Pynq Project
Neural Network
Report
EE 6043

Paul Devouge, Julie Richard

January 9, 2023

Professor: Dr Emanuel Popovici

Contents

List of Figures

1 Introduction

This project aims to simulate and implement a neural network on the Pynq-z2 board. The first step of this project is the pynq lab that allows to discover how to interact with the board and its overlay by Ethernet. We were provided with a notebook implementing a Neural Network (NN) in Python. The notebook explains what a NN is and the steps to create such a NN in order to recognize a handwritten digit.

NB: Some files of the project were compiled in SystemVerilog, to allow the use of array in the ports of the modules. SystemVerilog is very similar to Verilog, but the possibility to declare ports as arrays is more convenient here.

All of the codes are available on Github [4].

2 Neural Network

2.1 Definition

A neural network consists of a number of layers of interconnected nodes. The nodes are called neurons, which loosely model the neurons in a biological brain [1]. Neural networks (NN) are composed of a first layer, the input layer, one or more hidden layers and an output layer. Each node, or artificial neuron, connects to one another and has an associated weight and threshold. If the output of an individual node is greater than the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is sent to the next layer of the network.[2]

Figure 1 shows the schematic of a NN :

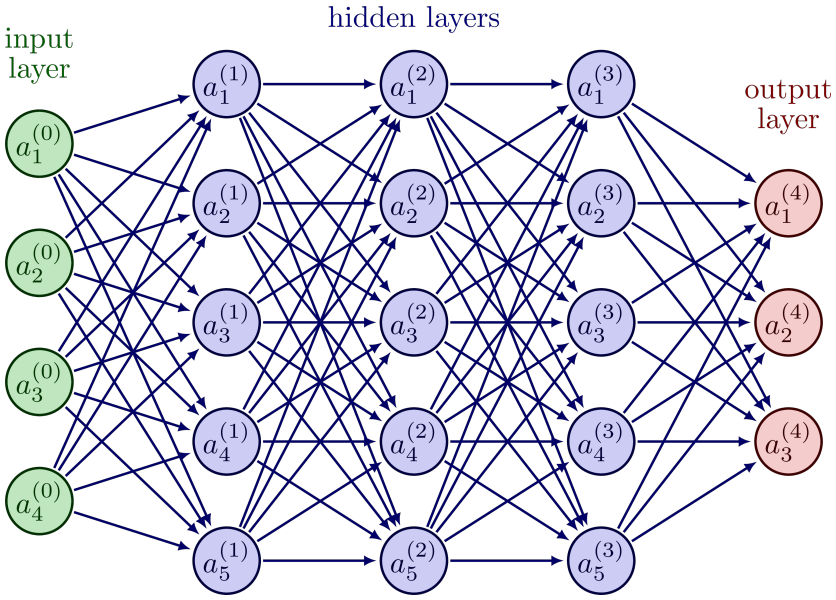

Figure 1: Layers of a Neural Network

2.2 Project structure

The goal of this project is to create a neural network capable of recognizing a handwritten digit (between 0 and 9) represented as a 28 by 28 pixels image. According to the Jupyter Notebook, our NN should be composed of 4 layers.

Figure 2 shows the differents layers and steps to identify the handwritten digit.

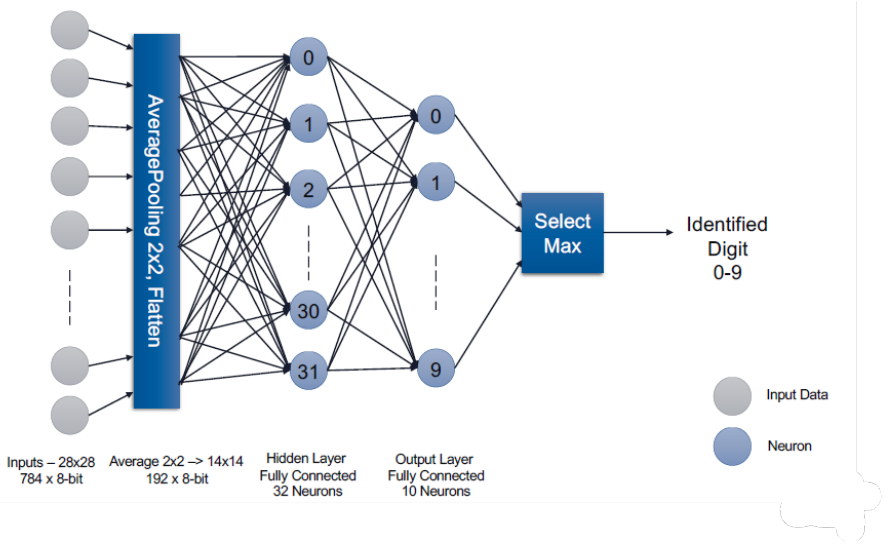

Figure 2: Project structure

The input image is coded on 28x28=784 pixels, each of them being coded on 1 byte. The input layer corresponds to the pixel values arranged as shown in Figure 3.

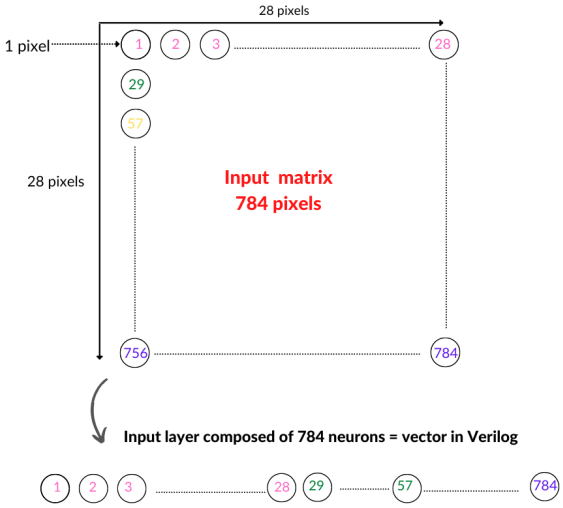

Figure 3: Input layer in Verilog

The first step consists in calculating a new image by using an average pooling of size 2 by 2 pixels. That means we will obtain a new image of 14x14=192 pixels.

Then there is the first hidden layer (dense layer 1) whose neurons are fully connected (we call this a dense layer) to the previous layer (result of average pooling). Layer 1 and Layer 2 are composed of the weights and biases computed in the Jupyter notebook. The method of calculation will be seen later in the report.

Finally, Layer 2 or the output layer consists of 10 neurons (again arranged in a dense layer). Each final neuron corresponds to a single digit and the neuron with the highest value is the one corresponding to the handwritten digit.

Note: In our verilog code, we present the 28 by 28 pixels image as a vector of 784 elements. This accounts for the missing flatten layer and is easier to use in Verilog.

2.3 Average Pooling

2.3.1 Method of the average pooling

The first step is an averaging to obtain a compressed image of 192 pixels. Figure 4 shows the overall process. It calculates the average of 4 pixels arranged in a 2x2 matrix.

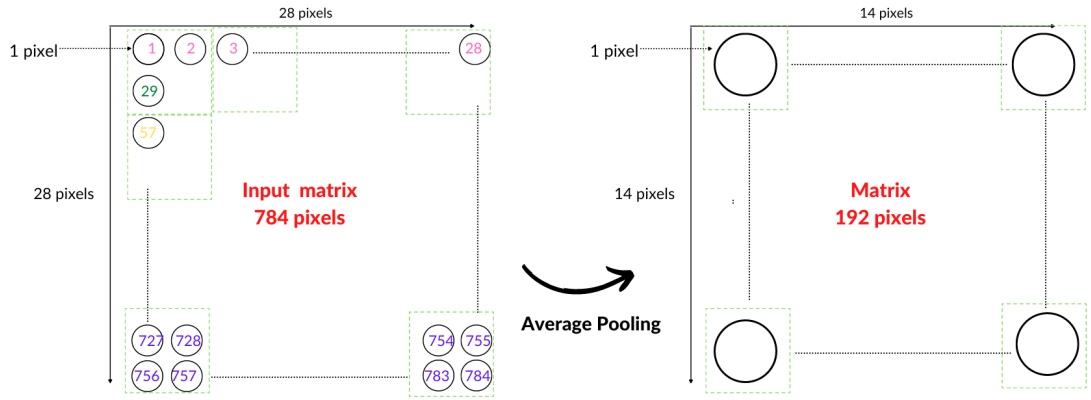

Figure 4: Average pooling schematic

2.3.2 Intermediate module : avg_pooling

To code this step, we created a submodule used in the first layer of the main module neural_network. This intermediate module called avg_pooling calculates the average of four pixels (the ones of our 2x2 matrix). It takes in arguments the four pixel values, the clock signal and the enabling signal. Figure 5 describes the code of avg_pooling :

```
module avg_pooling(
    input clk,
    input pool_en,
    input [7:0] in1,
    input [7:0] in2,
    input [7:0] in3,
    input [7:0] in4,
    output [7:0] out,
    output  pool_done
    );

    reg [15:0] pool_out;

    always @(posedge clk) begin
        if(pool_en == 1) begin
            pool_out <= (in1+in2+in3+in4)>> 2;
        end

    end

    assign out = pool_out[7:0];
    assign pool_done = (pool_out==(in1+in2+in3+in4)>> 2)? 1:0;

endmodule
```

Figure 5: avg_pooling code

This module returns two signals : the first one is the average and the second one, pool_done, indicates if the average has been successfully calculated. One can notice that the calculation of average does not use a simple division (since Verilog does not implement division other than those by power of two) but a right shift. Right shifting the sum of the four pixels by 2 bits actually corresponds to a division by 4.

Note: as avg_pooling is synchronous, it is necessary to use a register pool_out in order to store the value. All pixel values are coded on 8 bits so the average will also be coded on 8 bits. However, since the sum of our pixels can exceed 8 bits, it is necessary to use a 16 bits register in order to avoid overflowing.

2.3.3 First step of the Neural Network code

The first step of the Neural Network code can be written by using the previous module. This step calculates the compressed matrix (14x14 pixels). Each pixel is associated with an address corresponding to its index in the input vector. We go through the input vector, and calculate the average of 4 pixels. Figure 6 shows the code of avg_pooling_layer in SystemVerilog.

```
module neural_network(
    input clk,
    input enable,
    input reset,
    input [7:0] img [0:783],
    output [7:0] digit_out,
    output NN_done
    );

    /* Average pooling layer */

    reg pool_enable;
    wire finished_pool;
    reg signed [15:0] pool [0:195];

    // Pixel value registers
    wire signed [7:0] pool_in1;
    wire signed [7:0] pool_in2;
    wire signed [7:0] pool_in3;
    wire signed [7:0] pool_in4;
    wire signed [7:0] pool_final;

    // Pixel address registers
    reg [15:0] pool_in1_addr;
    reg [15:0] pool_in2_addr;
    reg [15:0] pool_in3_addr;
    reg [15:0] pool_in4_addr;
    reg [15:0] pool_final_addr = 0;
    reg [15:0] pool_addr = 0;
    reg [15:0] pool_row = 0;
    // Initialize addresses
    initial
    begin
        pool_in1_addr <= 8'b0000_0000;
        pool_in2_addr <= 8'b0000_0001;
        pool_in3_addr <= 8'b0001_1100;
        pool_in4_addr <= 8'b0001_1101;
        pool_enable <= 1'b1;
    end

    avg_pooling AvgPooling(clk,pool_enable,pool_in1,pool_in2,pool_in3,pool_in4,pool_final,finished_pool);

    // Load pixel values
    assign pool_in1 = ((img[pool_in1_addr]));
    assign pool_in2 = ((img[pool_in2_addr]));
    assign pool_in3 = ((img[pool_in3_addr]));
    assign pool_in4 = ((img[pool_in4_addr]));

    always @(posedge clk) begin
    if(reset) begin
        pool_in1_addr <= 8'b0000_0000;
        pool_in2_addr <= 8'b0000_0001;
        pool_in3_addr <= 8'b0001_1100;
        pool_in4_addr <= 8'b0001_1101;
        pool_final_addr <= 0;
        pool_row <= 0;
        pool_addr <= 0;
        pool_enable <= 1'b1;
    end
     else if(enable) begin
         if(finished_pool) begin // Average done
             pool[pool_final_addr] = pool_final;
             pool_addr = pool_addr + 2; // Increment address
             pool_row = pool_row + 2;
             if(pool_row == 28) begin // End of row, go down by 2 rows
                 pool_addr = pool_addr + pool_row;
                 pool_row = 0;
             end
             if(pool_in4_addr == 783) begin // Global averaging done
                 pool_enable <= 0;
             end
             else if(pool_in4_addr != 783) begin // Update addresses
                 pool_in1_addr <= pool_addr;
                 pool_in2_addr <= pool_addr + 1;
                 pool_in3_addr <= pool_addr + 28;
                 pool_in4_addr <= pool_addr + 29;
                 pool_final_addr <= pool_final_addr + 1;
             end
         end
     end
     end
```

Figure 6: Average pooling part

The first four pixels of the matrix correspond to indexes 0, 1, 28 and 29 in the input vector. We store the corresponding value of each pixel in pool_in k (where k = 1,2,3 or 4) at every clock rising edge. If the enable signal is equal to 1 and the calculation of the current average is done, we can go to the next addresses of the next 2x2 matrix. If

the row index is equal to 28 (ie end of the row), it means that we need to change row (down shifting by two lines). If the last pixel of the current matrix is equal to 783, it means that the global average is done and the enable signal is held low. In the code, pool_adress gives the first top left pixel of the matrix, pool_adress + 1 gives the top right pixel, pool_adress+ 28 gives the bottom left pixel and finally the last bottom right pixel of the matrix is given by pool_adress + 29. pool_final_adress gives the index of the new calculated pixel. This calculation is explained in Figure 4.

2.4 Hidden & Output Layer : Module Dense Layer

This section highlights the method to determine the parameters of the hidden layer and the output layer. We have obtained both by using the same method.

2.4.1 Neuron parameters

A neuron is composed of n weights (one per input) and one bias value. The explanation of how a neuron is modelized will be seen in the next subsection. We first need to understand how these weights and biases were obtained.
In the provided Jupyter Notebook, we were given a functioning NN model, with the associated weights and biases for every dense layer. However, those were coded on 32 bits (float values), which is a waste of resources for a small architecture like our Pynq board. Therefore, we did some scaling, to fit these values into 8 bits.
To do so, we simply associated the highest float value with the highest possible 8 bits value (127 since the weights values are signed values). This gave us a scaling factor that we applied everywhere.

2.4.2 Single neuron output and dense layer

Figure 7 shows the method of calculation for every neuron in the dense layers. The weights we are using are given by the Jupyter notebook. The values In_k with $0 <= k <= n-1$ and n the number of neurons are the input values (coming from the previous layer). The output of a neuron is a weighted sum of all the previous layer outputs by the neuron weights. We also add a bias value to this sum.

Finally, the neuron value goes through a last step called Activation. This is just a simple function (called activation function) applied to the output of our neuron. In our case the activation function is the ReLu function. If the output of the neuron is positive, after activation the result is the same positive integer. However, if the value is negative, the result will be 0.

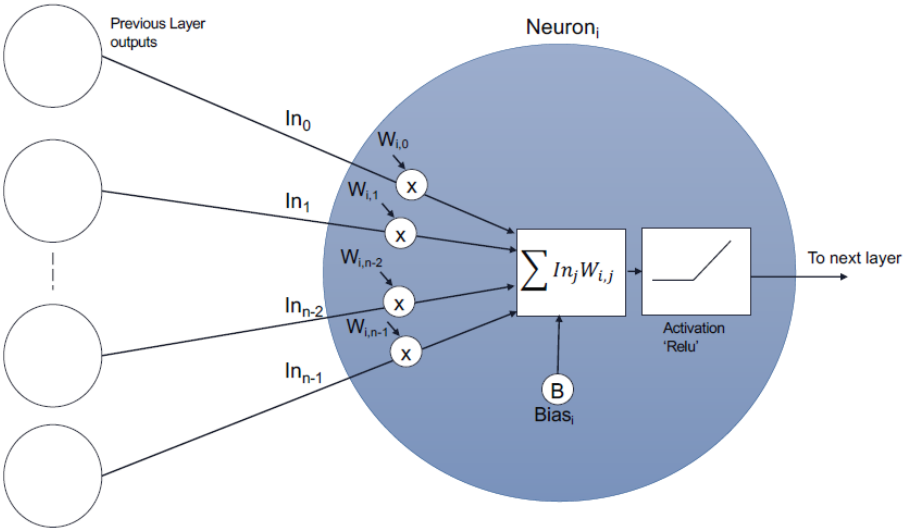

Figure 7: Second step of the Neural Network code (Hidden layer)

First of all, we implemented a simple neuron module. The latter calculates the weighted sum of all the inputs and adds the bias value. Figure 8 shows the code for this module.

```
module neuron #(parameter IN_SIZE=196, WIDTH = 8)(
    input clk,
    input en,
    input reset,
    input signed [2*WIDTH-1:0] in_data[0:IN_SIZE-1],
    input signed[WIDTH-1:0] weight[0:IN_SIZE-1],
    input signed[WIDTH-1:0] bias,
    output signed[4*WIDTH-1:0] neuron_out,
    output neuron_done
    );

    integer addr = 0;
    reg done = 0;

    reg signed [4*WIDTH-1:0] product = 0;
    reg signed [4*WIDTH-1:0] out = 0;

    always @(posedge clk) begin
        if(reset) begin
            done <= 0;
            addr <= 0;
        end
        else if(en) begin
            if(addr < IN_SIZE-1) begin
                product <= in_data[addr]*weight[addr]; //Calculate weighted input
                out <= out+product; //Sum each weighted input

            end
            if(addr == IN_SIZE-1) begin //Neuron output available
                done <= 1;
            end else begin
                addr <= addr + 1'b1;
                done <= 0;
            end
        end
    end

    assign neuron_out = out + bias; //Add bias
    assign neuron_done = done;

endmodule
```

Figure 8: Neuron code

Figure 9 presents the code of the module dense_layer used in the hidden and output layer. It takes in argument the clock signal, an enable signal, the input vector, the associated weights (for each neuron) and the corresponding biases (both calculated in the Jupyter Notebook). The output called neuron_out will store the neurons values and the other one layer_done indicates if the layer output is available. The overall module simply consists of a declaration of NEURON_NB neuron modules, all calculating the output of a single neuron.

```
module dense_layer # (parameter NEURON_NB=32, IN_SIZE=196, WIDTH=8)(
    input clk,
    input layer_en,
    input reset,
    input signed[2*WIDTH-1:0] in_data [0:IN_SIZE-1],
    input signed[WIDTH-1:0] weights [0:NEURON_NB-1][0:IN_SIZE-1],
    input signed[WIDTH-1:0] biases [0:NEURON_NB-1],
    output signed[4*WIDTH-1:0] neuron_out [0:NEURON_NB-1],
    output layer_done
    );

    reg [0:NEURON_NB-1] neuron_done;
    reg done = 0;

    neuron #(.IN_SIZE(IN_SIZE), .WIDTH(WIDTH)) dense_neuron[0:NEURON_NB-1]
        (.clk(clk), .en(layer_en), .reset(reset),
        .in_data(in_data), .weight(weights), .bias(biases),
        .neuron_out(neuron_out), .neuron_done(neuron_done)); // Neuron submodules

    always @(posedge clk) begin
        if(neuron_done == '1) begin //All neurons done
            done <= 1;
        end
    end

    assign layer_done = done;


endmodule
```

Figure 9: Dense_layer code

2.4.3 ReLu activation function

We also coded the ReLu function. It is a very simple module that checks wether the input is positive or not. Depending on the case, the output is the same as the input or is equal to 0.

```
module relu #(parameter WIDTH = 8)(
    input signed [4*WIDTH-1:0] data_in,
    output signed [2*WIDTH-1:0] data_out
    );

    wire signed [4*WIDTH-1:0] temp;

    assign temp = (data_in > 0)? data_in:0; //Take data_in if > 0, 0 else
    assign data_out = temp >> 8; //Rescale element and store into data_out

endmodule
```

Figure 10: ReLu activation function code

2.4.4 Dense Layer 1 : Hidden Layer

The hidden layer is composed of 32 neurons, whose weights are all coded on 8 bits. The module is also synchronous to the clock signal clk and is activated when its enable signal is set to 1. Note that we did not show the implementation of the module dense_layer1, since it is simply a declaration of the dense_layer module, with the correct weights and biases, followed by 32 relu modules.
In our NN module, the code checks if the average_pooling is finished. In fact, since initially pool_enable=1 and finished_dense=0, we know that if pool_enable is set to 0 that means we can switch to the hidden layer. The enable signal of layer2 can be set to 1.

```
115     /* Hidden layer */
116
117     reg dense1_enable;
118     wire finished_dense1;
119     reg signed [15:0] dense1_res [0:31];
120     initial dense1_enable <= 0;
121
122     dense_layer1 layer2 (.clk(clk), .enable(dense1_enable), .reset(reset),
123                          .pooled_img(pool), .layer_out(dense1_res),
124                          .layer_done(finished_dense1));
125
126     always @(posedge clk) begin
127         if(reset) begin
128             dense1_enable <= 0;
129         end
130         else if(enable) begin
131             if(pool_enable == 0 && finished_dense1 == 0) begin // Pooling done
132                 dense1_enable <= 1;
133             end
134             else dense1_enable <= 0; // Hidden layer done
135         end
136     end
```

Figure 11: Second step of the Neural Network code (Hidden layer)

2.4.5 Dense Layer 2 : Output Layer

This step works exactly in the same way as the second one. To set the enable signal to 1, we check if the two previous steps are done.

```
138         /* Output layer */
139
140         reg dense2_enable;
141         wire finished_dense2;
142         reg signed [15:0] dense2_res [0:9];
143         initial dense2_enable <= 0;
144
145         dense_layer2 layer3 (.clk(clk), .enable(dense2_enable), .reset(reset),
146                              .in_data(dense1_res), .layer_out(dense2_res),
147                              .layer_done(finished_dense2));
148
149         always @(posedge clk) begin
150             if(reset) begin
151                 dense2_enable <= 0;
152             end
153             else if(enable) begin
154                 if(pool_enable == 0 && finished_dense1 == 1 && finished_dense2 == 0) begin
155                     dense2_enable <= 1;
156                 end
157                 else dense2_enable <= 0; // Output layer done
158             end
159         end
```

Figure 12: Third step of the Neural Network code(Output layer)

2.5 Select Max

This function is the last one that allows to identify the digit in the picture. It returns the maximum index from the output layer, which correspond to the digit with the highest probability. Figure 13 shows the code of select_max :

```
module select_max # (parameter NEURON_NB=10, WIDTH=8)(
    input clk,
    input enable,
    input reset,
    input signed[2*WIDTH-1:0] in_data [0:NEURON_NB-1],
    output [WIDTH-1:0] digit,
    output layer_done
    );

    integer i = 0;
    reg signed [2*WIDTH-1:0] max = 0;
    reg signed [WIDTH-1:0] index = 0;
    reg done = 0;

    always @(posedge clk) begin
        if(reset) begin
            done <= 0;
            i <= 0;
            max <= 0;
            index <= 0;
        end
        else if(enable) begin
            if (in_data[i] >= max) begin //Update maximum and max index
                    max <= in_data[i];
                    index <= i;
                end
            if(i < 10) i <= i + 1;
                else done <= 1;
        end
    end

    assign digit = index;
    assign layer_done = done;

endmodule
```

Figure 13: Select_max module code

We know that the output layer is composed of 10 neurons whose values are coded on 8 bits. The select_max module implements a simple search of maximum : it scans each neuron output in the list, comparing each element to the current maximum and stops at the end of the list. The output is the maximum associated index.

This is the last module composing our Neural Network body. Figure 31 shows the corresponding code :

```
161     /* Handwritten digit selection layer */
162
163     reg max_enable;
164     wire digit_recog_done;
165     reg [7:0] digit;
166
167     initial max_enable <= 0;
168
169     select_max last_layer (.clk(clk), .enable(max_enable), .reset(reset),
170                            .in_data(dense2_res), .digit(digit),
171                            .layer_done(digit_recog_done));
172
173     always @(posedge clk) begin
174         if(reset) begin
175             max_enable <= 0;
176         end
177         else if(enable) begin
178             if(finished_dense2 == 1) begin // Output layer done
179                 max_enable <= 1;
180             end
181             else max_enable <= 0;
182         end
183     end
184
185     assign digit_out = digit;
186     assign NN_done = digit_recog_done;
187
188 endmodule
```

Figure 14: Last part of the NN : Select_max

The idea is the same as for previous modules, the enable signal allows the layer to begin only if the output layer's processing is completed. When it is the case, the enable signal of select_max module is set to 1 and the search for the maximum can begin. The signal layer_done also indicates that the NN infering is finished since select_max is the last step.

3 Simulation

This part describes the simulation results of every module. Figure 15 shows the image that we used to test the neural network. It is the number 3 that the code has to find.

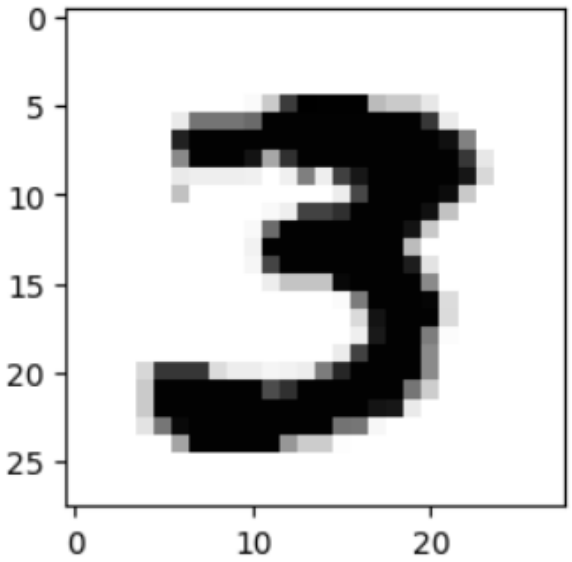

Figure 15: Initial image

3.1 Avg_pooling_layer test bench

Firstly, the average has been tested using behavioral simulation. Figure 16 shows the testbench used.

3.1.1 Testbench

The unpacked array test_data_a corresponds to the matrix of the previous input image (found with the Jupyter Notebook's results). Figure 16 presents the test bench in Verilog to test the module avg_pooling_layer.

```
module avg_pooling_tb;

    logic signed [7:0] test_data_a [0:783] = '{ 0, 4, 8, 4, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 40, 103, 127, 127, 127, 127,
    48, 40, 40, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 19, 79, 79, 79, 84, 126,
    126, 126, 126, 126, 126, 126, 126, 126, 105, 19, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 113, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126,
    120, 73, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 69, 126, 126, 126, 119, 56, 107, 126,
    126, 126, 126, 126, 126, 126, 126, 126, 105, 21, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    19, 17, 17, 17, 15, 0, 15, 74, 17, 102, 117, 126, 126, 126, 126, 126, 118, 32,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 45, 0, 0, 0, 0, 0, 0, 0, 0, 17, 99, 126, 126, 126,
    126, 122, 40, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 16, 101, 101,
    108, 126, 126, 126, 126, 120, 44, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 5, 83, 126, 126, 126, 126, 126, 126, 126, 119, 41, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 13, 126, 126, 126, 126, 126, 126, 126, 126, 48, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 100, 126, 126, 126, 126, 126, 126,
    126, 115, 24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 18, 43, 43,
    43, 124, 126, 126, 126, 126, 69, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 3, 76, 126, 126, 126, 125, 29, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 31, 119, 126, 126, 126, 30, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 116, 126, 126, 75, 3, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 18, 101, 126, 126, 126, 69, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 33, 105, 105, 105, 29, 18, 18, 10, 13, 18, 75, 111, 126,
    126, 126, 126, 69, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 40, 126, 126, 126, 126, 126, 126,
    97, 107, 126, 126, 126, 126, 126, 126, 78, 38, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 40,
    126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 118, 117, 20, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 24, 78, 123, 126, 126, 126, 126, 126, 126, 126, 126, 79, 78,
    8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 58, 126, 126, 126, 126, 126, 63,
    39, 39, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

localparam period = 20;
reg clk, enable, reset;
reg signed [7:0] img [0:783];
avg_pooling_layer avg_pooling(.clk(clk), .reset(reset), .enable(enable), .img(img));

initial begin
clk = 0;
enable = 0;
reset = 0;
img = test_data_a;
#145
enable = 1;
#200
reset = 1;
img[0] = 12;
#50
reset = 0;
end

always begin
#10 clk = ~clk;
end

endmodule
```

Figure 16: Avg_pooling testbench

3.1.2 Simulation result

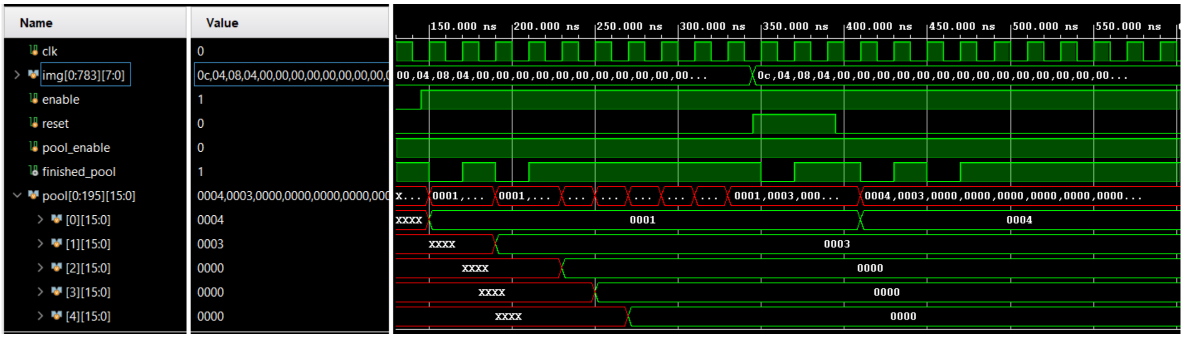

Figure 17: First part of the simulation

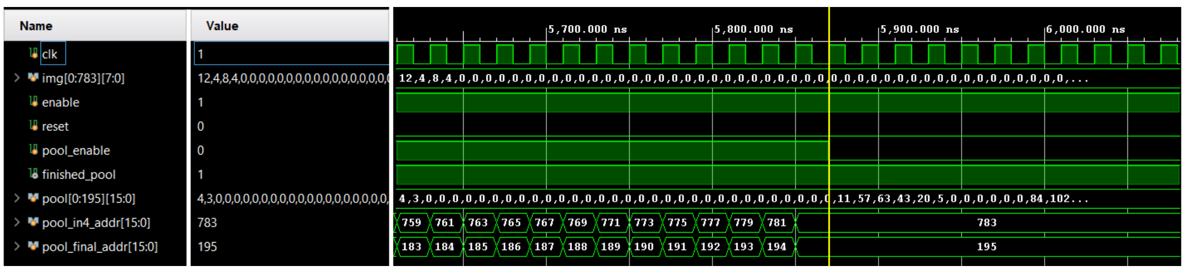

Figure 18: Last part of the simulation

To see the effect of the average pooling layer, we have modified the matrix and changed the value of pixel 1, 2 and 3 to respectively 4, 8 and 4, so that the first 2x2 matrix is composed of (0,4,0,0) and the second of (8,4,0,0). The expected result would be a vector beginning with 1 and 3 since 4/4=1 and (4+8)/4=3. We can see that when finished_pool=1, we have for output vector: pool=(1,3,0,0,...). On reset, we change the value of the first pixel to 12. As a result, our output vector now begins with 12+4/4=4. Our module seems to be working fine.
The visual result is the following Figure 19 where the average has caused a blurred effect.

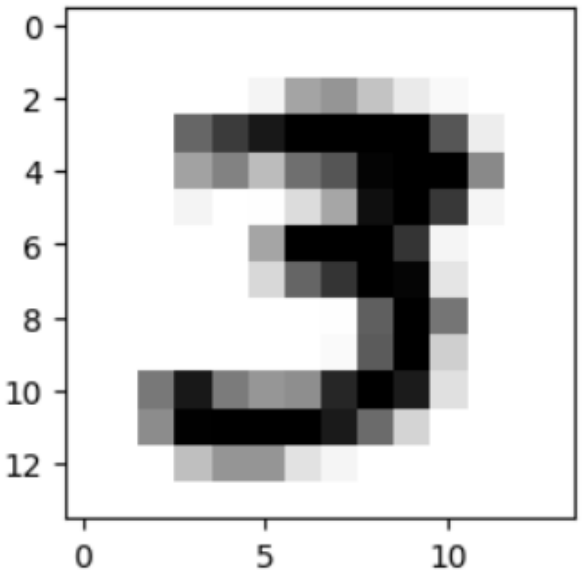

Figure 19: Image after averaging

3.2 First_dense_layer test bench

The second testbench is used to test the first dense layer (hidden layer). According to the Jupyter notebook, the expected outputs for the integer weights and biases are presented in Figure 20 :

```
Expected outputs layer 2:
[-2.4321923e+00  4.4020076e+00  1.4786428e+00 -2.3039148e+00
  1.3765686e+00 -1.8513665e+00 -8.6221498e-01  6.3652925e+00
  2.4855187e+00  6.2899482e-01 -3.5702057e+00  5.2940774e+00
  6.4958501e+00  2.7108696e+00  4.1395383e+00  1.2894821e+00
  2.9004097e-01  1.5978433e+00  6.8652928e-01 -2.7947688e+00
  4.1528697e+00  2.8426323e+00  3.8853951e+00 -2.6051440e+00
 -2.7367640e+00  5.2274246e+00 -1.5765849e-01  2.1494327e+00
 -2.1395786e+00  1.6301031e+00  3.9052432e+00  5.6025609e-03]

Expected outputs using integer weights, biases and input data:
[-16948.  28430.   9890. -15085.   8073. -13240.  -8106.  43679.  14125.
   2629. -26621.  32796.  43525.  18153.  26697.   7498.    391.   9280.
   6819. -20104.  25896.  19658.  25654. -19868. -19340.  35150.  -2057.
  14065. -16032.   9493.  24261.    291.]
```

Figure 20: Expected python results

3.2.1 Testbench

```
module dense1_tb;

logic signed [15:0] test_data_a [0:195] = '{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 11, 58,
 64, 44, 20, 6, 0, 0, 0, 0, 0, 0, 85, 103, 116, 127, 127, 127, 127, 93, 18, 0,
 0, 0, 0, 0, 58, 72, 48, 81, 93, 125, 127, 127, 69, 0, 0, 0, 0, 0, 11, 0, 1,
 29, 57, 120, 127, 104, 10, 0, 0, 0, 0, 0, 0, 0, 57, 127, 127, 127, 105, 10,
 0, 0, 0, 0, 0, 0, 0, 0, 32, 85, 106, 127, 124, 23, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 1, 88, 127, 78, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 90, 127, 37, 0, 0, 0, 0,
 0, 77, 116, 75, 63, 67, 110, 127, 115, 27, 0, 0, 0, 0, 0, 68, 126, 127, 127,
 127, 115, 83, 35, 0, 0, 0, 0, 0, 0, 0, 46, 64, 64, 26, 10, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

localparam period = 20;

reg clk, enable, reset, done;
reg signed [15:0] img [0:195];
reg signed [7:0] out [0:32];

dense_layer1 layer1 (.clk(clk), .enable(enable), .reset(reset), .pooled_img(img),
                     .layer_out(out), .layer_done(done));

initial begin
clk = 0;
enable = 0;
reset = 0;
img = test_data_a;
#145
enable = 1;
#200
reset = 1;
#50
reset = 0;
end

always begin
#10 clk = ~clk;
end

endmodule
```

Figure 21: Test bench

3.2.2 Simulation

The simulation's results are shown in Figure 22. If we compare the following results (output: -16948, 28430, 9890...) with the expected ones : we can conclude that this module is working perfectly because the results are the same.
NB: The reset signal is not presented in this screenshot because that is not the most

important part. But it was working just as fine.

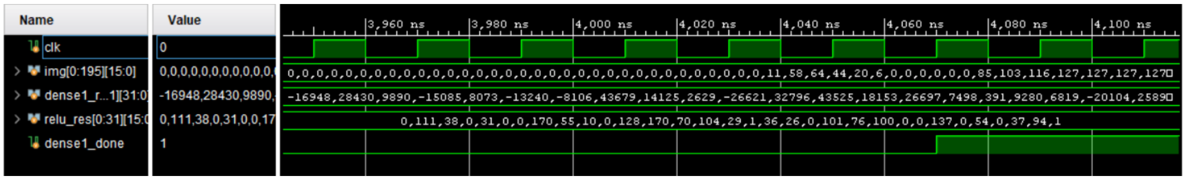

Figure 22: Dense 1 for image 27

3.3 Second_dense_layer test bench

Since the second dense layer is using the same exact modules as the first one, and is written in the same way, we can expect it to be working just as good as the previous one. Figure 24 presents the transition from one layer to another (layer enable signals switching), from the neural_network testbench (see section 3.5).

3.3.1 Simulation

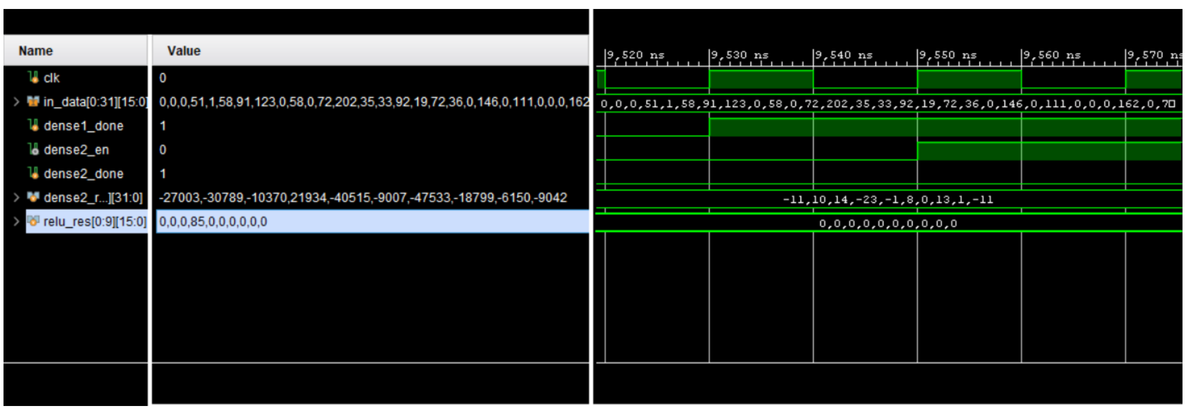

Figure 23: Switch from dense 1 to 2

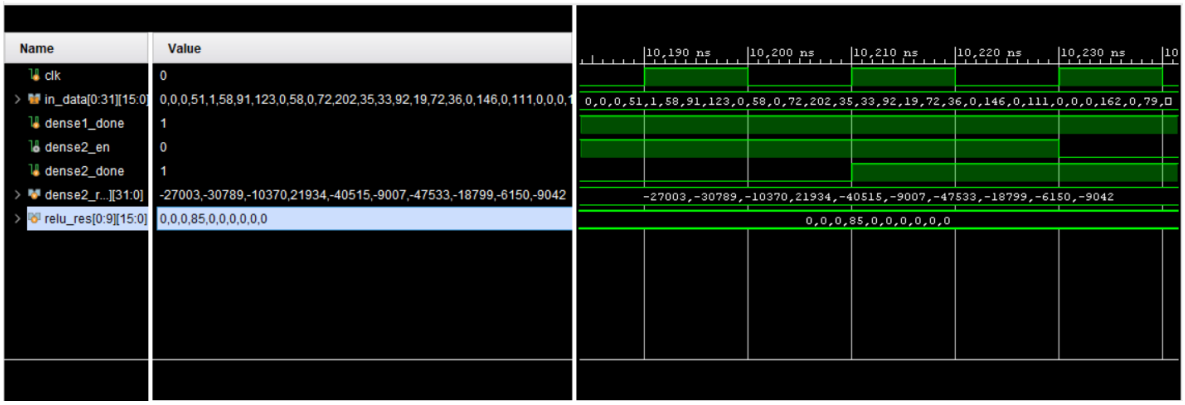

Figure 24: Dense 2 simulation

3.4 Select_max

3.4.1 Testbench

To test our module we use the following testbench.

```
module select_max_tb;

logic signed [15:0] test_data_a [0:9] = {0,0,5,85,0,10,0,0,0,0};

localparam period = 20;
reg clk, enable, reset, done;
reg signed [15:0] in_data [0:9];
reg [7:0] digit;

select_max s_max(.clk(clk), .reset(reset), .enable(enable), .in_data(in_data),
                 .digit(digit), .layer_done(done));

initial begin
clk = 0;
enable = 0;
reset = 0;
in_data = test_data_a;
#145
enable = 1;
#200
reset = 1;;
#50
reset = 0;
end

always begin
#10 clk = ~clk;
end

endmodule
```

Figure 25: select_max testbench

3.4.2 Simulation

The simulation results for the module select_max are shown in Figure 26. The search for a maximum can be observed: the input vector called in data has for maximum the value 85. When the index becomes 3, we can see that one cycle later, the maximum is updated to the value 85, and digit takes the value 3 (associated index). We can also see that the reset works: the done signal is not set to 1 and the entire maximum search starts again.

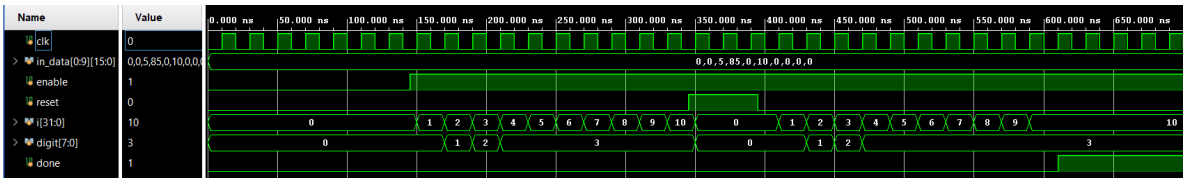

Figure 26: Search of maximum

3.5 Neural_Network testbench

This last part tests all previous modules together. If the interfacing works, the NN result should be the handwritten digit in the initial picture.

3.5.1 Testbench

The neural network is the last module to test. The associated testbench is presented in Figure 27. The image to be processed is the same matrix as previously, and is called test_data_a. Another one has been tested (test_data_b) with different values representing the number 2. The result was consistent, indicating that our module is properly working.

```
module NN_tb;

    logic signed [7:0] test_data_a [0:783] = '{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 40, 103, 127, 127, 127,
    127, 48, 40, 40, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 19, 79, 79, 79, 84, 126,
    126, 126, 126, 126, 126, 126, 126, 126, 105, 19, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    113, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 120, 73, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 69, 126, 126, 126, 119, 56, 107, 126, 126, 126, 126, 126,
    126, 126, 126, 126, 105, 21, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 19, 17, 17, 17, 15, 0, 15,
    74, 17, 102, 117, 126, 126, 126, 126, 126, 118, 32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 45,
    0, 0, 0, 0, 0, 0, 0, 0, 17, 99, 126, 126, 126, 126, 122, 40, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 5, 16, 101, 101, 108, 126, 126, 126, 126, 120, 44, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 83, 126, 126, 126, 126, 126, 126, 126, 119, 41, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 13, 126, 126, 126, 126, 126, 126, 126,
    126, 48, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 100, 126, 126, 126,
    126, 126, 126, 126, 115, 24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 18,
    43, 43, 43, 124, 126, 126, 126, 126, 69, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 3, 76, 126, 126, 126, 125, 29, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 31, 119, 126, 126, 126, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 116, 126, 126, 75, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 18, 101, 126, 126, 126, 69, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 33, 105, 105, 105, 29, 18, 18, 10, 13, 18, 75, 111, 126, 126, 126, 126, 69, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 40, 126, 126, 126, 126, 126, 126, 97, 107, 126, 126, 126, 126, 126,
    126, 78, 38, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 40, 126, 126, 126, 126, 126, 126, 126, 126,
    126, 126, 126, 126, 118, 117, 20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 24, 78, 123, 126,
    126, 126, 126, 126, 126, 126, 126, 79, 78, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 58, 126, 126, 126, 126, 126, 63, 39, 39, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

localparam period = 20;

reg clk, enable;
reg signed [7:0] img [0:783];
reg [7:0] digit_out;
reg NN_done;

neural_network NN(.clk(clk), .enable(enable), .img(img),
                  .digit_out(digit_out), .NN_done(NN_done));

initial begin
clk = 0;
enable = 0;
img = test_data_a;
#145
enable = 1;
end

always begin
#10 clk = ~clk;
end

endmodule
```

Figure 27: Neural_network testbench

The testbench only consists in loading the module neural_network with our vector (synchronized to the clock signal clk) and enabling the module later, to make sure that everything is working as intended.

3.5.2 Simulation - Final results

The initial image represents the number : 3. So we expect to obtain this result at the end of the simulation. Figure 28 shows the result found and it is exactly 3.

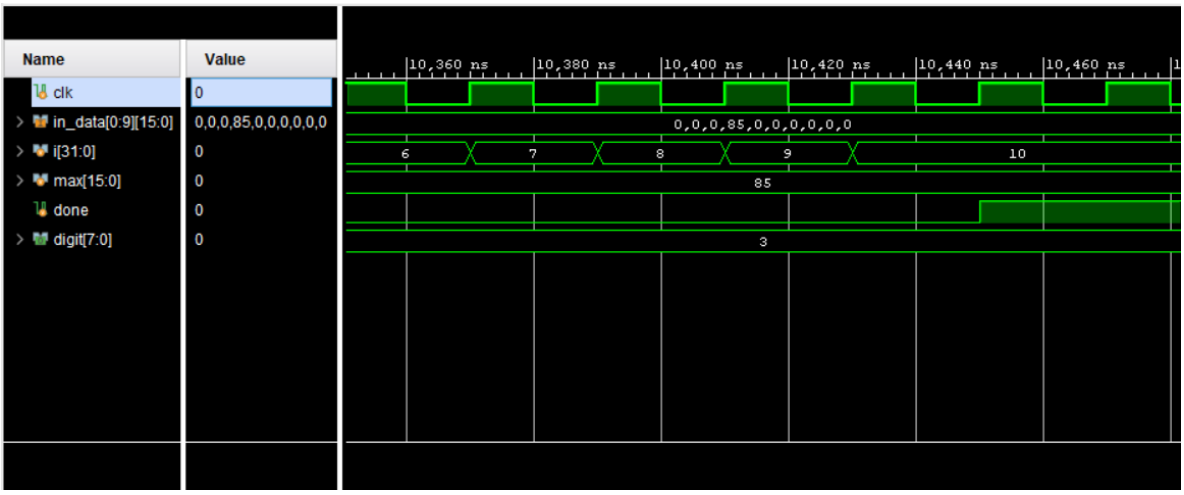

Figure 28: Final Result: 3

4 Implementation

We tried to implement our module into the Pynq board. Sadly, we did not succeed but we still present our results and progress in the following subsections.

4.1 Turning our module into an IP block

The first thing to do was to transform our module into an IP block, to add it into the design block of our board.
To do so, we used the tool "Create and package new IP", and choose to make an AXI4 IP block (the same as the others used in the Pynq board design). Vivado asked us how much registers our IP should contain and we chose 6:

- One for the control signals: enable and reset
- One for the input pixel value
- One for the pixel address
- One to indicate if the pixel has been stored in memory
- One to indicate if our NN is done
- And finally one for the output digit

Indeed, it is easier to use an address register and a value register rather than have 784 register, one for each pixel (and it would not have been possible either, the maximum number of registers being 512).
We used the tutorial in [3] to write our own custom IP block.

NB: We chose to separate the 'done' signals from the control signals by using 2 other registers. Since those signals are only 1 bit wide, we could have as well used the same register as the enable and reset.

4.1.1 User logic

In the associated neural_network_v1_0_S00_AXI file, we added our own logic, ie our neural network module.

```
// Add user logic here

neural_network NN (.clk(S_AXI_ACLK),.enable(slv_reg0[0]),.reset(slv_reg0[1]),
                   .img(img),.digit_out(digit),.NN_done(NN_done));

always @(posedge S_AXI_ACLK) begin
    img[slv_reg1] <= slv_reg2;
end
assign pixel_stored = (img[slv_reg1] == slv_reg2)? 1:0;
// User logic ends
```

Figure 29: User logic implementation

We decided to associate the first register slv_reg0 to the interfacing signals, the 2nd to the address of the pixel and the 3rd to the pixel value. The first bit of slv_reg0 sets the enable value while the second bit sets the reset value.

4.1.2 Output registers

We declared some wires: pixel_stored, digit_out and NN_done, one to signal when the pixel value is stored in the memory, the other two, to be connected to the NN associated outputs.
These wires are then used to assign the values of the last registers.

```
else begin
  slv_reg3 <= pixel_stored;
  slv_reg4 <= digit;
  slv_reg5 <= NN_done;
```

Figure 30: IP output registers

4.2 Adding the IP to our design

With that being done, it seems possible to interact with our module by writing in and reading from the right registers (all spaced by a 0x04 offset). We then encapsulated our IP block and finally added it to the Pynq block design.

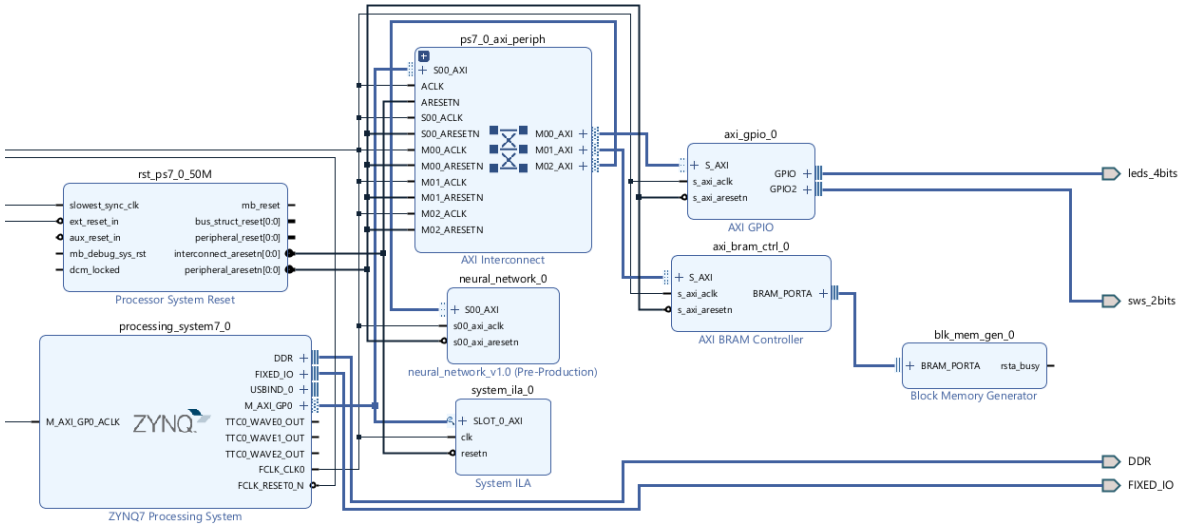

Figure 31: Final Pynq block design

After writing the bitstream and creating the new overlay (as it had been done in the pynq lab), we can see that the board recognize the new IP block.

```
Type:            Overlay
String form:     <pynq.overlay.Overlay object at 0xb387bb50>
File:            /usr/local/share/pynq-venv/lib/python3.10/site-packages/pynq/overlay.py
Docstring:
Default documentation for overlay pynq_lab.bit. The following
attributes are available on this overlay:

IP Blocks
----------
axi_gpio_0           : pynq.lib.axigpio.AxiGPIO
neural_network_0     : pynq.overlay.DefaultIP
processing_system7_0 : pynq.overlay.DefaultIP
```

Figure 32: Pynq overlay

Nevertheless, it seems that our IP block does not properly implement our neural network. Using the ILA module we checked that we were able to write the data into our img array. Still, after enabling the NN by writing the value 1 to slv_reg0 (offset 0x00), we never receive the NN_done signal and are stuck forever in our while loop.

```
Entrée [2]: enable=0x00
            pixel_addr=0x04
            pixel_val=0x08
            pixel_done=0x0C
            digit_out=0x10
            NN_done=0x14

Entrée [9]: nn=overlay.neural_network_0
            nn?

Entrée [10]: test_vector = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             addr = 0
             pixel = test_vector[addr]

             nn.write(enable,0)
             nn.write(pixel_addr,addr)
             nn.write(pixel_val,pixel)
             for addr in range(1,len(test_vector)):
                 while(nn.read(pixel_done) != 1):
                     sleep(1)
                 pixel = test_vector[addr]
                 nn.write(pixel_addr,addr)
                 nn.write(pixel_val,pixel)

Entrée [11]: nn.write(enable,1)
             nn.write(enable,1)
             output_digit=0
             while(nn.read(NN_done)!=1):
                 sleep(1)
             output_digit=nn.read(digit_out)

             ---------------------------------------------------------------------------
             KeyboardInterrupt                         Traceback (most recent call last)
```

Figure 33: NN Jupyter interface

5 Conclusion

In this report we explored the implementation of a neural network on the Pynq board. We learnt the basic architecture of a neural network, how to implement it in verilog and how to create a custom IP block.
Sadly, we did not succeed in interfacing the module with the board. The subject in itself was nevertheless very interesting. The development of AI has accelerated in recent months and it is clear that it will be more and more present. Its implementation in limited hardware (be it because of power or memory) is a growing subject. And FPGAs have the benefits of consuming less power, being more flexible and sometimes even faster than GPUs [5].

References

[1] Contributors to Wikimedia projects. (2001, 2nd October). Artificial neural network - Wikipedia. Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Artificial_neural_network

[2] What are Neural Networks ? — IBM. (s. d.). IBM - Deutschland — IBM. https://www.ibm.com/topics/neural-networks

[3] Creating a new Verilog Module Overlay. (2020, July). RogerPease - Pynq. https://discuss.pynq.io/t/tutorial-creating-a-new-verilog-module-overlay/1530

[4] FPGA handwritten digit recognition. (2022, January). P. Devouge, J. Richard - Github. https://github.com/PCov3r/FPGA_Handwritten_digit_recognition

[5] Deep Learning Hardware: FPGA Vs. GPU (2018, 20th December). Farhad Fallahlalehzari - Semiconductor Engineering. https://semiengineering.com/deep-learning-hardware-fpga-vs-gpu/