I, Thomas Anderson, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
Built-In Self Training of Hardware-Based Neural Networks

Student's name: Thomas Anderson

This work and its defense approved by:

Committee chair: Wen-Ben Jone, Ph.D.

Committee member: Ali Minai, Ph.D.

Committee member: Ranganadha Vemuri, Ph.D.

20107
Built-In Self Training of Hardware-Based Neural Networks

A thesis submitted to the
Division of Research and Advanced Studies
of the
University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of
Electrical Engineering and Computer Science
of the College of Engineering and Applied Science
November 16, 2017

by
Thomas Anderson
BS Electrical Engineering, University of Cincinnati, April 2017

Thesis Advisor and Committee Chair: Dr. Wen-Ben Jone
Abstract

Artificial neural networks and deep learning are a topic of increasing interest in computing. This has spurred investigation into dedicated hardware like accelerators to speed up the training and inference processes. This work proposes a new hardware architecture called Built-In Self Training (BISTr) for both training a network and performing inferences. The architecture combines principles from the Built-In Self Testing (BIST) VLSI paradigm with the backpropagation learning algorithm. The primary focus of the work is to present the BISTr architecture and verify its efficacy.

The development of the architecture began with analysis of the backpropagation algorithm and the derivation of new equations. Once the derivations were complete, the hardware was designed and all of the functional components were tested using VHDL from the bottom to top level. An automatic synthesis tool was created to generate the code used and tested in the experimental phase. The application tested during the experiments was function approximation. The new architecture was trained successfully for a couple of the test cases. The other test cases were not successful, but this was due to the data representation used in the VHDL code and not a result of the hardware design itself. The area overhead of the added hardware and speed performance were analyzed briefly. The results showed that: (1) the area overhead was significant (around 3 times the area without the additional hardware) and (2) the theoretical speed performance of the design is very good.

The new BISTr architecture was proven to work and had a good theoretical speed performance. However, the architecture presented in this work cannot be implemented for large neural networks due to the large amount of area overhead. Further work would be required to expand upon the idea presented in this paper and improve it: (1) development of an alternative design that is more practical to implement, (2) more rigorous testing of area and speed, (3) implementation of other training methods and functionality, and (4) additions to the synthesis tool to increase its capability.
For Katie
Acknowledgments

My sincerest thanks go out to my advisor Dr. Wen-Ben Jone. He was always there to encourage me and keep pushing me forward. Dr. Jone was always available to talk about any issues I encountered during this work. I especially want to thank him for accommodating my move away from campus and working with me remotely on the thesis. His guidance and advice were instrumental in how the thesis turned out.

I would also like to extend my thanks to my committee members Dr. Ranganadha Vemuri and Dr. Ali Minai for reviewing and providing feedback on this work. There was a lot of content to go over, so I really appreciate the time and effort they put forth.

Major thanks to my sister Katie and my parents David and Joyce for their moral support and love. They’ve been behind me every step of the way cheering me on and picking me up when I was down. I also want to thank my parents for proofreading my thesis for me.

Thank you all.
## Contents

1 Introduction ................................................. 1

2 Background ............................................... 4
   2.1 Artificial Neural Networks .......................... 4
   2.2 Training and the Backpropagation Algorithm ...... 6
   2.3 BIST VLSI Paradigm .................................. 7
   2.4 Function Approximation ............................... 8

3 Fundamental Concept ...................................... 12
   3.1 Automating the Learning Process .................... 12
   3.2 Basic Data Flow in the BISTr Paradigm .......... 13
   3.3 BISTr Controller ...................................... 16
   3.4 Data path and Controller Integration ............... 16

4 Theoretical Analysis ...................................... 17
   4.1 A Mathematical Presentation of Backpropagation .. 17
      4.1.1 Training Set ..................................... 17
      4.1.2 Performance Index ............................... 17
      4.1.3 Feedforward Propagation ....................... 18
      4.1.4 Steepest Descent ................................. 19
      4.1.5 Chain Rule for Calculating Derivatives ...... 20
      4.1.6 Definition of Sensitivity ....................... 21
      4.1.7 Updated Steepest Descent Algorithm .......... 21
      4.1.8 Backpropagation of the Sensitivities .......... 21
      4.2 Adapting the Equations to the BISTr Paradigm .. 22
         4.2.1 Forward Propagation in BISTr ............... 22
         4.2.2 Backpropagation in BISTr ..................... 23
         4.2.3 Weight Updating in BISTr ..................... 25
      4.3 Training Iteration Example ........................ 27

5 Detailed Design Overview ............................... 32
   5.1 Subnode Architectures ............................... 33
      5.1.1 Weight Subnode Architecture .................... 33
      5.1.2 Bias Subnode Architecture ....................... 36
5.2 Neuron Architecture .......................................................... 37
  5.2.1 Feedforward Behavior .................................................... 38
  5.2.2 Backpropagation Behavior ............................................. 40
  5.2.3 Weight Updating Behavior ............................................ 40
  5.2.4 Additional Considerations ........................................... 41
5.3 Data Path ......................................................................... 41
  5.3.1 Expected Output Scan .................................................... 41
  5.3.2 Feedforward Propagation Mode ...................................... 43
  5.3.3 Backpropagation Mode .................................................. 45
  5.3.4 Weight Updating Mode .................................................. 45
5.4 BISTr Controller ................................................................. 46
5.5 Boundary and Internal Scan Chains ..................................... 49
  5.5.1 Internal Scan Chain ....................................................... 50
  5.5.2 Boundary Scan Chain .................................................... 51
5.6 Output Sensitivity Generators ............................................ 52
5.7 Alternative Design involving DRAM and Buffer ..................... 53

6 Automatic Network Synthesis Tool ....................................... 56
  6.1 Synthesis Tool Overview ................................................... 56
    6.1.1 Introduction .............................................................. 56
    6.1.2 Program Inputs and Outputs ....................................... 57
    6.1.3 Program Flow ........................................................... 57
  6.2 Network File Structure ...................................................... 63
    6.2.1 Header Format .......................................................... 63
    6.2.2 Layer Format ............................................................ 63
  6.3 Training File Structure ...................................................... 64
    6.3.1 Header Format .......................................................... 64
    6.3.2 Training Commands ................................................... 65
  6.4 Example ........................................................................... 68
  6.5 How to Use the Synthesis Tool .......................................... 73

7 Experimental Results .......................................................... 74
  7.1 Application – Function Approximation ............................... 74
    7.1.1 Arctangent Function ................................................... 75
    7.1.2 Cubic Function ......................................................... 75
    7.1.3 Quadratic Function ..................................................... 75
    7.1.4 Absolute Value Times Sine Function .............................. 76
    7.1.5 Sine Plus X Function .................................................. 76
  7.2 MATLAB Simulation Results .............................................. 77
    7.2.1 Arctangent Function ................................................... 77
    7.2.2 Cubic Function ......................................................... 79
    7.2.3 Quadratic Function ..................................................... 81
    7.2.4 Absolute Value Times Sine Function .............................. 83
    7.2.5 Sine Plus X Function .................................................. 85
    7.2.6 Numerical Results ...................................................... 87
7.3 VHDL Simulation Results ........................................ 87
  7.3.1 Arctangent Function ....................................... 88
  7.3.2 Cubic Function ........................................... 90
  7.3.3 Quadratic Function ....................................... 92
  7.3.4 Absolute Value Times Sine Function ...................... 94
  7.3.5 Sine Plus X Function ..................................... 96
  7.3.6 Numerical Results ....................................... 98

8 Discussion .................................................. 99
  8.1 Experimental Results ....................................... 99
  8.2 Circuit Design ............................................. 104
  8.3 Area and Performance Results .............................. 106

9 Conclusion and Future Research .............................. 107
  9.1 Conclusion ................................................ 107
  9.2 Future Research .......................................... 108
    9.2.1 Additions to the Synthesis Tool ....................... 108
    9.2.2 Alternative Design with DRAM and Buffer ............ 109
    9.2.3 Other Training Methods and Functionality ........... 109
    9.2.4 Area and Speed Performance .......................... 110
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Mathematical representation of an artificial neuron [23]</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>High level view of a simple neural network</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Output of the basic sine function from [-2, 2]</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Neural network used to approximate the basic sine function</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Neural network approximation before backpropagation training</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>Neural network approximation after backpropagation training</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Forward data flow in a BISTr neural network</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Backpropagation data flow in a BISTr neural network</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Weight updating in a BISTr neural network</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Weight naming convention between two layers of a neural network</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Hardware for feedforward operation of BISTr</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>Hardware for backpropagation of sensitivities in BISTr</td>
<td>25</td>
</tr>
<tr>
<td>4.4</td>
<td>Alternate hardware for backpropagation in BISTr</td>
<td>26</td>
</tr>
<tr>
<td>4.5</td>
<td>Hardware for weight updating in BISTr</td>
<td>26</td>
</tr>
<tr>
<td>4.6</td>
<td>Hardware for bias updating in BISTr</td>
<td>27</td>
</tr>
<tr>
<td>4.7</td>
<td>Sine function approximation network</td>
<td>29</td>
</tr>
<tr>
<td>4.8</td>
<td>Network response before training process</td>
<td>29</td>
</tr>
<tr>
<td>4.9</td>
<td>Network response after training process</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>Top level box diagram of a complete BISTr neural network</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Weight subnode architecture circuit diagram</td>
<td>34</td>
</tr>
<tr>
<td>5.3</td>
<td>Bias subnode architecture circuit diagram</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Neuron architecture circuit diagram</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Summation unit in a neuron with supplementary MUX units</td>
<td>42</td>
</tr>
<tr>
<td>5.6</td>
<td>Modules and lines active during the EO scan mode of training</td>
<td>43</td>
</tr>
<tr>
<td>5.7</td>
<td>Modules and lines active during the feedforward mode of training</td>
<td>44</td>
</tr>
<tr>
<td>5.8</td>
<td>Modules and lines active during the backpropagation mode of training</td>
<td>44</td>
</tr>
<tr>
<td>5.9</td>
<td>Modules and lines active during the weight updating mode of training</td>
<td>45</td>
</tr>
<tr>
<td>5.10</td>
<td>BISTr controller state diagram</td>
<td>48</td>
</tr>
<tr>
<td>5.11</td>
<td>Diagram showing the internal scan chain organization</td>
<td>50</td>
</tr>
<tr>
<td>5.12</td>
<td>Boundary scan chain and output sensitivity generators</td>
<td>52</td>
</tr>
<tr>
<td>5.13</td>
<td>Alternative chip design with added memory buffer and external DRAM</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>Program inputs and outputs</td>
<td>57</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Weight subnode operation .............................................. 35
7.1 Table of numerical results for all MATLAB simulations ........... 87
7.2 Table of numerical results for all MATLAB simulations ........... 98
Chapter 1

Introduction

Significant research advances have been made in the topics of artificial neural networks (ANNs) and deep learning in recent years [1, 2, 3, 4, 5]. As computing technology has improved and been developed, more and more complex neural networks and artificial intelligence have been created. As the networks become more capable, the number of applications in which deep learning are used has increased. Some of the applications in which ANNs and AI are used include medical diagnosis, autonomous vehicle control, stock market prediction, pattern and sequence recognition, playing games like chess, and social network filtering [1, 2, 3, 4, 5]. The area continues to be of great interest in research. Due to the greater demand from more complex neural networks and AI, meeting the computing needs of neural networks is a pertinent topic in the field. In recent years, research has been done into how dedicated hardware can help solve some of the issues facing neural networks and deep learning.

One of the major concerns in deep learning is the training of the neural networks [6, 7, 8, 9, 10]. Training is integral to AI and neural networks. The ANNs cannot perform their intended task without being trained first. Depending on the complexity of the network being trained, training can take a very long time. Since networks are becoming increasingly complex, the time required to train them is also growing. Before advances in training optimization, the process could take months [16]. Even though training times have improved to a matter of days, there is still room for improvement. Training time remains one of the major criticisms leveled at artificial neural networks and deep learning. Reducing the length of the training process is critical to increasing the commercial appeal of ANNs and deep learning outside of research.

Hardware solutions are being investigated to address this issue [11, 12, 13, 14, 15]. Both the training process and the normal operation of the network (also called making inferences) require significant computing resources [16]. This includes everything from memory and interconnects to the units processing the data. The emergence of General Purpose Graphics Processing Units (GPGPUs) has increased the amount of parallel computing power available for neural network applications [11, 12]. Multiple GPGPUs operating in parallel are a potent
tool in training and evaluating complex neural networks [17]. While the power of general purpose hardware has improved, some research is targeting application-specific hardware dedicated entirely to neural networks. One such neural network processor is the Tensor Processing Unit (TPU) developed by Google [18]. Dedicated hardware solutions like the TPU and other NN accelerators have the potential to push neural network computing even farther.

This work proposes a new, hardware-based neural network architecture called Built-In Self Training (or BISTr for short) for training and performing inferences. The architecture adapts ideas from the BIST VLSI paradigm to supplement the existing hardware (used for performing inferences) to add the training capability [19, 20, 21]. This additional hardware allows the existing hardware used for forward propagation of data (during inferences) to be reused for backpropagation and weight updating. In addition to the hardware inserted in the network itself, the design incorporates more hardware outside the network to make the training possible. The most notable of the hardware external to the network is the BISTr controller, which controls the training hardware and makes the interface easy for the user.

The modules in the architecture were designed by hand. Accurate function of the modules was tested using the VHSIC Hardware Description Language (or VHDL) [22]. Each module was simulated using VHDL to verify correct behavior. The modules in the design were tested from the bottom level up to the top. This includes everything from registers and multiplexers to an individual neuron. Once the individual behavior of the modules was verified, all of the modules were put together to simulate the behavior of an entire (albeit simple) neural network. This includes all the additional BISTr hardware required to make the training work. This simulation verified that the entire design was working properly.

Before moving into the experimental phase of the work, a code synthesis tool was created that can automatically generate the code to represent the network in VHDL. The tool was developed with the intent to make the experimental process easier and more fluid. The synthesis tool takes an input file describing the structure of the network and then produces the VHDL files containing the entire network code. The tool also handles hooking up all of the wires between the modules in the design. Additionally, the synthesis tool can generate a testbench file that can perform the training process in a VHDL simulator. This testbench is generated based on another file containing instructions to carry out during the simulation.

The experiments used to test the design involved training neural networks of various sizes to approximate the output of mathematical functions. The VHDL simulations did not produce accurate training results for most of the cases tested. The cause of the issue was thoroughly investigated and the findings are presented in Chapter 8. The problem stemmed from the data representation used in the VHDL simulations and not from any error in the hardware design or function. The results showed that the design can work, but needs modifications to be more effective. Most notably, a better data representation could be used to address this issue.

Though it was not the primary focus of the work, some analysis of the area and speed
performance of the design was performed. Due to the small size of the network tested, the area analysis revealed that the area of a network with the BISTr hardware added was about 3 times the area of the same network without the additional hardware. Such an area overhead is prohibitive to physical implementation of the design. As the design is presented in this thesis, only small to medium size neural networks could be implemented physically on a chip. The speed performance of the design is great. However, the issue of the physical implementation prevents this performance from being realistically achievable. An alternative design discussed briefly could potentially be implemented, but would be slower. Future research suggestions are provided at the end of the thesis to address the issues with the current design and provide ways in which it can be augmented.

The chapters in this thesis are organized as follows:

Chapter 2: reviews background information about artificial neural networks, training algorithms, the BIST VLSI paradigm, and function approximations.

Chapter 3: briefly explains the fundamental concept of the work.

Chapter 4: provides a detailed description of the backpropagation training algorithm how the algorithm was adapted for the design, and a training example.

Chapter 5: shows the proposed design in detail, including circuit diagrams, signal definitions, functional descriptions, and the data flow of the design.

Chapter 6: presents the automatic synthesis tool program developed for the work, explaining how it works, the input file specifications, and how to use it.

Chapter 7: describes the experiments performed to test the design and provides the results of the experiments.

Chapter 8: discusses the results of the experiments, design choices, how issues with design can be fixed, and the area/speed performance of the design.

Chapter 9: provides the conclusions of this work and presents possible future work.
Chapter 2

Background

This thesis combines knowledge from a few different areas. These include artificial neural networks (ANNs), the Built-In Self Testing (BIST) VLSI paradigm, and mathematical function approximation. The section will provide a background in each of these areas sufficient enough to understand the content of this thesis.

2.1 Artificial Neural Networks

The human brain contains millions of cells called neurons. These neurons are densely interconnected with each other using structures known as synapses. A neuron receives inputs from other neurons via its dendrites. If the neuron receives enough stimulus from its dendrites, it will activate (also referred to as firing) and send a signal using its output axon. This axon connects to other neurons’ dendrites at the aforementioned synapses. In this way, neurons activate and cause each other to fire as we think or process sensory information. The connections between neurons in our brains are constantly changing. The strength of the connections between neurons in the brain is altered when forming new memories or learning a new skill. When learning how to ride a bike, the neurons responsible for the coordination and motor skill will become more tightly connected. In the future when the person rides a bike again, the corresponding neurons will start to fire. Due to the strong connection of the neurons, the person will be able to ride the bike effortlessly.

Artificial neural networks (ANNs) attempt to replicate the way the brain works to solve various computing tasks. Some of these tasks include function approximation, classification tasks such as pattern or image recognition, stock market prediction, data processing, and more. The fundamental building block of ANNs is the artificial neuron. Figure 2.1 shows the diagram of a basic artificial neuron.

The artificial neuron contains many of the same parts as biological neurons in the brain.
Axons from neurons that serve as inputs to the neuron in question are connected to the input dendrites of the neuron cell body via synapses. The strength of the synaptic connection is modeled by associating a weight to the connection between the axon and dendrite. From a mathematical standpoint, the weight (e.g. $w_0$) is multiplied by the input value ($x_0$), and the result is passed to the cell body. All of the weighted inputs are then summed up to produce a net sum. The net sum can be described using the following equation (where $m$ is the number of inputs):

$$n = \sum_{i=1}^{m} (w_i x_i) + b$$  \hfill (2.1)

The net sum is then passed through a block called the activation function. This function operates on the net sum and produces the final output that is sent to the output axon. Depending on the application of the neural network the neuron is in, a number of activation functions can be used here. Some of these functions include linear, rectified linear, and logistic. More information on activation functions can be found here. The equation for the neuron output after the activation function is as follows:

$$a = f(n) = f\left(\sum_{i=1}^{m} (w_i x_i) + b\right)$$  \hfill (2.2)

These artificial neurons can be connected together to form an artificial neural network. Figure 2.2 shows a high level view of what an artificial neural network looks like. Neurons are generally organized into layers. The final layer of the neural network is typically called the output layer. Any layers before the output layer are called hidden layers because their direct output is not seen. The first layer of an ANN is sometimes referred to as the input layer. An ANN can have any number of layers and each layer in the ANN can have any number of neurons (as long as there is at least one). Typically, the output of every neuron
in one layer is connected to an input of every neuron in the next layer. A network in which every layer is completely connected to the next is referred to as a fully-connected network. The networks used in this work are all fully-connected ANNs. A lack of connection between two neurons can be simulated in a fully-connected network by having a weight of zero on the edge (or synapse) between them.

Figure 2.2: High level view of a simple neural network

2.2 Training and the Backpropagation Algorithm

Neural networks do not start out able to perform the tasks they are used for. Just like the human brain, they must be trained to perform these tasks. The weights and biases in the artificial neural network are initialized with small random values (typically centered around 0). This is done to prevent all the weights from being symmetrical. Additionally, starting with random weights generally produces better results than using a predetermined set. During the training process, the weights and biases are adjusted until the network produces a desirable result. This is usually when the output gets within an acceptable range of the desired result.

Numerous training algorithms exist for teaching artificial neural networks. Each algorithm operates differently and has its own pros and cons. Training algorithms are generally divided into one of three categories: supervised, unsupervised, and reinforcement learning. In unsupervised algorithms, the test inputs are provided to the network with no classification or categorization [24]. In other words, the network learns without the accuracy of its results being evaluated. A couple examples of unsupervised learning are clustering and Heb-
bian learning. *Supervised learning* contrasts this by providing what the expected or desired outputs are for a given set of inputs [25]. The response of the network is compared to the expected value and the error is used to alter the weights and biases. There are quite a few supervised learning algorithms including backpropagation, which is the main focus of this thesis. *Reinforcement training* differs from supervised training in that the goal is not to find an optimal performance or value. This type of learning explores the trade-off between utilizing the knowledge possessed by the network and exploring unknown territory to gain new knowledge [26].

The *backpropagation* algorithm is a supervised learning algorithm. It involves comparing the response of the network to the expected value for that set of inputs. The error between the expected and actual output values is then propagated backwards through the network and used to adjust the values of the weights. One key parameter in the backpropagation algorithm is the *learning rate*, represented in this thesis using the Greek letter alpha (\(\alpha\)). The learning rate controls how much the weights and biases are adjusted when they are updated. The values of the learning rate are typically small decimals (around 0.05 to 0.2). A detailed breakdown of the backpropagation algorithm (including the equations involved) is provided in Section 4.1.

### 2.3 BIST VLSI Paradigm

The Built-In Self Testing or BIST paradigm in VLSI design was created to improve the testability of VLSI integrated circuits (ICs) [19, 20, 21]. Testing of very large scale circuits can be both costly and time consuming. BIST adds hardware to an existing IC design to allow the circuitry to test itself. Since the hardware on the chip is handling some of the testing, the external test equipment used to test can be simpler and less expensive. One way BIST achieves this is by reducing the number of I/O signals that have to be driven or examined by the test equipment. BIST also has the added benefit of reducing the duration of test cycles (since the at-speed on-chip signals are faster than test signals coming from off-chip) [19, 20, 21].

BIST circuitry is divided into two categories, on-line and off-line hardware. *On-line* BIST hardware is added to the original circuit called the circuit under test (CUT) [19, 20, 21]. It can be inserted into the original design or alongside it. On-line hardware either operates concurrently with normal operation or non-concurrently when the CUT is idle. *Off-line* BIST hardware are external modules that can interact with the CUT by providing test patterns to the primary inputs or capturing the primary outputs [19, 20, 21]. This hardware is not active during normal operation. BISTr uses non-concurrent on-line hardware and off-line hardware in its design.

One major aspect of testing that BIST covers is test pattern generation (TPG). Without BIST, the external test equipment would have to provide the test patterns. One common way to generate test patterns in BIST is to use a pseudorandom number generator (PRNG).
PRNGs are typically implemented using a linear feedback shift register (LFSR). BISTr does not use automatic test pattern generation due to the nature of the backpropagation algorithm. Memory scan chains are a feature from BIST that are used in the BISTr design. A scan chain involves linking together D flip-flops, registers or other memory elements in a design for testing purposes. Scan chains can be used to (1) test the memory elements in the design for defects, (2) test intermediate sections of the circuit by scanning in desired values, and (3) scan out intermediate test values before the final output is produced. Scan elements can be controlled by the BIST hardware to accept inputs from either the scan chain or their normal source in the CUT. BISTr has two separate scan chains, each with its own purpose. Another major aspect of testing handled by BIST is output response analysis. These modules (called output response analyzers or ORAs) check the primary outputs for correctness when test patterns are applied. They may also compress the result of the analysis so it is easier to interpret. BISTr does not contain any ORAs that perform the function described here, but a module in BISTr was inspired by the ORAs used in BIST.

2.4 Function Approximation

In this thesis, function approximation will be used to refer specifically to the approximation of mathematical functions and not the broader topic. Function approximation involves taking a mathematical function (e.g. \( y = x^2 - 7x + 10 \) or \( y = 1 + \sin\left(\frac{\pi}{4}x\right) \)) and approximating the value of the output \( y \) using some means. In this work, the means of approximating the functions will be the neural networks. Function approximation is typically only done over a certain range of inputs (e.g. \([1,6]\) or \([0,2]\)). There are multiple reasons for this. One is that many functions such as polynomials, exponentials, and some trigonometric functions do not converge to a single value when \( x \) approaches \( \infty \) or \( -\infty \). The other is a limitation of how the approximation is calculated. For neural network approximations, increasing the range of the approximation also tends to increase the number of neurons required to perform the approximation.

The goal of function approximation with neural networks is to get the output of the network as close as possible to the output of the actual mathematical function being approximated. The actual output of the function is used as the target or expected output for the neural network to reach. During the training process, the error between the network response and the actual function is used to adjust the weights and biases in the network so the response becomes more accurate. One major consideration when approximating functions with neural networks is the size of the network being used. If too few neurons are used, the approximation may not be good. Using more neurons is typically better, but using too many neurons can sometimes produce worse results (called over-fitting). Finding the right network structure for approximating a given function requires some experimentation.

To help illustrate the idea of function approximation, a brief example will be provided
The goal is to approximate the following basic sine function using a neural network:

\[ y(x) = 1 + \sin \left( \frac{\pi}{4} x \right) \quad \text{for} \quad -2 \leq x \leq 2 \]  

(2.3)

Figure 2.3 shows the output of the basic sine function over the range [-2, 2]. The function will be approximated using the neural network shown in Figure 2.4. The network has two layers, the first having two neurons and the second having one. The activation functions for the neurons in the first layer is the logistic function. The neuron in the second layer has a linear activation function. The network has a single input (denoted as \( p \) in the figure) and a single output (\( a^2_1 \)).

Figure 2.3: Output of the basic sine function from [-2, 2]

Figure 2.4: Neural network used to approximate the basic sine function

Figure 2.5 shows the response of the network using randomized initial weights before training has been performed. The network response does not resemble the sine function.
value at all. The backpropagation learning algorithm is applied by running through the range of the approximation \([-2,2]\) multiple times (100 to be precise) and adjusting the weights and biases until the training is complete. The network response after the training is shown in Figure 2.6. After the training, the response of the network is very close to the basic sine function. A much more rigorous example can be found in Section 4.3 which includes the mathematical calculations involved in the backpropagation algorithm.

Figure 2.5: Neural network approximation before backpropagation training
Figure 2.6: Neural network approximation after backpropagation training
Chapter 3

Fundamental Concept

3.1 Automating the Learning Process

The training of a neural network can be broken down into three main components: forward propagation of the inputs through the network, backpropagation of the sensitivities used to update the weights, and the weight updating process itself. When considering translating these steps to a dedicated-hardware-based approach, forward propagation of the training inputs is handled in parallel by the arrays of multipliers, adders, and activation functions. However, some hardware applications may still be implemented by GPUs or CPUs, and therefore instruction executions would be used in place of dedicated hardware. The other two steps are typically where hardware and software implementations of training diverge. The mathematical matrix representation of a neural network is convenient to use conceptually, but it can only offer so much computationally. While matrix multiply-accumulate (MAC) or GPU vector processing can help speed up the training process in software, they still have drawbacks from a hardware-based network perspective. If the weights are stored locally on the neural network chip (and not loaded from DRAM or other external sources), then training must occur off chip (possibly with an accelerator) and then the weights must be loaded onto the chip. If a conventional load-store architecture is used for training, then the weights must be loaded, operated on, and then stored. These traditional software training methods do not leverage the hardware available as part of the hardware neural network.

One of the benefits of implementing neural networks with dedicated hardware is the wealth of resources the implementation provides, namely the multipliers and adders. The dedicated hardware is used during the forward propagation step, but goes unused during backpropagation and weight updating. This contrasts many neural network accelerators, which can use MAC units for both forward and backward propagation. The main goal of BISTr is to reuse the already available hardware to accomplish the backpropagation and weight updating steps much faster by exploiting the parallel hardware. When broken down, the backpropagation process involves only multiplication and summation. The weight up-
dating process is similar, requiring multiplication and subtraction. Both the multiplier and summation units are already present in the neural network hardware. Naturally, additional hardware must be added to make this reuse possible (specifics are covered in Chapter 5).

BISTr uses a divide and conquer approach to speed up the backpropagation and weight updating processes. For smaller networks (on the scale of hundreds to thousands of neurons), this is possible because the information needed to backpropagate the sensitivities and update the weights can be processed and stored locally at each node (neuron). For modern neural networks which contain millions of parameters (weights and biases), such an approach is not feasible. In these situations, the same approach as accelerators or GPUs implementations must be used, storing the parameters in DRAM or on disk and loading them into registers only when they are needed. The beauty of the design comes from the interconnectivity of the neural network and how that aids the backpropagation process. Chapter 4 provides a detailed breakdown of the equations and theory behind the approach. Factoring out time needed to load the parameters from DRAM, BISTr can calculate the sensitivity vector for any given layer in just one clock cycle. If the network had access to every weight and bias simultaneously, weight updating could be performed in only two clock cycles. As before, this is inviable if the parameters are stored in DRAM instead of inside the neurons themselves. The next section covers the basic data flow during one training cycle in the BISTr design.

### 3.2 Basic Data Flow in the BISTr Paradigm

The training cycle begins with normal forward propagation of the test inputs. At each neuron in each layer, the inputs are multiplied by their corresponding weight, summed, and then the sum goes through the activation function. For smaller networks, every input to each layer can be stored in a register to be used in the weight updating process later. In larger networks, the inputs to the layer would be stored externally (e.g. DRAM). Figure 3.1 below shows the forward data flow for a simple example network.

After the final output(s) of the network has been determined, the BISTr circuitry switches from forward propagation mode to backpropagation. In this mode, sensitivities based on the error between the final outputs and the expected results are propagated backwards through the network. These sensitivities will be used to change the weights in the next step. Figure 3.2 shows the backward propagation process for the same example network as in Figure 3.1.

The final step in the training process is the weight updating, shown in Figure 3.3. The weight updating happens individually at each neuron, instead of cascading through the network like the forward propagation or backpropagation steps. The weights are either stored locally at each neuron or they are stored in DRAM and then read into registers during the weight updating process. Alpha is the learning coefficient (supplied to each neuron). The input to the neuron and the sensitivity were calculated and stored during the forward propagation and backpropagation stages respectively.
Figure 3.1: Forward data flow in a BISTr neural network

Figure 3.2: Backpropagation data flow in a BISTr neural network
Figure 3.3: Weight updating in a BISTr neural network

\[ W^1(k + 1) = W^1(k) - \alpha \overrightarrow{p} \overrightarrow{s}^1 \]

\[ W^2(k + 1) = W^2(k) - \alpha \overrightarrow{a} \overrightarrow{s}^2 \]

\[ W^3(k + 1) = W^3(k) - \alpha \overrightarrow{a} \overrightarrow{s}^3 \]
After the weights are updated, the network is ready to begin another training cycle using either the same inputs or different inputs (e.g., images) or to carry out its normal operation if the training is finished. Controlling the BISTr circuitry is simple, and made even easier via the BISTr controller.

### 3.3 BISTr Controller

Using a controller to regulate the operation of the BISTr circuitry was adapted from the BIST VLSI paradigm. The BISTr controller is a finite state machine (FSM) responsible for issuing the proper control signals to configure the neurons to operate the corresponding modes. The user controls the BISTr circuitry by issuing a test mode select (TMS) signal to the BISTr controller. It also contains the ports via which the two scan chains can be accessed and utilized. The learning rate (coefficient) is input by the user into the BISTr controller. One key difference between the BIST and BISTr controllers is that the BISTr controller does not have the same boundary scan as BIST. Instead of the boundary scan covering the primary inputs and outputs as it does in BIST, the boundary scan in BISTr only serves to scan in and out the expected (target) outputs for the training process. The primary inputs and outputs aside, the BISTr controller serves as the main interface used to interact with the circuit. The design of the BISTr controller including the ports, state diagram, etc. is detailed in Section 5.4.

### 3.4 Data path and Controller Integration

The BISTr controller has minimal access to the main data path of the neural network. The two scan chains to which it can pass data are the Expected Output scan chain (used at the beginning of the backpropagation mode) and the Weight/Bias scan chain (which is along the data path). It supplies the learning coefficient to the neurons during weight updating. The BISTr controller can also reset all the registers in the design (including neuron inputs and sensitivities). The biggest impact the BISTr controller has on the data path and the flow of data through the circuit is controlling the direction of the flow. The nets between the neurons must be bi-directional since data must flow both forwards and backwards between the neurons. The BISTr controller is responsible for managing the tri-state buffers (TSBs) that enforce the direction of data flow.
Chapter 4

Theoretical Analysis

4.1 A Mathematical Presentation of Backpropagation

The presentation provided here is based on the one found in Chapter 11 of [27]. Most of the equations in this section are from that textbook, and are labeled as such.

4.1.1 Training Set

Backpropagation is a supervised learning algorithm. For each set of inputs used to train the network, there is a corresponding set of target (or expected) outputs. The entire set of inputs and their corresponding expected outputs is called the *training set*. The training set can be described mathematically as a collection of sets, wherein each set contains a vector of the inputs for a given training pattern and the respective expected output vector. The following equation shows a training set containing $Q$ elements.

$\{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_Q, t_Q\}$ (11.9 in [27])

The input vectors are denoted by $p_1, p_2$, etc. and the target outputs are represented by $t_1$, $t_2$, and so on. This convention will be used for the remainder of the document.

4.1.2 Performance Index

During the training process, the inputs are applied to the network, and the result produced by the network is graded against the expected output. This grade is commonly called the *performance index*. The performance index used in the backpropagation algorithm is mean square error. Mean square error can be calculated as follows:

$F(x) = E(e^2) = E((t - a)^2)$ (11.10 in [27])
Here, the vector $\mathbf{x}$ represents all of the weights and biases stored in the network. The actual output of the network is denoted by the symbol $\mathbf{a}$. The function $E$ is the expected error of the argument. The goal of the backpropagation algorithm is to minimize the mean square error between the expected and actual outputs. This is accomplished by modifying the network parameters (i.e., the weights and biases). Rather than use the expected error for an entire training process (running through every element in the training set), it is easier to approximate this by simply using the squared error at a given iteration $k$.

$$
\hat{F}(\mathbf{x}) = (\mathbf{t}(k) - \mathbf{a}(k))^T(\mathbf{t}(k) - \mathbf{a}(k)) = \mathbf{e}^T(k)\mathbf{e}(k) \quad (11.12 \text{ in [27]})
$$

This approximation of the mean square error will be referred to with the symbol $\hat{F}$ or $\hat{F}(\mathbf{x})$. $\mathbf{t}(k)$ and $\mathbf{a}(k)$ are the vectors containing the expected outputs and actual outputs for a given iteration $k$, respectively. The vector $\mathbf{e}(k)$ is shorthand for the difference between $\mathbf{t}(k)$ and $\mathbf{a}(k)$. The superscript $T$ denotes the matrix transpose operation.

### 4.1.3 Feedforward Propagation

Before delving into the backpropagation algorithm, it is useful to cover the equation describing forward propagation behavior since terminology from the forward propagation step will be used during the backpropagation process. The following equation describes the forward propagation process for layer $m+1$ of the network:

$$
\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \quad \text{for} \quad m = 0, 1, \ldots, M - 1 \quad (11.6 \text{ in [27]})
$$

The vectors $\mathbf{a}^{m+1}$ and $\mathbf{a}^m$ contain the outputs of layers $m+1$ and $m$ respectively. The output of the previous layer (in this case, $m$) is the input to the current layer ($m+1$). $\mathbf{W}^{m+1}$ is the weight matrix for layer $m+1$ of the network. It contains the weight for every edge between neurons in the $m$th layer and the $(m+1)$th layer. The vector $\mathbf{b}^{m+1}$ contains the bias for each neuron in layer $m+1$. The vector of activation functions for each neuron in layer $m+1$ is represented by $\mathbf{f}^{m+1}$. The term $\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}$ will often be referred to as the net sum, or $\mathbf{n}^{m+1}$ for simplicity. Therefore, the previous equation can be rewritten as:

$$
\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{n}^{m+1}) \quad \text{for} \quad m = 0, 1, \ldots, M - 1 \quad (4.1)
$$

A capital $M$ ($M$) will be used to refer to the total number of layers in the network. The network inputs ($\mathbf{p}$) serve as the input to the first layer of neurons.

$$
\mathbf{a}^0 = \mathbf{p} \quad (11.7 \text{ in [27]})
$$

Likewise, the output from the last layer of neurons (layer $M$) are the network outputs ($\mathbf{a}$).

$$
\mathbf{a} = \mathbf{a}^M \quad (11.8 \text{ in [27]})
$$
4.1.4 Steepest Descent

The backpropagation algorithm uses either stochastic gradient descent or steepest descent to alter the weights and biases to reduce the mean square error. This work uses steepest descent for the weight updating process. The details of steepest descent will not be provided here, just the equations that will be used to modify the weights and biases.

\[
\begin{align*}
w_{i,j}^m(k+1) &= w_{i,j}^m(k) - \alpha \frac{\partial \hat{F}}{\partial w_{i,j}^m}\quad (11.13 \text{ in [27]}) \\
b_i^m(k+1) &= b_i^m(k) - \alpha \frac{\partial \hat{F}}{\partial b_i^m}\quad (11.14 \text{ in [27]})
\end{align*}
\]

The term \(w_{i,j}^m\) refers to the weight along the edge between the \(j\)th neuron of the \((m-1)\)th layer and the \(i\)th neuron in the \(m\)th layer. The convention used in this document for referring to weights is \(w_{\text{destination}, \text{source}}\). Figure 4.1 demonstrates the weight naming convention between layers 5 and 6 of an example network.

![Weight naming convention between two layers of a neural network](image)

Figure 4.1: Weight naming convention between two layers of a neural network

Similarly, the term \(b_i^m\) represents the bias in the \(i\)th neuron in the \(m\)th layer. The value in parentheses \((k \text{ or } k+1)\) indicates the iteration of the training process. Thus, \(w_{i,j}^m(k+1)\) will be the value of \(w_{i,j}^m\) after the \((k+1)\)th training cycle. The symbol \(\alpha\) is the learning rate, used to control how quickly the weights and biases change. The last terms are the partial derivative of the estimated error with respect to the given weight or bias. Calculating these partial derivatives are the crux of the backpropagation algorithm.
4.1.5 Chain Rule for Calculating Derivatives

What makes calculating the partial derivatives tricky is that the output, and therefore the error, is an indirect function of all the weights and biases in the network. For example, the output of a three layer network could be described as thus:

\[
a = f^3(W^3a^2 + b^3)
\]

\[
= f^3(W^3 f^2(W^2a^1 + b^2) + b^3)
\]

\[
= f^3(W^3 f^2(W^2 f^1(W^1a^0 + b^1) + b^2) + b^3)
\]

Viewing the network in this way makes the need for the chain rule of derivatives easier to see, since the final output is the result of nested functions. A quick review of the chain rule will be provided here for reference. Consider a function \( f \) that is a function of a variable \( n (f(n)) \). The variable \( n \) is also a function of a third variable \( w (n(w)) \). How would the derivative of \( f \) with respect to \( w \) \( (\frac{df}{dw}) \) be calculated? The chain rule provides the answer. The chain rule is described below (using the variables provided):

\[
\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw} 
\]

(11.15 in [27])

The chain rule states that the derivative of \( f \) with respect to \( w \) can be calculated by taking the derivative of \( f \) with respect to \( n \) and multiplying it by the derivative of \( n \) with respect to \( w \). The chain rule will be useful for calculating the partial derivative terms in the steepest descent equations from the previous section. Using the chain rule, the partial derivatives can be deconstructed as follows:

\[
\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} 
\]

(11.18 in [27])

\[
\frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} 
\]

(11.19 in [27])

The first term in the right-hand side of the equations \( (\frac{\partial \hat{F}}{\partial n_i^m}) \), is more complicated and will be discussed in the next section. The second term in each equation \( (\frac{\partial n_i^m}{\partial w_{i,j}^m} \text{ and } \frac{\partial n_i^m}{\partial b_i^m}) \) is easier to calculate. Recall that \( n_i^m \) is the net sum of neuron \( i \) in layer \( m \). It is calculated as follows:

\[
n_i^m = \sum_{j=1}^{s_{m-1}} (w_{i,j}^m a_j^{m-1}) + b_i^m 
\]

(11.20 in [27])

From the equation above, the respective derivatives can be calculated easily:

\[
\frac{\partial n_i^m}{\partial w_{i,j}^m} = a_{j}^{m-1}, \quad \frac{\partial n_i^m}{\partial b_i^m} = 1 
\]

(11.21 in [27])
4.1.6 Definition of Sensitivity

Rather than refer to the following term as “the partial derivative of the estimated error with respect to the net sum of neuron \( i \) in layer \( m \)” for the remainder of the document, the term sensitivity will be used to describe it. The definition of the sensitivity is below:

\[
  s_i^m = \frac{\partial \hat{F}}{\partial n_i^m} \quad \text{(11.22 in [27])}
\]

Substituting the sensitivity in for the \( \frac{\partial \hat{F}}{\partial n_i^m} \) terms in equations 11.18 in [27] and 11.19 in [27] yields:

\[
  \frac{\partial \hat{F}}{\partial w_{i,j}^m} = s_i^m a_j^{m-1} \quad \text{(11.23 in [27])}
\]

\[
  \frac{\partial \hat{F}}{\partial b_i^m} = s_i^m \quad \text{(11.24 in [27])}
\]

4.1.7 Updated Steepest Descent Algorithm

With the equations presented in the prior sections, the steepest descent algorithm can be updated. The new equations are as follows:

\[
  w_{i,j}^m(k + 1) = w_{i,j}^m(k) - \alpha s_i^m a_j^{m-1} \quad \text{(11.25 in [27])}
\]

\[
  b_i^m(k + 1) = b_i^m(k) - \alpha s_i^m \quad \text{(11.26 in [27])}
\]

Here are the equations presented in matrix form:

\[
  \mathbf{W}^m(k + 1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m(a^{m-1})^T \quad \text{(11.27 in [27])}
\]

\[
  \mathbf{b}^m(k + 1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m \quad \text{(11.28 in [27])}
\]

4.1.8 Backpropagation of the Sensitivities

The last issue remaining before the backpropagation algorithm is complete is the matter of computing all the sensitivities. This will again require use of the chain rule, from which the term backpropagation gets its name. In order to calculate the sensitivities for layer \( m \), the sensitivities from layer \( (m + 1) \) are taken into account.

The recurrence relation for the sensitivities depends upon the following Jacobian matrix. The derivation of the matrix is not covered here but can be found in [27].

\[
  \frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1} \mathbf{F}^m(\mathbf{n}^m) \quad \text{(11.33 in [27])}
\]
where

\[ \tilde{F}^m(n^m) = \begin{bmatrix} \dot{f}_1^m(n_1^m) & 0 & \cdots & 0 \\ 0 & \dot{f}_2^m(n_2^m) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \dot{f}_3^m(n_{S^m}^m) \end{bmatrix} \]  

(11.34 in [27])

After applying the chain rule, the equation for calculating the sensitivities in layer \( m \) (based on those from layer \( (m + 1) \)) is:

\[ s^m = \tilde{F}^m(n^m)(W^{m+1})^T s^{m+1} \]  

(11.35 in [27])

This equation shows that there are three things necessary to calculate the sensitivity for a given layer: (1) the derivative of the activation functions in that layer with respect to their net sums, (2) the weights between the current and next layer, and (3) the sensitivities from the next layer. The sensitivities for the final layer are computed first. Then, the calculation ripples back layer by layer until all the sensitivities are computed (hence the name backpropagation). The recurrence relationship is shown in the equation below (where \( M \) is the final layer in the network):

\[ s^M \rightarrow s^{M-1} \rightarrow \cdots s^2 \rightarrow s^1 \]  

(11.36 in [27])

One final question must be answered before the backpropagation algorithm is complete: how are the sensitivities for the last layer \( (M) \) calculated? As before, the derivation of the following equations is not covered here, but can be found in [27]. The equation is presented for a single neuron (Equation 11.39) and in matrix form (Equation 11.40):

\[ s_i^M = -2(t_i - a_i)\dot{f}_i^M(n_i^M) \]  

(11.39 in [27])

\[ s^M = -2(t - a)f^M(n^M) \]  

(11.40 in [27])

### 4.2 Adapting the Equations to the BISTr Paradigm

The following subsections break down how the different parts of the training process—forward propagation, backpropagation, and weight updating—are handled in BISTr. The purpose of the sections is to present the equations used in the BISTr process (specifically those not discussed in the last section) and to provide a preliminary view of the hardware used to implement them. The complete overview of the hardware is presented in Chapter 5.

#### 4.2.1 Forward Propagation in BISTr

The feedforward operation of a BISTr neural network is relatively straightforward. The inputs from the previous layer \( (a_i^{m-1}) \) are multiplied with their corresponding weight \( (w_{i,j}^{m-1}) \)
and are summed together along with the bias ($b_i^m$). Then, the net sum ($n_i^m$) goes through the activation function ($f_i^m(n_i^m)$) and produces the output for the neuron ($a_i^m$). The equations for the calculation of the net sum and the neuron output for neuron $i$ in layer $m$ are provided in the respective equations below:

$$n_i^m = \sum_{j=1}^{S_{m-1}} (w_{i,j}^m a_{j}^{m-1}) + b_i^m \quad (4.2)$$

$$a_i^m = f_i^m(n_i^m) \quad (4.3)$$

Translating the equations above into hardware is a fairly easy task. All that is required are some multiplication units, a summation unit, and the activation function. The derivative of the activation function included in Figure 4.2 will be used in the backpropagation process.

![Figure 4.2: Hardware for feedforward operation of BISTr](image)

### 4.2.2 Backpropagation in BISTr

Equation 11.35 in [27] in Section 4.1.8 describes the necessary components needed to calculate the sensitivities in a given layer. However, it is presented in matrix form, which makes it difficult to see the calculation of the individual sensitivity for each neuron. BISTr calculates the sensitivities individually at each neuron, so the equations will be derived here using an example. The example will make use of Figure 4.1. Refer back to that figure to get a visual representation of the derivation process.

The goal in this example is to calculate the three sensitivities for layer 5. This will require three parts: (1) the Jacobian matrix of activation function derivatives for layer five ($\mathbf{F}^5(n^5)$),
(2) the weight matrix for layer six ($W^6$), and (3) the sensitivities from layer six ($s^6$). The sensitivity vector for layer six can be described as follows:

$$s^6 = \begin{bmatrix} s_1^6 \\ s_2^6 \end{bmatrix}$$

The weight matrix for layer six is given as:

$$W^6 = \begin{bmatrix} w_{1,1}^6 & w_{1,2}^6 & w_{1,3}^6 \\ w_{2,1}^6 & w_{2,2}^6 & w_{2,3}^6 \end{bmatrix}$$

Finally, the Jacobian matrix of activation function derivatives for layer five is:

$$\mathbf{F}^5(n^5) = \begin{bmatrix} f^5_1(n_1^5) & 0 & 0 \\ 0 & f^5_2(n_2^5) & 0 \\ 0 & 0 & f^5_3(n_3^5) \end{bmatrix}$$

The first step is to take the transpose of the weight matrix and multiply it by the sensitivity vector for layer six.

$$(W^6)^T s^6 = \begin{bmatrix} w_{1,1}^6 & w_{1,2}^6 & w_{1,3}^6 \\ w_{2,1}^6 & w_{2,2}^6 & w_{2,3}^6 \end{bmatrix} \begin{bmatrix} s_1^6 \\ s_2^6 \end{bmatrix} = \begin{bmatrix} s_1^6 w_{1,1}^6 + s_2^6 w_{2,1}^6 \\ s_1^6 w_{1,2}^6 + s_2^6 w_{2,2}^6 \\ s_1^6 w_{1,3}^6 + s_2^6 w_{2,3}^6 \end{bmatrix}$$

Then, the product is multiplied with the Jacobian of activation function derivatives.

$$s^5 = \mathbf{F}^5(n^5)(W^6)^T s^6 = \begin{bmatrix} f^5_1(n_1^5) & 0 & 0 \\ 0 & f^5_2(n_2^5) & 0 \\ 0 & 0 & f^5_3(n_3^5) \end{bmatrix} \begin{bmatrix} s_1^6 w_{1,1}^6 + s_2^6 w_{2,1}^6 \\ s_1^6 w_{1,2}^6 + s_2^6 w_{2,2}^6 \\ s_1^6 w_{1,3}^6 + s_2^6 w_{2,3}^6 \end{bmatrix}$$

From this expression, the sensitivity for a given neuron can be extracted. For example, the sensitivity for neuron 1 in layer 5 is:

$$s_1^5 = f^5_1(n_1^5)(s_1^6 w_{1,1}^6 + s_2^6 w_{2,1}^6)$$

This can be generalized to:

$$s_i^m = f^m_i(n_i^m)(s_1^{m+1} w_{1,i}^{m+1} + s_2^{m+1} w_{2,i}^{m+1} + \cdots + s_{m+1}^{m+1} w_{S_{m+1},i}^{m+1})$$

$$= f^m_i(n_i^m) \sum_{j=1}^{S_{m+1}} s_j^{m+1} w_{j,i}^{m+1}$$
For easier reference later, the summation term in Equation 4.10 will be called the weighted sum \( (w_s^{m+1}) \) as defined in the equation below:

\[
w_s^{m+1} = \sum_{j=1}^{s_m+1} s_j^{m+1} w_j^{m+1}
\]  

(4.11)

Therefore, Equation 4.10 can also be defined as:

\[
s_i^m = \hat{f}_i^m(n_i^m) w_s^{m+1}
\]  

(4.12)

As with the feedforward operation, the hardware to implement the backpropagation process is not complex. Multipliers are required to multiply the sensitivities from the next layer by the corresponding weights. A summation unit is needed to generate the weighted sum. Finally, one more multiplier is used to multiply the weighted sum by the derivative of the activation function and produce the sensitivity. The process is shown from right to left in Figure 4.3 to indicate the flow of data coming backwards from the next layer.

![Figure 4.3: Hardware for backpropagation of sensitivities in BISTr](image)

Before moving on to the weight updating, it is prudent to present an alternate layout of the hardware in the previous figure. The need for this may not be apparent now, but will be explained in Chapter 5. Instead of applying the weights for the next layer sensitivities in this layer, it is assumed that they were already weighted before arriving. The summation and multiplication by the derivative are the same. After the sensitivity for this neuron is generated, it is then multiplied by the weights for the previous layer and the weighted sensitivities are propagated backwards. Figure 4.4 shows the alternate hardware implementation.

### 4.2.3 Weight Updating in BISTr

Updating the weights and biases is a computationally simple task, requiring at most two multiplications and one subtraction. The equations derived in the previous section are
Figure 4.4: Alternate hardware for backpropagation in BISTr

The hardware implementations of Equations 4.13 and 4.14 are shown in Figures 4.5 and 4.6 respectively. Registers are used to store the subsequent values of $w_{i,j}^m$ and $b_i^m$.

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha s_i^m a_j^{m-1}$</td>
<td>(4.13)</td>
</tr>
<tr>
<td>$b_i^m(k+1) = b_i^m(k) - \alpha s_i^m$</td>
<td>(4.14)</td>
</tr>
</tbody>
</table>

The hardware implementations of Equations 4.13 and 4.14 are shown in Figures 4.5 and 4.6 respectively. Registers are used to store the subsequent values of $w_{i,j}^m$ and $b_i^m$. 
4.3 Training Iteration Example

To help demonstrate the training process in action, a complete iteration of training on a simple network will be provided here. The network that will be used is a 1-3-1 network. It has one input, three neurons in the first layer, and one neuron in the final layer (and therefore one output). The three neurons in the first layer all have logistic activation functions. The last neuron has a linear activation function. A diagram showing the network structure is provided in Figure 4.7. The network will be used to approximate the function:

\[ y(p) = 1 - \sin \left( \frac{\pi}{2} p \right) \quad \text{for} \quad -2 \leq p \leq 2 \]  \hspace{1cm} (4.15)

To generate a training set, the function \( y(p) \) will be evaluated at a set of values over the interval \(-2 \leq p \leq 2\). For uniformity, points 0.1 units apart will be used (-2, -1.9, -1.8, \ldots). Utilizing this interval spacing for the points, the training set has a total of 41 elements. The function \( y(p) \) is evaluated at these 41 points to complete the training set.

\[ \{-2,1\}, \{-1.9,1.1564\}, \ldots, \{2,1\} \]  \hspace{1cm} (4.16)

The last thing required before starting the training process are initial values for the weights and biases. These are typically selected to be small, random values. The initial values that will be used in this example are as follows:

\[
W^i(0) = \begin{bmatrix} 0.0938 \\ 0.9150 \\ 0.9298 \end{bmatrix}, \quad b^i(0) = \begin{bmatrix} -0.6848 \\ 0.9412 \\ 0.9143 \end{bmatrix}
\]
\( \mathbf{W}^2(0) = [-0.0292 \ 0.6006 \ -0.7162], \mathbf{b}^2(0) = [-0.1565] \)

The network output for the initial parameters is shown in Figure 4.8, along with the actual output of the sine function for comparison.

The training process can now begin. While the training set can be presented in any order, it is common to choose the points randomly. For this example, the value of \( p = 1 \). Therefore, the output of the first layer is as follows:

\[
\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1 p + \mathbf{b}^1) = \text{logsig}\left(\begin{bmatrix} 0.0938 \\ 0.9150 \\ 0.9298 \end{bmatrix} + \begin{bmatrix} -0.6848 \\ 0.9412 \\ 0.9143 \end{bmatrix}\right) = \text{logsig}\left(\begin{bmatrix} -0.5910 \\ 1.8562 \\ 1.8441 \end{bmatrix}\right) = \begin{bmatrix} 0.3564 \\ 0.8649 \\ 0.8634 \end{bmatrix}
\]

The output of the second layer is consequently:

\[
\mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2 \mathbf{a}^1 + \mathbf{b}^2) = \text{linear}\left(\begin{bmatrix} -0.0292 \\ 0.6006 \\ -0.7162 \end{bmatrix} \begin{bmatrix} 0.3564 \\ 0.8649 \\ 0.8634 \end{bmatrix} + [0.1565] \right) = -0.2659
\]

The expected output (\( t \)) is the value of \( y(1) = 1 - \sin(\frac{\pi}{2}) = 0 \). As such, the error is:

\[
e = t - a = (0 - -0.2659) = 0.2659
\]

Now that forward propagation is complete, the next step in the training iteration is to backpropagate the sensitivities. To do that, the derivatives of the activation functions are required. In the first layer, the activation function is the logistic function. Its derivative is as follows:

\[
\dot{f}^1(n) = \frac{d}{dn}\left(\frac{1}{1 + e^{-n}}\right) = \frac{e^{-n}}{(1 + e^{-n})^2} = \left(1 - \frac{1}{1 + e^{-n}}\right)\left(\frac{1}{1 + e^{-n}}\right) = (1 - \mathbf{a}^1)(\mathbf{a}^1)
\]

The activation function for the second layer is a linear function. Its derivative is:

\[
\dot{f}^2(n) = \frac{d}{dn}(n) = 1
\]

With the preceding equations in hand, the sensitivities can now be computed. The sensitivity for the second layer is:

\[
s^2 = -2\mathbf{F}^2(t - a) = -2(\dot{f}^2(n^2))(0.2659) = -2(1)(0.2659) = -0.5317
\]

The sensitivities for the first layer can be calculated using the backpropagated sensitivity
Figure 4.7: Sine function approximation network

Figure 4.8: Network response before training process
from the second layer:

\[ s^1 = F^1(n^1)(W^2)^T s^2 \]

\[
= \begin{bmatrix}
(1 - a^1_1)(a^1_1) & 0 & 0 \\
0 & (1 - a^1_2)(a^1_2) & 0 \\
0 & 0 & (1 - a^1_3)(a^1_3)
\end{bmatrix}
\begin{bmatrix}
-0.0292 \\
0.6006 \\
-0.7162
\end{bmatrix}
\begin{bmatrix}
-0.5317
\end{bmatrix}
\]

\[
= \begin{bmatrix}
(1 - 0.3564)(0.3564) & 0 & 0 \\
0 & (1 - 0.8649)(0.8649) & 0 \\
0 & 0 & (1 - 0.8634)(0.8634)
\end{bmatrix}
\begin{bmatrix}
-0.0292 \\
0.6006 \\
-0.7162
\end{bmatrix}
\begin{bmatrix}
-0.5317
\end{bmatrix}
\]

\[
= \begin{bmatrix}
0.2294 & 0 & 0 \\
0 & 0.1169 & 0 \\
0 & 0 & 0.1179
\end{bmatrix}
\begin{bmatrix}
0.0155 \\
-0.3193 \\
0.3808
\end{bmatrix}
\]

\[
= \begin{bmatrix}
0.0036 \\
-0.0373 \\
0.0449
\end{bmatrix}
\]

The last step in the training process is to update the weights and biases. In this example, the learning rate \( \alpha = 0.1 \). The weight updating equations are as follows:

\[
W^2(1) = W^2(0) - \alpha (a^1)^T s^2
\]

\[
= \begin{bmatrix}
-0.0292 & 0.6006 & -0.7162
\end{bmatrix}
- 0.1 \begin{bmatrix}
0.3564 & 0.8649 & 0.8634
\end{bmatrix}
\begin{bmatrix}
-0.5317
\end{bmatrix}
\]

\[
= \begin{bmatrix}
-0.0102 & 0.6466 & -0.6703
\end{bmatrix}
\]

\[
b^2(1) = b^2(0) - \alpha s^2 = \begin{bmatrix}
-0.1565
\end{bmatrix}
- 0.1 \begin{bmatrix}
-0.5317
\end{bmatrix}
= \begin{bmatrix}
-0.1033
\end{bmatrix}
\]

\[
W^1(1) = W^1(0) - \alpha (a^0)^T s^1
\]

\[
= \begin{bmatrix}
0.0938 \\
0.9150 \\
0.9298
\end{bmatrix}
- 0.1 \begin{bmatrix}
0.0036 \\
-0.0373 \\
0.0449
\end{bmatrix}
\]

\[
= \begin{bmatrix}
0.0934 \\
0.9187 \\
0.9253
\end{bmatrix}
\]

\[
b^1(1) = b^1(0) - \alpha s^1 = \begin{bmatrix}
-0.6848 \\
0.9412 \\
0.9143
\end{bmatrix}
- 0.1 \begin{bmatrix}
0.0036 \\
-0.0373 \\
0.0449
\end{bmatrix}
= \begin{bmatrix}
-0.6852 \\
0.9449 \\
0.9098
\end{bmatrix}
\]

With that, the training iteration is complete. To continue the training process, another input would be selected and the entire process repeated. The training process would continue in this manner until a condition is met. This condition is usually when the difference between the network output and the expected output is acceptably small. The complete training process typically requires numerous passes through the training set. Figure 4.9 shows the approximation network after going through 1000 passes of the training set.
Figure 4.9: Network response after training process
Chapter 5

Detailed Design Overview

This chapter covers the details of the hardware design of BISTr. It starts by building up the design of a single neuron in the BISTr paradigm, then widens the focus to the overall data path and flow of data in a BISTr neural network. Finally, the chapter concludes by covering a few more specifics of the hardware design: the BISTr controller, scan chains, and output sensitivity generators. Additionally, an alternate design utilizing a buffer and off-chip DRAM module is included at the end due to limitations with the original design.

The design can work with a wide variety of data representations (signed or unsigned, varying bit width, fixed-point or floating-point numbers, etc.) provided the proper adders, multipliers, registers, and so on are used. The trade-offs of choosing different data represen-
tations will not be discussed here. The design in this work uses a signed, 16-bit, fixed-point representation. It was chosen for simplicity, ease of use, range, and decent precision. The 16 bits are divided up as follows [12 bits . 4 bits], where the 12 most-significant bits are the integer part and the lowest 4 bits are the decimal part. For example, the binary value (00011100000101)₂ or (3845)₁₀ would be (000111100000101₁₀)₂ = 240.3125 in the representation used. To convert a number from an integer form to this fixed-point representation, divide it by 16.

5.1 Subnode Architectures

The neurons in BISTr rely heavily on smaller modules called subnodes. These subnodes make up the core of the neuron architecture, and are key to the operation of BISTr in general. There are two types of subnodes, weight subnodes and bias subnodes. The two are largely similar in structure, aside from a few differences.

5.1.1 Weight Subnode Architecture

As its name implies, the weight subnodes are responsible for storing, operating on, and updating the weights for a neuron. Figure 5.2 shows the circuit diagram for the weight subnode. This section will examine subnode \( j \) in neuron \( i \) of layer \( m \).

The I/O signals that the subnode processes are defined here:

- **feedforward** A control signal indicating if the network is in feedforward or backpropagation mode. A low (0) indicates backpropagation mode and a high (1) indicates feedforward mode.
- **weight_update** A control signal that dictates if the network is in weight update mode or not. A low (0) means the network is not in weight updating mode while a high (1) means that the network is updating weights.
- **mult_2** A control signal used to indicate whether the network is in the first or second phase of weight updating (since two multiplications are required for weights). A low (0) indicates the first phase and a high (1) indicates the second phase.
- **scan_control** A control signal that dictates where the scan register gets its input. A low (0) means the register takes input from the subtractor in the node. A high (1) means the internal scan chain is active and the scan register takes input from the scan_in port. Consequently, the next weight register in the scan chain gets its input from scan_out.
- **scan_in** Input used to scan weights into/through this register (16 bits).
- **scan_out** Output that connects to the next scan register in the chain (16 bits).
- \( \alpha \) Input for the learning rate used during training (16 bits).
Figure 5.2: Weight subnode architecture circuit diagram
The subnode has four primary modes of operation, as dictated by the three control signals: feedforward, weight_update, and mult_2. These control signals are used as the select bits on the multiplexers (MUXes) and demultiplexers (DEMUXes) in the module. $\text{S}_1$ is controlled directly by weight_update. $\text{S}_0$ is derived from the logical inclusive or of feedforward and mult_2 ($\text{feedforward} + \text{mult}_2 = \text{S}_0$). The motivation for the multiplexers is to reuse the multiplication unit in the subnode as much as possible. Perhaps the best way to summarize the subnode operation is to list the multiplexer outputs in a table. The output of the top 4x1 MUX will be called $O_1$ since it is the first operand of the multiplier. The output of the bottom 4x1 MUX is similarly called $O_2$. Table 5.1 shows the four modes of operation and the outputs of all the MUXes for each mode.

### Mode 1: Feedforward Mode

While in feedforward mode, the subnode acts as simple multiplier, the operands being the input from the previous layer ($a_j^{m-1}$) and the corresponding weight along the edge ($w_{i,j}^m$). As the name implies, feedforward mode is active when the feedforward signal is high (and weight_update and mult_2 are low). The result of the multiplication is passed directly from the demultiplexer to another MUX and finally the output of the subnode ($\text{out}_j$) to go to the summation unit in the neuron. Feedforward mode is used during normal inference (when the network is running normally outside of training) as well as the first step during the training process.
Mode 2: Backpropagation Mode

In backpropagation mode, the subnode is performing two tasks. The first of these tasks is sending the weighted sensitivity from the next layer \((w_{j,i}^{m+1}s_{j}^{m+1})\) to the summation unit in the neuron via \(out_{j}\). After the sensitivity \((s_{i}^{m})\) for the neuron is ready (it is generated at the neuron level), the subnode multiplies the sensitivity by the weight stored in the node. The result is passed through a tri-state buffer (only active during backpropagation mode) that allows it to be sent back to the neuron in layer \(m-1\). The tri-state buffer is required to prevent contention of the wire since the \(a_{j}^{m-1}\) port is bidirectional. Backpropagation mode is active when the feedforward, weight_update, and mult_2 signals are all low.

Mode 3: Weight Updating Step 1

The weight updating process takes two steps. The reason for this is that two multiplications are required to update the weight. If the same multiplication unit is used for both, they must be performed in sequence. As such, the first multiplication takes the input from the forward propagation mode \((a_{j}^{m-1})\) and multiplies it with the sensitivity for the neuron \((s_{i}^{m})\). This result \((a_{j}^{m-1}s_{i}^{m})\) is stored in a register to be used in the next step. Weight Updating Step 1 is active when feedforward and mult_2 are low and weight_update is high.

Mode 4: Weight Updating Step 2

This step performs the second multiplication of the weight updating process. The two operands in this step are \(a_{j}^{m-1}s_{i}^{m}\) and the learning rate \(\alpha\). The final product from this step is \(\alpha a_{j}^{m-1}s_{i}^{m}\). This is subtracted from the current weight at the node \((w_{i,j}^{m}(k)\) for iteration \(k)\) to complete the weight updating process. The complete equation this process realizes is shown here:

\[
w_{i,j}^{m}(k + 1) = w_{i,j}^{m}(k) - \alpha s_{i}^{m}a_{j}^{m-1}
\]

This mode is active when feedforward is low and both weight_update and mult_2 are high.

5.1.2 Bias Subnode Architecture

By comparison, the bias subnodes are much simpler than the weight subnodes. This is due to a lesser need to reuse the multiplication unit. As such, the multiplexers have been removed, along with two of the registers. The scan chain control, input, and output are the same as in the weight subnode. Figure 5.3 shows the architecture of the bias subnode.

During feedforward operation, the bias subnode must present the bias on its output line. The bias can be presented as is without being multiplied by another number. Therefore, on the diagram, the output of the register storing the bias is routed directly to a bit-wise AND
array that leads to the output for the node. The control signal feedforward is used as the other input to the AND array so the bias does not show up on the output line when the network is not in feedforward mode. The bias subnode is not active during backpropagation mode since the bias is not used in the calculation of the sensitivities.

The updating of the bias is also easier than with the weights. Only one multiplication is required, so an intermediate product does not need to be stored in a register. The two terms required to update the bias are the learning rate alpha ($\alpha$) and the sensitivity ($s^m_i$). These two terms are connected to the multiplier in the bias subnode, whose output connects to another bit-wise AND array. As before, the mult2 control signal is used to allow the data to pass to the subtractor. This ensures that the bias is only updated during the second phase of the weight updating process. The other input for the subtractor is the bias ($b^m_i$). This satisfies the equation for bias updating:

$$b^m_i(k + 1) = b^m_i(k) - \alpha s^m_i$$

5.2 Neuron Architecture

Much of the hardware that comprises a BISTr neuron is contained within the subnodes. A good deal of the computation (but not all) takes place at the subnode level as well. In addition to elaborating on the design and operation of the neuron level, this section discusses how the subnodes operate on a macro-level within the neuron. As with the subnnodes, presenting
the operation of the neuron in terms of the various modes (feedforward, backpropagation, and weight updating) is helpful in breaking down the complete operation of the neuron. The complete diagram of the neuron architecture is presented in Figure 5.4.

Many of the I/O signals at the neuron level are also used at the subnode level. Since the aforementioned signals have already been defined, they will not be repeated here. Refer back to Section 5.1.1 for the definitions not present here.

**register_reset**  Global reset signal that resets all registers in the subnodes and neuron.

**n_scan_control**  Controls the internal scan chain for all subnodes in the neuron (16 bits).

**n_scan_in**  Internal scan chain input for the neuron. Connects to n_scan_out from the previous neuron in the scan chain (16 bits).

**n_scan_out**  Internal scan chain output for the neuron. Connects to n_scan_in for the next neuron in the scan chain (16 bits).

**a_{m-1}^1, a_{m-1}^2, ...**  Bidirectional ports connecting this neuron to the neurons in layer $m-1$. They serve as inputs to the neuron while in feedforward mode. During backpropagation, the weighted sensitivity is transmitted back to the neurons in layer $m-1$ using these lines (16 bits).

**a_i^m**  Neuron output after the activation function has been applied. Goes to all neurons in layer $m+1$ (16 bits).

### 5.2.1 Feedforward Behavior

The neuron is responsible for calculating three values during feedforward mode. The first of these values is the net sum. As derived in the previous chapter, the net sum is given as:

$$n_i^m = \sum_{j=1}^{S_{m-1}} (w_{i,j}^m a_j^{m-1}) + b_i^m$$

The subnodes handle the multiplication of the corresponding weights ($w_{i,j}^m$) and inputs ($a_j^{m-1}$). Those products come out of the respective $out_j$ ports on the subnodes and are summed up by the summation unit in the neuron. The second value that needs to be calculated is the neuron output $a_i^m$. It is obtained by passing the net sum through the activation function $f_i^m$ as shown in the following equation:

$$a_i^m = f_i^m(n_i^m)$$

The last value that the neuron needs to calculate in feedforward mode is the derivative of the activation function for the given net sum. It is calculated as follows:

$$\frac{df_i^m}{dn} = \dot{f}_i^m(n_i^m)$$

This derivative is stored in a register to be used during the backpropagation step.
Figure 5.4: Neuron architecture circuit diagram
The last thing to note about the feedforward mode operation is the fan out of the neuron output $a_i^m$. The reason the fan out occurs within the neuron module (instead of between the neurons) is necessitated by the backpropagation step and will be explained in more detail there. Each of the lines going to the neurons in layer $m + 1$ has a tri-state buffer that is controlled by feedforward such that the output of the activation function is only present on the line while in feedforward mode.

### 5.2.2 Backpropagation Behavior

The neuron executes a few steps during the backpropagation phase. The most logical order to present the steps is in the order in which they are calculated. On the circuit diagram, this order goes from right to left.

The first step during backpropagation is to sum up all the weighted sensitivities from the next layer ($m + 1$). The weighted sensitivities come in on the lines that carry the neuron output ($a_i^m$). They are fed to the subnodes which feed them into the summation unit via their $out_j$ ports. The equation for the weighted sum was derived in the last chapter but is provided again here for easy reference.

$$w_i^{s_{m+1}} = \sum_{j=1}^{s_{m+1}} s_j^{m+1} w_{j,i}^{m+1}$$

The second step in the backpropagation phase is to calculate the sensitivity ($s_i^m$). The two values needed to calculate the sensitivity are the weighted sum that was just calculated and the derivative of the activation function which was calculated and stored during the forward propagation phase. The equation for calculating the sensitivity is:

$$s_i^m = f_i^m(n_i^m) w_i^{s_{m+1}}$$

The sensitivity is then given to each of the weight subnodes, which multiply the sensitivity by the weight they store. These weighted sensitivities are sent to layer $m - 1$ via the bidirectional lines (labeled $a_i^{m-1}, a_i^{m-1}, a_i^{m-1}, \ldots$ on the diagram).

### 5.2.3 Weight Updating Behavior

All the weight updating operations happen internally within the individual subnodes. The neuron supplies the learning rate and the sensitivity to the subnodes, and the weights and bias are stored locally in each subnode. Refer back to Section 5.1.1 for a more detailed description of the weight updating hardware.
5.2.4 Additional Considerations

The design of the neuron is fairly easy to extend as the number of inputs or outputs varies. If more inputs are needed, add more subnodes. If more outputs are required, increase the fanout of the neuron output. But what if the number of inputs and outputs don’t match? Luckily, the answer to this question is not too complicated. One of three situations will be present depending on the number of inputs and outputs.

1. (inputs > outputs) — If the number of inputs exceeds the number of outputs, the number of weight subnodes will outnumber the fanout of the neuron output. In this case, the extra weight subnodes do not receive a weighted sensitivity and have that port grounded instead.

2. (input = outputs) — The ideal case, but unlikely to actually occur. Each weight subnode handles one fanout of the neuron output.

3. (inputs < outputs) — The number of outputs exceeds the number of inputs. There are more fanouts of the neuron output than there are weight subnodes. The extra weighted sensitivities are routed to additional 2x1 multiplexers whose output connects to the summation unit as shown in Figure 5.5.

5.3 Data Path

A brief description of the data flow was provided in Section 3.2. This section will elaborate more on the data path, including which elements of BISTr hardware are active in each mode. As before, the explanation will follow the order of a training cycle in BISTr. For each mode, the appropriate lines and modules on the top down diagram will be highlighted to indicate that they are active. Any lines or modules that are not highlighted are not used for that mode. The user control of the BISTr controller is always active and is therefore not highlighted.

5.3.1 Expected Output Scan

The first step in the training process in BISTr is to scan in the expected outputs into the boundary scan chain. The hardware involved is described in Section 5.5.2. The expected outputs will be used later in the training process. The user passes the target outputs to the boundary scan registers via the port EOTDI on the BISTr controller (details in section 5.4). In Figure 5.6, this is represented with the Boundary Scan Chain I/O line. The control signals issued by the BISTr controller during this mode enable the boundary scan registers to accept new inputs.
Figure 5.5: Summation unit in a neuron with supplementary MUX units
In the current design, the expected output scan is its own, isolated step. Given that scanning is a relatively slow process, this operation could be overlapped with feedforward propagation mode in a future revision. This would help to mitigate the time taken by the expected output scan. All of the expected outputs would need to be scanned before the backpropagation begins. Alternatively, the expected outputs could be loaded into the registers via a memory access model if a memory buffer (detailed in section 5.7) is used.

5.3.2 Feedforward Propagation Mode

After the expected outputs are scanned in, the next step of the BISTr training process is feedforward propagation mode. Figure 5.7 shows the lines and modules active in this mode. This also shows the behavior during inference (normal operation not during training). The inputs to the network (also called the primary inputs) are fed to the network. The data then propagates forward through the network layer by layer, being processed the neurons. After the data goes through the last layer of neurons, it appears on the primary outputs. The BISTr controller configures the neurons to operate in feedforward mode using the control signal feedfoward.
Figure 5.7: Modules and lines active during the feedforward mode of training

Figure 5.8: Modules and lines active during the backpropagation mode of training
5.3.3 Backpropagation Mode

The next phase of training is backpropagation mode. As seen in Figure 5.8, a lot is active during this mode. The data in this mode flows from right to left in the diagram. The flow starts with the output sensitivity generators (detailed in Section 5.6). The output sensitivity generators (OSGs) calculate the error between the network outputs and the expected outputs stored in the boundary scan chain. The OSGs produce the first set of sensitivities handed to the neurons in the output layer. From there, the sensitivities propagate backwards through the entire network until they reach the first layer. The BISTr controller is issuing control signals that put the network hardware in backpropagation mode.

5.3.4 Weight Updating Mode

![Diagram of Neural Network and BISTr Controller](image)

Figure 5.9: Modules and lines active during the weight updating mode of training

The final phase of training in BISTr is the weight updating mode. Figure 5.9 shows the active modules during this step. The weight updating happens internally in each of the neurons, and no data flows between them to make this happen. The sensitivities propagated backwards and the neuron inputs (propagated forward in that mode) are used as part of the weight updating process. The control signals given by the BISTr controller are responsible for the execution of the weight updating process. After the weight updating phase is complete, the training cycle is over. Another training cycle can be started, or the network can be used normally (in feedforward propagation mode).
5.4 BISTr Controller

The BISTr controller is a finite-state machine (FSM) that dictates the operation of the BISTr circuitry. Its primary responsibility is to oversee the training process by issuing control signals that configure the circuitry in the neurons. It also has direct control over the internal and boundary scan chains. The BISTr controller is designed to be easy for the user to control, with the majority of the operation depending on one control signal. Since the BISTr controller is a simple FSM, the best way to elaborate on its behavior is to provide a description of its inputs and outputs, then present and explain its state diagram.

The inputs and outputs of the BISTr controller are as follows:

- **TCLK** [input] (Test Clock) Global clock signal used by all BISTr circuitry.
- **TRST** [input] (Test Reset) Signal that resets the state to NORMAL_MODE when high (1). Takes effect on next rising edge of the clock (TCLK).
- **TMS** [input] (Test Mode Select) Signal that controls the state transitions. State transitions happen upon the rising edge of the clock (TCLK).
- **WBTDI** [input, 16 bits] (Weight/Bias Test Data In) Port where the user inputs weights and biases when using the internal scan chain. Connects directly to the nscan_start signal output from the controller.
- **WBTDO** [output, 16 bits] (Weight/Bias Test Data Out) Port where the weights and biases come out for the user to read when using the internal scan chain. Connects directly to the nscan_end signal input to the controller.
- **EOTDI** [input, 16 bits] (Expected Output Test Data In) Port where the user inputs the expected outputs during the training process. Connects directly to the bscan_start signal output from the controller.
- **EOTDO** [output, 16 bits] (Expected Output Test Data Out) Port where the expected outputs come out for the user to read from the boundary scan. Connects directly to the bscan_end signal input to the controller.
- **LEARN_RATE** [input, 16 bits] (Learning Rate) User input that controls the learning rate during the training process. Can be changed whenever, but is only recommended between training iterations. Connects directly to the controller output alpha which is sent to all the neurons.
- **nscan_start** [output, 16 bits] Connects to the start of the internal neuron scan chain that contains the weights and biases.
- **nscan_end** [input, 16 bits] Connects to the end of the internal neuron scan chain that contains the weights and biases.
- **bscan_start** [output, 16 bits] Connects to the start of the boundary scan chain that contains the expected outputs for a given training iteration.
- **bscan_end** [input, 16 bits] Connects to the end of the boundary scan chain that contains the expected outputs for a given training iteration.
- **alpha** [output, 16 bits] Signal that distributes the learning rate to all the neurons in the BISTr circuit.
- **mult_2** [output] Control signal that indicates whether the neurons should be
executing step 1 or step 2 of the weight updating process during training. A low (0) indicates step 1 and a high (1) step 2.

**feedforward**

[output] Control signal that indicates whether the network is in feedforward or backpropagation mode. A low (0) indicates backpropagation mode and a high (1) means feedforward mode.

**weight_update**

[output] Control signal that dictates whether the weights are being updated or not during the training process. A low (0) means the weights are not being updated and a high (1) means they are.

**nscan_enable**

[output] Control signal that dictates whether the weight and bias registers in the neurons accept input from the scan chain (1) or from the weight updating hardware (0).

**bscan_enable**

[output] Control signal that acts as a register enable for the registers in the scan chain. They will accept inputs when it is high (1), but not when the signal is low (0).

**register_reset**

[output] Control signal that serves as a global reset to all registers in the design, including the neurons and scan chains.

One important detail to note are the 16 bit ports on the BISTr controller (WBTDI, WBTDO, EOTDI, and EOTDO to name a few). To simplify and speed up the simulation, each port is treated as 16 parallel lines. In reality, this is unrealistic to do for so many ports, particularly for the signals that come or go from off the chip (the signals in all caps). To alleviate this issue, the ports can be serialized such that the data comes in/goes out on a single pin. The data can then be parallelized as needed once on the chip.

This can also apply to the weight/bias scan chain itself. Scan chains are usually only one bit in width. In the design presented here, the data moves in parallel as it does with the ports on the BISTr controller. This means that there are 16 scan lines operating in parallel. As before, this was done to simplify the code and make the simulation easier. A similar approach could be used to modify the scan chain such that the data flows serially through the chain instead of in parallel. Therefore, only one bit would be required for the scan chain.

The state diagram for the BISTr controller is shown in Figure 5.10. The explanation of the behavior will begin at the NORMAL_MODE state. This is the default state for the FSM, as initiating the reset will bring it back to this state. NORMAL_MODE is also the state for when the neural network is making inferences (operating normally and not training). As such, the feedforward control signal is being issued while in this state. Before traversing the diagram, the labeling must be explained. Only one signal, TMS (Test Mode Select), influences how the FSM transitions between states. Therefore, whenever a 0 or 1 appears on the diagram next to an arrow, it means that the FSM will transition to the state pointed to by the arrow if TMS has that value at the next rising edge of the clock.

Starting at NORMAL_MODE, the FSM will stay in that state as long as TMS is held low (0). If TMS is held high (1) for a clock cycle, the FSM will transition to START_TRAIN. In this state, the diagram branches. In the next clock cycle, if TMS is held high (1), the FSM
Figure 5.10: BISTr controller state diagram

will transition to SCAN.OR.RESET. Conversely, if TMS is low (0), the training process will begin, starting with EO.SCAN. The former branch will be explained first.

The SCAN.OR.RESET state contains another branch. If TMS is high at the next rising edge (1) of the clock, the state will become TL.RESET. The TL.RESET state issues the global reset signal (register_reset) to all the registers. The FSM will stay in this state as long as TMS is held high (1). Once TMS is low (0) for one clock cycle, the FSM will transition back to normal mode. If the other branch is taken at SCAN.OR.RESET (TMS = 0), the state will become WB.SCAN. In this state, the user can scan weights and biases into (or out of) the registers in the network using the internal scan chain. Both the nscan_enable and weight_update signals will be issued so the registers will be in scan mode and accept new inputs. The FSM will remain in this state as long as TMS = 0. Once TMS is held high (1) for one clock cycle, the FSM will return to NORMAL.MODE.
Having exhausted the SCAN_OR_RESET branch, the explanation will resume from taking the 0 branch at START_TRAIN. The first state in the main training process is EO_SCAN. In this state, the user scans in the expected outputs using the boundary scan port (EOTDI). The signal bscan_enable is issued in this state so the registers will be enabled. This state will be held as long as TMS is low (0). Once all the expected outputs are scanned in, TMS should be changed to high (1) so the FSM can transition to the next state.

The next state is FPROP_TD. The test inputs must be presented at the primary inputs at this state. The operation is identical to that of NORMAL_MODE, as the network is in feedforward mode so the test data can be propagated forward through the network. This state must be held for as many clock cycles as it takes for the test data to propagate completely through the network (equal to the number of layers in the network). However, there is no issue with holding TMS high (1) for longer than this amount. Once the forward propagation of the test inputs is complete, the training can move onto the next step. This is accomplished by changing TMS to low (0).

In the BACKPROPAGATE state, the control signal feedforward is switched to low (0) to switch the network mode to backpropagation. As with the previous state, this state must be held for a sufficient number of clock cycles (again equal to the number of layers) to allow the sensitivities to propagate backwards through the entire network. This state will be maintained as long as TMS is held low (0). Once TMS is high (1) for one clock cycle, the state will transition to WEIGHT_UPDATE_1.

As mentioned in previous chapters, the weight updating process takes two clock cycles. Consequently, there are two states to achieve this. In the first of these two, WEIGHT_UPDATE_1, the weight_update control signal is issued, but mult_2 is not. After one clock cycle, the FSM transitions unconditionally to WEIGHT_UPDATE_2. In this state, both weight_update and mult_2 are issued. After one clock cycle, the state again unconditionally transitions back to NORMAL_MODE, completing the training iteration. From here, another training cycle can be initiated the same way.

5.5 Boundary and Internal Scan Chains

The BISTr design incorporates two scan chains. The internal scan chain, also referred to as the neuron scan chain or weight/bias scan chain, provides the user the ability to directly access the weights and biases that are stored in the nodes. Every single weight and bias in the network is connected to this scan chain. The other scan chain is the boundary scan chain, also known as the expected output scan chain. The function of this scan chain is to provide the expected or target outputs during the training process. The internal scan chain will be explained first, followed by the boundary scan chain.
5.5.1 Internal Scan Chain

The internal scan chain is controlled by the BISTr controller. To use the neuron scan chain the user has to navigate to the WB_SCAN state. A full explanation of the BISTr controller design and how to control it can be found in Section 5.4. As a reminder from the previous section, this scan chain operates in parallel in the design presented. The user can input weights into this scan chain via the port WBTDI on the BISTr controller. The output of the scan chain comes out of the port WBTDO. The registers in this chain are scan registers, meaning they can accept input from two places. While the internal scan chain is active, they receive their input from the previous register in the chain. During normal operation, they receive input from elsewhere within the subnode. The control signal nscan_enable dictates this behavior. Perhaps the best way to explain the organization of the internal scan chain is to provide an example and then the general case. Figure 5.11 shows an example of the internal scan chain for a BISTr neural network.

![Diagram showing the internal scan chain organization](image)

Figure 5.11: Diagram showing the internal scan chain organization
The network in this example has one input and one output. It has three layers, each layer containing three, two, and one neuron(s) respectively. The signal nscan_in is called nscan_start in the final design. Likewise, nscan_out is called nscan_end. The first register in the scan chain is the weight for neuron 1 in layer 1 \((w_{1,1}^1)\). The following register is the bias for neuron 1 in layer 1 \((b_1^1)\). After exiting the first neuron, the scan chain goes to the second neuron in layer 1. After going through the third neuron in layer one, the scan chain connects to the first neuron in layer 2, starting with the weight corresponding to the first input to that layer \((w_{1,1}^2)\). It continues going through the weights in order, then goes through the bias for that neuron. The same is done for the other neuron in layer 2. Finally, the scan chain goes through the neuron in layer 3 before finishing.

When using the scan chain, remember that data follows the first-in, first-out (FIFO) ordering. Therefore, when scanning in a set of weights and biases, the proper order would be backwards like so, where \(b_3^3\) is input first and \(w_{1,1}^1\) last:

\[
\{b_3^3, w_{1,2}^3, b_1^2, w_{2,3}^2, w_{2,1}^2, b_1^3, w_{1,1}^2, w_{1,1}^3, b_1^1, b_2^1, w_{2,1}^1, b_1^1, w_{1,1}^1\}
\]

The data will also come out of the scan chain in this order.

Moving on to the general case, the organization of the internal scan chain is as follows. The chain starts with the first neuron in layer one. The chain progresses through the weights in the first neuron in ascending order, ending with the bias. All neurons are traversed this way. The scan chain continues from one neuron to the next in the first layer, again in ascending order. Once all the neurons in layer one have been included in the scan chain, it moves on to layer two and repeats the same ordering. This continues until all layers have been covered by the scan chain.

### 5.5.2 Boundary Scan Chain

The boundary scan is accessed as the first step of the training process. It is used during the EO_SCAN state in the BISTr controller state diagram. The user inputs the expected outputs via the port EOTDI and the outputs of the scan chain exit via EOTDO. As the registers in this scan chain only serve to hold the data to be used by the output sensitivity generators, they are simply regular registers instead of scan registers. The bscan_enable control signal acts as their enable signal. The organization of the boundary scan chain is much simpler than the internal scan chain. The expected outputs are in ascending order, going from \(t_1\) to \(t_{SM}\), where \(SM\) is the number of neurons in the final layer.

Figure 5.12 shows the boundary scan chain, along with how it connects to the output sensitivity generators. In this diagram, the last expected output is labeled with a subscript lowercase m \((t_m)\). As with the internal scan chain, the correct order to provide the expected outputs is in reverse order, as shown here:

\[
\{t_{SM}, t_{SM-1}, \ldots, t_2, t_1\}
\]
5.6 Output Sensitivity Generators

The output sensitivity generators (OSGs) are simple modules by themselves, but there’s a bit of hardware around them that also warrants attention. The boundary scan registers (on the left in Figure 5.12, labeled \( t_1 \), etc.) were covered in the previous section. The boundary scan registers provide one of the two inputs that the OSG operates on, the expected output. The other input is the output for that given neuron (aka \( a_1^M \) or \( a_m^M \) in the diagram).

The signal coming from the network is bidirectional. While the network is in feedforward mode, the output from the neuron will be present on this line. Said output is stored in a register so the value will be maintained when the network is switched to backpropagation mode. This is done for two reasons. First, it prevents spurious data from being broadcast to the primary output. Second, it ensures that the OSG receives only the output of the neuron and not feedback from its own output. Once backpropagation mode is activated, the OSG will produce its output. The output is run through a tri-state buffer that is only active during backpropagation mode. This prevents contention on the line while the network is in feedforward mode.

Now that the hardware around the OSGs has been elaborated upon, their behavior will be described. As derived in the previous chapter, the equation to generate the sensitivities at the output is as follows:

\[
\delta_i^M = -2(t_i - a_i) \hat{j}_i^M(n_i^M)
\]
The multiplication by the derivative \( f_i^M(n_i^M) \) occurs within the output neuron itself. Therefore, the value that needs to be provided to the neuron is referred to as the partial sensitivity. It can be calculated via the following equation, where \( t_i \) is the expected output of neuron \( i \) and \( a_i \) is the actual output:

\[
ps_i^M = -2(t_i - a_i)
\]

Rather than multiplying by -2, which would involve including a multiplier, two small changes will be made to the equation to work around this. The first is to swap the order of subtraction, removing the negative sign in the front:

\[
ps_i^M = 2(a_i - t_i)
\]

Instead of multiplying by 2, the result of the subtraction of \( a_i \) and \( t_i \) can be left shifted by one bit instead, which accomplishes the same result. The equation now reads:

\[
ps_i^M = 1 << (a_i - t_i)
\]

This concludes the description of the output sensitivity generators. They are incredibly simple, but are required to start the backpropagation process and get it rolling.

### 5.7 Alternative Design involving DRAM and Buffer

The design as it is presented in this chapter can only be used with small to medium sized neural networks. As mentioned in Chapter 3, modern neural networks can contain many thousands of neurons and millions of parameters. The presented design cannot handle networks of this size. There are simply too many neurons, weights, etc. to reasonably fit onto a single chip. This section will briefly describe an alternate design that solves this issue by adding a few modules and some additional functionality. This description will only detail the idea at a high level, as the specific design is outside the scope of this work. Figure 5.13 shows a block diagram of the alternate design.

Since every neuron in the network cannot fit on the same chip, the alternate design fixes this issue by only implementing the hardware for a few layers. The weights and biases will primarily be stored on the external DRAM memory until they are needed in the network. When required, the BISTr chip will load them from the DRAM chip into the memory buffer on the BISTr chip, and then into the registers in the network from there. In order to complete one forward propagation, the BISTr chip must follow these steps:

1. Load weights and biases for first set of layers from memory buffer to neuron registers
2. Present inputs to first set of layers and propagate forward
3. Store intermediate outputs from this set of layers in the memory buffer
4. Load the weights/biases for next set of layers from memory buffer to neuron registers
5. Propagate the intermediate outputs through the next set of layers

6. Repeat steps 3-5 until the final outputs have been calculated

Naturally, the hardware layers on the chip must be able to accommodate the biggest layers present in the network. Additionally, the BISTr controller must be modified to handle the swapping of weights/biases as well as managing the intermediate outputs. To increase the efficiency of this mode, the loads and stores can be overlapped with the network operation. For example, the next set of weights and biases can be moved from the DRAM to the memory buffer during step 2 or step 5 while the network is working. Data could also be moved back to the DRAM module while the weights and biases are being loaded into the network.

The backpropagation mode would operate very similarly in the alternate design. No modification needs to be made to the OSGs and boundary scan chain. After the partial sensitivities for the output layer are generated, backpropagation will continue for the number of hardware layers present on the chip. Once the lowest numbered layer of the set is reached, the intermediate sensitivities must be stored. The weighted sensitivities coming back along the Inputs lines would be reapplied on the Output lines once the weights and biases for the
next (lower numbers since the direction of traversal is backwards) set of layers are ready. As with feedforward mode, network execution can be overlapped with memory transfer between the DRAM module and the memory buffer. This would continue until all the sensitivities have been generated.

Similar to the other modes, weight updating must also be modified to be done in an iterative manner. The weight and biases for a set of layers, the intermediate outputs from forward propagation, and the sensitivities for those layers from backpropagation must all be loaded to perform the weight updating for that set of layers. The same process would continue for the other sets of layers until the weights and biases have been updated for the entire network. The next set of weights and biases could be pulled into the buffer while the current set is being updated.

All of these changes will increase the complexity of the BISTr controller tremendously. Not only will the feedforward, backpropagation, and weight updating modes change, but the controller must also multitask if the design is to be effective. The BISTr controller would have to manage all the loads and stores between the memory buffer and the partial neural network and the data transfer between the external DRAM module and the memory buffer. Alternatively, the memory management could be handled using instructions like those used in Tensorflow and other accelerators. The BISTr controller would still have to coordinate with the processor issuing these instructions, however. Adding this functionality to the BISTr controller would be the most difficult part of implementing the alternate design.
Chapter 6

Automatic Network Synthesis Tool

6.1 Synthesis Tool Overview

6.1.1 Introduction

The last chapter presented the design of the BISTr hardware. However, designing the hardware is not enough to complete the entire design process. The design needs to be verified to ensure that it is operating as expected. The process of taking the RT level design presented in the previous chapter and having a physical chip (or chips) fabricated for physical testing would be time consuming and outside the scope of this work. The goal is to present and verify that the idea of BISTr can work and evaluate how well it performs. Therefore, simulating the hardware network behavior using a hardware description language (VHDL in this case) will be sufficient.

Simulating the network using an HDL presents its own issue, however. Given the potentially large size of the networks and the sheer number of interconnects, writing all the code by hand would be lengthy and frustrating. To solve this issue, an automatic network synthesis tool was written. The program, written in C++, can generate the VHDL code for the network, as well as a testbench file. The synthesis tool exploits the regularity of the network to automatically generate all the neurons and connect them. This chapter will:

- Detail the inputs and outputs for the program
- Go through the flow of the program
- Give an in-depth description of the input file formats
- Provide examples of the input files
- Summarize how to use the synthesis tool
6.1.2 Program Inputs and Outputs

The synthesis tool accepts two files (one of which is optional) as its inputs. The first of these is a file that specifies the structure of the network. This file includes information such as the number of layers, the number of primary inputs, how many neurons are in each layer, and which activation functions the neurons have. In short, this file contains all the specifics needed to generate a neural network along with the corresponding BISTr hardware. The second input file (which is optional, but recommended) is a file describing a training process for the neural network provided in the other file. This file contains commands which allow the synthesis tool to generate a testbench that trains the network. The network specification file format is described in Section 6.2. Likewise, the training file specification is presented in Section 6.3.

The program will output one or two things depending on whether a training file was provided. It will always produce a set of files containing VHDL code that represents the entire network and all of its components. The specific files produced will be detailed in the program flow presented in Section 6.1.3. The files are divided up such that individual modules are easy to find. If a training file was submitted to the program, it will generate a testbench corresponding to the commands issued in the training file. The network code and testbench can then be loaded into an IDE or other simulator to be compiled and simulated. The synthesis tool receives the input file names as program arguments. The network spec file is the first argument, and the optional second argument is the training file. The files output by the program will be in the same folder as the executable file.

6.1.3 Program Flow

This section will detail how the synthesis tool works and break down the operation into a few major sections. The sections will cover how the program interprets the input files, as well as which output files are generated and how the program determines which files to generate. The explanations will be high level to avoid talking about the code at length (since the program is over 2000 lines long). Figure 6.2 shows the program flow for the synthesis tool.
Reading Network Parameters

In this portion of the code, the program opens the network spec file and reads in the information that the file contains. If the network file provided via the program arguments cannot be opened, the program terminates here. The data read by the program includes the number of layers in the network and the number of primary inputs. After receiving those two parameters, the program starts to read in information about the layers themselves. For each layer, the program stores how many neurons are in that layer, and which activation functions each neuron has. Once the network parameters have been read, the program moves on to the next section.

For example, regard the network in Figure 6.3. This network has 4 primary inputs and 3 layers. The first layer contains 4 neurons, each with 4 inputs, 3 outputs, and a logistic activation function. The second layer has 3 neurons with 4 inputs and 2 outputs. Two of the neurons in this layer have logistic activation functions while the other has a linear activation function. The third and final layer contains 2 neurons, both with 2 inputs, 1 output, and linear activation functions.
Generating Standard Components

After reading in the network parameters, the program generates the VHDL code for the standard components in the design. These components are common to networks of any size and do not change based on the size of the network. As such, their VHDL code is static and can be produced independently of the network parameters. All of them are required. They include standard logic cells (MUXes, DEMUXes, registers, TSBs, subtractors, and multipliers), the two types of subnodes (for weights and biases respectively), the output sensitivity generators (OSGs), and the BISTr controller.

The standard logic cells are all grouped in a single file named Standard Logic Cells.vhd. The two subnode files for weights and biases are Subnode.vhd and Bias Subnode.vhd respectively. The OSG module is contained in the file called Output Sensativity Generator.vhd. Lastly, the BISTr controller is written to the file BISTr Controller.vhd.

Generating Neuron Modules

Generating the neuron modules is where things get more complicated. While they contain the same parts, the neurons differ in the number of inputs and outputs they have, as well as their activation function. To generate all the different neurons required for the network, the synthesis tool iterates through the layers, generating neurons as needed.

Each layer has a fixed number of inputs and outputs. For the first layer, the number of inputs is equal to the number of primary inputs specified in the network file. For any other layer, the number of inputs is the number of neurons in the previous layer. Likewise, the number of outputs is equal to the number of neurons in the next layer. The final layer is the exception to this, where each neuron has only one output. The program tracks the number of inputs and outputs in each layer, as they are needed to generate the neurons in that layer.

The last information required to generate a neuron module is the activation function. Using the list stored when the network parameters were read in, the synthesis tool pulls the activation functions present in the given layer. The tool also keeps track of which types of neurons have already been created so duplicates are not produced. A duplicate is a neuron with the same number of inputs, outputs, and the same activation function. This is particularly helpful when all the neurons in a layer have the same activation function.

The naming convention of the neurons (for both the files and the modules) is as follows:

neuron_[# inputs]_[# outputs]_[activation function] (.vhd)

As such, a neuron with 20 inputs, 5 outputs, and a logistic activation function would have the following module name:

neuron_20_5_logsig (.vhd)
Once the synthesis tool decides that a given neuron needs to be created, it starts generating the code. The code creates the entity based on the naming convention just described. The input and output signals are all generated based on the number of layer inputs and outputs. Moving on to the architecture portion of the generated code, the components are largely independent of the size of the neuron. Only the activation function is different here. Next are the signal declarations. Many of these signals are only declared once, but some—like the scan chain signals—are dependent on the dynamic parameters like the number of inputs.

The description of the neuron’s architecture comes after the signal declarations. This largely consists of the module instantiations (creating a module of the given type and connecting it to the inputs, outputs, and other signals in the design). The synthesis tool creates the appropriate number of weight subnodes and the one bias subnode. It will also create the additional 2x1 MUXes in the event that the outputs outnumber the inputs. The tool instantiates the activation function module and the hardware to calculate its derivative. It also creates the correct number of TSBs for the output fanout. Next, the behavioral description is generated. The last section of code generated is the behavioral description of the activation function (in its own module and not in the neuron module code).

![Figure 6.4: Example network with neuron module names](image)

Following the example presented in Figure 6.3, four different types of neuron entities would need to be generated. Figure 6.4 shows the example with the types of neurons required as specified by the naming convention. The neurons in the first layer all have 4 inputs, 3 outputs, and logistic activation functions. Therefore, only one type of neuron needs to be generated for this layer (neuron\_4\_3\_logsig). In the second layer, one of the neurons has a linear function while the others have logistic. Therefore, two types of neurons need to be created for this layer (neuron\_4\_2\_logsig and neuron\_4\_2\_linear). The final layer is similar to the first, containing 2 neurons with 3 inputs, 1 output, and linear activation functions. Only
one type of neuron is required by this layer (neuron.3.1.linear), making the total number 4. The following is a complete list of neuron types as well as how many instances of them would be required to build the entire network:

- neuron.4.3.logsig(.vhd): 4
- neuron.4.2.logsig(.vhd): 2
- neuron.4.2.linear(.vhd): 1
- neuron.3.1.linear(.vhd): 2

Generating the Top Module

At this stage, all the components needed to construct the entire network have been generated. All that remains is to create the network in code and connect all the modules. The network will be contained in a module called the top module (the filename is Top Module.vhd). As with the neurons, the first step is to generate the entity and port descriptions. The primary inputs are inputs to the top module. Similarly, the network outputs are output from the top module. The next step is to list all the components in the design. This includes all the neuron modules, the BISTr controller, the OSG module, and a few others.

Depending on the size of the network, the top module can have a massive amount of intermediate signals. Between each layer and the next, there is a signal connecting every neuron in the given layer to every neuron in the next layer. For example, if the first layer had 12 neurons and the second layer had 6 neurons, there would be 72 signals between the two layers. The naming convention for these signals involves the numbers of the two layers (e.g. 5 and 6) and the numbers of the neurons the signals connect (e.g. 10 and 5). For the example in Figure 6.5, the signal between neurons lay0_neuron3 and lay1_neuron2 would be called lay0.1.nrn3.2. Scan chain signals for the internal neuron scan chain and the expected output scan chain, as well as control signal lines are among the other signals generated.

The final section of the top module is dedicated to component instantiations. The code generated here can get quite lengthy if the network is big. The tool starts by generating code for all the neuron modules. A major part of the component instantiation involves mapping the signals in the top module to the ports on the component. For the control signals, this is easy. The neuron I/O is a bit more complicated to map, but is made easier by the naming convention mentioned in the previous paragraph. Using that convention, the input and output signals can be hooked up via simple for loops. The tool goes layer by layer, neuron by neuron. The names of the instances of the neuron modules from the example in Figures 6.3 and 6.4 are shown in Figure 6.5. After generating all the neuron modules, the synthesis tool generates the hardware at the output, including the OSGs, boundary scan registers, and output TSBs. The last module created is the BISTr controller.
Generating the Testbench File

The last section of code will only execute if a training file is provided. This section generates a testbench for the top module based on the training file. It reads from the training file and generates the appropriate code based on the contents of the training file. The file output by this section is called Top Module Testbench.vhd.

Since this file is a testbench, the testbench entity has no inputs or outputs. All the I/O to and from the top module is handled by internal signals to the testbench. The only component in the testbench is the top module. The signals generated by the testbench are the simulated inputs for the top module and outputs to capture the module’s response. After the signals comes the instantiation of the unit under test (the top module).

The testbench has two behavioral processes. One is a simple clock that is fed to the design. The second is the test process that executes the testbench. The first part of the test process involves initializing the input signals. After that, the BISTr controller is reset to NORMAL_MODE. The last part of the setup of the testbench involves scanning the weights and biases into the neurons so the network is ready for training.

The next part of the test process depends on the training file. Here, the program reads in a command from the file (specific commands are detailed in Section 6.3), and generates the corresponding code based on that command. This is where a training cycle can be run or the response of the network can be displayed. The section goes on as long as there are commands in the training file. Once the training file has been completely read in and the testbench code generated, the test process is ended and the testbench file wrapped up.


6.2 Network File Structure

This section will cover the network specification file format that the synthesis tool uses to generate the VHDL code. The file has two main parts, the header and the layer descriptions. The format for both is provided in the following sections, with a complete example appearing in Section 6.4.

6.2.1 Header Format

The header of the network file contains two important pieces of information: how many layers the network has and the number of primary inputs. The number of layers should be greater than or equal to one. The same is true for the number of primary inputs. The number of layers should be on the first line of the text file, with the number of inputs on the second. Examples are provided below:

ex. 1
[# layers]
[# inputs]

ex. 2
8
64

6.2.2 Layer Format

After the header, the file specifies the structure of the layers. The program throws away the line following the number of inputs so it can be used as a dividing line. Any characters on this line will work, except for nothing or only spaces. Each layer has three pieces of information that the program needs to describe the layer. The first is the number of neurons in the layer. This goes on the first line for given layer. The second line contains a string specifying whether all the neurons in the layer have the same activation function, or different activation functions. Based on the second input, the program will respond differently to the third piece of information. If the string is “same”, the program only reads the next line before ending the reading for this layer. This line contains the activation function string which tells the program which activation function to insert. Currently, the two options are a linear function and a logistic function, though this could be extended. However, if the string is “different”, the synthesis tool will read in the next number of lines equal to the number of neurons in the layer. Each line contains the activation function for the respective neuron. This must be used even if all but one of the neurons have the same activation function. Examples are
provided below:

ex. 1
[# neurons in the layer]
[“same” or “different”]
[first activation function]
: (if necessary)

ex. 2
20
same
logsig

ex. 3
5
different
logsig
linear
logsig
logsig
linear

6.3 Training File Structure

Automating the testbench generation process is perhaps the most useful part of the synthesis tool. Generating the code for the network is a vital piece, but automatically generating a testbench file to simulate the training of the network helps tremendously. The training file is organized similarly to the network file. There is a header, followed by commands and their arguments, if necessary. Unlike the network file, the training file must be terminated with the string “end_file” so the program knows when to terminate its while loop.

6.3.1 Header Format

The file begins with a one-time command that dictates whether the initial weights and biases scanned into the network are randomly generated by the program or provided by the user. The two commands for these respective options are “random” and “choose”, respectively. If random is chosen, no additional arguments are required. If “choose” is selected, the weights and biases must be provided afterwards, one per line. Keep in mind that the weights and biases will be scanned in the order provided, which means the last value provided will end
up in the first weight of the first neuron in the first layer. Refer back to Section 5.5.1 for a more specific example. NOTE: the number of weights and biases provided must match the length of the internal scan chain. A few examples are provided here:

ex. 1
   [“random” or “choose”]
   [weights and biases if “choose” selected]

ex. 2
   random

ex. 3
   choose
   0.25
   -0.9375
   0.4375
   0
   0.75

6.3.2 Training Commands

The synthesis tool offers six commands that can be used for the training process. As with the network file, the commands are divided by throwing away a line in between them. The command string is read in first, followed by any arguments, if necessary. This section will detail all the commands and how they’re used.

**set_learn_rate**

This is a simple command which allows the learning rate to be adjusted. It is worth nothing that the learning rate is initialized to zero in the testbench, so this command should be used at least once before the training starts. The learning rate is provided as an argument on the next line. An example of the command usage is provided below:

ex.
   set_learn_rate
   0.1825
loop and end_loop

VHDL has a functionality which allows for sequences to be looped. This is particularly helpful since it prevents repeated code and saves time. In the context of neural network training, the looping functionality can be used to repeat going through the training set as many times as the user desires. To accommodate this functionality in VHDL, the synthesis tool allows the user to insert loops wherever appropriate. Everything after the loop command in the file up until the end_loop command is issued will be included in the loop. Loops can be nested if desired. The only argument the loop command needs is the number of iterations to loop. The syntax of the commands is as follows:

ex.

```
loop
100
```

.. (other commands)

```
end loop
```

display

This command allows the user to display the network outputs for a given set of inputs. This can be done at any point to monitor the training process, or at the end to verify correct behavior. The arguments for this command are the inputs that will be fed to the network. The outputs can be observed in the simulation waveform at the point where the display command appears in the order of the training file. The inputs are given in ascending order (for this example, 15.9375 is first \( p_1 \), -2 is second \( p_2 \), and so on). The syntax for a network with 5 inputs could be:

ex.

```
display
15.9375
-2
0
6.25
0.875
```
training_set

This command is the backbone of the testbench generation process. It generates the code needed to perform one complete training iteration. This includes the feedforward, backpropagation, and weight updating steps. Two sets of data are required by this command. The first is the set of expected outputs for the given test inputs. Remember that the expected outputs should be provided in scan order, which means that the first expected output is provided last. For reference, the detailed scan chain design is presented in Section 5.6. After the expected outputs, there is another dividing line to separate the data visually for the user. The second set of data provided are the test inputs for that training iteration. As with the display command, they are provided in ascending order. A training set command for a network with four inputs and three outputs is provided below. The expected outputs are 10, 20, and 30. The test inputs are 16, 32, 64, and 128.

ex.

    training_set
    30
    20
    10

    16
    32
    64
    128

scan_weights_and_biases

The last command is used for scanning the weights and biases. It functions identically to the choose command in the header, but can be issued at any time. The arguments are the weights and biases to be scanned in. The weights and biases currently stored in the neurons will be scanned out (and can be observed) while the new weights and biases scan in. This is particularly useful for viewing the weight and bias values after the training process has been completed. In the following example, 6.25 is the bias of the last neuron in the final layer and 0.0625 is the first weight of the first neuron in the first layer. The scan chain order is explained more thoroughly in Section 5.5.1. The syntax is as follows:

ex.

    scan_weights_and_biases
    6.25
    6.1875
6.4 Example

This section will present a complete network file and training file. Since this is an example, the network and training files presented will be focused around showing the syntax of the input files and how the tool works. As such, the training file will be largely nonsensical, and the network will not be trained for a specific purpose. The training file will be a more reasonable length to include in full as a result. Here is the network structure that will be translated into BISTr neural network code using the synthesis tool:

4 layers
8 primary inputs
1st layer: 20 neurons, same activation function (logistic)
2nd layer: 10 neurons, different activation functions
3rd layer: 8 neurons, same activation function (linear)
4th layer: 4 neurons, different activation functions

Here is the network file that describes the network above:

4
8
___________
20
same
logsig
___________
10
different
logsig
logsig
linear
linear
logsig
linear
logsig
linear

; 0.125
0.0625
Now that the structure of the network has been detailed, the training file is next. For this “training set”, randomly selected initial weights and biases will be used. The first thing to do is set the learning rate, which will be set to 0.2 for this example. Next, a test pattern will be displayed. After this, the real training begins. The training process is contained in a loop that executes 30 times. The training set contains four elements. Recall that the expected outputs are provided first (in reverse order), then the corresponding inputs (in ascending order). After training, the network response to the four training inputs is displayed, followed by the test pattern provided at the very beginning. After this, the training file ends. The training file is provided below:

```plaintext
random

set_learn_rate
0.2

display
0.5
2
-0.1235
1.5467
-3.1415
7
0.9999
2.81828

loop
30

training_set
```
<table>
<thead>
<tr>
<th>training_set</th>
<th>-5</th>
<th>-2</th>
<th>10</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0567</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1456</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1234</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5698</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.623</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.3548</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.910</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
10
2
-2
-10

10
8
6
4
3
2
1
0

end_loop

display
0.0567
2.1456
1.1234
3.5698
5.623
8.3548
9.910
10

display
0.0567
5.87
1.1234
3.659
5.623
7
6.75
8

display
8
2
5
7
3
4
2
0
The complete list of files output by the program for this example is as follows:

Bias Subnode.vhd
BISTr Controller.vhd
neuron_8_1_linear.vhd
neuron_8_1_logsig.vhd
neuron_8_10_logsig.vhd
neuron_10_4_linear.vhd
neuron_20_8_linear.vhd
neuron_20_8_logsig.vhd
Output Sensativity Generator.vhd
Standard Logic Cells.vhd
Subnode.vhd
Top Module Testbench.vhd
Top Module.vhd

The Bias Subnode, BISTr Controller, Output Sensitivity Generator, Standard Logic Cells, and Subnode are the same for every network, and are always generated in the same way. The Top Module and Top Module Testbench are generated for every network, but vary. The
remaining neuron files comprise the types of neurons necessary to build the network. The first layer has 8 inputs, 10 outputs, and logistic activation functions (neuron_{8,1}_logsig). For the second layer, the neurons have 20 inputs, 8 outputs, and linear or logistic functions (neuron_{20,8}_linear and logsig). In the third layer, the neurons have 10 inputs, 4 outputs, and linear functions (neuron_{10,4}_linear). Finally, the neurons in the last layer have 8 inputs, 1 output, and linear or logistic functions (neuron_{8,1}_linear and logsig).

### 6.5 How to Use the Synthesis Tool

This section will go through the process of using the synthesis tool. It will provide the steps necessary to generate a network and simulate it. The instructions are kept general to avoid being platform specific. It is assumed the user has knowledge of how to call the program from the command line and how to use an HDL simulator.

1. Write the network spec file as a plain text file
2. Write the training file as a plain text file
3. Place the network and training files in the same folder as the program executable file
4. Run the synthesis tool executable. The network and training files must be provided to the program via input arguments. A command line call of the program in Linux would look like this:

```
./AutomaticNetworkGenerationTool network1.txt net1_train.txt
```
5. Create a project in an HDL simulator program (e.g. ModelSim)
6. Import all of the VHDL source files generated by the synthesis tool
7. Simulate the top_module_tb entity
8. Observe the simulation results
Chapter 7

Experimental Results

7.1 Application – Function Approximation

Function approximation was the application used to test the BISTr hardware. It was selected because it is easy to understand without requiring additional research or background. Additionally, the network structures used to achieve the function approximations in this section were not too large, which networks for other applications like image classification can be. This chapter will present several functions that will be approximated by networks simulated both in MATLAB and in VHDL with the BISTr circuitry.

The goal of function approximation is simply to train a neural network such that its output matches that of a given mathematical function over a certain range. The functions that will be approximated here all have one input and one output only. All of them use a similar network structure. The structure can be described as 1-n-1, indicating the network has one input, n neurons in the first layer, and one neuron in the second layer. All of the neurons in the first layer of these networks have logistic activation functions. The second neuron in all of the networks has a linear activation function.

The results of the simulations will be presented as graphs, where the mathematical function and the approximation are plotted together. In addition, the square error between the real function and the approximation will be plotted over the range of the approximation. Some numerical results such as the mean square error and average error over the range will also be included in a table following the graphs.

The following list contains all the functions that were approximated using the neural network. For each function, information such as the range of the approximation, the step between elements in the training set, and the size of the training set will be provided. Also included are parameters about the network like the number of neurons in the first layer, the number of passes through the training set, and the learning rate used for the training. A static number of passes through the training set was used instead of stopping once the error
was within a certain range because the latter is difficult to implement in a VHDL simulation. For reasons presented and discussed later, the VHDL simulations would likely have not terminated if the latter method was used. The number of passes used in both simulations was determined by testing the parameter in MATLAB first. The same parameters were used for the MATLAB and VHDL simulations.

### 7.1.1 Arctangent Function

\[ y = \frac{16}{\pi} \arctan(x) \]  

Range: \([-20, 20]\)  
Step: 1  
Size of training set: 41 elements

Number of neurons in first layer: 5  
Number of passes through training set: 300  
Learning rate: 0.125  
Number of network parameters (weights/biases): 16

### 7.1.2 Cubic Function

\[ y = x^3 - 6x^2 + 8x = x(x - 2)(x - 4) \]  

Range: \([0, 4]\)  
Step: 0.125  
Size of training set: 33 elements

Number of neurons in first layer: 12  
Number of passes through training set: 1500  
Learning rate: 0.0625  
Number of network parameters (weights/biases): 37

### 7.1.3 Quadratic Function

\[ y = x^2 - 7x + 10 = (x - 2)(x - 5) \]
Range: [1, 6]  
Step: 0.125  
Size of training set: 41 elements

Number of neurons in first layer: 32  
Number of passes through training set: 1200  
Learning rate: 0.0625  
Number of network parameters (weights/biases): 97

### 7.1.4 Absolute Value Times Sine Function

\[ y = |x| \sin(x) \]  
(7.4)

Range: [−5, 5]  
Step: 0.25  
Size of training set: 41 elements

Number of neurons in first layer: 24  
Number of passes through training set: 1200  
Learning rate: 0.0625  
Number of network parameters (weights/biases): 73

### 7.1.5 Sine Plus X Function

\[ y = \sin(x) + x \]  
(7.5)

Range: [−6, 6]  
Step: 0.25  
Size of training set: 49 elements

Number of neurons in first layer: 20  
Number of passes through training set: 1200  
Learning rate: 0.0625  
Number of network parameters (weights/biases): 61
7.2 MATLAB Simulation Results

Each of the functions was simulated three separate times in MATLAB using a script. The MATLAB simulations start with randomized weights and biases from -1 to 1. The data representation used in MATLAB is double precision floating point. The graphs of the networks after training are presented in the following subsections, organized by function. Table 7.1 contains the statistical data for all the functions, and is located in a subsection at the end of this section.

7.2.1 Arctangent Function

![Graph of Arctangent Function Approximation](image)

![Graph of Square Error](image)

Figure 7.1: Arctangent function approximation simulation #1 in MATLAB
Figure 7.2: Arctangent function approximation simulation #2 in MATLAB

Figure 7.3: Arctangent function approximation simulation #3 in MATLAB
7.2.2 Cubic Function

Figure 7.4: Cubic function approximation simulation #1 in MATLAB
Figure 7.5: Cubic function approximation simulation #2 in MATLAB

Figure 7.6: Cubic function approximation simulation #3 in MATLAB
7.2.3 Quadratic Function

Figure 7.7: Quadratic function approximation simulation #1 in MATLAB
Figure 7.8: Quadratic function approximation simulation #2 in MATLAB

Figure 7.9: Quadratic function approximation simulation #3 in MATLAB
7.2.4 Absolute Value Times Sine Function

Figure 7.10: Absolute Value Times Sine function approximation simulation #1 in MATLAB
Figure 7.11: Absolute Value Times Sine function approximation simulation #2 in MATLAB

Figure 7.12: Absolute Value Times Sine function approximation simulation #3 in MATLAB
7.2.5 Sine Plus X Function

Figure 7.13: Sine Plus X function approximation simulation #1 in MATLAB
Figure 7.14: Sine Plus X function approximation simulation #2 in MATLAB

Figure 7.15: Sine Plus X function approximation simulation #3 in MATLAB
7.2.6 Numerical Results

All of the MATLAB simulations yielded accurate training results, which are displayed in Table 7.1. The mean square error was typically on the order of $10^{-3}$ with a few exceptions that were in the hundredth place. The percent error column is calculated by taking the mean error, dividing it by the vertical range of the function (for the horizontal range covered by the approximation), then multiplying by 100. It shows the average amount the approximation deviates from the real function compared to the total vertical range of the function. Most of the simulations had a percent error of less than 1%, with the maximum being 1.6%.

<table>
<thead>
<tr>
<th>Function</th>
<th>Sim #</th>
<th>Mean Error</th>
<th>Mean Square Error</th>
<th>Mean Error /Range</th>
<th>Percent Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arctangent</td>
<td>1</td>
<td>0.033523379</td>
<td>0.001747433</td>
<td>0.002095211</td>
<td>0.209521118</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.019785759</td>
<td>0.000616475</td>
<td>0.00123661</td>
<td>0.123660993</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.032665766</td>
<td>0.001471578</td>
<td>0.003201611</td>
<td>0.20416104</td>
</tr>
<tr>
<td>Cubic</td>
<td>1</td>
<td>0.09901955</td>
<td>0.011877642</td>
<td>0.015884038</td>
<td>1.588403822</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.052008493</td>
<td>0.004606123</td>
<td>0.008342846</td>
<td>0.834284632</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.089618594</td>
<td>0.018974934</td>
<td>0.014376001</td>
<td>1.437600123</td>
</tr>
<tr>
<td>Quadratic</td>
<td>1</td>
<td>0.049230528</td>
<td>0.003636516</td>
<td>0.007876885</td>
<td>0.787688453</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.068906378</td>
<td>0.008412581</td>
<td>0.011025021</td>
<td>1.102502052</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.051666359</td>
<td>0.005751844</td>
<td>0.008266617</td>
<td>0.826661736</td>
</tr>
<tr>
<td>AbsTimesSine</td>
<td>1</td>
<td>0.079757466</td>
<td>0.011912687</td>
<td>0.008317389</td>
<td>0.831738938</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.048422862</td>
<td>0.004201249</td>
<td>0.005049707</td>
<td>0.50497066</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.068301247</td>
<td>0.007913541</td>
<td>0.007122695</td>
<td>0.712269454</td>
</tr>
<tr>
<td>SinePlusX</td>
<td>1</td>
<td>0.050527021</td>
<td>0.005438567</td>
<td>0.004162466</td>
<td>0.441624638</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.047638998</td>
<td>0.004950677</td>
<td>0.004163823</td>
<td>0.41638256</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.044556691</td>
<td>0.003555007</td>
<td>0.003894148</td>
<td>0.38941772</td>
</tr>
</tbody>
</table>

Table 7.1: Table of numerical results for all MATLAB simulations

7.3 VHDL Simulation Results

As with the MATLAB simulations, each of the functions was simulated three times in VHDL, each with a different set of randomized initial weights. To match the MATLAB simulations, the range of the random values for the initial weights was from -1 to 1. As mentioned before, the data representation used in the VHDL Simulations is signed, 16-bit, fixed-point numbers with 12 bits in the integer portion and 4 in the decimal. Graphs of the final results of the simulations are provided in the following subsections, with the statistical results compiled in Table 7.2 at the end of the section.
7.3.1 Arctangent Function

Figure 7.16: Arctangent function approximation simulation #1 in VHDL
Figure 7.17: Arctangent function approximation simulation #2 in VHDL

Figure 7.18: Arctangent function approximation simulation #3 in VHDL
7.3.2 Cubic Function

Figure 7.19: Cubic function approximation simulation #1 in VHDL
Figure 7.20: Cubic function approximation simulation #2 in VHDL

Figure 7.21: Cubic function approximation simulation #3 in VHDL
7.3.3 Quadratic Function

Figure 7.22: Quadratic function approximation simulation #1 in VHDL
Figure 7.23: Quadratic function approximation simulation #2 in VHDL

Figure 7.24: Quadratic function approximation simulation #3 in VHDL
7.3.4 Absolute Value Times Sine Function

Figure 7.25: Absolute Times Sine function approximation simulation #1 in VHDL
Figure 7.26: Absolute Times Sine function approximation simulation #2 in VHDL

Figure 7.27: Absolute Times Sine function approximation simulation #3 in VHDL
7.3.5 Sine Plus X Function

Figure 7.28: Sine Plus X function approximation simulation #1 in VHDL
Figure 7.29: Sine Plus X function approximation simulation #2 in VHDL

Figure 7.30: Sine Plus X function approximation simulation #3 in VHDL
7.3.6 Numerical Results

Of the five functions tested, only the arctangent function produced good results. The mean square error for all three trials was less than 1, and the average error deviated by no more than 5% when compared to the total vertical range of the function. The other functions had mean square error values ranging from 1.37 to as high as 70.44. The percent error values were also high, with many of them being in the range of 25 to 50%. The quadratic and absolute value times sine functions were consistently better than the cubic function. The sine plus x function was the most accurate if the third trial is excluded. The reason for the large discrepancy between the MATLAB and VHDL simulations is due to the data representation used in the VHDL code. Many *weight changes* that occurred in MATLAB were too small for the VHDL code to handle, which prevented the weights from being altered and the training completed as it was in MATLAB. Better normalization (or scaling of the values) could have helped to alleviate this issue. The BISTr hardware operated as expected in these scenarios, even if the end result produced was not as desired. A more detailed discussion of the cause and ways that it can be fixed can be found in Section 8.1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Sim #</th>
<th>Mean Error</th>
<th>Mean Square Error</th>
<th>Mean Error /Range</th>
<th>Percent Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arctangent</td>
<td>1</td>
<td>0.591463415</td>
<td>0.508610413</td>
<td>0.036966463</td>
<td>3.696646341</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.397007908</td>
<td>0.239662358</td>
<td>0.024812994</td>
<td>2.481299426</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.728658537</td>
<td>0.6468187</td>
<td>0.045541159</td>
<td>4.554115854</td>
</tr>
<tr>
<td>Cubic</td>
<td>1</td>
<td>2.767637311</td>
<td>13.36372699</td>
<td>0.443965427</td>
<td>44.39654268</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.546638258</td>
<td>11.80363141</td>
<td>0.408514272</td>
<td>40.85142719</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.304924242</td>
<td>15.82703931</td>
<td>0.530153317</td>
<td>53.01533175</td>
</tr>
<tr>
<td>Quadratic</td>
<td>1</td>
<td>1.576219512</td>
<td>4.401129002</td>
<td>0.252195122</td>
<td>25.2195122</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.328506098</td>
<td>3.232493331</td>
<td>0.212560976</td>
<td>21.25609756</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.72179878</td>
<td>5.361256669</td>
<td>0.275487805</td>
<td>27.54878049</td>
</tr>
<tr>
<td>AbsTimesSine</td>
<td>1</td>
<td>2.477046983</td>
<td>9.656261166</td>
<td>0.258315182</td>
<td>25.83151817</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.91540221</td>
<td>11.34386287</td>
<td>0.304028408</td>
<td>30.40284084</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2.585228417</td>
<td>10.11060068</td>
<td>0.269596723</td>
<td>26.95967227</td>
</tr>
<tr>
<td>SinePlusX</td>
<td>1</td>
<td>0.920195249</td>
<td>1.370753613</td>
<td>0.080428429</td>
<td>8.042842906</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.122195359</td>
<td>2.195657412</td>
<td>0.098083977</td>
<td>9.808397721</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6.25656119</td>
<td>70.43814876</td>
<td>0.546846322</td>
<td>54.68463223</td>
</tr>
</tbody>
</table>

Table 7.2: Table of numerical results for all MATLAB simulations
Chapter 8

Discussion

8.1 Experimental Results

The simulations performed in MATLAB served a few purposes. The first of these was to provide a point of comparison for the VHDL simulations. MATLAB also proved useful in testing out different network structures and testing parameters. It allowed for quick experimentation with the learning rate, range of approximation, and number of passes through the training set, to name a few. These details were fine-tuned in MATLAB such that the simulations produced reasonably accurate results in a timely manner. Of key importance was getting the required number of passes through the training set down for the sake of the VHDL simulations, which take much longer. MATLAB also had the added advantage of plotting the results in a way that made the quality of the training process readily apparent.

With the exception of the arctangent function, the VHDL simulation results are not even close to the MATLAB simulations. All of the components that make up the network hardware were tested from the bottom up to ensure proper functionality. That includes everything from the standard logic cells (i.e. registers, MUXes, multipliers, etc.) to the subnodes and neurons. Every component was verified to be working correctly. Since all of the components were functioning properly when tested on their own, the behavior of the network as a whole was perplexing. The conundrum was made all the more vexing because some of the test cases were clearly working.

A few avenues were explored in attempt to figure out the cause of the problem. The first involved checking the functionality of the entire network in VHDL to make sure everything was wired correctly and no errors were present in the synthesis tool coding. To do this, sets of weights and biases were trained in MATLAB (where the simulations were producing good results) and then scanned into the registers in the network using the neuron scan chain. If there were no errors in the network connections, one would expect the VHDL network response to match that of MATLAB. As seen in the following figures, putting the MATLAB weights in the VHDL network did produce good results as expected.
Figure 8.1: Arctangent approximation VHDL simulation using MATLAB weights

Figure 8.2: Cubic approximation VHDL simulation using MATLAB weights
Figure 8.3: Quadratic approximation VHDL simulation using MATLAB weights

Figure 8.4: Absolute Times Sine approximation VHDL simulation using MATLAB weights
Figure 8.5: Sine Plus X approximation VHDL simulation using MATLAB weights
These results confirmed that there were no wiring errors, and that the feedforward behavior of the network in VHDL was accurate. If not wiring errors, what could be causing the issue? The backpropagation process was deemed unlikely as the culprit since some functions were trained correctly. That left the weight updating process as the most probable cause. One hypothesis tested was that the VHDL simulation might take more passes through the training set to converge to a good result when compared to MATLAB. This was tested using the basic sine function (which had been trained properly). The number of passes through the training set was adjusted. The same initial weights were used to ensure any difference in the final result would be due to the length of the training process. The results of the simulations showed no difference as the number of passes increased past the amount needed to converge in the first place. Figure 8.6 shows this result for 300, 500, and 700 passes through the training set. As such, the hypothesis that more passes through the training set would help was discarded.

The last theory for the discrepancy between the MATLAB and VHDL results was a lack of precision in the data representation. As mentioned in Chapter 5, the VHDL simulations used a signed, fixed-point, 16-bit data representation. The numbers are scaled (by 16) such that 12 places are to the left of the decimal point and 4 are to the right (12 bits \(\cdot\) 4 bits). This representation was chosen early on in the research for the ease of coding, as well as avoiding large interconnect and data port sizes. In this representation, the smallest value that can be represented is 1/16, or 0.0625. For contrast, MATLAB uses double precision floating point numbers (which are 64 bits in size) in its calculations by default.

The issue causing the discrepancy between the MATLAB and VHDL simulations is that the amount by which the weights and biases are changing (\(\alpha s_i^m a_j^{m-1}\) and \(\alpha s_i^m\) respectively) is too small for the representation used in the VHDL code to represent. To check this
hypothesis, the changes in weights and biases were tracked in the MATLAB simulation. The changes in the MATLAB code were typically $10^{-4}$ to $10^{-6}$ in magnitude. Some were $10^{-1}$ or $10^{-2}$, but these were much less common. Since the smallest value the VHDL code can represent is $0.0625$ or $6.25 \times 10^{-2}$, many of the changes would get zeroed out because they are too small to represent. As a result, the weights would not change in the VHDL simulation when they are supposed to. What this means in the long term for the simulation is that the VHDL code will not accept changes when the error is too small.

Fortunately, the problem can be fixed. Increasing the learning rate ($\alpha$) does not help solve this problem because a higher learning rate has the consequence of adversely affecting the quality of the training process. To fix the precision problem, a few approaches could be taken. The first of these is to use a representation that allows for more bits of precision in the decimal point. That way, the changes on the order of $10^{-4}$ to $10^{-6}$ could be represented and not lost due to a lack of precision. Another approach would be using more bits to represent the data (e.g. 32 or 64 bits). Finally, using a floating point representation instead of fixed point. These fixes are not without their drawbacks, however. As mentioned before, using more bits means more interconnects, larger hardware modules, etc. The same goes for using a floating point representation, as floating point multipliers and adders are much more complicated than for fixed point numbers.

Given the size of the overhaul and time that would be required to fix the issue, implementing a fix is outside the scope of this work. The main goal of this work was to demonstrate that BISTr can work as a concept. Even though the first iteration design fell short of initial expectations, the efficacy of the design has still been demonstrated. With additional work, the flaws of BISTr could be addressed and the concept evolved. The potential is there, but BISTr needs more work before it can be practically implemented as a neural network platform.

8.2 Circuit Design

The circuit design went through a number of revisions before the final design presented in this thesis. Among the largest changes was a switch from a more serialized design to a parallel one. A couple factors influenced this decision. The first was the greater simplicity of a parallel design. In the serial design, weight updating was done at the neuron level instead of the subnode. This reduced the amount of hardware required, but increased the number of wires at the neuron level in addition to requiring the serialization to be monitored (to ensure all the weights were updated). The other reason a parallel design was chosen was the better speed performance it granted. With the parallel design, weight updating could be done in only two clock cycles instead of the number of weights stored at the neuron (which could be in the tens or hundreds). A serial design was also considered for the output sensativity generators, but was discarded for the same reasons.

The development of the subnodes was one of the major breakthroughs for the circuit
design. The major crux of making BISTr work involved reusing the hardware for the forward propagation step for backpropagation and weight updating. By breaking down the neuron level behavior for these processes into single operations, it became easy to see how the hardware could be reused. The subnode design focused around reusing the multiplication unit, since it was the most costly in terms of area. Using the multiplexers to control the operands for the multiplier was effective and convenient, since the control signals could be hooked up to the select bits. The subnodes became the building block for the rest of the neuron. Initially, the weight and bias subnodes were based on the same architecture, with a few alterations because the biases are processed differently than the weights (since they aren’t used in backpropagation). Upon further analysis, the bias subnode architecture was simplified and unnecessary components like the multiplexers were removed.

Since the weight updating was handled by the subnodes, the biggest hurdle in developing the neuron level architecture was performing the backpropagation behavior. Solving it required some derivation of additional equations from the standard backpropagation equations. Calculating the sensitivity was the first step in getting backpropagation to work. Viewing the sensitivity for the given neuron as a sum of the sensitivities from the next layer multiplied by the weight along the edge helped to make calculating it possible. After generating the weighted sum by reusing the summation unit (for feedforward propagation), multiplying by the derivative was the only step that remained. That and the weighting of the neuron’s sensitivity for backpropagation to the previous layer completed the neuron level design.

It was determined early on that bidirectional lines would be required to make BISTr work. Since there was already line from each neuron in one layer to the next, reusing them was a natural solution. The main issue then became how to control the data on the lines between the neurons. Luckily, the control signals that were used in the neurons and subnodes could also be used to control the flow of data between them. Tri-state buffers (TSBs) controlled by the signal feedforward or its complement were used to enforce the correct data on the bus. Isolating the I/O lines that used the bus from each other in this way prevented any spurious results. TSBs (and also registers) were used at the primary inputs and outputs to allow them access to the bidirectional lines without contaminating the information.

The BISTr controller was an integral part of the overall BISTr design. A FSM controller was adapted from the BIST VLSI paradigm for multiple reasons. Chief among these was making the interface easier for the user. The controller can be operated using a single digital signal (TMS). The BISTr controller also ensures that improper use of the BISTr hardware cannot occur. For example, control signals cannot be erroneously issued at the same time due to the controller. The BISTr controller also streamlines the training process, making it simple to execute. The BISTr controller also made the simulations in VHDL much more manageable.

The internal scan chain was necessary to allow the user the ability to access the weights in the neurons. The order of the registers in the scan chain was chosen to be logical (if a bit unintuitive). By comparison, the expected output scan chain is much more straightforward. The output sensativity generators are relatively simple modules, but they contain a couple
optimizations that reduce the amount of hardware required.

\section{Area and Performance Results}

Area overhead was of lesser concern when compared to performance and getting the design to work. One direct comparison was done with a 1-2-1 neural network approximating the function \( y = 8 + 8 \sin\left(\frac{\pi}{4}x\right) \) over the range \([-2,2]\). One of the two networks tested had the BISTr circuitry integrated, while the other did not. The overall design with the BISTr circuitry had around 3 times the area of the neural network circuit without it. Some BISTr components (like the BISTr controller, which only appears once) would have less impact with a neural network of a greater size. The multiplication units were by far the most costly unit in terms of area. In the BISTr design, the bias subnode has a multiplier that is not present in the non-BISTr design. For smaller neurons (like the ones in this test), this makes more of a difference. However, this detail does not change the fact that the BISTr circuitry takes up a substantial amount of area. Any area improvements that could be made in a future revision could have a substantial impact on the area cost of the overall design. More extensive area testing is required to get a better picture of which components contribute the most to the area of the design and where the cost could be reduced.

As the design progressed, the speed performance of BISTr became the primary goal. This goal influenced the decision to switch to a parallel based weight updating design. In theory, one iteration of training could be completed by the BISTr hardware in the number of clock cycles given by the following equation (where \( M \) is the number of layers and \( m \) is the number of outputs from the final layer \( M \)):

\[
CCT = 2 + m + 2M + 2 = 4 + m + 2M
\]  

(8.1)

The equation can be broken down as follows. Two clock cycles are required to start the training process. Then, it takes \( m \) clock cycles to scan in the expected outputs for the giving training iteration. Next, \( M \) clock cycles are required to propagate the data forward through the network. Likewise, \( M \) more clock cycles are needed to backpropagate the sensitivities. The final step is to update all the weights and biases, which takes two clock cycles.

The VHDL simulations performed for Chapter 7 used a clock period of 50 ns. This could be reduced further depending on the timing constraints of the hardware used to implement the BISTr network. Aside from the arctangent and cubic functions, which required 300 and 1500 passes through the training set respectively, the other functions required 1200 passes. The total simulation times ranged from around 7.39 ms for the arctangent to 35.29 ms for the sine plus x function. However, these times are only hypothetical. Since most neural networks would be too big to implement using the version of BISTr tested in this thesis, the times should be viewed more as the theoretical upper bound on the performance of the BISTr paradigm.
Chapter 9

Conclusion and Future Research

9.1 Conclusion

Built-In Self-Training was conceived by combining the Built-In Self-Testing VLSI paradigm with the backpropagation training method for neural networks. As neural networks and deep learning have become more prevalent in computing, the need for faster training and more effective hardware solutions has increased. The goal of BISTr was to create a circuit design that could perform both inferences and training using the same hardware in an efficient manner. The idea was initially envisioned as a self-contained design that could be implemented on a single chip. The core problem to be solved was to reuse the hardware involved in forward propagation (or inference) for the backpropagation algorithm. In particular, BISTr aimed to use parallelization to speed up the training process.

Adapting the backpropagation algorithm to a hardware based implementation required analysis and derivation of some new equations. The impetus for deriving the new equations was to find an efficient way to divide and conquer the sensitativity calculation so each of the neurons in a given layer could operate independently. Once the equations were derived, the hardware design followed suit. The components were tested using VHDL from the ground up, starting with the basic components like the multiplexers, multipliers, and registers. One key component introduced in the design were the subnodes, compact blocks within the neurons that handled the operations involving the weights in feedforward, backpropagation, and weight updating modes. Once the neuron design had been finalized, the other elements of BISTr like the controller, scan chains, and output sensitativity generators were added until a final version could be tested. However, a more efficient way of generating the code for the networks was needed to avoid the long and tedious process of coding them by hand.

An automatic synthesis tool was created with the intent of making the experimental phase easier. The synthesis tool takes formatted text files that specify the structure of the network and how it is to be trained and generates all the necessary VHDL code to represent and simulate the network. The tool exploits the regularity of the network structure to help
generate the code. The module generation and automatic wiring of signals between modules can be performed using cleverly-operated for loops. The tool was coded in C++ using a command line interface for passing the program the network and training files. The UI was kept simple to focus on the accuracy of the tool and to ensure the code generated compiled easily. Since the tool was developed to fit the needs of this work, the functionality of the current version is somewhat limited. Still, the synthesis tool was excellent for its intended purpose and helped expedite the experimental process tremendously.

The experimental results were not quite as good as they were expected to be, though not for the reason one might expect. The experiments involved approximating five different functions (six if the one used in functional testing of the hardware is included) using neural networks. The five different functions were simulated in MATLAB first to provide a reference for the BISTr results to be compared against. The BISTr simulations were performed in VHDL using the code generated by the synthesis tool. A couple of test cases were handled by the BISTr circuitry fairly well. While not ideal, the results for these cases were reasonably accurate. The other functions did not produce such great results. After investigation, the cause of the erroneous behavior was discovered to be the data representation used in the VHDL code. The hardware modules were operating properly, but the issue came from the amounts by which the weights were changing. In the MATLAB simulation, the weights would often change by values on the order of $10^{-4}$ to $10^{-6}$. These amounts were too small for the representation used in VHDL to handle, so they were treated as zeros and did not alter the weights as intended. The problem can be fixed if a better representation is used.

In the end, BISTr did successfully translate the backpropagation algorithm into a hardware solution. While the experimental results were not quite what was expected, the issue was not with the BISTr hardware but with the data representation used. Analysis of the area and performance of BISTr showed that, while BISTr was effective from a performance standpoint, the area overhead was too great for BISTr to be implemented as a standalone solution except for networks of small to medium size. This prompted the proposal of an alternate design with a memory buffer and external DRAM module that functioned more like a neural network accelerator. This work has provided a basis for the concept and the initial design. The results show that BISTr does work, but needs to be refined and developed further before it can be realistically implemented. The next section contains suggestions for a few ways in which this research can be expanded upon and evolved.

### 9.2 Future Research

#### 9.2.1 Additions to the Synthesis Tool

The synthesis tool was initially conceived as a way to make the experimental phase faster and easier. Writing the code for the networks tested in the experimental results would have been very long and tedious to do by hand. The tool was developed to suit the needs of this
work. As such, it is a simple program written in C++ to be used on the command line. Artificial neural networks can be much more complex than the examples tested in this paper. In future work, the synthesis tool could be expanded to handle more of these complexities.

Among the additions that could be made to the synthesis tool would be more types of activation functions. The only two used in this work were logistic and linear functions. More functions could be implemented in the next version of the tool such as hard limit (hardlim), saturating linear, and rectified linear [28] (or positive linear) to name a few. Another addition would be to add other types of modules than fully connected neurons. For example, convolution and max pooling units that are popular in deep learning applications could be added. One major improvement that could be made to the tool would be to improve the interface. The command line interface and scripting files were chosen because they were easy to work with and expedited the progress on the paper. With enough time and effort, the synthesis tool could be implemented using a GUI. Additionally, the network and training inputs used to generate the network and testbench code could be reworked to be more user friendly. More training options for commands and ways to import the training set would make the synthesis tool much more robust.

9.2.2 Alternative Design with DRAM and Buffer

As discussed in the previous chapters, the design presented (and tested) in this paper is ideal, but unrealistic. While smaller networks like the ones tested in this work could be implemented on a single chip without hassle, modern neural networks tend to be much larger and contain thousands to millions of parameters. To help make the BISTr concept more realizable, the alternative design presented in Section 5.7 would need to be augmented and finished.

It is difficult to say (without a final design to test) how the design with the DRAM and memory buffer would perform compared to the idealized version presented here. The efficiency of the alternative design would depend on how well loads and stores between levels of the memory hierarchy are overlapped with execution of the BISTr circuitry. Finding these optimizations may not be too difficult, but reworking the BISTr controller to handle such overlaps will prove challenging. Developing this alternate design would be necessary for the BISTr concept to move from just an idea to an actual implementation.

9.2.3 Other Training Methods and Functionality

In its current form, BISTr only implements the backpropagation algorithm using steepest descent. Backpropagation was chosen because the hardware to implement it was not complicated. However, there are other training algorithms used in deep learning. Future work could investigate how these training algorithms might be implemented in a similar way to BISTr. This applies to both supervised and unsupervised learning. While unlikely that
different learning methods could coexist on the same chip, covering more learning methods would make the idea of BISTr more appealing as a whole.

In addition to implementing other training methods, more functionality could be added to BISTr to make it more capable. The version presented in this paper was fairly simple in its construction, covering the functionality that was absolutely necessary to function. While the learning rate (\(\alpha\)) can be adjusted between training iterations, that is the limit to the flexibility allowed by the hardware. A future implementation could add features which could enhance the quality or accessibility of the training process.

9.2.4 Area and Speed Performance

The main focus of this work was determining the viability of BISTr as a concept. Less attention was given to the areas of area and speed performance. The theoretical speed of BISTr is very attractive, but cannot be achieved except for small neural networks. In future work, more attention should be given to the area and speed, especially when it comes to optimizing these aspects. Lowering the area cost of adding BISTr circuitry would allow for more of it to be fit onto a single chip. When it comes to speed, the goal would be to drive performance as close to the theoretical upper limit as possible.
References


