiffres manuscrits.

FIGURE 1.11: Exemples de chiffres écrits à la main
[6]

1.9.1.1 Division de la base de données MNIST

Cette base de données est constituée de :

- 55000 exemples pour l'entrainement du réseau (mnist.train) ;
- 10000 exemples pour tester le réseau (mnist.test) ;
- 5000 exemples pour valider le réseau (mnist.validation).

Cette division est très importante pour s'assurer que le réseau que nous avons construit lors de l'entrainement arrive à généraliser sur des images ou des types d'écritures qu'il n'a pas rencontré lors de son entrainement.

1.9.1.2 Exemples constituants la base de données MNIST

Chaque exemple se trouvant dans la base de données MNIST se compose donc de l'image d'un chiffre écrit à la main (mnist.train.images) et d'une étiquette (mnist.train.labels) précisant de quel chiffre il s'agit.
Chaque image est de taille 28 pixels par 28 pixels qui peut être interprété par une matrice de nombres. Tel que chaque nombre représente le niveau de gris d'un pixel de l'image, on aura alors une matrice de taille 28X28.
La figure 1.12, nous avons une image du chiffre 1 manuscrit et sa représentations matricielle.

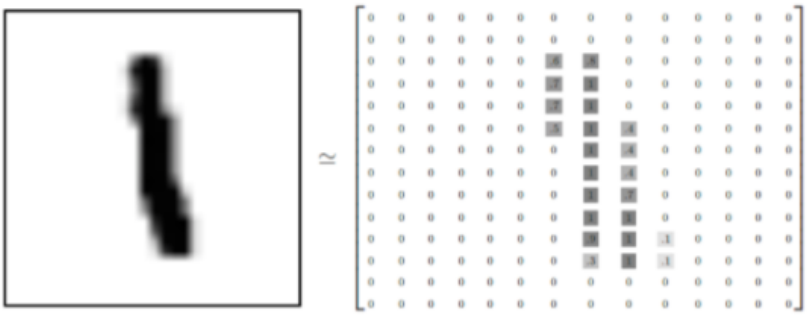


FIGURE 1.12: Représentation du chiffre 1 par une matrice
[7]

1.9.1.3 Représentation des données MNIST

Pour simplifier l'utilisation de ces images, la base de données MNIST est constitué de vecteurs et non de matrices de tailles 28X28=278 nombres.

Ainsi, nous utiliserons des tenseurs dans lesquels nous introduirons selon qu'on entraine, on teste ou on valide notre réseau de neurones, les différentes parties de la base de données.
Par exemple pour l'entrainement nous aurons un tenseur d'une dimension [55000, 784] comme sur la figure 1.13, alors que durant l'entrainement le tenseur sera de dimension [10000, 784].

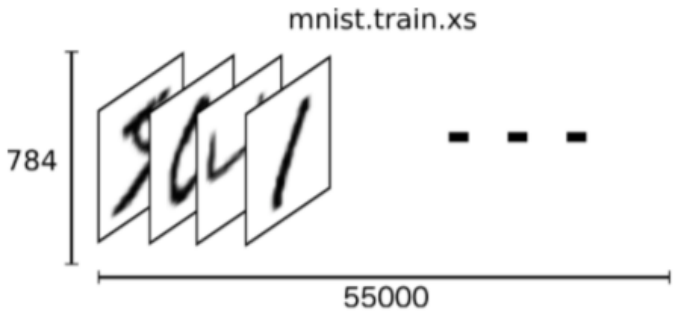


FIGURE 1.13: Représentation du tenseur d'entrainement
[7]

Quant aux étiquettes, elles seront représentées par des vecteurs "one-hot", ce qui veut dire que toutes les dimensions de ce vecteur seront à zéro sauf celle qui représente le chiffre correspondant à l'image.

Par exemple, pour le chiffre 5 nous aurons un vecteur one-hot : [0,0,0,0,0,1,0,0,0,0] [7].

1.9.2 La base de données CIFAR

Les CIFAR-10 et CIFAR-100 sont des sous-ensembles étiquetés du jeu de données de 80 millions d'images minuscules. Ils ont été recueillis par Alex Krizhevsky, Vinod Nair et Geoffrey Hinton.

1.9.2.1 CIFAR-10

L'ensemble de données CIFAR-10 se compose de 60000 images couleur de taille 32x32 regroupées en 10 classes, avec 6000 images par classe. Il y a 50000 images d'entraînement et 10000 images de test, un exemple est illustré sur la figure 1.14.

L'ensemble de données est divisé en cinq lots de formation et un lot de test, chacun avec 10000 images. Le lot de test contient exactement 1000 images sélectionnées au hasard de chaque classe. Les lots de formation contiennent les images restantes dans un ordre aléatoire, mais certains lots d'entraînement peuvent contenir plus d'images d'une classe que l'autre. Les lots de formation contiennent exactement 5000 images de chaque classe.

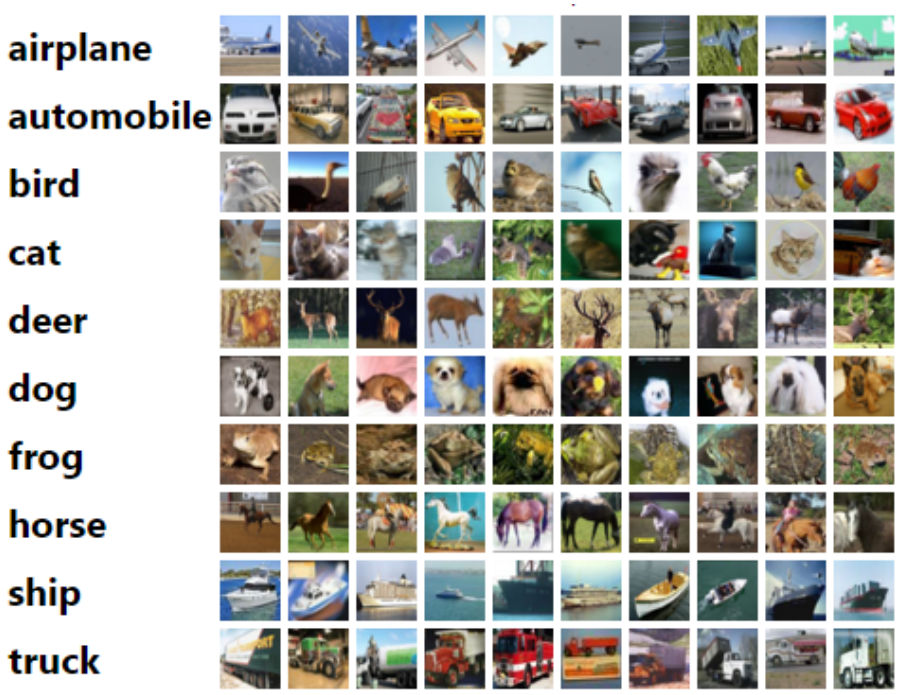


FIGURE 1.14: Classes de la base de données CIFAR-10
[8]

1.9.2.2 CIFAR-100

Cet ensemble de données ressemble à CIFAR-10, sauf qu'il possède 100 classes contenant 600 images chacune. Il y a 500 images d'entraînement et 100 images de test par classe. Les 100 classes du CIFAR-100 sont regroupées en 20 superclasses. Chaque image est livrée avec une étiquette "fine" (la classe à laquelle elle appartient) et une étiquette "grossière" (la superclasse à laquelle elle appartient).
Par exemple, on a les classes : castor, dauphin, loutre, phoque, baleine et mammifères aquatiques est la superclasse auxquelles ces classes appartiennent.

1.10 Conclusion

Dans ce chapitre, nous avons clarifié certaines notions de l'intelligence artificielle. Nous avons pû voir ce qu'est un réseau de neurones et de quoi il est constitué mais aussi comment est-ce qu'il fonctionne. Nous avons aussi vu les bases de données MNIST et CIFAR utilisées pour la reconnaissance d'images.
Ainsi, dans la suite de ce projet nous verrons un exemple réel d'utilisation de réseaux de neurones pour la reconnaissance de chiffres écrits à la main .

Chapitre 2

Introduction à la carte de développement PYNQ

2.1 Introduction

FPGA ou Field-Programmable Gate Array est comme son nom l'indique un circuit logique programmable donc qui peut être programmé après sa fabrication plusieurs fois pour les adapter à la tâche qu'on leur assigne pour réaliser ainsi une accélération dans la phase de calcul.
Les algorithmes d'intelligence artificielle demandent beaucoup de puissance de calcul mais aussi consomment beaucoup d'énergie.
Face à ce problème, le projet Catapult de Microsoft décide de traiter les algorithmes de classement des recherches Bing avec des FPGA et ont constaté une augmentation de performances.
Désormais, les systèmes FPGA sont le choix adapté pour favoriser une plus grande efficacité de traitement.
La principale raison de l'implémentation d'accélérateurs sur FPGA est d'augmenter la vitesse des calculs.

2.2 Pourquoi utiliser un FPGA ?

Il y a diverses raisons qui nous ont incitées à utiliser un FPGA dans ce projet, nous en énumérerons quelques-unes :

- Une des raisons principales est la rapidité des FPGA
- La reconfigurabilité des FPGA permet de tester différents réseaux de neurones
- La grande flexibilité et performance des FPGA
- Le parallélisme des FPGA

2.3 Constitution d'un FPGA

Un FPGA est constituée d'une matrice de blocs logiques programmables (Configurable Logic Blocks CBL) entourés de blocs d'entrées et de sorties (Input/Output Buffers IOB) eux aussi programmables. Ces blocs sont reliés entre eux par un réseaux d'interconnexions elles aussi programmables (Configurable Interconnect Mesh). La figure ci-dessous illustre cet ensemble de blocs 2.1.

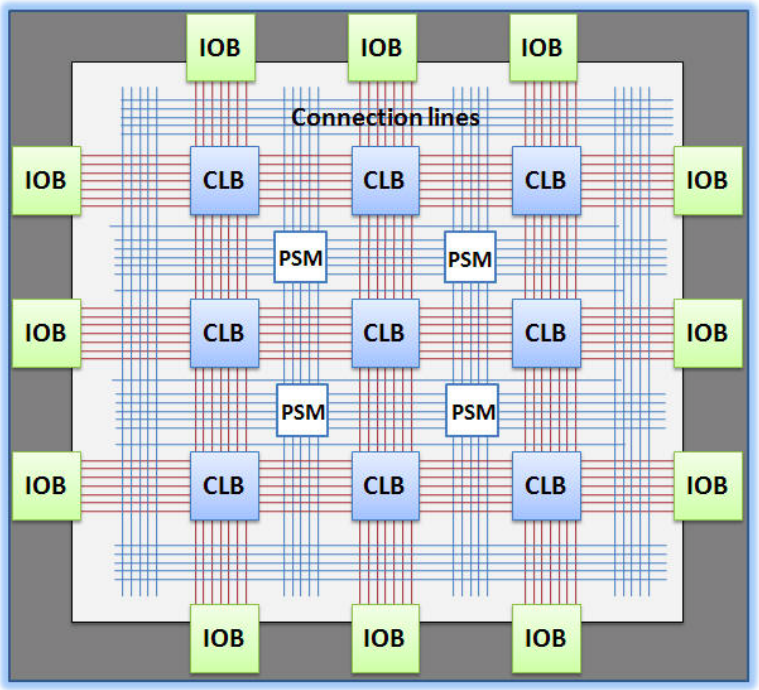


FIGURE 2.1: FPGA
[9]

2.4 Présentation de la PYNQ

PYNQ est un projet open source de Xilinx permettant de faciliter le design des systèmes embarqués avec la carte Xilinx Zynq.
L'utilisation du langage python et de ses librairies permet d'exploiter les bénéfices de la logique programmable et des microprocesseurs dans la Zynq pour créer des applications de haute performance.
La PYNQ est la première carte Zynq à supporter le langage de programmation Python.
En matière de matériel, PYNQ intègre un Xilinx Zynq Z-7020 SoC intégré, 512 Mo de DDR SDRAM, HMDI In / Out, Entrée et sortie audio, deux ports PMOD et prise en charge de l'en-tête d'interface Arduino. La carte peut être configurée depuis la carte

SD (Secure Digital) ou QSPI (Quad Serial Peripheral Interface).
PYNQ = (Python + Zynq) Dev Board.

Un noyau Linux gère le Zynq SoC avec un package spécifique qui prend en charge toutes les fonctionnalités de PYNQ. En utilisant ce paquet, il est possible de placer des Overlays matérielles (en réalité des fichiers binaires développés dans Vivado) dans la logique programmable du Zynq.
Le Zynq intègre un processeur multi-coeurs (Dual-core ARM® Cortex®-A9) et un Field Programmable Gate Array (FPGA) dans un seul circuit intégré.
Les circuits logiques programmables sont présentés sous forme de bibliothèques matérielles appelées Overlays. Ces Overlays sont analogues aux bibliothèques de logiciels, on sélectionne l'Overlay qui correspond le mieux à l'application visée. On peut accéder à l'Overlay via une interface de programmation d'application (API). La création d'une nouvelle superposition nécessite la conception de circuits logiques programmables. Les Overlays, comme les bibliothèques de logiciels, sont conçues pour être configurables et réutilisées aussi souvent que possible dans d'innombrables applications.
La carte Xilinx Zynq All Programmable est donc un SOC (System On Chip) basé sur un processeur ARM Cortex-A9 double cœur (appelé Processing System ou PS). Une puce Zynq comprend également une partie FPGA (appelée programmable logic ou PL). Le sous-système ARM SoC comprend également un certain nombre de périphériques dédiés (contrôleurs de mémoire, USB, UART, IIC, SPI, etc.).
Sur la carte Pynq-Z1, le contrôleur de mémoire DDR, Ethernet, USB, SD, UART est connecté sur le tableau.

La partie FPGA est reconfigurable et peut être utilisé pour implémenter des fonctions de haute performance dans le matériel. Cependant, le design FPGA est une tâche spécialisée qui requiert des connaissances et une expertise approfondies en matière d'ingénierie matérielle. Les Overlays, ou les bibliothèques de matériel, sont des conceptions FPGA programmables / configurables qui étendent l'application utilisateur à partir du système de traitement de Zynq dans la logique programmable. Les Overlays peuvent être utilisées pour accélérer une application logicielle ou pour personnaliser la plateforme matérielle pour une application particulière.

Par exemple, le traitement d'image est une application typique où les FPGA peuvent fournir une accélération. On peut utiliser un Overlay de manière similaire à une bibliothèque de logiciels pour exécuter certaines des fonctions de traitement d'image (par exemple détection de bordure) sur la partie FPGA. Les Overlays peuvent être chargés sur le FPGA de façon dynamique, au besoin, tout comme une bibliothèque de logiciels à partir de Python sur demande.
Sur les figures 2.2 et 2.3, la carte de développement PYNQ est illustrée.

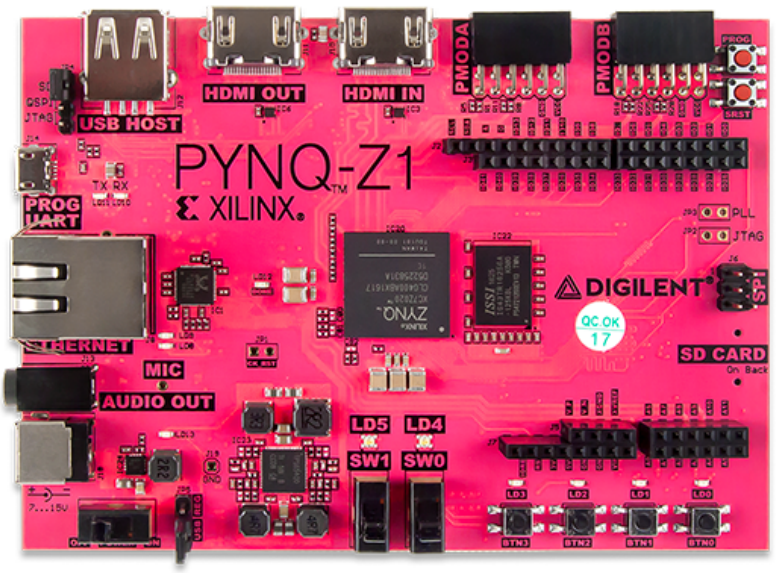


FIGURE 2.2: PYNQ
[10]

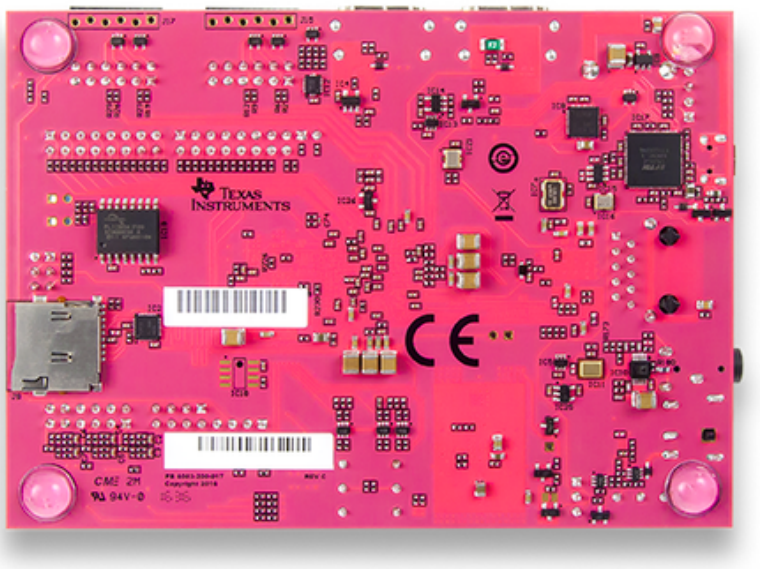


FIGURE 2.3: PYNQ
[10]

Processor : Dual-Core ARM® Cortex®-A9
FPGA : 1.3 M reconfigurable gates
Memory : 512MB DDR3 / FLASH

Storage : Micro SD card slot
Video : HDMI In and HDMI Out
Audio : Mic in, Line Out
Network : 10/100/1000 Ethernet
Expansion : USB Host connected to ARM PS
Interfaces : 1x Arduino Header, 2x Pmod (49 GPIO)
GPIO : 16 GPIO (65 in total with Arduino and Pmods)
Other I/O : 6x User LEDs, 4x Pushbuttons, 2x Switches
Dimensions : 3.44" x 4.81" (87mm x 122mm)

2.4.1 Les avantages de la PYNQ

- l'exécution matérielle parallèle ;
- le traitement des vidéos à grande vitesse ;
- les algorithmes d'accélération matérielle ;
- le traitement de signal en temps réel ;
- une bande passante élevé pour les entrées/sorties ;
- une faible latence de contrôle .

2.4.2 JUPYTER notebook

Un serveur portable Jupyter tourne dans le noyau Linux qui tourne sur le Zynq SoC de PYNQ. Cette interface est utilisée pour développer les applications PYNQ. Le notebook Jupyter et les Overlays sont au cœur de la méthodologie de développement PYNQ.
Le notebook JUPYTER est donc une application WEB qui regroupe un outil permettant de créer des documents multimédia intégrant du texte, des formules mathématiques, des graphiques, des images, voire des animations et des vidéos et une interface qui permet d'exécuter du code informatique.
Pour l'exécution du code informatique, Jupyter s'appuie sur des programmes indépendants capable d'interpréter le langage dans lequel est écrit ce code. Dans la terminologie de Jupyter ces programmes sont appelés des noyaux (kernel).
Les documents Jupyter sont appelés des notebooks. Ces notebooks ont l'extension .ipynb.
Le PYNQ-Z1 peut être facilement programmée dans Jupyter Notebook en utilisant Python. Ainsi, l'utilisation bibliothèques matérielles et de superpositions sur la logique programmable est possible. Les bibliothèques matérielles ou les superpositions peuvent accélérer le fonctionnement du PYNQ-Z1 et personnaliser la plateforme matérielle et les interfaces.

2.4.3 Vivado

Vivado est un logiciel produit par Xilinx pour la synthèse et l'analyse des design HDL, remplaçant Xilinx ISE avec des fonctionnalités supplémentaires pour le système sur un développement de puce et une synthèse de haut niveau. Vivado représente une réécriture et une réflexion sur l'ensemble du flux de conception (par rapport à ISE) .

Vivado inclut un simulateur de logique intégré, il a été décrit comme un «outil EDA complet à la fine pointe de la technologie»[11].

Vivado permet de synthétiser (compiler) les conceptions, d'effectuer des analyses de temps, d'examiner les diagrammes de RTL, de simuler la réaction d'un design à différents stimuli et de configurer le périphérique cible avec le programmeur. Vivado est un environnement de conception pour les produits FPGA de Xilinx et est étroitement couplé à l'architecture de ses puces.

2.5 Les Overlays

Qu'est-ce qu'un Overlay ? L'Overlay est un design qui est chargé dans la logique programmable Zynq SoC (PL). L'Overlay peut être conçue pour accélérer une fonction dans la logique programmable ou fournir une capacité d'interfaçage en utilisant le PL, c'est ce qui donne à Pynq ses capacités uniques.

Ce qui est important dans un Overlay, c'est qu'il n'y a pas de processus de synthèse de haut niveau Python-à-PL impliqué. Au lieu de cela, l'Overlay est développé en utilisant l'une des méthodologies de conception standard de Xilinx (SDSoC, Vivado ou Vivado HLS). Une fois le fichier bit crée, il suffit de l'intégrer à l'architecture Pynq et d'établir les paramètres requis pour communiquer avec Python.

Il sera alors facile de s'intégrer à l'environnement Python à l'aide du fichier bit et d'autres fichiers fournis avec la construction Vivado. Dans ce projet, cela est réalisé grâce à la classe Python MMIO, ce qui permet d'interagir avec des designs dans le PL grâce à des lectures et des écritures mémorisées.

2.5.1 Création d'overlays

Les overlays sont analogues aux bibliothèques de logiciels. On peut télécharger des Overlays dans le PL (Programmable Logic) Zynq à l'exécution du code python pour fournir les fonctionnalités requises par l'application logicielle.

Un Overlay est une classe de conception logique programmable. Les conceptions logiques programmables sont généralement très optimisées pour une tâche spécifique. Cependant, les Overlays sont conçus pour être configurables et réutilisables

pour un large éventail d'applications. Un Overlay PYNQ aura une interface Python, permettant son utilisation comme tout autre paquet Python.

Un Overlay se compose de deux parties principales ; La conception de la logique programmable (PL) et l'API (Application Programming Interface) Python.

Le logiciel Xilinx Vivado est utilisé pour créer la conception PL. Cela générera un fichier bitstream ou binaire (fichier .bit) qui est utilisé pour programmer le Zynq PL.

Il existe des différences entre le processus de conception Zynq standard et la conception d'Overlays pour PYNQ. Un projet Vivado pour une conception Zynq se compose de deux parties ; La conception PL et les paramètres de configuration PS. La configuration PS comprend des réglages pour les horloges système, y compris les horloges utilisées dans le PL.
L'image PYNQ utilisée pour démarrer la carte configure le Zynq PS au démarrage. Les Overlays sont téléchargés selon le besoin et ne reconfigurent pas le Zynq PS. Cela signifie que durant la conception d'overlays il faudra s'assurer que les paramètres de PS dans le projet Vivado correspondent aux paramètres d'image PYNQ.

Les paramètres suivants doivent être utilisés pour un nouveau projet d'Overlays de Vivado :

Paramètres du projet Vivado :

- Dispositif cible : xc7z020clg400-1

- Configuration de l'horloge PL :

  - FCLK_CLK0 : 100.00MHz ;
  - FCLK_CLK1 : 142.86MHz ;
  - FCLK_CLK2 : 200.00MHz ;
  - FCLK_CLK3 : 166.67MHz.

Il est meilleur de commencer par une conception d'overlay existante pour s'assurer que les paramètres PS sont corrects. Durant ce projet, l'overlay existant utilisé et le "base overlay"

Le fichier tcl doit également être exporté avec le flux de bits. Cela permet d'analyser les informations sur l'Overlay dans Python (par exemple, la liste des IP dans l'Overlay).

Le paquet Overlay génère un dictionnaire appelé ip_dict contenant les noms d'IP dans un Overlay spécifique (par exemple base.bit). Le dictionnaire peut être utilisé pour référencer une adresse IP par son nom dans le code Python plutôt que par une

adresse codée. Il peut également vérifier l'IP disponible dans un Overlay.

2.5.2 Interfacer un Overlay

2.5.2.1 MMIO

PYNQ comprend la classe MMIO en Python pour simplifier la communication entre le Zynq PS et le PL. Une fois que l'Overlay a été créé et que la carte mémoire est connue, le MMIO peut être utilisé pour accéder aux emplacements mappés en mémoire dans le PL en python sur le notebook.

2.5.2.2 GPIO

Les GPIO (General Purpose Input Output) entre le Zynq PS et PL peuvent être utilisés par le code Python comme interface de contrôle aux Overlays. Les informations sur un GPIO sont conservées dans le dictionnaire GPIO d'un Overlay similaire à l'ip_dict.

2.5.2.3 CFFI

CFFI (C Foreign Function Interface) fournit un moyen simple d'interface avec le code C à partir de Python. Le paquet CFFI est préinstallé dans l'image PYNQ. Il prend en charge un mode de compatibilité ABI (Application Binary Interface) en ligne, qui permet de charger et d'exécuter dynamiquement des fonctions à partir de modules exécutables et un mode API qui permet de créer des modules d'extension C.
Les fonctions C dans une bibliothèque partagée peuvent être appelées à partir de Python à l'aide de l'interface C Foreign Function (CFFI). La bibliothèque partagée peut être compilée en ligne en utilisant le CFFI de Python, ou elle peut être compilée hors ligne.

2.5.3 Utilisation d'un Overlay

Le PL peut être reconfiguré dynamiquement avec de nouveaux Overlays lorsque le système fonctionne.

2.5.4 L'Overlay "base"

L'Overlay de base est l'Overlay par défaut inclus avec l'image PYNQ-Z1 et est automatiquement chargée dans la Logique programmable lorsque le système démarre.

Pour créer un Overlay PYNQ, il est recommendé d'utiliser l'Overlay base existant de la conception de Vivado en tant que point de départ. La raison derrière cela est que l'Overlay fournie prédéfinit toutes les configurations requises pour le PS de Zynq SoC et définit l'interface PS-PL.

La première chose à explorer est l'interconnexion implémentée à l'aide des interfaces AXI. Au niveau le plus simple, les interconnexions AXI haute performance de Zynq SoC sont utilisées pour les interfaces nécessitant que la DMA transfère des données vers ou à partir de la mémoire DDR. Pour l'Overlay PYNQ, ces interconnexions sont utilisées pour le trace buffer pour les ports Arduino et PMOD et le flux vidéo.

Les interfaces Master General-Purpose AXI sont utilisées pour contrôler et configurer les blocs restants dans la conception. Plus intéressant, ces ports sont utilisés pour configurer les BRAM MicroBlaze à la volée. Le processeur MicroBlaze communique avec le port de protection PMODS et Arduino, fournissant une interface transparente.

Cet Overlay comprend le matériel illustré sur la figure 2.4 et qui est :

- HDMI In ;
- HDMI Out ;
- Mic in ;
- Audio out ;
- User LEDs, Switches, Pushbuttons ;
- 2x PMOD IOP ;
- Arduino IOP ;
- Trace buffer.

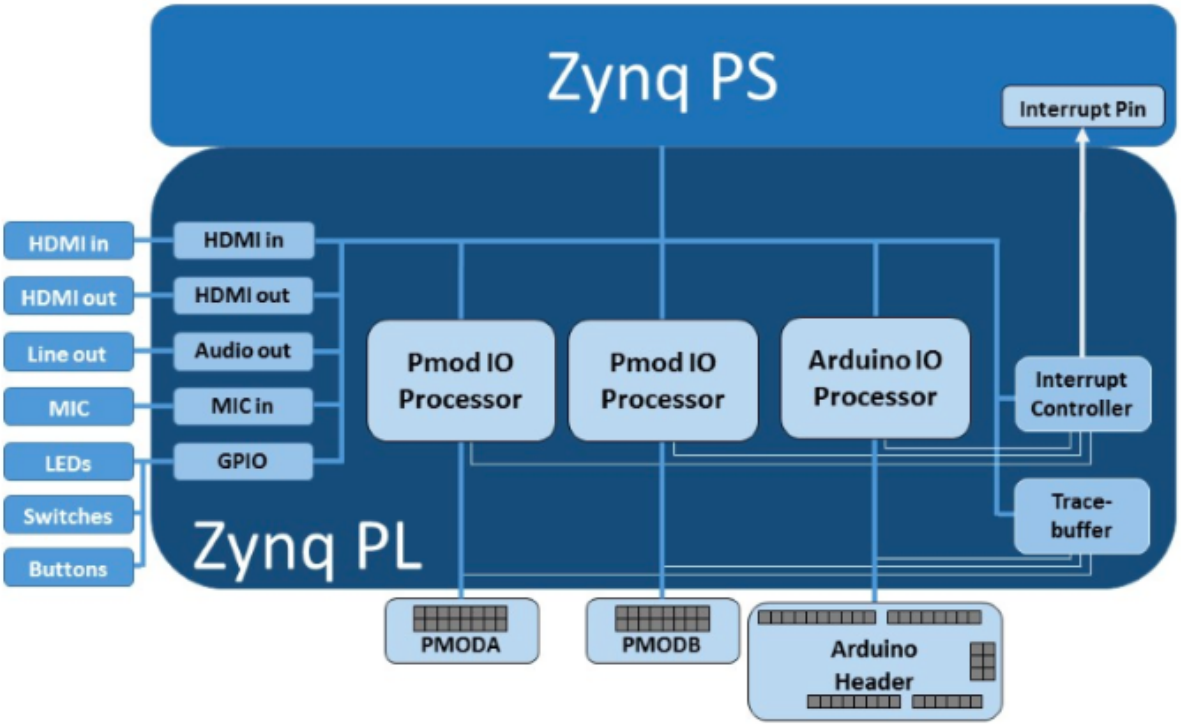


FIGURE 2.4: Overlay base
[12]

2.5.4.1 HDMI

Les contrôleurs HDMI sont connectés directement aux interfaces HDMI. Il n'y a pas de circuit HDMI externe.

Les deux interfaces HDMI sont connectées à la mémoire DDR. La vidéo peut être transmise de l'entrée HDMI à la mémoire, et de la mémoire vers la sortie HDMI. Cela permet de traiter des données vidéo de python, ou d'écrire une image ou un flux vidéo de Python vers la sortie HDMI.

HDMI In

L'interface HDMI dans la carte de développement Pynq peut capturer des résolutions HDMI standard. Une fois qu'une source HDMI a été connectée et que le contrôleur HDMI a démarré, il détectera automatiquement les données entrantes. La résolution peut être lue à partir de la classe HDMI Python, et les données d'image peuvent être transmises à la mémoire DDR.

HDMI Out

L'interface HDMI out prend en charge les résolutions suivantes :

- 640x480 ;

- 800x600;
- 1024x768;
- 1280x1024;
- 1920x1080.

Les données peuvent être transmises à partir de la mémoire DDR vers la sortie HDMI. L'instance Python HDMI Out contient des blocs de trame pour permettre l'affichage en douceur des données vidéo.

2.5.4.2 Mic in

La carte PYNQ-Z1 possède un microphone intégré et est directement connecté aux broches PL Zynq. Cela signifie que le tableau n'a pas de codec audio. Le Mic génère des données audio au format PDM.

2.5.4.3 Audio out

L'entrée de Audio out est connectée à une prise audio standard de 3,5 mm sur la carte. La sortie audio est mono piloté par PWM (Pulse Width Modulation).

2.5.4.4 User IO

La carte PYNQ-Z1 comprend deux LEDs tri colors, 2 commutateurs, 4 boutons poussoirs et 4 LEDs individuelles. Dans l'Overlay de base, User IO est directement connecté au PS et peut être contrôlé directement à partir de Python.

2.5.4.5 IOPs

Les IOP sont des sous-systèmes de processeurs IO dédiés qui permettent aux périphériques avec différentes normes d'IO d'être connectés au système sur demande. Cela permet à un programmeur de logiciel d'utiliser une large gamme de périphériques avec différentes interfaces et protocoles. Le même Overlay peut être utilisé pour supporter différents périphériques.

Il existe deux types d'IOP : Pmod et Arduino. Les deux types d'IOP ont une architecture similaire, mais ont différentes configurations d'IP pour se connecter à des périphériques compatibles.

2.5.4.6 Trace buffer

Un Trace Buffer est disponible et peut être utilisé pour capturer des données sur les interfaces Pmod et Arduino pour le déboggage. Le Trace buffer est directement connecté au DDR. Cela permet aux données des interfaces d'être transmises à la mémoire DDR pour analyse dans Python.

Les données de traçage peuvent être affichées sous forme de formes d'onde décodées dans un notebook Jupyter.

2.6 Interfaces et bus de communication de la PYNQ

2.6.1 Introduction au protocole AXI

Le protocole AXI :
- convient à des modèles à large bande passante et à faible latence ;
- fournit un fonctionnement à haute fréquence sans utiliser de ponts complexes ;
- répond aux exigences d'interface d'une large gamme de composants ;
- convient aux contrôleurs de mémoire avec une latence d'accès initiale élevée ;
- offre une flexibilité dans la mise en œuvre des architectures d'interconnexion ;
- est compatible avec les interfaces AHB et APB existantes.

Les principales caractéristiques du protocole AXI sont les suivantes :
- séparer les phases d'adresse/contrôle et de données ;
- prise en charge des transferts de données non alignés, en utilisant des stroboscopes d'octets ;
- utilise des transactions en rafale avec seulement l'adresse de départ émise ;
- séparer les canaux de données de lecture et d'écriture, qui peuvent fournir un accès direct à la mémoire (DMA) à faible coût ;
- permet l'envoi de plusieurs adresses ;
- permet d'ajouter facilement des registres pour assurer la fermeture du chronométrage.

Le protocole AXI comprend les extensions optionnelles qui couvrent la signalisation pour un fonctionnement à faible puissance.

2.6.2 Architecture de l'AXI

Le protocole AXI est basé sur l'éclatement et définit les canaux de transaction indépendants suivants :
- lire l'adresse ;
- lire les données ;
- écrire une adresse ;
- écrire des données ;
- rédiger une réponse.

Une voie d'adresse contient des informations de contrôle qui décrivent la nature des données à transférer. Les données sont transférées entre maître et esclave en utilisant soit :
- Un canal de données d'écriture pour transférer des données du maître à l'esclave. Dans une transaction d'écriture, l'esclave utilise le canal de réponse en

écriture pour signaler l'achèvement du transfert au maître.

- Un canal de données de lecture pour transférer des données de l'esclave au maître.

Le protocole AXI :

- permet de délivrer des informations d'adresse avant le transfert de données réel ;
- prend en charge plusieurs opérations ;
- prend en charge l'achèvement des transactions hors-service.

Les figures 2.6 et 2.5, illustrent l'architecture des canaux de lecture et d'écriture entre maître et esclave.

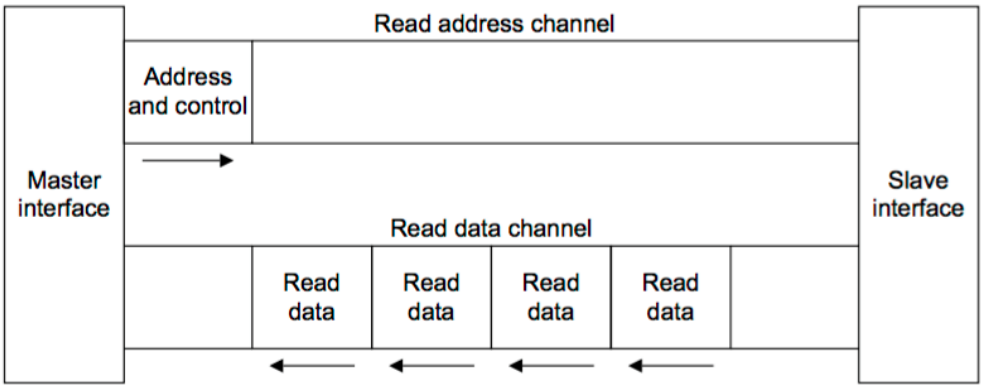


FIGURE 2.5: Architecture des canaux de lectures
[13]

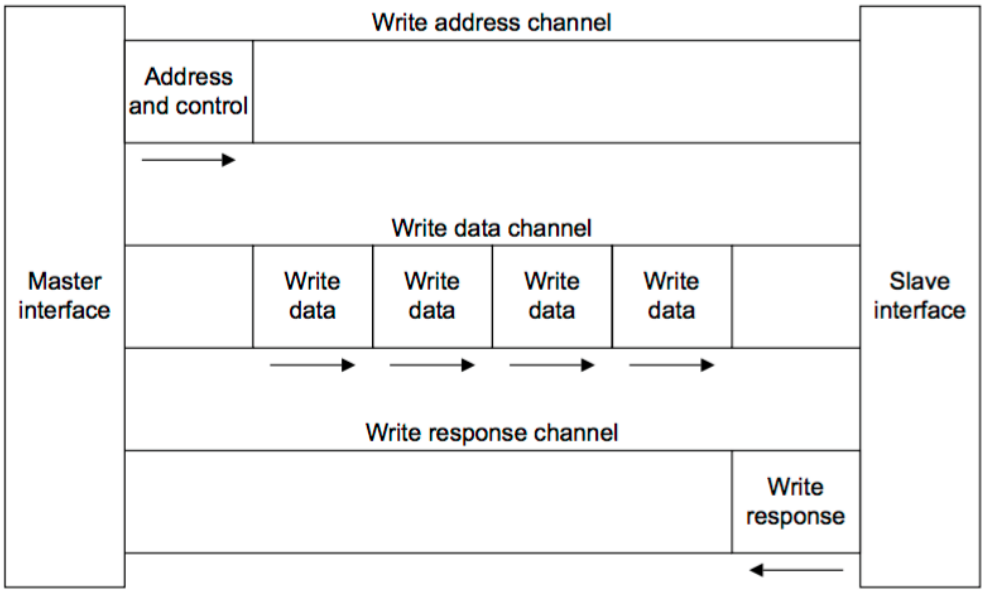


FIGURE 2.6: Architecture des canaux d'écritures
[13]

Chacun des canaux indépendants se compose d'un ensemble de signaux d'information et des signaux VALID et READY qui fournissent un mécanisme de poignée de main bidirectionnel.
La source d'information utilise le signal VALID pour montrer quand une adresse valide, des données ou des informations de contrôle sont disponibles sur le canal. La destination utilise le signal READY pour montrer quand elle peut accepter les informations. Le canal de données de lecture et le canal de données d'écriture comprennent également un signal LAST pour indiquer le transfert de l'élément de données final dans une transaction.

Les opérations de lecture et d'écriture ont chacune leur propre canal d'adresse. Le canal d'adresse approprié contient toutes les informations d'adresse et de contrôle requises pour une transaction.

Le canal de données de lecture transmet à la fois les données de lecture et les informations de réponse de lecture de l'esclave au maître et comprend :

- le bus de données, qui peut être 8, 16, 32, 64, 128, 256, 512 ou 1024 bits de largeur ;
- un signal de réponse en lecture indiquant l'état d'achèvement de la transaction de lecture.

Le canal de données d'écriture porte les données d'écriture du maître à l'esclave et comprend :

- le bus de données, qui peut être 8, 16, 32, 64, 128, 256, 512 ou 1024 bits de

largeur
- un signal stroboscopique de voie d'octet pour chaque huit bits de données indiquant quels sont les octets des données valides.

L'écriture des informations sur les chaînes de données est toujours traitée comme tampon, de sorte que le maître puisse effectuer des transactions d'écriture sans que l'esclave connaisse les transactions d'écriture précédentes.
Un esclave utilise le canal de réponse d'écriture pour répondre aux transactions d'écriture. Toutes les transactions d'écriture nécessitent une signalisation d'achèvement sur le canal de réponse en écriture l'achèvement est signalé uniquement pour une transaction complète, pas pour chaque transfert de données dans une transaction.

Un système typique se compose d'un certain nombre d'appareils maître et esclave connectés entre eux par une forme d'interconnexion, on peut voir tout cela sur la figure 2.7.

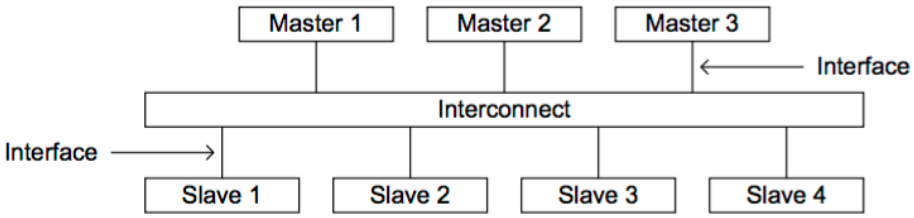


FIGURE 2.7: Interfaces et interconnexions
[13]

Le protocole AXI fournit une définition d'interface unique pour les interfaces :
- entre un maître et l'interconnexion ;
- entre un esclave et l'interconnexion ;
- entre un maître et un esclave.

Cette définition d'interface prend en charge une variété d'implémentations d'interconnexion différentes.

2.6.3 Communication entre Maître et esclave

Le Maître change la valeur du signal VALID et la maintient à '1' lorsque les données sont disponibles.
L'esclave change la valeur du signal READY s'il est capable d'accepter des données.
Le signal DATA ainsi que d'autres signaux sont transférés lorsque VALID et READY sont simultanément à la valeur 1 .
Le Maître envoie les données suivantes et autres signaux ou alors remets la valeur de VALID à zéro.

Alors l'esclave remet READY à zéro dès qu'il ne peut plus accepter les données.
Un schéma illustratif **??** et proposé, ainsi que les chronogrammes **??**, pour mieux comprendre ce protocole.

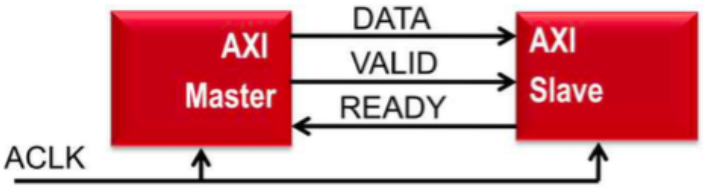


FIGURE 2.8: Communication entre maître et esclave
[14]

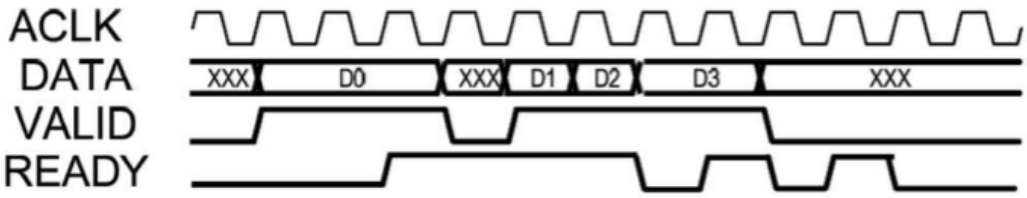


FIGURE 2.9: Signaux utilisés dans la communication entre maître et esclave
[14]

2.6.4 Les protocoles de communication AXI

Il existe 3 interfaces AXI différentes, la figure 2.10 l'illustre.

2.6.4.1 AXI4 full

Les fonctionnalités clé de l'AXI4-full sont :
- toutes les transactions ont une longueur de rafale allant jusqu'à 256
- AXI4-Lite prend en charge une largeur de bus de données de 32 bits jusqu'à 1024 bits.

2.6.4.2 AXI4 lite

Le protocole AXI comprend la spécification AXI4-Lite, un sous-ensemble d'AXI4 pour la communication avec un contrôle plus simple ne nécessitant pas la fonctionnalité complète d'AXI4.
Les fonctionnalités clé de l'AXI4-Lite sont :
- toutes les transactions ont une longueur de rafale de 1
- tous les accès aux données utilisent toute la largeur du bus de données
- AXI4-Lite prend en charge une largeur de bus de données de 32 bits ou 64 bits.

- tous les accès ne sont pas modifiables,
- Les accès exclusifs ne sont pas pris en charge.

2.6.4.3 AXI4 stream

Les fonctionnalités clé de l'AXI4-stream sont :
- toutes les transactions ont une longueur de rafale illimitée
- AXI4-Lite prend en charge une largeur de bus de données de n'importe quel nombre de bits.

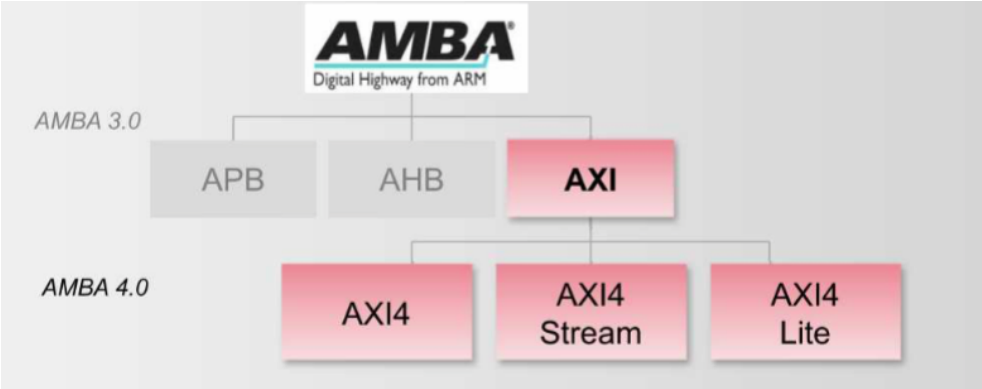


FIGURE 2.10: Les protocoles de communication AXI
[14]

	AXI4-FULL	AXI4-Lite	AXI4-Stream
Dédié à	Systèmes haute performance et mappés en mémoire	Interfaces de type registre	IP avec adresse de départ uniquement (transactions en rafale avec seulement l'adresse de départ émise)
Flot de données	jusqu'à 256	1	illimité
Largeur des données	32 jusqu'à 1024 bits	32 ou 64 bits	n'importe quel nombre de bits
Exemples d'applications	mémoire	logique de contrôle	DSP, vidéo, communication

TABLE 2.1: Les protocoles de communication AXI

2.7 Conclusion

Dans ce chapitre, nous avons présenté la carte de développement Pynq (Python Productivity for Zynq), les avantages de cette carte de développement, les protocoles de communications entre la partie PS et PL de la Pynq. Nous avons constaté que cette carte permet un co-design hardware (par la créaction d'Overlay) et software (grâce au codes écrits en python sur le serveur portable Jupyter).
Dans le chapitre suivant, nous allons proposer des modèles de réseaux de neurones artificiels répondant au problème de la reconnaissance de chiffres manuscrits afin d'intégrer la solution sur la carte de développement Pynq.

Chapitre 3

Intégration de réseaux de neurones sur PYNQ

3.1 Introduction

Dans ce chapitre, nous allons aborder le problème de la classification de chiffres manuscrits grâce aux réseaux de neurones.
Après avoir vu dans le chapitre 1 une introduction aux réseaux de neurones, et dans le chapitre 2 une présentation de notre carte de développement Pynq, nous pouvons mieux comprendre comment aborder le problème de classification grâce aux réseaux de neurones et comment intégrer la solution sur la carte de développement.

3.2 Réseaux de neurones définis

Durant ce projet, diverses structures de réseaux de neurones ont été testés, les codes python se trouvent en annexe et les précisions obtenues après exécution de ces codes sont résumées dans le tableau suivant :

Réseau de neurones	Précision
Sans couches cachées	92.36%
Avec une seule couche cachée	97.63%
Avec deux couches cachées	98.35%
Convolutif	99.18%

TABLE 3.1: Différents réseaux de neurones et leur précision

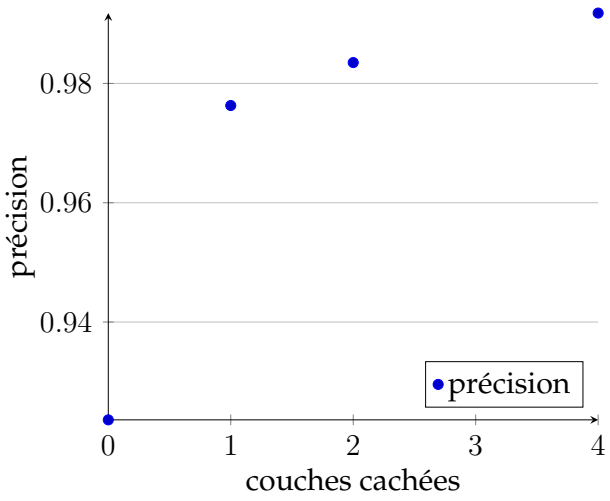


FIGURE 3.1: Variation de la précision en fonction du nombre de couches cachées

Pour le réseau de neurones sans couche cachée on voit bien que la précision est de 92.36% .
Ceci est un assez bon résultat mais on peut arriver à un meilleur résultat en introduisant une couche cachée entre la couche d'entrée et la couche de sortie. Comme on peut le voir la précision augmente est atteint 97.63%. Ce résultat est largement meilleur que le précédant mais on peut voir que la précision augmente en utilisant d'autres fonctions d'activation et en utilisant la bibliothèque Keras, on arrive à une précision de 98.3%.
Encore une fois ce résultat est meilleur que le précédent mais pourrait être encore meilleur en utilisant un certain type de réseau de neurones plus complexes "les réseaux de neurones convolutifs".

3.3 Réseau de neurones sans couches cachées

Dans cette partie, nous allons voir étape par étape comment construire et entrainer le réseau de neurones sans couches cachées.

3.3.1 Importation des bibliothèques

Nous commençons impérativement par importer les bibliothèques Python dont nous aurons besoin. Dans cet exemple, nous aurons besoin de TensorFlow.
TensorFlow est une bibliothèque de logiciels open source pour l'apprentissage par machine à travers une gamme de tâches. Elle a été développée par Google pour répondre à leurs besoins de systèmes capables de construire et de former des réseaux de neurones pour détecter et déchiffrer des modèles et des corrélations analogues à

l'apprentissage et au raisonnement que les humains utilisent [15].

Pour faire un calcul numérique efficace dans Python, on peut utiliser des bibliothèques comme NumPy qui font des opérations coûteuses telles que la multiplication matricielle en dehors de Python, en utilisant un code hautement efficace implanté dans un autre langage. Mais, cette solution n'est pas idéale car elle occasionne beaucoup de surcharge pour revenir à Python après chaque opération.

TensorFlow fait également ses calculs lourds à l'extérieur de Python, mais cela prend des mesures supplémentaires pour éviter cette surcharge. Au lieu de gérer une seule opération coûteuse indépendamment de Python. Pour cela, nous utiliserons Tensorflow pour cet exemple.

3.3.2 Importation de la base de données

Nous devons importer la base de données MNIST qui va servir à entrainer et tester le réseau de neurone construit.

3.3.3 Création du réseau de neurones

En utilisant la bibliothèque TensorFlow, on va définir les éléments constituant notre réseau de neurone :

- une matrice X d'entrée de taille [None, 784] qui représentera couche d'entrée qui recevra les données MNIST (données pour entrainer, valider et tester le réseau).
  - None : cette taille changera au cours de l'exécution de notre programme, elle passera de 55000 à 10000 à 5000 selon qu'on entraine, valide ou teste notre réseau de neurones. C'est pour cela qu'on a utilisé None et non une taille définit.
  - 784 : la taille de l'image 28 X 28.
- une matrice W de taille [784,10] représentant les poids du réseau de neurones,
- un vecteur B de taille [10] représentant les biais du réseau de neurones
- un vecteur Y de taille [10] représentant les sorties du réseau de neurones. Un vecteur codé en one hot, permettant de déterminer quel chiffre est représenté sur l'image fourni en entrée, en mettant à un le bit représentant la position de ce chiffre.
  $Y = XW + B$ (3.1)

3.3.4 Fonctions d'activation

Nous savons que chaque image dans MNIST est un chiffre manuscrit entre zéro et neuf. Donc il n'y a que dix possibilités pour une image donnée. Nous désirons être capable de regarder une image et donner les probabilités pour chaque chiffre.

Par exemple, notre modèle pourrait avoir en entrée une image représentant le chiffre neuf et être à 80% sûr que c'est un neuf, mais donner 5% de chances qu'il s'agisse d'un huit et une petite probabilité que ça soit tous les autres car il n'est pas sûr à 100%.

Il s'agit d'un cas classique où une régression softmax est un modèle naturel et simple car Softmax nous donne une liste de valeurs entre 0 et 1 qui quand ils s'ajoutent donnent 1. La fonction d'entrainement sera donc la fonction softmax, la dernière couche sera une couche softmax.

Une régression softmax comporte deux étapes : d'abord, nous ajoutons la preuve que notre contribution se trouve dans certaines classes, puis nous convertissons cette évidence en probabilités.

$$softmax = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{3.2}$$

Pour entrainer notre modèle, nous devons définir ce que cela signifie pour un modèle d'être mauvais. Cela est appelé le coût, ou la perte, et cela représente la distance de notre modèle par rapport au résultat souhaité. Nous essayons de minimiser cette erreur, et plus la marge d'erreur est faible, meilleure est notre modèle.

La fonction qu'on va utiliser est appelée «entropie croisée» défini comme suit :

$$H_{y'}(y) = -\sum_i y_{i'} \log(y_i) \tag{3.3}$$

Où :
- y est notre distribution de probabilité ;
- y' est la vraie distribution (le vecteur représentant l'étiquette fourni par la base de donnée

Maintenant que nous savons ce que nous voulons que notre modèle fasse, On va faire utiliser TensorFlow pour l'obliger à le faire. TensorFlow peut utiliser automatiquement l'algorithme de rétropropagation pour déterminer efficacement comment les variables de poids affectent la perte que nous voulons minimiser. Ensuite, il va appliquer notre choix d'algorithme d'optimisation pour modifier les variables et réduire la perte.

Dans ce cas, nous demandons à TensorFlow de minimiser cross_entropy à l'aide de l'algorithme de descente en gradient avec un taux d'apprentissage de 0,5. La descente en gradient est une procédure simple, où TensorFlow déplace simplement chaque variable dans la direction qui réduit le coût. Mais TensorFlow fournit également de nombreux autres algorithmes d'optimisation.

Ce que TensorFlow fait réellement ici, consiste à ajouter de nouvelles opérations au graphique qui implémente la répartition en arrière et la descente en gradient. Ensuite, il ramène à une seule opération qui, lorsqu'il est exécuté, fait une étape de formation en descente de gradient, modifiant légèrement les variables pour réduire la perte.

3.3.5 Evaluation du modèle

L'étiquette correcte est fournie par la base de données. Nous devons vérifier si le résultat donné par notre modèle correspond au résultat fourni par la base de données.
Nous allons utiliser la fonction argmax, cette fonction donne l'index de l'entrée la plus élevée dans un vecteur. Il nous permet alors de passer d'un vecteur dont les valeurs sont des probabilités à un vecteur one hot que nous comparerons avec l'étiquette correcte pour évaluer le modèle.

3.4 Réseau de neurones avec deux couches cachées

Pour ce réseau de neurones, nous avons utilisé deux couches cachées et une bibliothèque différente de celles utilisées dans les autres réseaux de neurones, la bibliothèque Keras. Le code Python se trouve en annexe 1.2.2.

3.4.1 Réseau de neurones utilisé

Le réseau de neurones utilisé peut comme on l'a vu précédemment être constitué d'une, de plusieurs ou d'aucune couche cachée selon la précision voulu.
Pour cet exemple, nous allons choisir de travailler avec deux couches cachées et 512 neurones dans chacune d'elles et c'est cet exemple qui a été implémentée sur la PYNQ 1.2.3.
La couche d'entrée reçoit les pixels de l'image, il nous faut alors 784 neurones, ce chiffre est fixe, on ne peut donc pas jouer sur le nombre de neurones de la couche d'entrée.
Le nombre de neurones de la couche de sortie est lui aussi fixé à 10, représentant le nombre de chiffres que l'on peut avoir soit [0 1 2 3 4 5 6 7 8 9].
La seule couche dont le nombre de neurones n'est pas fixé est la couche cachée, on peut alors jouer sur ce nombre pour tenter d'avoir le maximum de précision en sortie. On peut aussi jouer sur le nombre de couches cachées.

3.4.2 Bibliothèque Keras

Keras est une API (Application Programming Interface) de réseaux de neurones de haut niveau, écrite en Python et capable de fonctionner sur TensorFlow ou Theano. Elle a été développée en mettant l'accent sur l'expérimentation rapide. Être capable d'aller de l'idée à un résultat avec le moins de délai possible est la clé pour faire de bonnes recherches.
Nous allons utiliser Keras car nous avons besoin d'une bibliothèque d'apprentissage en profondeur qui permet de construire un prototype facile et rapide (par convivialité, modularité et extensibilité).

3.4.3 Importation de la base de données

Nous devons importer la base de données MNIST qui va servir à entrainer et tester le réseau de neurone construit.
Les commandes utilisées pour cela se trouvent dans le code Python 1.2.2.

3.4.4 Création du réseau de neurones

En utilisant la bibliothèque Keras avec backend TensorFlow, il est très simple de créer un réseau de neurones. Il suffit de définir le modèle de réseau de neurones que nous désirons créer, dans notre cas c'est un modèle séquentiel, puis il suffit d'ajouter les couches l'une après l'autre en précisant le nombre de neurones de chaque couche et la fonction d'activation que l'on désire appliquer à chaque couche.
Dans cet exemple, nous aurons deux couches cachées avec chacune 512 neurones.

3.4.5 Fonctions d'activation

Dans cet exemple, nous utilisons les fonctions d'activation ReLU et Softmax. Nous avons pu voir la fonction Softmax précédemment 3.3.4.
La fonction ReLU quant à elle est une fonction d'activation définit comme suit :

$$f(x) = max(0, x) \tag{3.4}$$

Où x est l'entrée d'un neurone.

Nous utilisons aussi la fonction "Dropout" qui est une technique permettant d'ignorer les neurones sélectionnés au hasard pendant l'entraînement. Ils sont "abandonnés" au hasard. Cela signifie que leur contribution à l'activation des neurones en aval est temporairement enlevée sur le passage en avant et aucune mise à jour de poids n'est appliquée au neurone sur le passage vers l'arrière.

À mesure que le réseau de neurones apprend, les poids des neurones s'installent dans leur contexte au sein du réseau. Les poids des neurones sont accordés pour des caractéristiques spécifiques offrant une certaine spécialisation. Les neurones voisins deviennent dépendant de cette spécialisation, qui, si elle est portée trop loin, peut entraîner un modèle fragile trop spécialisé dans les données de formation. Cette dépendance sur le contexte d'un neurone pendant l'entraînement est appelée à des co-adaptations complexes.

L'effet du "Dropout" est que le réseau devient moins sensible aux poids spécifiques des neurones. Cela entraîne à son tour un réseau qui est capable d'une meilleure généralisation et moins susceptible de se passer de la formation.

Une "époque" décrit le nombre de fois que l'algorithme voit l'ensemble des données en entier. Donc chaque fois que l'algorithme a vu tous les échantillons dans l'ensemble de données, une époque s'est terminée.

Une «itération» décrit le nombre de fois qu'un «lot» de données a traversé l'algorithme. Donc, chaque fois que nous passons un lot de données via le réseau de neurones, vous avez complété une "itération"

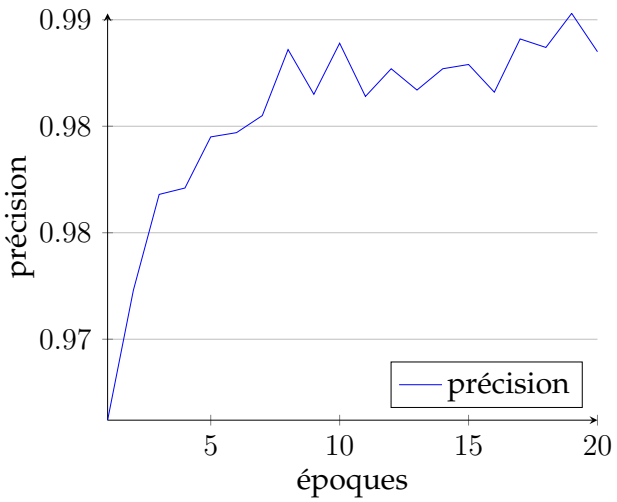


FIGURE 3.2: Variation de la précision au cours de l'entrainement du réseau à deux couches cachées

Dans le graphe **??**, on peut voir les variations de la précision en fonction du nombre d'époques. La précision augmente rapidement au début dû à l'apprentissage du réseau mais commence à fluctuer considérablement à partir de 8 époques. Le réseau de neurones durant son entrainement donne des valeurs aux poids et regarde si cela augmente sa précision ou la diminue puis change en fonction de ce résultat. Parfois ces hypothèses sont bonnes mais parfois elles ne le sont pas, ceci

explique les fluctuations de sa précision, le réseau apprend des bonnes et des mauvaises expériences, comme l'être humain.

3.4.6 Intégration

3.4.6.1 Enregistrement des paramètres

Avant de pouvoir implémenter notre réseau de neurones, il faut d'abord tirer les paramètres du réseau qu'on a entrainé précédemment. A la fin du dernier code python se trouvant en annexe 1.2.2, on voit la partie permettant d'enregistrer les paramètres du réseau de neurones entrainé dans un fichier compressé pour pouvoir les réutiliser par la suite pour l'intégration du réseau.

3.4.6.2 Intégration du réseau dans le Notebook

Sur la Pynq, on crée un nouveau Notebook dans lequel on va définir notre réseau de neurones avec le même nombre de couches et de neurones que celui dont on a tiré les poids. Par la suite, on va importer le fichier dans lequel on a enregistré nos poids et on va les injecter dans notre réseau.
Ainsi, notre modèle est construit, on peut lui injecter en entrée des images et observer les résultats qu'il nous donne.
On peut voir en annexe 1.2.3, l'exécution du notebook avec une exemple d'image en entrée.
On voit sur la figure 3.3, un schéma résumant le travail effectué sur le notebook.

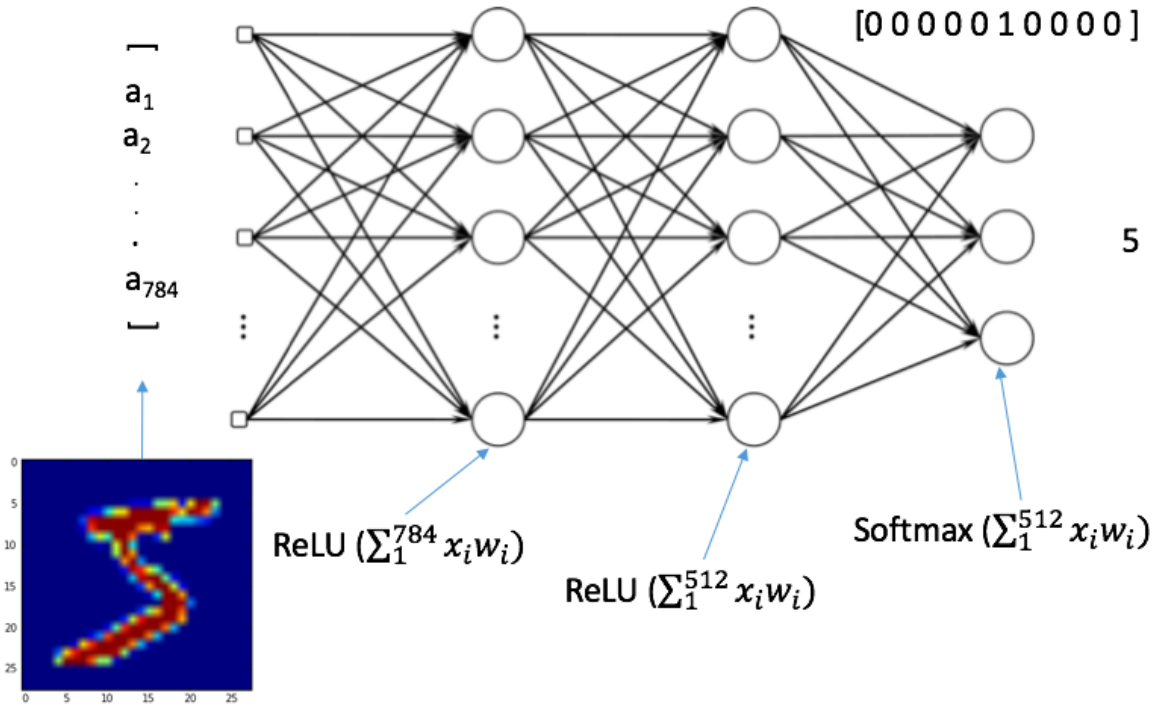


FIGURE 3.3: Schéma illustratif de l'intégration du réseau

3.5 Conclusion

Dans ce chapitre, nous avons vu étape par étape comment définir et entrainer un réseau de neurone, tirer ses paramètres après entrainement et les implémenter sur une carte de développement PYNQ.
Ce réseau de neurone défini sur la PYNQ arrive à reconnaitre les chiffres représentés sur les images qu'on lui fournit en entrée avec la même précision que le réseau dont les poids ont été tirés.
Maintenant que nous avons implémenté le réseau de neurones sur la partie FPGA de la Pynq, nous allons étudier les approches possibles nous permettant d'accélérer les calculs effectués durant le traitement d'images par le réseau de neurones dans le FPGA.

Chapitre 4

Accélération de réseaux de neurones

4.1 Introduction

Dans les chapitres précédents, nous avons vu comment classifier les chiffres manuscrits grâce aux réseaux de neurones et comment implémenter la solution sur la partie FPGA de la Pynq. Nous avons aussi présenté l'Overlay base et ses différents constituants.
Dans ce chapitre, nous allons voir comment créer de nouveaux Overlays à partir de l'Overlay base dans le but d'explorer les différentes solutions permettant d'accélérer un réseau de neurones.
Nous allons donc exposer les tests d'intégration réalisés afin d'optimiser les calculs effectués dans un réseau de neurones en intégrant des composants d'accélération matérielle.

4.2 Overlay base avec un bloc mémoire

Après avoir recréé l'Overlay de base du matériel pour notre carte de développement PYNQ, nous allons maintenant modifier l'Overlay pour ajouter notre propre périphérique mappé en mémoire. Comme nous modifions l'Overlay de base, ce sera un nouvel Overlay, que nous devons intégrer correctement dans l'environnement PYNQ.

Dans cet Overlay, nous ajoutons une nouvelle mémoire bloc dans le PL dans laquelle nous pouvons lire et écrire à l'aide de l'environnement Python.

Pour ce faire, nous devons faire ce qui suit dans le design Vivado :

- Créer un nouveau port AXI (port 14) sur l'interconnexion AXI connectée au Master général 0
- Importer un nouveau contrôleur BRAM et le configurer pour avoir un seul port

- Utiliser le Block Memory Generator pour créer un BRAM. Régler le mode sur le contrôleur BRAM, RAM à un seul port
- Mettre le nouveau contrôleur BRAM sur la carte de mémoire PS de Zynq SoC

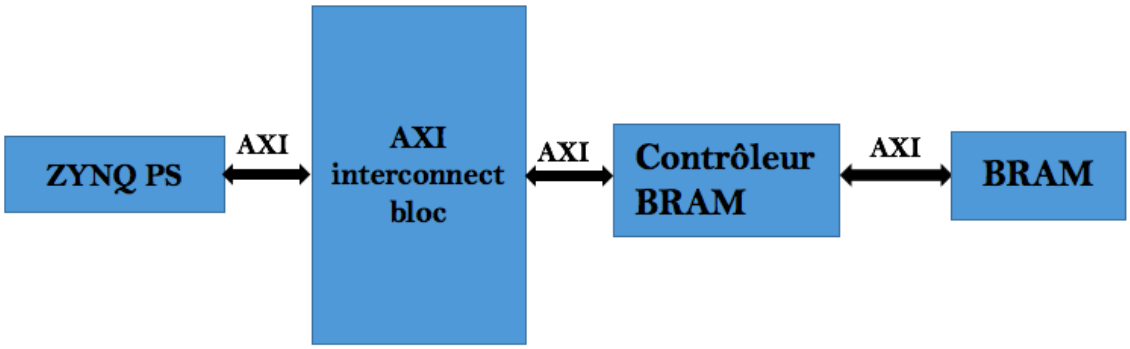


FIGURE 4.1: Schéma simplifié du nouvel Overlay

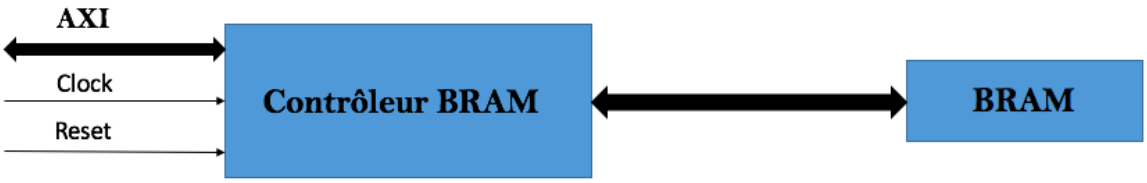


FIGURE 4.2: Les blocs ajoutés à l'Overlay base

Maintenant que ces quatre étapes sont complétées, nous sommes prêts à construire le fichier bit. Une fois que le fichier a été généré, nous sommes à mi-chemin pour construire un Overlay que nous pouvons utiliser dans notre conception. L'autre moitié nécessite la génération d'un script TCL qui définit la carte d'adresse du fichier bit. Pour ce faire, nous devons utiliser la commande :

Write_bd_tcl <nom.tcl>

Une fois que nous avons les fichiers TCL et bit, nous pouvons passer à l'étape suivante, c'est-à-dire importer les fichiers et créer les pilotes et les applications.

C'est là que nous devons allumer la carte de développement PYNQ et nous connecter au réseau avec notre Ordinateur. Une fois la configuration PYNQ téléchargée, il ne reste plus qu'à copier les fichiers TCL et bit dans la PYNQ.

Une fois que cela a été téléchargé, nous devons créer un notebook pour l'utiliser. Nous devons utiliser le module de l'Overlay existant fourni avec le paquet PYNQ pour ce faire. Ce module nous permettra de télécharger l'Overlay dans le PL du PYNQ. Une fois téléchargé, nous devons vérifier qu'il a été téléchargé correctement,

ce que nous pouvons faire en utilisant la fonction ol.is_loaded ().

L'interface la plus simple pour communiquer avec le nouvel Overlay est d'utiliser le module MMIO dans le paquet PYNQ. Ce module nous permet d'interagir directement avec des périphériques mappés en mémoire. D'abord, nous devons définir une nouvelle classe dans laquelle nous pouvons déclarer les fonctions permettant d'interagir avec l'Overlay.

Pour cet Overlay, nous n'avons besoin que d'une fonction d'initialisation, une fonction de lecture et une fonction d'écriture pour vérifier que la mémoire qu'on a ajouté fonctionne bien comme voulu.

4.3 Overlay base avec un bloc de multiplication en VHDL

Pour la création de cette Overlay il faut suivre les étapes suivantes :

- dans vivado ouvrir le projet de l'overlay base sans modification ;
- grâce à l'outil Create and Package IP, créer un nouvel IP core ayant une interface AXI4 Lite ;
- modifier le code VHDL pour avoir un bloc qui réalise la multiplication de deux nombres flottants et donne le résultat en sortie sur deux registres (pour prendre en compte l'overflow) ;
- créer un nouveau port AXI (port 5) sur l'interconnexion AXI connectée au Master général 0 ;
- intégrer le bloc au design de base et le connecter au port AXI (port 5) sur l'interconnexion AXI connectée au Master général 0 ;
- connecter l'horloge et le reset du bloc ajouté à ceux du processing system.

Par la suite, il suffit de suivre les mêmes étapes que pour le premier Overlay qu'on a créé, en générant les fichiers bit et TCL et en les intégrant dans la PYNQ pour tester leur fonctionnement.

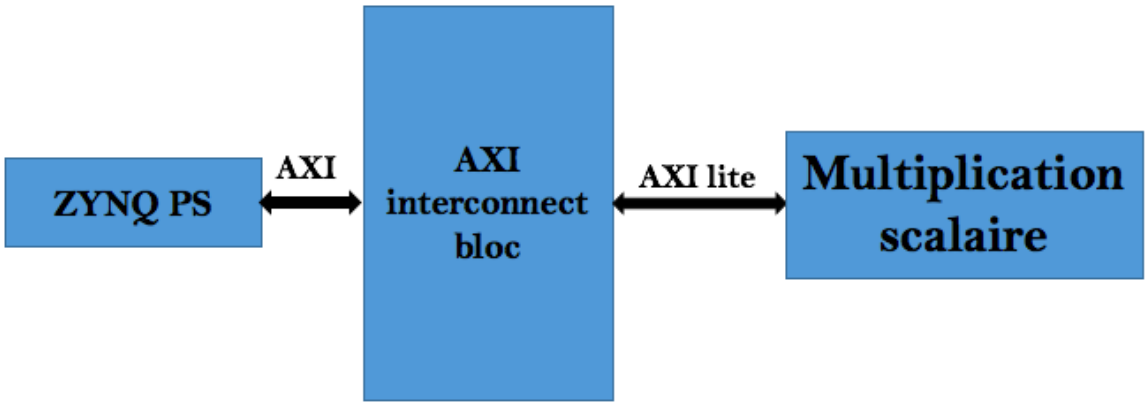


FIGURE 4.3: Schéma simplifié du nouvel Overlay avec bloc de multiplication en VHDL

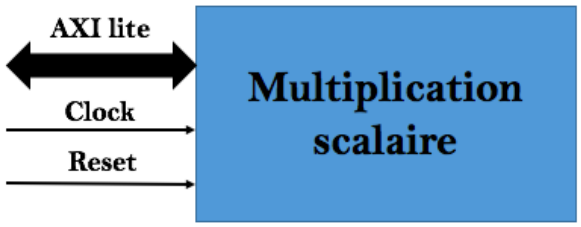


FIGURE 4.4: Bloc de multiplication en VHDL

4.4 Overlay base avec un bloc de somme de deux vecteurs

Pour la création de cette Overlay il faut suivre les étapes suivantes :

- dans vivado ouvrir le projet de l'overlay base sans modification ;
- grâce à l'outil Create and Package IP, créer un nouvel IP core ayant 3 interfaces AXI dont :
  - une interface AXI4 Lite ;
  - deux interfaces AXI4 Full.
- modifier le code VHDL pour avoir un bloc qui réalise la somme de deux vecteurs de nombres flottants ;
- créer 3 nouveaux ports AXI sur l'interconnexion AXI connectées au Master général ;
- intégrer le bloc au design de base et le connecter aux ports AXI ;
- connecter l'horloge et le reset du bloc ajouté à ceux du processing system.

Ce bloc est constitué de 4 composants :

- composant d'entrée connecté à l'interface AXI4 Full ;
- composant de sortie connecté à l'interface AXI4 Full ;
- composant de contrôle connecté à l'interface AXI4 Lite ;
- composant effectuant les calculs, connecté en interne avec tous les autres composants.

Les trois composants connectés à une interface AXI4 communiquent avec le bus AXI selon des protocoles bien spécifiques permettant de synchroniser le travail d'écriture et de lecture de la mémoire.

4.4.1 Composant d'entrée

Ce composant est utilisé pour stocker les données reçues en entrée dans sa mémoire et les transférer au composant de calculs qui effectuera la somme des vecteurs reçus.
La mémoire de ce composant est subdivisée en 4 blocs, telle que nous avons 4 array contenant 256 registres de 8 bits et qui sont disposées l'une à la suite de l'autre pour nous donner une mémoire de 256 mots de 32 bits.
Le composant d'entrée reçoit les données une à une (une donnée de 32 bits à la fois) à travers le bus AXI les stocke dans sa mémoire et les concatène dans deux registres de 128 X 32 bits chacun puis les transfère vers le composant de calcul via une liaison interne.
Il est caractérisé par :

- une interface Full en mode esclave ;
- une Largeur des données de 32 bits ;
- une mémoire de taille 1024.

4.4.2 Composant de calculs

Ce composant reçoit ,de la part du composant d'entrée, deux vecteurs concaténés, il calcule la somme de ces deux vecteurs et envoie vers le composant de sortie le vecteur de somme concaténé d'une taille de 128 X 32 bits.

4.4.3 Composant de sortie

Ce composant reçoit en entrée le vecteur somme concaténé, l'enregistre dans sa mémoire et envoie les données une à une par le bus AXI au moment de la lecture.
Ce composant est caractérisé par :

- une interface Full en mode esclave ;
- une Largeur des données de 32 bits ;
- une mémoire de taille 1024 Octets.

La mémoire est subdivisée en 4 blocs, telle que nous avons 4 array contenant 256 registres de 8 bits et qui sont disposées l'une à la suite de l'autre pour nous donner une mémoire de 256 mots de 32 bits.

4.4.4 Composant de contrôle

Ce composant est utilisé pour le contrôle, il permet d'initialiser les vecteurs A et B concaténés, il permet de lancer le calcul ou encore de voir l'état de certains registres pour mieux comprendre son fonctionnement au cours du temps et ainsi pouvoir mieux debugger si un problème survient.
Nous pouvons écrire dans ses registres à partir du Notebook de la Pynq ou encore instantanément à partir du programme pour les registres d'état.
Ce composant est caractérisé par :

- une interface Lite en mode esclave ;
- une Largeur des données de 32 bits ;
- un nombre de registres variant de 4 à 512 selon le besoin.

Dans notre projet, nous n'avons défini que 10 registres de 32 bits chacun dont uniquement 3 sont utilisés et le reste n'est qu'une précaution en cas de besoin.

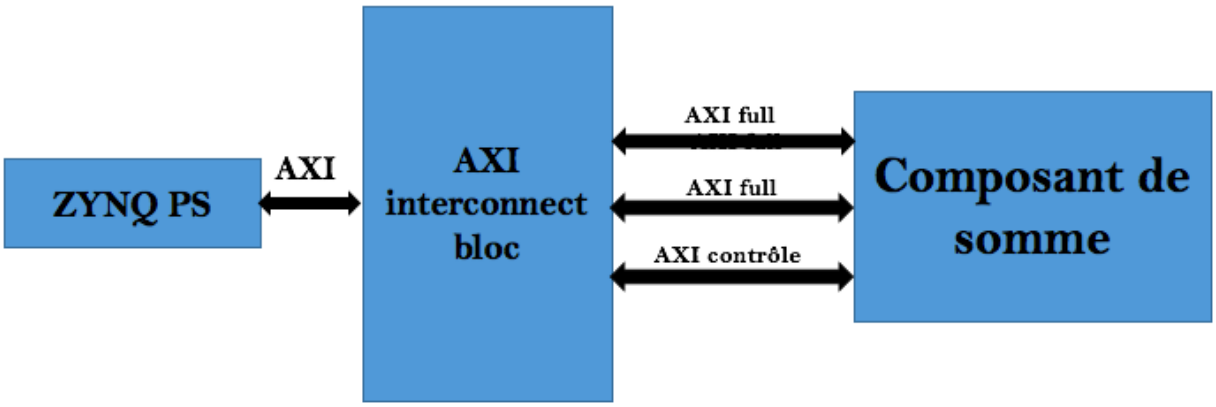


FIGURE 4.5: Schéma simplifié du nouvel Overlay avec accélérateur de calcul de la somme de deux vecteurs

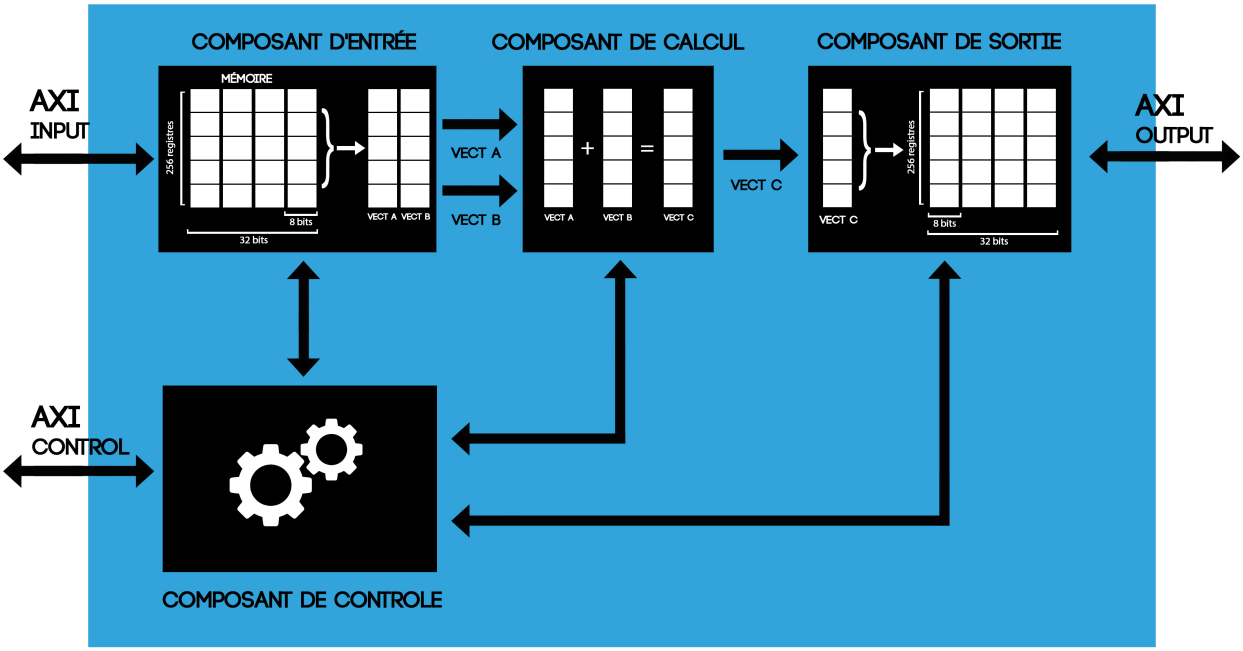


FIGURE 4.6: Accélérateur de calcul de la somme de deux vecteurs

4.5 Overlay base avec un bloc de produit de deux vecteurs plus accumulation

Ce composant est très similaire au précédent, il a les mêmes composants d'entrée, sortie et contrôle. On a remplacé la somme de deux vecteurs par le produit élément par élément de deux vecteurs suivis de la somme des éléments du vecteur résultant, ce qui nous donne une valeur scalaire en sortie au lieu d'un vecteur

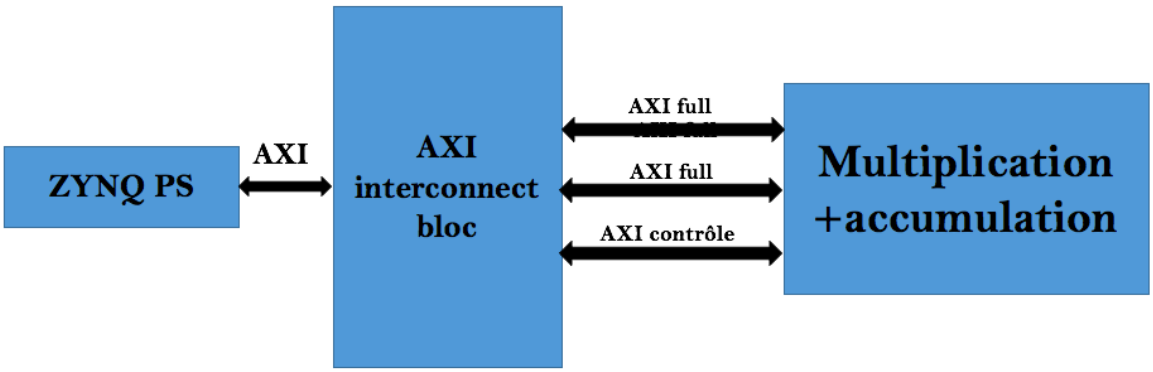


FIGURE 4.7: Accélérateur de calculs

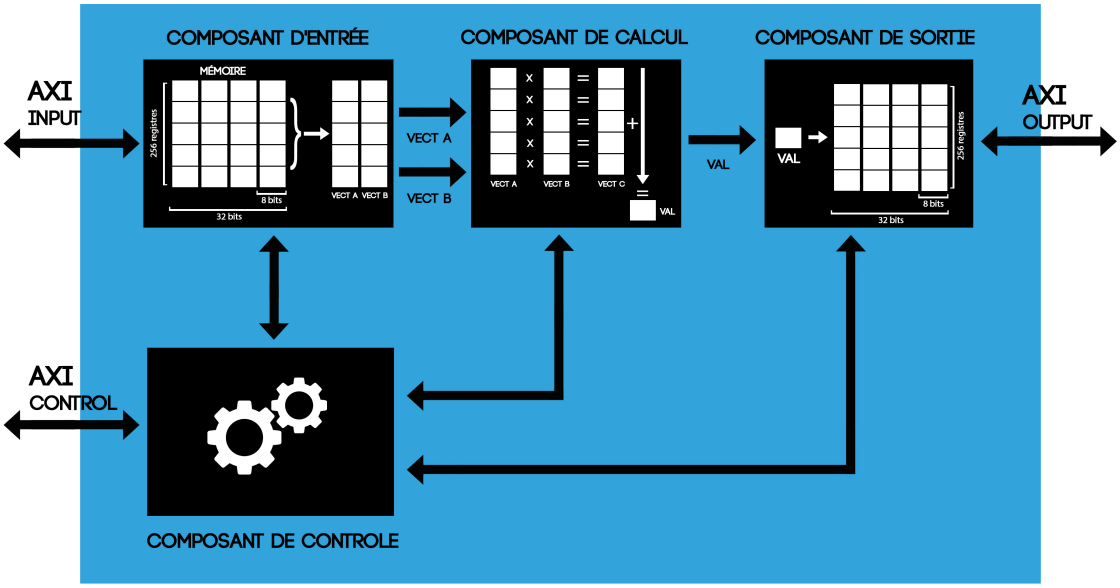


FIGURE 4.8: Exemple de réseau de neurones

4.6 Résultats de la synthèse

Ressource	Estimation	Disponibles	Utilisation
LUT	31004	53200	58.28%
LUTRAM	2513	17400	14.44%
FF	38388	106400	36.08%
BRAM	68	140	48.93%
DSP	6	220	2.73%
IO	118	125	94.40%
BUFG	6	32	18.75%
MMCM	2	4	50%

TABLE 4.1: Ressources utilisées par l'overlay base

Ressource	Estimation	Disponibles	Utilisation
LUT	36033	53200	67.73%
LUTRAM	3135	17400	18.02%
FF	47036	106400	44.21%
BRAM	70	140	50.36%
DSP	6	220	2.73%
IO	118	125	94.40%
BUFG	6	32	18.75%
MMCM	2	4	50%

TABLE 4.2: Ressources utilisées par l'overlay base avec composant d'accélération de la somme des vecteurs

Ressource	Utilisation
LUT	5029
LUTRAM	622
FF	8648
BRAM	2

TABLE 4.3: Ressources utilisées par le composant d'accélération de la somme des vecteurs

4.7 Conclusion

Dans ce chapitre, différents tests d'intégration ont été réalisés afin de mieux comprendre la communication avec la Pynq et de se familiariser avec les protocoles de communication du bus AXI4.
Un travail d'accélération a été entrepris afin d'optimiser les calculs dans un réseau de neurones.

Chapitre 5

Conclusion

Dans ce travail, différents modèles de réseaux de neurones artificiels ont été proposés pour répondre au problème de la classification des chiffres manuscrits.
Cette modèles ont réussi à accomplir une grande précision mais ce n'est pas la meilleure précision possible. Il existe des réseaux de neurones à plusieurs couches cachées telles que les réseaux de neurones convolutifs qui regroupent plusieurs couches cachées effectuant chacune un travail différent et qui arrivent à une plus grande précision mais qui sont plus difficiles à intégrer sur PYNQ à cause de la grande masse de données nécessaire à leur traitement. Elles consomment plus de ressources.
Un exemple de réseau convolutif est proposé en annexe sans que son implémentation ne soit effectuée.
Le travail d'accélération entrepris a permis une parallélisation de certains calculs mais pas de tous les calculs. La suite logique de ce travail serait donc de travailler sur l'accélération de tous les calculs effectués dans les réseaux de neurones simple pour commencer mais qui seraient réutilisables par les réseaux de neurones convolutifs.
Plusieurs améliorations de ce travail peuvent donc être proposées :

- l'utilisation des réseaux de neurones convolutifs ;
- la conception d'accélérateurs matériels pour les calculs sur les matrices ;
- l'intégration de composants permettant l'accélération de tous les calculs entrepris dans les réseaux de neurones.

Bibliographie

[1] *Les Reseaux de Neurones*. URL : http://outilsrecherche.over-blog.com/pages/Notes_212_le_cortex_humain_Les_Reseaux_de_Neurones-3191916.html.

[2] Oussama WEREFELLI. *Réseaux de neurones artificiels*. URL : https://fr.slideshare.net/OussamaWerfelli/rseaux-de-neurones-artificiels/7.

[3] *Playground tensorflow*. URL : http://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.36891&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=falsel.

[4] Jiqiong Qiu DEVFEST. *First step deep learning*. Jan 2016. URL : https://fr.slideshare.net/SfeirGroup/first-step-deep-learning-by-jiqiong-qiu-devfest-2016.

[5] *Convolutional networks*. URL : http://cs231n.github.io/convolutional-networks/#conv.

[6] *Average categorize some mnist digits*. URL : http://write-up.semantic-db.org/195-average-categorize-some-mnist-digits.html.

[7] *MNIST for ML Beginners*. URL : https://www.tensorflow.org/tutorials/mnist/beginners/.

[8] *Cifar-10 dataset*. URL : http://caincc.com/post/cifar-10-dataset.

[9] *FPGA*. URL : http://jjmk.dk/MMMI/PLDs/FPGA/fpga.htm.

[10] *PYNQ*. URL : https://forums.xilinx.com/t5/Xcell-Daily-Blog/Python-Zynq-PYNQ-which-runs-on-Digilent-s-new-229-pink-PYNQ-Z1/ba-p/726277.

[11] *Vivado*. URL : https://en.wikipedia.org/wiki/Xilinx_Vivado.

[12] URL : http://pynq.readthedocs.io/en/latest/6_overlays.html.

[13] *AMBA AXITM and ACETM Protocol Specification*. URL : http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf.

[14] Amer BAGHDADI. *Systèmes embarqués*. URL : https://formations.telecom-bretagne.eu/fad/course/index.php.

[15] *TensorFlow*. URL : https://en.wikipedia.org/wiki/TensorFlow.

[16] REFERENCE*. *Deep MNIST for Experts*. URL : https://www.tensorflow.org/tutorials/mnist/pros/.

[17] Michael NIELSEN. *Neural Networks and Deep Learning*. Jan 2017. URL : http://neuralnetworksanddeeplearning.com/index.html.

[18] *Réseau de neurones artificiels*. URL : https://fr.wikipedia.org/wiki/R%C3%A9seau_de_neurones_artificiels.

[19] Adam TAYLOR. *Exploring PYNQ's Base PL*. URL : https://forums.xilinx.com/t5/Xcell-Daily-Blog/Adam-Taylor-s-MicroZed-Chronicles-Part-157-Exploring-PYNQ-s-Base/ba-p/734085.

Annexe A

Complément

Epoque	Précision
1	96.62%¸
2	97.23%¸
3	97.68%¸
4	97.71%¸
5	97.95%¸
6	97.97%¸
7	98.05%¸
8	98.36%¸
9	98.15%¸
10	98.39%¸
11	98.14%¸
12	98.27%¸
13	98.17%¸
14	98.27%¸
15	98.29%¸
16	98.16%¸
17	98.41%¸
18	98.37%¸
19	98.53%¸
20	98.35%¸

TABLE 1.1: Variation de la précision au cours de l'entrainement pour le réseau de neurones à deux couches cachées

Dans la figure 1.1, on a tiré le schéma bloc de l'Overlay base à partir du logiciel Vivado pour illustrer les différents blocs le constituant et les interconnexions entre chaque bloc.

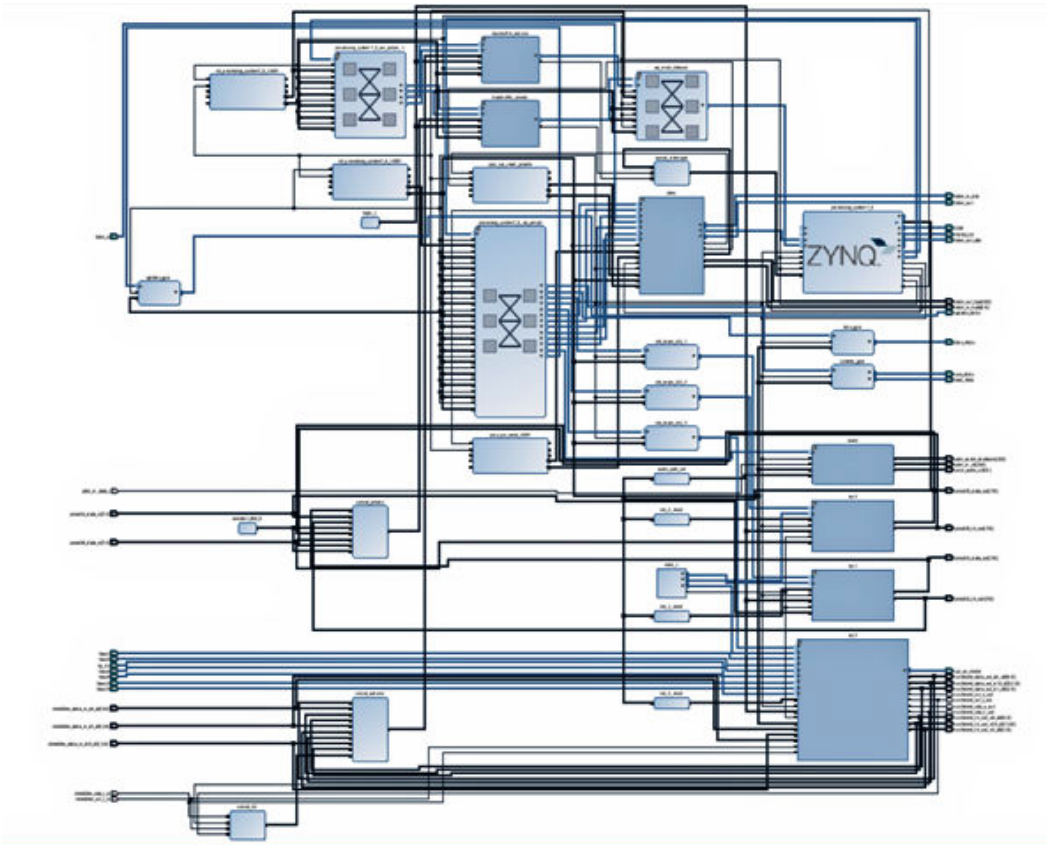


FIGURE 1.1: Schéma tiré de Vivado illustrant l'Overlay base
[19]

1.0.1 Description des signaux utilisés pour la communication du bus AXI

1.0.1.1 Signaux globaux

Signal	Source	Description
ACLK	Source d'horloge	Signal d'horloge global
ARESETn	Source de reset	Signal de réinitialisation globale, actif à l'état BAS

1.0.1.2 Signaux de lecture du canal d'adresse

Signal	Source	Description
AWID	Maître	Ecrire l'ID de l'adresse. Ce signal est la balise d'identification pour le groupe d'adresses d'écriture de signaux
AWADDR	Maître	Adresse d'écriture. L'adresse d'écriture donne l'adresse du premier transfert dans une transaction d'éclatement d'écriture
AWLEN	Maître	Longueur de rafale. La longueur de rafale donne le nombre exact de transferts dans une rafale. Cette information détermine le nombre de transferts de données associés à l'adresse.
AWSIZE	Maître	Taille de la rafale. Ce signal indique la taille de chaque transfert dans une rafale
AWBURST	Maître	Type de rafale. Le type de rafale et les informations de taille. Détermine comment l'adresse est calculée pour chaque rafale
AWLOCK	Maître	Type de verrouillage. Fournit des informations supplémentaires sur les caractéristiques atomiques du transfert
AWCACHE	Maître	Type de mémoire. Ce signal indique comment les transactions doivent progresser dans un système
AWPROT	Maître	Type de protection. Ce signal indique le niveau de privilège et de sécurité de la transaction et si la transaction est un accès aux données ou un accès à l'instruction.
AWQOS	Maître	Qualité de service, (Quality of Service QoS). L'identifiant QoS envoyé pour chaque transaction d'écriture.
AWREGION	Maître	Identificateur de région. Permet à une interface physique unique sur un esclave d'être utilisée pour plusieurs interfaces logiques.
AWUSER	Maître	Signal utilisateur. Signal optionnel défini par l'utilisateur dans le canal d'adresse d'écriture.
AWVALID	Maître	Adresse d'écriture valide. Ce signal indique que le canal signale une écriture et des informations d'adresse et de contrôlevalides.
AWREADY	Esclave	Adresse d'écriture prête. Ce signal indique que l'esclave est prêt à accepter une adresse et des signaux de commande associés

TABLE 1.2: Signaux de lecture du canal d'adresse

1.0.1.3 Signaux d'écriture du canal de données

Signal	Source	Description
WID	Maître	Ecrire une étiquette d'identification. Ce signal est la balise ID du transfert de données d'écriture.
WDATA	Maître	Ecrire des données.
WSTRB	Maître	Écrire des stroboscopes. Ce signal indique quelles voies d'octet contiennent des données valides. Il y a une écriture Bit stroboscopique pour chaque huit bits du bus de données d'écriture.
WLAST	Maître	Écrire en dernier. Ce signal indique le dernier transfert dans une rafale d'écriture.
WUSER	Maître	Signal utilisateur. Signal optionnel défini par l'utilisateur dans le canal de données d'écriture.
WVALID	Maître	Écriture valide. Ce signal indique que des données d'écriture et des scroboscopes valides sont disponibles.
WREADY	esclave	Écriture prête. Ce signal indique que l'esclave peut accepter les données d'écriture.

TABLE 1.3: Signaux d'écriture du canal de données

1.0.1.4 Signaux d'écriture du canal de réponse

Signal	Source	Description
BID	Esclave	Étiquette d'identification de réponse. Ce signal est la balise d'identification de la réponse d'écriture.
BRESP	Esclave	Réponse d'écriture. Ce signal indique l'état de la transaction d'écriture.
BUSER	Esclave	Signal utilisateur. Signal optionnel défini par l'utilisateur dans le canal de réponse en écriture.
BVALID	Esclave	Réponse d'écriture valide. Ce signal indique que le canal indique une réponse d'écriture valide.
BREADY	Maître	Réponse prête. Ce signal indique que le maître peut accepter une réponse d'écriture.

TABLE 1.4: Signaux d'écriture du canal de données

1.0.1.5 Signaux de lecture du canal d'adresse

Signal	Source	Description
ARID	Maître	Ecrire l'ID de l'adresse. Ce signal est la balise d'identification pour le groupe d'adresses de lecture de signaux
ARADDR	Maître	Adresse de lecture. L'adresse de lecture donne l'adresse du premier transfert dans une transaction d'éclatement de lecture
ARLEN	Maître	Longueur de rafale. La longueur de rafale donne le nombre exact de transferts dans une rafale. Cette information détermine le nombre de transferts de données associés à l'adresse.
ARSIZE	Maître	Taille de la rafale. Ce signal indique la taille de chaque transfert dans une rafale
ARBURST	Maître	Type de rafale. Le type de rafale et les informations de taille. Détermine comment l'adresse est calculée pour chaque rafale
ARLOCK	Maître	Type de verrouillage. Fournit des informations supplémentaires sur les caractéristiques atomiques du transfert
ARCACHE	Maître	Type de mémoire. Ce signal indique comment les transactions doivent progresser dans un système
ARPROT	Maître	Type de protection. Ce signal indique le niveau de privilège et de sécurité de la transaction et si la transaction est un accès aux données ou un accès à l'instruction.
ARQOS	Maître	Qualité de service, (Quality of Service QoS). L'identifiant QoS envoyé pour chaque transaction de lecture.
ARREGION	Maître	Identificateur de région. Permet à une interface physique unique sur un esclave d'être utilisée pour plusieurs interfaces logiques.
ARUSER	Maître	Signal utilisateur. Signal optionnel défini par l'utilisateur dans le canal d'adresse de lecture.
ARVALID	Maître	Adresse de lecture valide. Ce signal indique que le canal signale une lecture et des informations d'adresse et de contrôle valides.
ARREADY	Esclave	Adresse de lecture prête. Ce signal indique que l'esclave est prêt à accepter une adresse et des signaux de commande associés

TABLE 1.5: Signaux de lecture du canal d'adresse

1.0.1.6 Signaux de lecture du canal de données

Signal	Source	Description
RID	Maître	Lire une étiquette d'identification. Ce signal est la balise ID du transfert de données de lecture.
RDATA	Maître	Lire des données.
RSTRB	Maître	Écrire des stroboscopes. Ce signal indique quelles voies d'octet contiennent des données valides. Il y a une lecture Bit stroboscopique pour chaque huit bits du bus de données de lecture.
RLAST	Maître	Écrire en dernier. Ce signal indique le dernier transfert dans une rafale de lecture.
RUSER	Maître	Signal utilisateur. Signal optionnel défini par l'utilisateur dans le canal de données de lecture.
RVALID	Maître	lecture valide. Ce signal indique que des données de lecture et des scroboscopes valides sont disponibles.
RREADY	esclave	lecture prête. Ce signal indique que l'esclave peut accepter les données de lecture.

TABLE 1.6: Signaux de lecture du canal de données

Annexe A

Codes utilisés

1.1 Introduction

Dans ce chapitre, nous allons retrouver tous les codes pythons utilisé au long de ce projet.

1.1.1 Algorithme de réseau de neurones sans couche cachée

1.2 Code de réseaux de neurones sans couches cachées

```
from tensorflow.examples.tutorials.mnist import input_data
import argparse
import sys
import tensorflow as tf
FLAGS = None

def main(_):
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

  # Create the model
  x = tf.placeholder(tf.float32, [None, 784])
  W = tf.Variable(tf.zeros([784, 10]))
  b = tf.Variable(tf.zeros([10]))
  y = tf.matmul(x, W) + b

  # Define loss and optimizer
  y_ = tf.placeholder(tf.float32, [None, 10])
  cross_entropy = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
  train_step = tf.train.GradientDescentOptimizer(0.3).minimize(
                                                  cross_entropy)

  sess = tf.InteractiveSession()
  tf.global_variables_initializer().run()
```

```
  # Train
  for _ in range(5000):
    batch_xs, batch_ys = mnist.train.next_batch(200)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

  # Test trained model
  correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
  print(sess.run(accuracy, feed_dict={x: mnist.test.images,
                                      y_: mnist.test.labels}))

if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('--data_dir', type=str,
  default='/tmp/tensorflow/mnist/input_data',
                      help='Directory for storing input data')
  FLAGS, unparsed = parser.parse_known_args()
  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

1.2.1 Code Python de réseau de neurones avec une couche cachée

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
import argparse
import sys
FLAGS = None

def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)

def main(_):
  sess = tf.InteractiveSession()
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
  x = tf.placeholder(tf.float32, [None, 784])
  y_ = tf.placeholder(tf.float32, [None, 10])
  W0 = weight_variable([784, 700])
  b0 = bias_variable([1, 700])
  y = tf.tanh(tf.matmul(x, W0) + b0, name=None)
  W1 = weight_variable([700, 10])
  b1 = bias_variable([1, 10])
  y_conv = tf.matmul(y, W1) + b1
```

```
  keep_prob = tf.placeholder(tf.float32)

  cross_entropy = tf.reduce_mean(
  tf.nn.softmax_cross_entropy_with_logits(
  labels=y_,logits= y_conv))

  train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
  correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))


  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

  sess.run(tf.global_variables_initializer())

 for i in range(20000):
      batch = mnist.train.next_batch(50)
      if i%100 == 0:
          train_accuracy = accuracy.eval(feed_dict={
                                      x:batch[0],
                                      y_: batch[1],
                                      keep_prob: 1.0})
          print("step %d, training accuracy %g"%(i, train_accuracy))
      sess.run(train_step,feed_dict={x: batch[0],
                                     y_: batch[1],
                                     keep_prob: 0.5})

  print("test accuracy %g"%accuracy.eval(feed_dict={
                                      x: mnist.test.images,
                                      y_: mnist.test.labels,
                                      keep_prob: 1.0}))

if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('--data_dir', type=str,
  default='/tmp/tensorflow/mnist/input_data',
                      help='Directory for storing input data')
  FLAGS, unparsed = parser.parse_known_args()
  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

1.2.2 Code Python de réseau de neurones avec deux couches cachées

Dans cette algorithme, on utilise la bibliothèque Keras pour définir notre réseau de neurones, comme suit :

```
import numpy as np
import keras
```

```
import keras.layers
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop
from keras.utils import np_utils



batch_size = 128
num_classes = 10
nb_epoch = 20

# the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

np.savez_compressed("mnist.npz", x_train = x_train,
x_test=x_test, y_train=y_train, y_test=y_test)


x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = np_utils.to_categorical(y_train, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)



model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

model.summary()

model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])
```

```
history = model.fit(x_train, y_train,
                    batch_size=batch_size, nb_epoch=nb_epoch,
                    verbose=1, validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

np.set_printoptions(threshold=np.nan)

weights1 = model.layers[0].get_weights()
weights2 = model.layers[2].get_weights()
weights3 = model.layers[4].get_weights()
w0 = weights1[0]
b0 = weights1[1]
w1 = weights2[0]
b1 = weights2[1]
w2 = weights3[0]
b2 = weights3[1]


np.savez_compressed("weights.npz",w0=w0,
b0=b0,w1=w1,b1=b1,w2=w2,b2=b2)

print(np.shape(w2))


np.savez_compressed("mnist_reshaped.npz",x_train = x_train,
x_test=x_test, y_train=y_train, y_test=y_test)
```

1.2.3 Construction du réseau de neurones sur PYNQ

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import keras
        import keras.layers
        from keras.datasets import mnist
        from keras.models import Sequential
        from keras.layers import Dense, Dropout
        from keras.optimizers import RMSprop
        from keras.utils import np_utils
```

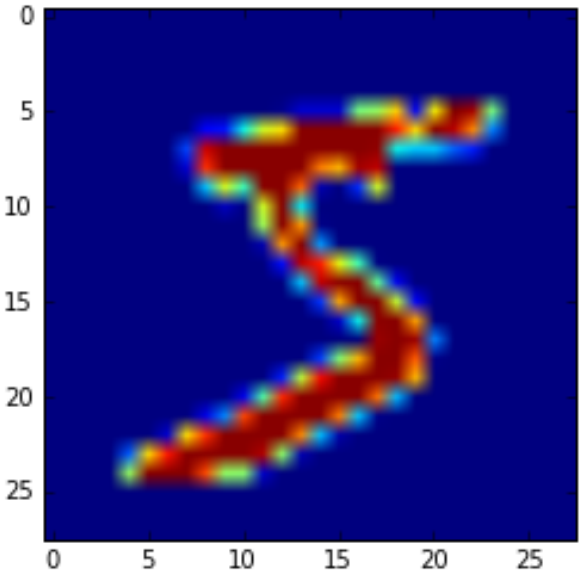
```
Using Theano backend.
```

```
In [2]: def ReLU(x):
            return x * (x > 0)
```

```
        def softmax(x):
            return np.exp(x) / np.sum(np.exp(x), axis=0)
```

In [3]:
```
nploader = np.load("/home/xilinx/docs/mnist.npz")
nploader.keys()
x_sh=nploader['x_train']
nploader.close()
print(x_sh[0,:].shape)
plt.imshow(x_sh[0,:])
plt.show()
```

```
(28, 28)
```



In [4]:
```
nploader = np.load("/home/xilinx/docs/mnist_reshaped.npz")
nploader.keys()
x_train=nploader['x_train']

x_test=nploader['x_test']
y_train=nploader['y_train']
y_test=nploader['y_test']
nploader.close()
```

In [5]:
```
print((x_train).shape)
print((y_train).shape)
print((x_test).shape)
print((y_test).shape)
```

```
(60000, 784)
(60000, 10)
(10000, 784)
(10000, 10)
```

```
In [6]: nploader = np.load("/home/xilinx/docs/weights.npz")
        nploader.keys()
        w0 = nploader['w0']

        b0 = nploader['b0']

        w1 = nploader['w1']


        b1 = nploader['b1']

        w2 = nploader['w2']

        b2 = nploader['b2']

        nploader.close()
```

```
In [7]: y1=np.zeros((1,512))
        y1_=np.zeros((1,512))
        y2=np.zeros((1,512))
        y2_=np.zeros((1,512))
        y=np.zeros((1,10))
        y_=np.zeros((1,10))
```

```
In [8]: y1_=(np.dot(x_train[0,:],w0))+b0
        y1=ReLU(y1_)
        y2_=(np.dot(y1,w1))+b1
        y2=ReLU(y2_)




        y_=(np.dot(y2,w2))+b2
        y=softmax(y_)
```

```
In [11]: print(np.argmax(y))
         print(np.argmax(y_train[0,:]))
         print(y_train[0,:])
```

```
5
5
[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
```

```
In [12]: if np.alltrue(np.argmax(y_train[0,:]) == np.argmax(y)):
             print("Gd answer")
         else:
             print("Bad answer!!!!!!")
```

```
Gd answer
```

1.2.4 Algorithme du réseau de neurone convolutif

```
from tensorflow.examples.tutorials.mnist import input_data


import tensorflow as tf

import argparse
import sys

FLAGS = None


def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)

def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')



FLAGS = None


def main(_):
  sess = tf.InteractiveSession()

  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
```

```
  x = tf.placeholder(tf.float32, [None, 784])
  y_ = tf.placeholder(tf.float32, [None, 10])

  W_conv1 = weight_variable([5, 5, 1, 32])
  b_conv1 = bias_variable([32])

  x_image = tf.reshape(x, [-1,28,28,1])
  h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
  h_pool1 = max_pool_2x2(h_conv1)

  W_conv2 = weight_variable([5, 5, 32, 64])
  b_conv2 = bias_variable([64])

  h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
  h_pool2 = max_pool_2x2(h_conv2)

  W_fc1 = weight_variable([7 * 7 * 64, 1024])
  b_fc1 = bias_variable([1024])
  h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
  h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

  keep_prob = tf.placeholder(tf.float32)
  h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
  W_fc2 = weight_variable([1024, 10])
  b_fc2 = bias_variable([10])

  y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

  cross_entropy = tf.reduce_mean(\\
  tf.nn.softmax_cross_entropy_with_logits(logits=y_conv, labels=y_))
  train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
  correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))


  sess.run(tf.global_variables_initializer())


  for i in range(20000):
      batch = mnist.train.next_batch(50)
      if i%100 == 0:
          train_accuracy = accuracy.eval(feed_dict={
                                      x:batch[0],
                                      y_: batch[1],
                                      keep_prob: 1.0})
          print("step %d, training accuracy %g"%(i, train_accuracy))
sess.run(train_step,feed_dict={x: batch[0],
                    y_: batch[1], keep_prob: 0.5})
```

```
  print("test accuracy %g"%accuracy.eval(feed_dict={
                                        x: mnist.test.images,
                                        y_: mnist.test.labels,
                                        keep_prob: 1.0}))



if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('--data_dir', type=str,
  default='/tmp/tensorflow/mnist/input_data',
  help='Directory for storing input data')
  FLAGS, unparsed = parser.parse_known_args()
  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```