Rapport

Implantation de réseaux de neurones sur cibles matérielles et logicielles

Élèves :
Omar Dahmani
Paiul Nicolae
François VANLERBERGHE

Enseignant :
Bertrand.Legal
Jérémie.crenne

Projet avancé S9

Bordeaux INP
ENSEIRB
MATMECA

Table des matières

1 Introduction

Avec l'avènement de l'intelligence artificielle, et l'augmentation des capacités technologiques, nombreuses sont les recherches tentant de simuler le comportement humain et également de reproduire ses capacités cognitives. Dans ce contexte, le **Deep learning** est l'un des axes de recherche les plus explorés.

Dans ce projet, on a essayé d'implémenter un réseau de neurones convolutifs sur des cibles matérielles et logicielles. Le but derrière cette implémentation est la reconnaissance de chiffres manuscrits.

2 Objectifs

Pour atteindre l'objectif final du projet, nous allons essayer dans un premier temps d'introduire la notion de réseau de neurones, puis nous allons détailler les différentes étapes de la création et de l'implémentation d'un réseau de neurones sur CPU sous Python.

Dans un deuxième temps, nous allons utiliser l'outil **hls4ml** pour convertir le code python en code VHDL, afin de pouvoir l'implémenter sur une carte FPGA.

3 Construction du réseau de neurones

Pour comprendre le Deep learning et plus précisément les réseaux de neurones. Il ne suffit pas de s'intéresser aux aspects technologiques et algorithmiques, mais de commencer par analyser le côté humain. Car souvent l'homme a tendance à chercher à se reproduire et simuler les différentes qualités qui le différent des autres espèces.

En effet, le Deep learning vise à simuler le cerveau humain par des procédés informatiques. Ainsi un réseau de neurones est un système inspiré du fonctionnement des neurones biologiques.

3.1 Définition d'un réseau de neurones

Tout d'abord, un neurone est une fonction algébrique non linéaire et bornée, dont la valeur dépend de paramètres appelés coefficients ou poids.

L'objectif étant de prédire une sortie (**Y**) à travers un ensemble de données (**X1**,**X2**) (*cf figure2*). Un des moyens d'y arriver est de simuler la réponse d'un neurone "*artificiel*" et donc mettre au point un algorithme permettant de traiter et pondérer les observations pour en prédire une caractéristique.

En effet, un neurone "*biologique*" rempli la même fonction en analysant les différentes entrées du système nerveux et les traduit par la suite en impulsions musculaires responsables des différentes réactions.

Les figures ci-dessous montrent la similarité entre un neurone artificiel et un neurone biologique au niveau de leurs structures. Pour un neurone artificiel, les dendrites reçoivent des signaux et la fréquence de chaque signal dépend du poids Wi. Ces signaux sont ensuite transmis vers le corps de la cellule. Cette dernière est caractérisée par deux éléments importants : la fonction et la condition d'activation. C'est en vérifiant la condition d'activation que l'axone peut émettre à nouveau des signaux aux neurones qui suivent, à travers leurs dendrites, et ainsi de suite.

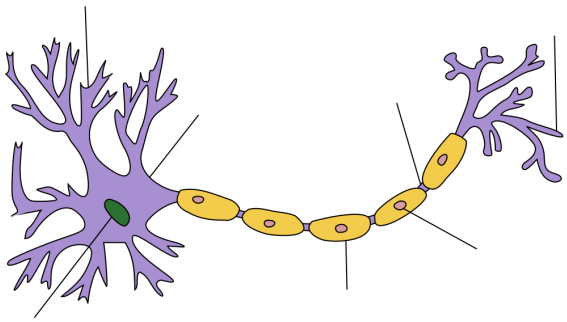

FIGURE 1 – Modèle biologique d'un neurone

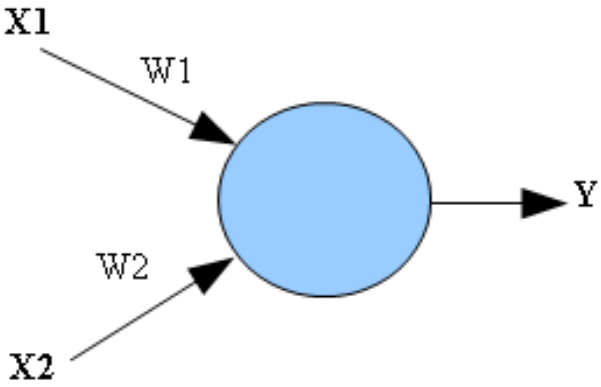

FIGURE 2 – Modèle artificiel d'un neurone

Maintenant qu'on sait définir un neurone, on peut définir un réseau de neurones comme plusieurs neurones inter-connectés grâce à des liens pondérés (Wi), où chaque noeud est caractérisé par l'ensemble [Fi (fonction d'activation)+Ci (condition d'activation)], le but étant de trouver la combinaison optimale des Wi/Fi/Ci pour avoir les meilleurs résultats en terme de prédictions.

3.2 ANN : Création d'un réseau de neurones artificiels

Dans cette partie, nous allons voir comment construire le réseau de neurones le plus simple : le réseau de neurones artificiels (**ANN**). La figure ci-dessous illustre l'architecture d'un réseau de neurones artificiels :

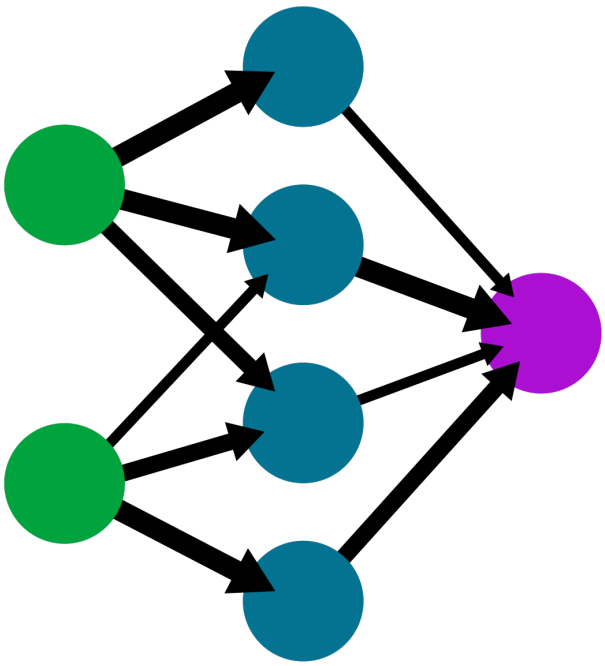

FIGURE 3 – Modèle simplifié d'un réseau de neurones artificiels

On remarque que le réseau est constitué de trois couches :

— **La couche d'entrée** : c'est la couche qui fait le lien entre chacune des données à traiter par le réseau de neurones et les couches cachées.

— **La couche cachée** : Elle est nommée ainsi car elle n'a pas de contacts directs avec l'extérieur. Les fonctions d'activations, ainsi que la taille de la couche, peuvent varier en fonctions des besoins, afin d'avoir de bons résultats

— **La couche de sortie** : Cette couche dépend du nombre de valeurs que nous souhaitons prédire. Dans notre cas, 10 valeurs de chiffres manuscrits. Cette couche contient également une fonction d'activation pour obtenir la probabilité d'appartenance des données d'entrée au groupe.

3.3 CNN : Création d'un réseau de neurones convolutifs

Il existe aujourd'hui de nombreux réseaux de neurones artificiels différents. Nous nous intéressons particulièrement à une sous-catégorie : les réseaux de neurones convolutifs.
Les réseaux de neurones convolutifs sont particulièrement efficaces pour la reconnaissance et la classification d'images. Ils ont été inspirés par le cortex visuel des animaux et repose sur le pré-traitement de petites quantités d'informations. Tout comme les autres réseaux ils sont composés de plusieurs couches ayant des rôles différents. Les couches de notre réseau peuvent être décomposées en 2 blocs. Le premier bloc est le bloc convolutif. Il est celui qui caractérise ce type de réseau. En effet ce bloc effectue des opérations de filtrage par convolution pour extraire les features. Il est composé de 3 types de couches différentes se succédant :

— **La couche de convolution** : Son but est de repérer la présence de feature sur l'image. Pour cela on réalise un filtrage en faisant glisser une fenêtre représentant la feature sur l'image en réalisant le produit de convolution de cette feature avec chaque portion de l'image. Ainsi la reconnaissance n'est pas sensible à la localisation des éléments sur l'image et reste efficace en cas de glissement. On obtient en sortie de chacun de ces filtres ce qu'on appelle une feature map représentant où se situe les features recherché sur l'image.

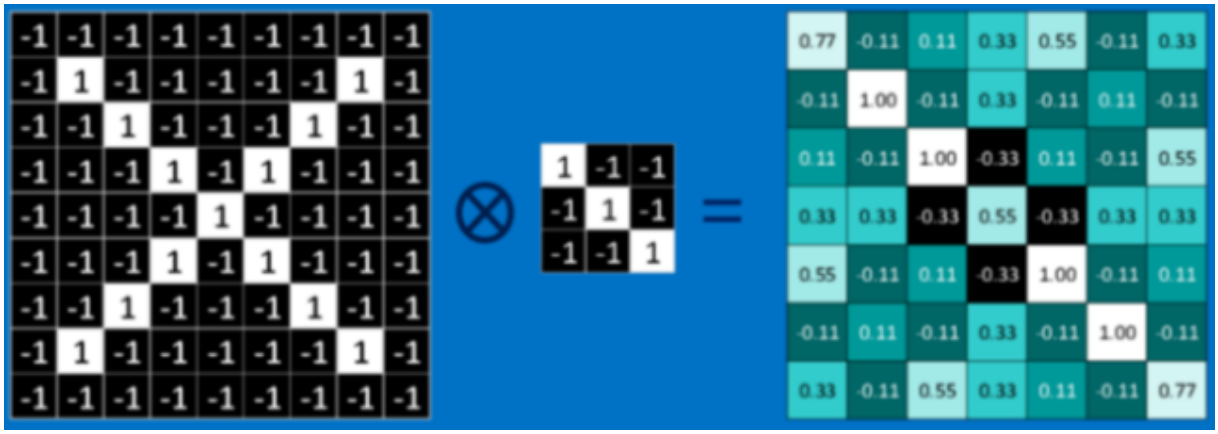

FIGURE 4 – Schéma de fonctionnement de la couche de convolution

— **La couche de correction** : ReLU (Rectified Linear Units) : Son rôle est d'éliminer les valeurs négatives en plaçant les valeur à 0 si nécessaire.

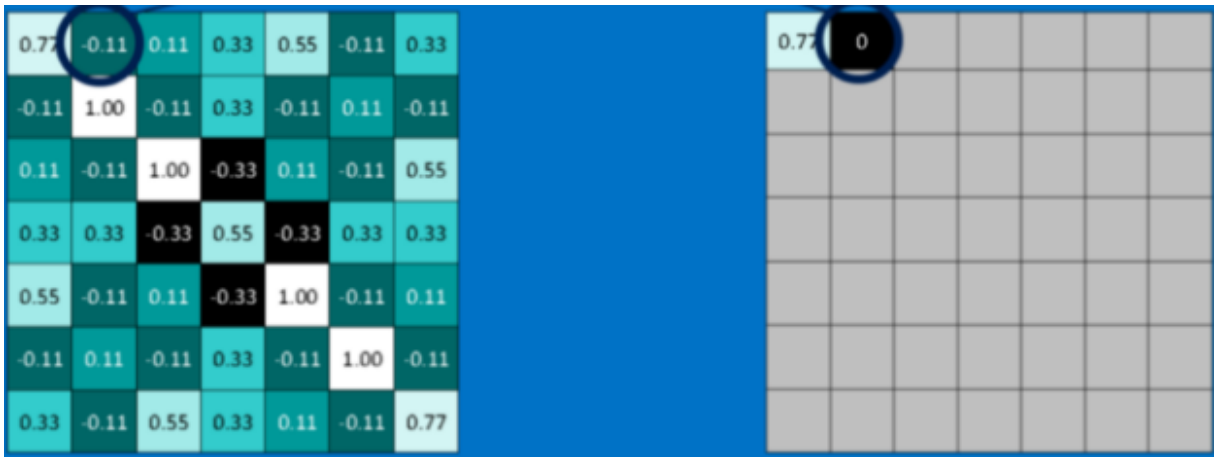

FIGURE 5 – Schéma de fonctionnement de la couche ReLU

— **La couche de pooling** : Le nombre de calcul augmente très rapidement avec la taille de l'image et le nombre de neurones, rendant les calculs lourds. Le rôle de la couche pooling est de réduire la taille des features en ne conservant que les caractéristiques les plus importantes. L'image est donc découpé en carré de 4 ou 9 pixels dont ne garde que le plus important. On obtient ainsi une images 4 ou 9 fois plus petite.

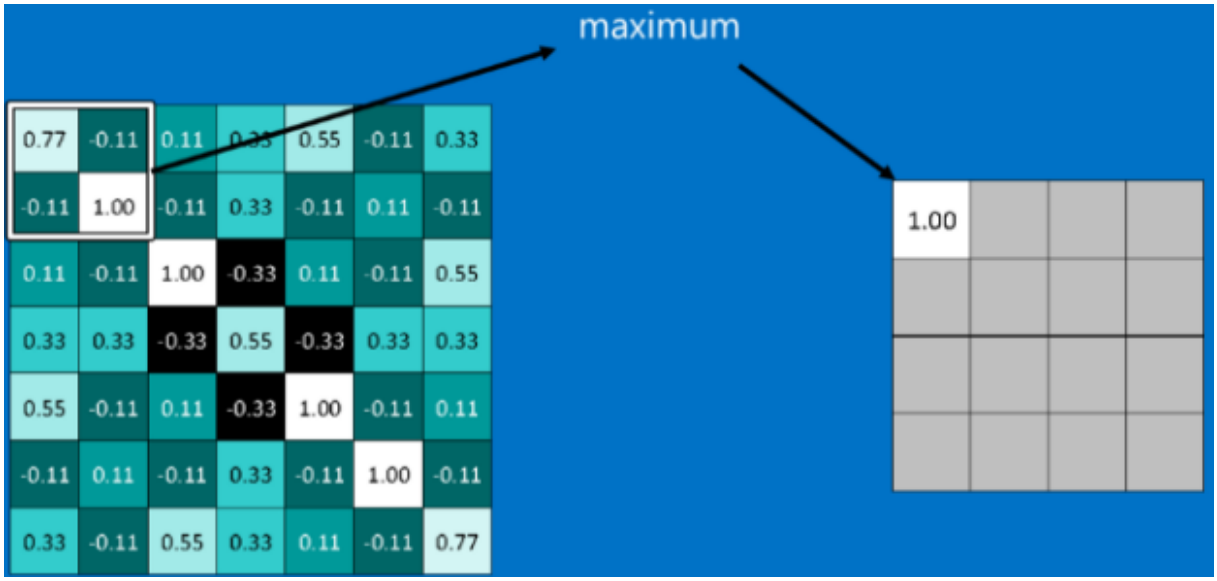

FIGURE 6 – Schéma de fonctionnement de la couche de pooling

Le second bloc est présent dans tout les types de réseaux de neurones et est appelé couche fully-connected. Son rôle est de classifier les images en renvoyant en sortie un vecteur donnant la probabilité que l'image appartienne à telles ou telles catégories. Pour cela, la couche applique une combinaison linéaire sur les valeurs reçu en entrée en tenant des différents poids.

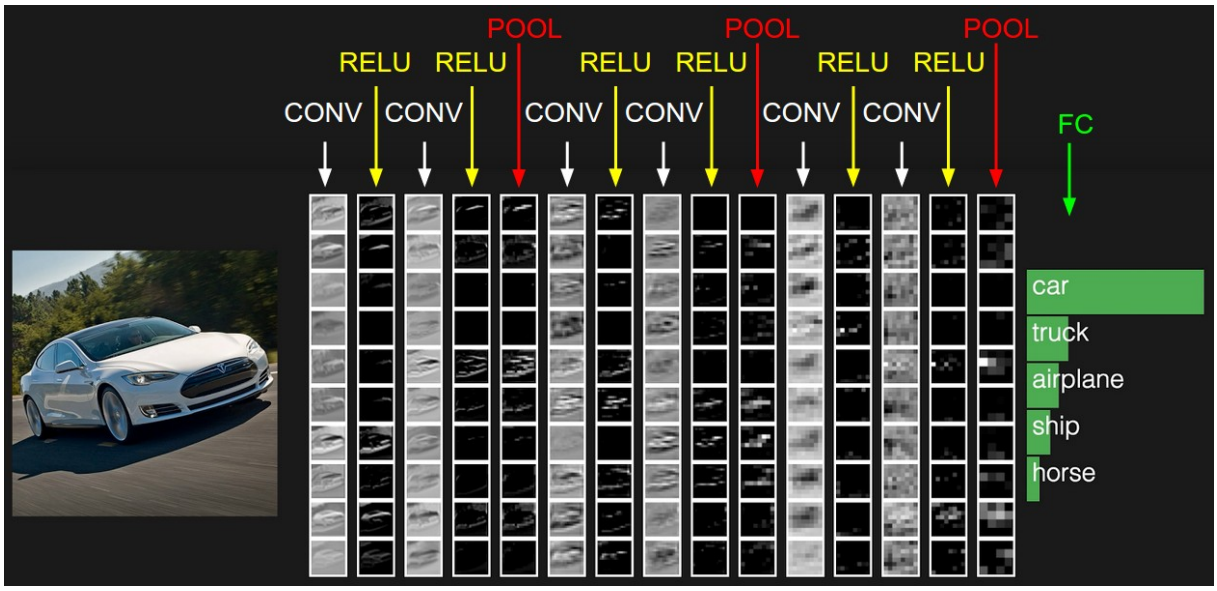

FIGURE 7 – Exemple de fonctionnement d'un réseau de neurone convolutif

Dans cet exemple nous pouvons voir qu'après une succession de filtres l'image d'entrée est presque certainement une voiture. Ce type de réseau de neurone est particulièrement car il donne un taux de certitude sur le résultat. De plus l'apprentissage automatique simplifie grandement sa création et réduit les erreurs. Nous allons donc à présent essayer de l'implémenter sur CPU.

4 Implantation sur CPU

Dans cette partie nous allons implémenter un réseau de neurones convolutifs sur un CPU. Pour cela nous allons utiliser le langage de programmation Python 3.7. Python est un langage de programmation interprété et orienté objet. Python est très utilisé dans l'intelligence artificielle car il offre un très grand choix au niveau des librairies dédiées à l'intelligence artificielle.

Pour créer un réseau de neurones la première étape est de choisir les données utilisées pour l'entraînement de celui-ci.

Dans le cadre de notre projet nous avons utilisé la base de données **MNIST**, pour Mixed National Institute of Standards and Technology. MNIST est une base de données de chiffres écrits à la main, très utilisée en apprentissage automatique. MNIST regroupe 70000 images de 28x28 pixels en niveau de gris, normalisées et centrées. Les images sont regroupées en deux catégories il y a 60000 images d'apprentissage et 10000 images de test. Les images d'apprentissage sont utilisées pour entraîner le réseau et les images de test permettent de voir les performances du réseau de neurones. Exemple de données de la base de données MNIST :

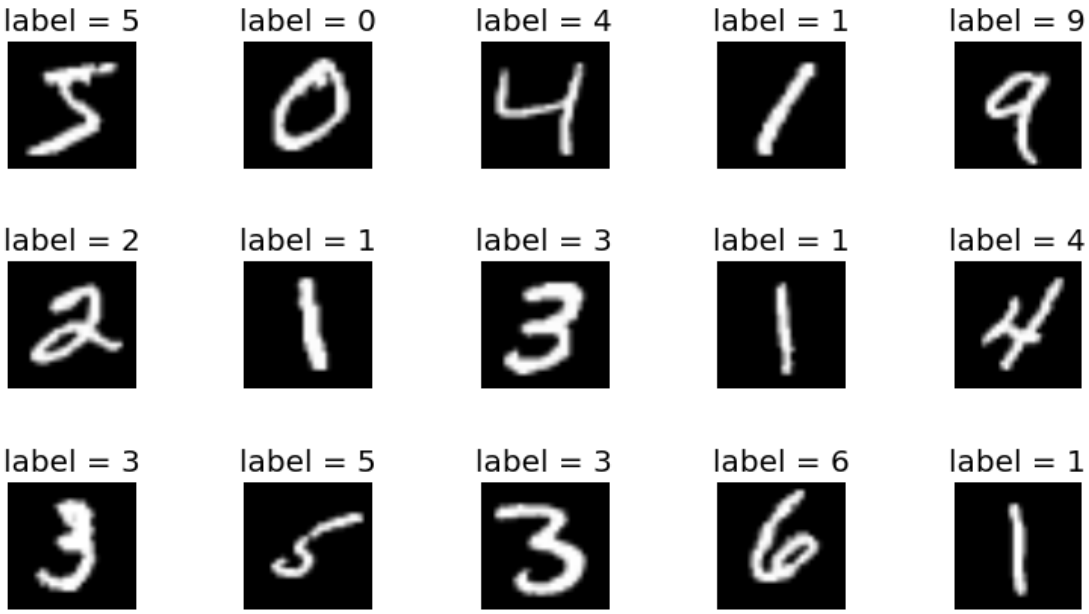

FIGURE 8 – Exécution de mire640.c

Ensuite nous avons choisi les bibliothèques Python utilisées pour l'entraînement du réseau de neurones. Nous avons choisi TensorFlow car TensorFlow est l'un des outils les plus utilisés en intelligence artificielle. TensorFlow est un outil open source d'apprentissage automatique développé par Google. Pour créer le réseau de neurones convolutifs nous avons utilisé l'API de haut niveau de TensorFLow Keras. Keras permet de créer et d'entraîner des modèles de deep learning. Keras présente trois avantages majeurs :

- Interface simple et cohérente, optimisée pour les cas d'utilisations courantes
- Les modèles Keras sont créés en connectant des composants configurables, avec quelques restrictions
- Création d'éléments de base personnalisés pour exprimer de nouvelles idées de recherche

Après avoir choisi les données et les bibliothèques utilisées nous avons choisi une architecture pour le réseau de neurones convolutifs. Nous avons décidé de choisir l'architecture la plus communément utilisée.

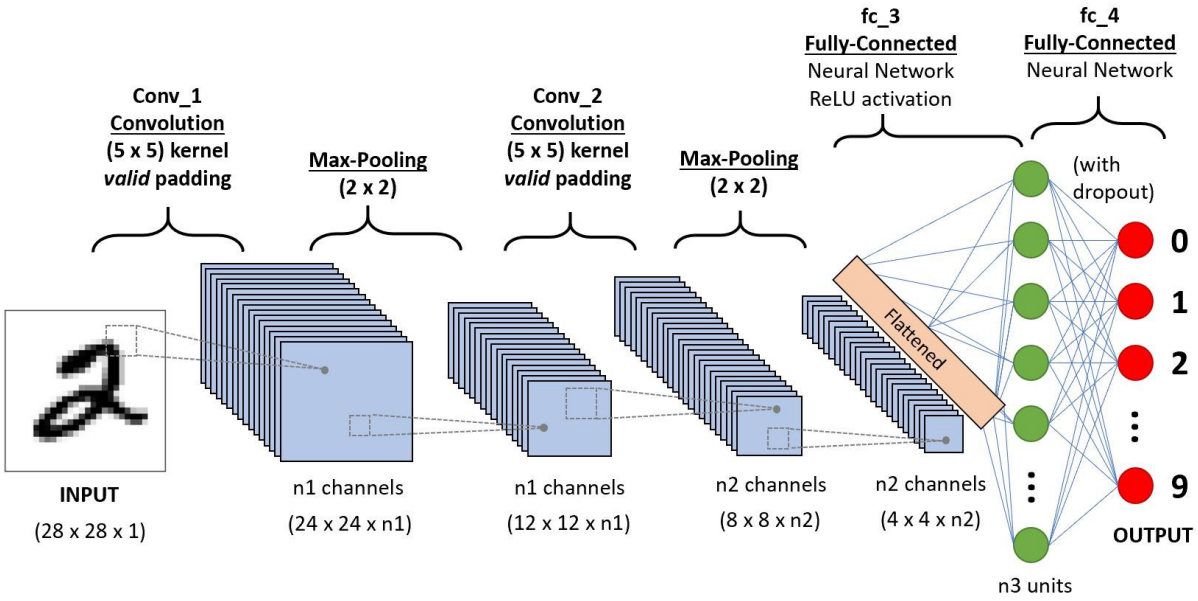

FIGURE 9 – Exécution de mire640.c

Cette architecture est composée de 7 couches. La première couche est la couche Input celle-ci prend en entrée les images de la base de donnée MNIST. Ensuite la première couche de convolution qui est le bloc de construction de base d'un réseau de neurones convolutifs. Trois hyperparamètres permettent de dimensionner le volume de cette couche :

- Profondeur : nombre de noyaux de convolution ou nombre de neurones associés à une même partie de l'image en entrée
- Pas : contrôle le chevauchement des zones analysées. Plus le pas est petit, plus les zones analysées se chevauchent et plus le volume de sortie sera grand
- Zero padding : permet de mettre des zéros à la frontière de l'image d'entrée

Pour cette couche on a spécifié les hyperparamètres suivants :

- Profondeur de 4 et les noyaux ont une dimension de 5x5
- Pas de 1x1
- On n'utilise pas le zero padding

La troisième couche est la couche de max pooling qui permet le sous-échantillonage de l'image. Dans cette couche l'image d'entrée en découpée en série de carré d'une taille 2x2 pixels et on prend le maximum des valeurs dans chaque carré ceci ,permet d'avoir une compression d'un facteur 4 de l'image, comme représente sur le schéma suivant :

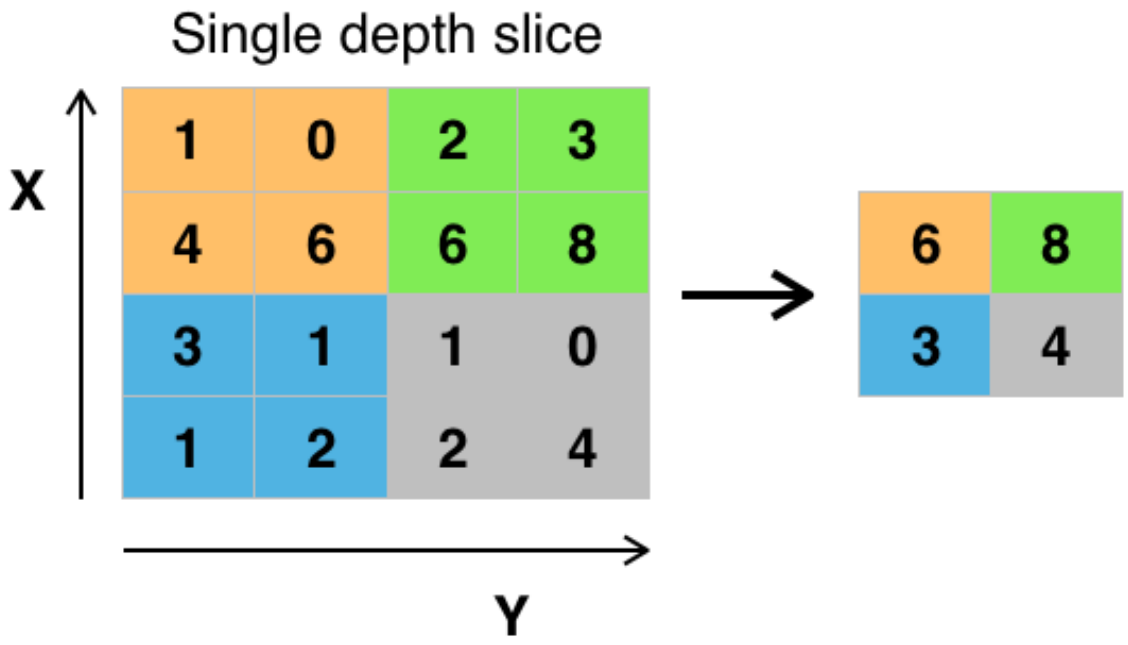

FIGURE 10 – Max pooling

On utilise une couche de max pooling entre deux couches convolutives successives d'une architecture de réseau de neurones convolutifs pour réduire le sur-apprentissage. L'opération de pooling crée aussi une forme d'invariance par translation.

La couche suivante est une deuxième couche de convolution, on garde les mêmes paramètre que pour la première couche de convolution mais on change le nombre de filtres pour égaliser le calcul à chaque couche. Donc pour cette couche on a 8 filtres. Ensuite on a une deuxième couche de max pooling suivit d'une couche entièrement connectée de 64 neurones. Les neurones dans une couche entièrement connectée ont des connexions vers toutes les sorties de la couche précédente. La couche en sortie est également une couche entièrement connectée de 10 neurones représentant les 10 chiffres écrits à la main.

On obtient alors le code Python suivant :

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
import hls4ml

batch_size = 200
num_classes = 10
epochs = 10

mnist = keras.datasets.mnist

#image size
img_x, img_y = 28, 28

(train_images, train_labels), (test_images, test_labels)= mnist.load_data()
```

```
# reshape the data into a 4D tensor - (sample_number, x_img_size, y_img_size, num_cha
train_images = train_images.reshape(train_images.shape[0], img_x, img_y, 1)
test_images = test_images.reshape(test_images.shape[0], img_x, img_y, 1)
input_shape = (img_x, img_y, 1)

# convert the data to the right type
train_images = train_images.astype('float32')
test_images = test_images.astype('float32')
train_images /= 255
test_images /= 255


# convert class vectors to binary class matrices - this is for use in the
# categorical_crossentropy loss below
train_labels = keras.utils.to_categorical(train_labels, num_classes)
test_labels = keras.utils.to_categorical(test_labels, num_classes)

model = keras.Sequential()
model.add(keras.layers.Conv2D(4, kernel_size=(5, 5), strides=(1, 1),
                 activation='relu',
                 input_shape=input_shape))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(keras.layers.Conv2D(8, (5, 5), activation='relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(64, activation='relu'))
model.add(keras.layers.Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])


class AccuracyHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.acc = []

    def on_epoch_end(self, batch, logs={}):
        self.acc.append(logs.get('acc'))

history = AccuracyHistory()

model.fit(train_images, train_labels,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(test_images, test_labels),
          callbacks=[history])
```

```
score = model.evaluate(test_images, test_labels, verbose=0)

probability_model = tf.keras.Sequential([model,
    tf.keras.layers.Softmax()])
predictions = probability_model.predict(test_images)

for layer in model.layers: print(layer.get_config(), layer.get_weights())

print("************************************")


print('train_images shape:', train_images.shape)
print(train_images.shape[0], 'train samples')
print(test_images.shape[0], 'test samples')
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Après entraînement du réseau on obtient les résultats suivants sur un processeur Intel Core i5-8250u 1.6 GHz :

```
Epoch 1/10
300/300 [==============================] - 11s 34ms/step - loss: 1.0393 - accuracy: 0.6949 - val_loss: 0.1956 - val_accuracy:
0.9391
Epoch 2/10
300/300 [==============================] - 9s 32ms/step - loss: 0.1944 - accuracy: 0.9409 - val_loss: 0.1208 - val_accuracy: 0
.9626
Epoch 3/10
300/300 [==============================] - 10s 32ms/step - loss: 0.1295 - accuracy: 0.9593 - val_loss: 0.0941 - val_accuracy:
0.9691
Epoch 4/10
300/300 [==============================] - 10s 32ms/step - loss: 0.1034 - accuracy: 0.9685 - val_loss: 0.0775 - val_accuracy:
0.9741
Epoch 5/10
300/300 [==============================] - 10s 32ms/step - loss: 0.0893 - accuracy: 0.9722 - val_loss: 0.0759 - val_accuracy:
0.9739
Epoch 6/10
300/300 [==============================] - 10s 33ms/step - loss: 0.0769 - accuracy: 0.9766 - val_loss: 0.0624 - val_accuracy:
0.9780
Epoch 7/10
300/300 [==============================] - 10s 32ms/step - loss: 0.0670 - accuracy: 0.9790 - val_loss: 0.0629 - val_accuracy:
0.9796
Epoch 8/10
300/300 [==============================] - 10s 33ms/step - loss: 0.0627 - accuracy: 0.9811 - val_loss: 0.0595 - val_accuracy:
0.9782
Epoch 9/10
300/300 [==============================] - 10s 33ms/step - loss: 0.0595 - accuracy: 0.9812 - val_loss: 0.0541 - val_accuracy:
0.9822
Epoch 10/10
300/300 [==============================] - 9s 31ms/step - loss: 0.0563 - accuracy: 0.9830 - val_loss: 0.0513 - val_accuracy: 0
.9820
```

FIGURE 11 – Résultats après entraînement du réseau

On constate que le réseau de neurones convolutif est très performant on obtient une précision de 98,28% pour un temps d'entraînement de 1min40.

5 Implantation sur FPGA

Après avoir réussi à implémenter un réseau de neurones convolutifs sur un CPU, on va essayer d'implémenter le même réseau sur une carte FPGA. Compte tenu de la différence en terme de performance entre les deux.
Pour cela on va utiliser le même script python précédemment implémenté sur CPU. Mais pour le rendre fonctionnel sur une FPGA, on va devoir le convertir en langage de description VHDL.

5.1 Présentation de l'outil hls4ml

Pour convertir le code Python établit précédemment en code VHDL on va utiliser l'outil hls4ml. Hls4ml est une bibliothèque Python développée pour implémenter des algorithmes d'apprentissage automatique sur FPGA. Hls4ml traduit le code Python en un modèle HLS qu'on peut ensuite implémenter sur une carte FPGA.

Hls4ml possède une large documentation et des exemples pour la prise en main de l'outil et offre un très grand choix de personnalisation des paramètres. On peut ainsi choisir la période de l'horloge, de paralléliser ou non les algorithmes, la taille des données en entrée, sortie, des données. On peut également contrôler l'utilisation des DSP.

5.2 Génération du modèle HLS

La première étape est de se servir de l'outil HLS4ML pour générer le modèle HLS correspondant à notre réseau.
Pour cela, on rajoute les lignes suivantes après compilation du réseau, cela permet de générer un projet HLS avec les différents fichiers nécessaires à la compilation.

Cette étape prends énormément de temps vu que le processus de conversion est très compliqué, en plus de la présence de plusieurs couches cachées ainsi que les convolutions.

```
config = hls4ml.utils.config_from_keras_model(classifier,granularity='model');
hls_model = hls4ml.converters.convert_from_keras_model(classifier, hls_config=config,
print("-----------------------------------")
hls4ml.utils.plot_model(hls_model, show_shapes=True, show_precision=True, to_file=None
hls_model.compile();
```

5.3 Gestion du projet complet

Théoriquement, après avoir générer le code VHDL décrivant notre réseau de neurones, on doit l'intégrer au sein de notre architecture et ensuite tester son fonctionnement sur une carte FPGA.

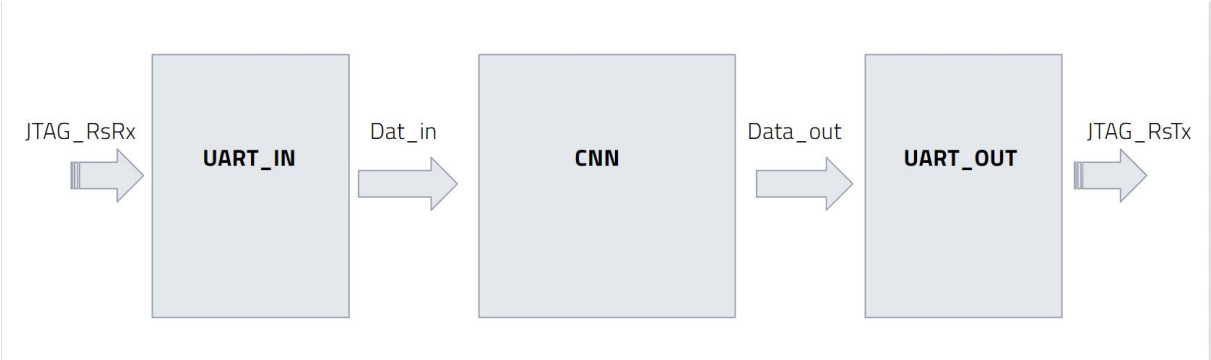

FIGURE 12 – Schéma de l'architecture complète

La présence des deux modules UART, permet de gérer les communications entre la carte FPGA et l'ordinateur afin de transmettre les images d'entrée via le protocle **JTAG**.

En effet, le module UART_IN permet de convertir les images d'entrée en vecteurs de pixels (*Data_in*) ensuite exploité par le réseau de neurones pour prédire le chiffre manuscrit en entrée. cette probabilité est transmise au module UART_OUT pour l'envoyer finalement via le port JTAG à l'ordinateur.

6 Contraintes techniques

Malheureusement, on a pas pu aller jusqu'au bout de notre implémentation, vu qu'on a rencontrés des problèmes techniques liés à l'utilisation de l'outil HLS4ML

En effet, comme le montre la figure ci-dessous la présence de boucle dans notre algorithme cause des problèmes de mémoires. Et par conséquent une erreur de compilation.

```
ERROR: [XFORM 203-504] Stop unrolling loop 'ConvOutHeight' (firmware/nnet_utils/nnet_conv2d.h:245) in function 'nnet::conv_2d_
latency_cl<ap_fixed<16, 6, (ap_q_mode)5, (ap_o_mode)3, 0>, ap_fixed<16, 6, (ap_q_mode)5, (ap_o_mode)3, 0>, config5>' because i
t may cause large runtime and excessive memory usage due to increase in code size. Please avoid unrolling the loop or form sub
-functions for code in the loop body.
```

FIGURE 13 – Message d'erreur lors de la compilation du réseau

Cette erreur est très fréquente pour ce genre de projet où on utilise beaucoup de ressources en même temps, et les développeurs derrière l'outil HLS4ML essayent de la résoudre.

7 Conclusion

Malgré les difficultés rencontrées lors de ce projet, nous avons pu en tirer plusieurs leçons.

Tout d'abord, nous avons réussi a découvrir un nouveau champs de connaissances : celui de l'intelligence artificielle et des réseaux de neurones. Nous avons notamment appris à construire un réseau de neurones et à l'implémenter sur un CPU.

Le défi était de pouvoir utiliser une carte FPGA, pour comparer ses performances avec celles d'un CPU. En effet, l'utilisation d'une carte FPGA permet d'optimiser les performances, de par le fait qu'une FPGA est programmée pour remplir une seule fonction contrairement au CPU.

Au niveau des performances, l'implémentation FPGA sera donc beaucoup plus rapide et consommera moins de ressources.