Verilog Implementation of Fully Connected Neural Network Layer

Varun Mishra

VLSI Design

Professor Bo Yuan

***Abstract*–Based on the project suggestion, this report details an implementation of the fully-connected (FC) neural network that operates with a single layer containing ten neurons, utilizing eight-bit quantization. The report explores the methodologies involved in synthesizing a digital design for neural network operations like matrix-vector multiplication and activation functions. This implementation demonstrates the feasibility of deploying a simple neural network architecture on hardware for resource-constrained applications such as edge devices. Simulation results will validate the functionality of the design, highlighting its accuracy and timing performance under test scenarios. Future work will aim to expand the network's complexity and improve scalability for more advanced applications.**

Keywords–Fully-connected neural network, edge devices

I. Introduction

Neural networks are a "method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain" [1]. This process utilizes "interconnected nodes or neurons in a layered structure that resembles the human brain" [1]. One such type of neural network is the fully-connected neural network (FCNN). The FCNN "consists of fully connected layers that connect every neuron in one layer to every neuron in the other layer" [2]. Due to this structure, the fully connected network is structure agnostic, as "there are no special assumptions needed to be made about the input" [2]. As a result, the main advantages of this type of neural network is that the integration of all features of the preceding layers is ideal for complex pattern recognition, it can handle many different input data types, and it is simple to implement and widely supported by existing deep learning frameworks. This deep learning "architecture is considered one of the most diffused models since early neural networks studies, and it is still popular among modern deep methods" [3] and they are foundational building blocks in hybrid architectures.

This implementation bridges the gap between software-based machine learning and efficient hardware implementations. Specifically, in this project, the low-precision arithmetic reduces computational cost without significant loss in performance. By designing an FCNN hardware architecture, it can significantly boost performance and computational efficiency for specific applications.

The goal of this project is to design and implement a single-layer FCNN with ten neurons in eight-bit quantization using Verilog. Then, using a testbench simulation, its effectiveness and design can be validated.

II. Related Work

The implementation of neural networks in hardware has become a critical area of research due to the growing demand for efficient and low-latency machine learning solutions. Existing work focuses on optimizing performance for computationally intensive algorithms, particularly for embedded systems and edge devices.

One prominent example is Google's Tensor Processing Unit (TPU), designed to accelerate neural network computations for cloud-based machine learning tasks. The TPU features a systolic matrix multiply unit for efficient matrix-vector multiplication, enabling high throughput for 8-bit quantized operations [4].

Previous studies have also explored hardware-specific designs, such as DaDianNao and ShiDianNao, which integrate neural computational units into distributed architectures. These accelerators adopt

fixed-point arithmetic to enhance throughput while minimizing energy consumption. Such architectures underline the importance of optimizing memory access patterns and arithmetic precision in hardware implementations of neural networks [5].

This project builds on these advancements by implementing a fully connected neural network layer in Verilog. Unlike prior work that often emphasizes large-scale deployment or convolutional operations, this project focuses on an 8-bit quantized FCNN optimized for edge devices. It demonstrates the feasibility of lightweight hardware solutions for specific applications, bridging the gap between theoretical neural network design and practical hardware implementations. By leveraging parallelism and low-precision arithmetic, this work aims to contribute to the development of efficient and scalable neural network hardware.

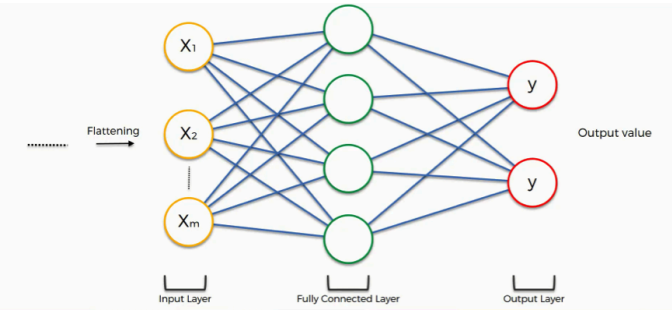
III. Data Description

To verify the design and effectiveness of the Verilog implementation, I used predefined, fixed arrays with integer values in the testbench. This approach was necessary due to the difficulty of directly importing large datasets into Verilog. Unlike Python, where datasets can be easily loaded and manipulated, Verilog does not support direct data imports, making it challenging to work with real or synthetic datasets. To address this, I manually defined small, representative input vectors and corresponding expected outputs, which were fed into the neural network model during simulation.

IV. Method Description

A. Fully Connected Neural Network Architecture:

In a fully connected neural network, every neuron at each layer is connected to every other neuron in the next layer. So in a one layer network, a sample diagram will look like this:



In this diagram, there are three input neurons, four neurons in the hidden layer, and two output neurons. So, each input neuron has four connecting weights that correspond to each of the neurons in the next layer. At the hidden layer, where the multiply-accumulate results are computed with the bias, and the activation function is applied for the output.

For this project, my implementation consists of ten input neurons, each connected to ten neurons. From which, there are ten output neurons that are calculated.

B. Underlying Concept

Each hidden layer node receives weighted inputs from the previous layer, performs a computation, and then passes that result through an activation function to produce an output. At each hidden layer node, there are four components: the input, weight, bias, and activation function.

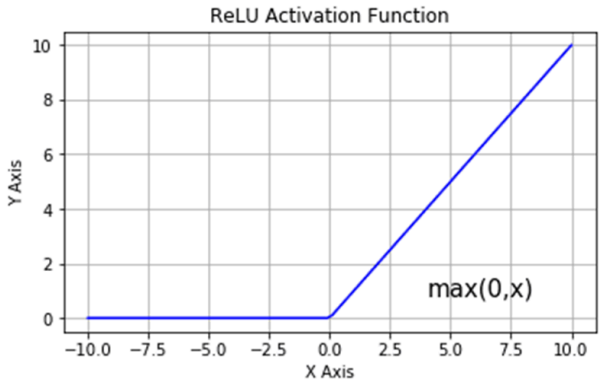
- Inputs: The input is the data value that is fed into the node from the previous value.
- Weights: These are learned parameters that scale the inputs. So, each input has an associated weight.
- Bias: This is an additional parameter that allows the model to better fit the data by shifting the output of the activation function.
- Activation function: The function applied to the weighted sum of the inputs and bias, often non-linear, to introduce non-linearity to the model.

For a hidden layer node, the output computation would look like this,

$$Output_i = ReLU(\Sigma_{j=0}^{n-1} inputs_j \times weights_{ij} + bias_i)$$

where, $inputs_j$ is the input to that node, $weights_j$ is the weight associated with each input at that specific node, $bias_i$ is the bias for the ith node, and ReLU is the activation function.

The activation function decides whether a neuron should be activated given its calculated weighted sum. The idea is to introduce non-linearity into the model, allowing the network to learn and represent complex patterns in data that a simple linear model cannot capture. I chose the ReLU activation function because "in recent years researchers have found that other activations, notably the rectified linear activation (commonly abbreviated ReLU or relu) $max(x, 0)$ work better" [6] for the fully connected networks. This is because for the ReLU function, "the slope is nonzero for a greater part of the input space, allowing non zero gradients to propagate" [6].



V. Model Description

The Verilog implementation of the fully connected neural network consists of a single module, neural_net, that computes the forward pass of the network by performing weighted sums of the inputs, adding biases, and applying the ReLU activation function to produce the output for each neuron.

A. Inputs, Weights, and Biases

```
1  module neural_net (
2      input clk,
3      input reset,
4      input [7:0] inputs [0:9],        // 10 inputs, 8 bits each
5      input [7:0] weights [99:0],      // 100 weights, 8 bits each
6      input [7:0] biases [0:9],        // 10 biases, 8 bits each
7      output reg [7:0] outputs [0:9]   // 10 outputs, 8 bits each
8  );
```

The network takes 10 input values, each 8 bits wide, represented as the inputs array. These inputs are fed into the network and serve as the features for the computation in the forward pass.

The weights array hold the 10 input neurons * 10 hidden layer neurons = 100 weights (8 bits each), corresponding to the connections between the 10 input neurons and the 10 output neurons. Each weight is indeed to correspond with the correct input-output connection. The weights are stored as a flat array, with the respective weight at position $i \times 10 + j$ representing the connection from the *j*th input to the *i*th output.

The biases array holds 10 biases (8 bits each) corresponding to the 10 output neurons. Each bias is added to the computed weighted sum for the respective neuron.

```
10     reg signed [15:0] mac_result [0:9]; // MAC result for each neuron
11     integer i, j, idx; // Loop variables
12
13     always @(posedge clk or posedge reset) begin
14         if (reset) begin
15             // Reset all outputs
16             for (i = 0; i < 10; i = i + 1) begin
17                 outputs[i] <= 8'b0;
18                 mac_result[i] <= 16'b0;
19             end
20         end else begin
21             // Compute MAC for each neuron
22             for (i = 0; i < 10; i = i + 1) begin
23                 mac_result[i] = biases[i]; // Start with the bias
24                 for (j = 0; j < 10; j = j + 1) begin
25                     idx = i * 10 + j;
26                     mac_result[i] = mac_result[i] +
27                                     (inputs[j] * weights[idx]);
28                 end
29
30                 // Apply ReLU activation and quantize to 8 bits
31                 if (mac_result[i] < 0) begin
32                     outputs[i] <= 8'b0;
33                 end else if (mac_result[i] > 255) begin
34                     outputs[i] <= 8'd255; // Saturate to 8 bits
35                 end else begin
36                     outputs[i] <= mac_result[i]; // Truncate to 8 bits
37                 end
38             end
39         end
40     end
```

B. Clocking and Reset

This module operates synchronously with the clock signal, clk. The reset signal, reset, ensures that all outputs and intermediate results are cleared to zero when activated, ensuring proper initialization of the module at the start of each operation.

C. Computation

The forward pass starts by computing the multiply-accumulate (MAC) operation for each of the 10 output neurons. Each output neuron receives weighted sums of the inputs along with the corresponding bias. This logic is implemented in the nested for loop from lines 22 to 38. The process involves iterating over all 10 inputs and accumulating the product of each input value and its corresponding weight. The bias is added to this sum, forming the pre-activation value for the neuron.

At line 26, the following product is accumulated into the mac_result intermediate variable:

$$Output_i = inputs_j \times weights_{ij} + bias_i$$

where the output is stored into the respective index in mac_result for each neuron.

The mac_result register is a 16-bit signed integer to allow for overflow from the multiplication and summation operation. Any overflow is then truncated to 8-bits with the activation function logic.

D. Activation Function

After calculating the intermediate weighted sum with the bias, the ReLU activation function is applied. The ReLU activation function is implemented from lines 31 to 37:

$$output[i] = max(mac\_result[i], 0)$$

Due to the 8-bit quantization requirement, I used conditional logic to saturate or set the weighted sum

appropriately. If the result is negative, the output is set to 0, and if the result is greater than 255 (the maximum value that can be represented with 8 bits), it is saturated to 255. This ensures that the output values are constrained to the 8-bit range [0, 255], while fulfilling the ReLU max function.

VI. Experimental Procedure and Results

In order to verify the design of the Verilog implementation, I wrote a Python script to compare the output values.

```python
Codeium: Refactor | Explain | X
def relu(x):
    """ReLU activation function."""
    return np.maximum(0, x)

Codeium: Refactor | Explain | X
def quantize_to_8bit(value):
    """Quantize the value to 8 bits, clamping between 0 and 255."""
    return np.clip(value, 0, 255)

Codeium: Refactor | Explain | X
def neural_net(inputs, weights, biases):
    """Feedforward function of a neural network with quantization and ReLU."""
    outputs = np.zeros(10, dtype=np.uint8)  # 10 outputs, 8 bits each (quantized)

    # Perform MAC and apply ReLU activation for each neuron
    mac_result = np.zeros(10, dtype=int)  # Array to hold MAC results for each neuron

    for i in range(10):  # For each neuron
        mac_result[i] = biases[i]  # Start with the bias for neuron i

        # Perform MAC operation
        for j in range(10):  # For each input
            mac_result[i] += inputs[j] * weights[i * 10 + j]  # Weighted sum for each neuron

    # Apply ReLU activation and quantize to 8 bits for each neuron
    print(f"MAC results before ReLU and quantization: {mac_result}")

    # Apply ReLU activation and quantize over the entire array
    mac_result = relu(mac_result)  # Apply ReLU to all elements
    mac_result = quantize_to_8bit(mac_result)  # Quantize to 8-bit values

    outputs = mac_result.astype(np.uint8)  # Store the final quantized outputs

    return outputs
```

```python
# Example test
if __name__ == "__main__":
    # Example inputs, weights, and biases
    inputs = np.array([9 - i for i in range(10)], dtype=np.uint8)

    # Initialize weights (800 values in a flattened array)
    weights = np.array([i for i in range(100)], dtype=np.uint8)  # 100 values, 8-bit

    # Initialize biases (10 values, 8-bit)
    biases = np.array([i for i in range(10)], dtype=np.uint8)

    print(weights)
    print(inputs)
    print(biases)
    # Run the neural network
    outputs = neural_net(inputs, weights, biases)

    # Print the results
    print("Outputs:", outputs)
```

A. Overview of Python implementation

- The ReLU activation function is represented with the relu() function from lines 3 to 5. It is implemented on line 5 as np.maximum(0, x), which returns 0 for any negative value and the value itself for nonnegative inputs.
- The quantize_to_8bit() function limits the output values to the 8-bit range (0, 255) by using the np.clip(value, 0, 255) on line 9.
- The neural_net() function performs the core computation of the neural network by initializing an array, mac_result, to hold the weighted sum of inputs for each neuron. The same weighted sum calculation is implemented on line 23 by incorporating the bias first on line 19.
- The inputs, weights, and biases are initialized as NumPy arrays from lines 29 to 45.

B. Experiment

To verify the design, I manually fixed the values for the Python implementation because it is challenging to import data sets into Verilog. For this example, the inputs are generated as a simple range from 9 down to 0 and the bias indexes are generated from 0 to 9. The weights are generated from 0 to 100. In order to compare the results, I loaded the same values in the Verilog testbench.

```
24      initial begin
25          // Initialize clock and reset
26          clk = 0;
27          reset = 1;
28          #5 reset = 0;
29
30          // Apply test data: Initialize inputs and biases
31        for (i = 0; i < 10; i = i + 1) begin
32          inputs[i] = 9 - i;
33          biases[i] = i;
34        end
35
36
37          // Initialize weights array using a loop (count up and down)
38
39        for (i = 0; i < 100; i = i + 1) begin
40              weights[i] = i; // Assign the value to the weights array
41          end
```

Here is the Python output result:

```
varunmishra@Varuns-Air final % python3 fcnn_8_bit.py
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
  24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
  48 49  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
  22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
  46 47 48 49]
 [9 8 7 6 5 4 3 2 1 0]
 [0 1 2 3 4 5 6 7 8 9]
 /Users/varunmishra/Documents/vlsi/final/fcnn_8_bit.py:23: RuntimeWarning: overflow encountered
   mac_result[i] += inputs[j] * weights[i * 10 + j]  # Weighted sum for each neuron
 MAC results before ReLU and quantization: [ 120  571 1022 1217  900  125  576 1027 1222  905]
 Outputs: [120 255 255 255 255 125 255 255 255 255]
```

Here is the Verilog output result:

```
# SLP: 11 (64.71%) signals in SLP and 6 (35.29%) interface signals
# ELAB2: Elaboration final pass complete - time: 0.1 [s].
# KERNEL: SLP loading done - time: 0.0 [s].
# KERNEL: Warning: You are using the Riviera-PRO EDU Edition. The perfor
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@a
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 5502 kB (elbread=427 elab2=4940 kernel
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: inputs: '{9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
# KERNEL: weights: '{99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87,
# KERNEL: biases: '{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
# KERNEL: Outputs: '{120, 255, 255, 255, 255, 255, 255, 255, 255, 255}
```

The outputs are the same, so the Verilog design implementation for this FCNN is valid.

VII. Conclusion

This report presented the implementation of a fully connected neural network (FCNN) with a single hidden layer and ten neurons, operating with 8-bit

quantization in Verilog. The design demonstrates the feasibility of deploying a simple neural network on hardware, with the primary focus on low-precision arithmetic to reduce computational cost, making it well-suited for resource-constrained applications such as edge devices. The use of low-precision calculations, while maintaining an acceptable level of accuracy, allows for the efficient deployment of the network in environments where memory, power, and processing power are limited.

The results from the simulation, which confirmed the validity of the Verilog design, highlight the potential of using small-scale neural networks in edge computing. With a fixed number of outputs (10 classifications in this case), this architecture could be extended to real-time applications like image or voice classification. The 8-bit quantization ensures that the system remains resource-efficient, and with further layers added, the network could tackle more complex tasks such as multi-class classification or object detection.

While this implementation is modest in terms of its complexity, it serves as a foundation for more advanced models. As neural networks are scaled up with additional layers and neurons, this approach could be adapted for a variety of edge computing applications, such as facial recognition, voice commands, and sensor data analysis. The ability to process data directly on edge devices with minimal latency and power consumption opens new possibilities in fields like IoT, autonomous systems, and real-time decision-making, where traditional cloud-based computing may not be viable.

VIII. References

[1] AWS, “What is a neural network? AI and ML guide - AWS,” *Amazon Web Services, Inc.*, 2024. https://aws.amazon.com/what-is/neural-network/

[2] P. Mahajan, “Fully Connected vs Convolutional Neural Networks,” *Medium*, Oct. 23, 2020. https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5 (accessed Dec. 24, 2024).

[3] L. F. S. Scabini and O. M. Bruno, “Structure and performance of fully connected neural networks: Emerging complex network properties,” *Physica A: Statistical Mechanics and its Applications*, vol. 615, p. 128585, Apr. 2023, doi: https://doi.org/10.1016/j.physa.2023.128585.

[4] N. P. Jouppi *et al.*, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” *arXiv:1704.04760 [cs]*, Apr. 2017, Available: https://arxiv.org/abs/1704.04760

[5] Z. Du *et al.*, “ShiDianNao,” Jun. 2015, doi: https://doi.org/10.1145/2749469.2750389.

[6] 4. Fully Connected Deep Networks, “TensorFlow for Deep Learning,” *O’Reilly | Safari*, 2019. https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html