Exploration of Neural Network Implementations on Field-Programmable Gate Arrays

Raiyan Siddique

December 18, 2023

1 Introduction

The goal of this project was to implement and understand various implementation of Neural Networks in digital design using Field-Programmable Gate Arrays, FPGAs, in order to understand strategies for optimization and hardware acceleration. Initially, the focus was on implementing a basic and explicit neural network structure that was trainable, and then later moved into exploring strategies for more advanced optimizations, specifically in the context of Tensor Processing Units. The explicit neural network structure is complete and usable, however only the main components of the TPU was implemented due to time constraints.

2 Explicit Neural Network

This exploration began by creating an explicitly defined neural network, with the benchmark structure being that for a XOR gate. A XOR gate is our chosen representation of a simple non-linear system that highlights the use fullness of Neural Networks of emulating complex systems. Creating the building blocks to enable this, proves that this system could be used to build a neural network given an explicit structure.

2.1 Neuron Implementation

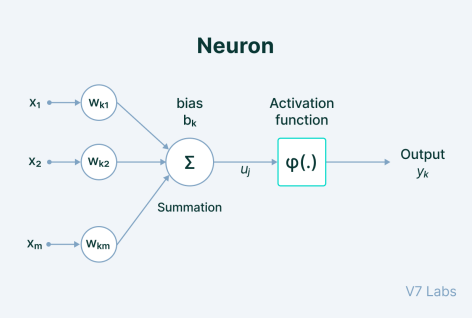

Figure 1: Visual of the fundamental operations of a neuron.

The first and most basic unit that needs to be developed to create a neural network is a neuron. This basic implementation takes all the neuron operations as is and condenses them all into one module. The two primary arithmetic operations were addition and multiplication. Although addition is a fairly simple and efficient operation, in high speed networks, multiplication becomes very time intensive and would be the primary inefficiency in the system. Furthermore, the neccessity of having decimal numbers to represent that weights and biases would add more complexity to the system, especially if the conventional representation of floating point numbers in bits was used.

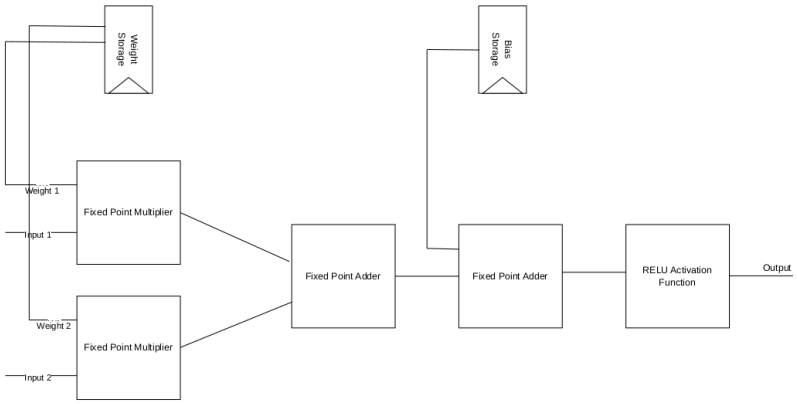

Figure 2: Block Design of Neuron

2.1.1 Representing Floating Point Numbers

The first major design decision made was deciding on an appropriate representation of floating point numbers as this would determine the granularity and precision of our weights and the complexity of our arithmetic operations. There were two major options to choose from. The first standard was the general representation of floating point numbers where there is a sign bit, exponent bit (M) and the mantissa(E), where the number is represented by $-1S \times M \times 2E$. The second and more favorable option was to use a fixed point representation of numbers where there is a sign bit, the bits representing the whole number (Q), and the bits representing the decimal numbers (M), given in a Q.M structure. The Q component is treated as a regular binary number, where the first bit in Q is 2^0 and the second bit is 2^1 ect. The M bits have the reverse structure, where the MSB of M is 2^{-1} and the second MSB is 2^{-2} onward. Although this structure gives less granularity than the floating point representation this structure has 2 major advantages. First, the addition operation remains the same when using fixed point, whereas there is a lot more overhead in adding together two floating point numbers. Second, floating point multiplication is much more complicated than fixed point complications, meaning that when implemented on the FPGA we are unable to take advantage of Digital Signal Processing Slices. Fixed point multiplication is the same process as multiplying two regular signed numbers together, meaning that this representation enables us to use the accelerated multiplication process that the DSP slices have, greatly increasing the efficiency of this implementation.

2.1.2 Activation Function

There were two options considered for activation functions, the classic Sigmoid function and the Rectified Linear Unit Function. The sigmoid function is limited to values between 0 and 1, which would significantly reduce the probability of overflow happening in multiplications, which would lead to a loss of meaningful information. However, because the calculation of the sigmoid function would create unneccessary overhead and significantly reduce computation speed, we opted to use the simpler RELU function. Even with pre-computed values for the Sigmoid Function stored in BRAM, the algorithm to associate registers to values would still be more complicated and take up more space then simply passing any positive value through the activation function. Furthermore, continuously accessing memory in this fashion would be un-scalable as the neural network would be limited in computation at a rate of N (dependent on number of read ports) number of read operation per clock cycle or the repeated activation functions would consume all the memory on our FPGA. Thus, RELU became the obvious choice for a hardware implementation.

Because our Fixed Point Multiplier and Adders have overflow checking, our RELU function does flatten out at some maximum value, however, that is taken into consideration when training.

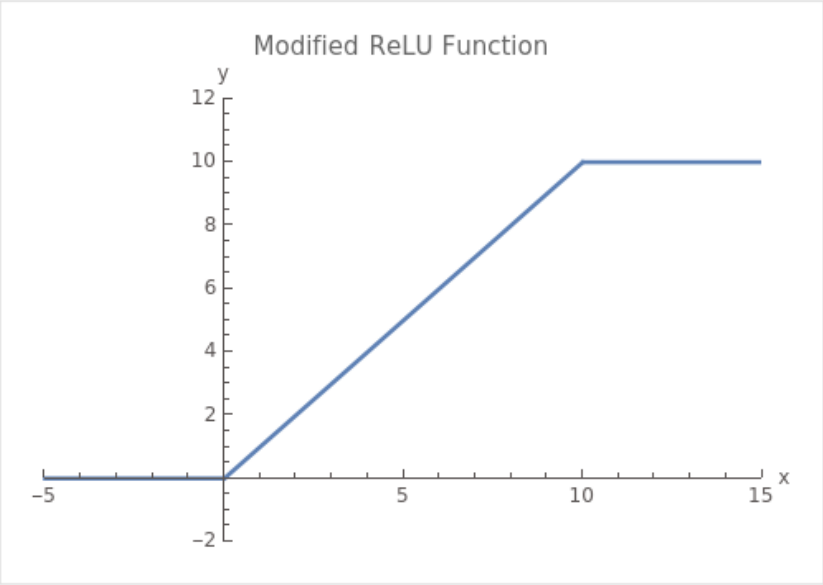

Figure 3: Modified RELU Function Example

2.2 Neural Network Instantiating

With this fundamental unit built we have the components needed to build any complexity of neural networks. By connecting the outputs and inputs of different neurons, we can model the XOR gate. By training the weights off chip using Python, we are able to store them in the necessary registers.

2.3 Trainer

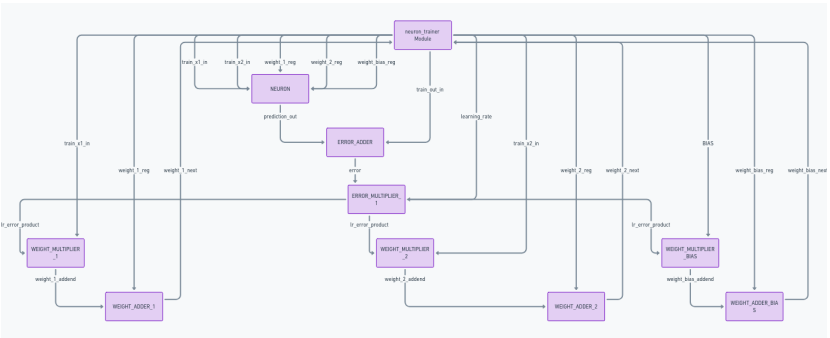

Figure 4: Block Diagram for Trainer

Instead of beginning with the large problem of back propagation, this trainer is made using a basic gradient descent implementation. This approach iteratively adjusts the neurons weight and bias to minimize the loss function, in this case just simple error function. The neuron module receives input features and makes a prediction, which is then compared to some expected output. This error is then used in the gradient descent step, where the weights are updated in the direction to reduce error. To control the rate of this process, the Learning Rate parameter is used to control and prevent overshooting while training.

2.4 Limitations

There are a lot of computational limits to this implementation. The first and biggest limitation is the number of computations that can be done in a given time. Because each node has instantiated its own multiplication unit, the size of the neural network as well as the number of computations that can be

done in a given clock cycles is severely limited. Furthermore, if this design is every fabricated as an ASIC it would be very difficult for it to be used in multiple use cases, as the structure of the neural network connect be changed after it is fabricated.

The training algorithms that can be implemented are also limited. Although back propagation would inherently cost a lot of memory, in this implementation this would also be a very time intensive process.

3 Tensor Processing Unit

Because of the limtations of the above approach, other avenues of implementation were researched. A robust and commonly used accelerator for machine learning applications are Tensor Processing Units. Developed by Google, the TPU accelerates a lot of machine learning and neural network operations.

3.1 Google's Design

Although the specifics of the TPU design are not open source, the general components and block architectures are.

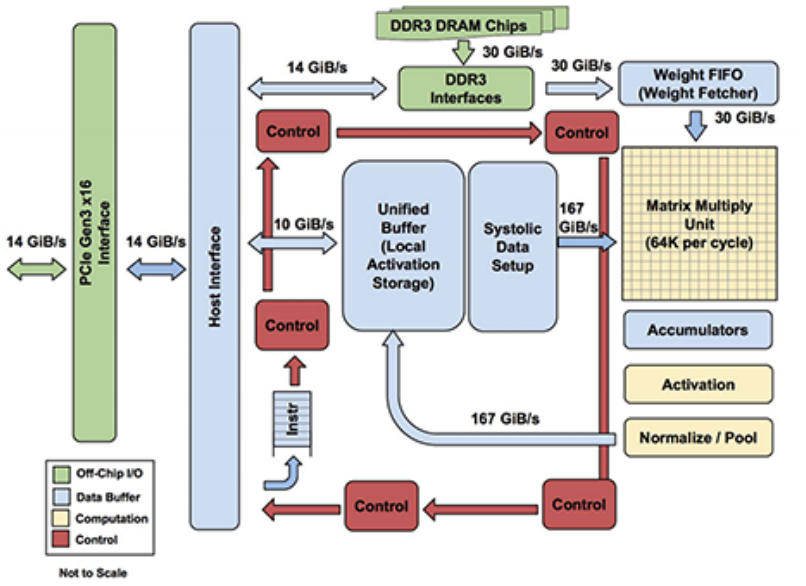

Figure 5: Block Diagram of Google's TPU

The core of the design relies on the Matrix Multiply Unit (MXU). This MXU, operating on 128x128 matrices, relies on the principles of reduced precision arithmetic, to balance computational intensity and the needs of neural network processing. Complementing the MXU are Vector Processing Units (VPUs), which manage element-wise operations that are commonplace in neural networks. There is an on-chip Unified Buffer, a high-bandwidth memory system, routed to minimize latency and efficiently supply data to the MXU. The Instruction Buffer and Control Unit are designed to optimize the execution of machine learning models through effective operation scheduling and data flow management. Although implementing the entire architecture would be very time intensive, specific components were develope and other were designed.

3.2 Matrix Multiplication Unit

The core of the Matrix Multiplication is the Systolic Array, an optimized way to implement matrix multiplication.

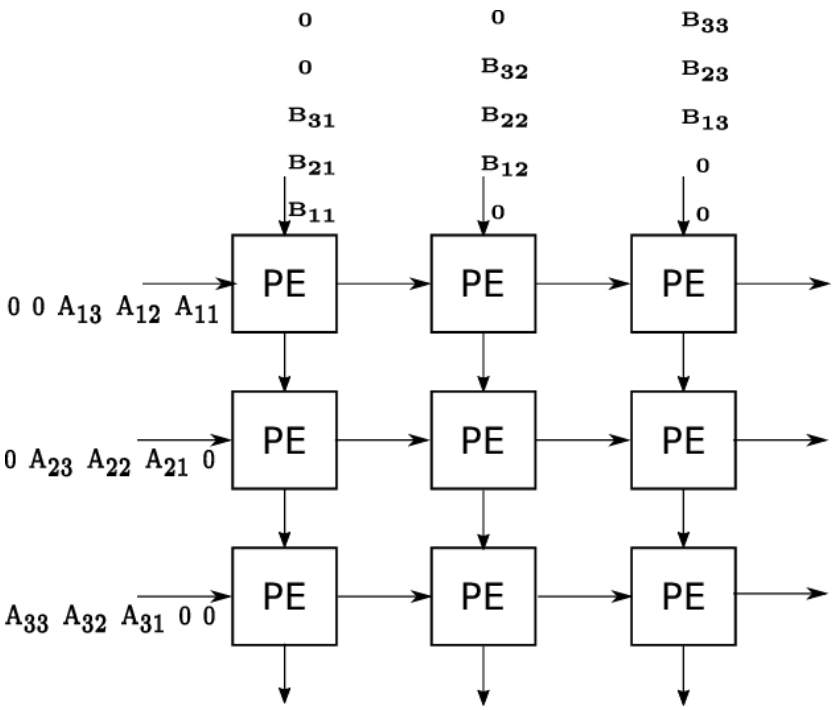

Figure 6: Systolic Array Structure

Each PE unit holds a specific value, depending on which of the two common schemes are used. The first is for the PE unit to hold the value of the result of matrix multiplication. For example, in the first PE (1, 1), at the first clock cycle B11 * A11 is added to the result of matrix multiplication which can call C11. C11's initial value is 0, but after the first cycle it is the result of A11 * B11. At the next clock cycle C11 is then added to B21 * A12, making the result of C11 = A11*B11 + B21*B12. The arrays between PEs represent how values propagate through the Systolic Array. Each horizontal arrow takes in the previous A value in the PE and each vertical arrow takes in the previous B value from the previous PE. This happens until the only values being entered into the systolic array nodes are 0s.

The second scheme as each PE node store the weight value, which we will call C this scheme. B represents the result and A represents the input layer or other matrix being multiplied. Instead of the two input matrices being propagated through the systolic array, the result and one of the arrays are propagated through, meaning the the final result matrix is being shifted out of the array.

Although the second scheme is more efficient, as there are less memory accesses needed because the weights can stay stored in the PE for multiple multiplications, there is a lot more complex overhead that would make this project significantly more complicated, specifically in accumulating the weights.

Thus the first scheme was implemented, as it would require a lot less logic overhead and does not require a separate accumulator block, as the systolic array accumulates values for us.

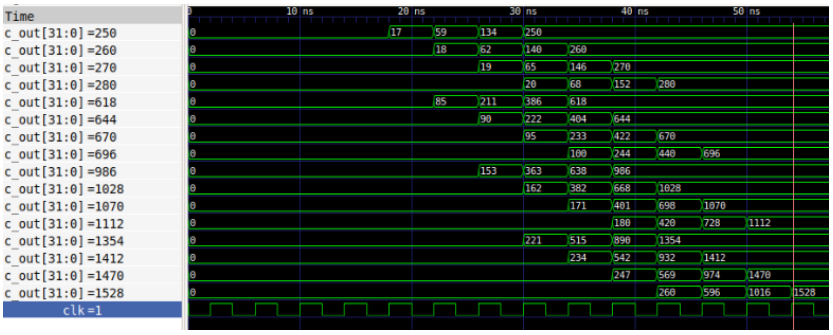

Figure 7: Implementation results of 4x4 multiplication where A is 1 through 16 row-wise and B is 17 through 32 row wise.

The results show how much faster matrix multiplication, which in this case represents a layer of computations in a neural network, can be done. This process is further optimized by using DSP blocks instead of regular multiplication, which also significantly speeds up the compute process. However, this also means the size of the systolic array is also dependent on the number of DSP blocks. This means that for larger problems there needs to be a control for block matrix multiplication, where each block is a break down of a larger matrix multiplication.

3.3 Theoretical Implementation

If given more time, more blocks would be created to make the controller for block matrix multiplication and dimension analysis. The scaffolding and starting implementation of this controller was done, where the input layer and weight dimensions are stored in the associated BRAM. As the matrix multiplication occurs these values are updated, such that no mistake is made in the dimension and systolic array usage in the MXU. Because the weights and input layers are all store in BRAM, an AXI4 lite protocol would be used to read and write to the memory.

Instead of using the CISC instruction set the Google uses there are only a select few instructions that would need to be used. Load Weights, Read Weights, Load Memory, Read Memory, Multiply, Accumulate and Activate. This simple architecture would allow for any matrix operations that need to be done be computed efficiently and stored.

Other optimizations that were explored for the systolic array were multi pumping and pre-adding in the systolic array nodes. However, for the sake of this exploration these optimizations were drawn out, but not implemented.

4 GitHub Repository

Parts of this project were done in collaboration with Marc Efitime
`https://github.com/raiyansiddique/FPGA_Neural_Network`