COMPARISON OF THE TALLY NUMBERING SYSTEM TO TRADITIONAL ARITHMETIC SYSTEMS IN FIELD PROGRAMMABLE GATE ARRAYS

By

ROBERT PAUL SHREDOW

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

EASTERN WASHINGTON UNIVERSITY
Department of Computer Science

APRIL 2020

To the Faculty of Eastern Washington University:

The members of the Committee appointed to examine the thesis of ROBERT PAUL SHREDOW find it satisfactory and recommend that it be accepted.

Kosuke Imamura, Ph.D., Chair

Paul Schimpf, Ph.D.

Awlad Hossain, Ph.D.

ACKNOWLEDGMENT

I would like to thank Dr. Kosuke Imamura. Who suggested this topic of research and aided me with my thesis and research. I would also like to thank Joseph Dumoulin adjunct professor from NextIT who teaches machine learning for helping me with of my questions and helped with ideas, suggestions and research papers.

I would also like to thank Dr. Paul Schmipf and Dr. Awlad Hossain for the time they dedicated to reviewing my work and for serving as my second and third committee members, respectively.

COMPARISON OF THE TALLY NUMBERING SYSTEM TO

TRADITIONAL ARITHMETIC SYSTEMS

IN FIELD PROGRAMMABLE GATE ARRAYS

Abstract

by Robert Paul Shredow, MS.
Eastern Washington University
April 2020

This research explores the use of heterogeneous computing platforms for use in machine learning as well as different neural network architectures. These platforms and architectures can be used to accelerate the complex operations that are required for machine learning, more specifically neural networks. The use of different architectures, implementing different types of numbering and mathematics systems is explored in hopes of accelerating mathematical functions. The heterogeneous computing platform explored in this thesis is a Field Programmable Gate Arrays (FPGA), specifically a SoC/FPGA which is a ARM CPU and a FPGA in the same chip. FPGAs are unique because they are low power, high customizable hardware with bit level control. A new numbering system, called Tally, can be simulated in software but only implemented in hardware with a FPGA, without time consuming and expensive ASIC (application specific integrated circuit) being designed. This thesis explores two different types of neural networks are explored the first a simple XOR (exclusive OR) gate neural network tested in the Tally system, 16-bit fixed point and 32-bit floating point. The MNIST (handwritten numbers) dataset is used with a pre-trained multi-layer perceptron with both 16-bit Fixed Point and 32-bit Floating Point numbers. This study will act as a preliminary exploration of the Tally System and an exercise in learning implementation of neural networks in hardware.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter One

INTRODUCTION

1.1 Motivation

With artificial intelligence, more specifically deep neural networks a branch of machine learning becoming more ubiquitous. This trend is not slowing down anytime soon but accelerating. Traditional general computing platforms that perform sequential computations, even if they may use multiple cores/threads, while good for prototyping are too slow for practical use with traditional methods of computation. Luckily there are already heterogeneous computing platforms that already exist to help alleviate this problem. In the field of machine learning, deep neural networks have become very popular. Deep neural networks take input values, perform math operations, mainly matrix multiplication, where the values are multiplied by weights and biases. Matrix multiplication is a good example of "Single Instruction Multiple Data" process where different data is processed through the same process at the same time. While some CPUs may be optimized for SIMD instructions and have multiple cores/threads they still fall short when it comes to machine learning operations. It is shown later on in this thesis there are methods of performing math operations faster using different number and math types, on different platforms.

1.2 Introduction to the Terasic DE-10 Nano SoC/FPGA

This thesis will focus on a neural network accelerated by an FPGA that is on a System on a Chip (SoC). In this instance, a System on a Chip is an 800 MHz Dual Core ARM Cortex-A9 processor attached to an Intel Cyclone V FPGA through an AXI (Advanced eXtensible Interface) bus. The system is connected to 1 gigabyte of memory. The FPGA has 115k logic elements, one hundred twelve 19 bit multipliers, and 5,570 Kbits of block memory, with various other I/O. The ARM processor part is known as and will be referred to as the Hard Processor System (HPS) as referred to by Intel. Terasic provided a version of Linux that runs the he HPS part of the SoC/FPGA that has a desktop user interface but most development is done through secure shell (SSH). The board comes with I/O common to FPGA development boards such as two 2x20 general purpose I/O, an Arduino header, 8 LEDs, 4 slide switches, 3 push buttons, a 10 pin analog input, an analog to digital converter, and a 4 pin SPI interface. The HPS part comes with 1 gigabit ethernet port, a USB OTG port, a USB micro-AB connector, a micro SD slot, an accelerometer, UART to USB and various other I/O. (Terasic, 2017)

1.3 Introduction to the Tally System

This thesis will not only focus on hardware but on different number representations in binary and their relation with neural networks. The numbering systems chosen are 16 bit fixed point and 32 bit floating point, the reason those were chosen are given later, and as well as exploring the capabilities of a new number representation called the Tally Numbering System. The Tally System uses the 1's in a bit string to represent a number, typically a 1 in a bitstring will represent a 1 in decimal. For example, 00011111 = 5 because there are five 1s. This is also referred to as the Hamming Weight and in this case is also called the population count, popcount, sideways sum or bit summation. Mathematical functions

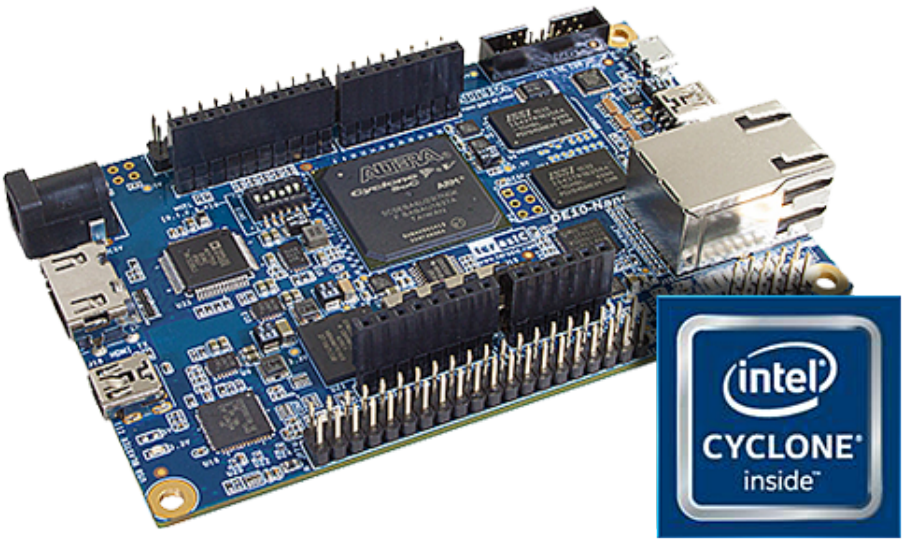


Figure 1.1 Terasic DE10-Nano SoC/FPGA development board with Cyclone V FPGA

such as add are done differently from fixed and floating point and are optimized for speed first and resource usage second. Addition is done by bit shifting right, subtraction is done by bit shifting left. Multiplication and division is done by reflecting a bit string through a matrix with slope of the multiplier or divisor. Uncommon mathematics functions such as sigmoid can be precomputed and done with a lookup table. The Tally Numbering System will be explained in detail later on in thesis as well as benefits and drawbacks. It will also be compared to other more traditional systems such as 16-bit fixed point, 32-bit floating point as well as more experimental systems.

Chapter Two

BACKGROUND

In this section we will overview different computing architectures for deep neural networks and how they perform math functions for neural networks. Graphics Processing Units are designed to do matrix operations because graphics processors need to perform vector/matrix math very quickly for graphics operations which involves a lot matrix operations to emulate a 3D world. GPUs have hundreds or thousands of processing cores to perform matrix math very quickly usually in 32-bit floating point format. General Purpose Graphics Processors have been created to leverage its highly parallel nature into areas such as machine learning, scientific computing and hashing algorithms. GPU companies such as Nvidia and AMD have released software to program GPUs such as CUDA from Nvidia and OpenCL from AMD. Nvidia has even created a "Tensorcore" that does mixed precision calculations for deep learning.

The Nvidia TensorCore and Google's Tensor Processing Unit are created to do matrix multiply operations for neural networks exclusively. A Tensor Processing Unit is an Application Specific Integrated Circuit that is built for high volume low precision, as little as 8 bits computation. A single TPU can process as much as 100 million photos a day. But the TPU is only usable through Google Cloud. (*An in-depth look at Google's first Tensor Processing Unit (TPU)* n.d.) Nvidia's Tensor Core is capable of mixed precision multiply accumulates, (multiply then add) of 16 and 32 bit floating point and 4 and 8 bit integer math. Nvidia's

Volta has 640 Tensor Cores, each performing 64 floating point fused multiply add operations per clock cycle that can deliver up to 125 teraflops (1 TeraFLOP = 1 trillion floating point operations per second). (*Tensor Cores in NVIDIA Volta GPU Architecture* n.d.) Other architectures that can be used are computer clusters which are servers networked together with CPUs that have a few dozen cores. An example of software that does machine learning using compute clusters is Apache Spark. (*Machine Learning Library (MLlib) Guide* n.d.)

Another compute platform that has been used for speeding up other calculations such as the Fast Fourier Transform used in Digital Signal Processing are Field Programmable Gate Arrays (FPGA). An FPGA is a reconfigurable integrated circuit that can be designed to implement any type of logic so long as it has the hardware to do so. The core of an FPGA are combinational logic blocks which typically consist of Look Up Tables (LUTs), Full Adders and Flip-Flops. FPGAs also contain memory that can be as simple as flip flops to entire blocks of random access memory (RAM). All of this is attached by a set of interconnects based on the logic the user wants to implement. Many FPGAs have multipliers for use with Digital Signal Processing. The benefits of using FPGAs for neural networks and matrix multiplication are that the architecture is completely customizable. Registers and data bus widths can be customized to fit the application. FPGA allows for a high amount of customization when creating a pipeline for a neural network. A disadvantage is that high customization requires use of Hardware Description Languages such as Verilog and VHDL which results in long development times. This problem can be minimized by using OpenCL.

OpenCL first developed by Apple, Inc is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs, and other types of processors. OpenCL includes a language for developing kernels (functions that execute on hardware devices) as well as application programming interfaces (APIs) that define and control the various platforms. OpenCL allows for parallel computing using task-based and data-based parallelism. (Moore, 2017)

Most of these devices are in data centers with access to almost unlimited amounts of

power and cooling. Training and inference can be done with high precision floating point numbers on devices that are not resource constrained. The purpose of this thesis is to develop methods of applying neural networks for inference on resource constrained devices such as small processors, FPGAs and even microprocessors on mobile and Internet of Things devices (IoT).

Chapter Three

RELATED WORK

Extensive research exists on the subject of trying to increase performance of neural networks in hardware. Most of which involve manipulating the weights and other neural network variables. There are several different approaches to making neural networks perform better in hardware, the most common of which is using different bit precision in both floating point, fixed point and integer numbers to perform MAC (multiply accumulate) operations faster, decrease memory bandwidth etc. Another similar approach is training a neural network in full precision 32-bit floating point and using Quantization to convert those weights produced to a number that is smaller and easier to perform calculations on such as 16-bit fixed point. This approach is common in FPGA neural networks and referenced in a paper in a collection of papers called, "FPGA Implementations of Neural Networks." (Omondi, 2006) Another approach that closest fits with the goals of the Tally System are Binary and Ternary Neural Networks. Binary neural networks reduce weights to 1 and -1 and ternary neural networks reduce weights to 1, 0 and -1 and use 1-bit XNOR (exclusive not OR) operations to replace expensive multiply operations and bit shifts for add and subtract operations. Other ways to reduce computation that will not be elaborated on because it does not apply directly to this thesis and would affect different types of neural network architecture in much the same way. The first is reducing the amount of parameters or weights. There are several ways to do this, one way is pruning or removing connections between neurons that are deemed insignificant,

the real effect of this is reducing a weight to 0. This is taken from biological systems in which synapses in mammals decay as they get older. Another method is to reduce the amount of weights is to use low rank approximation to compress weight matrices by reducing the rank and thereby reducing the amount of weights needed to be stored and calculated.

3.1 Quantization

Quantization is the process of constraining an input from a continuous or otherwise large set of values (such as the real numbers) to a discrete set (such as the integers). An example, instead of having weights be any real number between 0 and 1 we can constrain them to every 0.1, so values can only be 0, 0.1, 0.2 ... 0.9, 1.0. This gives 11 numbers to deal with instead of infinity. Quantization is important to neural networks because it reduces computational demand and increases power efficiency. These are caused by increased computation efficiency and reduced amount of memory accesses. Using the lower-bit quantized data requires less data movement, both on-chip and off-chip, which reduces memory bandwidth and saves significant energy. Lower-precision mathematical operations, such as an 8-bit integer multiply versus a 32-bit floating point multiply, consume less energy and increase compute efficiency, thus reducing power consumption. In addition, reducing the number of bits for representing the neural network's parameters results in less memory storage. While an obvious effect of using quantization is a decrease in accuracy there are ways of preparing the network for quantization during the training phase. (Louizos, Reisser, and Blankevoort, 2018) Quantization is necessary with the Tally System because if a weight had a large amount of decimal precision the amount of bits necessary to represent the number would be large. For example, representing the the number 0.1284 without quantization would require a 10,000 bit register and an even larger register to add or multiply with another number of the same resolution. Later it is shown that the 16-bit fixed point neural network for the MNIST data set is Quantized from converting from floating point weights to integer that represents a fixed point

number.

3.2 Binarized Neural Networks

Research in Binarized Neural Networks (BNNs) is the closest research I have found that aligns with the goals of the Tally System. A Binarized Neural Network is a neural network where the weights and activations are binarized to either 1 or -1. The obvious advantage of this is that storing a single bit takes up much less memory and memory bandwidth than 32-bit floating point numbers and less arithmetic than conventional Deep Neural Networks. MACs in BNNs are also much more computationally cheaper than floating point multiply accumulates. Instead of multiply, BNNs do a bit-wise XNOR and bit shift to accumulate. The activation function is just a binarization of the result of the MAC i.e. 1 if the number is positive and -1 if the number is negative. A paper shows it is possible to train two BNNs on the MNIST, CIFAR-10 and SVHN, and achieve nearly state of the art results. (Courbariaux, Hubara, and Bengio, 2016). The authors have found ways to incorporate binary weights in conjunction with high precision weight necessary for gradient descent to train neural networks whereas other neural networks use just conventional techniques. BNNs are closest to the Tally System because it attempts to use computationally cheap logic gates and bit shifts instead of expensive multiply, accumulate and activation functions logistic sigmoid and hyperbolic tangent. Such a system seemingly would be ideal for FPGAs that can operate on the bit level whereas the authors of the paper used GPUs which contain cores that operate in 32-bit floating point arithmetic although they made their own updated kernel that increased performance 7 times over the normal kernel.

Chapter Four

METHODOLOGY

A XOR (exclusive OR) neural network was chosen to compare the performance of the Tally numbering system to 32 bit floating point and 16 bit fixed point. It was chosen because the range of the weights are much smaller than that of the MNIST multi-layer perceptron. The logic of a XOR gate with two inputs is if both inputs are low then the output is low, also if both inputs are high then the output is low, the only time the output is high is if one input is high and the other is low. The logic table is shown in Figure 4.1. The XOR neural network has an input layer, a hidden layer and a output layer. The input layer has two inputs, the hidden layer has two nodes and there is one output, 1 or 0 as shown in Figure 4.2.

The MNIST dataset is a famous dataset of handwritten numbers commonly used in machine learning. The dataset has 60,000 images of hand written numbers, 60,000 for the training set and 10,000 for the testing set. Each number is a 28 by 28 pixel 8-bit gray scale image.

As mentioned, the XOR Neural Network was chosen because the weights vary a lot less then the MNIST weights. In Figure 4.2, the weights are either 20 or -20 and the biases are are 30, -30 or -10. Since the Tally System is signed magnitude so any positive number can also be negative. The MNIST Neural Network has weights that vary quite a bit. Analyzing just the hidden weights of the MNIST Neural network the average of the weights is -0.00161099, the standard deviation is 0.1026613, the max value is 0.871943882, and the variance is

0.010539343. This shows that the average weight value is very low, 1/1000th, the variance shows that the weights do not vary that much from each other and the standard deviation shows that 65% of weights between 0.1 and -0.1. To adequately represent MNIST weights in the Tally System each bit would need to represent at least 0.001 which may not even be able to represent weights accurately. To represent weights up to 1 then the register sizes would need to be at least 1000 bits. To represent weights up to the max weight which is 0.871943882 using bits that represent 0.001 you would need registers up to 872 bits. To add two 1000 bit numbers you would need a register 2000 bits long and to multiply two 1000 bit registers you would need a 1,000,000 bit register. This is clearly impractical and takes up too much memory and resources. This is one of the drawbacks of the Tally System, a 32-bit floating point number can represent $\pm(2 - 2^{23}) * 2^{127} = \pm 3.4028235 * 10^{38}$ values and 16-bit fixed point can represent $\pm 2^{16} = \pm 65,636$ values whereas a 32-bit or 16-bit Tally number can only represent 32 or 16 values respectively. This problem may be able to be minimized by training discretized neural networks as mentioned in the related works section and will be elaborated upon in the future works section.

INPUTS		OUTPUTS
A	B	Y = A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4.1 Truth Table for a 2-Input Exclustive OR Gate

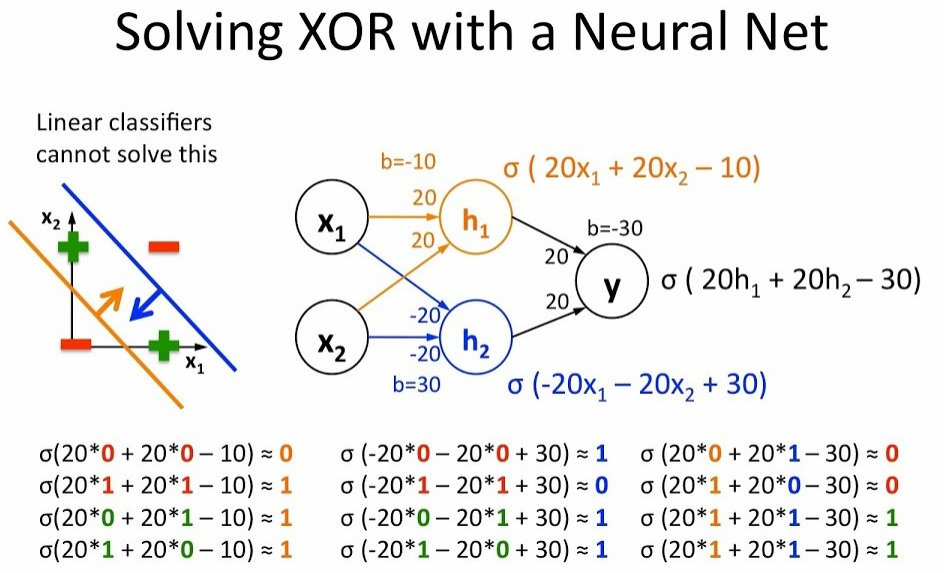


Figure 4.2 XOR Gate Neural Network with a hidden layer with two nodes. (Lavrenko, 2015)

4.1 Implementation of 32-bit Floating Point XOR Neural Network

The 32-bit floating point XOR neural network takes two floating point numbers from the Hard Processor System (HPS) either 1 or 0, puts them as input to the two input nodes. A separate node module was created, (a module is the object-oriented equivalent of a class in Verilog) that takes two inputs, two weights from predefined weights that are hard coded, as well as a hard coded bias and an output. From there the output goes through a sigmoid module. The sigmoid module either could have been built into the node module, called from the node module or called from the main program, I don't believe it makes any difference so I just called it from main program. From there the outputs of the two sigmoids coming from the two nodes go into an output node then the output of that node goes into another sigmoid, the output of that sigmoid is the output of the solution of the XOR. The neural network is illustrated above here h1, h2 and y are the nodes. Figure 4.2

```
1 module node(
2         input clk,
```

```
        input [31:0] in1,
        input [31:0] in2,
        input [31:0] weight1,
        input [31:0] weight2,
        input [31:0] bias,
        output [31:0] out
);

wire [31:0] mul1OutAdd1In;
wire [31:0] mul2OutAdd1In;
wire [31:0] add1OutAdd2In;

mult    mult_1 (
        .clock ( clk ),
        .dataa ( in1 ),
        .datab ( weight1 ),
        .result ( mul1OutAdd1In )
        );

mult    mult_2 (
        .clock ( clk ),
        .dataa ( in2 ),
        .datab ( weight2 ),
        .result ( mul2OutAdd1In )
        );

add     add_1 (
        .clock ( clk ),
        .dataa ( mul1OutAdd1In ),
        .datab ( mul2OutAdd1In ),
        .result ( add1OutAdd2In )
        );

add     add_2 (
        .clock ( clk ),
        .dataa ( add1OutAdd2In ),
        .datab ( bias ),
        .result ( out )
        );

endmodule
```

As mentioned above the input to the 32-bit floating point node module takes two inputs, two weights, one bias, a clock signal and gives an output. The module takes the two inputs and multiplies them by their respective weights, in the case for this neural network, both weights for each node are the same, then adds the two products then adds the sum of the products and a bias and outputs the answer. For floating point multiplication and addition native Quartus modules are used from the Intel Quartus IP Catalog, both add and multiply have the option of either 32 bit or 64 bit floating point multiplication and addition along with

various options like asynchronous clear, clock enable etc. Both addition and multiplication modules take a clock input, two data inputs and a result output and all modules are tied together with 32-bit wires. Quartus IP Catalog mathematics modules are used under the assumption they are optimized for the FPGA hardware.

```
node n1(
        .clk(CLOCK1_50),
        .in1(inputLayer[0]),
        .in2(inputLayer[1]),
        .weight1(32'b01000001101000000000000000000000),
        .weight2(32'b01000001101000000000000000000000),
        .bias(32'b11000001001000000000000000000000),
        .out(node1OutSig1In)
);
```

Above is instantiating and calling the node function from the main module, The input of node 1 is the clock input, which is 50 MHz, the two inputs from inputLayer that comes from the HPS, the two weights in binary floating point, this is required because Verilog uses integer math natively and the bias which is also in binary for the same reason. The output is also in floating point. Both weights are 20 and the bias is -10.

```
module sigmoid(
        input clk,
        input [31:0] in,
        output [31:0] out
);

wire [31:0] expOutAddIn;
wire [31:0] addOutInvIn;

wire [31:0] addIn;

assign addIn[30:1] = in[30:1];
assign addIn[31] = ~in[31];

expo    expo_inst (
        .clock ( clk ),
        .data ( addIn ),
        .result ( expOutAddIn )
        );

add     add_inst (
        .clock ( clk ),
        .dataa ( expOutAddIn ),
        .datab ( 32'b00111111100000000000000000000000 ),
        .result ( addOutInvIn )
        );

inverse inverse_inst (
        .clock ( clk ),
```

```
30          .data ( addOutInvIn ),
31          .result ( out )
32          );
33
34  endmodule
```

The sigmoid module takes a clock signal required for the native floating point modules used, an 32-bit floating point input and output. Since a sigmoid takes the negative of the input the signed bit of the 32-bit floating point number is inverted then fed into the native exponential function from the Quartus IP Catalog and an output is given. The output of the exponential function is fed into the add module mentioned before where it is added with a floating point 1 then that result is fed into a inverse function then outputs the result and is the overall output from the sigmoid module. 32-bit was chosen for each module even though natural exponential and inverse allows for single, double and custom precision floating point math.

```
1  sigmoid s1(
2          .clk(CLOCK1_50),
3          .in(node1OutSig1In),
4          .out(sig1OutNode3In)
5  );
```

4.1.1 HPS C Program

The function of a C program that the HPS uses is to take two inputs from the user, in this case a 0 or a 1, scans them in as floating point numbers, they are stored in a floating point struct that parses the sign, exponent and mantissa parts of the floating point number. That struct is a parameter to the function floatToInt. The reason we need to convert the float to integer is because numbers are sent and stored as binary integers in the FPGA and Verilog. The floatToInt function takes the sign part and bit shifts it 31 bits to the left, shifts the exponent part to the left 23 bits and leaves the mantissa where it is then does a bit-wise OR to get the integer and returns the integer. That integer is then passed to the function hpstofpga along with an address and a register number of where it is to be stored in the FPGA. The hpstofpga function uses two parallel IOs (input/output) to pass information to the FPGA, a 32-bit lightweight parallel IO and a regular 32-bit parallel IO. The lightweight

IO is a low latency IO I use for sending the address of where I want the data to go in an array, the register I want it to go into and whether to write or read with a read/write bit. In much the same way the floating point integer is made, the address is the first 20 bits of the integer, the register is the next 10 bits and the read/write bit is the 30th bit. This information is parsed on the FPGA side to write to a data location. The other IO is just used to transmit the data after the data location is sent. Then the fpgatohps function is called that takes an address and register as parameters and returns an integer from the FPGA much in the same way the hpstofpga function does using the lightweight and regular IOs. Then the intToFloat function parses the integer to a float in the same way you would convert a binary number floating point number to a decimal floating point number. Then the function prints the result to the console, which should either be 1 or 0.

```
typedef union {
    float f;
    struct
    {
        unsigned int mantissa : 23;
        unsigned int exponent : 8;
        unsigned int sign : 1;
    } raw;

unsigned int floatToInt(myfloat var)
{
    return ((var.raw.sign << 31) | (var.raw.mantissa) | (var.raw.exponent <<
    23));
}

float intToFloat(unsigned int num) {
    int s = 0;

    unsigned int temp = (num >> 31);

    if(temp == 1) {
        num = num - pow(2,31);
        s = 1;
    }

    unsigned int exponent = bitExtracted(num, 31, 24);
    unsigned int mantissa = bitExtracted(num, 23, 1);

    int e = exponent - 127;
    float m = (float) (mantissa / pow(2,23));

    return pow(-1, s) * pow(2,e) * (m+1);
}

void hpstofpga(int addr, int reg, int data) {
    *(lw_pio_ptr) = ((addr) | (reg << 20) | (1 << 30));
    *(axi_pio_ptr) = data;
}
```

```
38
39 int fpgatohps(int addr, int reg) {
40     *(lw_pio_ptr) = (addr | (reg << 20));
41     return *(axi_pio_read_ptr) ;
42 }
43 } myfloat;
```

4.2 Implementation of 16-bit Fixed Point XOR Neural Network

16 bit fixed point was chosen because for the neural network because it has been shown that 16 is the minimum amount of bits required for a back propagation algorithm to converge. (Holt and Baker, 1991) Like the 32-bit floating point XOR neural network, the 16-bit fixed point XOR Neural Network takes two floating point numbers from the HPS (Hard Processor System) either 1 or 0, then takes them as inputs to the two input nodes. Also, like the floating point version, a node module was created that also takes the same inputs, the two inputs from the HPS, the weights, and a bias, along with the output. A separate module was created for the fixed point sigmoid because it uses a piecewise linear approximation instead of built in functions so it is best made into its own independent function. This module is also called from the main module. The same layout is used, the two outputs of the two nodes go into two sigmoids then the output of the two sigmoids go into the output node and the output goes into a final sigmoid where the answer comes out and is either the 1 or 0. The neural network is illustrated above here h1, h2 and y are the nodes.

No special mathematics functions are used for multiplication or addition because all numbers, either integer or fraction are scaled up and treated as integers inside the FPGA. In this case we are using 10 fractional bits so every number gets multiplied by $2^{10} = 1024$. For example, if the weight is 1.2345 then it is multiplied by 1024 which is 1264.128 which is just rounded to 1264, type casted as an integer and sent to the FPGA where integer math is done. To get an answer we from multiplication we get the integer product from the FPGA

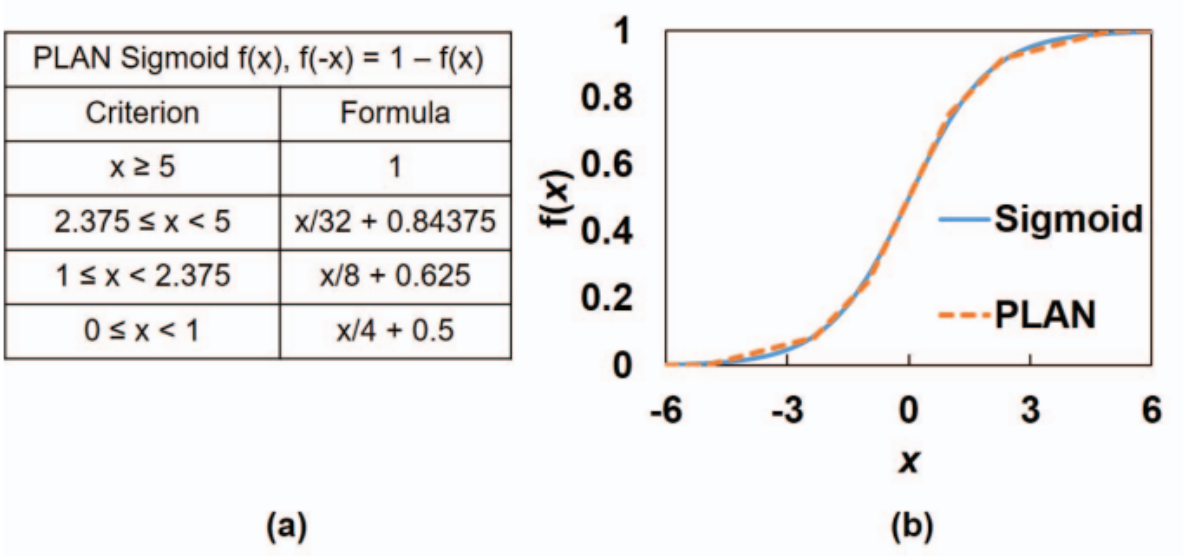
and divide by 1024^2 then cast to a float in the HPS or get a substring of bits from the product in the FPGA then only divide by 1024. For example if the two numbers being multiplied are 16 bits then the product will be 32 bits, to get a 16 bit answer if there are 10 fractional bits, you would pull out bits 10 through 25 of the 32 bit product, that answer would only need to be divided by 1024 to get a decimal.

Below is the Verilog code for a Node. It has 2 regular inputs that should be either 1 or 0 or 1024 or 0 since it is scaled up by 2^{10}, two weights which should be the same, a bias and an output. The first always block runs when both inputs in1 and in2 change then they are multiplied by the weights. Then those two products are added with a bias when the products change. 16-bit products are extracted from the 32-bit products as mentioned before. Inputs and outputs are signed for weights and bias that are negative. Negative numbers are represented in two's complement.

```
module node(
        input signed [15:0] in1,
        input signed [15:0] in2,
        input signed [15:0] weight1,
        input signed [15:0] weight2,
        input signed [15:0] bias,
        output signed [15:0] out
);

reg signed [31:0] mult1;
reg signed [31:0] mult2;

always @ (in1,in2) begin
        mult1 <= in1 * weight1;
        mult2 <= in2 * weight2;
end

always @ (mult1, mult2) begin
        out <= mult1[25:10] + mult2[25:10] + bias;
end

endmodule
```

To implement a sigmoid function, since FPGAs and Verilog do not support functions like e^x, and inverse and in neural networks activation functions such as sigmoid are often the bottlenecks, we need a fast and efficient implementation of the sigmoid function. There are several approaches to implementing a sigmoid in hardware. The classical ways of imple-

menting a sigmoid are lookup tables and a truncation of the Taylor series expansion. Other ways are divided into sum-of-steps, piece-wise linear, and a combination of the classical ways or others. The method that has been shown to have a good combination of low error and resource utilization is the PLAN approximation of implementing a sigmoid function. (Tisan, Oniga, and Attila, 2009) The PLAN approximation stands for Piecewise Linear Approximation of a Nonlinear function. Much like it sounds it implements a linear approximation on different ranges of the sigmoid function illustrated in Figure 4.3.



PLAN Sigmoid f(x), f(-x) = 1 – f(x)	
Criterion	Formula
x ≥ 5	1
2.375 ≤ x < 5	x/32 + 0.84375
1 ≤ x < 2.375	x/8 + 0.625
0 ≤ x < 1	x/4 + 0.5

Figure 4.3 (a) Expression of PLAN and (b) original and PLAN sigmoid plots.(Cheng, Yu, and Hashimoto, 2019)

```
module sigmoid (
    input signed [15:0] in,
    output signed [15:0] out
);

reg signed [15:0] result;
reg signed [15:0] temp;

always @ (in) begin
        if( in >= 5120 )
                out <= 16'd1024;
        else if( in < 5120 && in >= 2432) begin
                result = (in >> 5);
                out =  result + 16'd864;
        end
        else if( in < 2432 && in >= 1024) begin
```

```
                result = (in >> 3);
                out = result + 16'd640;
        end
        else if( in < 1024 && in >= 0) begin
                result = in >> 2;
                out <= result + 16'd512;
        end
        else if( in < 0 && in >= -1024) begin
                temp = -in;
                result = temp >> 2;
                out <= 16'd512 - result;
        end
        else if( in < -1024 && in >= -2432) begin
                temp = -in;
                result = (temp >> 3);
                out = 16'd384 - result;
        end
        else if( in < -2432 && in >= -5120) begin
                temp = -in;
                result = (temp >> 5);
                out = 16'd160 - result;
        end
        else if( in < -5120)
                out <= 16'd0;
end
endmodule
```

4.2.1 HPS C Program

The function of a C program that the HPS uses is to take two inputs from the user, in this case a 0 or a 1, scans them in as floating point numbers, then they are multiplied by the amount of fractional bits to the 2nd power, $2^{fractionalbits}$. That number is cast to an integer then sent the FPGA using the aforementioned hpstofpga functions and gets the result from the FPGA with the fpgatohps function then that result is divided by the number of fractional bits to the second power, $2^{fractionalbits}$ then type cast into a float and printed to terminal.

4.3 Implementation of Tally XOR Neural Network

The Tally Numbering system uses each bit in a bit string to represent a number, typically 1 but can be used to represent fractional numbers as well like 0.1. An example being if there are 5 bits in a byte then the byte represents a 5 if each bit has the value of 1. If each bit has

the value of 0.1 it would take 10 bits to equal a 1. To add and subtract, we bit shift left or right. For example, 4 plus 2 we bit shift left by 2. Bit shifting is computationally cheaper and quicker than adder circuits. To multiply or divide we reflect the bit string through a matrix with a slope to represent the multiplier. For example, if a bit string has 8 bits and we want to multiply by 0.5 (or divide by 2) the slope of a line in the matrix would be 0.5 and we get 4 bits out. This eliminates the use for multiplier blocks in the FPGA which are a limited resource. The sigmoid is also implemented.

The multiply module uses a lookup table in the form of a case statement where it matches an input slope to an array that represent the matrix the input is going to be reflected through. Below is an example of a slope of 1 so the effect is multiplying by 1, so the input would equal the output. A bitwise AND is done for each bit of the input number and for each row of the 2D matrix, so the result is only 1 if both the bit in the input bit string is 1 and if there is a 1 in the matrix element. Then we take that whole bit string and do and OR operation to it so if there is a 1 anywhere in the bit string the result is a 1, otherwise 0. Then the signed bits of the slope and the input are put through a bit-wise OR which will give the signed bit then the temp array is the output.

```
always @ (posedge clk) begin
    case(slope_in)
    20'b00000000000000000001: begin //1
        matrix[19] = 20'b10000000000000000000;
        matrix[18] = 20'b01000000000000000000;
        matrix[17] = 20'b00100000000000000000;
        matrix[16] = 20'b00010000000000000000;
        matrix[15] = 20'b00001000000000000000;
        matrix[14] = 20'b00000100000000000000;
        matrix[13] = 20'b00000010000000000000;
        matrix[12] = 20'b00000001000000000000;
        matrix[11] = 20'b00000000100000000000;
        matrix[10] = 20'b00000000010000000000;
        matrix[9] =  20'b00000000001000000000;
        matrix[8] =  20'b00000000000100000000;
        matrix[7] =  20'b00000000000010000000;
        matrix[6] =  20'b00000000000001000000;
        matrix[5] =  20'b00000000000000100000;
        matrix[4] =  20'b00000000000000010000;
        matrix[3] =  20'b00000000000000001000;
        matrix[2] =  20'b00000000000000000100;
        matrix[1] =  20'b00000000000000000010;
        matrix[0] =  20'b00000000000000000001;
    end
```

```
25
26      temp[0] = |(input1 & matrix[0]);
27      temp[1] = |(input1 & matrix[1]);
28      temp[2] = |(input1 & matrix[2]);
29      temp[3] = |(input1 & matrix[3]);
30      temp[4] = |(input1 & matrix[4]);
31      temp[5] = |(input1 & matrix[5]);
32      temp[6] = |(input1 & matrix[6]);
33      temp[7] = |(input1 & matrix[7]);
34      temp[8] = |(input1 & matrix[8]);
35      temp[9] = |(input1 & matrix[9]);
36      temp[10] = |(input1 & matrix[10]);
37      temp[11] = |(input1 & matrix[11]);
38      temp[12] = |(input1 & matrix[12]);
39      temp[13] = |(input1 & matrix[13]);
40      temp[14] = |(input1 & matrix[14]);
41      temp[15] = |(input1 & matrix[15]);
42      temp[16] = |(input1 & matrix[16]);
43      temp[17] = |(input1 & matrix[17]);
44      temp[18] = |(input1 & matrix[18]);
45      temp[19] = |(input1 & matrix[19]);
46
47      out[19:0] = temp;
48      out[20] = in[20] ^ slope[20];
49  end
```

The add module is relatively simple conceptually but the implementation is more complicated. There is first a lookup table (case statement) to find the value of each input as an integer. I believe a lookup table is faster then adding each bit but may take more memory/FPGA space. There are several if statements to for sign and number value because with the Tally system bit shifting a number right by a larger value the value of the bit string doesn't result in a negative number but a 0. For example 1111 (equivalent to 4) minus 1111111 (equivalent to 7) would result in 4 bit shifted to the right 7 times which would be a 0 not a -3. So 7 would need to be subtracted by 4 with a negative signed bit. This is a function that may be able to optimized more or redesigned since it uses conventional adds and subtracts.

The sigmoid is a simple lookup table too, since in this instance a 1 equals a 1, that is for every 1 bit equals a value of 1, inputting values into a sigmoid function, since the upper limit of sigmoid is 1 and the lower is 0 and since we are not using fractional bits the only result can be 1 or 0. The module takes the signed bit and only the first 5 bits of the accumulate because for the sigmoid function anything above a 5 is a 1 and anything below a -5 is a 0.

The lookup table shows that any input 0 or above is a 1 since the sigmoid of 0 is 0.5 and rounding up equals a 1 and anything below a 0 is 0 rounding down.

```
module sigmoid(
        input [50:0] in,
        output reg [20:0] out
);

integer total;

wire [5:0] in6;

assign in6[4:0] = in[4:0];
assign in6[5] = in[50];

always @ (in6) begin
        case (in6)
                6'b011111: out <= 21'b000000000000000000001; // 5
                6'b001111: out <= 21'b000000000000000000001; // 4
                6'b000111: out <= 21'b000000000000000000001; // 3
                6'b000011: out <= 21'b000000000000000000001; // 2
                6'b000001: out <= 21'b000000000000000000001; // 1
                6'b000000: out <= 21'b000000000000000000001; // 0
                6'b100001: out <= 21'b000000000000000000000; // -1
                6'b100011: out <= 21'b000000000000000000000; // -2
                6'b100111: out <= 21'b000000000000000000000; // -3
                6'b101111: out <= 21'b000000000000000000000; // -4
                6'b111111: out <= 21'b000000000000000000000; // -5
        endcase
end

endmodule
```

Unlike the floating and fixed point neural networks, A node module was not made. All the multiply, add and sigmoid modules are instantiated in the main module. Both inputs go into the multiply modules then the output of one is added with the bias the other, another add then adds that output and the other multiply output then that goes into a sigmoid. That is done for both nodes then the output of those are input into the output node in the same way.

As mentioned before a Tally bit can represent a fraction as well as an integer. If there were a neural network that has weights normalized between 1 and 0 with weights that are not too small, then this fractional representation could be used, without upscaling the weights to be integers. If the weights were too small like if weights needed to be represented down to 0.01 to be accurate then we would need 100 bit registers to just represent the numbers,

and they would need to be even larger for addition and multiplication. Any higher precision would be orders of magnitude worse. This would take up too much space and hardware in comparison to more traditional methods like fixed point.

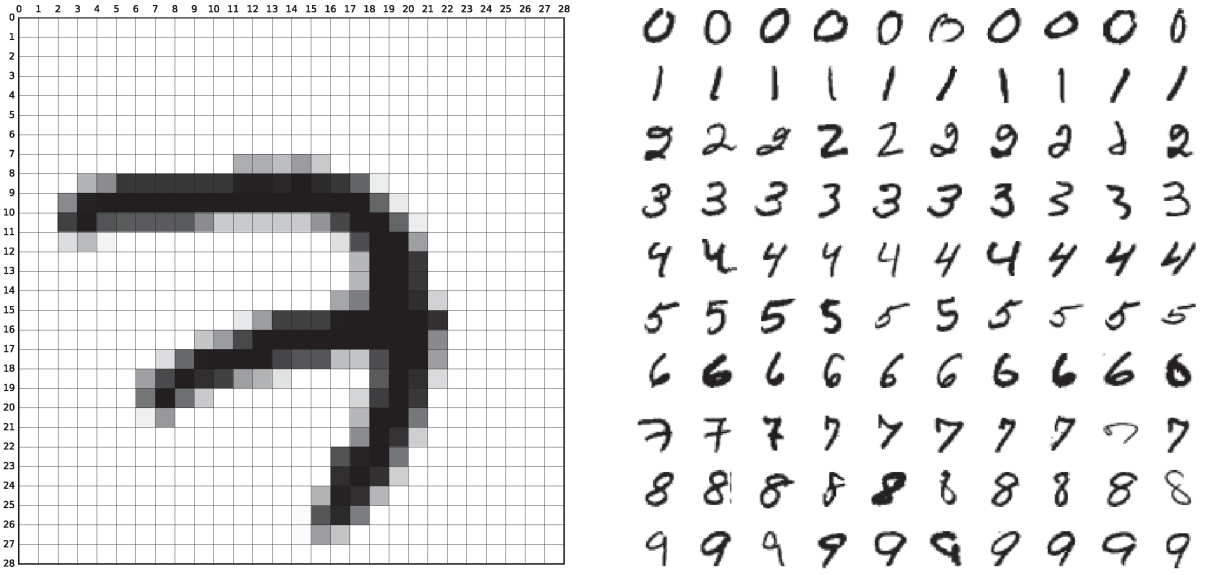
4.3.1 HPS C Program

The C program gets input from the user and then the input is converted to the number of 1's bits the user input. The number is raised to the power of 2, that is 2^x, then subtract one. For example, if the user typed in 4, the number would be raised to the power of 2 like 2^4 which is 16 or 10000 in binary then subtract 1 we get 15 which is 01111 in binary which is four 1s. Then that is sent to the FPGA through the parallel IO and AXI. That is the input to the Tally neural network. The C program gets the output in Tally form then takes the integer representation, subtracts one and takes the base 2 logarithm and that is the integer representation. It basically does the opposite of converting an integer to a Tally number.

4.4 MNIST Neural Network

As mentioned before the MNIST dataset is a set of 60,000 images of hand written numbers from 0 to 9. Each image is 8-bit grey scale image of 28 by 28 pixels, so there are 256 black and white values and 784 pixels total as illustrated in Figure 4.4. The neural network is a Multi-Layer Perceptron with two layers, a input layer with 784 inputs (an input for each pixel), 40 hidden nodes, and 10 output nodes (one for each number 0 through 9) similar to what is illustrated in Figure 4.5 The number of hidden nodes is completely arbitrary as are most hyper parameters in neural networks. Hyperparameters such as the number of hidden nodes are changed to try to increase accuracy. In this case 40 was chosen so the 32-bit floating point version would fit onto the FPGA. To get the weights for the neural network a multi-layer perceptron was trained in MATLAB/Octave using code downloaded from GitHub made by David Stutz from a seminar paper called "Introduction to Neural Networks." (Stutz, 2014)

This multi-layer perceptron uses logistic sigmoid activation function, a sum of squared error function and is initialized with random weights. Since this thesis is not about training of neural networks except in that it pertains to the inference part of the neural network the details of neural network training will be left out.



(**a**) MNIST sample belonging to the digit '7'.

(**b**) 100 samples from the MNIST training set.

Figure 4.4 MNIST Dataset Examples and Layout (Baldominos, Saez, and Isasi, 2019)

4.4.1 FPGA Architecture

The overall architecture is a partial pipeline where the HPS feeds the FPGA input data an image/number at a time, a 784 node input vector. The weights were loaded at the beginning of the program that represent a 784 x 40 matrix and a 40 x 10 matrix. Vector-matrix multiplication is done, the details of which will be elaborated on later, and the result is then a 40 node vector which is taken out of the FPGA to the HPS where the activation function, which a logistic sigmoid, is applied then the vector is loaded again into the FPGA. Why the HPS is used to apply a sigmoid will be detailed later in the thesis. Then another

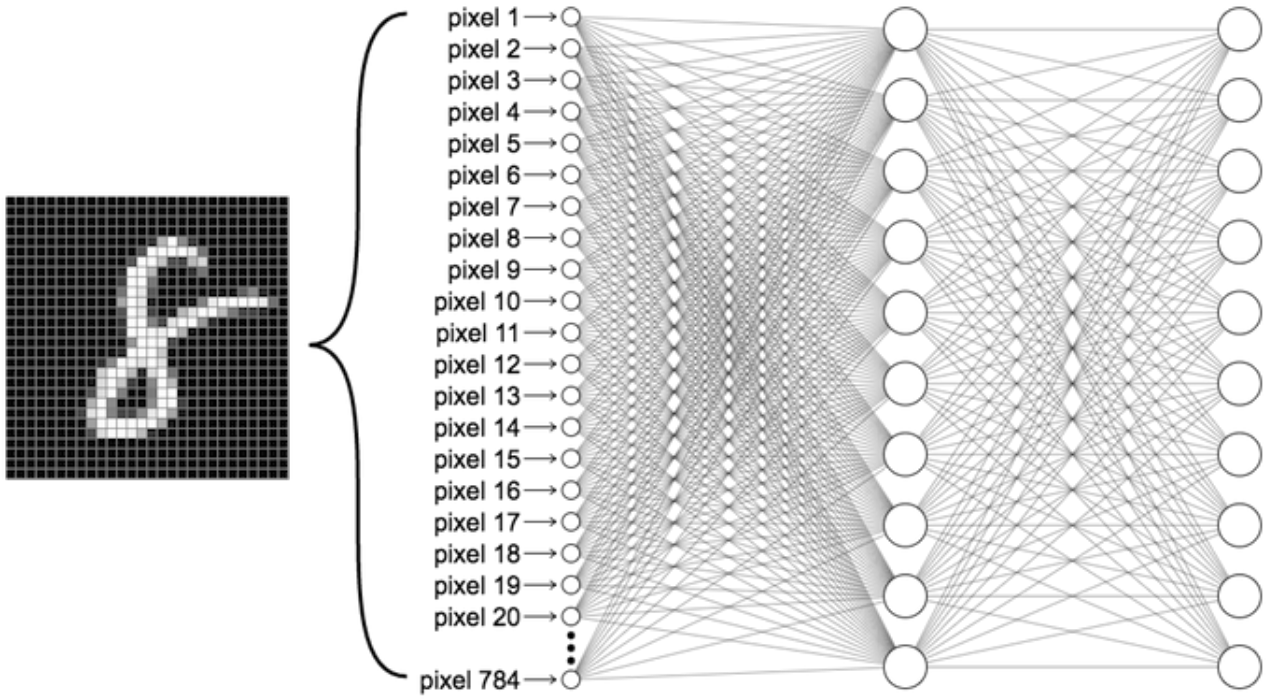


Figure 4.5 Illustration of the MNIST Multi-layer Perceptron (ML4A, n.d.)

vector-matrix multiply is done with the 40 x 10 matrix of weights the resulting 10 node vector is sent to the HPS where the sigmoid activation function is applied and the highest value is found. The index of the highest value is the resulting number. That number is then compared with what the number is supposed to be from the test data. This process is done for all 10,000 test numbers and the number of correct matches is counted and compared to the pure HPS version.

The design of the pure HPS neural network is that it takes the whole test data matrix, there are 10,000 test numbers, so the test matrix is 10,000 x 784, that matrix is multiplied by the weight matrix which is 784 x 40. This is accomplished by two nested for loops. The resulting 10,000 x 40 matrix goes through the activation function which is a sigmoid using nested for-loops to iterate through the matrix. Then that matrix is matrix multiplied the same way the other one was. This results in a 10,000 x 10 matrix which is put through the sigmoid activation function again. Then for each of the 10,000 lines the highest number of

the 10 is found then that index is stored and compared to the actual result of the test data set.

Loading the data into the FPGA from the HPS is done through the Advanced Extensible Interface (AXI). In the Cyclone V FPGA/SOC has a lightweight AXI and a regular AXI. The lightweight AXI is a 32-bit wide low latency interface between the FPGA and HPS. The normal AXI can but 32, 64 and 128 bits wide. I use the light weight AXI for addressing data before it is sent. The address is broken down into 3 parts, the first 20 bits are the address in a vector, the next 10 bits represent the register number and the 30th bit represent read or write, the same as used in the XOR neural network section. The regular AXI sends data in integer form from the HPS into binary on the FPGA. So weights need to be calculated from decimal to an integer number that represents the number in binary. This was done in the same ways it was in the XOR NN sections.

Ideally there would be a multiplier for every multiply operation but that would be unrealistic since the input layer is 784 and the hidden layer has varied anywhere from 100 hidden nodes to 40 hidden nodes. Even at the lowest amount of nodes that would require 784 x 40 multipliers but the FPGA only has 224 18x19 multipliers and 112 Digital Signal Processing (DSP) blocks. If there were a multiplier for every multiply operation and an adder for every add then the only delay would be propagation delay for matrix multiply operations. But since resources are limited a multiply-accumulate (MAC) architecture was chosen. A MAC in this case takes the input vector and every row from the weight matrix and multiplies each element from both vectors and adds them in the process. Basically a dot product. Theoretically, using a MAC architecture you could have up to 112 hidden nodes without having to reuse MACs since there are 112 DSP blocks. This is not able to be done with 32-bit floating point as it takes too much FPGA resources as shown in the MNIST section of the Evaluation chapter.

The Quartus development software has a built-in module for multiply accumulate. The built in module is meant for integer numbers but we can treat fixed point numbers as integers

scaled up by a factor of 2^N where N is the number of fractional bits in the fixed point number. For fixed point I used 16 bits with 12 fractional bits, 4 integer bits in twos complement to represent negative numbers. I chose these because the highest weight or layer values do not exceed 3 bits or 15. To convert decimal numbers to integers we multiply the decimal number by 2^{12} because there are 12 fractional bits. 2 to the 12 power is 4096. An example, $1.5 * 0.5 = (4,096 * 1.5) * (4,096 * 0.5) = 6,144 * 2,048 = 12,582,912$, divide that number by $(4,096 * 4,096) = 16,777,216$ to convert back to decimal, $12,582,912/16,777,216 = 0.75 = 1.5 * 0.5$. This just takes advantage of the distributive property of multiplication and addition. The advantage of using the built in integer MAC, other than having to make my own, is that each multiply-accumulate operation (multiply two numbers then add to running total) takes one clock cycle. So the number of clock cycles a MAC would need to calculate the dot product of two vectors would equal the length of the vectors, in this case 784.

In the case of floating point numbers there is no MAC for the Cyclone V FPGA but there is a floating point multiply and a floating point add module of varying input size. I made a MAC from both of these in a design similar to that of what I tried to do with the fixed point modules by making a MAC out of one floating point multiply module and a floating point add module where the multiply multiplies the vector elements and the add, adds the output of the multiply with the running total. The floating point multiply module has a 5, 6, 10, and 11 clock cycles and the adder module has 7 through 14 clock cycles of latency. So total per multiply accumulate operation the least latency would be 12 clock cycles where the highest would be 25. This is obviously a lot more than 1 clock cycle the integer MAC takes. So we could assume the floating point neural network take more time than the fixed point one because all other operations are the same.

An efficient implementation of the sigmoid function for 32-bit floating point and 16-bit fixed point is different for both. For the 32-bit floating point neural network a sigmoid function would need to be made the same as the XOR Neural Network, tying together a logarithm function, an add function and an inverse function. Making a sigmoid function for

each of the 40 nodes would use too much resources for a neural network that already uses 89% (37,305 out of 41,910 ALMs) of FPGA logic units. It would also take 40 more DSP blocks for multiplication, using a total of 80 of the 112 DSP blocks. So a single sigmoid function can only be used if the FPGA even has enough logic units or ALMs left over for a single one. That single sigmoid module would have to apply the activation function to all of the nodes in the hidden layer and output layer sequentially.

For the 16-bit fixed point neural network, the PLAN Sigmoid would be used like with the XOR Neural Network. The piece-wise linear approximation of the sigmoid function uses much less FPGA resources then the 32-bit floating point sigmoid. The PLAN of the sigmoid divides up the sigmoid function into 7 parts and approximates each section with a linear function with a slope and a y-intercept as shown in Figure 4.3 (a). Each slope is the inverse of a power of two so only bit shifts are used for multiplication, so no actual DSP or multiplier blocks are needed. Since the 16-bit fixed point neural network only uses about 9% of FPGA logic units (3,653 out of 41,910 ALMs) it is conceivable that a 40 sigmoid modules could be instantiated for each of the 40 hidden nodes and can calculate the sigmoid of each node in parallel and reused for each of the 10 output nodes.

The only trade off in using the PLAN of a sigmoid over the floating point sigmoid module is accuracy as the floating point sigmoid function is more accurate. For an apples to apples comparison, only a single fixed point sigmoid function should be used sequentially like the floating point one is used. Even then the PLAN sigmoid would be faster since the individual parts of the floating point sigmoid function, logarithm, add and inverse, take several clock cycles of latency each, whereas the fixed point sigmoid only requires a multiply and an add.

Chapter Five

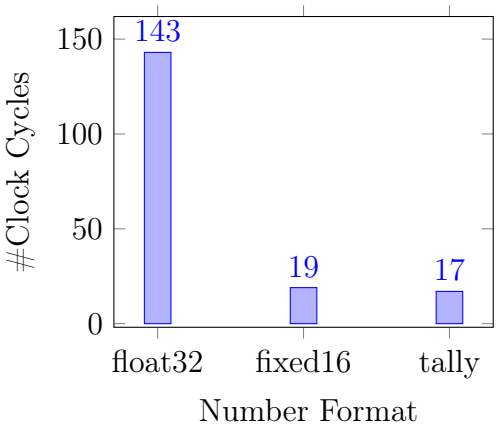
EVALUATION

The purpose of heterogeneous computing is to speed up computations and other metrics that may improve as a result is use of memory, power consumption, use of hardware resources while trying to maintain accuracy. These are also the goals of the Tally System, by replacing computationally and resource expensive, built in multipliers that are limited and decrease the amount of computation time by replacing the multipliers and adders with logic gates and bit shifts. Therefore the metrics that the Tally System, 16-bit fixed point and 32-bit floating point will be measure on are speed and resources used. There is no way of monitoring power usage in the Cyclone V FPGA [cite cyclone v manual] but it can be inferred that the amount of FPGA resources used is proportional to the amount of power used since dynamic power usage consumes more power than static power usage of FPGA elements.

5.1 XOR Neural Network

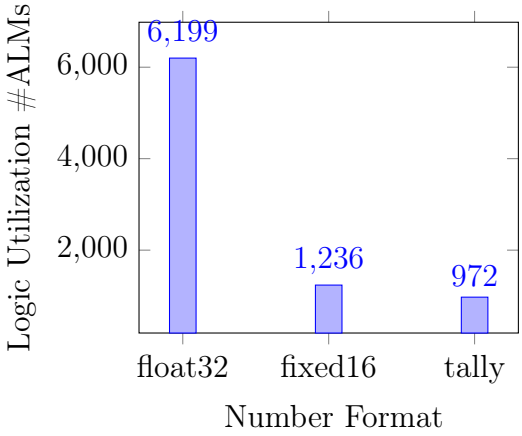
The comparison of 32-bit floating point, 16-bit fixed point and the Tally System using the same XOR neural network will be compared based on compute time in clock cycles, how many logic elements used, how many multipliers used and how many registers/memory used.

5.1.1 Runtime

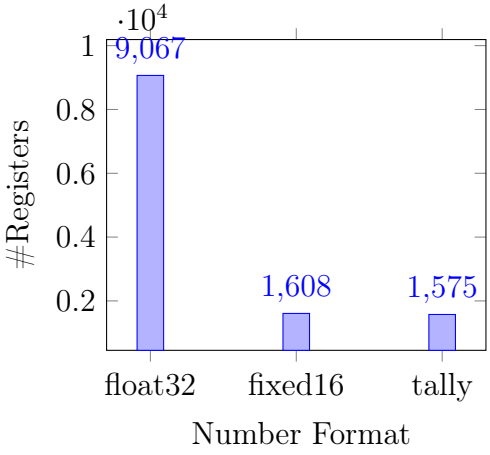


For the XOR Neural Network the number of clock cycles was measured inside the FPGA because the the network is too fast to measure with the C program. The neural network in each number format is done before the HPS can query the FPGA again. The FPGA clock for each of the neural networks is fifty megahertz so to get actual time one simply just multiply the number of clock cycles each one takes by the inverse of fifty million. As we can see the Tally neural network is only a few clock cycles less than fixed point neural network. There maybe different ways of optimizing the Tally math functions to shave off a few more clock cycles but overall it doesn't seem like it can be improved significantly. The floating point neural network takes more than seven times as much clock cycles as both fixed and tally this seems like an obvious trade off for flexibility and accuracy.

5.1.2 Resources

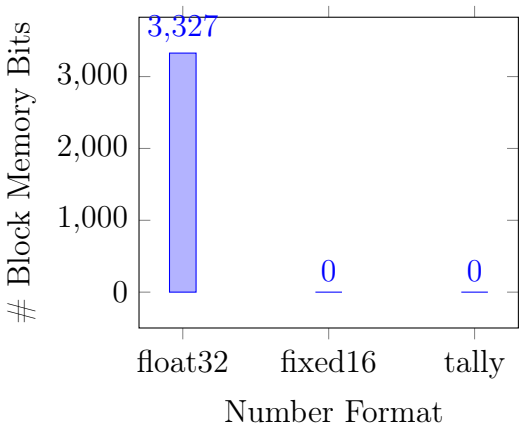


Unsurprisingly, the 32-bit floating point neural network takes up much more FPGA logic elements and resources than the other two neural networks. This may be obvious in comparison to the 16-bit fixed point neural network since the data width is twice as much, 32 versus 16 but its not as obvious when compared to the Tally neural network which uses registers of varying data length and since it takes more bits to represent a number. One might assume that using logic gates and shifts to do math operations is consumes much less resources than floating point and fixed point. This is particularly evident if we also take into account DSP Blocks.

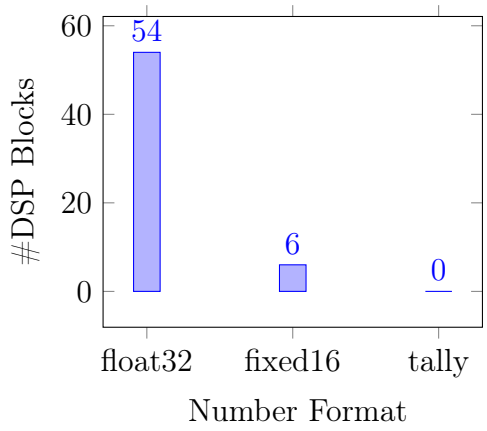


The number of registers used is proportional to the number of logic elements used which makes sense. Registers are the storage elements within the FPGA that we use to form the

cores of things like counters, shift registers, state machines, and DSP functions, basically anything that requires us to store a value between clock edges.



The floating point neural network also used block memory whereas the fixed point and tally networks did not. None of the three were explicitly designed to use block memory so use of block memory was probably decided by the compiler.



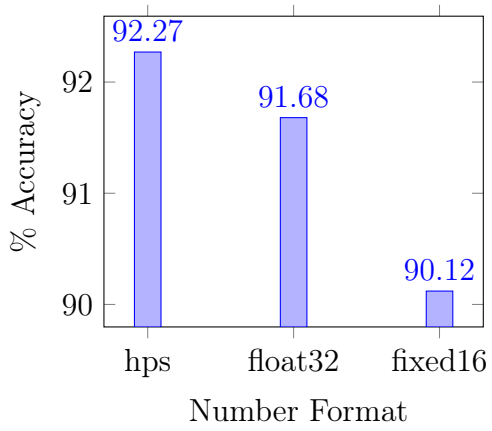
Digital Signal Processing Blocks (DSP Blocks) are specific hardware in a FPGA designed to do multiply operations quickly and efficiently for signal processing. The Cyclone V FPGA has 112 variable precision DSP Block and 224 18x19 multipliers (*Terasic DE10-Nano: IoT* 2019). As we can see the floating point neural network takes almost half the DSP Blocks for a simple 3 node neural network where each node has two multiplies and each node outputs into a sigmoid function. The fixed point consumes less DSP blocks because numbers are

only 16-bits wide and sigmoid is implemented using the PLAN function which reduces down into a linear function. Tally of course uses zero multiplier blocks.

5.2 MNIST Neural Network

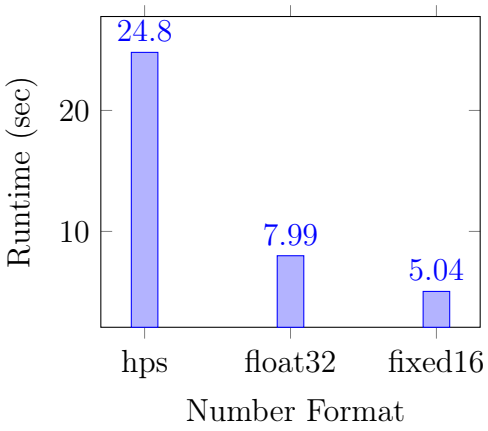
The MNIST Neural Networks was an attempt at building a larger neural network in an FPGA and seeing how it applies to HPS/CPU only, 32-bit floating point, 16-bit fixed point and the Tally system. It was later decided that the Tally system would require registers that are too large given the weights that we were working with but a lot was learned about how to implement a larger neural network in a FPGA and how it could be done using the Tally System. This will be detailed in the following sections as well.

5.2.1 Accuracy



The accuracy of the HPS should be the goal since the trained neural network isn't 100% accurate. The results were expected since the 32-bit floating point favors accuracy over speed and usage of hardware resources. When instantiating floating point modules in Verilog and Quartus, the option of speed was favored over accuracy which could account for the less than 1% accuracy drop off. There is an expected drop off of accuracy of the fixed point neural network due to the fact is uses half as much bits as floating point and only 12 fractional bits.

5.2.2 Runtime

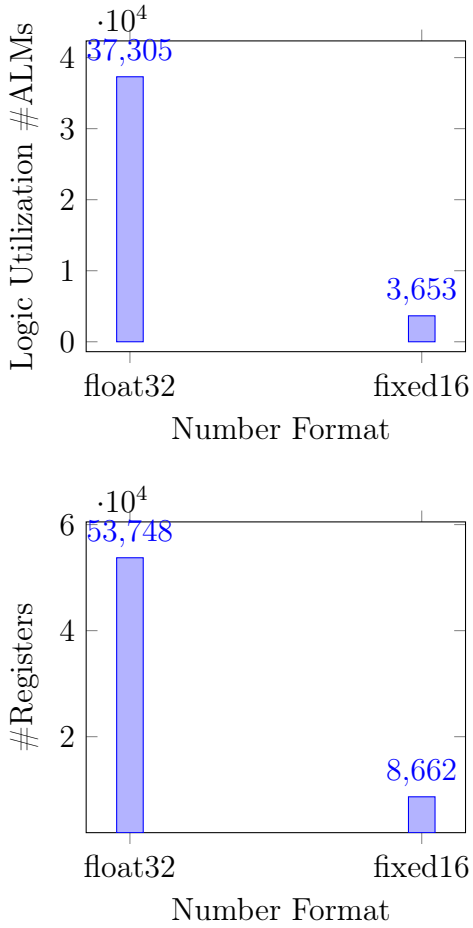


The speed up was obviously expected since the FPGA neural networks have a MAC for each node in the neural network that run simultaneously, whereas the HPS does matrix multiplication sequentially. The faster fixed point neural network is expected since Verilog and FPGAs support integer math natively and Quartus software has a integer MAC module that does multiply and add functions in the same clock cycle. Whereas floating point modules must be instantiated for floating point math and a floating point MAC was created out of floating point multiply and add modules since the Cyclone V does not support floating point MACs. Each multiply and add module have a built in delay several clock cycles, for the floating point neural network the delays are 8 and 6 clock cycles respectively, totaling 14 clock cycles for each MAC operation. For that reason one might expect the fixed point neural network to be significantly faster than the floating point network since a fixed point multiply-accumulate operation happens in one clock cycle as opposed to 14 for the floating point multiply-accumulate. The fixed point network is only 36% faster than the floating point network.

For the Tally System, if the neural network is small enough it many be able to be fully connected like the XOR Neural Network since its is not limited by the number of multipliers but most likely a Tally MAC would need to be created. A Tally MAC module would take two numbers, multiply them then add the product to a running total. The amount of clock

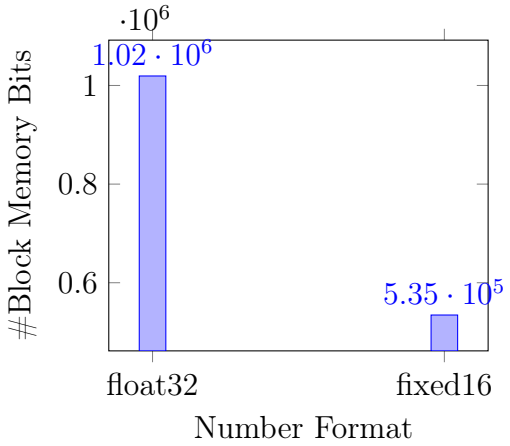
cycles a Tally MAC module would require for each multiply-accumulate operation would depend on propagation time since both the multiply and addition Tally modules are based off propagation time not the clock.

5.2.3 Resources

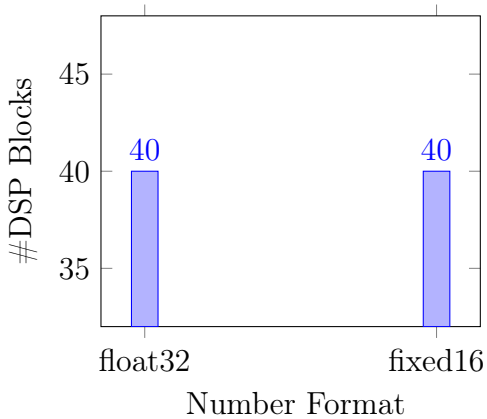


The amount of logic elements and registers used is quite high for the 32-bit floating point probably more than is accounted for by having a data width that is twice as high as 16-bit fixed point. A lot of logic elements and registers are probably used for floating point math since FPGAs are meant for integer math and not floating point math. Like the XOR neural networks the amount of registers used for each neural network is roughly proportional to the amount of logic elements used. For the Tally neural network, it would depend on how

big registers are to represent weights. The larger registers are to represent weights the more memory is used, and the larger look up tables are in math functions.



It is not surprising the 32-bit floating point neural network uses approximately twice as much block memory bits as the 16-bit fixed point since each network has the same amount of weights but obviously 32-bits is twice as much as 16-bits. How many block memory bits a Tally neural network would be dependent on what each bit represents and the highest weight we want to represent. For example, if each bit represented 0.1 and we want to represent -1 to 1 we would use 11 bits for each weight, 10 bits for values and 1 for sign. A high amount of precision could be used since even the floating point neural network only used 18% of the 5.6 megabits of block memory although the size of the look up tables would become large and take up a lot of logic elements.



Both floating and fixed point neural networks have the same amount of DSP Block because both have 40 hidden nodes in the hidden layer and they are reused in the output layer with 10 nodes. The Tally neural network would use zero multipliers so its not relevant to this section.

Chapter Six

FUTURE WORK

6.1 Quantization and the Tally System

A big problem that occurred while doing this project was trying to represent the MNIST weights in Tally bits. The bit strings would be too large to represent small fractional numbers. One way to mitigate this is with Quantization. Quantization as mentioned in the Related Work section is representing weights in a lower bit precision numbers such as 8-bit integer (int8). This has the advantage of quicker calculations and less memory bandwidth and less memory used but less accuracy in representing weight values. There are two ways to do Quantization, during training and post-training. Post training quantization causes reduced accuracy with conversion from 32 or 64 bit floating point numbers to something like 8 or 16 bit integer numbers. Luckily neural networks tend to be resistant to noise so accuracy loss can be minimal. Accuracy loss is less for larger neural networks but can affect smaller neural networks more. The other approach is "quantization aware training" which is exactly what it sounds like. We quantise the weights and activations of the neural network during training and let the network learn the new range of the weights. Once the network is aware of the quantization it can adjust it's weights in an attempt to minimize the loss.

For the Tally system 8-bit twos compliment integer would still be too high as the range is $-2^{N-1}, 2^{N-1} - 1$ or -128 to 127, so in Tally that would require 129 bit registers, where the

most significant bit is the signed bit and the rest are integer bits. Luckily, there has been and there is on going research into "aggressive quantization" that quantizes weights even further down into 4-bit or even 2-bit integer as with binary and ternary neural networks. With 2 or 4 bit quantization, "quantization aware training" is necessary because as one would expect quantizing high precision floating point weights into 4 bit weights post training, causes significant accuracy losses. 4-bit integer weights would be close to ideal because the range is of 4 bit twos compliment numbers is -8 to 7 which for the Tally system would only require 9 bit registers to represent numbers, 17 bit registers for sums and 65 bit registers for products. Also, since Tally does not conform to a base two number system it can be any amount of integers, not just powers of two. Although all the research done in aggressive quantization involves binary numbers. Another advantage of Tally is that when weights are up scaled into integer they need to be down scaled to pass through an activation function such as logistic sigmoid or hyperbolic tangent, but since the current implementation of sigmoid in Tally uses look up tables the down scaling can simply be factored into the look up table calculations. Also with Tally, instead of quantized training into a range of integers one could do quantized training into the natural range of the weights and represent every bit with a discretized fraction such as 0.1 and not worry about down scaling weights for the activation function.

6.2 Binary Neural Networks and the Tally System

The only research found that closely resembles the Tally System are Binarized Neural Networks. BNNs have the same goal of using cheap logic gates and bit shifts as opposed to expensive and slow multiplier and adder hardware to implement MAC operations for neural networks. BNNs reduce weights into two values, -1 and 1. Multipliers are replaced with Exclusive NOR (XNOR) gates and accumulate (adds) with bit shifts. The activation function simply takes the accumulate result and if it is greater than 0 then the output is

1 and if it is less than 0 then the output is -1. It has been shown that it is possible to train BNNs on MNIST, CIFAR-10 and SVHN and achieve nearly state-of-the-art results. (Courbariaux, Hubara, and Bengio, 2016) While it may seem obvious that BNNs may have the advantage in a lower memory requirement in that weights are represented with only two numbers and don't require the use of what can be large look up tables like the Tally System, although Tally may be able to be more accurate with its higher precision. Both will need to be implemented in comparable neural network architectures to compare metrics such as speed, resource utilization and accuracy.

Comparing mathematics functions of both methods, BNNs multiply function are XNOR gates which are made up of AND, OR and NOT gates, while Tally uses a look up table then uses AND gates to compare bits in bit strings then OR gates to find if there is a bit in every bit string. The add function both both BNNs and Tally use bit shifts, although Tally may require more bit shifts per add operation. For an activation function BNNs use a simple if-else statement where Tally uses a look up table. While both methods may seem comparably fast, Tally may up more resources such as memory and registers although BNNs may require more neurons to be as accurate. All of the BNNs examined have had thousands of hidden nodes per layer such as 4,096 or 8,192 (usually powers of 2). Tally may be able to use the same amount of nodes as conventional floating point neural networks or quantized fixed point or integer neural networks. In the context of this paper, both would need to be implemented in a FPGA. In the Binary Neural Networks paper by Courbariaux etc. (Courbariaux, Hubara, and Bengio, 2016) BNNs were implemented in a optimized GPU kernel not customizable bit level manipulation such as FPGAs. BNNs seem uniquely well suited for FPGAs as opposed to GPUs which use hundreds of 32-bit floating point processors.

Chapter Seven

CONCLUSION

It has been shown that the Tally System does work for simple neural networks such as a XOR neural network. The implementation of the Tally neural network used less resources and was as fast as if not faster than 16-bit fixed point number representation and integer arithmetic. In comparison to 32-bit floating point numbers and math Tally was much faster and used much less resources. Although, Tally is not as flexible mathematically or able to represent numbers as accurately as either 16-bit fixed point or 32-bit floating point without using extremely large registers.

The concept of neural network quantization was introduced as a way to mitigate Tally's lack of high precision number representation by representing weights in discretized values. More specfically, "aggressive quantization" which can be used to train neural networks to discretize weights to 4-bit twos compliment values i.e. integers between and including -8 to 7. Neural Network Quantization is an area of research of its own, especially quantization done while training.

It has also been shown that matrix multiply operations can be accelerated in a SoC/FPGA with a high speed AXI bus by running MAC operations in parallel. A generic matrix multiply could be implemented in a SoC/FPGA to accelerate neural network calculations for C programs. It would most likely need to be implemented in fixed point or integer because it has been shown that a 40 neuron MAC in 32-bit floating point takes 89% of the FPGAs

resources whereas the same 40 neuron MAC in 16-bit fixed point only took 9% of FPGA resources.

REFERENCES

An in-depth look at Google's first Tensor Processing Unit (TPU). URL: https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

Baldominos, Alejandro, Yago Saez, and Pedro Isasi (2019). *A Survey of Handwritten Character Recognition with MNIST and EMNIST*. URL: https://www.mdpi.com/2076-3417/9/15/3169.

Cheng, TaiYu, Jaehoon Yu, and Masanori Hashimoto (2019). *Minimizing Power for Neural Network Training with Logarithm-Approximate Floating-Point Multiplier*. URL: https://ieeexplore.ieee.org/abstract/document/8862162.

Courbariaux, Matthieu, Itay Hubara, and Yoshua Bengio (2016). *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to 1 or -1*. URL: https://arxiv.org/abs/1602.02830.

Holt, Jordan L. and Thomas E. Baker (1991). *Back propagation simulations using limited precision calculations: Semantic Scholar*. URL: https://www.semanticscholar.org/paper/Back-propagation-simulations-using-limited-Holt-Baker/ba4c527a2fccf3245116d540c51a8bbb8fdb77bc.

Lavrenko, Victor (2015). *Neural Networks 6: solving XOR with a hidden layer*. URL: https://www.youtube.com/watch?v=kNPGXgzxoHw.

Louizos, Christos, Matthias Reisser, and Tijmen Blankevoort (2018). *Relaxed Quantization for Discretized Neural Networks*. URL: https://arxiv.org/abs/1810.01875.

Machine Learning Library (MLlib) Guide. URL: https://spark.apache.org/docs/latest/ml-guide.html.

ML4A. URL: https://ml4a.github.io/ml4a/looking_inside_neural_nets/.

Moore, Andrew (2017). *FPGAs for Dummies*. Vol. 2. John Wiley and Sons.

Omondi, Amos (2006). *FPGA Implementations of Neural Networks*. DOI: 10.1007/0-387-28487-7.

Stutz, David (2014). *Introduction to Neural Networks*. https://github.com/davidstutz/matlab-mnist-two-layer-perceptron.

Tensor Cores in NVIDIA Volta GPU Architecture. URL: https://www.nvidia.com/en-us/data-center/tensorcore/?ncid=afm-chs-44270&ranMID=44270&ranEAID=a1LgFw09t88&ranSiteID=a1LgFw09t88-aV0qMG.XRRhg7Q9xPAm2cA.

Terasic (2017). *Terasic - SoC Platform - Cyclone - DE10-Nano Kit*. URL: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1046 (visited on 09/30/2018).

Terasic DE10-Nano: IoT (2019). URL: https://software.intel.com/en-us/iot/hardware/fpga/de10-nano.

Tisan, Alin, Stefan Oniga, and Buchman Attila (2009). *Digital Implementation of The Sigmoid Function for FPGA ...* URL: https://www.researchgate.net/publication/228618304_Digital_Implementation_of_The_Sigmoid_Function_for_FPGA_Circuits.