Filière Électronique

Systèmes embarqués

Réseau de Neurones sur CPU, GPU et FPGA pour reconnaissance de caractères

OULED-AMEUR Amjad
SOUBIRANE Remy
AHL ZOUAOUI Othmane
CHAOUFI Sara
2019 - 2020

Encadrants : M. Bertrand LE GAL
M. Jérémie CRENNE

Table of Contents

Partie 1

Introduction

1.1 Intelligence artificielle, Machine learning et Deep learning

Aujourd'hui, on vit dans l'âge de l'intelligence artificielle, où les chercheurs sont intéressés par le développement des produits qui ne demandent pas beaucoup d'efforts pour être utilisés.

L'intelligence artificielle peut être définie comme le concept qui consiste à mettre en oeuvre des théories et des techniques pour que les machines soient capables à assumer des capacités humaines comme l'apprentissage, l'adaptation, l'entraînement et l'auto-correction.

L'apprentissage automatique, ou Machine Learning en anglais, est l'étude scientifique des algorithmes et des modèles statistiques utilisés par les systèmes informatiques pour effectuer une certaine tache sans utiliser des instructions explicites à chaque utilisation.

L'apprentissage approfondi, ou Deep Learning en anglais, regroupe une des classes des algorithmes de l'apprentissage automatique qui utilisent plusieurs couches pour extraire progressivement un niveau plus haut des caractéristiques, à partir de la couche d'entrée.

La figure suivante montre la relation entre les trois notions précédemment définis : l'apprentissage automatique est un champs de l'intelligence artificielle, et l'apprentissage approfondi est un champs de l'apprentissage automatique.

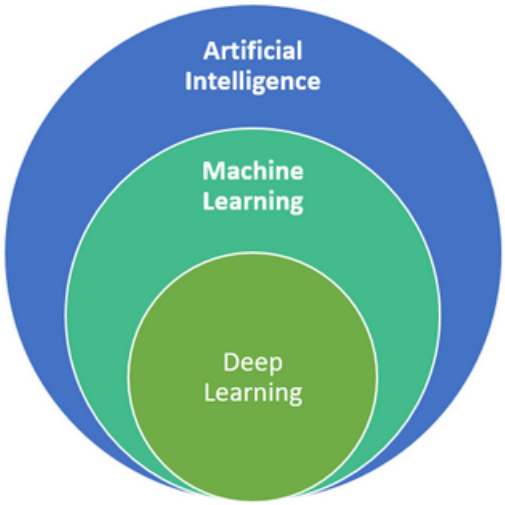

FIGURE 1.1 – Artificial Intelligence, Machine Learning and Deep learning

L'apprentissage approfondi a plusieurs application dans des domaines différents :

— **Reconnaissance** : cette catégorie regroupe la reconnaissance automatique de la parole (vocale) en se basant sur les modèles de Markov cachés et des réseaux de neurones artificielles(ANN) et la reconnaissance des images qui peut utiliser la base de données MNIST pour faire des tests. C'est d'ailleurs dans cette application où s'inscrit notre projet.

— **Traitement automatique**
Le traitement automatique des arts visuels est proche du process utilisé dans la reconnaissance d'image. Les DNNs sont capables d'identifier la période du style dans une peinture, d'appliquer un style artistique à une photo, générer une image basée sur plusieurs champs d'entrées visuelles. Le traitement automatique du langage naturel est basé sur les réseaux de neurones est couvre le syntaxe, sémantique, traitement du signal de parole et graphie, extraction d'informations et bibliométrie.
— **Science de vie et Médecine** L'apprentissage approfondi permet de prédire les taux et les effets dans les environnements chimiques dans les nutriments et les drogues, ce qui est utile dans la découverte des drogues et la toxicologie. La classification des cellules cancérigènes et la détection des lésions sont également des applications du Deep learning.
— **Marketing** Les réseaux de neurones approfondis sont utilisés pour calculer de façon approximative la valeur des actions du marketing direct possibles et extraire des fonctionnalités significatives pour proposer aux consommateurs des modèles de recommandations.
— **Finance** La détection des fraudes financière est l'application principale dans le domaine de finance.
— **Militaire** Le département de défense de l'USA utilise le Deep Learning pour entraîner des robots pour des missions d'observations.

1.2 A propos du projet

L'objectif de ce projet est de contourner les réseaux de neurones, surtout ceux de l'apprentissage approfondi, et construire un réseau de neurones pour la détection des chiffres écrits à la main. Ensuite, on utilise comme cibles le CPU et GPU pour implémenter le réseau. Finalement, on doit réaliser une implémentation sur FPGA .

Pour ce projet, on utilise MNIST, qui est une base de données de chiffres écrits à la main.

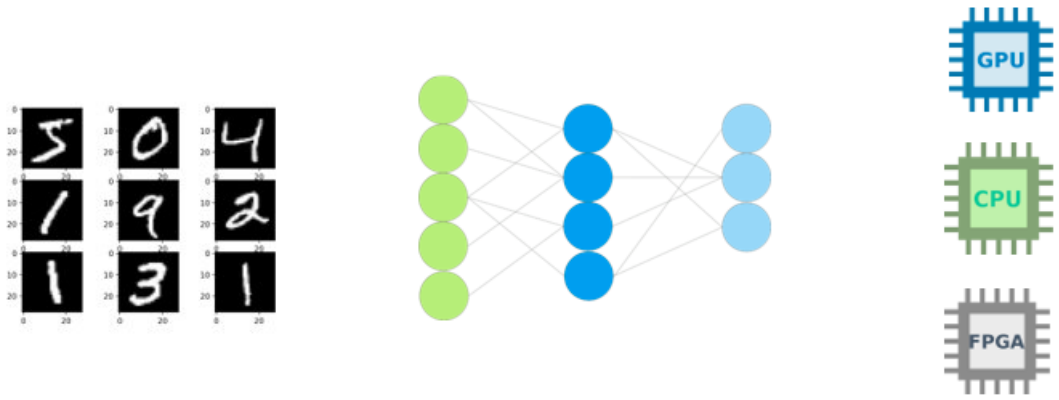

Figure 1.2 – Structure du projet

1.3 Étapes du projet

On a commencé par étudier l'architecture des réseaux de neuronnes et le Deep learning. Ensuite, on a commencé l'écriture du code en C++ du réseau de neurones sans bibliothèques ainsi que sur Python en utilisant Keras comme bibliothèque. On a découvert l'outil HLS4ML, en parallèle nous avons effectué l'implémentation sur CPU et GPU avec Cuda. Ensuite, on a fait un premier test de l'outil HLS4ML avec un simple réseau de neurones. On a fait l'adaptation de la synthèse HLS4ML pour les limitations matérielles de notre cible en utilisant cette fois le modèle de réseau de neurones complet. Finalement, on a effectué la conversion en VHDL et on a effectué des changements sur l'architecture pour optimiser les ressources utilisées.

Partie 2

Implémentation de l'architecture du réseau neuronal sur C++

2.1 Introduction

Les réseaux neuronaux représentent, de nos jours, une approche prometteuse donnant la possibilité de contourner plusieurs problèmes de perception et de raisonnement, ou ce qu'en appel les problématiques de l'apprentissage profond. A titre illustratif, notre application a comme vocation de détecter l'écriture manuscrite des 10 digits de zéro à neuf. Dans cette approche nous partons du principe que nous souhaitons construire l'architecture entière d'un réseau de neurones dédié à notre application uniquement en utilisant le programmation orientée objet en C++ dans la perspective de convertir cette première architecture en un programme en systemC et traduire ce dernier en description matérielle en utilisant l'outil magique Vivado_Hls. Nous commençons d'abord par détailler la théorie des réseaux neuronaux afin d'appréhender le fonctionnement de l'architecture et faciliter la justification des choix entretenus durant la programmation du réseau.

2.2 La théorie des réseaux neuronaux

2.2.1 Aspects historiques

L'année 1943 était témoin de l'apparition des premiers neurones s'inspirant des neurones biologiques et apte à mémoriser des fonctions booléennes de base. L'idée de McCulloch et W. Pitts était donc de percevoir les neurones biologiques comme des portes logiques effectuant des opérations booléennes. Ces réseaux de neurones artificiels, mimant le fonctionnement du système nerveux humain sont massivement parallèles et peuvent apprendre et mémoriser les informations d'interconnexion entre neurones.
Le premier modèle fonctionnel des réseaux de neurones était le Perceptron proposé en 1958 par Rosenblatt. Il s'agissait du premier modèle capable d'apprentissage. Le premier modèle à résoudre un problème de quantification était celui de Kohonen en 1982 dédié aux problématiques de reconnaissance et de classification.

2.2.2 Notions de base

Forward propagation

Nous commençons d'abord par définir ce qu'est un réseau de neurone. Un réseau neuronal est la représentation en graphes plus ou moins profond, d'objets élémentaires, les neurones formels.

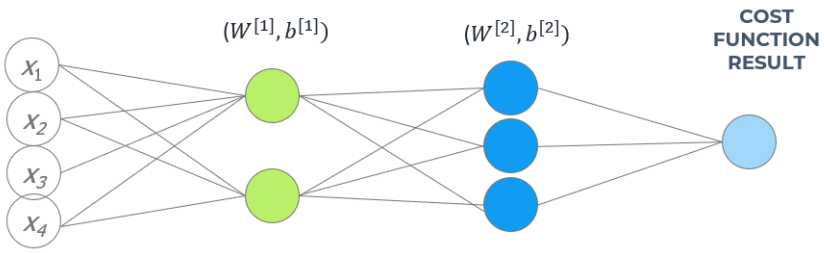

FIGURE 2.1 – Un exemple de réseau de neurones.

Un neurone est une unité de calcul qui réalise une simple somme pondérée (combinaison affine) de ses entrées et passe le résultat dans une fonction d'activation qui a pour rôle d'introduire des non-linéarité. Un neurone est donc avant tout une fonction mathématique dont la représentation graphique est donnée par la figure ci-dessous.

FIGURE 2.2 – Représentation d'un neurone.

Les différents types de neurones se différencient par la nature de leur fonction d'activation qui peut avoir différentes formes et donc divers paramètres. Les principaux types sont :

- a) Tout ou rien
- b) Fonction signe
- c) Plus ou moins à signe.
- d) Fonction affine.
- e) Saturation (ReLU function)
- f) Sigmoïde.
- g) Fonction arc-tangente ;
- h) Fonction radiale de base du type gaussienne.

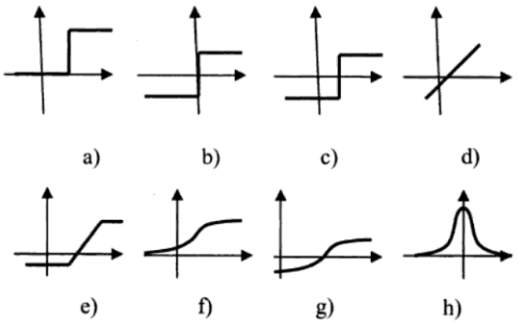

FIGURE 2.3 – Les différents types des fonctions d'activations

Pour une classification à m classes, comme il est le cas dans notre application, le neurone de sortie intègre une fonction d'activation du type softmax dont les m valeurs de sortie correspondent à des probabilités d'appartenance à chacune des classes.

$$a^{[1]} = g^{[1]}(z^{[1]}) \qquad \text{ACTIVATION FUNCTION}$$

Un neurone formel n'est donc rien d'un autre qu'une unité opérationnelle qui effectue une somme pondérée suivie d'une non-linéarité. L'association des ces unités forme ce qu'on a défini par un réseau. Le réseau est un maillage de plusieurs neurones organisés en couches. Une couche est un ensemble de neurones n'ayant pas de connexion entre eux. La couche d'entrée qui contient les pixels entrant à notre réseau n'introduit aucune modification et donc n'est pas comptabilisée. Un neurone d'une couche cachées est connecté en entrée à chacun des neurones de la couche précédente et en sortie à chaque unité de la couche en aval. Chaque couche a un rôle précis et est caractérisé par le nombre de neurones et plusieurs d'autres paramètres. Parmi ces paramètres, on retrouve la matrice des poids synaptiques qui sont les coefficients de pondération.
Après avoir calculé la fonction d'activation de la couche actuelle, il est temps de passer à la couche suivante en affectant l'activation de la couche courante aux entrées de la couche suivante. Nous itérons cette procédure jusqu'à la dernière couche où on doit calculer l'erreur qui est donnée par la formule ci-dessous. Nous calculons ensuite l'erreur totale relative à l'ensemble de l'apprentissage (x1, y1), …, (xn, yn) qui est mesuré par la somme des erreurs et c'est ce qu'on appelle le coût (Cost)

$$erreur = -y.\ln\big(p(x)\big) - (1-y).\ln\,(1-p(x))$$

$$Cost(W,b) = \sum_{i=1}^{n} erreur(i)$$

Où i correspond à l'i-ème entrée du modèle au cours de l'entraînement.

Backward propagation

Parmi les caractéristiques d'un réseau de neurones, la plus fondamentale est l'aptitude à apprendre de son environnement suite à des mis à jour vis-à-vis des stimulations de cet environnement.
La liaison entre deux neurones de deux couches différentes est affectée d'un coefficient de pondération qui est mis à jour par apprentissage du réseau. L'apprentissage des poids consiste à minimiser la fonction de coût calculée précédemment. Une façon de calculer le minimum de coût est d'utiliser l'algorithme de la descente du gradient (Gradient descent). Il s'agit tout simplement d'un algorithme itératif d'optimisation qui va changer à chaque itération les valeurs des coefficients de pondération et des biais afin de trouver le minimum de la fonction coût qui est une fonction convexe comme le montre la figure suivante.

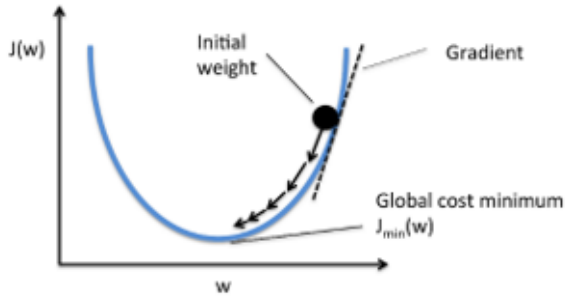

Figure 2.4 – Représentation de l'algorithme de la descente du gradient

Pour implémenter cet algorithme, nous partons d'un point initial aléatoire, ce qui revient à initialiser les poids et les biais de manière aléatoire. Après avoir calculé le coût dans la forward propagation, nous calculons la dérivée de cette fonction par rapport à chacun des paramètres pour

en déduire la dérivée (le gradient) par rapport au paramètres W et b. L'étape suivante est la mise à jour des paramètres du modèle. On progresse d'un certain pas dans la direction de la pente, ce pas est dit Learning Rate ou bien vitesse d'apprentissage. C'est un hyperparamètre qui a un impact sur la performance finale du modèle et qu'il faut choisir avec soin. Il est à mentionner qu'un bon Learning Rate est aux alentours des 0.001.

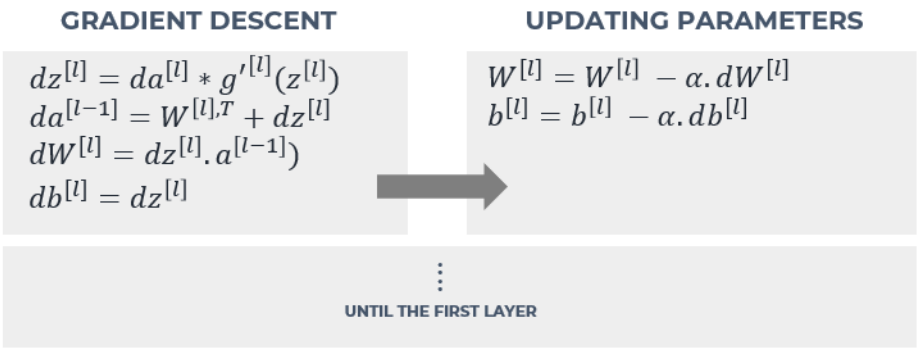

Figure 2.5 – Représentation de la propagation dans le sens inverse

A chaque itération, l'algorithme avancera d'un pas et trouvera un nouveau couple (W, b) qui réduit le résultat de la fonction de coût. Après un nombre d'itération, le coût d'erreur se réduit et l'idéal c'est atteindre un coût nul. C'est l'un des algorithmes les plus utilisés dans les problèmes d'apprentissage supervisé et c'est ce qui fait que notre modèle apprend et s'améliore.

2.3 Architecture du zéro en c++

Comme cela a été mentionnée antérieurement, nous souhaitons coder notre architecture du réseau neuronal en utilisant uniquement la bibliothèque standard du C++ sans avoir recours à aucune bibliothèque spécialisée en accélération de calcul ou bibliothèque externe. Nous partons alors de ce principe pour aboutir à une architecture matérielle. Ce fut une approche exigeante du point de vue qu'il s'agit d'une réalisation d'un réseau générique avec plusieurs couches de profondeur. La vocation de la partie suivante est de détailler et justifier les différentes démarches et choix qui ont été entretenu lors du développement de cette application.

2.3.1 Définition de l'architecture de base

Avant de se mettre à la mise en œuvre de notre application, il fallait tout d'abord choisir l'architecture du réseau, c'est-à-dire les entrées, le nombre de couches cachées, l'agencement des neurones dans chacune des couches ainsi que les paramètres de test et d'entrainement.

Contraintes et limitations

Pour mettre en œuvre notre application, plusieurs contraintes s'imposent :

- Cette application est destinée à être implémenté sur une carte FPGA de la famille Nexys4, cela implique que les ressources matérielles sont très limitées. Cette première contrainte impose que notre architecture de réseau doit être minimaliste et doit être composée d'un nombre minimum de couches et de neurones.
- La base de données qui serait utilisée pour l'entraînement et le test est la fameuse MNIST. Il s'agit d'un jeu de donnée de chiffres écrit à la main très utilisé pour les applicatifs du Deep Learning et notamment la reconnaissance de l'écriture manuscrite. Les images de cette

bibliothèque sont de dimensions 28x28 pixels. Cela impose que la couche d'entrée de notre modèle devrait être constituée de 784 neurones.
— La vocation de cette application est de déterminer quel chiffre figure sur l'image. La couche de sortie devrait être alors composée de 10 neurones, qui correspondent à la probabilité relative à chacun des dix digits de zéro à neuf.
— Le réseau devrait être entrainé en utilisant des ressources de l'ordinateur personnel (La mémoire RAM pour stocker les données qui viendront alimenter notre réseau). Cela impose qu'il faut choisir une taille du batch suffisamment grande pour finir l'entrainement rapidement et qui, en même temps, n'excède pas la taille de mémoire.

Architecture de base

En tenant compte de ces contraintes, nous partons sur une architecture contenant un minimum de couches. 784 neurones sont indispensable pour la couche d'entrée, 32 pour la couche cachée et 10 unités pour la couche de sortie qui ont pour rôle de donner les probabilités de chacun des digits.

2.3.2 Implémentation

Organisation des fichiers et Diagramme UML

Nous présentons dans ce paragraphe les principaux fichiers source constituant le squelette de notre application.

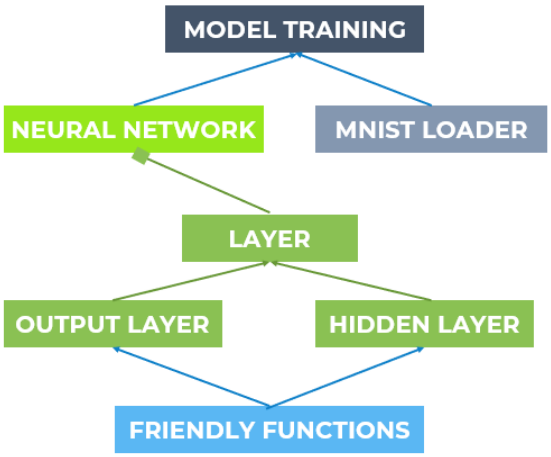

FIGURE 2.6 – Représentation de la propagation dans le sens inverse

Notre programme est composé de plusieurs fichiers source qui sont organisés de la manière suivante :

— **Layer** : Ce fichier contient la définition de la classe Layer, le constructeur de la classe, le destructeur et toutes les méthodes d'initialisation, de chargement et de sauvegarde des paramètres de couches.
— **Output Layer** : Ce fichier contient la définition et les méthodes associées à la classe output layer qui est une classe fille de la classe Layer.
— **Hidden Layer** : Il définit la classe hidden_layer qui est également une sous-classe de la classe Layer.
— **Neural Network** :Ce fichier définit la classe du réseau qui rien autre qu'une instanciation de plusieurs objets layer et d'une définition de paramètres généraux du réseau. Ce fichier contient également les méthodes associées au réseau.

— **MNIST loader** :Il s'occupe du chargement des pixels de la base de données MNIST après importation du fichier du dataset. Il charge également les Labels qui sont les résultats à comparer avec les prédictions du réseau.
— **Main** : Le fichier main, quant à lui, il rassemble toutes les classes définies précédemment pour réaliser un entraînement du modèle.

Implémentation de l'architecture

Dans cette partie, nous expliquons en détails le rôle de chacune des méthodes et attributs de chaque classe constituant l'ensemble de ce programme. Avant de commencer à détailler les différents choix mis en place, il est important de mentionner que les coefficients de ce programme ont été pris du type long double pour garder une précision élevée à notre modèle. Les différentes et principales variables du modèle telles les matrices de poids, de z et les résultats d'activation ont été représentées par des types vector<vector<long double».

— **Friendly Functions**
Ce fichier regroupe les fonctions qui ont été utiles pour tous les autres fichiers du programme et il fait objet d'include dans chacun de ces fichiers. Il contient principalement le code source des fonctions d'activation de notre réseau. Dans ce fichier, La fonction softmax a la particularité d'être une fonction vectorisée, c'est-à-dire qu'elle prend en argument une matrice de long double. La seconde fonction d'activation implémentée dans ce fichier est la fonction ReLU. La forme de cette fonction non-linéaire est donnée par la figure X.f. Elle prend en argument un type long double et renvoie le résultat en même type. Ce fichier contient également la fonction Transpose qui prend en argument une matrice du type vector<vector<long double» et retourne la matrice transposée du même type.

— **Classe Layer**
La classe layer est très importante dans notre architecture, elle vient définir une couche qui est l'objet élémentaire du réseau. Une classe couche est définie par les attributs public suivants :

CLASSE LAYER
• **nUnits :** Entier qui définit le nombre de neurones dans la couche. • **nInputs :** Entier qui définit le nombre d'entrées associées à la couche. • **Activ :** un string qui définit le nom de la fonction d'activation de la couche. • **LearningRate :** un flottant qui définit le Learning rate de la couche ou bien du réseau. • **DropoutRate :** un flottant qui définit le pourcentage du dropout choisi pour la couche. • **X, a, w, z :** Matrices du type vector<vector<long double>> qui définissent les paramètres de la couche. • **dX, da, dw, dz :** Matrices du même type définissant les gradients des paramètres. • **wSaved, bSaved :** Matrices du même type permettant de stocker les nouveaux paramètres
• **Layer () :** Constructeur principale de la classe. • **~&Layer : destructeur.** • **get_nUnits() :** Retourne le nombre de neurones dans une couche (pour debug) • **get_nInputs() :** Retourne le nombre d'entrées d'une couche (pour debug) • **init () :** Cette méthode initialise tous les paramètres de la couche. • **initRand () :** Cette méthode initialise les paramètres W et B aléatoirement. • **InitZero () :** Initialise les paramètres matrices W et B par des zéros • **SaveParams () :** Sauvegarde les nouveaux paramètres de la couche (poids et biais) • **LoadParams () :** Méthode qui charge les paramètres de la couche. • **PrintLayer () :** Méthode qui affiche les paramètres de couche (pour debug)

Nous détaillons ensuite la classe Hidden Layer et la classe Output Layer qui sont des classes qui héritent de la classe Layer. La couche d'entrée, quant à elle, elle n'a pas besoin d'être modélisée vu qu'elle n'a pour rôle que l'extraction et l'organisation des pixels. La raison pour laquelle nous avons choisi d'implémenter deux classes filles, c'est parce que les couches de sortie et les couches cachées ont des comportements différents. La couche de sortie est caractérisée par un calcul de la fonction coût et la fonction d'activation qui est du type softmax alors que la couche cachée est caractérisée par une fonction d'activation différente.

— **Sous-classe Hidden Layer**
Dans cette classe, nous héritons les attributs de la classe layer et nous ajoutons des méthodes de Forward Propagation et Backward Propagation. Ces fonctions sont différentes par rapport à la couche de sortie, vu que dans cette dernière, durant la propagation directe, nous calculons le coût.

— **Sous-classe Output-Layer**
Dans l'objectif de calculer le coût et en appliquant la formule vue dans un paragraphe précédent, nous aurions besoin de savoir les vraies valeurs contenues dans chacune des images afin de les comparer avec les prédictions de notre réseau. Le constructeur de cette classe est donc différent des autres classes. Il commence par charger à partir de la classe MNIST_DATASET, qui sera vue prochainement, les valeurs réelles dites labels.

— **Classe Neural_Network** :
La vocation de cette classe est de représenter un réseau neuronal qui est une instanciation d'objets couches (couches cachées et couches de sortie). Les attributs de cette classe sont les caractéristiques du réseau.

CLASSE NEURAL NETWORK
• **BatchSize :** Entier qui définit le nombre d'image à envoyer dans chaque paquet d'entrée • **State :** un booléen qui définit s'il s'agit d'un entraînement ou bien d'une inférence de test sur le réseau. La différence est en l'absence ou la présence de la Backward propagation. • **nHiddenLayers :** un entier qui définit le nombre de couche cachés constituant le réseau • **hLayer** :** un flottant qui définit le Learning rate de la couche ou bien du réseau. • **outLayer* :** Un pointeur vers la couche de sortie du réseau. • **nEpochs :** Entier qui détermine le nombre d'epoch prévu pour l'entraînement du réseau. • **X :** Vecteur bidimensionnel contenant les images en entrée du réseau qui sont bien les entrées de la première couche du réseau. • **Y :** Vecteur unidimensionnel contenant les valeurs des prédictions (Labels)
• **NeuralNetwork () :** Constructeur principale de la classe du réseau neuronal. • **~&NeuralNetwork :** destructeur. • **init () :** On initialise le réseau en initialisant toutes les couches du réseau. • **entireForProp () :** La forward propagation du réseau entier est réalisée en appliquant la fonction linéaire suivie de la fonction d'activation dans chacune des couches. • **EntireBackProp () :** La Backward propagation du réseau est effectué en appliquant l'algorithme de descente de gradient suivi d'une mise à jour des paramètres pour chacune des couches. • **Train () :** Cette méthode permet de lancer l'entraînement d'un modèle après définition des différents paramètres. Elle prend en argument le jeu de données d'entraînement. • **Test () :** réalise une inférence sur le réseau. • **Save () :** Sauvegarde les nouveaux paramètres de chacune des couches (poids et biais) • **Load () :** Méthode qui charge les paramètres de chacune des couches (poids et biais). • **PrintNet () :** Méthode qui affiche les paramètres des couches (pour debug)

— **Classe Mnist_Data_Set** :
Cette classe modélise un jeu de données dédié à l'entraînement, au test et à la validation de notre réseau. Elle contient toutes les méthodes qui permettent de charger une base de données, de trancher la base de données en des paquets (batch). Cette classe est définie comme suit :

CLASSE MNIST_DATA_SET
• **size :** Entier qui définit le nombre d'image contenu dans le dataset en entier. • **nImages :** Entier désignant le nombre d'images dans la base de données. • **nLabels :** Entier désignant le nombre de labels dans la base de données. • **imageSet :** Matrice des images, chaque colonne de la matrice est une image de 784 pixels, donc matrice de 784 lignes. • **LabelSet :** Vecteur contenant les labels à comparer avec les prédictions en sortie de réseau.
• **MnistDataSet () :** Constructeur principale de la classe du réseau neuronal. Cette classe a un second constructeur qui prend en argument le chemin vers les fichiers de la base de données et appelle les deux méthodes de charge. • **~&MnistDataSet :** destructeur. • **LoadImages () :** On prend en argument le chemin vers le fichier d'images du dataset. On charge cette dernière dans les attributs de la classe correspondants. • **LoadLabels () :** On prend en argument le chemin vers le fichier d'images du dataset. On charge cette dernière dans les attributs de la classe correspondants. • **Split () :** Cette fonction va s'occuper de découper la base de données chargée en des paquets, dits batch.

— **Fichier Main et entraînement du modèle** :
Maintenant que tous les objets indispensables pour entraîner un modèle sont réunis, il est temps de présenter le fichier main qui forme plusieurs instanciation des différentes classes. Nous commençons tout d'abord par instancier deux objets du type **MnistDataSet**, le premier pour créer un dataset d'entraînement et le second pour le test du réseau. Nous appelons ensuite les deux méthodes de chargement des jeux de données **LoadLabels** et **LoadImages**. Nous créons ensuite notre réseau en spécifiant dans les arguments du constructeur la taille du batch fixée à 128 échantillons, le nombre de couches cachées qui a été pris à 1 , le nombre d'entrée à 784. Nous spécifions également le type du jeu de données instancié précédemment à TrainingSet et nous fixons le dropout_rate à 0.25.

2.3.3 Résultats et perspectives

L'idée de cette implémentation était d'abord de monter l'architecture du réseau de neurones dédiée à la détection d'écriture manuscrite uniquement en utilisant la librairie standard en C++ dans la perspective de convertir cette architecture en un programme en systemC et finalement le traduire ce dernier en description matérielle via Vivado_Hls. Pour conclure, nous avons pu monter l'architecture d'un réseau simpliste sur C++ et l'entraîner sur les ressources du CPU.
Cependant, par faute de temps, nous n'avons pas pu traduire ce code en utilisant la bibliothèque logicielle systemC pour aboutir à une architecture matérielle implantable sur FPGA. Cette première description modulaire a formé une bonne base en ce qui concerne les réseaux de neurones et nous a permis d'appréhender de plus près ce qui se passe dans les fonctions haut niveau, que ça soit sur les fonctions prédéfinies en python sur les framework Keras ou TensorFlow. Ce choix a été donc abandonné pour se concentrer sur d'autres approches encore plus prometteuse et faisable, notamment l'approche de synthèse HLS en utilisant l'outil HLS4ML.

Partie 3

Architecture CPU/GPU : Python, Keras et TensorFlow

Toujours dans une optique de découverte des réseaux de neurones et de comparaison des solutions disponibles pour leur utilisation, cette partie a été consacrée à l'utilisation du langage python avec l'API Keras et le framework Tensorflow. Pour prendre en main ces différents outils, le livre *DEEP LEARNING with Python* de François Chollet aux éditions MANNING a été utilisé. Un code créé à partir de ce livre servira de test pour la comparaison de performances sur cible CPU et GPU.

3.1 Présentation des outils

TensorFlow est l'outil open source d'apprentissage automatique de Google actuellement le plus utilisé pour des application dans le domaine de l'IA. Il s'est donc imposé comme un outil incontournable à tester et évaluer pour ce projet. L'un de ses avantage est son interface pour le langage Python le rendant simple d'utilisation. Il permet notamment de rendre l'IA plus accessible pour le développement du deep learning et l'utilisation de réseaux de neurones. Tensorflow, pour représenter les données et opérations manipulées, utilise des tenseurs (objets mathématiques assimilables à des matrices de dimensions supérieures à 2).

Pour utiliser Tensorflow et simplifier son utilisation, l'API Keras (application programming interface) a été choisie. Celle-ci, de haut niveau, se compose de commandes très simples et intuitives, permettant de construire un réseau de neurones et de l'intégrer dans un algorithme. Pour résumer, le code écrit en Python appelle des fonctions de l'API Keras qui, elles, appellent des fonctions de Tensorflow pour finalement créer le réseau souhaité.

Selon la cible hardware choisie, il est nécessaire d'utiliser une version de Tensorflow adaptée. Dans les expérimentations suivantes, TensorFlow sera utilisé pour des réseaux sur cible CPU et TensorFlow-GPU pour des réseaux sur GPU. Sur ce dernier, il est aussi indispensable d'ajouter la plateforme CUDA permettant d'effectuer des calculs sur GPU et la bibliothèque CuDNN (pour "Deep Neural Network") servant à implémenter des opérations spécifiques aux réseaux de neurones sur GPU. Tout cela est résumé sur la figure 3.3.1 montrant les bibliothèques et cibles associées.

Pour développer le code Python sur système d'exploitation Windows, l'environnement Pyzo a été choisi. Bien que facultatif, celui-ci était connu depuis la prépa et possède des fonctionnalités pratiques comme une gestion de shells efficace.

Au final, les versions des logiciels utilisés sont :

1. Python 3.6.8
2. Pyzo 4.9.0 Windows
3. Keras 2.3.1
4. Tensorflow et Tensorflow-GPU 2.0.0
5. CUDA v10.1.168 et CUDNN v7.6.5.32

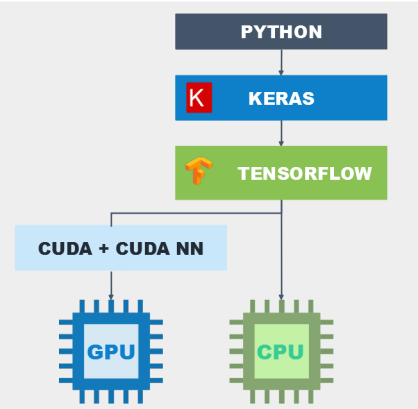

FIGURE 3.1 – Schéma des différentes bibliothèques utilisées selon la cible choisie

3.2 Syntaxe Python et Keras pour la création d'un réseau de neurones

La conception d'un réseau de neurone en python grâce à l'API Keras demande l'utilisation d'une syntaxe particulière, reprenant de manière transparente les étapes de fabrication théoriques de ce type de réseau. Le livre de référence *DEEP LEARNING with Python* a permis d'identifier 5 étapes dans la génération d'un réseau de neurone en python. À cela est ajouté une étape d'utilisation permettant d'utiliser un réseau entraîné pour une application précise. Ces étapes sont résumées dans le tableau 3.2.

Étape	Rôle	Syntaxe Python/Keras
1	Choix du **type de modèle** (ici séquentiel)	network = models.Sequential()
2	Choix des **couches du réseau** (layers) Ajouter autant de layers que nécessaire.	network.add(layers.Dense(output_shape, activation, input_shape))
3	Choix des **méthodes d'apprentissage** : optimiseur, loss function et métrique d'évaluation des résultats	network.compile(optimizer, loss, metrics)
4	**Entrainement** du réseau	network.fit(train_images, train_labels, epochs)
5	**Evaluation** des résultats	network.evaluate(test_images, test_labels)
6	**Utilisation** du réseau (ici reconnaissance du caractère)	network.predict(image)

En bleu les paramètres et données d'entrée à indiquer.

FIGURE 3.2 – Tableau représentant les fonctions à utiliser pour construire une architecture de réseau de neurones

1. Type de réseau : "Séquentiel" désigne un empilement linéaire de couches de neurones (layers). Il est le type de modèle généralement utilisé dans Keras, son alternative étant une classe permettant de créer un modèle plus personnalisé et éventuellement plus complexe.

2. Couches du réseau : chaque layers possède un certain nombre d'entrées et de sorties à indiquer. L'opération spécifiée ici est celle qui, pour chaque noeud, associe à une sortie une ou plusieurs entrées à travers un calcul. La syntaxe proposée ici avec "Dense" suggère que le calcul effectué pour

chaque noeud est le suivant :

$$Output = f_{activation}(W \otimes Input + b)$$

Avec :

1. $Output$ le tenseur de sortie
2. $f_{activation}$ la fonction d'activation
3. $\otimes$ l'opérateur de produit tensoriel
4. W le tenseur des poids des neurones
5. $Input$ le tenseur d'entrée
6. b un tenseur supplémentaire pour l'opération linéaire effectuée avec les tenseurs de poids et d'entrée.

On retrouve dans ce calcul l'opération linéaire effectuée avec les poids (explicitée dans le chapitre 1), et une seconde opération (fonction d'activation). Celle-ci permet d'introduire une non-linéarité indiquant si le neurone est activé ou non, c'est-à-dire si le résultat de son opération sera propagé au neurones suivants.

3. Méthode d'apprentissage Cette étape consiste à choisir la méthode avec laquelle le réseau va mettre à jour les matrices de poids de ses neurones pour "apprendre". En premier lieu, la loss function (fonction de perte) a pour but de calculer une représentation du résulat de la reconnaissance d'une image par le réseau. Plus le résultat en sortie de la loss function est faible, plus il témoigne du bon fonctionnement du réseau. Le but est alors de modifier les poids des neurones de manière à diminuer ce résultat. Cela est réalisé par l'optimizer qui correspond à un calcul mathématique mettant à jour les poids de chaque layer de la façon la plus optimale possible. Différentes fonction "optimizer" et "loss function" seront testées et comparées par la suite.

4. Entraînement du réseau Cette étape constitue l'auto-apprentissage du réseau à reconnaître des images. Pour cela, une banque d'images issue de la bibliothèque MNIST est passée en argument avec leur labels associés (résultats attendus de la reconnaissance). Le paramètre "epoch" indique combien de fois le réseau s'entraîne avec la banque complète (i.e. avec chaque image).

5. Évaluation des résultats L'évaluation du réseau de neurones s'effectue tout simplement en le testant sur une nouvelle banque d'images et de labels associés (issues aussi de MNIST mais différentes de celles d'entraînement). Les paramètres évalués sont ceux spécifiés dans "metrics" de l'étape 3.

6. Évaluation des résultats Il s'agit ici d'utiliser le réseau pour reconnaître 1 image. C'est la fonction qui est utilisée pour développer une application utilisant un réseau entraîné.

3.3 Implémentation CPU GPU

3.3.1 Création d'une première implémentation sur GPU

Dans un premier temps, les différents outils permettant l'écriture d'un code python on été installés sur environnement windows. Il est important de noter que les versions des logiciels et outils sont cruciales pour assurer leur compatibilités.

Dans un second temps, un code pour la reconnaissance des caractères de MNIST est créé à partir du tutoriel du livre. Celui-ci est toute fois modifié pour reprendre l'architecture choisie dans le chapitre précédent concernant l'architecture bas niveau. Une fois le fonctionnement du réseau et la reconnaissance de caractères validée, deux fonctionnalités sont ajoutées pour ajouter des métriques de performance : le chronométrage d'exécution des différentes parties du code à l'aide du module

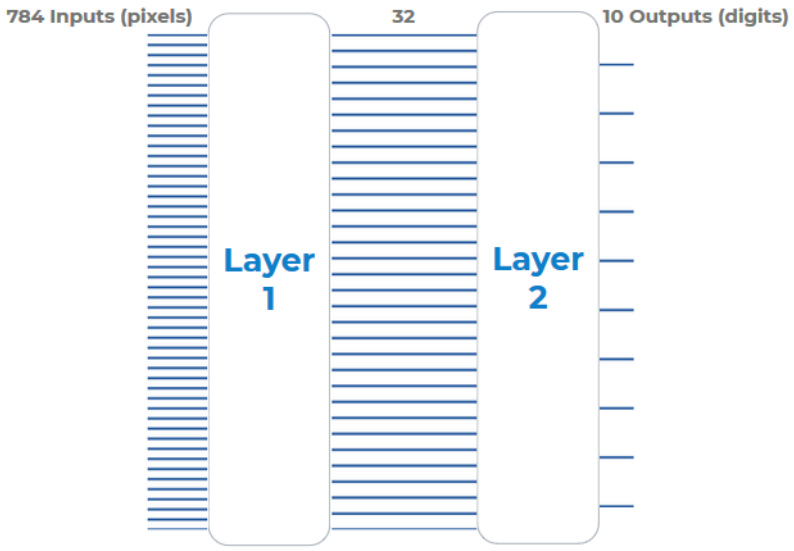

FIGURE 3.3 – Rappel de l'architecture globale du réseau choisie pour la reconnaissance de caractère

"time" de python et l'affichage des caractères à reconnaître à l'aide de matplotlib. A noter que sur la plateforme windows, la fonction time.clock() est la plus précise est sera donc celle utilisée. Pour évaluer sa précision, un code python est utilisé et exécuté après avoir chargé le processeur à 99% et la mémoire à 72%.

Code pour la détermination de la précision de la fonction "time.clock()"

```
 import time as tm
ti = 0
tf = 0
tot = 0
for i in range (0, 1001) :  #1000 fois
    ti = tm.clock()         #sauvegarder la date ti
    tf = tm.clock()         #sauvegarder la date tf
    tot = tot + (tf-ti)     #calculer la différence
print(tot/i)                #afficher moyenne sur 1000
 ↪  lancés
```

Après plusieurs lancés, dans le cas le plus défavorable, la valeur retournée est $3,6.10^{-7}s$. Il ne sera donc pas possible de mesurer une durée inférieure et en majorant, la précision des mesures temporelles est alors fixée à $10^{-6}s$.

A partir du code python décrivant le réseau de neurone, le but est de modifier les différents paramètres de celui-ci pour optimiser son fonctionnement. Les paramètres à faire varier sont : le nombre d'époques, la loss function, l'optimiseur, les layers intermédiaires et le nombre d'images utilisées pour l'entraînement. Il a été choisi de ne pas comparer tous les paramètres possibles à cause de leur grand nombre. De manière à automatiser les tests, deux fonctions sont implémentées :

- **train_and_test(epoch, NB_TEST=100, loss_fct ='binary_crossentropy', opt='rmsprop')** créé puis entraine un réseau selon les paramètres entrés, mesure le temps d'entrainement, le temps d'évaluation et la précision puis teste la reconnaissance de 100 caractères en mesurant là aussi le temps d'exécution.
- **average_test(epoch, NB_TEST=100, loss_fct ='binary_crossentropy' ,opt='rmsprop')**

calcule les résultats moyens dans l'exécution de 3 fois d'affilée de `train_and_test`.
Le code disponible est fourni en annexe.

3.3.2 Adaptation pour exécution sur GPU

Si le code développé précédent est exécutable sur GPU, quelques modifications ont du être apportées.

Premièrement, Tensorflow est remplacé par Tensorflow-GPU. Deuxièmement, CUDA et CUDNN sont installés. Cette partie a été très chronophage à cause de la non-compatibilité de certaine version de CUDA avec le matériel ainsi que les autres logiciels utilisés. Finalement le code est fonctionnel sur GPU et est identique au code CPU pour permettre une comparaison la plus pertinente possible. Troisièmement, le logiciel Nvidia Profiler (fourni lors de l'installation de CUDA et de ses librairies) est pris en main. Les différentes expérimentations sont alors reconduites sur GPU et les performances étudiées dans la partie suivante.

3.4 Résultats et analyse des performances

Dans cette section, les performances évaluées correspondent au temps d'entraînement du réseau, au temps de reconnaissance d'un caractère et à la précision de reconnaissance (pourcentage de caractères correctement détectés). Par soucis de temps, seulement deux optimizers (rmsprop et adam) seront comparés.

3.4.1 Résultats de la reconnaissance de caractères

Observons tout d'abord un résultat de reconnaissance de caractère 3.4.1. La reconnaissance a été effectuée ici avec les paramètres suivants pour lesquels la précision obtenue est de 95% : layers 32+10, rmsprop, 5 époques, categorical_crossentropy et l'ensemble des images de MNIST pour l'entraînement. Il peut être constaté tout de suite que le chiffre 2 présent sur l'image est effectivement reconnu en sortie par le réseau.

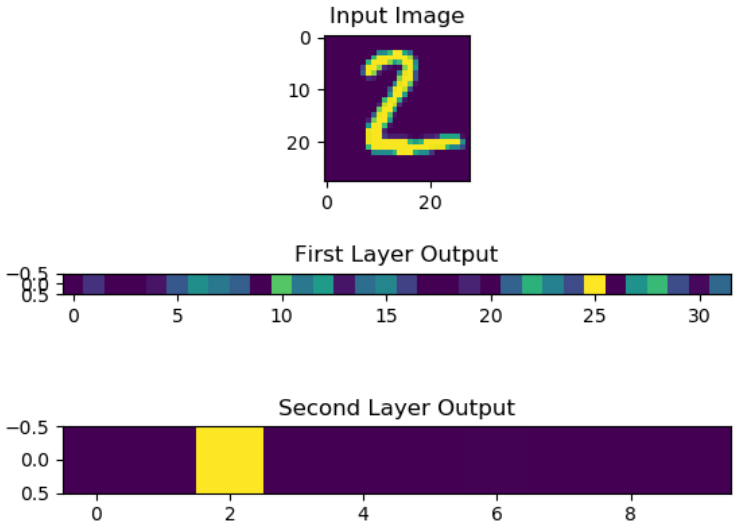

FIGURE 3.4 – Entrée, sortie et layer intermédiaire du réseau lorsque celui-ci est correctement entraîné.

Il est à noter cependant que les résultats du layer intermédiaire ne sont pas "interprétables". En effet, lors de l'entraînement, le réseau développe sa propre "méthode de reconnaissance" grâce à l'optimizer et la loss function. Le résultat obtenu ici est donc un calcul intermédiaire déterminé par le réseau comme étant le plus optimal pour reconnaître les caractères. Ce résultat est seulement utilisable et compréhensible par la machine.

Comparons cela avec un réseau mal entraîné. Celui-ci est réalisé à partir du précédent en utilisant cette fois, seulement 100 images pour l'entraînement et 1 seule époque. Dans ce cas, la précision retenue mesurée est de 18%. Il est alors évident que le réseau reconnaît mal le caractère (ici un 3 au lieu d'un 4), mais il peut être observé que celui-ci "hésite". En effet, le tableau de sortie indique les probabilités que l'image d'entrée soit l'un des 10 chiffres. Lorsque le réseau est correctement entraîné, seul le nombre détecté possède une probabilité forte en sortie, elle est nulle pour les autres.

L'ensemble des mesures ayant pu être effectuées sont listés dans le tableau fourni en annexe pour une question de lisibilité. Dans les parties suivantes, nous analyserons ces résultats dans le détail.

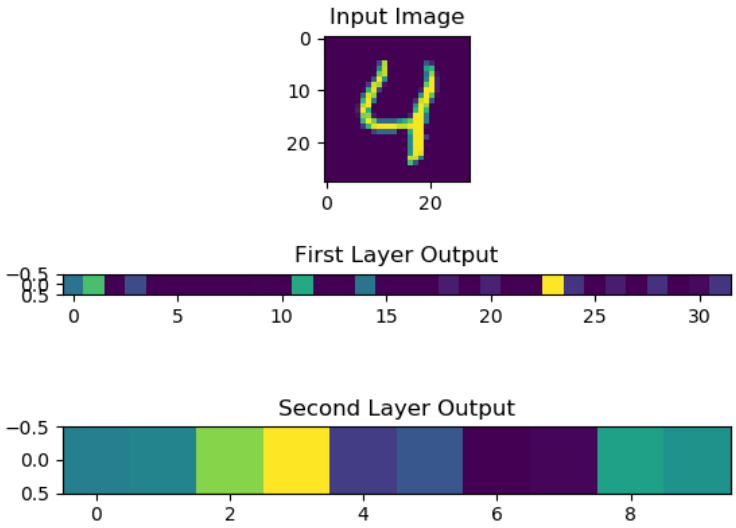

FIGURE 3.5 – Entrée, sortie et layer intermédiaire du réseau lorsque celui-ci est mal entraîné.

3.4.2 Performances temporelles et précision

Plusieurs paramètres influencent les temps d'entraînement du réseau et de reconnaissance des caractères.

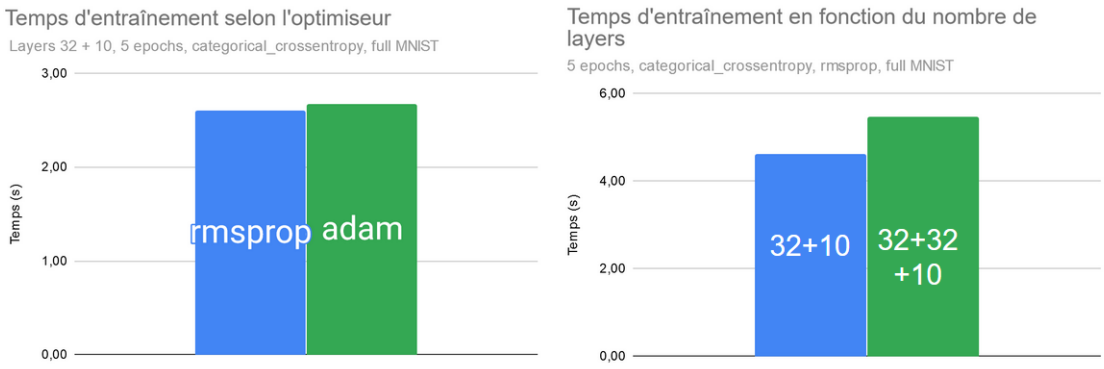

FIGURE 3.6 – Évolution du temps d'entraînement selon les paramètres.

Première remarque pouvant être faite à partir du tableau donné en annexe, le temps d'entraînement augmente en fonction du nombre d'époques (évident car on entraîne plusieurs fois le réseau avec les mêmes données). Le type d'optimizer choisi peut aussi ralentir le temps d'entraînement. Par exemple avec rmsprop, l'entraînement est plus rapide qu'avec l'optimizer adam. L'architecture choisie est aussi déterminante : en augmentant le nombre de layers utilisés, l'entraînement est plus long.

De plus, par rapport à l'optimizer, la précision atteinte par rmsprop est plus élevée que pour adam. Il serait donc plus intéressant d'utiliser le premier, sauf dans des cas de réseaux plus complexes où adam est en théorie plus adapté. De même, un nombre plus grand de layers intermédiaires

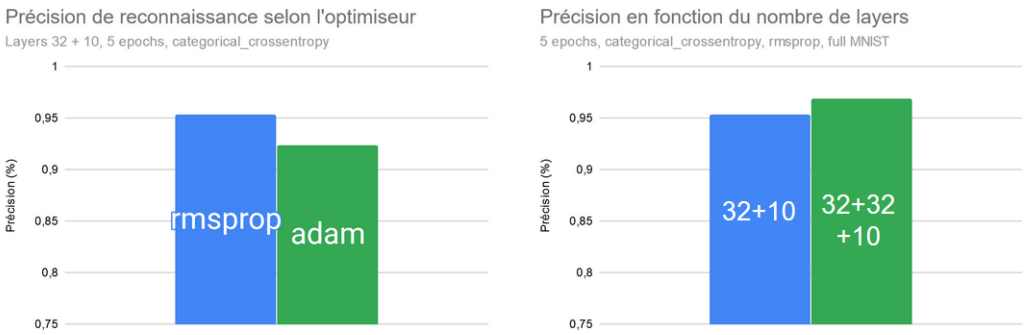

FIGURE 3.7 – Évolution de la précision selon les paramètres.

augmente la précision de la reconnaissance au détriment comme nous l'avons vu de la rapidité.

Deux loss functions ont pu aussi être comparées : binary_crossentropy et categorical_crossentropy. On observe alors que la première permet d'obtenir une précision plus importante que la seconde mais avec un temps d'entraînement plus long. Cependant, l'utilisation de binary_crossentropy est normalement réservée à des problème binaire lorsque categorical_crossentropy est utilisée pour des problèmes de catégorisation (comme ici). L'utilisation de binary_crossentropy n'est donc en théorie pas appropriée pour notre application malgré des résultats convaincants.

Au niveau du temps de reconnaissance des caractères après entraînement, celui-ci n'est pas significativement impacté par les paramètres choisis. Les valeurs obtenus sur CPU et GPU étant bien différentes, l'ajustement des paramètres n'engendre pas de résultats pertinents. Ces temps sont de 0,080s environ sur GPU et de 0,050s environ sur CPU. Ainsi, dans le meilleur des cas mesurés (layers 32+10, rmsprop, categorical_crossentropy, sans compter le cas utilisant binary_crossentropy), le débit obtenu est de **1923 images/s reconnues**. L'étape d'entraînement est donc à priori celle qui est la plus déterminante dans l'optimisation des performances d'un réseau de neurones.

3.4.3 CPU vs GPU

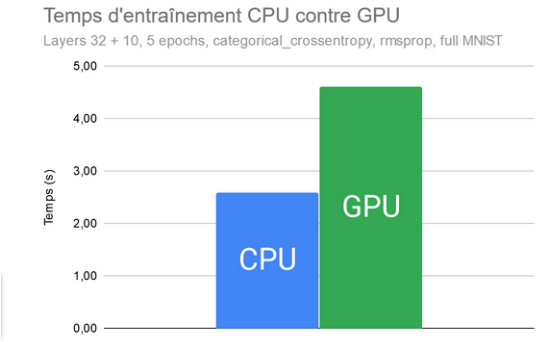

FIGURE 3.8 – Temps d'entraînement sur CPU et GPU (précisions mesurées identiques à 0,001% près, valant 0,953% environ)

Au niveau de la comparaison CPU et GPU, nous constatons que le réseau implémenté sur CPU est bien plus rapide que sur GPU (gain de 56% de vitesse d'entraînement). Il est donc décidé d'étudier plus en détail l'implémentation sur GPU avec l'outil **Nvidia Visual Profiler** proposé par le fabriquant.

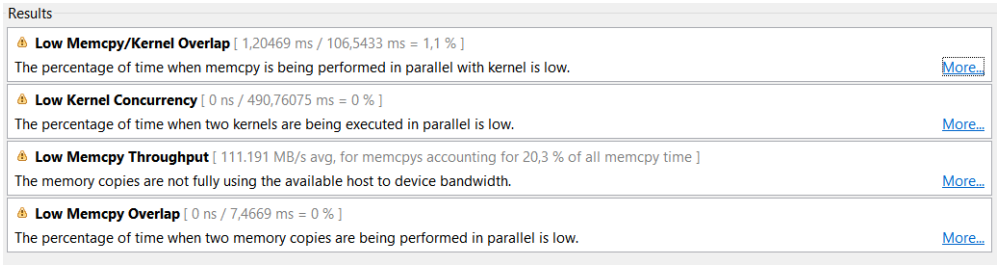

FIGURE 3.9 – Résultat de l'analyse avec Nvidia Visual Profiler de l'architecture du réseau

Nous pouvons alors analyser cette différence par la non optimisation de la solution GPU. Selon Nvidia Visual Profiler, les ressources du GPU ne sont pas utilisée de manière efficace. De plus, le calcul sur GPU demande la copie des valeurs et données depuis l'ordinateur vers celui-ci ce qui pénalise beaucoup les performances pour des calculs relativement faibles comme ici (la mesure du temps de copie des informations n'a pu être faite à cause de la complexité de l'étude des résultats fournis par Nvidia Visual Profiler). Cette implémentation sur GPU serait donc plus pertinente dans le cas de réseaux plus complexes et d'architecture plus importantes.

3.4.4 Consommation

Finalement, il est possible d'évaluer la consommation énergétique du réseau implémenté sur GPU.

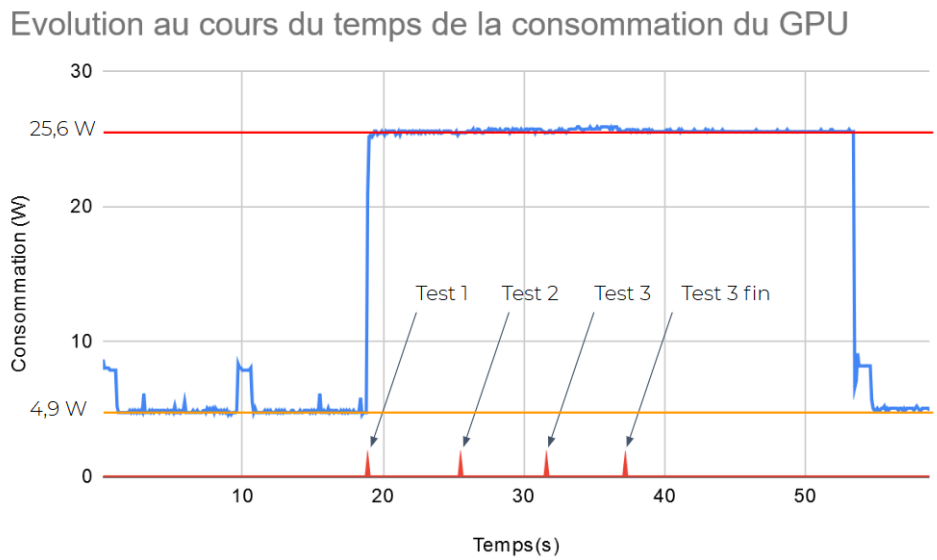

FIGURE 3.10 – Mesure de consommation de puissance électrique par le GPU

La consommation atteinte à pour but d'être comparée avec l'implémentation sur FPGA. A l'aide de Nvidia Visual Profiler mesurant un point toutes des 100 ms, et en exécutant un algorithme de reconnaissance d'image 3 fois à la suite avec le réseau le plus performant décrit précédemment, nous observons deux paliers de consommation : au repos le GPU consomme 4.9W et en exécution 25.6W.

3.5 Conclusion

Pour conclure ce chapitre, nous avons pu mettre en oeuvre différentes solutions sur CPU et GPU pour évaluer les performances et les optimisations à apporter pour obtenir le meilleur compromis suivant l'application et les besoins visés (vitesse contre précision).

Comparaison des performances en fonction des différents paramètres

Layers	Processing Unit	Loss function	Optimizer	Epochs	Training Time (s)	nizing 100 imag	Accuracy (%)
32 + 10	CPU	binary_crossentropy	rmsprop	5	2,75	0,05	0,9912
32 + 10	CPU	binary_crossentropy	rmsprop	1	0,78	0,05	0,9857
32 + 10	CPU	categorical_crossentropy	rmsprop	5	2,60	0,052	0,9531
32 + 10	CPU	categorical_crossentropy	rmsprop	4	2,03	0,049	0,9506
32 + 10	CPU	categorical_crossentropy	rmsprop	3	1,56	0,053	0,9459
32 + 10	CPU	categorical_crossentropy	rmsprop	2	1,16	0,052	0,9392
32 + 10	CPU	categorical_crossentropy	rmsprop	1	0,72	0,052	0,9217
32 + 10	CPU	categorical_crossentropy	adam	1	0,74	0,058	0,9239
32 + 10	CPU	categorical_crossentropy	adam 0,001 0,9 0,999	5	2,67	0,054	0,9545
32 + 10	GPU	binary_crossentropy	rmsprop	5	5,07	0,08	0,991
32 + 10	GPU	binary_crossentropy	rmsprop	1	1,24	0,078	0,9852
32 + 10	GPU	categorical_crossentropy	rmsprop	5	4,61	0,081	0,9534
32 + 10	GPU	categorical_crossentropy	rmsprop	4	3,78	0,08	0,9499
32 + 10	GPU	categorical_crossentropy	rmsprop	3	2,83	0,079	0,9459
32 + 10	GPU	categorical_crossentropy	rmsprop	2	1,92	0,082	0,9375
32 + 10	GPU	categorical_crossentropy	rmsprop	1	1,06	0,08	0,9255
32 + 32 +10	GPU	categorical_crossentropy	rmsprop	5	5,47	0,089	0,9694
32 + 32 +10	GPU	categorical_crossentropy	rmsprop	1	1,12	0,082	0,9244
32 + 10	GPU	categorical_crossentropy	adam 0,001 0,9 0,999	NA	NA	NA	NA
Next measures done with 1/5 of MNIST dataset (12000 pictures) to train the network							
32 + 10	GPU	categorical_crossentropy	rmsprop	40	7,74	0,09	0,9424
32 + 10	GPU	categorical_crossentropy	rmsprop	30	5,83	0,089	0,9408
32 + 10	GPU	categorical_crossentropy	rmsprop	20	3,97	0,097	0,9345
32 + 10	GPU	categorical_crossentropy	rmsprop	10	2,07	0,087	0,9304
32 + 10	GPU	categorical_crossentropy	rmsprop	9	2,06	0,0866	0,9284
32 + 10	GPU	categorical_crossentropy	rmsprop	8	1,74	0,09	0,9268
32 + 10	GPU	categorical_crossentropy	rmsprop	7	1,52	0,086	0,9259
32 + 10	GPU	categorical_crossentropy	rmsprop	6	1,45	0,0839	0,9269
32 + 10	GPU	categorical_crossentropy	rmsprop	5	1,11	0,084	0,9186
32 + 10	GPU	categorical_crossentropy	rmsprop	4	1,05	0,0882	0,9132
32 + 10	GPU	categorical_crossentropy	rmsprop	3	0,73	0,087	0,9089
32 + 10	GPU	categorical_crossentropy	rmsprop	2	0,59	0,092	0,8998
32 + 10	GPU	categorical_crossentropy	rmsprop	1	0,39	0,085	0,8764

FIGURE 3.11 – Tableau récapitulatif de tous les résultats et mesures effectuées lors de l'étude sur CPU/GPU

Partie 4

Architecture FPGA : HLS4ML

4.1 Introduction à HLS4ML

Après l'utilisation du CPU et GPU pour implémenter le réseau de neurones. L'objectif dans cette partie est d'effectuer l'implémentation sur FPGA.

Pour les implémentations des réseaux de neurones, une cible FPGA est un choix pratique. Implémenter un réseau de neurones nécessite l'utilisation de plusieurs opérations mathématiques et logique. Une cible FPGA permet de réaliser ces opérations grâce à ses unités de calcul.

Pour réaliser cette implémentation, on utilise la même base de données, librairie et Backend que ceux utilisés dans la partie 3. Cependant on a besoin d'un outil pour effectuer la conversion à un langage de type HDL. Puisque notre application s'inscrit dans le domaine du Deep Learning, on utilise l'outil HLS4ML qui permet d'effectuer la conversion du modèle Keras du réseau de neurones à un modèle en SystemC.La figure suivante montre les étapes de l'implémentation sur FPGA.

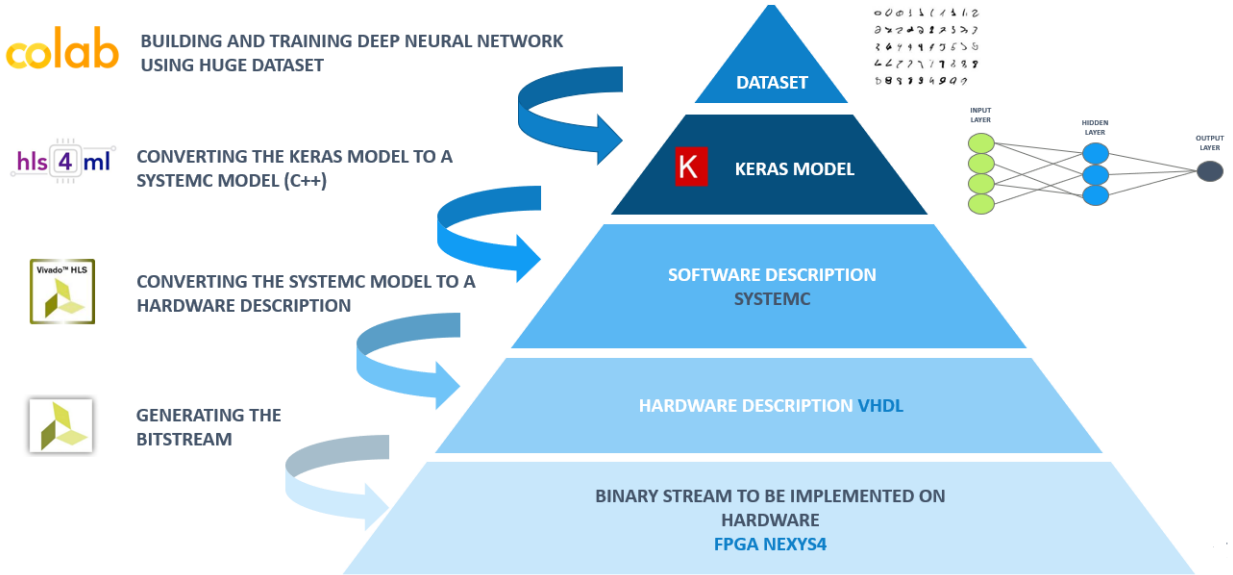

FIGURE 4.1 – Etapes de l'implémentation sur FPGA

Plus précisément, on construit le modèle du réseau de neurone en utilisant l'outil de Google Colab. On utilise la base de données MNIST. Ensuite on récupère après exécution les deux fichiers de l'architecture du modèle et des poids sauvegardés dans des fichiers de type : json et h5. On accède au fichier de configuration de HLS4ML, on intègre les fichiers précédemment générés et on effectue les choix nécessaires pour la configuration. On génére la description software en SystemC

avec HLS4ML. Ensuite on effectue la conversion en description hardware en VHDL. On génére le bitstream et on implémente sur FPGA. Notre cible est NEXYS4.

Cependant, le résultat qu'on a récuppérer ne suffit pas pour effectuer une implémentation directe sur FPGA après conversion en VHDL. On doit ajouter quelques blocs comme démontré dans la figure 4.2. Le design complet se compose de :

- **Wrapper UART** : qui permet d'effectuer la mise en série des représentations des pixels des images de la bases de données pour présenter des données compatibles avec le réseau de neurones. La carte dispose d'un nombre d'entrée limités d'où la nécessité de ce bloc.
- **Bloc HLS4ML** ce bloc représente le résultat en VHDL du modèle généré par l'outil HLS4ML.
- **Sélection** ce bloc permet de déterminer sortie parmi les 10 possibles avec la probabilité maximales. Il permet de déterminer le chiffre qu'on a prédit à partir de l'image d'entrée.
- **Afficheur sept segments** L'objectif de ce bloc est d'afficher le chiffre prédit en utilisant l'afficheur sept segments.

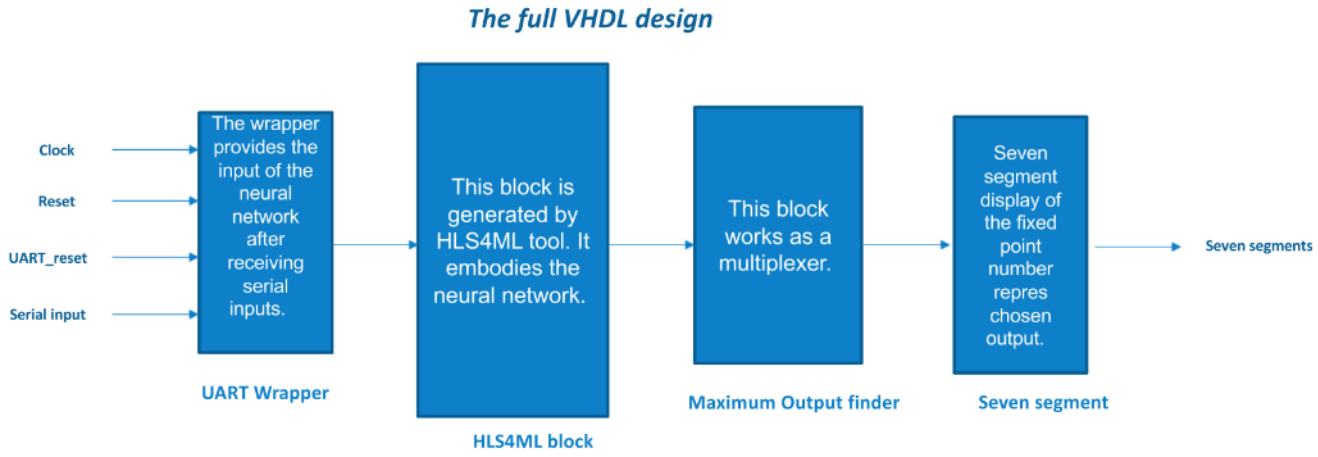

FIGURE 4.2 – Architecture VHDL

4.2 Evaluation des résultats

Après avoir introduit proprement l'outil *HLS4ML*, nous allons l'utiliser pour implémenter le réseau de neurones lié à MNIST sur la carte FPGA *NEXYS4 A7*. Premièrement, nous avons utilisé Keras pour concevoir un réseau de neurones minimaliste afin que son implémentation sur FPGA soit possible en terme de ressources matérielles utilisées. En effet, ce réseau se compose d'une seule couche ayant 10 neurones correspondant aux prédictions de chacun des 10 chiffres à reconnaître. En effet, Ce modèle permet d'avoir une précision de 28%. Ensuite, nous utilisons l'outil *HLS4ML* pour transformer le modèle Keras à un modèle en *SystemC*. Nous avons choisi les paramètres suivants pour la transformer réalisé par HLS4ML :

- IOType : io_parallel. Nous envoyons nos données en parallèle à travers le module *UART WRAPPER* introduit précédemment.

- Precision : ap_fixed<16,6>. Une précision de 16 bits pour coder en virgules fixes, dont 10 bits et 6 bits pour coder respectivement la partie décimale et la partie entière. Cette précision a

permis à la carte FPGA de fournir les mêmes résultats en sortie que le modèle Keras.

— ReuseFactor : 2. Nous avons remarqué que plus nous augmentons la valeur du ReuseFactor, moins on consomme les ressources matérielles du FPGA. Ceci dit, le temps de la synthèse HLS augmente considérablement avec l'augmentation de la valeur du ReuseFactor choisie.

— Strategy : Resource. Comme les ressources matérielles de la carte *NEXYS 4 A7* sont relativement très limitées par rapport à notre problèmes de la détection de caractères, nous avons choisi de faire une optimisation en ressource et non pas en latence.

— Compression : True. En activant la compression, l'outil *HLS4ML* détectera et éliminera les neurones ayant un faible impact sur la sortie du réseau de neurones à implémenter.

Après avoir appliqué la transformation du modèle Keras au modèle en SystemC en utilisant les paramètres ci-dessus avec l'outil *HLS4ML*, nous avons réalisé la synthèse HLS pour transformer le modele en SystemC en une description matérielle en *VHDL*. Ensuite, nous avons réalisé la synhtèse VHDL, l'implémentation et la génération du *bitstream* du projet VHDL issu de la synthèse *HLS*. En effet, à la fin de l'implémentation nous avons obtenu le rapport d'implémentation suivant :

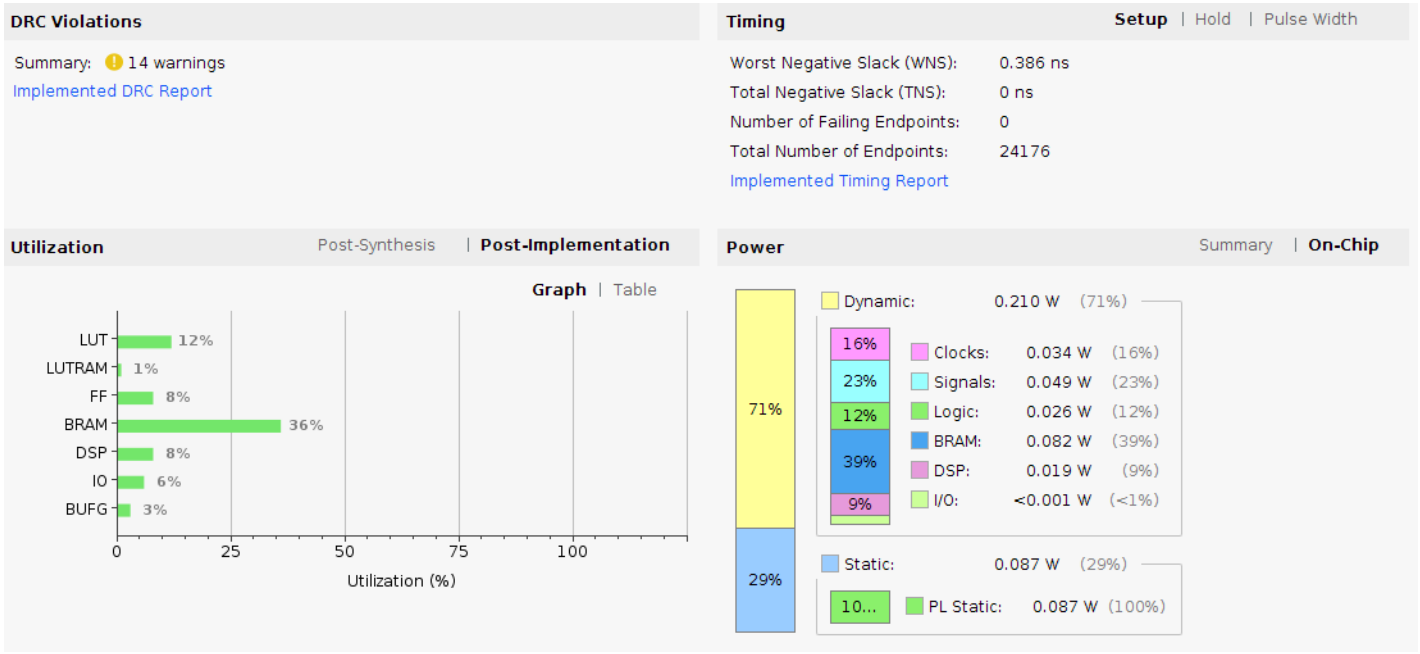

FIGURE 4.3 – Résultat de l'implémentation VHDL

Nous constatons que le réseau de neurones implémenté utilise raisonnablement les ressources matérielles disponibles dans la carte NEXYS4 A7. De plus, le chemin critique est bien correct. Ceci dit, l'implémentation échoue lorsque nous essayons d'ajouter une couche supplémentaire dans notre réseau de neurones. Ceci fait partie de nos futurs challenges.

Afin de tester le bon fonctionnement de notre implémentation sur FPGA. Nous avons développé un script en Python pour lire la base de données MNIST et d'envoyer une des images de cette base de données via la communication série UART. Le script Python est illustré et commenté ci-dessous :

```python
from numpy import loadtxt       #chargement les données à traiter
import matplotlib.pyplot as plt #affichage l'image à traiter
import serial                   #gestion de la communication en série
import math                     #réalisation de la transformation en virgule fixe
import codecs                   #encodage et transformation binaires des données
import sys                      #pour lire l'arugment passé en ligne de commande

# Indice de l'image dans la base de données
im_idx = int(sys.argv[1])

# nombre de bits pour coder la partie décimale dans la représentation en virgule fixe
fixed_point_q = 10

################### Fonctions utiles ##################
def uart_tx_im(neural_network_input):
    # transformation les valeurs flottantes en des valeurs en virgules fixes
    input_decimal = [math.ceil(i * 2**fixed_point_q) for i in neural_network_input]
    # inversement de l'octet le plus fort avec l'octet le plus faible pour chaque mot de 16 bits
    input_decimal = [((i << 8) | (i >> 8)) & 0xFFFF for i in input_decimal]
    # transformation en une représentation binaire
    input_2bytes =  [i.to_bytes(2, 'big') for i in input_decimal]
    # envoie des données octet par octet
    for i in input_2bytes :
        ser.write(i)

#########################################################
# établissement de la connexion avec le module UART de la carte FPGA
ser = serial.Serial("/dev/ttyUSB1", 115200)

# chargement des images d'entrées et les sorties correspondantes
input_images     = loadtxt('inputs.csv', delimiter=',')
expected_outputs = loadtxt('outputs_indexes.csv', delimiter=',')

# choix de l'image à envoyer
neural_network_input = input_images[im_idx]
expected_output      = expected_outputs[im_idx]

# envoie de l'image au module UART de la carte FPGA
print("SENDING THE IMAGE TO THE FPGA ...")
uart_tx_im(neural_network_input)
print("DONE!")

# affichage du résultat théorique
print("EXPECTED DIGIT :", expected_output)

# fin du programme
print("EXIT SUCCESSFULLY")
```

En exécutant ce script python, les images se sont envoyées à la carte FPGA avec succès. Nous avons pu observer, à chaque traitement d'image, le chiffre prédit par le réseau de neurones implémenté sur FPGA sur son afficheur 7 segments. Après la réalisation de quelques tests sur les images MNIST, les résultats prédits par la carte PFGA ont été les mêmes que les résultats théoriques. Cependant, le temps ne nous a pas permis de faire un test automatisé sur toute la base de données MNIST pour évaluer la précision exacte du modèle implémenté.

Partie 5

Conclusion

L'implémentation de réseau de neurones sur CPU, GPU et FPGA a été un projet très pertinent, enrichissant et instructif. Ce projet nous a permis d'apprendre et d'exploiter les concepts de base du *Deep Learning*. Nous avons eu également l'occasion d'approfondir nos connaissances théoriques et pratiques sur les APIs et langages suivants : *C++, SystemC, Python, VHDL et CUDA*. En effet, notre tentative de coder notre architecture du réseau neuronal en utilisant uniquement la bibliothèque standard du C++, sans avoir recours à aucune bibliothèque spécialisée en accélération de calcul ou bibliothèque externe, a été une bonne occasion pour comprendre la théorie mathématique d'un réseau de neurones, notamment les équations mathématiques mises en oeuvres. De plus, pendant l'implémentation CPU/GPU, nous avons exploité les outils incontournables dans le *Deep Learning* : *TensorFlow, Keras et CUDA* ; cette phase a été très enrichissante et productives. Finalement, l'implémentation sur FPGA a nécessité l'utilisation du *firmware HLS4ML* qui a été un allié robuste pour transformer nos modèles Keras en des modèles HLS, qui sont synthétisables avec *Vivado HLS*. Pour conclure, cette expérience de reconnaissance de caractères avec réseau de neurones sur CPU, GPU et FPGA a aboutit vers des résultats prometteurs, et nous a été très instructive.

Partie 6

Bibliography

1. ARTIFICIAL INTELLIGENCE : DEFINITION, TRENDS, TECHNIQUES, AND CASES Joost N. Kok, Egbert J. W. Boers, Walter A. Kosters, and Peter van der Putten Leiden Institute of Advanced Computer Sc ience, Leiden University, the Netherlands
2. HLS4ML site https ://fastmachinelearning.org