

Handwritten Digit Recognition on FPGA

Contents

1	Introduction	3
1.1	Neural Network	3
1.1.1	Neural Network - Example	3
1.2	Why neural networks?	3
1.3	Field Programmable Gate Arrays (FPGAs)	3
1.4	Why FPGAs?	4
1.5	FPGA Design Workflow	5
1.6	Coding in FPGA	5
1.7	Hardware Synthesis in FPGA	6
1.8	High-Level Synthesis (HLS)	6
1.9	FPGAs for Neural Networks	6
2	Neural Network Implementation - Making a Model	8
2.1	Model Training	8
2.1.1	Training Data	8
2.2	Structure of Model	9
2.3	Extraction of weights and biases from the network	10
3	Neural Network Implementation - High Level Implementation in Vivado HLS	11
3.1	Functions Implementations	11
3.1.1	Exponential Approximation	11
3.1.2	ReLU Activation	12
3.1.3	Softmax Activation	12
3.2	Layers Implementations	12
3.2.1	Dense Layers	12
3.3	Top-level Function	13
4	Testing and Verification	15
4.1	Testbench	15
4.2	Breakdown of testbench code	16
4.3	Inputting Image to Model	17

5	Results	18
5.1	Performance Metrics	18
5.2	Output of Testbench in Vivado HLS	18
6	Conclusion	20

Chapter 1

Introduction

1.1 Neural Network

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain. It creates an adaptive system that computers use to learn from their mistakes and improve continuously. Thus, artificial neural networks attempt to solve complicated problems, like summarizing documents or recognizing faces, with greater accuracy[1].

1.1.1 Neural Network - Example

Figure 1.1 shows a simple neural network which has one input, one hidden and one output layer. Input and output layers contain two neurons each while hidden layer contain 3 neurons.

1.2 Why neural networks?

Neural networks can help computers make intelligent decisions with limited human assistance. This is because they can learn and model the relationships between input and output data that are nonlinear and complex. They are widely used in making smart systems nowadays.

1.3 Field Programmable Gate Arrays (FPGAs)

FPGAs are integrated circuits that can be configured by the user after manufacturing. They consist of an array of programmable logic blocks and a hierarchy of reconfigurable interconnects, allowing the creation of custom hardware circuits.

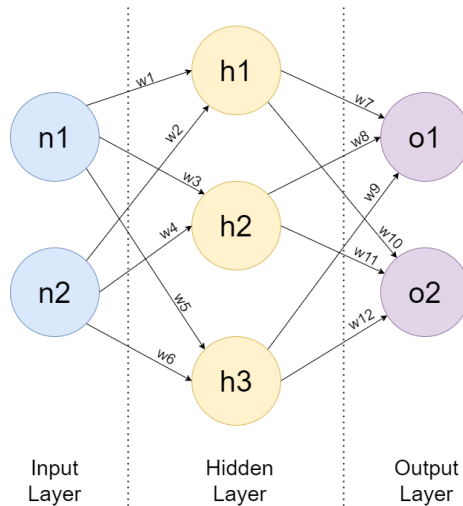


Figure 1.1: Neural Network Example

1.4 Why FPGAs?

Field Programmable Gate Arrays (FPGAs) offer several advantages for implementing neural networks:

- **Parallelism:** FPGAs can exploit the inherent parallelism in neural network computations, allowing for concurrent execution of multiple operations. This leads to significant speedup compared to sequential processing on general-purpose processors.
- **Customization:** FPGAs can be customized to implement specific neural network architectures and operations, optimizing the hardware for the target application. This flexibility allows designers to balance performance, power consumption, and resource utilization.
- **Low Latency:** FPGAs provide deterministic and low-latency processing, which is critical for real-time applications such as autonomous driving, robotics, and financial trading.
- **Energy Efficiency:** FPGAs can be more energy-efficient than CPUs and GPUs for certain tasks, as they can be tailored to perform only the necessary computations with minimal overhead.
- **Reconfigurability:** FPGAs can be reconfigured to implement different neural network models or updated designs without changing the hardware. This reconfigurability makes FPGAs suitable for research and development environments where algorithms are constantly evolving.

1.5 FPGA Design Workflow

The workflow of an FPGA Based Design[2] is given in the figure 1.2. Figure 1.2 depicts a typical workflow for designing, verifying, and implementing a chip architecture. It begins with the "Concept" phase, where the idea and specifications for the chip are defined. This is followed by the "Design" phase, which includes developing the chip architecture and writing the hardware description language (HDL) code. The design is then subjected to "Verification" through simulation and testing to ensure it meets the specified requirements and functions correctly. If any issues are detected, the design is revised. Once verified, the design moves to the "Implementation" phase, where it undergoes synthesis to convert the HDL code into a gate-level netlist, and then place and route to arrange and connect the gates physically. This structured approach ensures that each step is thoroughly checked and corrected before progressing, thereby reducing errors and ensuring successful chip fabrication.

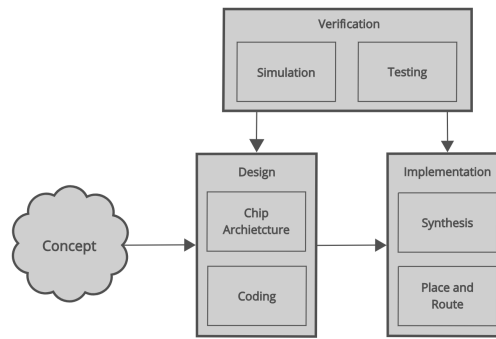


Figure 1.2: FPGA Based Design Workflow

1.6 Coding in FPGA

Coding for FPGAs (Field Programmable Gate Arrays) involves writing hardware description language (HDL) code to define the behavior and structure of digital circuits. The two most commonly used HDLs are VHDL (VHSIC Hardware Description Language) and Verilog. Coding in FPGA involves several key considerations and steps to ensure efficient and functional hardware design. Coding for FPGAs requires a deep understanding of both hardware and software principles. Efficient FPGA coding can lead to highly optimized designs that exploit the parallelism and reconfigurability of FPGAs, making them ideal for applications requiring high performance, low latency, and energy efficiency. Proper coding practices, thorough verification, and effective use of synthesis tools are essential to successful FPGA design.

1.7 Hardware Synthesis in FPGA

During the FPGA synthesis process[3], a high description design or an HDL design is converted into a gate level representation or a logic component. This means that FPGA code provided in high level language such as VHDL or Verilog is converted into logic gates using a synthesis tool as shown in figure 1.3.

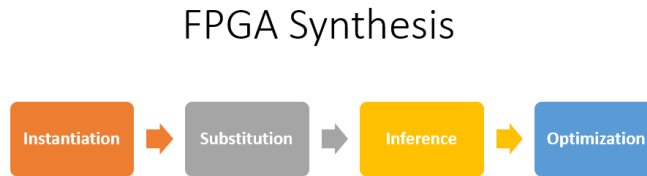


Figure 1.3: FPGA Synthesis Workflow

1.8 High-Level Synthesis (HLS)

HLS is a method of transforming a high-level algorithmic description of a circuit into a register-transfer level (RTL) design. HLS tools can automatically generate hardware descriptions from C, C++, or SystemC code, significantly reducing the design time. Its general workflow can be seen in figure 1.4.

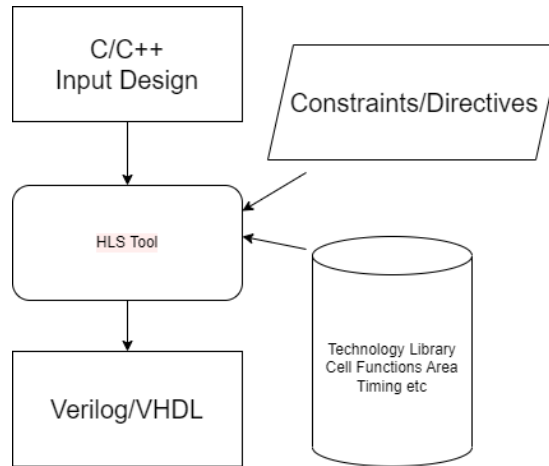


Figure 1.4: HLS Synthesis Workflow

1.9 FPGAs for Neural Networks

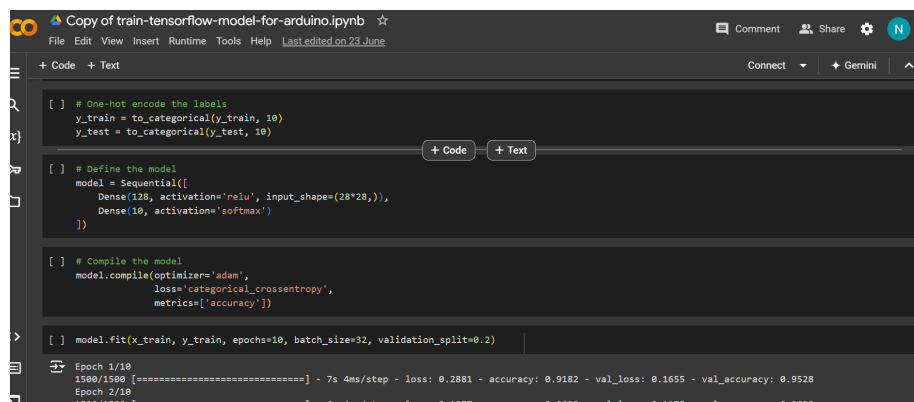
FPGAs present a compelling option for neural network implementations due to their flexibility, performance, and efficiency. They are particularly suited for applications requiring real-time processing, low latency, and energy efficiency. With ongoing advancements in FPGA technology and development tools, their adoption in neural network applications is expected to grow, bridging the gap between general-purpose processors and dedicated hardware accelerators.

Chapter 2

Neural Network Implementation - Making a Model

2.1 Model Training

First step is to train a machine learning model using tools like Jupyter Notebook, Google Colab etc. In this project, we have used Google Colab for training a tensor flow model for handwritten digit recognition as shown in figure 2.1.



```
[ ] # One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

[ ] # Define the model
model = Sequential([
    Dense(128, activation='relu', input_shape=(28*28,)),
    Dense(10, activation='softmax')
])

[ ] # Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

[ ] model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2)

Epoch 1/10
1500/1500 [=====] - 7s 4ms/step - loss: 0.2881 - accuracy: 0.9182 - val_loss: 0.1655 - val_accuracy: 0.9528
Epoch 2/10
1500/1500 [=====] - 6s 4ms/step - loss: 0.1277 - accuracy: 0.9628 - val_loss: 0.1175 - val_accuracy: 0.9638
```

Figure 2.1: Model Training in Google Colab

2.1.1 Training Data

For training, we have used MNIST dataset. The MNIST (Modified National Institute of Standards and Technology) dataset is a benchmark dataset widely

used in the field of machine learning and computer vision, specifically for handwritten digit recognition. It consists of images of handwritten digits from 0 to 9 and serves as a standard for evaluating and comparing the performance of various algorithms and models. It consist of images as shown in figure 2.2.

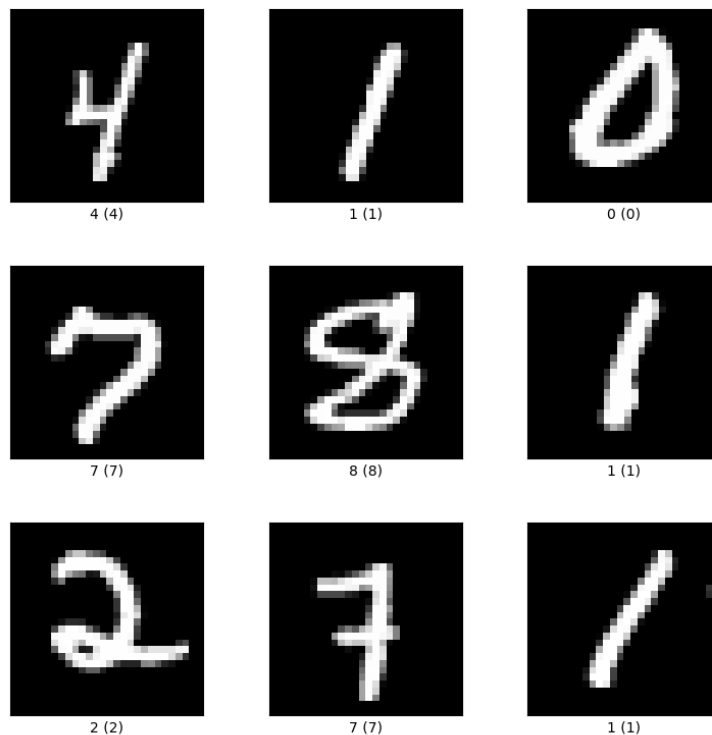


Figure 2.2: MNIST Dataset Sample

2.2 Structure of Model

The model consists of two dense (fully connected) layers. The first layer has 128 hidden units, and the second layer has 10 output units corresponding to 10 possible classes. The network architecture is as follows:

- Input layer: 784 units (28x28 pixels)

- Dense layer 1: 128 units
- ReLU activation
- Dense layer 2: 10 units
- Softmax activation

2.3 Extraction of weights and biases from the network

Using the below Python Code, We succesfully extracted the weights and biases from the trained model. These weights and biases were then used in Vivado HLS[4] C++ code.

```
# Save the trained model weights and biases
hidden_weights , hidden_biases = model.layers [1].get_weights ()
output_weights , output_biases = model.layers [2].get_weights ()

# Save the weights and biases to a file
np.savez ( 'mnist_weights_biases.npz' ,
          hidden_weights=hidden_weights ,
          hidden_biases=hidden_biases ,
          output_weights=output_weights ,
          output_biases=output_biases )
```

Chapter 3

Neural Network Implementation - High Level Implementation in Vivado HLS

3.1 Functions Implementations

Following sections shows the code for implementation of the functions present in the model.

3.1.1 Exponential Approximation

We begin with the exponential function. The exponential function is approximated using a truncated Taylor series for computational efficiency:

```
data_t exp_approx(data_t x) {  
    const int num_terms = 10;  
    data_t result = 1.0;  
    data_t term = 1.0;  
  
    for (int i = 1; i < num_terms; ++i) {  
        term *= x / i;  
        result += term;  
    }  
    return result;  
}
```

3.1.2 ReLU Activation

The ReLU activation function is applied element-wise to the output of the first dense layer:

```
void relu(data_t input[HIDDEN_SIZE], data_t output[HIDDEN_SIZE]) {  
    for (int i = 0; i < HIDDEN_SIZE; i++) {  
        output[i] = input[i] > 0 ? input[i] : 0;  
    }  
}
```

3.1.3 Softmax Activation

The softmax function normalizes the output of the second dense layer to a probability distribution:

```
void softmax(data_t input[OUTPUT_SIZE], data_t output[OUTPUT_SIZE]) {  
    data_t max_val = input[0];  
    for (int i = 1; i < OUTPUT_SIZE; i++) {  
        if (input[i] > max_val) {  
            max_val = input[i];  
        }  
    }  
  
    data_t sum = 0;  
    for (int i = 0; i < OUTPUT_SIZE; i++) {  
        output[i] = exp_approx(input[i] - max_val);  
        sum += output[i];  
    }  
  
    for (int i = 0; i < OUTPUT_SIZE; i++) {  
        output[i] /= sum;  
    }  
}
```

3.2 Layers Implementations

Following sections shows the code for implementation of the layers present in the model.

3.2.1 Dense Layers

The dense layers are implemented as follows:

```
void dense_1(data_t input[INPUT_SIZE], data_t output[HIDDEN_SIZE],  
const data_t weights[INPUT_SIZE][HIDDEN_SIZE], const data_t bias[HIDDEN_SIZE]) {
```

```

    for (int i = 0; i < HIDDEN_SIZE; i++) {
        data_t sum = bias[i];
        for (int j = 0; j < INPUT_SIZE; j++) {
            sum += input[j] * weights[i][j];
        }
        output[i] = sum;
    }
}

void dense_2(data_t input[HIDDEN_SIZE], data_t output[OUTPUT_SIZE],
const data_t weights[HIDDEN_SIZE][OUTPUT_SIZE], const data_t bias[OUTPUT_SIZE])
    for (int i = 0; i < OUTPUT_SIZE; i++) {
        data_t sum = bias[i];
        for (int j = 0; j < HIDDEN_SIZE; j++) {
            sum += input[j] * weights[i][j];
        }
        output[i] = sum;
    }
}

```

3.3 Top-level Function

The top-level function integrates the layers and activation functions:

```

void neural_network(data_t input[INPUT_SIZE], data_t output[OUTPUT_SIZE],
const data_t dense_1_weights[INPUT_SIZE][HIDDEN_SIZE],

const data_t dense_2_weights[HIDDEN_SIZE][OUTPUT_SIZE],

#pragma HLS INTERFACE m_axi port=input offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=dense_1_weights offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=dense_1_bias offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=dense_2_weights offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=dense_2_bias offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=return bundle=control

data_t dense_1_out[HIDDEN_SIZE];
data_t relu_out[HIDDEN_SIZE];
data_t dense_2_out[OUTPUT_SIZE];

// Layer 1: Dense
dense_1(input, dense_1_out, dense_1_weights, dense_1_bias);

```

```
// Layer 1: ReLU Activation
relu(dense_1_out , relu_out);

// Layer 2: Dense
dense_2(relu_out , dense_2_out , dense_2_weights , dense_2_bias);

// Layer 2: Softmax Activation
softmax(dense_2_out , output);
}
```

Chapter 4

Testing and Verification

This chapter describes the testing of the neural network on the FPGA. In this phase, we write a testbench code for testing the design under test which is our neural network for recognizing handwritten digits.

4.1 Testbench

Below is the code for testbench for our project design.

```
#include <iostream>
#include <cmath>
#include "mnist_image_array.h"

#define INPUT_SIZE 784 // 28 * 28
#define HIDDEN_SIZE 128
#define OUTPUT_SIZE 10

typedef float data_t;

// Function prototypes
void neural_network(data_t input[INPUT_SIZE], data_t output[OUTPUT_SIZE],
                   const data_t dense_1_weights[INPUT_SIZE][HIDDEN_SIZE], const
                   const data_t dense_2_weights[HIDDEN_SIZE][OUTPUT_SIZE], cons

// Helper function to print the output array
void print_output(data_t output[OUTPUT_SIZE]) {
    float max = 0;
    int index, i;
    for ( i = 0; i < OUTPUT_SIZE; i++) {
        if(output[i] > max){
            max = output[i];
            index = i;
        }
    }
}
```



```

    }
    std::cout << "Output[" << i << "]" <=" " << output[i] << std::endl;
}
    std::cout << "Predicted-Number=" << index << " -with-P=" << max << std::
}

int main() {
    // Declare and initialize input data in .h

    // Declare and initialize output array
    data_t output[OUTPUT_SIZE];

    // Declare and initialize dense_1_weights and dense_1_bias
    const data_t dense_1_weights[INPUT_SIZE][HIDDEN_SIZE] = {..};
    const data_t dense_1_bias[HIDDEN_SIZE] = {..};

    // Declare and initialize dense_2_weights and dense_2_bias
    const data_t dense_2_weights[HIDDEN_SIZE][OUTPUT_SIZE] = {..};

    const data_t dense_2_bias[OUTPUT_SIZE] = {...};

    // Call the neural network function
    neural_network(image, output, dense_1_weights, dense_1_bias, dense_2_weights

    // Print the output
    print_output(output);

    return 0;
}
}

```

4.2 Breakdown of testbench code

Here is a brief summary of the testbench code, we wrote for testing the project design.

- The **main** function starts by declaring and initializing the input data, output array, and weights and biases for both dense layers.
- The input data is assumed to be declared and initialized in the included header file.
- The weights and biases are initialized to empty arrays here to save space.
- The **neural_network** function is called with the input image and weight/bias parameters.

- The output of the neural network is then printed using the `print_output` function.

4.3 Inputting Image to Model

Using a Python script, we were able to convert a 28x28 image to a 784 elements, one dimensional array.

```
from PIL import Image
import numpy as np

# Function to process image and save as C array
def image_to_c_array(image_path, output_path):
    # Open the image file
    img = Image.open(image_path).convert('L') # Convert to grayscale

    # Resize image to 28x28 if it's not already
    img = img.resize((28, 28))

    # Convert image to numpy array and normalize pixel values
    img_array = np.array(img) / 255.0

    # Flatten the array
    flat_array = img_array.flatten()

    # Convert to C array format
    c_array_str = ','.join(map(str, flat_array))
    c_array = f"float image[784] = {{ {c_array_str} }};"

    # Write to output file
    with open(output_path, 'w') as file:
        file.write(c_array)

    print(f"C array saved to {output_path}")
```

Chapter 5

Results

The performance and accuracy of the implemented neural network are evaluated. This chapter includes the testing methodology, performance metrics, and comparison with other implementations.

5.1 Performance Metrics

The performance of the network is measured in terms of latency, throughput, and resource utilization on the FPGA. The results are summarized in the following figure 5.1. By doing some approximations, we were able to decrease the LUT utilization to almost 50%. We changed the datatype from float to 16 bit `ap_fixed` in which only 8 bits were reserved for the floating point and rest 8 for the integer part.

5.2 Output of Testbench in Vivado HLS

Figure 5.2 shows the output of project design in Vivado HLS for the input image shown in 5.3.

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48A	FF	LUT
Expression	-	-	0	48
FIFO	-	-	-	-
Instance	-	15	4817	5809
Memory	2	-	-	-
Multiplexer	-	-	-	120
Register	-	-	243	-
Total	2	15	5060	5977
Available	32	16	11440	5720
Utilization (%)	6	93	44	104

Figure 5.1: Summary of the Utilization Estimates of the Project Design

```

Vivado HLS Console
Output[0] = 0.263051
Output[1] = -0.000982068
Output[2] = 0.0866956
Output[3] = 0.23408
Output[4] = 0.332029
Output[5] = 0.00996678
Output[6] = 0.0821274
Output[7] = 0.001479
Output[8] = -0.000300867
Output[9] = -0.00814611
Predicted Number = 4 with P = 0.332029
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [VIVADO_HLS]

```

Figure 5.2: Output for Handwritten Image of 4



Figure 5.3: Input Image for 4

Chapter 6

Conclusion

This report demonstrated the implementation of a neural network on an FPGA using HLS. The design achieved efficient performance by leveraging hardware parallelism and custom approximations for mathematical functions. Future work includes optimizing the design further and extending it to more complex networks. We can export the generated RTL to ISE using the Export RTL in Vivado HLS. In this way, we can implement this neural network on FPGA hardware.

References

- [1] <https://aws.amazon.com/what-is/neural-network>
- [2] <https://fpgatutorial.com/introduction-to-fpga-development/>
- [3] <https://hardwarebee.com/understanding-fpga-logic-synthesis/>
- [4] <https://www.xilinx.com/products/design-tools/vivado/high-level-design.html>