Deep Inference in Hardware

Contents

1 Introduction

Deep Inference in Hardware *is a concept project implemented at UAH to achieve inference in FPGA. In addition to being a custom implementation in a re-configurable hardware, it also provides an opportunity to be interfaced as a peripheral to a standard computer system. For this project, the peripheral is interfaced with NIOS-II computer system for accelerated inference operation. Our initial implementation shows that the inference operation achieves speedup of nearly* ***400*** *times when compared with similar inference routine implemented in software*.

"*AI is the new electricity*" a famous quote by Dr Andrew NG is enough to state the importance of the intelligence derived from computers. The quote makes a prediction that AI is the force that will soon be as widely used as electricity. The precursor of this soon-to-occur trend can be already seen in the way our lives are impacted by intelligent machines such as getting suggestions while writing emails to tagging family and friends in social networking sites are some of the activities that we are already doing. Teller-less grocery experience and cars that drive themselves are some of the works in progress. All these advancements have already improved lives in many ways while many other ways to improve lives are still being explored.

Amidst all these, it is worthwhile to note that Neural Network forms an integral component of machine intelligence. Neural Network is one of many tools of machine learning that helps machine acquire decision making capability. However, the popularity of Neural Network has risen significantly in the recent years because of the availability of machines capable of performing computations that is required for Neural Network. While other machine learning models are typically suited for particular problems, any classification type problem can best be solved by Neural Network if cost of implementation is not an issue, the solution to which is provided by recent advancement in hardware technologies.

The term Artificial Neural Network (ANN) refers to the Neural Network that is implemented by humans. To not confuse with the Neural Network in human nervous system, the word Artificially is added as a prefix. However, the idea on the construct of the ANN is derived from neuron in human nervous system.

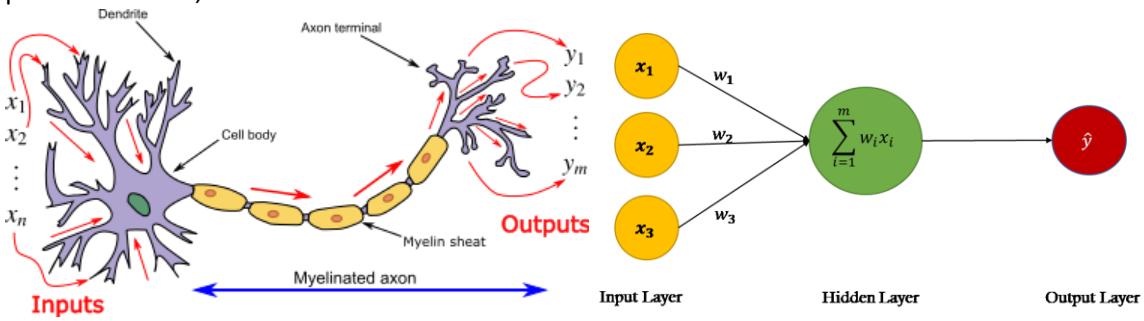

Figure 1 Human Neuron vs Artificial Neuron

In the images presented in Figure 1, the image to the left is a human neuron while the image on the right is the artificial neuron. Structurally, we can see that both of these neurons comprise of three important components: *Inputs*, *Weights (Connections)* and *Outputs*. The inputs are sensory signals that are received from the environments, weights are some constants that are wired to determine the contribution of each input values in decision making process and outputs are weighted combination of

inputs. The image presented in Figure 1 shows a very simplified representation of artificial neuron with three inputs and a single output leaving out another important term called *bias*. Bias are some constants that are added to the weighted combination of inputs in a neuron to give a DC shift.

In an artificial neuron, the weighted combination of the inputs is not directly passed as outputs. However, a decision-making technique is applied through something called *Activation Function*. This means, the weighted combination is passed on as input to the Activation Function the output of which is passed as output of the neuron. Following section shows some of the popular activation functions:

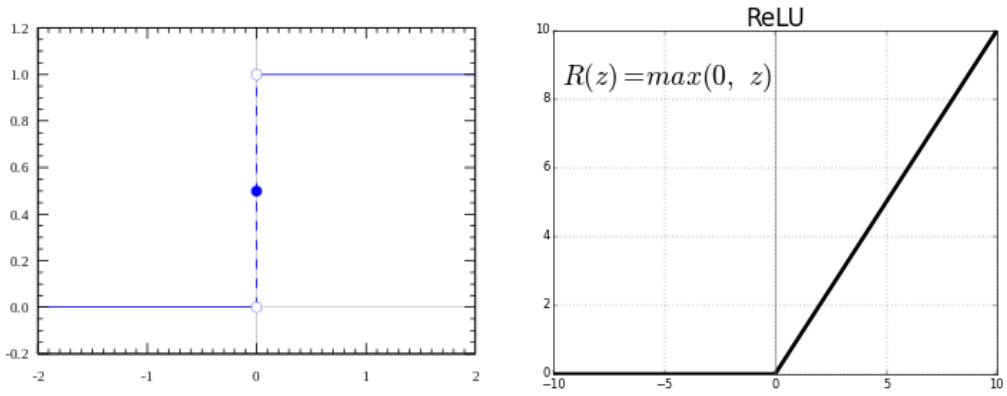

Figure 2 Step Function and ReLU Function

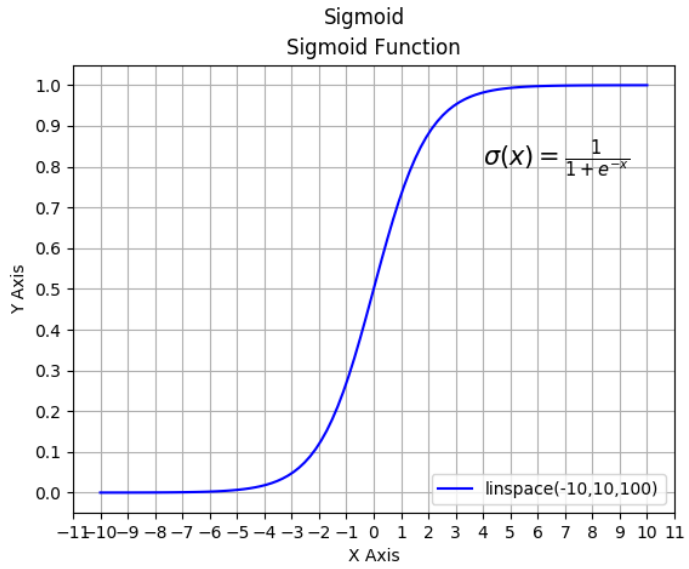

Figure 3 Sigmoid Function

2 Mathematical View

2.1 A Neuron

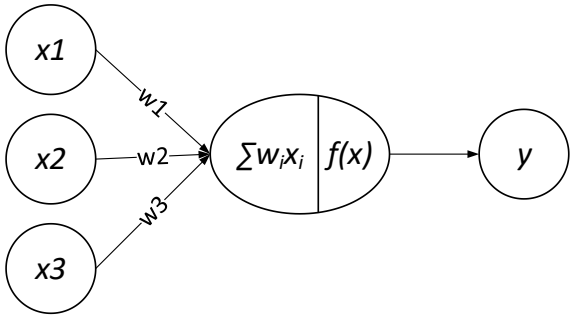

Figure 4 A Single Neuron in ANN

Figure 4 shows a single neuron with three inputs. Each weight in the connection between each inputs and the activation is weighted as *w1,w2* and *w3*. The weighted combination of the inputs *x1,x2* and *x3* is

$$\sum wixi = w1 * x1 + w2 * x2 + w3 * x3$$

which is essentially *dot product* of the two vectors, input vector and the weight vector. After applying activation function *f(.)* to the weighted combination of inputs, the output of the neuron is computed as

$$y = f(\sum wixi)$$

2.2 A Neural Network with Two Neurons

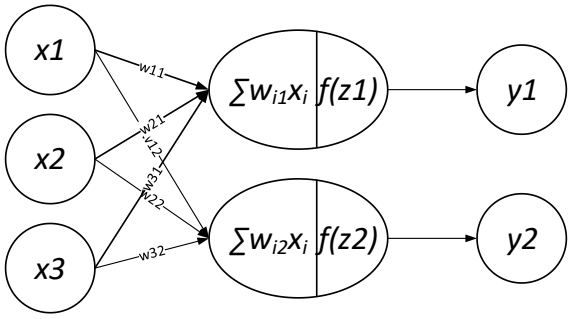

Figure 5 ANN with two neurons

The mathematical view changes slightly in case of multiple neurons. However, the problem can be seen as being scaled. On the first neuron, the weighted inputs can be still seen as dot product similar to above

$$\sum z1 = w11 * x1 + w21 * x2 + w31 * x3$$

$$\sum z2 = w12 * x1 + w22 * x2 + w32 * x3$$

If we look at the scaling of problem, it essentially is transferred to *vector- matrix multiplication* from a dot-product of vectors.

2.3 Deep Neural Network

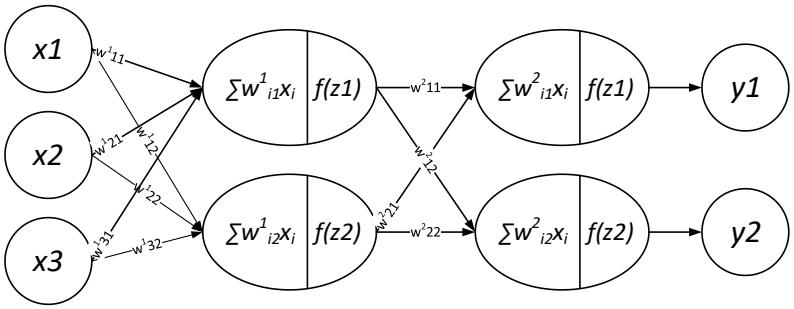

Figure 6 ANN Model with a single hidden layer

Figure 6 shows a network model that contains a single hidden layer. The problem presented in Figure 5 now scales to a cascaded problem with same operation in between. Thus mathematically, it is still a vector matrix multiplication with multiple layers.

Any ANN with multiple layers between input and output is termed as Deep Neural Network (DNN). Cascading another layer in Figure 6 (similar to cascading a layer in Figure 5 to produce Figure 6) should create a DNN where the mathematical operations are similarly scaled.

2.4 Bias

The weighted inputs in any layers are not directly fed to the activation function. Rather, they are added with Bias so that a DC shift is achieved. Thus mathematically, before passing on to the activation function, an addition operation should be performed which is not represented in the images above for ease of illustration.

3 Consideration for Hardware Implementation

3.1 Parameters and Hyperparameters

Parameters in an ANN are quantities that are obtained from learning or training process. In our case, these parameters are weight values and co-efficient of the bias values. For training purpose, the technique that was adapted was *backpropagation*. In backpropagation, random values are assigned to the weights and coefficients of bias terms. Output is predicted and based on the shift of predicted value from actual output value, the weights are updated. This training is performed in workstation machine using *SKLearn* package in python.

The number of layers in the ANN, learning rate as well as the number of neurons in ANN are some of the *hyperparameters* in the design. For our demonstration purpose the number of neurons in each hidden layers were fixed at 4 while the number of hidden layers as decided as 2.

3.2 Numerical Consideration

Since the weights and input value as well as internal values and output produced by an ANN are floating point numbers, the operation in hardware involving floating point numbers are costly. Thus, all the number used for our computation in hardware are in fixed point number representation. Since DE2-115 FPGA board has embedded 9-bit multipliers, the bit length of fixed-point representation was taken as 9 with 4 fractional bits. This leaves 1 most significant position for sign bit and 4 next bits for whole number portion.

3.3 Activation Function

Different activation function presented in earlier section are Unit Step Function, Rectified Linear Unit (ReLU), and Sigmoid. The Unit Step Function is the simplest of all where the out is 1 for any positive input value while is 0 otherwise. ReLU function, however, outputs the input value as output for positive input values but outputs 0 for negative input values. Sigmoid function involves reciprocal of exponential function as is thereby complex in computation compared with Unit Step and ReLU function.

For our design, choice of ReLU is made. This is because ReLU is simple and easy to implement in hardware while also preserving the input information while having non-zero input.

4 Final Hardware Implementation

Once the trained model is obtained from workstation machine, System Verilog codes are generated based on the weights obtained. For demonstration purpose, an attempt on learning a complex mathematical function is done. Here, we took two functions and added them together such that each data points will have 2 input X-values and a single Y-value. Since Y-value is a continuous variable, a class separation was performed such that Y>0 is classified as Class 1 and Y<=0 is classified as Class 0.

This should map to any learning problem with two input variables and a single output class variable.

The codes for implementation of each module is presented as Section 5 Appendix.

4.1 Function 1

For the first equation, the value of X from -7 to 7 is taken with 0.01 increment. This is then passed onto **y1=sin(x^2+2x+3)** equation.

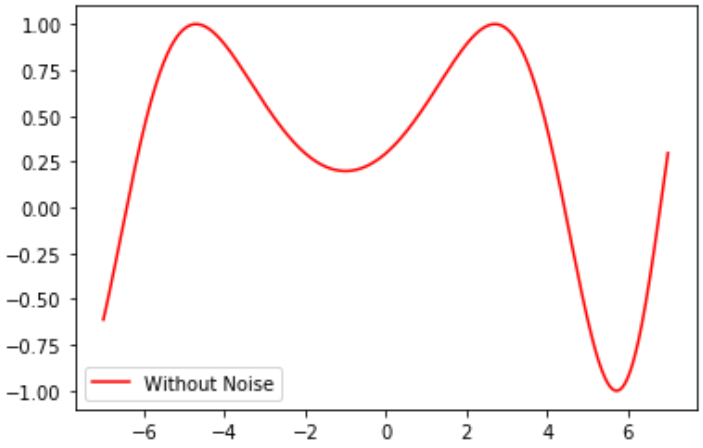

Figure 7 First. Equation

4.2 Function 2

For second equation, the value of X is taken from -7.01 to 6.99 with 0.01 increment. These X values are then passed to equation **y2=cos(x2+2x+3)**.

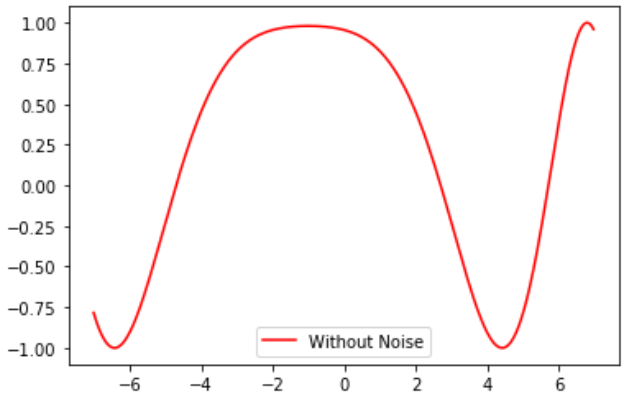

Figure 8 Second Equation

4.3 Addition of Function 1 and Function 2

For the final output, the values are added for y values such that **f(x1,x2) = y1+y2**.Then, finally they are categorized to two classes.

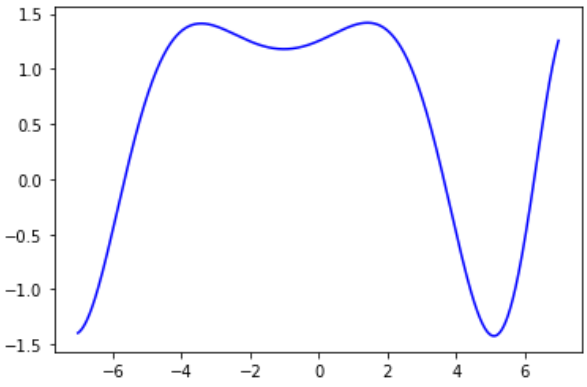

Figure 9 Combination of First and Second Equation

On the workstation machine, following shows the accuracy rates:

```
I: Train accuracy = 0.9809523809523809
I: Test accuracy = 0.9857142857142858
```

4.4 Compilation Report

Flow Status	Successful - Mon Dec 09 13:36:31 2019
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	DE2_115_Computer
Top-level Entity Name	DE2_115_Computer
Family	Cyclone IV E
Device	EP4CE115F29C8
Timing Models	Final
Total logic elements	36,778 / 114,480 (32 %)
Total registers	20996
Total pins	376 / 529 (71 %)
Total virtual pins	0
Total memory bits	296,714 / 3,981,312 (7 %)
Embedded Multiplier 9-bit elements	40 / 532 (8 %)
Total PLLs	3 / 4 (75 %)

Figure 10 Compilation Report from Quartus

For the simple problem statement presented above, the network thus formed is also small. Thus only 32% of the total logic elements are utilized. Since fixed point multiplication was employed, the embedded multipliers are used by the compilers. Of the 532 total available multipliers, 40 embeded multipliers are used here.

4.5 Final Run output

Following shows one sample run where the speed up observed is 395 times. Here, 5 input vectors are provided to the software as well as hardware implementation. Each of the decision-making process is again repeated 5 times for measurement.

```
JTAG UART link established using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Welcome to the NN machine in DE2-115 machine
--------------------------------------------
```

```
This code profiles time taken to run the software implementation vs hardware implementation
Software Implementation Follows
SW (0): For ip1=6.2800 and ip2=6.2700,
        predicted is 0.0000 (Class 0)
SW (1): For ip1=-5.6300 and ip2=-5.6400,
        predicted is 5.8396 (Class 1)
SW (2): For ip1=4.1400 and ip2=4.1300,
        predicted is 1.6928 (Class 1)
SW (3): For ip1=5.0000 and ip2=4.9900,
        predicted is 0.0000 (Class 0)
SW (4): For ip1=3.5000 and ip2=3.4000,
        predicted is 5.2628 (Class 1)
Software Implementation took 166776 clock cycles
HW(0): For ip1=6.2800(0x64) and ip2=6.2700(0x64),
        predicted is 0.0000(0x0) (Class 0)
HW(1): For ip1=-5.6300(0x15a) and ip2=-5.6400(0x15a),
        predicted is 5.2500(0x54) (Class 1)
HW(2): For ip1=4.1400(0x42) and ip2=4.1300(0x42),
        predicted is 1.9375(0x1f) (Class 1)
HW(3): For ip1=5.0000(0x50) and ip2=4.9900(0x4f),
        predicted is 0.0000(0x0) (Class 0)
HW(4): For ip1=3.5000(0x38) and ip2=3.4000(0x36),
        predicted is 5.3750(0x56) (Class 1)
Hardware Implementation took 420 clock cycles
Do you want to go again to measure?
```

In the above observation listed, we can see the classification made by both the Software Implementation and Hardware Implementation are same giving us an insight that the output produced are accurately classified. In the stub presented above, the speed up observed is

Clock Cycles in Software/Clock Cycles in Hardware	(166776)/(420) = 397.08

The same operation was repeated over 5 times for measurement. Following shows the speed up observed:

Clock Cycles in Software/Clock Cycles in Hardware	**833349)/(2110) = 395**

5 Appendix

5.1 Code for Avalon Interface

```
module nn_avalon_interface #(parameter W = 32, parameter FRACTION_WIDTH = 4, parameter BIT_WIDTH = 9,
        parameter NUM_LAYERS = 2, parameter num_input_units = 2, parameter num_output_units = 1,
        parameter num_neurons = 4)

        (input logic clock, reset, input logic[2:0] address, input logic[W - 1:0] writedata, output
logic[W - 1:0] readdata, input write, read, chipselect);

logic[BIT_WIDTH - 1:0] inputValue[num_input_units - 1:0];
logic[BIT_WIDTH - 1:0] outputPrediction[num_output_units - 1:0];

logic[W - 1:0] S_reg;

logic start, done;

assign start = S_reg[3];

myNeuralNet #(FRACTION_WIDTH, BIT_WIDTH, NUM_LAYERS, num_input_units, num_output_units, num_neurons)
myNN(clock, reset, start, done, inputValue, outputPrediction);

always@(posedge clock)
begin
if (reset)
begin
inputValue[0] = { BIT_WIDTH{1'b0}};
inputValue[1] = {BIT_WIDTH{1'b0}};
end
else if (chipselect == 1'b1 && write == 1'b1)
begin
        case(address)
                3'b000:
                        begin
                                inputValue[0] = writedata[BIT_WIDTH - 1:0];
                        end
                3'b001:
                        begin
                                inputValue[1] = writedata[BIT_WIDTH - 1:0];
                        end
                default:
                        begin
                                inputValue[0] = writedata[BIT_WIDTH - 1:0];
                        end
        endcase
end
end

always_comb
begin
        if (chipselect == 1'b1 && read == 1'b1)
        begin
                case(address)
                        3'b000: readdata[BIT_WIDTH-1:0] = inputValue[0];
                        3'b001: readdata[BIT_WIDTH-1:0] = inputValue[1];
                        3'b010: readdata[BIT_WIDTH-1:0] = outputPrediction[0];
                        default: readdata[BIT_WIDTH - 1:0] = outputPrediction[0];
                endcase
        end
        else
                readdata[BIT_WIDTH - 1:0] = outputPrediction[0];
end
endmodule
```

5.2 Code for Neural Network

```
module myNeuralNet #(parameter FRACTION_WIDTH = 4, parameter BIT_WIDTH = 9,
        parameter NUM_LAYERS = 2, parameter num_input_units = 2, parameter num_output_units = 1,
```

```
        parameter num_neurons = 4)
        (input logic clk, rst,
                input logic start,
                output logic done,
                input logic[BIT_WIDTH - 1:0] inputs[num_input_units - 1:0],
                output logic[BIT_WIDTH - 1:0] outputs[num_output_units - 1:0]);

logic[BIT_WIDTH - 1:0] weights_input_layer[num_input_units - 1:0][num_neurons - 1:0];
logic[BIT_WIDTH - 1:0] weights_output_layer[num_neurons - 1:0][num_output_units - 1:0];
logic[BIT_WIDTH - 1:0] weights_0[num_neurons - 1:0][num_neurons - 1:0];


logic[BIT_WIDTH - 1:0] inter_0[num_neurons - 1:0];
logic done_0;
logic[BIT_WIDTH - 1:0] inter_1[num_neurons - 1:0];
logic done_1;


assign weights_input_layer[0][0] = 9'b100000011;
assign weights_input_layer[0][1] = 9'b100010011;
assign weights_input_layer[0][2] = 9'b100000111;
assign weights_input_layer[0][3] = 9'b000000101;
assign weights_input_layer[1][0] = 9'b100000110;
assign weights_input_layer[1][1] = 9'b100000000;
assign weights_input_layer[1][2] = 9'b100000010;
assign weights_input_layer[1][3] = 9'b100010011;


assign weights_0[0][0] = 9'b000000000;
assign weights_0[0][1] = 9'b100000010;
assign weights_0[0][2] = 9'b000001001;
assign weights_0[0][3] = 9'b000000010;
assign weights_0[1][0] = 9'b000000000;
assign weights_0[1][1] = 9'b000010001;
assign weights_0[1][2] = 9'b100001100;
assign weights_0[1][3] = 9'b100001011;
assign weights_0[2][0] = 9'b000000000;
assign weights_0[2][1] = 9'b100000100;
assign weights_0[2][2] = 9'b000011101;
assign weights_0[2][3] = 9'b000101000;
assign weights_0[3][0] = 9'b000000000;
assign weights_0[3][1] = 9'b000010101;
assign weights_0[3][2] = 9'b100011001;
assign weights_0[3][3] = 9'b100011001;


assign weights_output_layer[0][0] = 9'b000000000;
assign weights_output_layer[1][0] = 9'b100011101;
assign weights_output_layer[2][0] = 9'b000011011;
assign weights_output_layer[3][0] = 9'b000100111;


singleLayerInput #(FRACTION_WIDTH, BIT_WIDTH, num_input_units, num_neurons)
sL_input(clk, rst, inputs, weights_input_layer, start, done_0, inter_0);

singleLayer #(FRACTION_WIDTH, BIT_WIDTH, num_neurons, num_neurons)
sL_1(clk, rst, inter_0, weights_0, start, done_1, inter_1);

singleLayerOutput #(FRACTION_WIDTH, BIT_WIDTH, num_neurons, num_output_units)
sL_output(clk, rst, inter_1, weights_output_layer, start, done, outputs);


endmodule
```

5.3 Code for Single Layer of NN

```
module singleLayer #(parameter FRACTION_WIDTH = 15, parameter BIT_WIDTH = 32,
        parameter inputSize = 5,
```

```
        parameter numNeurons = 5)
        (
                input logic clk, rst,
                input logic[BIT_WIDTH - 1:0] inputs[inputSize - 1:0],
                input logic[BIT_WIDTH - 1:0] weights[inputSize - 1:0][numNeurons - 1:0],
                input logic start,
                output logic done,
                output logic[BIT_WIDTH - 1:0] outputs[numNeurons - 1:0]
                );

//logic to indicate multiplication is done
logic done_mul;
assign done = done_mul;
//logic to hold intermediate values before activation
logic[BIT_WIDTH - 1:0] beforeActivation[numNeurons - 1:0];
logic[BIT_WIDTH - 1:0] outputsAfterBias[numNeurons - 1:0];
logic[BIT_WIDTH - 1:0] biasVal[numNeurons - 1:0];

assign biasVal[0] = 9'b100001101;
assign biasVal[1] = 9'b100010001;
assign biasVal[2] = 9'b000000000;
assign biasVal[3] = 9'b100000000;

//call matrix multiplication module
vecMatProd #(FRACTION_WIDTH, BIT_WIDTH, inputSize, numNeurons) vecMMul(
        .clk(clk),
        .start(start),
        .matA(inputs),
        .matB(weights),
        .result(beforeActivation),
        .done(done_mul)
);                      // let us call for activation here
genvar activation_var;
generate
for (activation_var = 0; activation_var < numNeurons; activation_var = activation_var + 1)
        begin: act1
        relu #(FRACTION_WIDTH, BIT_WIDTH) act1(outputsAfterBias[activation_var],
outputs[activation_var]);
qadd #(FRACTION_WIDTH, BIT_WIDTH) qa0(
        .a(biasVal[activation_var]),
        .b(beforeActivation[activation_var]),
        .c(outputsAfterBias[activation_var])
);
end
endgenerate

endmodule
```

5.4 Code for Vector Matrix Product

```
module vecMatProd #(parameter FRACTION_WIDTH = 15, parameter BIT_WIDTH = 32,
        parameter NUM_COL_VEC = 5, parameter NUM_COL_MAT = 5)
        (
                input logic clk, start,
                input logic[BIT_WIDTH - 1:0] matA[NUM_COL_VEC - 1:0],
                input logic[BIT_WIDTH - 1:0] matB[NUM_COL_VEC - 1:0][NUM_COL_MAT - 1:0],
                output logic[BIT_WIDTH - 1:0] result[NUM_COL_MAT - 1:0],
                output logic done
                );

//check internal done signals
logic[NUM_COL_MAT - 1:0] done_check;
logic[BIT_WIDTH - 1:0] matB_in[NUM_COL_MAT - 1:0][NUM_COL_VEC - 1:0];

assign done = (&done_check) ? 1'b1:1'b0;

// the matrix B to send to multiply is essentially transpose of the input matrix matB
transpose #(FRACTION_WIDTH, BIT_WIDTH, NUM_COL_VEC, NUM_COL_MAT) transp(
        .inMat(matB),
```

```
        .outMat(matB_in)
);

genvar i, j;
// we will lay out a collection of dot products since the result of matrix multiplication is a
collection of dot products
generate
for (j = 0; j < NUM_COL_MAT; j = j + 1) begin: col_mat
        dotproduct #(FRACTION_WIDTH, BIT_WIDTH, NUM_COL_VEC) dp(
                .clk(clk),
                .start_dot(start),
                .a_vec(matA),
                .b_vec(matB_in[j]),
                .result(result[j]),
                .done(done_check[j])
        );
end
endgenerate

endmodule
```

5.5 Code for Dot Product

```
module dotproduct #(parameter FRACTION_WIDTH = 15, parameter BIT_WIDTH = 32, parameter VECTOR_SIZE =
10) //Parameterized values
(
        input logic clk,
        input logic start_dot,
        input logic[BIT_WIDTH - 1:0] a_vec[VECTOR_SIZE - 1:0],
        input logic[BIT_WIDTH - 1:0] b_vec[VECTOR_SIZE - 1:0],
        output logic[BIT_WIDTH - 1:0] result,
        output logic done
        );
//following will hold the element wise multiplication result of the two vectors
logic[BIT_WIDTH - 1:0] products[VECTOR_SIZE - 1:0];
//following will come from prodOfVectors that will indicate the elementwise product completion
logic prod_complete;
//following will hold the done signal indicator
logic sum_complete;
//following will hold the final result value
logic[BIT_WIDTH - 1:0] sum_val;
//following will count the number of times addition was performed
integer Count;

//instantiate element wise product of vectors here
prodof2Vectors #(FRACTION_WIDTH, BIT_WIDTH, VECTOR_SIZE) pod(
        .clk(clk),
        .startMult(start_dot),
        .a_vec(a_vec),
        .b_vec(b_vec),
        .result(products),
        .done(prod_complete)
);

sumOfVectorElements #(FRACTION_WIDTH, BIT_WIDTH, VECTOR_SIZE) sov(
        .clk(clk),
        .start(prod_complete),
        .a_vec(products),
        .result(result),
        .done(sum_complete)
);
assign done = sum_complete;

endmodule
```

5.6 Code for Element wise Product of two vectors

```
module prodof2Vectors #(parameter FRACTION_WIDTH = 15, parameter BIT_WIDTH = 32, parameter VECTOR_SIZE
= 10)
(input logic clk, startMult,
```

```
        input[BIT_WIDTH - 1:0] a_vec[VECTOR_SIZE - 1:0],
        input[BIT_WIDTH - 1:0] b_vec[VECTOR_SIZE - 1:0],
        output[BIT_WIDTH - 1:0] result[VECTOR_SIZE - 1:0],
        output logic done
        );

logic[VECTOR_SIZE - 1:0] is_complete;
logic[VECTOR_SIZE - 1:0] is_overflow;


`ifdef QMULTS

assign done = (&is_complete) ? 1'b1:1'b0;
//create parallel logic so that all the multiplication is performed at once
// .. this will be true since there is no dependency between the results
genvar gi;
generate
for (gi = 0; gi < VECTOR_SIZE; gi = gi + 1) begin: mqs
        qmults #(FRACTION_WIDTH, BIT_WIDTH) qs(
                .i_multiplicand(a_vec[gi]),
                .i_multiplier(b_vec[gi]),
                .i_start(startMult),
                .i_clk(clk),
                .o_result_out(result[gi]),
                .o_complete(is_complete[gi]),
                .o_overflow(is_overflow[gi])
        );
end
endgenerate
`else
//create parallel logic so that all the multiplication is performed at once
// .. this will be true since there is no dependency between the results
genvar gi;
generate
for (gi = 0; gi < VECTOR_SIZE; gi = gi + 1) begin: mq
        qmult #(FRACTION_WIDTH, BIT_WIDTH) q(
                .i_multiplicand(a_vec[gi]),
                .i_multiplier(b_vec[gi]),
                .o_result(result[gi]),
                .ovr(is_overflow[gi])
        );
end
endgenerate

assign done = 1'b1;
`endif
endmodule
```

5.7 Code for Sum of all elements in a vector

```
module sumOfVectorElements #(parameter FRACTION_WIDTH = 15, parameter BIT_WIDTH = 32, parameter
VECTOR_SIZE = 10)
(input logic clk, start,
        input[BIT_WIDTH - 1:0] a_vec[VECTOR_SIZE - 1:0],
        output[BIT_WIDTH - 1:0] result,
        output logic done
        );

logic[BIT_WIDTH - 1:0] sum[VECTOR_SIZE - 2:0];
integer count, next_count;

qadd #(FRACTION_WIDTH, BIT_WIDTH) qa0(
        .a(a_vec[0]),
        .b(a_vec[1]),
        .c(sum[0])
);

genvar gi;
generate
```

```
for (gi = 2; gi < VECTOR_SIZE - 1; gi = gi + 1) begin: sv
        qadd #(FRACTION_WIDTH, BIT_WIDTH) qa(.a(sum[gi - 2]),
                .b(a_vec[gi]),
                .c(sum[gi - 1])
        );
end
endgenerate

qadd #(FRACTION_WIDTH, BIT_WIDTH) qah(
        .a(a_vec[VECTOR_SIZE - 1]),
        .b(sum[VECTOR_SIZE - 3]),
        .c(sum[VECTOR_SIZE - 2])
);

//let us have a logic to count and predict when the sum if found out
always@(posedge clk)
begin
if (start)
begin
if (done == 1'b1)
        begin
        next_count <= 0;
        done <= 1'b0;
        end
else if (count < VECTOR_SIZE)
        next_count = count + 1;
else
done <= 1'b1;
end
end

always_comb
begin
count <= next_count;
end

assign result = sum[VECTOR_SIZE - 2];

endmodule
```

5.8 Code for NIOS Interface

```
/*
File: NIOS interface.c
This file implements a simple NN in C in nios and compares the time it takes for the NIOS C with
the peripheral implemented.
*/
// mandatory include statement
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// base address for the NN accelerator in DE2-115 computer machine
#define nn_base ((volatile unsigned int*) 0x08200020)
#define nn_input_1 ((volatile unsigned int*) 0x08200020)
#define nn_input_2 ((volatile unsigned int*) 0x08200024)
#define nn_output ((volatile unsigned int*) 0x08200028)

// base address of the interval timer in DE2-115 machine
#define interval_timer_ptr ((volatile unsigned int *) 0xff202000)
// number of iterations to be done for profile operation
// .. final time is average of all the times measured
#define NUM_PROFILE 5
#define NUM_EXAMPLES 5
#define NUM_BITS 9
#define FRACTION_BITS 4


float input_array1[] = { 6.28,-5.63,4.14,5.0,3.5 };
```

```
float input_array2[] = { 6.27,-5.64,4.13,4.99,3.4 };

float weight_array0[] = { -0.1936, -1.2167, -0.4389, 0.3301, -0.4192, -0.0312, -0.1288, -1.2069 };
float weight_array1[] = { 0.0000, -0.1380, 0.6141, 0.1625, 0.0000, 1.0809, -0.7948, -0.7030, -0.0000, -
0.2572, 1.8539, 2.5529, -0.0000, 1.3271, -1.5748, -1.5998 };
float weight_array2[] = { 0.0000,-1.8357,1.7187,2.4818 };

float bias0[] = { 1.09838152, -1.49831044,  2.72732628, -1.31490558 };
float bias1[] = { -0.831537276, -1.09912261,  0.000301989131 ,-0.000242028198 };
float bias2[] = { -1.90968533 };

unsigned int input_units = 2;
unsigned int neurons = 4;
unsigned int layers = 2;
unsigned int output_units = 1;

void matMul(float* a_mat[], float* b_mat[], float* output_mat[], unsigned int d1, unsigned int d2,
unsigned int d3)
{
        float sum = 0;
        for (unsigned int i = 0; i < d1; i++)
        {
                for (unsigned int j = 0; j < d3; j++)
                {
                        sum = 0;
                        for (unsigned int k = 0; k < d2; k++)
                        {
                                sum += a_mat[i][k] * b_mat[k][j];
                        }
                        output_mat[i][j] = sum;
                }
        }
}
void relu(float* a_mat[], unsigned int d1, unsigned int d2)
{
        for (unsigned int i = 0; i < d1; i++)
        {
                for (unsigned int j = 0; j < d2; j++)
                {
                        if (a_mat[i][j] < 0)
                                a_mat[i][j] = 0;
                }
        }
}

void biasAdd(float* a_mat[], float* bias, unsigned int d1, unsigned int d2)
{
        for (unsigned int i = 0; i < d1; i++)
        {
                for (unsigned int j = 0; j < d2; j++)
                {
                        a_mat[i][j] += bias[j];
                }
        }
}

unsigned long software_time = 0;
unsigned long hardware_time = 0;

void myNNSoftware()
{
        // create space for input values
        float **input_vals = (float**)malloc(1 * sizeof(float*));
        for (unsigned int i = 0; i < 1; i++)
        {
                input_vals[i] = (float*)malloc(input_units * sizeof(float));
        }
        // create weights parameter for Input-H1
        float **w0 = (float**)malloc(input_units * sizeof(float*));
```

```
	for (unsigned int i = 0; i < input_units; i++)
	{
		w0[i] = (float*)malloc(neurons * sizeof(float));
	}
	// assign weights here: Input-H1
	for (unsigned int i = 0; i < input_units; i++)
	{
		for (unsigned int j = 0; j < neurons; j++)
			w0[i][j] = weight_array0[i*neurons + j];
	}
	// create space for intermediate values: Input to H1
	float **l1_outs = (float**)malloc(1 * sizeof(float*));
	for (unsigned int i = 0; i < 1; i++)
	{
		l1_outs[0] = (float*)malloc(neurons * sizeof(float));
	}


	// create weights here for H1-H2
	float **w1 = (float**)malloc(neurons * sizeof(float*));
	for (unsigned int i = 0; i < neurons; i++)
	{
		w1[i] = (float*)malloc(neurons * sizeof(float));
	}
	// assign weights here: H1-H2
	for (unsigned int i = 0; i < neurons; i++)
	{
		for (unsigned int j = 0; j < neurons; j++)
			w1[i][j] = weight_array1[i*neurons + j];
	}
	// create space for intermediate values: Output of H1 input of H2
	float **l2_outs = (float**)malloc(1 * sizeof(float*));
	for (unsigned int i = 0; i < 1; i++)
	{
		l2_outs[0] = (float*)malloc(neurons * sizeof(float));
	}

	// create weights here: H2-Output
	float **w2 = (float**)malloc(neurons * sizeof(float*));
	for (unsigned int i = 0; i < neurons; i++)
	{
		w2[i] = (float*)malloc(output_units * sizeof(float));
	}
	// assign weights here: H2:Op
	for (unsigned int i = 0; i < neurons; i++)
	{
		for (unsigned int j = 0; j < output_units; j++)
			w2[i][j] = weight_array2[i*output_units + j];
	}
	// create space for intermediate values: Output of H2
	float **l3_outs = (float**)malloc(1 * sizeof(float*));
	for (unsigned int i = 0; i < 1; i++)
	{
		l3_outs[0] = (float*)malloc(output_units * sizeof(float));
	}

	// we will count the number of cc here
	software_time = 0;
	for (unsigned int times_ = 0; times_ < NUM_PROFILE; times_++)
	{
		for (unsigned int iter = 0; iter < NUM_EXAMPLES; iter++)
		{
			//.. since the timer will count downwards, let us give maximum 32-bit value
			*(interval_timer_ptr + 2) = 0xffff;
			*(interval_timer_ptr + 3) = 0xffff;
			// continuous mode and start counting
			*(interval_timer_ptr + 1) = 0x06;

			input_vals[0][0] = input_array1[iter];
```

```
                        input_vals[0][1] = input_array2[iter];

                        // following implements the layer between the input and H1
                        matMul(input_vals, w0, l1_outs, 1, input_units, neurons);
                        biasAdd(l1_outs, bias0, 1, neurons);
                        relu(l1_outs, 1, neurons);

                        // following implements the layer between the H1 and H2
                        matMul(l1_outs, w1, l2_outs, 1, neurons, neurons);
                        biasAdd(l2_outs, bias1, 1, neurons);
                        relu(l2_outs, 1, neurons);

                        // following implements the layer between the H2 and output
                        matMul(l2_outs, w2, l3_outs, 1, neurons, output_units);
                        biasAdd(l3_outs, bias2, 1, output_units);
                        relu(l3_outs, 1, output_units);

                        //once the opration is done, we will grab the snapshot value
                        *(interval_timer_ptr + 4) = 1;
                        //let us stop it
                        *(interval_timer_ptr + 1) = 0x08;
                        software_time += (4294967295 - (*(interval_timer_ptr + 5)) * 65536 -
*(interval_timer_ptr + 4));

                        printf("SW (%d): For ip1=%4.4f and ip2=%4.4f, \n\tpredicted is %4.4f (Class
%d)\n", iter, input_array1[iter], input_array2[iter], l3_outs[0][0], l3_outs[0][0] > 0 ? 1 : 0);
                }
        }
        // freeing stuffs
        for (unsigned int i = 0; i < 1; i++)
        {
                free(input_vals[i]);
        }
        free(input_vals);
        for (unsigned int i = 0; i < input_units; i++)
        {
                free(w0[i]);
        }
        free(w0);
        free(l1_outs[0]);
        free(l1_outs);
        free(l2_outs[0]);
        free(l2_outs);
        free(l3_outs[0]);
        free(l3_outs);
        for (unsigned int i = 0; i < neurons; i++)
        {
                free(w1[i]);
        }
        free(w1);
        for (unsigned int i = 0; i < neurons; i++)
        {
                free(w2[i]);
        }
        free(w2);
}
// this function converts the input float to fixed point representation
unsigned int convertFixedPoint(float myIn)
{
        unsigned int retVal = 0;
        float fractPart = 0;
        if (myIn < 0)
        {
                retVal = 0x01;
        }
        myIn = fabsf(myIn);

        fractPart = myIn - ((long)myIn);
```

```
        // let us first deal with the whole number part
        for (int bitCount = NUM_BITS - FRACTION_BITS - 1 - 1; bitCount >= 0; bitCount--)
        {
                //shift one position to the left
                retVal <<= 1;
                if (myIn >= (0x01 << bitCount))
                {
                        retVal |= 0x01;
                        myIn -= 0x01 << bitCount;
                }
        }
        float div = 0.5;
        // for the fractional part
        for (unsigned int bitCount = FRACTION_BITS; bitCount > 0; bitCount--)
        {
                //shift one position to the left
                retVal <<= 1;
                if (fractPart >= (div))
                {
                        retVal |= 0x01;
                        fractPart -= div;
                }
                div /= 2;
        }
        return retVal;
}

float convertToFloat(unsigned int inVal)
{
        float retVal = 0;

        float mul = 0.5;
        for (int i = FRACTION_BITS - 1; i >= 0; i--)
        {
                if (inVal&(1 << i))
                {
                        retVal += mul;
                }
                mul /= 2;
        }
        for (unsigned int i = FRACTION_BITS; i < NUM_BITS - 1; i++)
        {
                if (inVal&(1 << i))
                {
                        retVal += (1 << (i - FRACTION_BITS));
                }
        }
        if (inVal&(1 << (NUM_BITS - 1)))
                retVal = 0 - retVal;

        return retVal;
}
void myNNHardware()
{
        // we will count the number of cc here
        hardware_time = 0;
        unsigned int myOut = 5;
        unsigned int myIn1 = 0;
        unsigned int myIn2 = 0;
        for (unsigned int times_ = 0; times_ < NUM_PROFILE; times_++)
        {
                for (unsigned int iter = 0; iter < NUM_EXAMPLES; iter++)
                {
                        myIn1 = convertFixedPoint(input_array1[iter]);
                        myIn2 = convertFixedPoint(input_array2[iter]);
                        //.. since the timer will count downwards, let us give maximum 32-bit value
                        *(interval_timer_ptr + 2) = 0xffff;
                        *(interval_timer_ptr + 3) = 0xffff;
                        // continuous mode and start counting
```

```
                        *(interval_timer_ptr + 1) = 0x06;

                        //convert to fixed point representation
                        *nn_input_1 = myIn1;
                        *nn_input_2 = myIn2;

                        // a simple delay
                        // for(volatile unsigned int i=0;i<2;i++);

                        myOut = *nn_output;

                        //once the opration is done, we will grab the snapshot value
                        *(interval_timer_ptr + 4) = 1;
                        //let us stop it
                        *(interval_timer_ptr + 1) = 0x08;
                        hardware_time += (4294967295 - (*(interval_timer_ptr + 5)) * 65536 -
*(interval_timer_ptr + 4));

                        printf("HW(%d): For ip1=%4.4f(0x%x) and ip2=%4.4f(0x%x), \n\tpredicted is
%4.4f(0x%x) (Class %d)\n", iter, input_array1[iter], *nn_input_1, input_array2[iter], *nn_input_2,
convertToFloat(myOut), myOut, myOut > 0 ? 1 : 0);
                }
        }
}
int main(void)
{
        printf("Welcome to the NN machine in DE2-115 machine\n");
        printf("-----------------------------------------------\n");

        char userIp = 'N';
        while (1)
        {
                printf("This code profiles time taken to run the software implementation vs hardware
implementation\n");
                printf("Software Implementation Follows\n");
                software_time = 0;
                myNNSoftware();
                printf("Software Implementation took %lu clock cycles\n", software_time);

                hardware_time = 0;
                myNNHardware();
                printf("Hardware Implementation took %lu clock cycles\n", hardware_time);

                printf("Do you want to go again to measure?\n");
                scanf("%c", &userIp);
                if (userIp != 'Y' && userIp != 'y')
                        break;
        }

        printf("Thank you for using the NN accelerator\n");
        return 0;
}
```