INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

RAPPORT DE PROJET VHDL

4EII

Camille Yver

Fabio Eid Morooka

Sommaire :

Introduction :

Ce projet a pour but de nous faire étudier le principe de réseau de neurone, et son implémentation en VHDL. Pour cela nous disposerons du logiciel ModelSim, qui nous permettra de simuler notre code, afin de vérifier son fonctionnement.

Notre travail sur le réseau de neurone se basera principalement sur l'utilisation même du réseau, nous considérerons ainsi les poids des neurones connus et constant pour chaque neurone. Notre travail se divisera en deux parties principales. Tout d'abord la création d'un seul neurone, de même que son entité de test. Par la suite la création du réseau dans son intégralité, organisé en couches.

Nous aurons donc à gérer la transmission de l'information à travers un neurone, puis la gestion de l'architecture des couches de neurone.

Principe :

Les réseaux de neurones sont très importants dans la compréhension des domaines tels que la classification ou l'intelligence artificielle. Ils se basent sur le principe biologique des neurones: l'information, sous forme de signal électrique, est acheminée jusqu'au corps cellulaire du neurone par ses dendrites, puis une fois traitée, est renvoyée sur l'axone. Les neurones communiquent ainsi entre eux afin de faire voyager et traiter les informations dans notre cerveau.

Le principe de neurone artificiel est proche. Une entité neurone reçoit un certain nombre d'information en entrée. A chaque entrée est associée un "poids" représentant la force de la connexion avec le neurone précédent. En sortie de ce neurone, une information, renvoyée vers plusieurs neurones.

Le principe de traitement de l'information dans notre cas est assez simple, il s'agit d'une fonction de feuillage tel que :

$Y = \sum_{i=0}^{N-1} W_i * X_i$, avec W_i le poids de l'entrée et X_i la valeur de l'entrée.

Ensuite, la valeur Y sera comparée à un paramètre T. Si Y<T alors la sortie du neurone prend une valeur minimale V_{min} (correspondant à une tension plus faible), il prend une valeur maximale V_{max} sinon.

C'est cette gestion de l'information que nous aurons à coder dans l'entité neurone

Entité Neurone :

Nous avons cherché tout d'abord à modéliser un seul neurone. On pose en entrée le poids de chaque entrée, comme constante générique. On crée par ailleurs des constantes globales au système, pour les valeurs minimale et maximale des sorties et pour le seuil T.

Notre entité prendra en entrée un tableau d'entier, représentant les valeurs à traiter par le neurone, et en sortie un entier résultant des équations présentées précédemment.

```
entity neurone is
        generic (Wi : IN tab_nc(1 to N) := (others => 1));
        port(   entreeXi : IN tab_nc(1 to N);
                sortie : OUT integer);
end neurone;
```

Figure 1 : Extrait de code – fichier : PROJET_NEURONE.vhd

```
architecture arch_neurone of neurone is
        signal poids: tab_nc (1 to N) := Wi;
begin

process(entreeXi)

variable Y : integer;

begin

  Y := 0;

        for i in 1 to N loop
                Y := Y + entreeXi(i) * Wi(i);
        end loop;

        if(Y>T) then
                sortie <= VMAX;
        elsif(Y<T or Y=T) then
                sortie <= VMIN;
        end if;

end process;
```

Figure 2 : Extrait de code – fichier : PROJET_NEURONE.vhd

L'architecture du neurone en tant que telle est assez succincte. Il suffit de faire les opérations dans un process, prenant en liste de sensibilité les entrées X_i.

L'entité de test rattachée au neurone consiste en la création de deux neurones, afin de tester leur bon fonctionnement. Nous avons ainsi pu les tester pour des poids différents mais une même entrée, et regarder leur sortie. Nous obtenons par exemple :

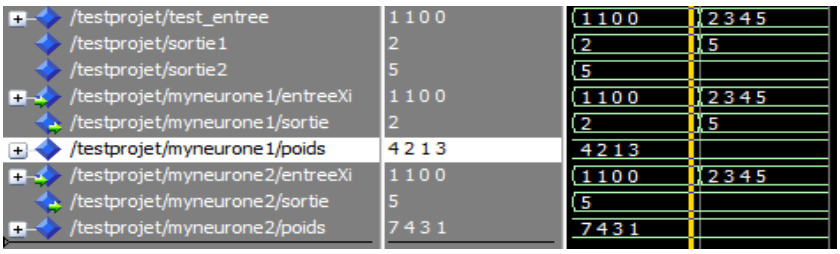


Figure 3 : Extrait de simulation – test_neurone

Nous avons donc un neurone qui a un comportement correct. Il est maintenant temps d'associer plusieurs neurones afin d'en construire un réseau.

Entité Couche et Ligne :

Pour construire le réseau de neurone, nous avons pensé au projet dans sa globalité. Nous nous sommes tout d'abord demandé comment créer les différents poids de chaque neurone, ainsi que les couches de neurones avec un nombre différent de neurones, et enfin nous avons réfléchi à comment sauvegarder les vecteurs de sorties pour les utiliser après comme entrée dans la prochaine couche, comment cela est expliqué sur la figure dessous.

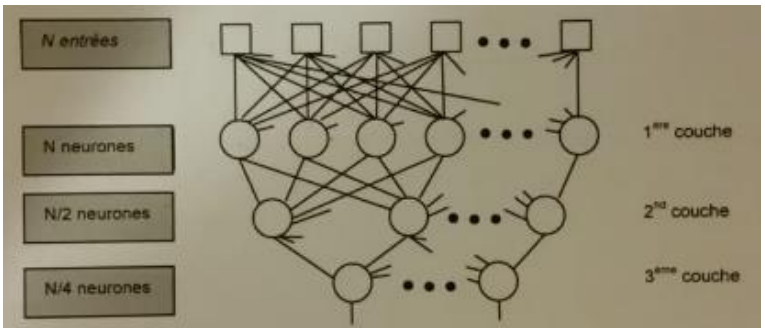


Figure 4 : Schéma explicatif du réseau en couche

```
package valeurs is

        ---Paramètres modifiables du réseau---
        CONSTANT N : integer := 4;
        CONSTANT couche : integer := 3;
        ------------------------------------

        CONSTANT T : integer := 10;
        CONSTANT VMAX : integer := 5;
        CONSTANT VMIN : integer := 2;

        type tab_nc is array(natural range <>) of integer;
        subtype indice is integer range 1 to N;

        subtype t_dim2 is tab_nc(1 to N);
        type array_nc is array(natural range <>) of t_dim2;

end package;
```

Nous avons tout d'abord créé une matrice d'entier dans le package initial, de cette façon on a directement une structure pour sauvegarder toutes les sorties de toutes les couches (de plus, on peut utiliser la dernière sortie comme entrée de la prochaine couche).

Figure 5 : Extrait de code – fichier : PROJET_NEURONE.vhd

On a ensuite établi la relation entre le nombre de couches et le nombre de neurones. On a conclu qu'ils sont liés par un logarithme de base 2 tel que :

$Nb = 2^{(N-1)}$, avec Nb le nombre de neurones dans la première couche et N le nombre des couches

Une ligne a ainsi moitié moins de neurones que la ligne précédente, la première ayant comme nombre de neurone le nombre d'entrée.

Nous avons donc initialement structuré notre projet avec des bases assez simple : les poids ne changent pas ni dans chaque neurone ni dans chaque couche et il n'y aura toujours qu'une sortie unique à la fin du réseau. Nous avons aussi établi que le nombre de couche et d'entrée seraient réglé de façon à obtenir ce résultat.

Notre principale difficulté au début n'était pas de créer plusieurs entités neurones dans une seule architecture, mais de créer plusieurs couches avec différents nombres de neurones.

Pour créer une ligne avec plusieurs neurones on a utilisé la fonction *for … generate*. De même pour les autres couches que l'on a créé avec une boucle *for,* où chaque itération pouvait changer le nombre de neurones voulu par ligne.

Ainsi la création des lignes de neurone et des couches se fait dans deux entités différentes, le *for...generate* ne permettant pas l'utilisation d'un process.

L'entité "ligne" se charge de la génération de tous les neurones de la couche, et retourne le résultat de la ligne (un vecteur de la sortie de chaque neurone de cette couche).

L'entité "couche" s'occupe de la "gestion" de toutes les lignes, c'est elle qui indique le nombre de neurones que chaque ligne va avoir. Elle se charge aussi de changer le vecteur d'entrée de chaque ligne.

```
        signal tab_interm : tab_nc(1 to N) := (others => 0);

begin
  --On génére ici chaque neurone d'une seule et unique ligne
        GEN_Neurone: for i in 1 to NouveauN generate
        myneuroneX: neurone
                generic map( Wi => random_number(N, i))
                port map (entreeXi => tab_entree_line, sortie => tab_interm(i));
        end generate GEN_Neurone;

        tab_sortie_line <= tab_interm;
```

Figure 6 : Extrait de code – fichier : PROJET_LIGNE.vhd

L'architecture du code de l'entité "ligne" est très simple, en effet il n'y a qu'une fonction *generate* pour créer les N neurones de la ligne (appelé "NouveauN" car chaque ligne sera créée avec un nombre différent de neurones).

La variable "tab_interm" est un signal qui sert à sauvegarder chaque sortie de chaque neurone de la couche que l'entité va générer. On réinitialise toujours cette variable à zéro, car les entrées manquantes pour les couches après la première doivent être considérées à 0.

Dans l'entité "couche" on a déclaré comme sortie du réseau le vecteur "sortie_final", qu'on a considéré comme la sortie du dernier vecteur que la dernière ligne a générée. La taille de ce vecteur dans un premier temps correspondait au nombre de neurones de la première couche.

```
entity couche_neurone is
        generic( NbCouche : integer);
        port( sortie_final : OUT tab_nc(1 to (N/(2**(couche-1)))));
end entity;
```

Figure 7 : Extrait de code – fichier : PROJET_COUCHE.vhd

Dans l'architecture de l'entité "couche", on a utilisé de nouveau la fonction *for...generate* pour générer chaque couche de neurones. A chaque itération la taille de la variable "NouveauN" qui est le nombre de neurones de la couche, ainsi que l'entrée "tab_entree_line" qui est l'entrée de la couche, sont changées.

```
--Les entrées sont générées pseudo-aléatoirement
tab_sortie(0)<= random_entree(N);

--On va créer ici chaque ligne du réseau de neurone

        GEN_Line: for i in 1 to NbCouche generate
                myline: generate_line
                                generic map( NouveauN => (N/(2**(i-1))))
                                port map ( tab_entree_line => tab_sortie(i-1), tab_sortie_line => tab_sortie(i));
        end generate GEN_Line;

        sortie_final <= tab_sortie(NbCouche)(1 to (N/(2**(couche-1))));
|
```

Figure 8 : Extrait de code – fichier : PROJET_COUCHE.vhd

Tests et Amélioration :

Nous avons ensuite cherché à tester notre solution.

Nous ne prenons pas en compte les cas où le nombre d'entrée est insuffisant par rapport au nombre de couche demandé, et nous restons dans des cas "normaux". La condition pour avoir un fonctionnement normal est la suivante:

$$Nb \leq 2^{(N-1)}$$

Ces deux variables peuvent être modifiées via le package "variables" qui est dans l'archive "PROJET_NEURONE.vhdl".

```
package valeurs is

        ---Paramètres modifiables du réseau---
        CONSTANT N : integer := 4;
        CONSTANT couche : integer := 3;
        ---------------------------------------
```

Figure 8 : Extrait de code – fichier : PROJET_NEURONE.vhd

L'entité de test du système reste assez simple. En effet le réseau ne prenant en entrée que le nombre de couche du système et en sortie le résultat de la dernière couche.

Au début nous ne pensions pas qu'il était possible d'avoir plusieurs sorties. Il a donc fallu au moment des tests modifier l'architecture afin que cela soit possible, et transformer la sortie, initialement un Integer, en tableau. Cela ne fut cependant pas compliqué.

Nous avons également amélioré le système afin qu'il ne montre que les cases du tableau contenant des valeurs non nuls, afin de ne pas afficher les valeurs inutiles, ainsi que la génération de nos entrées et des poids en les automatisant. Nous avons ainsi des nombres pseudo-aléatoire pour les poids :

```
  for i in 1 to valeur loop
uniform(seed1, seed2, rand1);    -- generate random number
uniform(seed3, seed4, rand2);    -- generate random number
   wi(i):= abs((i-j) mod integer((rand1*range_of_rand)+rand2+1.0));

  end loop;
```

Figure 9 : Extrait de code – fichier : PROJET_RDMNUMBER.vhd

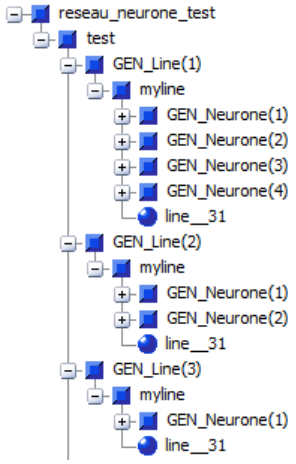
La fonction random créée pour générer les vecteurs de poids différents à chaque neurone est très simple. Nous utilisons la fonction “uniform” du VHDL, qui génère un nombre random utilisant 2 variables positives (seeds) et 1 variable réel (rands). La création de ces nombres pseudo-aléatoires sont fait à l’intérieur d’une boucle *for*.

On fait ensuite un calcul pour choisir la valeur du poids. On utilise alors plusieurs variables, la première *j* est le nombre de la couche qu’on va créer, *i* est le nombre de neurone dans la couche. Ces valeurs aléatoires seront finalement comprises entre 1 et 9.

Pour le vecteur d’entrée on a aussi créé une fonction pour le générer de manière plus rapide et itérative, selon le changement des valeurs de nombre de couche et/ou nombre de neurones. On a fait une fonction très simple que génère un vecteur de même taille que le nombre de neurones, ou la valeur est 2 dans une entrée paire et 5 si elle est impaire.

On peut finalement voir dans cette simulation que notre réseau se comporte de manière correcte :

Figure 10 : Extrait de simulation – réseau de neurone complet



On peut remarquer également qu’au final, on a bien la structure demandée, avec ici trois couches et quatre entrées.

Figure 11 : Visualisation des couches

Conclusion :

Ce projet nous aura permis d'aborder succinctement le concept de réseau de neurone et de voir son fonctionnement. Il nous aura également fait appliquer les fonctions de *generate*, que nous n'avions pas vu de manière expérimentale auparavant.

Globalement le projet ne nous a pas posé de problème majeur. Nous sommes parvenus à réaliser les différentes couches, avec une architecture cohérente et qui répond au cahier des charges. Nous avons cherché à rendre les poids des différents neurones aléatoires, pour pouvoir obtenir le plus de configuration possible sans avoir à trop changer le code. Cela n'a pas été une réussite complète, les poids se répétant d'une couche sur l'autre.

Cependant nous sommes plutôt satisfaits de la conduite de notre projet, qui a pu être fini dans les temps et dont le résultat est correct.

Annexe :

Entité Neurone et package :

```
library work;
library ieee;
use ieee.std_logic_1164.all;

package valeurs is

	---Paramètres modifiables du réseau---
	CONSTANT N : integer := 4;
	CONSTANT couche : integer := 3;
	--------------------------------------

	CONSTANT T : integer := 10;
	CONSTANT VMAX : integer := 5;
	CONSTANT VMIN : integer := 2;

	type tab_nc is array(natural range <>) of integer;
	subtype indice is integer range 1 to N;

	subtype t_dim2 is tab_nc(1 to N);
	type array_nc is array(natural range <>) of t_dim2;

end package;


library work;
library ieee;
use ieee.std_logic_1164.all;
use work.valeurs.all;

entity neurone is
	generic (Wi : IN tab_nc(1 to N) := (others => 1));
	port(	entreeXi : IN tab_nc(1 to N);
		sortie : OUT integer);
end neurone;
```

```
architecture arch_neurone of neurone is
        signal poids: tab_nc (1 to N) := Wi; --signal crée afin de récupérer les valeurs des poids choi-
sies aléatoirement en simulation
begin
process(entreeXi)
variable Y : integer;
begin
 Y := 0;
        for i in 1 to N loop
                Y := Y + entreeXi(i) * Wi(i);
        end loop;
        if(Y>T) then
                sortie <= VMAX;
        elsif(Y<T or Y=T) then
                sortie <= VMIN;
        end if;
end process;

end arch_neurone;
```

Entité Ligne :

```
library work;
library ieee;
use ieee.std_logic_1164.all;
use work.valeurs.all;
use work.random.all;

entity generate_line is
	generic( NouveauN: integer := 2);
	port(	tab_entree_line : IN tab_nc(1 to N);
		tab_sortie_line : OUT tab_nc(1 to N));
end entity;

architecture arch_generate_line of generate_line is

component neurone
	generic(Wi : IN tab_nc(1 to N) := (others => 1));
	port(	entreeXi : IN tab_nc(1 to N);
		sortie : OUT integer);
end component;
	signal tab_interm : tab_nc(1 to N) := (others => 0);
begin
 --On génére ici chaque neurone d'une seule et unique ligne
	GEN_Neurone: for i in 1 to NouveauN generate
	myneuroneX: neurone
		generic map( Wi => random_number(N, i)) --Les poids de chaque neurone pour une
même ligne est aléatoire. Entre deux lignes différentes ils auront le même poids
		port map (entreeXi => tab_entree_line, sortie => tab_interm(i));
	end generate GEN_Neurone;

	tab_sortie_line <= tab_interm;

end architecture;
```

Entité Couche :

```
library work;
library ieee;
use ieee.std_logic_1164.all;
use work.valeurs.all;
use work.random.all;

entity couche_neurone is
	generic( NbCouche : integer);
	port( sortie_final : OUT tab_nc(1 to (N/(2**(couche-1)))));
end entity;

architecture arch_couche_neurone of couche_neurone is

	signal NouveauN: integer := NbCouche;
	signal tab_sortie : array_nc(0 to NbCouche);

component generate_line
	generic( NouveauN: integer := 1);
	port(	tab_entree_line : IN tab_nc(1 to N);
		tab_sortie_line : OUT tab_nc(1 to N));
end component;

begin

--Les entrées sont générées pseudo-aléatoirement
tab_sortie(0)<= random_entree(N);

--On va créer ici chaque ligne du réseau de neurone

	GEN_Line: for i in 1 to NbCouche generate
		myline: generate_line
			generic map( NouveauN => (N/(2**(i-1))))
			port map ( tab_entree_line => tab_sortie(i-1), tab_sortie_line =>
tab_sortie(i));
	end generate GEN_Line;
	sortie_final <= tab_sortie(NbCouche)(1 to (N/(2**(couche-1))));
end architecture;
```

Entité de test neurone :

```
library work;
library ieee;
use ieee.std_logic_1164.all;
use work.valeurs.all;

entity testPROJET is
end entity;

architecture arch_testPROJET of testPROJET is
	signal test_entree : tab_nc(1 to N);
	signal sortie1, sortie2 : integer;

component neurone
	generic(Wi : IN tab_nc(1 to N) := (others => 1));
	port(	entreeXi : IN tab_nc(1 to N);
	sortie : OUT integer);
end component;

begin

test_entree <= (1,1, 0, 0), (2, 3, 4, 5) after 100 ns;
myneurone1: neurone
	generic map( Wi => (4, 2, 1, 3))
	port map (entreeXi => test_entree, sortie => sortie1);
myneurone2: neurone
	generic map( Wi => (7, 4, 3, 1))
	port map (entreeXi => test_entree, sortie => sortie2);

end arch_testPROJET;
```

Entité Test Réseau :

```
library work;
library ieee;
use ieee.std_logic_1164.all;
use work.valeurs.all;


entity reseau_neurone_test is
end entity;

architecture arch_reseau_neurone_test of reseau_neurone_test is
signal sortie_test : tab_nc(1 to (N/(2**(couche-1))));

component couche_neurone
        generic( NbCouche : integer);
        port( sortie_final : OUT tab_nc(1 to (N/(2**(couche-1)))));
end component;

begin
test : couche_neurone
        generic map(nbCouche => couche)
        port map(sortie_final => sortie_test);

end architecture;
```

Fonctions de génération des poids et des entrées :

```
library work;
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use work.valeurs.all;

package random is
        function random_number(valeur : integer; j: integer) return tab_nc;
        function random_entree(valeur : integer) return tab_nc;
end package random;

package body random is
function random_number(valeur : integer; j : integer) return tab_nc is

   variable seed1, seed2,seed3, seed4: positive;
   variable rand1,rand2: real;
   variable range_of_rand : real := 8.0;
   variable wi : tab_nc(1 to valeur):= (others => 1);

begin
   for i in 1 to valeur loop
        uniform(seed1, seed2, rand1);   -- generate random number
        uniform(seed3, seed4, rand2);   -- generate random number
        wi(i):= abs((i-j) mod integer((rand1*range_of_rand)+rand2+1.0));
   end loop;
        return wi;
end random_number;

function random_entree(valeur : integer) return tab_nc is
   variable wi : tab_nc(1 to valeur):= (others => 1);
--Les entrées vont prendre la valeur 2 si c'est une entrée paire, 5 sinon
begin
   for i in 1 to valeur loop
        if i mod 2 = 0 then
                wi(i):= VMIN;
        else
                wi(i) := VMAX;
        end if;
   end loop;
   return wi;
end random_entree;
end random ;
```

INSA Rennes
20 Avenue des Buttes de Coësmes
CS 70839
35708 Rennes Cedex 7

Tél. +33 (0) 2 23 23 82 00
Fax +33 (0) 2 23 23 83 96

www.insa-rennes.fr