FH | JOANNEUM
Electronics and
Computer Engineering

Implementation of Neural Network Structure in FPGA Environment

Master Thesis

written by

Christoph Polster

at the master degree programme
Electronics and Computer Engineering
of FH JOANNEUM – University of Applied Sciences, Austria

supervised by FH-Prof. DI Dr. Christian Netzberger

Graz, 2025-09-30

Obligatory signed declaration

I hereby confirm and declare that the present Master's thesis was composed by myself without any help from others and that the work contained herein is my own and that I have only used the specified sources and aids. The uploaded version is identical to any printed version submitted.

I also confirm that I have prepared this thesis in compliance with the principles of the FH JOANNEUM Guideline for Good Scientific Practice and Prevention of Research Misconduct.

I declare in particular that I have marked all content taken verbatim or in substance from third party works or my own works according to the rules of good scientific practice and that I have included clear references to all sources.

The present original thesis has not been submitted to another university in Austria or abroad for the award of an academic degree in this form.

I understand that the provision of incorrect information in this signed declaration may have legal consequences.

Graz, 2025 September 30

(Place, Date)

Signature

Abstract

The FH Joanneum Master's degree "Electronic and Computer Engineering" teaches students in the field of programming Field Programmable Gate Arrays (FPGAs) and artificial intelligence. Currently, a missing piece is a bridge that allows for deployment of simple neural networks on FPGAs. This thesis was created to fulfill this functionality.

The end product consists of a template VHDL file and a simple Python script which allows the creation of a custom VHDL file for a trained model of a neural network in Python.

Kurzfassung

Der Master Studiengang "Electronic and Computer Engineering" der FH Joanneum unterrichtet Studenten in den Bereichen der Programmierung von „Field Programmable Gate Arrays" (FPGAs) und künstlicher Intelligenz. Ein bisher abwesender Aspekt war eine Verbindung zwischen den beiden Fachrichtungen, welche die Anwendung von simplen neuralen Netzwerken auf FPGAs ermöglichen würde. Diese Arbeit soll diese Funktion zur Verfügung stellen.

Das Endprodukt besteht aus einer Vorlage-Datei in VHDL und einem einfachen Python-Skript, welches das Erstellen einer maßgeschneiderten VHDL-Datei basierend auf einem trainierten Model eines neuralen Netzwerks in Python ermöglicht.

Contents

1 Introduction

Machine learning and artificial intelligence play an important role in modern electronic applications. To learn its use, many students start off their programs as simple code in Python.

Adapting this code to fast processing units is difficult, since most existing tools have a high requirement for experience in the field to use or are expensive proprietary products.

1.1 Motivation

The FH Joanneum Master's degree "Electronic and Computer Engineering" teaches students in the field of FPGA programming in the language VHDL as well as in the field of machine learning and artificial intelligence via the "Keras" module in Python.

So far, no convenient tools exist to allow deployment of trained models on FPGAs. The goal of this Master's thesis is to create a bridge between simple neural network models in Python and deployment on FPGAs. This thesis will show how a neural network can be realized as a component in VHDL and how a trained model in Python can be converted to be deployed on such a neural network component.

1.2 State of the Art

There are some existing tools that can provide a somewhat similar solution to deployment

- Xilinx/AMD's "Vitis AI" requires its own extensive development environment and is therefore too unwieldly for simple applications[1].
- "hls4ml" is a Python package whose goals are similar to this thesis[2], in that it aims to create an IP core for an FPGA based on a given model in Python. However, it does so via HLS (high-level synthesis), which is currently not in the curriculum of the Master's degree.
- Xilinx/AMD's "Brevitas" is a Python package for optimizing the quantization of models[3]. It is part of a greater undertaking (FINN[4]) to develop and deploy quantized neural networks. Unfortunately, Brevitas' documentation is currently too sparse for effective use in this thesis.

[1] https://xilinx.github.io/Vitis-AI/3.5/html/index.html
[2] https://fastmachinelearning.org/hls4ml/intro/introduction.html
[3] https://github.com/Xilinx/brevitas
[4] https://xilinx.github.io/finn

1.3 Approaches

When implementing the function of a neural network from scratch, one of the primary decisions is in the choice of approach of how to perform the arithmetical calculations of the network.

For each node in the neural network, it is required that N multiplications and $N + 1$ additions be performed (depending on the number of nodes N in the previous layer, see Chapter 2.1).

Assuming a neural network consisting of L layers, with the largest layer containing N_{max} nodes, and each node requiring up to N_{max} multiplications as well as $N_{max} + 1$ additions, the number of required arithmetical operations (A_{OP}) behaves as follows

$$A_{OP} < L \cdot N_{max}(2N_{max} + 1) \qquad (1\text{-}1)$$

It is evident that the limiting behavior of the number of required arithmetical operations is $O\left({N_{max}}^2\right)$.

When creating a process for calculating the output of a neural network, the implementation of the required arithmetical calculations may vary on the resources available.

Variant A (100% resource sharing)

100% of the calculations are performed by the same resource. Therefore, the computation time of the network is directly proportional to the number of required arithmetical calculations ($O\left({N_{max}}^2\right)$), while the resource requirement is minimal. This variant also requires memory to store intermediate and final results.

This variant is usually performed when using a single-core general purpose computer to calculate the output of a neural network. The computer uses a single CPU to perform all arithmetic calculations and stores/fetches the intermediate results in/from memory.

Variant B (0% resource sharing)

0% of the calculations share the same resource (each is performed on its own exclusive resource). Therefore, the computation time of the network is only proportional to the number of layers in the network (as the calculations still need to be performed in the correct order), while the hardware requirement is at the maximum ($O\left({N_{max}}^2\right)$). This variant requires no memory, as the output of each arithmetical calculation directly leads to the next operation.

This variant is possible to implement on an FPGA (or custom hardware), but since the hardware requirement grows quadratically with the number of nodes per layer, this tends to limit the network to smaller sizes.

Variant C (X% resource sharing)

Some (X%) of the calculations share the same resource. The calculations are divided and distributed to dedicated resources. Therefore, the computation time of the network is lower than Variant A, although higher than Variant B. The hardware requirement is lower than Variant B, although higher than Variant A. This variant will require some memory to store intermediate and final results, although less than Variant A.

This variant is possible to implement in FPGA and features both advantages and drawbacks from Variants A&B (e.g. a decrease in hardware requirements will lead to an increase in computation time). Overall, this variant requires careful consideration based on given constraints for the design.

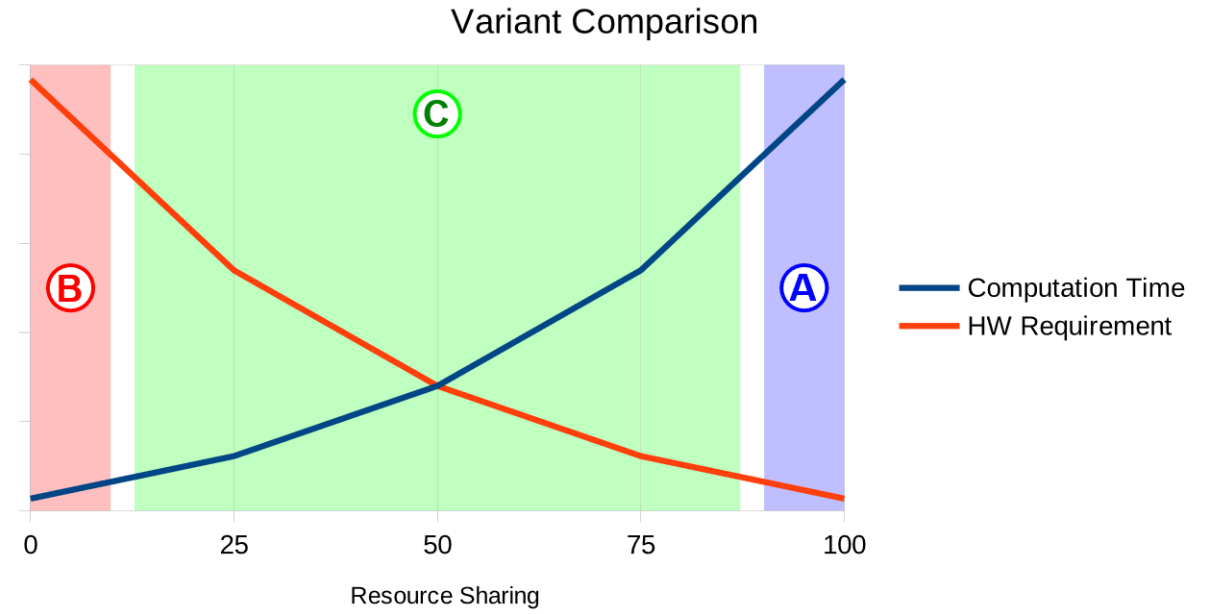


Fig. 1-1: Comparison of different approach variants.

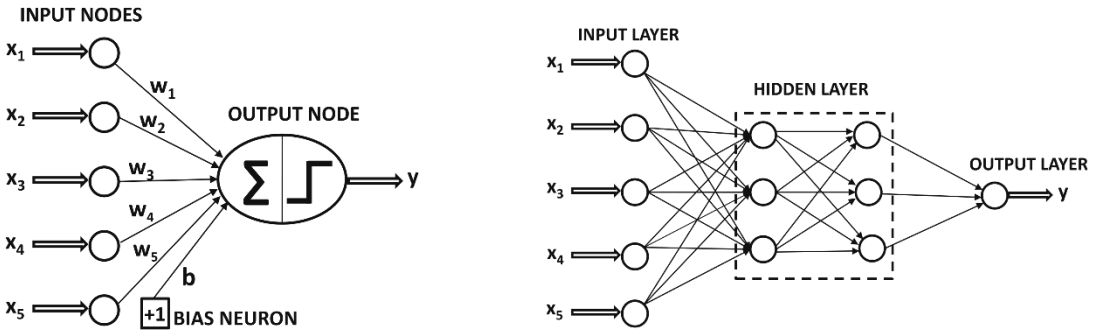
For the implementation of this thesis Variant C was chosen, as Variant A is already common practice on general purpose computers and Variant B is prone to quickly run into hardware limitations.

The resource sharing was chosen such that all calculations concerning a single node are performed by one exclusive resource (DSP slice, see Chapter 2.2). This reduces the previously quadratic growth in hardware requirement due to number of nodes N to linear growth ($O(N)$). In turn, computation time increases from being proportional to the number of layers L, to growing linearly with the number of nodes N (also $O(N)$, usually $N \gg L$).

2 Background

2.1 Neural network

A neural network is a term in the field of artificial intelligence which describes a structure consisting of "layers" which themselves consist of "nodes" (or "neurons").



(a) **(b)**

Fig. 2-1: a) Simplified model of a neuron with several input nodes and corresponding weights w_i as well as a bias value b [1, p. 5], b) model of a multi-layer neural network consisting of one input layer, one output layer and two hidden layers [1, p. 18].

The smallest part of a neural network is the "node" (Fig. 2-1a), which performs multiplication and summation on its inputs based on weights and bias specific to this node. This is followed by an activation function, which usually introduces non-linear behavior to the calculated result. Non-linearity is necessary to enable computational complexity in neural networks. [1]

$$y = f_{\text{activation}}\left(b + \sum_{i}^{N} x_i \cdot w_i\right) \tag{2-1}$$

A "layer" consists of a given number of nodes which work in parallel on the outputs of the previous layer. Starting from the "input layer", each layer performs its calculation and then "feeds" its outputs to the next layer until the final "output layer" (a "feed forward" network).

A neural network can consist of a single layer, or multiple layers where the first is referred to as the "input layer" and the last as the "output layer". The remaining layers are "hidden layers" as only the values at input and output layers are usually "visible".

In a “fully connected” neural network, all nodes of each layer are connected to each node of the previous layer. This will also be referred to as a “Dense” layer (see Chapter 2.4). [1]

2.2 What is an FPGA

FPGA stands for Field-Programmable Gate Array. It is a configurable integrated circuit consisting of many gates which can be programmed and reprogrammed after production of the device.

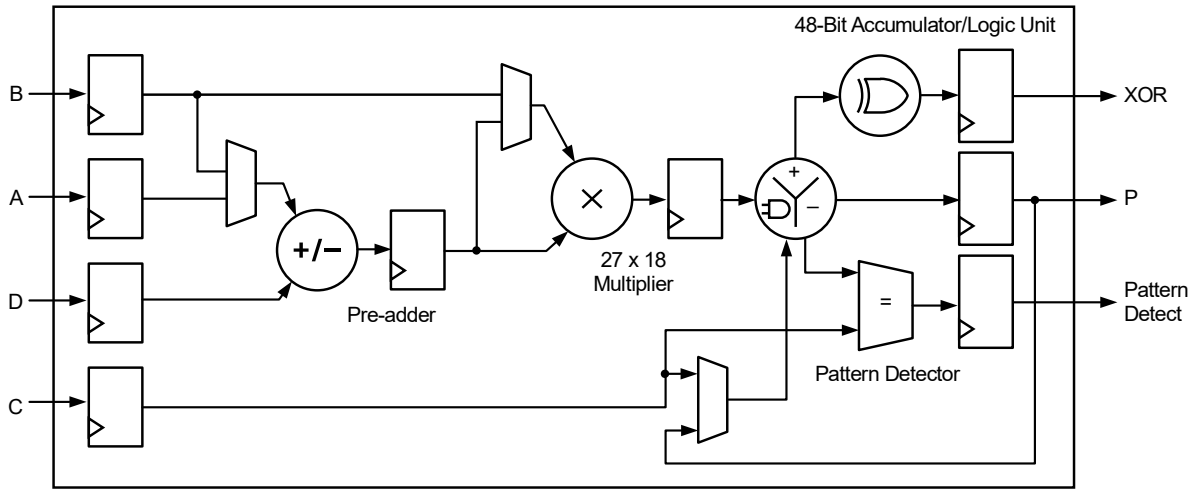


Fig. 2-2: DSP Slice Functionality – DSP48E2 [2].

A Zynq-7 series Artix-7 is an FPGA which features 215K logic cells and 740 DSP slices. A Digital Signal Processing (DSP) slice is a dedicated resource in the FPGA which implements common operations for digital signal processing [2], such as multiplication and addition.

The number of DSP slices is an important factor when choosing the size of the neural network as at least one DSP slice is required per node in the network.

2.3 What is VHDL

VHDL stands for “Very High Speed Integrated Circuit” Hardware Description Language. [3] As a hardware description language, VHDL describes which behavior and structure a component exhibits, without choosing a specific underlying fabrication technology. [4]

There are other hardware description languages (Verilog, SystemVerilog), but VHDL was chosen due to it being taught in the FH Joanneum Master’s degree “Electronic and Computer Engineering”, thus being familiar to students who would use the results of this thesis to test their own neural networks on FPGAs.

2.4 Python

Python is an open-source, high-level, general purpose programming language. [5] [6]

It is widely used for the task of machine learning, and there are many possible modules which support the design of neural networks. Due to their widespread use, Keras and PyTorch were chosen as the modules for which to create implementations for.

2.5 Neural networks in Python/Keras

Keras is an open-source API for creating and optimizing neural networks in Python. [7] [8]

Code 2-1: Example Keras usage – Simple two layer neural network with 2 nodes at the input layer, 4 nodes in the hidden layer and 1 node at the output layer.

```
1  from keras.models import Sequential
2  from keras.layers import Dense
3
4  model = Sequential()
5
6  model.add(Dense( units = 4, activation = 'relu', input_shape = (2,) ))
7  model.add(Dense( units = 1, activation = 'sigmoid' ))
```

An important feature of Keras is the “Sequential” model from the “keras.models” module. This will allow us to create a simple “feed forward” network by adding layer by layer. This allows us to infer the structure of the neural network and recreate it in VHDL later.

In Code 2-1 we create a “Sequential” model and add two “Dense” (or fully connected) layers to it, with “ReLu” and “sigmoid” as activation functions. The resulting model has two input nodes and one output node.

Code 2-2: Console output of Code 2-1.

```
model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 4)                 12
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 5
=================================================================
Total params: 17
Trainable params: 17
```

2.6 Neural networks in Python/PyTorch

PyTorch is an open-source API for creating and optimizing neural networks in Python. [9] [10]

Code 2-3: Example PyTorch usage – Simple parametrized, two layer neural network.

```
1  import torch
2  import torch.nn as nn
3
4  linear_stack = nn.Sequential(
5      nn.Linear(num_inputs, num_hidden),
6      nn.ReLU(),
7      nn.Linear(num_hidden, num_outputs),
8      nn.Hardsigmoid(),
9  )
```

We make use of the "Sequential" model of the "torch.nn" module. This constrains the network to consist of a sequence of layers (fully connected layer, activation function, max pooling, etc.) which are applied to the input, one after the other. That creates a simple "feed-forward" network, which is the target of this thesis.

While PyTorch offers different ways to create a feed-forward network, use of the "Sequential" model also allows us to infer the structure of the network later, when we convert the model to VHDL code.

In Code 2-3, we create a stack of sequential layers. It consists of two layers with "ReLu" and "hard sigmoid" as activation functions. The "linear" (or fully connected) layers and activation layers must be added in the correct order.

Code 2-4: Console output of Code 2-3.

```
print(linear_stack)
Sequential(
  (0): Linear(in_features=2, out_features=4, bias=True)
  (1): ReLU()
  (2): Linear(in_features=4, out_features=1, bias=True)
  (3): Hardsigmoid()
)
```

2.7 Fixed-point representation

When computational speed is not a priority (in prototyping or education), then neural networks are usually implemented in floating point representation. When computational speed becomes a priority, then the model must be adapted to an integer representation, since the mathematical operations become too costly in time or hardware, while these operations are cheaper to implement in programmable logic.

Rather than converting all weights and biases to their nearest integer value, the model is converted to "fixed-point precision." This retains some of the precision of float representation as all numbers are implicitly divided by a common quotient.

2.7.1 Standard number representation

$\pi \approx 3.14159265358979323846$

2.7.2 Floating-point representation[5]

0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

Binary: $\pi_2 = (-1)^{0} \cdot 10^{(10000000-01111111)} \cdot 1.10010010000111111011011$

Decimal: $\pi_{10} = (-1)^{0} \cdot 2^{(128-127)} \cdot 1.570796370506286621093 75$

$\pi \approx 3.14159274101257324219$

2.7.3 Fixed-point representation (Q-Format)

Q-Format (as defined by Texas Instruments [11] "A.2 Fractional Q Formats") represents a rational number by the number of integer bits `QI` and the number of fractional bits `QF`. The bit-width of a number in Q-Format is equal to `QI+QF+1`, as an extra bit is needed to store the sign of the number. [12]

$$Q[QI].[QF] \tag{2-2}$$

0		0	0	0	0	0	1	1	0	0	1	0	0	1	0	0
(-1)		2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$\pi \approx 3.140625$

In this case Q7.8 is a fixed-point number with a bit-width of 16.

While precision is trimmed to a fixed range, as opposed to floating-point representation, fixed-point arithmetic operations are integer arithmetic operations, which can be implemented at a much lower cost of area in programmable logic and is much faster than equivalent floating-point arithmetic operations.

2.7.4 Fixed-point arithmetic

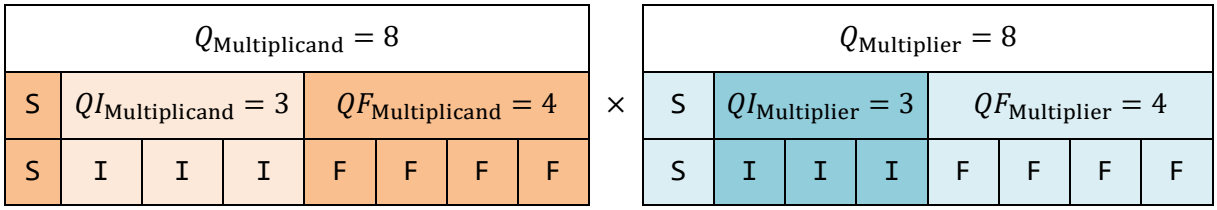
Addition for fixed-point numbers is the same as integer addition, as long as both numbers share the same fixed-point representation (which is the case for the matter of this thesis).

[5]Visualization of floating-point representation: https://float.exposed/0x40490fdb

Multiplication for fixed-point numbers is the same as integer multiplication, with an additional shift (or slice) of the multiplication result.

As with integer multiplication, the overall bit-width of the product is equal to the sum of bit-widths of the multiplicand and multiplier. The same goes for the integer and fractional bit-widths. [12]

Example: Multiplication of two Q3.4 fixed point numbers

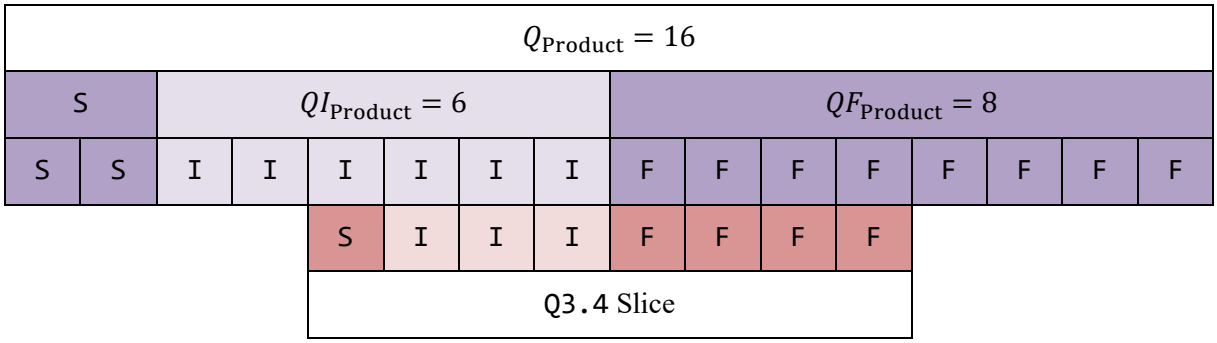


$$Q_{\text{Product}} = Q_{\text{Multiplicand}} + Q_{\text{Multiplier}} = 8 + 8 = 16 \tag{2-3}$$

$$QI_{\text{Product}} = QI_{\text{Multiplicand}} + QI_{\text{Multiplier}} = 3 + 3 = 6 \tag{2-4}$$

$$QF_{\text{Product}} = QF_{\text{Multiplicand}} + QF_{\text{Multiplier}} = 4 + 4 = 8 \tag{2-5}$$

It is noteworthy that this also results in two sign bits.



To gain the result in the original fixed-point representation, a slice of the result must be performed at the correct position.

This is based on the underlying assumption that the integer part of the result is not too large to fit into the integer width of the original fixed-point representation, or this will cause an overflow calculation error. Similarly, it is also assumed that the magnitude of the result is not too small to fit into the factional width of the original fixed-point representation.

It is assumed that during the design of the model, its weights and biases are driven in appropriate ranges such that this does not occur. Otherwise, larger bit-widths could be considered to ensure that the resulting node values are properly stored (see Chapter 3.1).

3 Implementation VHDL

This implementation aims to create an entire layer as the basic building block for recreating the neural network in VHDL. This `Layer-01` performs the calculation of the corresponding layer in the neural network and then provides its results for the next layer or as an output of the neural network.

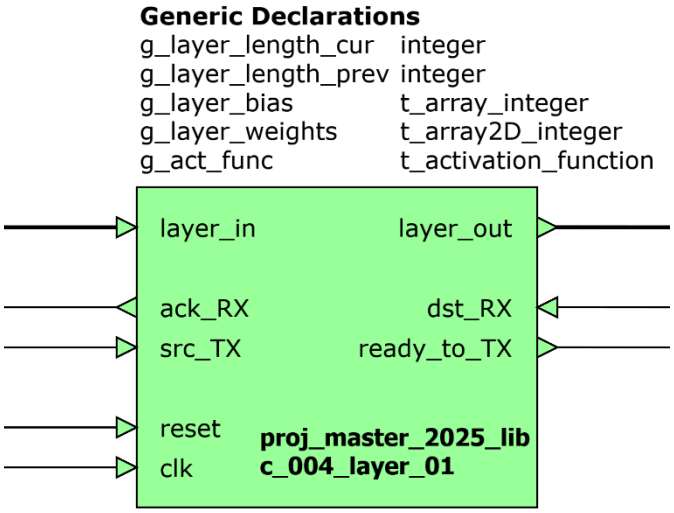


Fig. 3-1: Block diagram of Layer-01, showing inputs, outputs and generics. This component executes the calculations of the corresponding layer in the neural network.

The generics define the entirety of the behavior of `Layer-01`. The number of nodes in `Layer-01` as well as the number of the previous layer define the number of weights and biases of `Layer-01`.

3.1 Required package

In order to use custom types and constants, a package was created for this thesis (see Appendix I.A). This package is required for the generated VHDL-code of `Layer-01` to work.

Code 3-1: Bit-width definitions (p_002_generic_01_pkg.vhd, Appendix I.A).

```
16  constant c_DATA_WIDTH : integer := 16;   -- {$c_DATA_WIDTH}
17  constant c_DATA_QF    : integer := 8;    -- {$c_DATA_QF}
```

An important attribute defined in this package is the used bit-width of the project. This determines the fixed-point representation used in the calculation of `Layer-01` at `Q7.8`.

There may be advanced applications for which a fixed bit-width will not work well, but for the scope of this thesis it will suffice.

3.2 Inputs

Reset (reset)

Resets `Layer-01` into a defined reset state (see Chapter 3.5.1).

Clock (clk)

Clk signal for synchronous execution.

Layer input (layer_in)

A parallel bus of all node values from the previous layer or the inputs of the network.

Source Transmit (src_TX)

Signal that indicates that the source of `Layer-01` (either input or previous layer) is transmitting a valid value for calculation ("layer_in"). `Layer-01` acknowledges it with a strobe from "ack_RX."

Destination Receive (dst_RX)

This signal indicates that the destination of `Layer-01` (either output or next layer) has successfully received the current output of `Layer-01` ("layer_out"). This is normally a single strobe. Only after the output of `Layer-01` has been acknowledged by this signal, can the next calculation begin.

3.3 Outputs

Layer output (layer_out)

A parallel bus of all node values of `Layer-01`.

Acknowledge data received (ack_RX)

This signal indicates that `Layer-01` has successfully read the node values from its source (at "layer_in"). It is sent as a single strobe, when the signal "src_TX" input signal indicates that the source of `Layer-01` is ready to deliver new node values.

Ready to transmit data (ready_to_TX)

This signal indicates that the calculation of `Layer-01` is complete and ready to be transmitted to the destination of `Layer-01`. The destination normally acknowledges it with a strobe on "dst_RX".

3.4 Layer control procedure

The control procedure of Layer-01 is a two-process state machine, consisting of a process for "next-state logic" and a process for "current-state update". An example of this can be found in [13, p. 226].

3.4.1 Next-state logic

The first process is "next-state logic", which handles the actions inside a state, entering or leaving a state, as well as transitioning to the next state.

Code 3-2: State machine process, "next-state logic" (c_004_layer_01_rtl.vhd, Appendix I.B).

```
 66  P_STM : process(CUR_state, CUR_node_prev, src_TX, dst_RX)
 70  begin
 84    case(CUR_state) is
 86      when RESET_STATE =>
           [...]
 98      when IDLE_TX =>
           [...]
107      when IDLE_RX =>
           [...]
120      when BIAS_SETUP =>
           [...]
137      when MAC =>
           [...]
153      when ACT_FUNC =>
           [...]
220      when others =>
221        -- default catch others
222        NEX_state <= RESET_STATE;
223    end case;
224  end process;
```

Internal signals of the process' sensitivity list include

- CUR_state: The current state of the state machine.
- CUR_node_prev: The currently processed node index of the previous layer.

Input signals of the process' sensitivity list include:

- src_TX: Signal from the previous layer which indicates if new input data is available
- dst_RX: Signal from the next layer which acknowledges the output of this layer

In this process changes are exclusively applied "next"/NEX_ signals. These signals are updated in the second process ("current-state update"). All calculations of Layer-01 are performed in the "next-state logic" process.

3.4.2 Current-state update

The second process is “current state update”, which is a clocked process that updates all signals that are being overwritten by the “next-state” signals.

Code 3-3: State machine process, “current-state update” (c_004_layer_01_rtl.vhd, Appendix I.B).

```
226 P_CLK : process(clk)
227 begin
228   if rising_edge(clk) then
229     if reset = '1' then
230       -- internal signals
231       CUR_state <= RESET_STATE; -- default state: reset
232       CUR_node_prev <= 0;
233 
234       CUR_data_in   <= (others=>(others=>'0'));
235       CUR_data_accum <= (others=>(others=>'0'));
236 
237       -- outputs
238       ready_to_TX <= '0';
239       ack_RX <= '0';
240       layer_out <= (others=>(others=>'0'));
241     else
242       -- internal signals
243       CUR_state     <= NEX_state;
244       CUR_node_prev <= NEX_node_prev;
245 
246       CUR_data_in   <= NEX_data_in;
247       CUR_data_accum <= NEX_data_accum;
248 
249       -- outputs
250       ready_to_TX <= NEX_ready_to_TX;
251       ack_RX <= NEX_ack_RX;
252       layer_out   <= NEX_layer_out;
253     end if;
254   end if;
255 end process;
```

In this process changes are exclusively applied to internal “current”/CUR_ signals and output signals.

In case of a reset, all “current” internal signals and output signals are set to a default value. Otherwise, the “current” internal signals and output signals are updated to their previously scheduled “next” value.

3.5 Finite State Machine

The “next-state logic” process deals with the actions of the various states of the state machine, and how these states are transitioned.

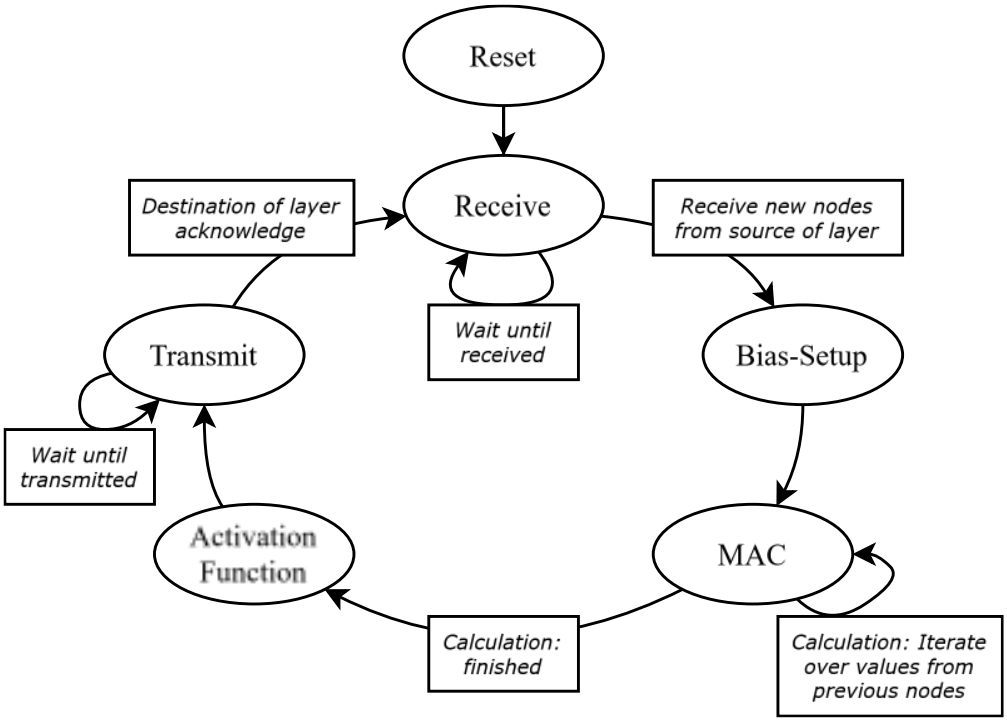


Fig. 3-2: Finite State Machine of Layer-01.

3.5.1 Reset

Starting from "Reset", Layer-01 goes to "Receive" state.

Code 3-4: State machine process – Reset state (c_004_layer_01_rtl.vhd, Appendix I.B).

```
85  -- Default reset state
86  when RESET_STATE =>
87    -- default: goto Receive state
88    NEX_state <= IDLE_RX;
89
90    NEX_node_prev <= 0;
91    NEX_data_in   <= (others=>(others=>'0'));
92    NEX_data_accum <= (others=>(others=>'0'));
93    NEX_ready_to_TX <= '0';
94    NEX_ack_RX <= '0';
95    NEX_layer_out <= (others=>(others=>'0'));
```

The "Reset" state sets all "next state" signals to a defined value. Otherwise, they would be undefined until they are eventually set for the first time in the other states of the state machine.

3.5.2 Receive

Layer-01 stays in "Receive" state until the layer receives new input data from its source (previous layer). Layer-01 acknowledges this and stores these input values. Then the next state is "Bias_Setup".

Code 3-5: State machine process – Receive state (c_004_layer_01_rtl.vhd, Appendix I.B).

```
106  -- we wait until we receive new data
107  when IDLE_RX =>
108    if src_TX = '1' then
109      NEX_ack_RX <= '1'; -- set to 1 during transition
110
111      --NEX_data_in <= layer_in; -- store input
112      for idx in layer_in'RANGE loop
113        NEX_data_in(idx) <= SIGNED(layer_in(idx));
114      end loop;
115
116      NEX_state <= BIAS_SETUP;
117    end if;
```

When in "Receive" state and source indicates, it is ready to transmit new input data (src_TX = '1'), then Layer-01 acknowledges this and processes the new input data:

- Next "ack_RX" is set to '1' to indicate that the data was received (will be reset in the following state).
- The new input values of the previous layer are cast to "signed" and stored in next "data_in".

3.5.3 Bias-Setup

The internal signals of the current node values are reset to their bias values. Then the next state is "Multiply-Accumulate".

Code 3-6: State machine process – Bias Setup state (c_004_layer_01_rtl.vhd, Appendix I.B).

```
119  -- we fill our accummulators with bias value
120  when BIAS_SETUP =>
121    NEX_ack_RX <= '0'; -- reset
122
123    -- initialize bias in layer nodes
124    LOOP_BIAS : FOR idx_node_cur in 0 to (g_layer_length_cur-1) LOOP
125      -- we create first entry:
126      NEX_data_accum(idx_node_cur) <=  SHIFT_LEFT(TO_SIGNED(
     g_layer_bias(idx_node_cur), 2*c_DATA_WIDTH), c_DATA_QF);
127    end LOOP;
128
129    NEX_state <= MAC;
132    NEX_node_prev <= g_layer_length_prev-1;
```

We reset the next "ack_RX" to '0', which was set as exit action in "Receive" state. Then all entries of the array containing the node values of Layer-01 (next "data_accum") are filled with their bias values.

This is achieved by loading the bias values from their integer generics, converting them to double width "signed" (width of multiplication results, see Chapter 2.7.4) and shifting them left by the number of fractional bits (to ensure compatibility with the multiplication results).

The next index of the nodes from the previous layer ("node_prev") is set to its starting position (starts at maximum and decrements to zero), as preparation for the next state.

3.5.4 Multiply-Accumulate

This state performs the calculation of the node values. It is reentered for all node values from the previous layer, as each time they are multiplied with their corresponding weight and accumulated. Then the next state is "Activation function".

Code 3-7: State machine process – Multiply-Accumulate (c_004_layer_01_rtl.vhd, Appendix I.B).

```
134  -- we accumulate the node values times their weights
135  when MAC =>
136    -- loop through nodes of THIS LAYER
137    LOOP_Node : FOR idx_node_cur in 0 to (g_layer_length_cur-1) LOOP
140      NEX_data_accum(idx_node_cur) <= CUR_data_accum(idx_node_cur) +
     TO_SIGNED(g_layer_weights( idx_node_cur, CUR_node_prev), c_DATA_WIDTH) *
     CUR_data_in(CUR_node_prev) ;
141    end LOOP;
142
143    -- decrement node of PREVIOUS LAYER
144    NEX_node_prev <= CUR_node_prev - 1;
145    -- exit if we have reached Zero
146    if CUR_node_prev = 0 then
147      NEX_state     <= ACT_FUNC;
148      NEX_node_prev <= 0;
149    end if;
```

A for-loop iterates over all nodes of this layer to calculate the next multiply-accumulate of this node based on the currently indexed node value from the previous layer. This state will then be called again with a decremented index to node value from previous layer. This continues until all node values from the previous layer have been evaluated.

It should be noted that the multiplication results are double the bit-width of the input values (see Chapter 2.7.4) and will be trimmed to fit the fixed-point representation in the "Activation function" state.

3.5.5 Activation function

In this state, the result of the multiply-accumulation process is evaluated by the chosen activation function.

Code 3-8: State machine process – Activation function state (c_004_layer_01_rtl.vhd, Appendix I.B).

```
151  when ACT_FUNC =>
153    case g_act_func is
154      when AF_SIGN =>
155        -- When: Sign Function
         [...]
```

```
171     when AF_RELU =>
172       -- When: ReLu Function
          [...]
183     when AF_HARD_SIGMOID =>
184       -- When: hard sigmoid Function
          [...]
208     when others =>
209       -- When: Identity Function
          [...]
213   end case;
214
215   NEX_ready_to_TX <= '1';
216   NEX_state <= IDLE_TX;
```

The activation function is selected based on the generic of the component. After performing the activation function, `Layer-01` indicates that it is ready to transmit its node values by setting next "ready_to_TX" to '1'. The next state is "Transmit".

Four activation functions are currently implemented:

- Sign function
- ReLu function (Rectified Linear unit)
- Hard-sigmoid function
- Identity function

Code 3-9: Constants for activation functions (c_004_layer_01_rtl.vhd, Appendix I.B).

```
57  CONSTANT c_pos_one : STD_LOGIC_VECTOR := STD_LOGIC_VECTOR(
    SHIFT_LEFT(TO_SIGNED(  1, c_DATA_WIDTH), c_DATA_QF) );
58  CONSTANT c_neg_one : STD_LOGIC_VECTOR := STD_LOGIC_VECTOR(
    SHIFT_LEFT(TO_SIGNED( -1, c_DATA_WIDTH), c_DATA_QF) );
59
60  CONSTANT c_s_dw_pos_3     : SIGNED :=  SHIFT_LEFT(TO_SIGNED(  3,
    2*c_DATA_WIDTH), 2*c_DATA_QF);
61  CONSTANT c_s_dw_neg_3     : SIGNED :=  SHIFT_LEFT(TO_SIGNED( -3,
    2*c_DATA_WIDTH), 2*c_DATA_QF);
62  CONSTANT c_s_pos_half  : SIGNED :=  TO_SIGNED( (2**(c_DATA_QF))/2,
    c_DATA_WIDTH );
63  CONSTANT c_s_pos_sixth : SIGNED :=  TO_SIGNED( (2**(c_DATA_QF))/6,
    c_DATA_WIDTH );
```

Several constants are defined at the beginning of the architecture, to allow for more readable code for the activation functions:

- c_pos_one ("STD_LOGIC_VECTOR"): +1 in fixed-point representation
- c_neg_one ("STD_LOGIC_VECTOR"): −1 in fixed-point representation
- c_s_dw_pos_3 ("signed"): +3 in double width fixed-point representation

- c_s_dw_neg_3 (“signed”): -3 in double width fixed-point representation
- c_s_pos_half (“signed”): $+\frac{1}{2}$ in fixed-point representation
- c_s_pos_sixth (“signed”): $+\frac{1}{6}$ in fixed-point representation

Sign function

Code 3-10: Activation function, Sign (c_004_layer_01_rtl.vhd, Appendix I.B).

```
156  when AF_SIGN =>
157    -- When: Sign Function
158    LOOP_AF_SIGN : FOR idx_node_this in 0 to (g_layer_length_cur-1) LOOP
159      -- check if the whole slice is Zero
160      if CUR_data_accum(idx_node_this)(CUR_data_accum(idx_node_this)'RANGE) =
     (CUR_data_accum(idx_node_this)'range => '0') then
161        -- is zero
162        NEX_layer_out(idx_node_this) <= (others => '0');
163      elsif CUR_data_accum(idx_node_this)(CUR_data_accum(idx_node_this)'HIGH)
     = '1' then
164        -- is negative
166        NEX_layer_out(idx_node_this) <= c_neg_one;
167      else
168        -- ELSE: is positive
169        NEX_layer_out(idx_node_this) <= c_pos_one;
170      end if;
171    end loop;
```

The value of each node in `Layer-01` is checked (“LOOP_AF_SIGN”). If it is filled with zeros, then the output is zero. Otherwise, for a negative node value (MSB equals '1'), the output is -1 and for positive node values (MSB equals '0') the output is $+1$.

$$\phi_{\text{sign}}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{if } x > 0 \end{cases} \tag{3-1}$$

ReLu function

Code 3-11: Activation function, ReLu (c_004_layer_01_rtl.vhd, Appendix I.B).

```
171  when AF_RELU =>
172    -- When: ReLu Function
173    LOOP_AF_RELU : FOR idx_node_this in 0 to (g_layer_length_cur-1) LOOP
174      if CUR_data_accum(idx_node_this)(CUR_data_accum(idx_node_this)'HIGH) =
     '1' then
175        -- is negative
176        NEX_layer_out(idx_node_this) <= (others => '0');
177      else
178        -- ELSE: is positive
179        NEX_layer_out(idx_node_this) <= STD_LOGIC_VECTOR(
     CUR_data_accum(idx_node_this)(c_DATA_WIDTH + c_DATA_QF - 1 downto
     c_DATA_QF) );
180      end if;
181    end loop;
```

The value of each node in Layer-01 is checked ("LOOP_AF_RELU"). If it is a negative node value (MSB equals '1'), the output is 0, otherwise the output is sliced from double width (result of multiplication, see Chapter 2.7.4) to single width and cast to "STD_LOGIC_VECTOR".

$$\phi_{\mathrm{ReLu}}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{3-2}$$

Hard sigmoid function

Code 3-12: Activation function, Hard sigmoid (c_004_layer_01_rtl.vhd, Appendix I.B).

```
183  when AF_HARD_SIGMOID =>
184    -- When: hard sigmoid Function
190    LOOP_AF_HARD_SIGMOID : FOR idx_node_this in 0 to (g_layer_length_cur-1)
       LOOP
191      if CUR_data_accum(idx_node_this) < c_s_dw_neg_3 then
192        -- IF is less than "-3": return "0"
193        NEX_layer_out(idx_node_this) <= (others => '0');
194      elsif CUR_data_accum(idx_node_this) > c_s_dw_pos_3 then
195        -- IF is greater than "+3": return "1"
196        NEX_layer_out(idx_node_this) <= c_pos_one;
197      else
198        -- ELSE: linear function: y = (x/6) + 0.5
199        v_dw_AF_temp1 :=
       signed(STD_LOGIC_VECTOR(CUR_data_accum(idx_node_this)(c_DATA_WIDTH +
       c_DATA_QF - 1 downto c_DATA_QF)));
200        v_dw_AF_temp2 := v_dw_AF_temp1 * c_s_pos_sixth;
201        v_dw_AF_temp1 := c_s_pos_half + v_dw_AF_temp2(c_DATA_WIDTH +
       c_DATA_QF - 1 downto c_DATA_QF);
202        NEX_layer_out(idx_node_this) <= STD_LOGIC_VECTOR( v_dw_AF_temp1 );
205      end if;
206    end loop;
```

The value of each node in Layer-01 is checked ("LOOP_AF_HARD_SIGMOID"). If the node value is less than -3, then the output is 0. If the node value is greater than $+3$, then the output is 1. Otherwise, the output is on a straight line function between $(0,1)$ for $-3 \leq x \leq +3$.

$$\phi_{\text{hard sigmoid}}(x) = \begin{cases} 0 & \text{if } x < -3 \\ \frac{x}{6} + \frac{1}{2} & \text{if } -3 \leq x \leq +3 \\ +1 & \text{if } x > +3 \end{cases} \tag{3-3}$$

Identity function

Code 3-13: Activation function, Identity (c_004_layer_01_rtl.vhd, Appendix I.B).

```
208  when others =>
209    -- When: Identity Function
210    LOOP_AF_IDENTITY : FOR idx_node_this in 0 to (g_layer_length_cur-1) LOOP
211      NEX_layer_out(idx_node_this) <= STD_LOGIC_VECTOR(
       CUR_data_accum(idx_node_this)(c_DATA_WIDTH + c_DATA_QF - 1 downto
       c_DATA_QF) );
212    end loop;
```

The value of each node in Layer-01 is checked ("LOOP_AF_IDENTITY"). In this case, only a slicing and casting to "STD_LOGIC_VECTOR" is necessary.

$$\phi_{\text{Identity}}(x) = x \tag{3-4}$$

For the activation functions "ReLu", and "Identity", the results of the accumulated multiplications are needed in their single-width fixed-point representations to create the output values. This requires that these results do not exceed the given bid-width of the model (see Chapter 2.7.4).

3.5.6 Transmit

Layer-01 stays in the "Transmit" state until it receives an acknowledge from its destination (next layer). Then the next state is "Receive".

Code 3-14: State machine process – Receive state (c_004_layer_01_rtl.vhd, Appendix I.B).

```
97  -- we send our result, until it is accepted
98  when IDLE_TX =>
99
100   if dst_RX = '1' then
101     -- goto reveicer mode
102     NEX_ready_to_TX <= '0';
103     NEX_state <= IDLE_RX;
104   end if;
```

Upon receiving the acknowledge from its destination, the next "ready_to_TX" is reset to '0'.

3.6 Testing

The functionality of a single Layer-01 with 2 inputs and 3 outputs was tested with the following parameters (inputs, outputs, weights and biases are in fixed-point representation, see Chapter 2.7):

Weights: $W = \begin{pmatrix} 40 & 4 \\ 90 & -14 \\ 0 & 0 \end{pmatrix}$

Biases: $\vec{b} = \begin{pmatrix} 51 \\ 5 \\ 0 \end{pmatrix}$

Activation: "Identity"

Input: $\vec{x} = \begin{pmatrix} -269 \\ 74 \end{pmatrix}$

$$\vec{y} = W\vec{x} + \vec{b} \tag{3-5}$$

The exact solution would be $(10.125, -93.617)$.

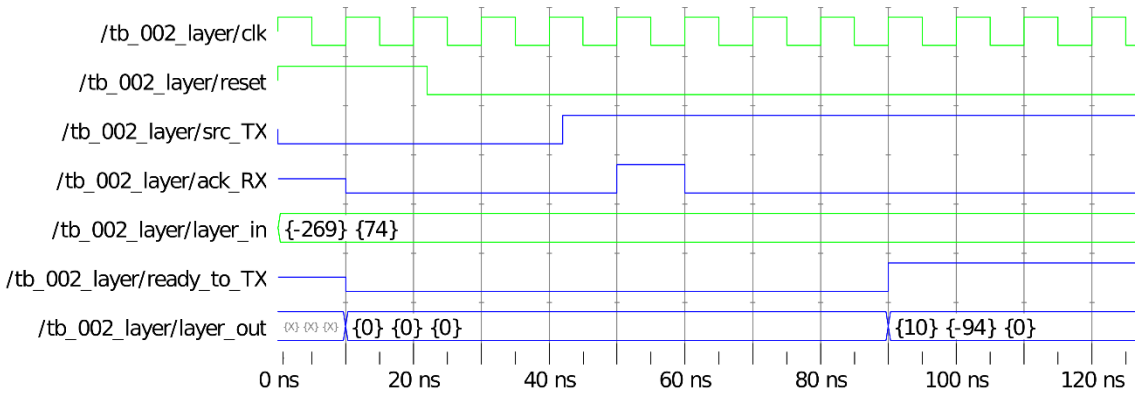


Fig. 3-3: Result of testing functionality of Layer-01 via ModelSim simulation.

Fig. 3-3 shows that after 4 clock cycles the result is $(10, -94)$, as expected since the output is a single width slice of the double width accumulated multiplication results.

3.7 Mediator Block

In order to test `Layer-01` or a chain thereof, a separate entity was created which handles the communication to and from the layer network.

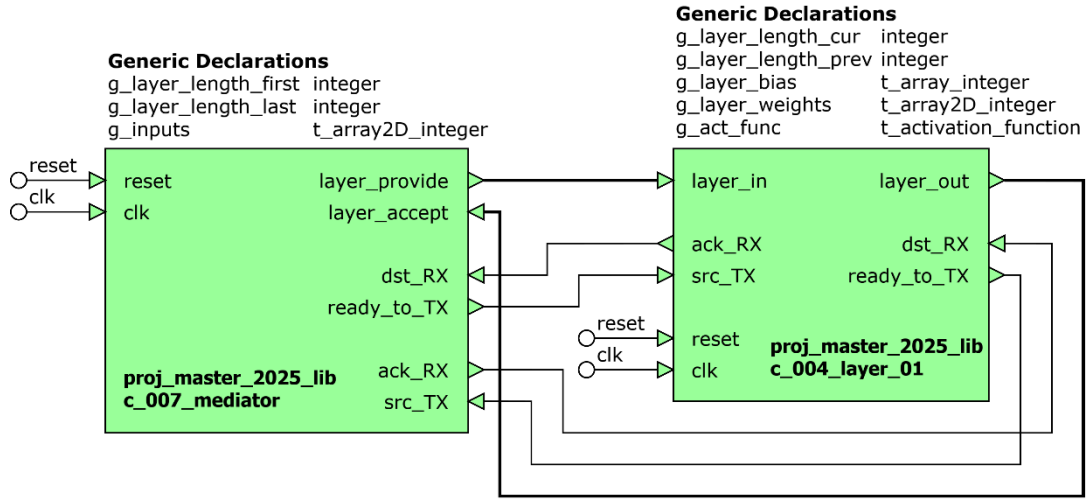


Fig. 3-4: Mediator block connected to single instance of Layer-01.

Since this `Mediator-Component` only serves testing purposes, its code won't be discussed in detail and can be found in the Appendix I.C. It will facilitate the testing of one `ANN-Component` instance in Chapter 4.3.

3.8 Template VHDL component

A “template component” (the ANN-Template) was created to allow for easy conversion from Python to VHDL.

This component is based on a single instantiation of Layer-01 and was reworked to contain several placeholder strings, which allow the conversion from the Python models to a VHDL component.

Code 3-15: ANN-Template placeholder – Name and date (Combined v2.vhd, Appendix I.D).

```
1  -- VHDL Entity {$NAME_ENTITY}
2
3  -- Created: {$DATE_TIME}
```

The most obvious placeholders are {$NAME_ENTITY} and {$DATE_TIME}, which stand for the components name and the time of the creation of the VHDL file. These repeat throughout the ANN-Template file.

Code 3-16: ANN-Template placeholder – Port declarations (Combined v2.vhd, Appendix I.D).

```
10 entity {$NAME_ENTITY} is
11    port(
12       clk          : in     std_logic;
13       reset        : in     std_logic;
15       src_TX      : in     std_logic;
16       ack_RX      : out    std_logic;
18       dst_RX      : in     std_logic;
19       ready_to_TX : out    std_logic;
23 {$PORT_LAYER_IN_OUT}
24    );
```

The placeholder {$PORT_LAYER_IN_OUT} stands for the input and output buses of the ANN-Template port. Depending on the number of input and output nodes of the neural network, the resulting component must feature buses of corresponding width.

Code 3-17: ANN-Template placeholder – Signal declarations and Instance port mappings (Combined v2.vhd, Appendix I.D).

```
43 architecture struct of {$NAME_ENTITY} is
47    -- Internal signal declarations
51 {$SIGNAL_DECLARATION}
   [...]
81 begin
84 {$INSTANCE_PORT_MAPPINGS}
86 end struct;
```

The placeholder {$SIGNAL_DECLARATION} stands for the internal signal necessary to interconnect the number of instanced Layer-01 components.

For a neural network of n layers, $n-1$ sets of interconnecting signals are required, while input and output layers are directly connected to the port signals of the `ANN-Template` (see Code 3-16).

The placeholder `{$INSTANCE_PORT_MAPPINGS}` stands for the instanced `Layer-01` components and their port to signal mappings.

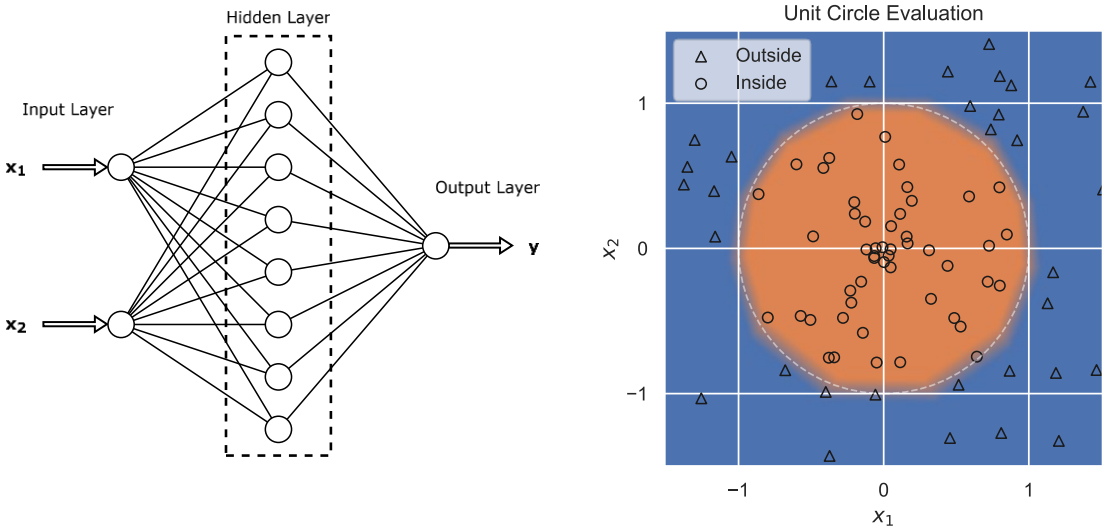
4 Implementation Python

There are several tools for creating artificial neural networks in Python. As indicated in Chapter 2.4, Keras and PyTorch were chosen as the APIs used for this task.

4.1 ANN creation and training

A test network was generated for both modules. Its task was to detect if a given 2D-point lies inside or outside the unit circle. This task was chosen for its relative simplicity, to easily verify the functionality of `Layer-01` and the generated neural network.

While training would be performed in floating-point representation, the created VHDL code would use weights and biases in fixed-point representation.



(a) **(b)**

Fig. 4-1: a) Diagram of the used neural network structure, b) results of the trained model – the Δ and O marker represent a few randomly chosen points in the 2D-plane which were evaluated to be either outside or inside the unit circle. Also, a grid representing the 2D-plane was evaluated to show the region and its boundaries where the neural network determines inside (orange) or outside (blue) the unit circle.

The network structure was chosen to consist of 2 input nodes (the x_1 and x_2 values of a given point in the 2D plane), 8 hidden layers, and 1 output layer (indicating if said point lies inside the unit circle).

4.1.1 Keras

As introduced in Chapter 2.5, the network was created as a “Sequential” model of Keras.

Code 4-1: Model creation with Keras (ANN_03_ann_keras.py, Appendix I.D).

```
29  model = Sequential()
30  L1 = 8
31
32  model.add(Dense( L1, activation = 'relu', input_shape = (2,) ))
34  model.add(Dense(1, activation = 'hard_sigmoid'))
36
37  model.compile(loss='MeanSquaredError', optimizer='adam',
    metrics=['accuracy'])
```

The activation functions of the hidden layer were chosen to be “ReLu” and the of the output layer to be “Hard sigmoid”. For training purposes, the loss function was chosen to be “MeanSquaredError” with the optimizer “Adam”.

Code 4-2: Console output of Keras model.

```
model.summary()
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 8)                 24
_________________________________________________________________
dense_9 (Dense)              (None, 1)                 9
=================================================================
Total params: 33
Trainable params: 33
Non-trainable params: 0
```

Code 4-3: Model training with Keras (ANN_03_ann_keras.py, Appendix I.D).

```
96  NUM_EPOCHS = 5
97
98  # model.fit(X_train, y_train, epochs=5, sample_weight=w_train)
99  model.fit_generator(generator=training_generator,
    validation_data=validation_generator,
100                     epochs=NUM_EPOCHS)
```

The model was trained for 5 Epochs with 10^5 training samples. After this, the model is stored to be used in the generation of the VHDL ANN code.

Code 4-4: Model storage (ANN_03_ann_keras.py, Appendix I.D).

```
109 fpath = os.path.join(path_out_dir, "circle_model.keras")
110 model.save(fpath, True, True)
```

4.1.2 PyTorch

As introduced in Chapter 2.6, the network was created as a “Sequential” model of PyTorch.

Code 4-5: Model creation with PyTorch (ANN_03c_ann_pytorch.py, Appendix I.F).

```
33  class NN(nn.Module): # inherit from nn.Module
34
35      def __init__(self, num_inputs, num_hidden, num_outputs):
36          super(NN, self).__init__()
45          self.linear_stack = nn.Sequential(
46              nn.Linear(num_inputs, num_hidden),
47              nn.ReLU(),
48              nn.Linear(num_hidden, num_outputs),
49              nn.Hardsigmoid(),
50          )
52
53      def forward(self, x):
54          # Perform the calculation of the model to determine the prediction
55          x = self.linear_stack(x)
56          return x
    [...]
67  #%% Hyperparameters
68  num_inputs = 2
69  num_hidden = 8
70  num_outputs = 1
71
72  batch_size = 64
73  num_epochs = 5
74
75  #%% Initialize Network
76
77  model = NN(num_inputs=num_inputs, num_hidden=num_hidden,
    num_outputs=num_outputs).to(device)
```

The activation functions of the hidden layer were chosen to be “ReLu” and the of the output layer to be “Hard sigmoid”.

Code 4-6: Console output of PyTorch model.

```
print(model)
NN(
  (linear_stack): Sequential(
    (0): Linear(in_features=2, out_features=8, bias=True)
    (1): ReLU()
    (2): Linear(in_features=8, out_features=1, bias=True)
    (3): Hardsigmoid()
  )
```

For training purposes, the loss function was chosen to be “MeanSquaredError” with the optimizer “Adam”. The model was trained for 5 Epochs with 10^6 training samples.

Code 4-7: Loss function and optimizer in PyTorch (ANN_03c_ann_pytorch.py, Appendix I.F).

```
119 #%% Loss and Optimizer
120 loss_module = nn.MSELoss()
121 optimizer = torch.optim.Adam(model.parameters())
    [...]
152 train_model(model, optimizer, train_data_loader, loss_module,
    num_epochs=num_epochs)
```

After this, the model is converted to a TorchScript module, which can be loaded without also storing and loading the `nn.module` class it is based on (see Code 4-5).

Code 4-8: Model Storage. (ANN_03c_ann_pytorch.py, Appendix I.F).

```
164 model_scripted = torch.jit.script(model) # Export to TorchScript
165 model_scripted.save('03c_ann_pytorch_model.pt') # Save
```

The TorchScript model is then stored, for use in the generation of the VHDL ANN code.

4.2 Conversion from Python to VHDL

Uses of both modules (Keras and PyTorch) yield a trained model each. These models must be converted to a VHDL component for further use in our FPGA environment.

One early issue with both modules is that the structure of the neural network must be reconstructed from the trained model. One assumption is, that the models were created as a “Sequential” model for their respective modules. Otherwise, this recreation will yield an empty model or fail.

4.2.1 Keras

Keras allows for storing and loading of the complete model. The file size is about 25KB for the model from Chapter 4.1.1.

Code 4-9: Loading the model (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
37 model = keras.models.load_model(fpath_input)
39 weights = model.get_weights()
43 ann_conf = model.get_config()
```

Reconstruction of the neural network

The `get_config()` function is used to reconstruct the structure of the neural network.

Code 4-10: Reconstructing the neural network (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
43 ann_conf = model.get_config()
44
45 n = ann_conf["name"]
47 assert n[:10] == "sequential"
48
49 ann_layers = ann_conf["layers"]
52 layers = {"num_layers":0, "LAYER":{}}
53 units_prev = 0
54 layer_idx = 0
55
56 for i,entry in enumerate(ann_layers):
57     # print("-- Layer", i, entry["class_name"])
58
59     if entry["class_name"] == "InputLayer":
60         layer_conf = entry["config"]
```

```
61          layers["input_size"] = layer_conf["batch_input_shape"][1]
62          units_prev = layer_conf["batch_input_shape"][1]
63
64      elif entry["class_name"] == "Dense":
65          layer_conf = entry["config"]
66          layers["LAYER"][layer_idx] = {"units": layer_conf["units"],
67                                        "units_prev": units_prev,
68                                        "activation":
    layer_conf["activation"]}
69          layers["num_layers"] += 1
70          units_prev = layer_conf["units"]
71          layer_idx += 1
74      else:
75          raise Exception("Unknown Type of Layer")
78      pass
```

First, the name of the model is assumed to begin with "sequential". This is due to the naming convention of the "Sequential" model (see Chapter 2.5).

Then all layers are evaluated. Only "InputLayer" and "Dense" (fully connected) layers are allowed. Should other layer types occur ("MaxPooling", "DropOut", etc), then the neural network cannot be reconstructed (for this version of the code).

If successful, a custom summary is printed to the console.

Code 4-11: Console output for reconstructed Keras model.

```
   ------------------------------
   -- Summary of Network:
   ------------------------------
Number Of Layers: 2

-- Layer 0
   Units: 8, Units_Prev: 2, Activation: relu
-- Layer 1
   Units: 1, Units_Prev: 8, Activation: hard_sigmoid
```

Conversion to fixed-point representation

The next step is the conversion of weights and biases from floating-point representation to fixed-point representation.

Code 4-12: Preparation for conversion to fixed-point representation (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
39  weights = model.get_weights()
    [...]
101 #%% CONVERSION
103 DATA_WIDTH = 16
104 DATA_QF = 8
105 FP_ONE = 2**DATA_QF
```

As introduced in Chapter 3.1, the chosen fixed-point representation for this thesis is `Q7.8`.

A large for-loop is started, which iterates over all layers of the neural network, in order to prepare the overall conversion from the Python model to a VHDL component.

Code 4-13: Conversion to fixed-point representation (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
114 for i in range(layers["num_layers"]):
115 
116     layer = layers["LAYER"][i]
117 
118     w = weights[2*i] * FP_ONE
119     w = np.int64(w)
120     w = np.transpose(w)
122     b = weights[2*i + 1] * FP_ONE
123     b = np.int64(b)
```

The weights and biases of the current layer are converted to fixed-point representation.

Preparation of replacement strings for the ANN-Template

Code 4-14: Preparation of replacement strings for the ANN-Template (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
126     txt_component =  f"U_{i} : c_004_layer_01" + "\n{};\n"
129     txt_generic = "generic map (\n{}\n)"
130     txt_port = "port map (\n{}\n)"
```

Each layer of the neural network is represented with an instance of a `Layer-01` component, which has its on generic and port map.

Component generic map

Code 4-15: Component generic map – Number of nodes (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
138     list_generics = []
140     list_generics.append("   "+f"g_layer_length_cur => {layer['units']}")
142     list_generics.append("   "+f"g_layer_length_prev =>
{layer['units_prev']}")
```

The generic map of each layer requires information about the number of nodes in the current layer as well as in the previous layer.

Code 4-16: Component generic map – Biases (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
147     txt_b1 = "( {} )"
148     if len(b) == 1:
149         txt_b1 = "( 0 => {} )"
150 
151     b1 = [str(x) for x in b]
152     b1 = txt_b1.format(", ".join(b1))
153     txt_bias = "   "+f"g_layer_bias => {b1}"
154     list_generics.append(txt_bias)
```

The biases are always a 1D-array of integers, so they concatenated for VHDL. Unless the bias consists of only one element. Then the appropriate formatting for 1-element arrays in VHDL must be used.

Code 4-17: Component generic map – Weights (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
161     txt_w1 = "( {} )"
162     if len(w) == 1:
163         txt_w1 = "( 0 => {} )"
164         pass
165 
166     list_w2 = []
167     for elem in w:
168         txt_w2 = "({})"
169         if len(elem) == 1:
170             txt_w2 = "( 0 => {} )"
171             pass
172 
173         w2 = [str(x) for x in elem]
174         w2 = txt_w2.format(", ".join(w2))
175         list_w2.append(w2)
176         pass
177     w1 = txt_w1.format(", ".join(list_w2))
178 
179     txt_weights = "  "+f"g_layer_weights => {w1}"
180     list_generics.append(txt_weights)
```

Similar to biases, the weights are concatenated as a 2D-array for VHDL. Just as with biases, the appropriate formatting for 1-element entries in the 2D-array must be adhered to.

Code 4-18: Component generic map – Activation function and composition (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
184     list_generics.append("  "+f"g_act_func =>
    AF_{layer['activation'].upper()}")
189     txt_generic = txt_generic.format(",\n".join(list_generics))
```

Finally, the activation function is added to the generic map.

Code 4-19: Component generic map – Console output.

```
generic map (
  g_layer_length_cur => 1,
  g_layer_length_prev => 8,
  g_layer_bias => ( 0 => 1186 ),
  g_layer_weights => ( 0 => (91, -967, -784, -839, -802, -1051, 22, -975) ),
  g_act_func => AF_HARD_SIGMOID
)
```

Signal mapping

To know which signals to connect for the port mapping of the current instance of a `Layer-01` component, the internal signals must first be determined.

Code 4-20: Signal determination for the current layer (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
251         signal_ACK_RX    = f"DST_RX_{i-1}"
252         signal_SRC_TX    = f"R2TX_{i-1}"
253         signal_Layer_in  = f"layer_{i-1}"
254 
```

```
255         signal_DST_RX    = f"DST_RX_{i}"
256         signal_R2TX      = f"R2TX_{i}"
257         signal_Layer_out = f"layer_{i}"
258
259         signal_decl_DST_RX = f"signal {signal_DST_RX} : std_logic;"
260         signal_decl_R2TX   = f"signal {signal_R2TX} : std_logic;"
261         signal_decl_Layer  = f"signal {signal_Layer_out} :
    t_array_data_stdlv(0 to {layer['units'] - 1});"
262         list_signals.append(signal_decl_DST_RX)
263         list_signals.append(signal_decl_R2TX)
264         list_signals.append(signal_decl_Layer)
```

The signals to the "source"/left side of the current layer are indexed at `i-1`, while signals to the "destination"/right side of the current layer are indexed at `i`. For each new right-side signal, a separate signal declaration is required before the "begin" statement of the VHDL architecture.

In the case of the first layer, there are no new left-side signals required, and in the case of the last layer, there are no new right-side signals required, as these mapped to the `ANN-Template` ports.

Component port map

Code 4-21: Component port map (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
270     list_ports = []
271
272     list_ports.append("  "+f"clk => clk")
273     list_ports.append("  "+f"reset => reset")
274     #-------
275     list_ports.append("  "+f"ack_RX => {signal_ACK_RX}")
276     list_ports.append("  "+f"src_TX => {signal_SRC_TX}")
277     list_ports.append("  "+f"layer_in => {signal_Layer_in}")
278     #-------
279     list_ports.append("  "+f"dst_RX => {signal_DST_RX}")
280     list_ports.append("  "+f"ready_to_TX => {signal_R2TX}")
281     list_ports.append("  "+f"layer_out => {signal_Layer_out}")
282
284     txt_port = txt_port.format(",\n".join(list_ports))
```

With left- and right-side signals determined, the port map of the current instance of a `Layer-01` component can be filled and composed.

Code 4-22: Component port map – Console output.

```
port map (
  clk => clk,
  reset => reset,
  ack_RX => DST_RX_0,
  src_TX => R2TX_0,
  layer_in => layer_0,
  dst_RX => dst_RX,
  ready_to_TX => ready_to_TX,
  layer_out => layer_out
);
```

The complete console output of the summary of the conversion to VHDL can be seen in Appendix I.H.

Execution of replacement for the ANN-Template

With all replacement strings generated, the placeholder strings in the ANN-Template (see Chapter 0) can be replaced to create the complete component of this neural network.

Code 4-23: Reconstructing the neural network (ANN_04_ann_to_vhdl.py, Appendix I.G).

```
366 with open(fPathIn) as f:
367     fileStr = f.read()
368     pass
369 
370 EntityName = "c_x_ANN_01"
371 fileNameOut = f"{EntityName}.vhd"
372 fPathOut = os.path.join(path_out_dir, fileNameOut)
373 
374 fileStr = fileStr.replace("{$NAME_ENTITY}", EntityName)
376 
377 tnow = datetime.datetime.now()
378 fileStr = fileStr.replace("{$DATE_TIME}", str(tnow))
380 fileStr = fileStr.replace("{$SIGNAL_DECLARATION}", txt_signals)
382 fileStr = fileStr.replace("{$INSTANCE_PORT_MAPPINGS}", inst_port_maps)
384 fileStr = fileStr.replace("{$PORT_LAYER_IN_OUT}", txt_port_layer_inout)
385 
386 with open(fPathOut, mode="w") as f:
387     f.write(fileStr)
388 
389 print(f"File '{fileNameOut}' created/overwritten!")
```

The created output file of this ANN-Component can then be used in the VHDL project.

4.2.2 PyTorch

PyTorch allows for storing and loading of the TorchScript model. The file size is about 9KB for the model from Chapter 4.1.2.

Code 4-24: Loading the model (ANN_04b_ann_to_vhdl_pytorch.py, Appendix I.I).

```
38 model = torch.jit.load('03c_ann_pytorch_model.pt')
39 model.eval()
```

Reconstruction of the neural network

The hierarchical structure of the model and its children is used to reconstruct the structure of the neural network.

Code 4-25: Reconstructing the neural network (ANN_04b_ann_to_vhdl_pytorch.py, Appendix I.I).

```
46 # fetch children of the model
47 chil_model = [ch for ch in model.children()]
48 
51 assert len(chil_model)==1
52 mod = chil_model[0]
53 
56 assert hasattr(mod, 'original_name')
```

```
 57  assert mod.original_name == "Sequential"
 58
 59  chil_seq = [ch for ch in mod.children()]
 60  ann_layers = chil_seq
 61
 63  list_layers = ["Linear",]
 69  dict_map_activation = {
 70      "ReLU": "RELU",
 71      "Hardsigmoid": "HARD_SIGMOID",
 72  }
 75  layers = {"num_layers":0, "LAYER":{}}
 76
 77  for i,entry in enumerate(ann_layers):
 78      n = entry.original_name
 81      if n in list_layers:
 83          if n == "Linear":
 85              layers["LAYER"][layers["num_layers"]] = {
 86                  "activation": "IDENTITY",
 87                  "pytorchIdx": i,
 88              }
 91              layers["num_layers"] += 1
 94      elif n in dict_map_activation:
 97          if n == "ReLU":
 98              layers["LAYER"][layers["num_layers"]-1]["activation"] =
     dict_map_activation[n]
100          elif n == "Hardsigmoid":
101              layers["LAYER"][layers["num_layers"]-1]["activation"] =
     dict_map_activation[n]
103      else:
104          raise Exception("Unknown Type of Layer")
107      pass
```

The reconstruction of the network with PyTorch is more complicated and less robust than with Keras.

First, the children of the model are evaluated. If there is only one child, then we check if its name is “Sequential”. This allows us to assume that this must be a “Sequential” model (see Chapter 2.6) and its children must be the stack of used sequential layers.

Then all layers are evaluated. Only “Linear” (fully connected) or activation layers are allowed. Should other layer types occur (“Convolution”, unknown activation functions, etc), then the neural network cannot be reconstructed (for this version of the code).

Code 4-26: Reconstructing the neural network (ANN_04b_ann_to_vhdl_pytorch.py, Appendix I.I).

```
109  #%% Fetch Weights and Biases
110
111  state_dict = model.state_dict()
112  postfix_weight = ".{}.weight"
113  postfix_bias   = ".{}.bias"
114
116  list_keys = [k for k in state_dict]
118  prefix = list_keys[0].split(".")[0]
119
```

```
120 for idx in range(layers["num_layers"]):
121     entry = layers["LAYER"][idx]
125     pytorchIdx = entry["pytorchIdx"]
126     key_weights = prefix + postfix_weight.format(pytorchIdx)
127     key_bias    = prefix + postfix_bias.format(pytorchIdx)
128 
129     entry["weights"] = state_dict[key_weights].numpy()
130     entry["bias"]    = state_dict[key_bias].numpy()
133     entry["units"], entry["units_prev"] = entry["weights"].shape
```

Since fully connected layers and activation functions are separate and equally likely occurrences in PyTorch, some additional reconstruction must be performed to reconstruct the structure required for the template component, which requires a layer to have weights and biases as well as an activation function. The weights and biases can be accessed from the “state dict” of the model.

If successful, a custom summary is printed to the console.

Code 4-27: Console output for reconstructed PyTorch model.

```
   ------------------------------
   -- Summary of Network:
   ------------------------------
Number Of Layers: 2

-- Layer 0
   Units: 8, Units_Prev: 2, Activation: RELU
-- Layer 1
   Units: 1, Units_Prev: 8, Activation: HARD_SIGMOID
```

Conversion to fixed-point representation

Code 4-28: Conversion to fixed-point representation (ANN_04b_ann_to_vhdl_pytorch.py, Appendix I.I).

```
159 DATA_WIDTH = 16
160 DATA_QF = 8
161 FP_ONE = 2**DATA_QF
    [...]
170 for i in range(layers["num_layers"]):
172     layer = layers["LAYER"][i]
173 
174     weights = layer["weights"]
175     w = weights * FP_ONE
176     w = np.int64(w)
179     bias = layer["bias"]
180     b = bias * FP_ONE
181     b = np.int64(b)
```

The conversion to fixed-point representation is similar to the code of the Keras module.

Creation of the neural network component

From this point forward, the creation of the neural network component is identical to the code used with the Keras module (see Chapter 4.2.1).

4.3 Testing

The ANN-Component was created with the trained neural network from Chapter 4.1. It's important to note that for the following observations, the network consists of 2 nodes in the input layer, 8 nodes in the hidden layer, and 1 node in the output layer.

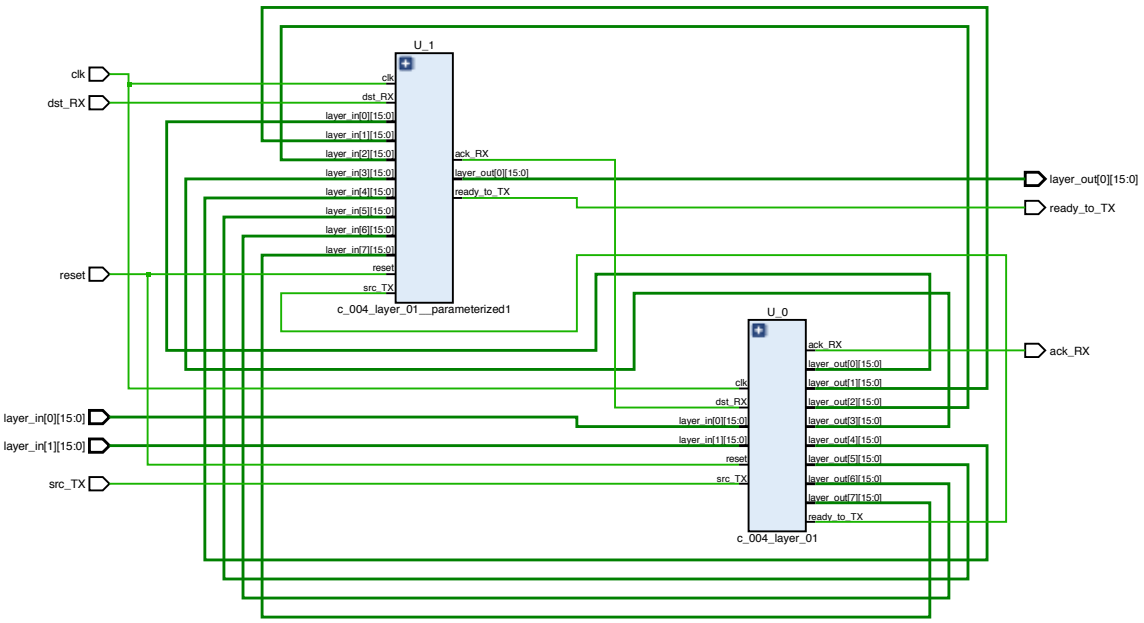


Fig. 4-2: Schematic of Elaborated Design in Vivado HLx 2019.1.

The Synthesis in Vivado of the ANN-Component alone shows that 10 DSP Slices were utilized for this design. This agrees with the minimum requirement of at least 9, as every node requires one DSP slice.

Depending on the FPGA device used, the size of the neural network may be restricted to the amount of DSP slices available.

Table 4-1: Post-Synthesis Resource Utilization for a Z-7010 device.

Resource	Estimation	Available	Utilization
LUT	561	17600	3.19 %
FF	501	35200	1.42 %
DSP	10	80	12.50 %
IO	54	100	54.00 %
BUFG	1	32	3.13 %

The Mediator-Component was used to test the function of the ANN-Component (using the network to “infer” the output). The following observations were made.

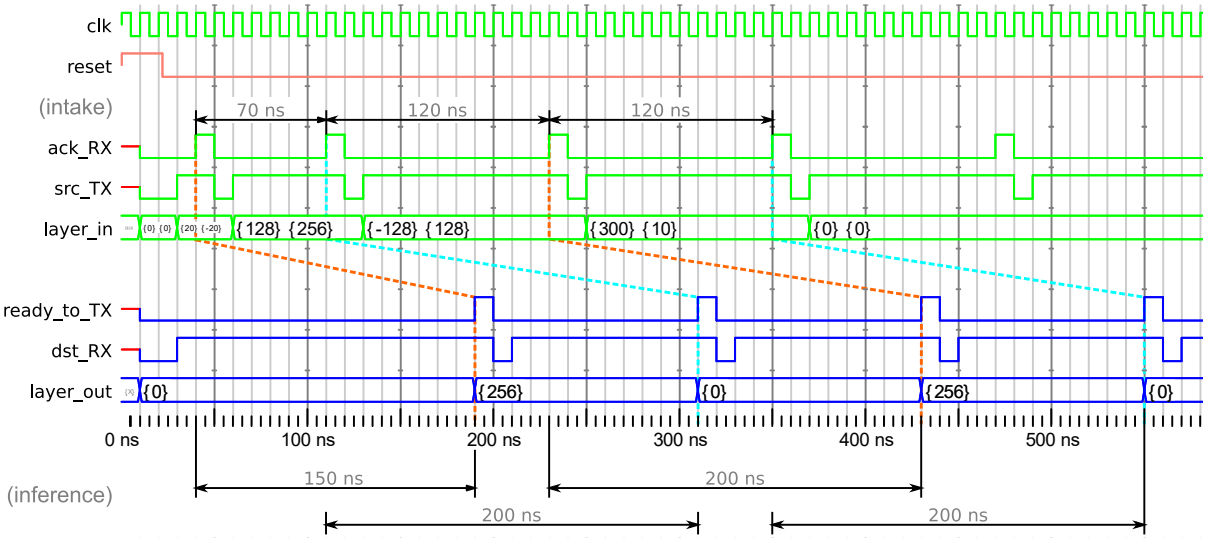


Fig. 4-3: Testing the ANN-Component with the Mediator-Component – The first inference of the ANN-Component is 50ns faster than the following inferences.

The “inference time” is the time that the ANN-Component takes from acknowledging the input value to transmitting the corresponding output value. The inference of the neural network takes 200ns. However, the first inference is 50ns faster than the following times.

It takes the ANN-Component a certain amount of time from the acknowledgement of one input value to the acknowledgement of the next input value. This time will be referred to as the “intake time”. The overall intake takes 120ns, while the first intake only takes 70ns.

Both cases can be explained by the fact that not all layers in the network have the same number of nodes. The first input can be processed faster, since in subsequent calculations all layers can only be traversed as quickly as the slowest layer.

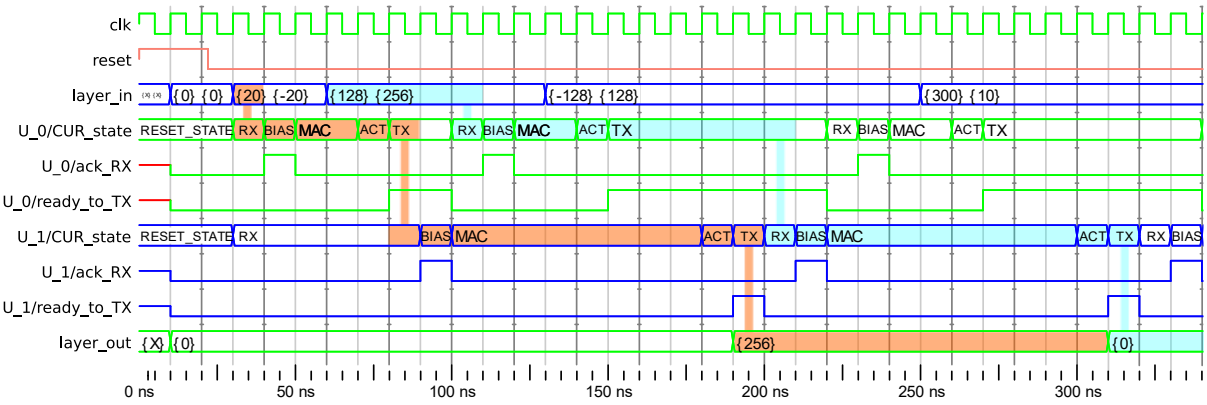


Fig. 4-4: Testing the ANN-Component with the Mediator-Component – Detailed look at the inner workings of the component shows the actions of the individual layers.

Fig. 4-4 depicts a closer look at the actions of the inner `Layer-01` components (U_0 and U_1). It shows that the first layer requires only 2 clock cycles to perform the actions in the "Multiply-Accumulate" state (MAC). However, in subsequent executions, the component U_0 must wait for the MAC state of component U_1 to complete, in order to successfully transmit its output to U_1.

This shows that the slowest layer in the network is the one with the most connections to nodes of the previous layer. It is also apparent that the time taken by the slowest layer will propagate to make all layers slow down accordingly.

Therefore, the speed of inference is inversely proportional to the largest layer size (N_{max}) and the number of layers (L), assuming a continuous stream of input values.

$$t_{\text{infer}} \propto N_{max} \cdot L \cdot t_{CLK} \tag{4-1}$$

$$f_{\text{infer}} \propto \frac{f_{CLK}}{N_{max} \cdot L} \tag{4-2}$$

Overall intake, however, is determined by only the largest layer size, as the layers are executed in parallel.

$$f_{\text{intake}} \propto \frac{f_{CLK}}{N_{max}} \tag{4-3}$$

5 Conclusion

It is possible to create a simple neural network in Python and deploy it in VHDL.

In this work we showed how to create a layer of a neural network in VHDL. We showed how to read the weights and biases as well as the structure of a neural network model and implement these into a finished component file of the complete neural network for both the "Keras" and PyTorch" modules.

Due to the component modularity of VHDL, it can be added to any design for an FPGA, as long as the number of required DSP slices is not exceeded. Otherwise, a larger amount of resource sharing may be required (see Chapter 1.3).

As the requirements for using this product are only one Python script and one VHDL template file, it is a very simple solution.

5.1 Outlook

The overall speed of `Layer-01` was not a primary concern for this thesis, so there is room for improving the overall execution speed of the state machine (see Chapter 3.5).

This work has implemented only a few activation functions, which take up a large part of the state machine (see Chapter 3.5.5). The execution of these could be moved to a separate component all together and their variety could be improved to account for more of the common activation functions used in applications of neural networks on FPGAs.

It may be interesting to explore HLS and the "hls4ml" Python package in future curriculums of the Master's degree (see Chapter 1.2), as they may provide new and interesting ways to develop neural networks for FPGAs for just algorithm deployment in general.

If Xilinx/AMD's "Brevitas" is continued to be updated and its documentation expanded (see Chapter 1.2), then it may be interesting to investigate Brevitas and FINN, if optimized quantized neural networks are of interest.

Bibliography

[1] C. C. Aggarwal, Neural Networks and Deep Learning, Springer, 2018.

[2] "UltraScale Architecture DSP Slice User Guide (UG579)," [Online]. Available: https://docs.amd.com/v/u/en-US/ug579-ultrascale-dsp. [Accessed 1 May 2025].

[3] M. Shahdad, "An overview of VHDL language and technology," in *23rd ACM/IEEE Design Automation Conference*, 1986.

[4] A. A. Sagahyroon, "From AHPL to VHDL: A Course in Hardware," *454 IEEE TRANSACTIONS ON EDUCATION,* vol. 4, no. 43, pp. 449-454, 4 November 2000.

[5] "Python Homepage," [Online]. Available: https://www.python.org/about/. [Accessed 1 May 2025].

[6] D. Kuhlman, "A Python Book: Beginning Python, Advanced Python, and Python Exercises," 12 April 2012. [Online]. Available: https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html. [Accessed 1 May 2025].

[7] "Keras Homepage," [Online]. Available: https://keras.io/. [Accessed 1 May 2025].

[8] "Keras on Github," [Online]. Available: https://github.com/keras-team/keras. [Accessed 1 May 2025].

[9] "PyTorch Homepage," [Online]. Available: https://pytorch.org/. [Accessed 1 May 2025].

[10] "PyTorch on Github," [Online]. Available: https://github.com/pytorch/pytorch. [Accessed 1 May 2025].

[11] Texas Instruments, "TMS320C64x DSP Library - Programmer's Reference," October 2003. [Online]. Available: https://www.ti.com/lit/ug/spru565b/spru565b.pdf. [Accessed 1 May 2025].

[12] E. L. Oberstar, "Fixed-Point Representation & Fractional Math," 30 August 2007. [Online]. Available:

https://web.archive.org/web/20171104111827/http://www.superkits.net/whitepapers/Fixed Point Representation & Fractional Math.pdf. [Accessed 1 May 2025].

[13] D. L. Perry, VHDL: Programming by Example (4th ed.), McGraw-Hill, 2002.

[14] "TorchScript documentation," [Online]. Available: https://docs.pytorch.org/docs/stable/jit.html. [Accessed 1 May 2025].

List of Figures

List of Tables

List of Code

I. Appendix

The Appendix features code segments relevant to the thesis. For further development, the code is also available at:

- https://github.com/polsterc16/MA25 (VHDL)
- https://github.com/polsterc16/MA25_Python (Python)

A. VHDL code: Package

The package created for this thesis to share types and constants between files.

p_002_generic_01_pkg.vhd

```
1  --
2  -- VHDL Package Header proj_master_2025_lib.p_002_generic_01
3  --
4  -- Created:
5  --          by - Admin.UNKNOWN (LAPTOP-7KFJT032)
6  --          at - 08:58:51 14.03.2025
7  --
8  -- using Mentor Graphics HDL Designer(TM) 2017.1a (Build 5)
9  --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13
14 PACKAGE p_002_generic_01 IS
15
16   constant c_DATA_WIDTH : integer := 16;  -- {$c_DATA_WIDTH}
17   constant c_DATA_QF    : integer := 8;   -- {$c_DATA_QF}
18
19   constant c_FP_INT_MAX : integer := 2**(c_DATA_WIDTH-1) - 1;
20   constant c_FP_INT_MIN : integer := -2**(c_DATA_WIDTH-1);
21   subtype t_fp_int is integer range c_FP_INT_MIN to c_FP_INT_MAX;
22
23   type t_array_integer is array(natural range <>) of t_fp_int;
24   type t_array2D_integer is array(natural range <>, natural range <>) of t_fp_int;
25
26   type t_array_data_stdlv is array(natural range <>) of
     std_logic_vector(c_DATA_WIDTH-1 downto 0);
27   type t_array_data_stdlv_dw is array(natural range <>) of
     std_logic_vector(2*c_DATA_WIDTH-1 downto 0);
28
29   type t_array_data_signed is array(natural range <>) of signed(c_DATA_WIDTH-1
     downto 0);
30   type t_array_data_signed_dw is array(natural range <>) of signed(2*c_DATA_WIDTH-1
     downto 0);
31
32   type t_stm_layer is (
33     RESET_STATE, -- default state to fall back on
```

```
34      IDLE_TX,  -- we have finished calculation and wait for our output to be
    received
35      IDLE_RX,  -- we are awaiting a valid input
36      BIAS_SETUP,  -- we are awaiting a valid input
37      MAC,     -- we are calculating the nodes
38      ACT_FUNC  -- performing activation function
39    );
40
41    type t_activation_function is (
42      AF_IDENTITY,
43      AF_SIGN,
44      AF_RELU,
45      AF_HARD_SIGMOID
46    );
47
48  END p_002_generic_01;
```

B. VHDL code: Layer

Code of Layer-01.

c_004_layer_01_rtl.vhd

```
1  --
2  -- VHDL Architecture proj_master_2025_lib.c_004_layer_01.rtl
3  --
4  -- Created:
5  --          by - Admin.UNKNOWN (LAPTOP-7KFJT032)
6  --          at - 08:52:41 14.03.2025
7  --
8  -- using Mentor Graphics HDL Designer(TM) 2017.1a (Build 5)
9  --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13
14 library proj_master_2025_lib;
15 use proj_master_2025_lib.p_002_generic_01.all;
16
17 entity c_004_layer_01 is
18    generic(
19       g_layer_length_cur  : integer               := 4;
20       g_layer_length_prev : integer               := 2;
21       g_layer_bias        : t_array_integer       := (0,0,0,0);
22       g_layer_weights     : t_array2D_integer     := ((0,0),(0,0),(0,0),(0,0));
23       g_act_func          : t_activation_function := AF_RELU
24    );
25    port(
26       clk         : in     std_logic;
27       reset       : in     std_logic;
28       --enable      : in     std_logic;
29       dst_RX      : in     std_logic;
30       src_TX      : in     std_logic;
31       ready_to_TX : out    std_logic;
32       ack_RX      : out    std_logic;
33       layer_in    : in     t_array_data_stdlv (0 to g_layer_length_prev-1);
34       layer_out   : out    t_array_data_stdlv (0 to g_layer_length_cur-1)
35    );
36
37 -- Declarations
38
39 end c_004_layer_01 ;
40
41 --
42 architecture rtl of c_004_layer_01 is
43   SIGNAL NEX_state, CUR_state : t_stm_layer;
44   SIGNAL NEX_node_prev : integer range -1 to g_layer_length_prev-1 := 0;
45   SIGNAL CUR_node_prev : integer range -1 to g_layer_length_prev-1 := 0;
46
47   SIGNAL NEX_ack_RX : std_logic := '0';
48   SIGNAL NEX_ready_to_TX : std_logic := '0';
49   SIGNAL NEX_layer_out   : t_array_data_stdlv(0 to g_layer_length_cur-1) :=
(others=>(others=>'0'));
50
51   SIGNAL CUR_data_in    : t_array_data_signed(0 to g_layer_length_prev-1);
52   SIGNAL NEX_data_in    : t_array_data_signed(0 to g_layer_length_prev-1);
53   SIGNAL CUR_data_accum  : t_array_data_signed_dw(0 to g_layer_length_cur-1);
```

```
54    SIGNAL NEX_data_accum  : t_array_data_signed_dw(0 to g_layer_length_cur-1);
55
56  --  CONSTANT c_ACT_FUNC   : t_activation_function := g_act_func;
57    CONSTANT c_pos_one : STD_LOGIC_VECTOR := STD_LOGIC_VECTOR( SHIFT_LEFT(TO_SIGNED(
      1, c_DATA_WIDTH), c_DATA_QF) );
58    CONSTANT c_neg_one : STD_LOGIC_VECTOR := STD_LOGIC_VECTOR( SHIFT_LEFT(TO_SIGNED(
      -1, c_DATA_WIDTH), c_DATA_QF) );
59
60    CONSTANT c_s_dw_pos_3     : SIGNED :=  SHIFT_LEFT(TO_SIGNED(  3, 2*c_DATA_WIDTH),
      2*c_DATA_QF);
61    CONSTANT c_s_dw_neg_3     : SIGNED :=  SHIFT_LEFT(TO_SIGNED( -3, 2*c_DATA_WIDTH),
      2*c_DATA_QF);
62    CONSTANT c_s_pos_half  : SIGNED :=  TO_SIGNED( (2**(c_DATA_QF))/2, c_DATA_WIDTH
      );
63    CONSTANT c_s_pos_sixth : SIGNED :=  TO_SIGNED( (2**(c_DATA_QF))/6, c_DATA_WIDTH
      );
64
65  begin
66    P_STM : process(CUR_state, CUR_node_prev, src_TX, dst_RX)
67      VARIABLE v_dw_AF_temp1 : SIGNED (c_DATA_WIDTH-1 downto 0);
68      VARIABLE v_dw_AF_temp2 : SIGNED (2*c_DATA_WIDTH-1 downto 0);
69
70    begin
71      -- -- default assignments
72      -- internal signals
73      NEX_state     <= CUR_state;
74      NEX_node_prev <= CUR_node_prev;
75
76      NEX_data_in   <= CUR_data_in;
77      NEX_data_accum <= CUR_data_accum;
78
79      -- signals to outputs
80      NEX_ack_RX <= ack_RX;
81      NEX_ready_to_TX <= ready_to_TX;
82      NEX_layer_out <= layer_out;
83
84      case(CUR_state) is
85        -- Default reset state
86        when RESET_STATE =>
87          -- default: goto Receive state
88          NEX_state <= IDLE_RX;
89
90          NEX_node_prev <= 0;
91          NEX_data_in   <= (others=>(others=>'0'));
92          NEX_data_accum <= (others=>(others=>'0'));
93          NEX_ready_to_TX <= '0';
94          NEX_ack_RX <= '0';
95          NEX_layer_out <= (others=>(others=>'0'));
96
97        -- we send our result, until it is accepted
98        when IDLE_TX =>
99
100         if dst_RX = '1' then
101           -- goto reveicer mode
102           NEX_ready_to_TX <= '0';
103           NEX_state <= IDLE_RX;
104         end if;
105
106       -- we wait until we receive new data
107       when IDLE_RX =>
```

```
108         if src_TX = '1' then
109           NEX_ack_RX <= '1'; -- set to 1 during transition
110 
111           --NEX_data_in <= layer_in; -- store input
112           for idx in layer_in'RANGE loop
113             NEX_data_in(idx) <= SIGNED(layer_in(idx));
114           end loop;
115 
116           NEX_state <= BIAS_SETUP;
117         end if;
118 
119       -- we fill our accummulators with bias value
120       when BIAS_SETUP =>
121         NEX_ack_RX <= '0'; -- reset
122 
123         -- initialize bias in layer nodes
124         LOOP_BIAS : FOR idx_node_cur in 0 to (g_layer_length_cur-1) LOOP
125           -- we create first entry:
126           NEX_data_accum(idx_node_cur) <=  SHIFT_LEFT(TO_SIGNED(
    g_layer_bias(idx_node_cur), 2*c_DATA_WIDTH), c_DATA_QF);
127         end LOOP;
128 
129         NEX_state <= MAC;
130 
131         -- nodes of PREVIOUS LAYER, which are decremented in MAC
132         NEX_node_prev <= g_layer_length_prev-1;
133 
134       -- we accumulate the node values times their weights
135       when MAC =>
136         -- loop through nodes of THIS LAYER
137         LOOP_Node : FOR idx_node_cur in 0 to (g_layer_length_cur-1) LOOP
138           -- Data (THIS LAYER node) = Data (THIS LAYER node) + Weight (of PREV
    LAYER node, relative to THIS LAYER node) * Data (PREV LAYER node)
139           -- NEX_data_accum(node_this) <= CUR_data_accum(node_this) +
    c_A_WEIGHTS(g_layer_index, node_this, CUR_node_prev) * CUR_data_in(CUR_node_prev) ;
140           NEX_data_accum(idx_node_cur) <= CUR_data_accum(idx_node_cur) +
    TO_SIGNED(g_layer_weights( idx_node_cur, CUR_node_prev), c_DATA_WIDTH) *
    CUR_data_in(CUR_node_prev) ;
141         end LOOP;
142 
143         -- decrement node of PREVIOUS LAYER
144         NEX_node_prev <= CUR_node_prev - 1;
145         -- exit if we have reached Zero
146         if CUR_node_prev = 0 then
147           NEX_state     <= ACT_FUNC;
148           NEX_node_prev <= 0;
149         end if;
150 
151       when ACT_FUNC =>
152         -- reminder: "CUR_data_accum" has 2x width of "layer_out" !
153         case g_act_func is
154           when AF_SIGN =>
155             -- When: Sign Function
156             LOOP_AF_SIGN : FOR idx_node_this in 0 to (g_layer_length_cur-1) LOOP
157               -- check if the whole slice is Zero
158               if CUR_data_accum(idx_node_this)(CUR_data_accum(idx_node_this)'RANGE)
    = (CUR_data_accum(idx_node_this)'range => '0') then
159                 -- is zero
160                 NEX_layer_out(idx_node_this) <= (others => '0');
```

```
161                 elsif CUR_data_accum(idx_node_this)(CUR_data_accum(idx_node_this)'HIGH) = '1' then
162                   -- is negative
163                   --NEX_layer_out(idx_node_this) <= STD_LOGIC_VECTOR( SHIFT_LEFT(TO_SIGNED( -1, 2*c_DATA_WIDTH), c_DATA_QF) );
164                   NEX_layer_out(idx_node_this) <= c_neg_one;
165                 else
166                   -- ELSE: is positive
167                   NEX_layer_out(idx_node_this) <= c_pos_one;
168                 end if;
169               end loop;
170
171             when AF_RELU =>
172               -- When: ReLu Function
173               LOOP_AF_RELU : FOR idx_node_this in 0 to (g_layer_length_cur-1) LOOP
174                 if CUR_data_accum(idx_node_this)(CUR_data_accum(idx_node_this)'HIGH) = '1' then
175                   -- is negative
176                   NEX_layer_out(idx_node_this) <= (others => '0');
177                 else
178                   -- ELSE: is positive
179                   NEX_layer_out(idx_node_this) <= STD_LOGIC_VECTOR( CUR_data_accum(idx_node_this)(c_DATA_WIDTH + c_DATA_QF - 1 downto c_DATA_QF) );
180                 end if;
181               end loop;
182
183             when AF_HARD_SIGMOID =>
184               -- When: hard sigmoid Function
185               -- https://keras.io/api/layers/activations/#hardsigmoid-function
186               -- 0 if if x <= -3
187               -- 1 if x >= 3
188               -- (x/6) + 0.5 if -3 < x < 3
189
190               LOOP_AF_HARD_SIGMOID : FOR idx_node_this in 0 to (g_layer_length_cur-1) LOOP
191                 if CUR_data_accum(idx_node_this) < c_s_dw_neg_3 then
192                   -- IF is less than "-3": return "0"
193                   NEX_layer_out(idx_node_this) <= (others => '0');
194                 elsif CUR_data_accum(idx_node_this) > c_s_dw_pos_3 then
195                   -- IF is greater than "+3": return "1"
196                   NEX_layer_out(idx_node_this) <= c_pos_one;
197                 else
198                   -- ELSE: linear function: y = (x/6) + 0.5
199                   v_dw_AF_temp1 := signed(STD_LOGIC_VECTOR(CUR_data_accum(idx_node_this)(c_DATA_WIDTH + c_DATA_QF - 1 downto c_DATA_QF)));
200                   v_dw_AF_temp2 := v_dw_AF_temp1 * c_s_pos_sixth;
201                   v_dw_AF_temp1 := c_s_pos_half + v_dw_AF_temp2(c_DATA_WIDTH + c_DATA_QF - 1 downto c_DATA_QF);
202                   NEX_layer_out(idx_node_this) <= STD_LOGIC_VECTOR( v_dw_AF_temp1 );
203
204                   --NEX_layer_out(idx_node_this) <= STD_LOGIC_VECTOR(CUR_data_accum(idx_node_this)(c_DATA_WIDTH + c_DATA_QF - 1 downto c_DATA_QF));
205                 end if;
206               end loop;
207
208             when others =>
209               -- When: Identity Function
```

```
210             LOOP_AF_IDENTITY : FOR idx_node_this in 0 to (g_layer_length_cur-1)
LOOP
211               NEX_layer_out(idx_node_this) <= STD_LOGIC_VECTOR(
CUR_data_accum(idx_node_this)(c_DATA_WIDTH + c_DATA_QF - 1 downto c_DATA_QF) );
212             end loop;
213           end case;
214
215           NEX_ready_to_TX <= '1';
216           NEX_state <= IDLE_TX;
217
218         when others =>
219           -- default catch others
220           NEX_state <= RESET_STATE;
221       end case;
222     end process;
223
224     P_CLK : process(clk)
225     begin
226       if rising_edge(clk) then
227         if reset = '1' then
228           -- internal signals
229           CUR_state <= RESET_STATE; -- default state: reset
230           CUR_node_prev <= 0;
231
232           CUR_data_in   <= (others=>(others=>'0'));
233           CUR_data_accum <= (others=>(others=>'0'));
234
235           -- outputs
236           ready_to_TX <= '0';
237           ack_RX <= '0';
238           layer_out <= (others=>(others=>'0'));
239         else
240           -- internal signals
241           CUR_state     <= NEX_state;
242           CUR_node_prev <= NEX_node_prev;
243
244           CUR_data_in   <= NEX_data_in;
245           CUR_data_accum <= NEX_data_accum;
246
247           -- outputs
248           ready_to_TX <= NEX_ready_to_TX;
249           ack_RX <= NEX_ack_RX;
250           layer_out   <= NEX_layer_out;
251         end if;
252       end if;
253     end process;
254 end architecture rtl;
```

C. VHDL code: Mediator

Code of Mediator block, which is used to test `Layer-01`.

For details on the “LENGTH” attribute of multi-dimensional arrays see [13, p. 441].

`c_007_mediator_rtl.vhd`

```
1  --
2  -- VHDL Architecture proj_master_2025_lib.c_007_mediator.rtl
3  --
4  -- Created:
5  --          by - Admin.UNKNOWN (LAPTOP-7KFJT032)
6  --          at - 13:57:03 10.04.2025
7  --
8  -- using Mentor Graphics HDL Designer(TM) 2017.1a (Build 5)
9  --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13
14 library proj_master_2025_lib;
15 use proj_master_2025_lib.p_002_generic_01.all;
16
17 entity c_007_mediator is
18    generic(
19       g_layer_length_first : integer           := 4;
20       g_layer_length_last  : integer           := 2;
21       g_inputs             : t_array2D_integer := ((0,0,0,0),(0,0,0,0))
22    );
23    port(
24       clk           : in     std_logic;
25       reset         : in     std_logic;
26       dst_RX        : in     std_logic;
27       src_TX        : in     std_logic;
28       ready_to_TX   : out    std_logic;
29       ack_RX        : out    std_logic;
30       layer_provide : out    t_array_data_stdlv (0 to g_layer_length_first-1);
31       layer_accept  : in     t_array_data_stdlv (0 to g_layer_length_last-1)
32    );
33
34 -- Declarations
35
36 end c_007_mediator ;
37
38 --
39 architecture rtl of c_007_mediator is
40   --SIGNAL NEX_state, CUR_state : t_stm_layer;
41   signal cnt: integer := 0;
42   signal cntAccept: integer := 0;
43
44   signal src_TX_last: std_logic;
45   signal dst_RX_last: std_logic;
46
47   signal inputs_accepted : t_array2D_integer (0 to g_inputs'length(1)-1, 0 to
   g_layer_length_last-1);
48
49 begin
50   P_CLK : process(clk)
```

```
51    begin
52      if rising_edge(clk) then
53        src_TX_last <= src_TX;
54        dst_RX_last <= dst_RX;
55
56        if reset = '1' then
57          cnt <= 0;
58          cntAccept <= 0;
59          ready_to_TX <= '0';
60          ack_RX <= '0';
61          layer_provide <= (others => (others => '0'));
62          inputs_accepted <= (others => (others => 1));
63        else
64          ready_to_TX <= '1';
65          ack_RX <= '1';
66
67          if cnt < g_inputs'LENGTH(1) then
68            -- we are at "g_inputs(cnt,:)"
69            LOOP_1 : FOR idx in 0 to g_inputs'LENGTH(2)-1 LOOP
70              layer_provide(idx) <= STD_LOGIC_VECTOR( TO_SIGNED(g_inputs( cnt, idx),
    c_DATA_WIDTH) );
71            end loop;
72          else
73            layer_provide <= (others => (others => '0'));
74          end if;
75
76          -- Detect when our Target has acknowledged our current transmission
77          if dst_RX = '1' and dst_RX_last = '0' then
78            cnt <= cnt + 1;
79          end if;
80
81          -- Detect when our Source has completed a calculation
82          if src_TX = '1' and src_TX_last = '0' then
83            cntAccept <= cntAccept + 1;
84
85            if cntAccept < inputs_accepted'LENGTH(1) then
86              LOOP_2 : FOR idx in 0 to inputs_accepted'LENGTH(2)-1 LOOP
87                inputs_accepted(cnt, idx) <= TO_INTEGER( SIGNED(layer_accept(idx) )
    );
88              end loop;
89            end if;
90          end if;
91        end if; -- if reset else
92      end if; -- if CLK
93    end process;
94  end architecture rtl;
```

D. VHDL code: Conversion template component

Code of template component, which will use one or more instances of `Layer-01` to recreate the neural network in VHDL.

Combined v2.vhd

```
1  -- VHDL Entity {$NAME_ENTITY}
2  --
3  -- Created: {$DATE_TIME}
4  --
5  --
6  LIBRARY ieee;
7  USE ieee.std_logic_1164.all;
8  USE ieee.numeric_std.all;
9
10 entity {$NAME_ENTITY} is
11    port(
12       clk          : in     std_logic;
13       reset        : in     std_logic;
14
15       src_TX       : in     std_logic;
16       ack_RX       : out    std_logic;
17
18       dst_RX       : in     std_logic;
19       ready_to_TX  : out    std_logic;
20
21       -- layer_in    : in     t_array_data_stdlv (0 to 1);
22       -- layer_out   : out    t_array_data_stdlv (0 to 1)
23 {$PORT_LAYER_IN_OUT}
24    );
25
26 -- Declarations
27
28 end {$NAME_ENTITY} ;
29
30 --
31 -- VHDL Architecture {$NAME_ENTITY}.struct
32 --
33 -- Created: {$DATE_TIME}
34 --
35 --
36 LIBRARY ieee;
37 USE ieee.std_logic_1164.all;
38 USE ieee.numeric_std.all;
39 library proj_master_2025_lib;
40 use proj_master_2025_lib.p_002_generic_01.all;
41
42
43 architecture struct of {$NAME_ENTITY} is
44
45    -- Architecture declarations
46
47    -- Internal signal declarations
48    -- signal dst_RX1      : std_logic;
49    -- signal layer_out1   : t_array_data_stdlv(0 to 1);
50    -- signal ready_to_TX1 : std_logic;
51 {$SIGNAL_DECLARATION}
52
```

```
53
54    -- Component Declarations
55    component c_004_layer_01
56    generic (
57       g_layer_length_cur  : integer               := 4;
58       g_layer_length_prev : integer               := 2;
59       g_layer_bias        : t_array_integer       := (0,0,0,0);
60       g_layer_weights     : t_array2D_integer     := ((0,0),(0,0),(0,0),(0,0));
61       g_act_func          : t_activation_function := AF_RELU
62    );
63    port (
64       clk         : in     std_logic ;
65       reset       : in     std_logic ;
66       dst_RX      : in     std_logic ;
67       src_TX      : in     std_logic ;
68       ready_to_TX : out    std_logic ;
69       ack_RX      : out    std_logic ;
70       layer_in    : in     t_array_data_stdlv (0 to g_layer_length_prev-1);
71       layer_out   : out    t_array_data_stdlv (0 to g_layer_length_cur-1)
72    );
73    end component;
74
75    -- Optional embedded configurations
76    -- pragma synthesis_off
77    for all : c_004_layer_01 use entity proj_master_2025_lib.c_004_layer_01;
78    -- pragma synthesis_on
79
80
81 begin
82
83    -- Instance port mappings.
84 {$INSTANCE_PORT_MAPPINGS}
85
86 end struct;
```

E. Python code: Keras ANN creation and training

Python code for generating the model and training it, using the Keras module.

ANN_03_ann_keras.py

```
# -*- coding: utf-8 -*-
"""
Created on Fri May 16 10:32:36 2025

@author: Admin
"""
#%% IMPORTS

import os

import numpy as np

from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout

import keras

#%%

path_out_dir = os.path.join("OUTPUT", "03_ann")
# Check whether the specified path exists or not
isExist = os.path.exists(path_out_dir)
if not isExist:
    # Create a new directory because it does not exist
    os.makedirs(path_out_dir)
    print("The new directory is created!"+"\n"+f"{path_out_dir}")
#%% ANN

model = Sequential()
L1 = 8

model.add(Dense( L1, activation = 'relu', input_shape = (2,) ))

model.add(Dense(1, activation = 'hard_sigmoid'))
# model.add(Dense(1, activation = 'sigmoid'))

model.compile(loss='MeanSquaredError', optimizer='adam', metrics=['accuracy'])
# model.compile(loss='BinaryCrossentropy', optimizer='adam', metrics=['accuracy'])

#%% Get Data
# https://stackoverflow.com/questions/70230687/how-keras-utils-sequence-
works/70319612#70319612

class DatasetCircle(keras.utils.Sequence):

    def __init__(self, length, batch_size=32, shuffle=True):
        # Initialization
        self.batch_size = batch_size
        self.shuffle = shuffle
        # self.x = x_in
        # self.y = y_in
        self.datalen = length
        self.indexes = np.arange(self.datalen)
        if self.shuffle:
```

```
54              np.random.shuffle(self.indexes)
55          self.__gen_data()
56
57      def __getitem__(self, index):
58          # get batch indexes from shuffled indexes
59          batch_indexes =
self.indexes[index*self.batch_size:(index+1)*self.batch_size]
60          x_batch = self.x[batch_indexes]
61          y_batch = self.y[batch_indexes]
62          return x_batch, y_batch
63
64      def __len__(self):
65          # Denotes the number of batches per epoch
66          return self.datalen // self.batch_size
67
68      def on_epoch_end(self):
69          # Updates indexes after each epoch
70          self.indexes = np.arange(self.datalen)
71          if self.shuffle:
72              np.random.shuffle(self.indexes)
73
74      def __gen_data(self):
75          # self.x = np.zeros((self.datalen,2))
76          # self.y = np.zeros((self.datalen,))
77
78          array_radius = np.random.rand(self.datalen)*2
79          array_angle = np.random.rand(self.datalen)*2*np.pi
80
81          self.x = np.array([array_radius*np.cos(array_angle),
array_radius*np.sin(array_angle)])
82          self.x = np.transpose(self.x)
83          self.y = array_radius<=1
84
85          pass
86
87  #%%
88
89  N = 1e5
90
91  training_generator   = DatasetCircle(int(N / 32)*32, batch_size=32)
92  validation_generator = DatasetCircle(int(N/2/32)*32, batch_size=32)
93
94  #%%
95  print("-- model training")
96  NUM_EPOCHS = 5
97
98  # model.fit(X_train, y_train, epochs=5, sample_weight=w_train)
99  model.fit_generator(generator=training_generator,
    validation_data=validation_generator,
100                     epochs=NUM_EPOCHS)
101
102 weights = model.get_weights()
103
104 print("-- training done")
105
106 #%% store model
107
108
109 fpath = os.path.join(path_out_dir, "circle_model.keras")
110 model.save(fpath, True, True)
```

```
111
112 raise Exception("End")
113
114 #%%
115 fpath = os.path.join(path_out_dir, "circle_model.keras")
116
117 model = keras.models.load_model(fpath)
```

F. Python code: PyTorch ANN creation and training

Python code for generating the model and training it, using the PyTorch module.

ANN_03c_ann_pytorch.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 23 11:44:53 2025

@author: Admin

https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial2/Introduction_to_PyTorch.html#Learning-by-example:-Continuous-XOR
"""
#%% IMPORTS

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

import torch.utils.data as data


from tqdm import tqdm


import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

#%% Network
# https://pytorch.org/docs/main/nn.html#loss-functions
# https://pytorch.org/docs/main/nn.html#non-linear-activations-weighted-sum-nonlinearity

# https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

class NN(nn.Module): # inherit from nn.Module

    def __init__(self, num_inputs, num_hidden, num_outputs):
        super(NN, self).__init__()
        # Initialize the modules we need to build the network
        # self.linear1 = nn.Linear(num_inputs, num_hidden)
        # self.linear2 = nn.Linear(num_hidden, num_outputs)
        # # self.act_relu = F.relu
        # self.act_relu = nn.ReLU()
        # # self.act_hsigm = F.hardsigmoid
        # self.act_hsigm = nn.Hardsigmoid()

        self.linear_stack = nn.Sequential(
            nn.Linear(num_inputs, num_hidden),
            nn.ReLU(),
            nn.Linear(num_hidden, num_outputs),
            nn.Hardsigmoid(),
        )
        pass
```

```
52
53      def forward(self, x):
54          # Perform the calculation of the model to determine the prediction
55          x = self.linear_stack(x)
56          return x
57
58
59  # save class
60  # torch.save(NN, "OUTPUT/03c_ann_pytorch_class.tar")
61
62  #%% Set Device
63  device = torch.device("cuda" if torch.cuda.is_available() else "cpu" )
64
65
66
67  #%% Hyperparameters
68  num_inputs = 2
69  num_hidden = 8
70  num_outputs = 1
71
72  batch_size = 64
73  num_epochs = 5
74
75  #%% Initialize Network
76
77  model = NN(num_inputs=num_inputs, num_hidden=num_hidden,
    num_outputs=num_outputs).to(device)
78  # Printing a module shows all its submodules
79  print(model)
80
81  #%% Dataset
82  class CircleDataset(data.Dataset):
83      def __init__(self, size=1000):
84          super().__init__()
85          size = int(size)
86          self.size = size
87          self.generate_dataset()
88
89
90
91
92      def generate_dataset(self):
93          self.data = torch.zeros((self.size,2), dtype=torch.float32)
94          self.label = torch.zeros((self.size), dtype=torch.float32)
95
96          args = torch.rand((self.size,2), dtype=torch.float32)
97          args[:,0] *= 2            # rand radius 0 to 2
98          args[:,1] *= 2*torch.pi   # rand angle 0 to 2*pi
99
100         self.data[:,0] = args[:,0] * torch.cos( args[:,1] ) # x pos
101         self.data[:,1] = args[:,0] * torch.sin( args[:,1] ) # y pos
102
103         self.label = (args[:,0]<1).long()
104
105     def __len__(self):
106         # Number of data point we have. Alternatively self.data.shape[0], or
    self.label.shape[0]
107         return self.size
108     def __getitem__(self, idx):
109         # Return the idx-th data point of the dataset
```

```
110         # If we have multiple things to return (data point and label), we can return them as tuple
111         return self.data[idx], self.label[idx]
112 
113 
114 #%% Get Training Dataset
115 # data_loader = data.DataLoader(dataset, batch_size=8, shuffle=True)
116 train_dataset = CircleDataset(size=1e6)
117 train_data_loader = data.DataLoader(train_dataset, batch_size=128, shuffle=True)
118 
119 #%% Loss and Optimizer
120 loss_module = nn.MSELoss()
121 optimizer = torch.optim.Adam(model.parameters())
122 
123 #%% Train Network
124 def train_model(model, optimizer, data_loader, loss_module, num_epochs=100):
125     # Set model to train mode
126     model.train()
127 
128     # Training loop
129     # for epoch in tqdm(range(num_epochs)):
130     for epoch in (range(num_epochs)):
131         # print(f"Epoch {epoch+1} / {num_epochs}")
132         for data_inputs, data_labels in tqdm(data_loader, desc=f"Epoch {epoch+1:02d}/{num_epochs:02d}"):
133             # Step 1: Move input data to device
134             data_inputs = data_inputs.to(device)
135             data_labels = data_labels.to(device)
136 
137             # Step 2: Run the model on the input data
138             preds = model(data_inputs)
139             preds = preds.squeeze(dim=1) # Output is [Batch size, 1], but we want [Batch size]
140 
141             # Step 3: Calculate the loss
142             loss = loss_module(preds, data_labels.float())
143 
144             # Step 4: Perform backpropagation
145             optimizer.zero_grad() # reset optimizer
146             loss.backward()
147 
148             # Step 5: Update the parameters
149             optimizer.step()
150 #%%
151 
152 train_model(model, optimizer, train_data_loader, loss_module, num_epochs=num_epochs)
153 #%% Save Model
154 state_dict = model.state_dict()
155 
156 # # torch.save(object, filename). For the filename, any extension can be used
157 # torch.save(state_dict, "OUTPUT/03c_ann_pytorch.tar")
158 
159 # # save whole model
160 # torch.save(model, "OUTPUT/03c_ann_pytorch_model.tar")
161 
162 
163 # ALTERNATIVE: https://pytorch.org/tutorials/beginner/saving_loading_models.html
164 model_scripted = torch.jit.script(model) # Export to TorchScript
165 model_scripted.save('03c_ann_pytorch_model.pt') # Save
```

G. Python code: Keras ANN conversion to VHDL

Python code for converting the model to a VHDL component, using the Keras module.

ANN_04_ann_to_vhdl.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 25 17:24:52 2025

@author: Admin
"""
#%% IMPORTS
import os
import datetime

import numpy as np

import keras

#%%

SUMMARY_NETWORK = True
SUMMARY_EDITS = True

#%%
path_out_dir = os.path.join("OUTPUT", "04_ann_to_vhdl")
# Check whether the specified path exists or not
isExist = os.path.exists(path_out_dir)
if not isExist:
    # Create a new directory because it does not exist
    os.makedirs(path_out_dir)
    print("The new directory is created!"+"\n"+f"{path_out_dir}")

del isExist

#%% Load Model

path_dir_03 = os.path.join("OUTPUT", "03_ann")

fpath_input = os.path.join(path_dir_03, "circle_model.keras")

model = keras.models.load_model(fpath_input)

weights = model.get_weights()

#%%

ann_conf = model.get_config()

n = ann_conf["name"]
# must be a sequential ANN
assert n[:10] == "sequential"

ann_layers = ann_conf["layers"]


layers = {"num_layers":0, "LAYER":{}}
units_prev = 0
layer_idx = 0
```

```
55
56  for i,entry in enumerate(ann_layers):
57      # print("-- Layer", i, entry["class_name"])
58
59      if entry["class_name"] == "InputLayer":
60          layer_conf = entry["config"]
61          layers["input_size"] = layer_conf["batch_input_shape"][1]
62          units_prev = layer_conf["batch_input_shape"][1]
63
64      elif entry["class_name"] == "Dense":
65          layer_conf = entry["config"]
66          layers["LAYER"][layer_idx] = {"units": layer_conf["units"],
67                                        "units_prev": units_prev,
68                                        "activation": layer_conf["activation"]}
69          layers["num_layers"] += 1
70          units_prev = layer_conf["units"]
71          layer_idx += 1
72
73          pass
74      else:
75          raise Exception("Unknown Type of Layer")
76          pass
77
78      pass
79
80  del units_prev, layer_idx, n, entry
81
82  #%% SUMMARY
83
84  if SUMMARY_NETWORK:
85      print("\n")
86      print("\t","-------------------------------")
87      print("\t","--","Summary of Network:")
88      print("\t","-------------------------------")
89      print("Number Of Layers:",layers["num_layers"])
90      print("")
91      for k in layers["LAYER"]:
92          entry = layers["LAYER"][k]
93          print("-- Layer",k)
94          print("\t", f'Units: {entry["units"]}, Units_Prev: {entry["units_prev"]},
    Activation: {entry["activation"]}')
95
96      print("\t","_____________________________________")
97      print("\t","_____________________________________")
98      print("\n")
99
100
101 #%% CONVERSION
102
103 DATA_WIDTH = 16
104 DATA_QF = 8
105 FP_ONE = 2**DATA_QF
106
107 inst_port_maps = [""]*layers["num_layers"]
108
109 list_signals = []
110
111 list_port_inout = []
112
113
```

```
114 for i in range(layers["num_layers"]):
115 
116     layer = layers["LAYER"][i]
117 
118     w = weights[2*i] * FP_ONE
119     w = np.int64(w)
120     w = np.transpose(w)
121 
122     b = weights[2*i + 1] * FP_ONE
123     b = np.int64(b)
124 
125 
126     txt_component =  f"U_{i} : c_004_layer_01" + "\n{};\n"
127 
128 
129     txt_generic = "generic map (\n{}\n)"
130     txt_port = "port map (\n{}\n)"
131 
132 
133 
134     #-------
135     #------- GENERIC MAP
136     #-------
137 
138     list_generics = []
139 
140     list_generics.append("  "+f"g_layer_length_cur => {layer['units']}")
141     #-------
142     list_generics.append("  "+f"g_layer_length_prev => {layer['units_prev']}")
143     #-------
144 
145     # Bias: is 1D array.
146     # We take care to correctly format, if the array only contains 1 element
147     txt_b1 = "( {} )"
148     if len(b) == 1:
149         txt_b1 = "( 0 => {} )"
150 
151     b1 = [str(x) for x in b]
152     b1 = txt_b1.format(", ".join(b1))
153     txt_bias = "  "+f"g_layer_bias => {b1}"
154     list_generics.append(txt_bias)
155     #-------
156 
157     # Weights: is a 2d array, where the first index is the current unit
158     # and the second index is the previous unit.
159     # We take care to correctly format, if an array only contains 1 element
160 
161     txt_w1 = "( {} )"
162     if len(w) == 1:
163         txt_w1 = "( 0 => {} )"
164         pass
165 
166     list_w2 = []
167     for elem in w:
168         txt_w2 = "({})"
169         if len(elem) == 1:
170             txt_w2 = "( 0 => {} )"
171             pass
172 
173         w2 = [str(x) for x in elem]
```

```
174             w2 = txt_w2.format(", ".join(w2))
175             list_w2.append(w2)
176             pass
177         w1 = txt_w1.format(", ".join(list_w2))
178 
179         txt_weights = "  "+f"g_layer_weights => {w1}"
180         list_generics.append(txt_weights)
181         #-------
182 
183 
184         list_generics.append("  "+f"g_act_func => AF_{layer['activation'].upper()}")
185         #-------
186 
187 
188 
189         txt_generic = txt_generic.format(",\n".join(list_generics))
190         del list_generics, txt_weights, list_w2, txt_w1, txt_w2, w, w1, w2
191         del txt_bias, txt_b1, b1, b, elem
192 
193 
194 
195         #-------
196         #------- PORT MAP
197         #-------
198 
199         # https://asciiflow.com/
200         #                          ┌──────────────────────┐
201         #                          │                      │
202         # ──────>                  │ CLK                  │
203         #                          │                      │
204         # ──────>                  │ RESET                │
205         #                          │                      │
206         #                          │                      │
207         # <──────                  │ ACK_RX         DST_RX│ <──────
208         #                          │                      │
209         # ──────>                  │ SRC_TX           R2TX│ ──────>
210         #                          │                      │
211         #                          │                      │
212         # ──────>                  │ Layer_in    Layer_out│ ──────>
213         #                          │                      │
214         #                          └──────────────────────┘
215 
216         signal_decl_DST_RX = ""
217         signal_decl_R2TX = ""
218         signal_decl_Layer = ""
219         if i == 0:
220             signal_ACK_RX    = "ack_RX"
221             signal_SRC_TX    = "src_TX"
222             signal_Layer_in  = "layer_in"
223 
224             signal_DST_RX    = f"DST_RX_{i}"
225             signal_R2TX      = f"R2TX_{i}"
226             signal_Layer_out = f"layer_{i}"
227 
228             signal_decl_DST_RX = f"signal {signal_DST_RX} : std_logic;"
229             signal_decl_R2TX   = f"signal {signal_R2TX} : std_logic;"
230             signal_decl_Layer  = f"signal {signal_Layer_out} : t_array_data_stdlv(0 to
    {layer['units'] - 1});"
231             list_signals.append(signal_decl_DST_RX)
232             list_signals.append(signal_decl_R2TX)
```

```
233         list_signals.append(signal_decl_Layer)
234 
235         list_port_inout.append(f"layer_in : in t_array_data_stdlv (0 to
    {layer['units_prev'] - 1})")
236         pass
237 
238     elif i == (layers["num_layers"]-1):
239         signal_ACK_RX    = f"DST_RX_{i-1}"
240         signal_SRC_TX    = f"R2TX_{i-1}"
241         signal_Layer_in  = f"layer_{i-1}"
242 
243         signal_DST_RX    = "dst_RX"
244         signal_R2TX      = "ready_to_TX"
245         signal_Layer_out = "layer_out"
246 
247         list_port_inout.append(f"layer_out : out t_array_data_stdlv (0 to
    {layer['units'] - 1})")
248         pass
249 
250     else:
251         signal_ACK_RX    = f"DST_RX_{i-1}"
252         signal_SRC_TX    = f"R2TX_{i-1}"
253         signal_Layer_in  = f"layer_{i-1}"
254 
255         signal_DST_RX    = f"DST_RX_{i}"
256         signal_R2TX      = f"R2TX_{i}"
257         signal_Layer_out = f"layer_{i}"
258 
259         signal_decl_DST_RX = f"signal {signal_DST_RX} : std_logic;"
260         signal_decl_R2TX   = f"signal {signal_R2TX} : std_logic;"
261         signal_decl_Layer  = f"signal {signal_Layer_out} : t_array_data_stdlv(0 to
    {layer['units'] - 1});"
262         list_signals.append(signal_decl_DST_RX)
263         list_signals.append(signal_decl_R2TX)
264         list_signals.append(signal_decl_Layer)
265         pass
266 
267 
268 
269 
270     list_ports = []
271 
272     list_ports.append("  "+f"clk => clk")
273     list_ports.append("  "+f"reset => reset")
274     #-------
275     list_ports.append("  "+f"ack_RX => {signal_ACK_RX}")
276     list_ports.append("  "+f"src_TX => {signal_SRC_TX}")
277     list_ports.append("  "+f"layer_in => {signal_Layer_in}")
278     #-------
279     list_ports.append("  "+f"dst_RX => {signal_DST_RX}")
280     list_ports.append("  "+f"ready_to_TX => {signal_R2TX}")
281     list_ports.append("  "+f"layer_out => {signal_Layer_out}")
282 
283     # print(list_ports)
284     txt_port = txt_port.format(",\n".join(list_ports))
285 
286     # print(txt_port)
287     del signal_ACK_RX, signal_SRC_TX, signal_Layer_in
288     del signal_DST_RX, signal_R2TX, signal_Layer_out
289     del signal_decl_DST_RX, signal_decl_R2TX, signal_decl_Layer
```

```
290     del list_ports
291 
292 
293 
294 
295     #-------
296     #------- Complete Component
297     #-------
298 
299     txt_component =  txt_component.format("\n".join([txt_generic, txt_port]))
300     # print(txt_component)
301     inst_port_maps[i] = txt_component
302 
303     del txt_component, txt_generic, txt_port
304 
305     pass
306 
307 #-------
308 #------- Signals
309 #-------
310 txt_signals = "\n".join(list_signals)
311 del list_signals
312 
313 
314 #-------
315 #------- Port Layer in out
316 #-------
317 txt_port_layer_inout = ";\n".join(list_port_inout)
318 del list_port_inout
319 
320 
321 
322 
323 inst_port_maps = "\n".join(inst_port_maps)
324 
325 
326 
327 
328 #%% SUMMARY
329 
330 if SUMMARY_EDITS:
331     print("\t","------------------------------")
332     print("\t","--","Edit in Port declarations:")
333     print("\t","------------------------------")
334     print(txt_port_layer_inout)
335 
336     print("\n")
337     print("\t","------------------------------")
338     print("\t","--","Edit in Signal declarations:")
339     print("\t","------------------------------")
340     print(txt_signals)
341 
342     print("\n")
343     print("\t","------------------------------")
344     print("\t","--","Edit in Instance port mappings:")
345     print("\t","------------------------------")
346     print(inst_port_maps)
347 
348     print("\t","_____________________________")
349     print("\t","_____________________________")
```

```
350     print("\n")
351 
352 #%%
353 
354 path_in_dir = os.path.join("INPUT", "04_ann_to_vhdl")
355 
356 if not os.path.exists(path_in_dir):
357     raise Exception(f"Path does not exist: \n{path_in_dir}")
358     pass
359 
360 file_name = "Combined v2.vhd"
361 fPathIn = os.path.join(path_in_dir, file_name)
362 # D:\DCD_workspace\python\MA25_Python\INPUT\04_ann_to_vhdl\Combined v2.vhd
363 
364 #%%
365 
366 with open(fPathIn) as f:
367     fileStr = f.read()
368     pass
369 
370 EntityName = "c_x_ANN_01"
371 fileNameOut = f"{EntityName}.vhd"
372 fPathOut = os.path.join(path_out_dir, fileNameOut)
373 
374 fileStr = fileStr.replace("{$NAME_ENTITY}", EntityName)
375 
376 
377 tnow = datetime.datetime.now()
378 fileStr = fileStr.replace("{$DATE_TIME}", str(tnow))
379 
380 fileStr = fileStr.replace("{$SIGNAL_DECLARATION}", txt_signals)
381 
382 fileStr = fileStr.replace("{$INSTANCE_PORT_MAPPINGS}", inst_port_maps)
383 
384 fileStr = fileStr.replace("{$PORT_LAYER_IN_OUT}", txt_port_layer_inout)
385 
386 with open(fPathOut, mode="w") as f:
387     f.write(fileStr)
388 
389 print(f"File '{fileNameOut}' created/overwritten!")
```

H. Console output: Keras ANN conversion to VHDL

Console output for converting the model to a VHDL component, using the Keras module.

```
    ------------------------------
    -- Summary of Network:
    ------------------------------
Number Of Layers: 2

-- Layer 0
    Units: 8, Units_Prev: 2, Activation: relu
-- Layer 1
    Units: 1, Units_Prev: 8, Activation: hard_sigmoid
    _________________________________
    _________________________________


    ------------------------------
    -- Edit in Port declarations:
    ------------------------------
layer_in : in t_array_data_stdlv (0 to 1);
layer_out : out t_array_data_stdlv (0 to 0)


    ------------------------------
    -- Edit in Signal declarations:
    ------------------------------
signal DST_RX_0 : std_logic;
signal R2TX_0 : std_logic;
signal layer_0 : t_array_data_stdlv(0 to 7);


    ------------------------------
    -- Edit in Instance port mappings:
    ------------------------------
U_0 : c_004_layer_01
generic map (
  g_layer_length_cur => 8,
  g_layer_length_prev => 2,
  g_layer_bias => ( -52, -357, -363, -446, -328, -336, -38, -369 ),
  g_layer_weights => ( (1, -29), (598, 64), (-17, -638), (-563, -456), (417, -386), (187,
608), (-16, -10), (-611, 334) ),
  g_act_func => AF_RELU
)
port map (
  clk => clk,
  reset => reset,
  ack_RX => ack_RX,
  src_TX => src_TX,
  layer_in => layer_in,
  dst_RX => DST_RX_0,
  ready_to_TX => R2TX_0,
  layer_out => layer_0
);

U_1 : c_004_layer_01
generic map (
  g_layer_length_cur => 1,
  g_layer_length_prev => 8,
```

```
  g_layer_bias => ( 0 => 1186 ),
  g_layer_weights => ( 0 => (91, -967, -784, -839, -802, -1051, 22, -975) ),
  g_act_func => AF_HARD_SIGMOID
)
port map (
  clk => clk,
  reset => reset,
  ack_RX => DST_RX_0,
  src_TX => R2TX_0,
  layer_in => layer_0,
  dst_RX => dst_RX,
  ready_to_TX => ready_to_TX,
  layer_out => layer_out
);
```

I. Python code: PyTorch ANN conversion to VHDL

Python code for converting the model to a VHDL component, using the PyTorch module.

ANN_04b_ann_to_vhdl_pytorch.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 24 12:25:11 2025

@author: Admin
"""
#%% IMPORTS
import os
import datetime

import numpy as np

import torch

#%%

SUMMARY_NETWORK = True
SUMMARY_EDITS = True

#%%
path_out_dir = os.path.join("OUTPUT", "04_ann_to_vhdl")
# Check whether the specified path exists or not
isExist = os.path.exists(path_out_dir)
if not isExist:
    # Create a new directory because it does not exist
    os.makedirs(path_out_dir)
    print("The new directory is created!"+"\n"+f"{path_out_dir}")

del isExist

#%% Load Model

""" INFO: We expect to only use loaded models via torch.jit.load of a TorchScript model!
    Therefore, we will rely on the 'original_name' attribute of the model children,
    to decide if they represent a valid network for this script."""

# https://pytorch.org/tutorials/beginner/saving_loading_models.html
model = torch.jit.load('03c_ann_pytorch_model.pt')
model.eval()


assert hasattr(model, 'original_name')
model_name = model.original_name


# fetch children of the model
chil_model = [ch for ch in model.children()]


# CHECK: We expect this to only contain 1 "Sequential" module als child (see 03c_ann_pytorch)
assert len(chil_model)==1 # must be only one element: a Sequential layer stack
mod = chil_model[0]
```

```
53
54
55  # CHECK: We expect it to be of the type "Sequential" (see 03c_ann_pytorch)
56  assert hasattr(mod, 'original_name')
57  assert mod.original_name == "Sequential"
58
59  chil_seq = [ch for ch in mod.children()]
60  ann_layers = chil_seq
61
62  #%%
63  list_layers = ["Linear",]
64  # list_activation = ["ReLU", "Hardsigmoid"]
65
66  # dict_map_pytorch_layer = {}
67
68
69  dict_map_activation = {
70      "ReLU": "RELU",
71      "Hardsigmoid": "HARD_SIGMOID",
72  }
73
74
75  layers = {"num_layers":0, "LAYER":{}}
76
77  for i,entry in enumerate(ann_layers):
78      n = entry.original_name
79      # print("-- Layer", i, f"[{n}]")
80
81      if n in list_layers:
82          # if is DENSE / Fully Connected Layer
83          if n == "Linear":
84              # layer_conf = entry["config"]
85              layers["LAYER"][layers["num_layers"]] = {
86                  "activation": "IDENTITY",
87                  "pytorchIdx": i,
88              }
89              # dict_map_pytorch_layer[layers["num_layers"]] = i
90
91              layers["num_layers"] += 1
92              pass
93          # else: if is conv layer, todo:future
94      elif n in dict_map_activation:
95          # if is Activation function (of prev layer)
96
97          if n == "ReLU":
98              layers["LAYER"][layers["num_layers"]-1]["activation"] =
    dict_map_activation[n]
99
100         elif n == "Hardsigmoid":
101             layers["LAYER"][layers["num_layers"]-1]["activation"] =
    dict_map_activation[n]
102
103     else:
104         raise Exception("Unknown Type of Layer")
105         pass
106
107     pass
108
109 #%% Fetch Weights and Biases
110
```

```
111 state_dict = model.state_dict()
112 postfix_weight = ".{}.weight"
113 postfix_bias   = ".{}.bias"
114 
115 # we must get the name of our dict entries
116 list_keys = [k for k in state_dict]
117 # so we fetch the first element and split with "." as separator
118 prefix = list_keys[0].split(".")[0]
119 
120 for idx in range(layers["num_layers"]):
121     entry = layers["LAYER"][idx]
122     # print(idx,entry)
123 
124     # the dict keys are a composite of prefix and postfix (formated with
    pytorchIdx)
125     pytorchIdx = entry["pytorchIdx"]
126     key_weights = prefix + postfix_weight.format(pytorchIdx)
127     key_bias    = prefix + postfix_bias.format(pytorchIdx)
128 
129     entry["weights"] = state_dict[key_weights].numpy()
130     entry["bias"]    = state_dict[key_bias].numpy()
131     # entry["units"]
132 
133     entry["units"], entry["units_prev"] = entry["weights"].shape
134 
135 
136 
137 
138 #%% SUMMARY
139 
140 if SUMMARY_NETWORK:
141     print("\n")
142     print("\t","------------------------------")
143     print("\t","--","Summary of Network:")
144     print("\t","------------------------------")
145     print("Number Of Layers:",layers["num_layers"])
146     print("")
147     for k in layers["LAYER"]:
148         entry = layers["LAYER"][k]
149         print("-- Layer",k)
150         print("\t", f'Units: {entry["units"]}, Units_Prev: {entry["units_prev"]},
    Activation: {entry["activation"]}')
151 
152     print("\t","_____________________________")
153     print("\t","_____________________________")
154     print("\n")
155 
156 
157 #%% CONVERSION
158 
159 DATA_WIDTH = 16
160 DATA_QF = 8
161 FP_ONE = 2**DATA_QF
162 
163 inst_port_maps = [""]*layers["num_layers"]
164 
165 list_signals = []
166 
167 list_port_inout = []
168 
```

```
169
170 for i in range(layers["num_layers"]):
171
172     layer = layers["LAYER"][i]
173
174     weights = layer["weights"]
175     w = weights * FP_ONE
176     w = np.int64(w)
177     # w = np.transpose(w)
178
179     bias = layer["bias"]
180     b = bias * FP_ONE
181     b = np.int64(b)
182
183
184     txt_component =  f"U_{i} : c_004_layer_01" + "\n{};\n"
185
186
187     txt_generic = "generic map (\n{}\n)"
188     txt_port = "port map (\n{}\n)"
189
190
191
192     #-------
193     #------- GENERIC MAP
194     #-------
195
196     list_generics = []
197
198     list_generics.append("  "+f"g_layer_length_cur => {layer['units']}")
199     #-------
200     list_generics.append("  "+f"g_layer_length_prev => {layer['units_prev']}")
201     #-------
202
203     # Bias: is 1D array.
204     # We take care to correctly format, if the array only contains 1 element
205     txt_b1 = "( {} )"
206     if len(b) == 1:
207         txt_b1 = "( 0 => {} )"
208
209     b1 = [str(x) for x in b]
210     b1 = txt_b1.format(", ".join(b1))
211     txt_bias = "  "+f"g_layer_bias => {b1}"
212     list_generics.append(txt_bias)
213     #-------
214
215     # Weights: is a 2d array, where the first index is the current unit
216     # and the second index is the previous unit.
217     # We take care to correctly format, if an array only contains 1 element
218
219     txt_w1 = "( {} )"
220     if len(w) == 1:
221         txt_w1 = "( 0 => {} )"
222         pass
223
224     list_w2 = []
225     for elem in w:
226         txt_w2 = "({})"
227         if len(elem) == 1:
228             txt_w2 = "( 0 => {} )"
```

```
            pass

        w2 = [str(x) for x in elem]
        w2 = txt_w2.format(", ".join(w2))
        list_w2.append(w2)
        pass
    w1 = txt_w1.format(", ".join(list_w2))

    txt_weights = "  "+f"g_layer_weights => {w1}"
    list_generics.append(txt_weights)
    #-------


    list_generics.append("  "+f"g_act_func => AF_{layer['activation'].upper()}")
    #-------



    txt_generic = txt_generic.format(",\n".join(list_generics))
    del list_generics, txt_weights, list_w2, txt_w1, txt_w2, w, w1, w2
    del txt_bias, txt_b1, b1, b, elem



    #-------
    #------- PORT MAP
    #-------

    # https://asciiflow.com/
    #      ┌──────────────────────┐
    #      │                      │
    # ────>│ CLK                  │
    #      │                      │
    # ────>│ RESET                │
    #      │                      │
    #      │                      │
    # <────│ ACK_RX         DST_RX│<────
    #      │                      │
    # ────>│ SRC_TX           R2TX│────>
    #      │                      │
    #      │                      │
    # ────>│ Layer_in    Layer_out│────>
    #      │                      │
    #      └──────────────────────┘

    signal_decl_DST_RX = ""
    signal_decl_R2TX = ""
    signal_decl_Layer = ""
    if i == 0:
        signal_ACK_RX    = "ack_RX"
        signal_SRC_TX    = "src_TX"
        signal_Layer_in  = "layer_in"

        signal_DST_RX    = f"DST_RX_{i}"
        signal_R2TX      = f"R2TX_{i}"
        signal_Layer_out = f"layer_{i}"

        signal_decl_DST_RX = f"signal {signal_DST_RX} : std_logic;"
        signal_decl_R2TX   = f"signal {signal_R2TX} : std_logic;"
```

```
288         signal_decl_Layer  = f"signal {signal_Layer_out} : t_array_data_stdlv(0 to
    {layer['units'] - 1});"
289         list_signals.append(signal_decl_DST_RX)
290         list_signals.append(signal_decl_R2TX)
291         list_signals.append(signal_decl_Layer)
292 
293         list_port_inout.append(f"layer_in : in t_array_data_stdlv (0 to
    {layer['units_prev'] - 1})")
294         pass
295 
296     elif i == (layers["num_layers"]-1):
297         signal_ACK_RX    = f"DST_RX_{i-1}"
298         signal_SRC_TX    = f"R2TX_{i-1}"
299         signal_Layer_in  = f"layer_{i-1}"
300 
301         signal_DST_RX    = "dst_RX"
302         signal_R2TX      = "ready_to_TX"
303         signal_Layer_out = "layer_out"
304 
305         list_port_inout.append(f"layer_out : out t_array_data_stdlv (0 to
    {layer['units'] - 1})")
306         pass
307 
308     else:
309         signal_ACK_RX    = f"DST_RX_{i-1}"
310         signal_SRC_TX    = f"R2TX_{i-1}"
311         signal_Layer_in  = f"layer_{i-1}"
312 
313         signal_DST_RX    = f"DST_RX_{i}"
314         signal_R2TX      = f"R2TX_{i}"
315         signal_Layer_out = f"layer_{i}"
316 
317         signal_decl_DST_RX = f"signal {signal_DST_RX} : std_logic;"
318         signal_decl_R2TX   = f"signal {signal_R2TX} : std_logic;"
319         signal_decl_Layer  = f"signal {signal_Layer_out} : t_array_data_stdlv(0 to
    {layer['units'] - 1});"
320         list_signals.append(signal_decl_DST_RX)
321         list_signals.append(signal_decl_R2TX)
322         list_signals.append(signal_decl_Layer)
323         pass
324 
325 
326 
327 
328     list_ports = []
329 
330     list_ports.append("  "+f"clk => clk")
331     list_ports.append("  "+f"reset => reset")
332     #-------
333     list_ports.append("  "+f"ack_RX => {signal_ACK_RX}")
334     list_ports.append("  "+f"src_TX => {signal_SRC_TX}")
335     list_ports.append("  "+f"layer_in => {signal_Layer_in}")
336     #-------
337     list_ports.append("  "+f"dst_RX => {signal_DST_RX}")
338     list_ports.append("  "+f"ready_to_TX => {signal_R2TX}")
339     list_ports.append("  "+f"layer_out => {signal_Layer_out}")
340 
341     # print(list_ports)
342     txt_port = txt_port.format(",\n".join(list_ports))
343 
```

```
344     # print(txt_port)
345     del signal_ACK_RX, signal_SRC_TX, signal_Layer_in
346     del signal_DST_RX, signal_R2TX, signal_Layer_out
347     del signal_decl_DST_RX, signal_decl_R2TX, signal_decl_Layer
348     del list_ports
349 
350 
351 
352 
353     #-------
354     #------- Complete Component
355     #-------
356 
357     txt_component =  txt_component.format("\n".join([txt_generic, txt_port]))
358     # print(txt_component)
359     inst_port_maps[i] = txt_component
360 
361     del txt_component, txt_generic, txt_port
362 
363     pass
364 
365 #-------
366 #------- Signals
367 #-------
368 txt_signals = "\n".join(list_signals)
369 del list_signals
370 
371 
372 #-------
373 #------- Port Layer in out
374 #-------
375 txt_port_layer_inout = ";\n".join(list_port_inout)
376 del list_port_inout
377 
378 
379 
380 
381 inst_port_maps = "\n".join(inst_port_maps)
382 
383 
384 
385 
386 #%% SUMMARY
387 
388 if SUMMARY_EDITS:
389     print("\t","------------------------------")
390     print("\t","--","Edit in Port declarations:")
391     print("\t","------------------------------")
392     print(txt_port_layer_inout)
393 
394     print("\n")
395     print("\t","------------------------------")
396     print("\t","--","Edit in Signal declarations:")
397     print("\t","------------------------------")
398     print(txt_signals)
399 
400     print("\n")
401     print("\t","------------------------------")
402     print("\t","--","Edit in Instance port mappings:")
403     print("\t","------------------------------")
```

```
404     print(inst_port_maps)
405 
406     print("\t","_____________________________")
407     print("\t","_____________________________")
408     print("\n")
409 
410 #%%
411 
412 path_in_dir = os.path.join("INPUT", "04_ann_to_vhdl")
413 
414 if not os.path.exists(path_in_dir):
415     raise Exception(f"Path does not exist: \n{path_in_dir}")
416     pass
417 
418 file_name = "Combined v2.vhd"
419 fPathIn = os.path.join(path_in_dir, file_name)
420 # D:\DCD_workspace\python\MA25_Python\INPUT\04_ann_to_vhdl\Combined v2.vhd
421 
422 #%%
423 
424 with open(fPathIn) as f:
425     fileStr = f.read()
426     pass
427 
428 EntityName = "c_x_ANN_02"
429 fileNameOut = f"{EntityName}.vhd"
430 fPathOut = os.path.join(path_out_dir, fileNameOut)
431 
432 fileStr = fileStr.replace("{$NAME_ENTITY}", EntityName)
433 
434 
435 tnow = datetime.datetime.now()
436 fileStr = fileStr.replace("{$DATE_TIME}", str(tnow))
437 
438 fileStr = fileStr.replace("{$SIGNAL_DECLARATION}", txt_signals)
439 
440 fileStr = fileStr.replace("{$INSTANCE_PORT_MAPPINGS}", inst_port_maps)
441 
442 fileStr = fileStr.replace("{$PORT_LAYER_IN_OUT}", txt_port_layer_inout)
443 
444 with open(fPathOut, mode="w") as f:
445     f.write(fileStr)
446 
447 print(f"File '{fileNameOut}' created/overwritten!")
```