TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

A Reconfigurable Logic Based Accelerator for Bioinspired DNN Architectures with Dendritic Structure and a Novel Learning Rule

---

Author:
Nikoletta PALATIANA

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Michail ZERVAKIS
Dr. Panayiota POIRAZI
(IMBB/FORTH)

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

April 10, 2024

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

A Reconfigurable Logic Based Accelerator for Bioinspired DNN Architectures with Dendritic Structure and a Novel Learning Rule

by Nikoletta PALATIANA

Artificial Neural Networks (ANNs) have been successfully used in Deep Learning architectures to solve a variety of challenging machine learning problems. Nevertheless they usually require a considerable amount of energy. In addition, they demonstrate weakness in continually learning new tasks without forgetting the previous ones. They require multiple sets of data and a considerable amount of trainable parameters. The brain, on the other hand, operates at a very low energy level without facing problems learning new things. By drawing inspiration from the human brain and overcoming the limitations of ANNs, the Poirazi lab at IMBB-FORTH developed a bio-inspired architecture that incorporates the dendritic structure and receptive field, along with a novel approach to Hebbian learning. In this thesis a lower-level Numpy implementation was developed based on their initial Keras implementation in order to analyze and understand this model and its training process in greater depth. This was followed by the design, implementation, and download of an FPGA-based architecture onto the Xilinx ZCU 102 board for training the ANN. Using the high parallelism and power efficiency of the FPGA, our architecture has accelerated training and reduced power consumption. In particular, our proposed FPGA implementation executes an epoch of training (for the MNIST dataset) in only 13.46 seconds rather than 490 seconds on the CPU (Keras) and 45 seconds on the GPU (Keras). Furthermore, it achieves 346 times greater energy efficiency than the CPU implementation (Keras) and 57.5 times greater energy efficiency than the GPU implementation (Keras).

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

A Reconfigurable Logic Based Accelerator for Bioinspired DNN Architectures with Dendritic Structure and a Novel Learning Rule

by Nikoletta PALATIANA

Τα Artificial Neural Networks (ANNs), έχουν χρησιμοποιηθεί με επιτυχία σε αρχιτεκτονικές βαθιάς μάθησης για την επίλυση μιας σειράς δύσκολων προβλημάτων μηχανικής μάθησης. Ωστόσο, προκειμένου να επιτύχουν τη μέγιστη απόδοση, απαιτούν συνήθως σημαντική ποσότητα ενέργειας. Επιπλέον, δυσκολεύονται στην εκμάθηση νέων εργασιών. Απαιτούν μεγάλο όγκο δεδομένων και σημαντικό αριθμό παραμέτρων. Ο εγκέφαλος, από την άλλη πλευρά, λειτουργεί σε πολύ χαμηλό επίπεδο ενέργειας χωρίς να αντιμετωπίζει προβλήματα στη συνεχή εκμάθηση. Αντλώντας έμπνευση από τον ανθρώπινο εγκέφαλο και προσπαθώντας να ξεπεραστούν οι περιορισμούς των ANNs, το εργαστήριο Poirazi στο IMBB-FORTH ανέπτυξε μια βιο-εμπνευσμένη αρχιτεκτονική που ενσωματώνει δενδριτική δομή και receptive field, μαζί με μια νέα προσέγγιση του Hebbian κανόνα. Με βάση την αρχική υλοποίηση στο Keras στην παρούσα διπλωματική αναπτύχθηκε μια προσέγγιση σε Numpy προκειμένου να αναλυθεί και να κατανοηθεί σε μεγαλύτερο βάθος αυτό το μοντέλο και η διαδικασία εκπαίδευσής του. Στη συνέχεια, σχεδιάστηκε, υλοποιήθηκε και μεταφορτώθηκε στην πλακέτα Xilinx ZCU 102 μια αρχιτεκτονική βασισμένη σε FPGA για τη διαδικασία εκπαίδευσης αυτού του βιοεμπνευσμένου ANN. Χρησιμοποιώντας τον υψηλό παραλληλισμό και την αποδοτικότητα ισχύος της FPGA, η αρχιτεκτονική μας κατάφερε να επιταχύνει την εκπαίδευση και να μειώσει την κατανάλωση ισχύος. Συγκεκριμένα, η προτεινόμενη υλοποίηση FPGA εκτελεί μια εποχή εκπαίδευσης (για το σύνολο δεδομένων MNIST) σε μόλις 13,46 δευτερόλεπτα αντί για 490 δευτερόλεπτα στην CPU (Keras) και τα 45 δευτερόλεπτα που απαιτεί η GPU (Keras). Επιπλέον, είναι 346 φορές πιο αποδοτική ενεργειακά συγκριτικά με την CPU και 57.5 φορές συγκριτικά με την GPU.

Acknowledgements

I want to express my sincere appreciation to my thesis advisor, Prof. Apostolos Dollas, for his guidance and support throughout this project. His patience, mentorship, expertise and feedback have been invaluable in shaping this work.

Furthermore, I would like to thank the rest of my thesis committee, Prof. Michail Zervakis and Prof. Panayiota Poirazi, for evaluating my work and the ICS/FORTH CARV team, especially Dr. Gregory Tsagatakis, for their guidance throughout this thesis.

In addition, I would like to express my gratitude to the Research Director Prof. Panayiota Poirazi and the Postdoctoral Researcher Dr. Spyridon Chavlis from the Poirazi lab of the IMBB-FORTH for providing me with this opportunity to work on this thesis. I would also like to thank Dr. Spyros Chavlis for the time he devoted to helping us understand the biological background of the bio-inspired features and for making sure that we did not feel intimidated by such an unfamiliar topic.

Also, I want to thank my friend and fellow student Lampros Pantzekos, with whom I spent many many hours studying, discussing and searching for solutions to our thesis.

Special thanks are due to my coworker and friend Dr. Panagiotis Mousouliotis. His expertise, and willingness to share ideas during our morning coffee breaks have significantly contributed to the development and refinement of this work.

To my family and friends, I am deeply thankful for their encouragement and understanding during this journey. Thanks for always supporting me.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **R**andom **A**ccess **M**emory
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
DDR4	**D**ouble **D**ata **R**ate type texbf4 memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GDDR6	**G**raphics **D**ouble **D**ata **R**ate type **6** memory
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**ight **P**erformance **C**omputing
LUT	**L**ook **U**p **T**able
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
RAM	**R**andom **A**ccess **M**emory
SDK	**S**oftware **D**evelopment **K**it
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SSE	**S**treaming **S**IMD **E**xtensions
SSD	**S**olid **S**tate **D**rive
TDP	**T**hermal **D**esign **P**ower
URAM	**U**ltra **R**andom **A**ccess **M**emory
USD	**U**nited **S**tates **D**ollar

Dedicated to my family and friends…

Chapter 1

Introduction

There have been many algorithms developed over the past few decades that can be used to enable a machine to predict and learn. A subset of artificial intelligence, machine learning, has been one of the most successful approaches in recent years. Among the main interests of researchers in this field is imitating the functions of the human brain. The human brain uses biological neurons to receive and transmit signals. There are billions of them inside the brain and they are interconnected. They receive nervous impulses through their dendrites, process this information in the soma, and then decide whether or not to send a neural impulse through its axon. Through synapses, nerve impulses are transmitted from one neuron to another. The most common way to model the human brain in computer science is using artificial neural networks. They consist of interconnected nodes (neurons) organized into layers, capable of learning complex patterns and making predictions from data through training algorithms. Typical ANNs have been widely and successfully used in demanding machine learning tasks such as computer vision, speech recognition, autonomous driving, etc. Despite this, their substantial energy consumption to achieve top performance raises serious concerns, while on the other hand, the brain consumes very little power (< 20 watts). In this regard, the question of human brain modeling remains very relevant. To resolve this problem, one approach is that the model should demonstrate a brain-like behavior based on principles of neurophysiology. Then, it is examined whether this approach is beneficial in terms of less energy consumption, better learning performance, and overcoming problems such as "transfer learning" and "catastrophic forgetting". Alternatively, a more neuromorphic hardware implementation could reduce energy consumption contrary to using CPUs and GPUs.

To achieve this, we consulted the opinion paper[1] of S. Chavlis and P. Poirazi

and studied their analysis of a bio-inspired neural network so that it could be combined with an FPGA implementation specifically designed to take advantage of the bio-inspired features.

1.1 Motivation

The typical artificial neural networks can become highly adept at solving a specific task, but they are usually unable to transfer those skills to another task without retraining, referred to as transfer learning, in contrast, the human brain can be trained to solve a particular problem with a small number of examples using very little energy and is able to transfer this knowledge to new tasks without retraining or erasing valuable prior knowledge. We have billions of neurons in our brains that transmit signals to and from the brain. Dendrites are structures inside of neurons that facilitate communication.. Dendrites act as filters by attenuating and modifying signals traveling to the cell. Even though dendrites do not transmit signals themselves, to the cell body, when their activity is additive, they can significantly affect the cell's activity. Moreover, dendrites are proposed to serve as the fundamental functional unit of the brain rather than neurons [2][3].

The dendritic structure, as well as other architectures and learning rules derived from biology, have demonstrated promising results in neural networks. It has been suggested that utilizing dendritic features can minimize resource requirements while maintaining optimal performance. As a result, challenges such as the transfer of knowledge and the continuous learning process could be addressed.

Further, dendritic structures appear to enhance sparsity in biological networks, a property associated with improved memory, and learning abilities [4][5]. Besides saving resources, sparsity enhances the network's discrimination capabilities[6] [7], such as the distinction between similar input patterns. Each soma is connected to its dendrites only, and each dendrite is connected to its synapses only. For example, leveraging the sparsification of dendritic ANNs, there is no need for a dropout layer, which is essential in denser counterparts, thus achieving a better saving of resources.

In most cases, ANNs are trained using the backpropagation algorithm. However, the latter has some drawbacks [8]. Among them is the inefficiency of calculating gradients for the whole network every time. The network

must also be symmetric and have multiple labeled training data and iterations. Most importantly, it fails in unsupervised learning. In contrast, biological plasticity rules can support both supervised and unsupervised learning. Therefore, we adopted a method that calculates the error locally within a layer without propagating the error backward. Our method is based on a biological property known as the Hebbian law. In accordance with this law, any two cells or systems of cells that are repeatedly active at the same time tend to become 'associated,' so that activity in one facilitates activity in the other. The Hebbian learning rule reflects a basic principle observed in biological neural systems, making it a biologically plausible learning model. There's no need for labeled data or explicit error signals during training. Additionally, it can support continuous adaptation and integration of new information.

Hebbian learning, however, tends to strengthen connections between neurons that fire together, which can lead to indiscriminate learning. Because of this, it might not always capture the exact pattern or feature needed. In light of this, we decided to examine the algorithm developed by S.Chavlis and P.Poirazi based on the Hebbian law, which combines a typical artificial neural network with an unsupervised layer that uses covariance matrix to determine the relationship between active weights (synapsis) and dendrites. Chapter 4 will provide a more detailed explanation of this.

As with many ANNs, this model requires a substantial amount of energy and time as well as resources. Based on Bhaduri's claim[9] dendritic properties can efficiently be used for data classification in hardware, and dendritic structures can reduce training parameters efficiently while maintaining network quality. Thus, we decided to model this network using an FPGA, following our goal of reducing energy consumption.

1.2 Scientific Contributions

The thesis model was introduced by the Postdoctoral Researcher S. Chavlis and the Research Director P. Poirazi, both from the Poirazi lab of Institute of Molecular Biology and Biotechnology of the Foundation for Research and Technology-Hellas (IMBB-FORTH). A neuro-inspired ANN architecture that incorporates dendritic structure and receptive fields was proposed. Regarding the learning rule, classic backpropagation is applied in combination with an additional learning rule, the so-called 'Covariance rule' (plasticity rule) applied to the first layer of the network. For updating the parameters of

the network, Adam optimization algorithm is used instead of classical gradient descent. A high-level software implementation in Keras was developed by S. Chavlis as a first implementation for the training process of this bio-inspired model. This implementation serves as a reference for this thesis. In this thesis, a lower-level implementation in Numpy is developed as the first step in understanding and analyzing this model and its training process in depth. The main algorithms used in the training process are presented, along with an analysis (profiling) of their usage in terms of execution time and memory. This thesis aims to build an FPGA-based architecture for the bio-inspired ANN to accelerate its training process and further reduce power/energy consumption. Through their high parallelism and power efficiency, FPGAs are capable of achieving this. In this thesis, the FPGA-based architecture is designed, implemented, and downloaded onto the Xilinx ZCU 102 evaluation board. By comparing our proposed FPGA implementation to CPU/GPU (the reference implementation in Keras), we significantly improved latency, throughput, and energy efficiency. The contribution of this thesis can be summarized as follows:

- Lower-level software implementation for the training process of the bio-inspired ANN in Numpy.
- System modeling - Profiling.
- Design of the FPGA-based architecture for the training process of the bio-inspired ANN.
- Implementation of the FPGA-based architecture using Vivado tools.
- Downloading of the FPGA implementation onto Xilinx ZCU 102 board.
- Our proposed FPGA implementation provides a latency speedup of 35.66x over CPUs (Keras) and 5.13x over GPUs (Keras).
- Our proposed FPGA implementation achieves 346 times greater energy efficiency than the CPU (Keras) and 57.5 times greater energy efficiency than the GPU (Keras).

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** An introduction to ANNs and DNNs is provided, as well as an analysis of forward propagation, back-propagation, and Adam optimization algorithm processes.

- **Chapter 3 - Related Work:** The related work of certain bio-inspired models and techniques and some FPGA implementations is described along with our thesis approach.
- **Chapter 4 - System Modeling:** In this chapter, there is a detailed explanation of the thesis bio-inspired DNN model, including its features, connectivity structure, and functionality. An extensive description of the software (Numpy) implementation, algorithms, tools, and data set is also provided.
- **Chapter 5 - FPGA Implementation:** In this chapter, the FPGA-based architecture for the training process of the bio-inspired ANN is designed, implemented using Vivado tools, and downloaded onto the Xilinx ZCU 102 evaluation board. This chapter also provides information about the Vivado tools, the FPGA platform, the AXI4 Interface Protocol, the PL-PS communication methods, and the memory configuration.
- **Chapter 6 - Results:** In this chapter, performance metrics are analyzed, including throughput, latency, power consumption, and energy consumption, alongside a comparison of our FPGA-based architecture with CPU and GPU implementations.
- **Chapter 7 - Conclusions and Future Work:** There is a discussion of future directions and ideas for possible extensions in this chapter.

Chapter 2

Theoretical Background

This section analyzes and explains the structure of the neurons and their functions, making it easier to understand the neural network model and its organization. In addition, a brief description of the mathematical background is provided.

2.1 Neuroscience

Neuroscience is a critical field of study that examines the structure and function of the human brain and nervous system. With the human brain comprising over 80 billion neurons and brain cells, each cell having approximately a thousand connections to other cells, understanding how neurons combine synaptic inputs is crucial for comprehending how the brain works. Neurons, also called nerve cells, are the fundamental units of the nervous system responsible for receiving and integrating inputs created by the synapses, which are junctions between two neurons through which they interact. In each dendrite, there are synapses where the presynaptic neuron meets with another neuron (post-synaptic) soma or dendrites. A neuron can be simultaneously presynaptic and post-synaptic in different situations. Therefore, it is essential to understand that a typical neuron has three essential parts: the axon, the soma (also known as the cell body), and numerous dendrites.

Axons are the principal transmitting elements of neurons, exhibiting considerable variation in length, with some extending over a meter within the body. In contrast, dendrites are thin, branch-like structures that emanate from the cell body of neurons. Dendrites play a crucial role in receiving and processing information from numerous presynaptic inputs, often numbering in the tens or even hundreds of thousands, via small dendritic processes known as

dendritic spines [10]. The integration of this information by dendrites determines whether a neuron will initiate an action potential, which is an electrical signal that rapidly propagates along the axon, from its initial segment to the synapse, where it is transmitted to the post-synaptic cell. The axons of presynaptic neurons transmit signals to post-synaptic cells via branches, with a single axon often forming synapses with up to a thousand post-synaptic neurons.

Ramon y Cajal's connectional specificity principle [11] postulates that neurons do not form random connections with one another but rather form specific connections between pre- and post-synaptic neurons, which, in turn, form a complex network. Synapses, however, are not static but are capable of undergoing changes in connectivity and characteristics. These alterations, known as synaptic plasticity, can be pre- or post-synaptic, or both, and may be of short or long duration. Thus, the remarkable ability of neurons to learn, memorize, and repair damage is attributed to their capacity for synaptic plasticity.

2.2 Artificial intelligence

Artificial intelligence (AI) refers to the ability of a digital computer or computer-controlled robot to perform tasks that are usually associated with intelligent beings. In recent times, the focus of AI researchers has shifted from creating fully intelligent machines to finding ways of using AI to solve specific and complex problems. Machine learning is a branch of AI that involves data analysis. It operates on the principle that systems can learn from data without explicit programming. With the vast amount of data produced globally, referred to as Big Data, machine learning can work more efficiently. Using datasets, machine learning learns patterns and features with the goal of making predictions about new unseen data based on the knowledge it has previously acquired. One particularly interesting subset of machine learning is deep learning. Deep learning involves using multiple layers of simple, adjustable computing elements. Although deep learning has been around since the 1970s, it gained popularity in 2011 in the fields of speech recognition and computer vision. Deep learning is inspired by the human brain as it attempts to mimic the structure and functions of neural cells, which are the fundamental units of the nervous system.

2.3 Classification problem

One of the most common machine learning applications is classification. In this task, the computer must specify the category the input belongs to, given k number of categories. A basic example of a classification problem is the MNIST dataset, which includes images of handwritten digits from 0 to 9.

2.4 Supervised learning

Machine learning involves two major types of learning: supervised and unsupervised learning. Supervised learning is the process of learning a function that maps an input to an output based on previously known examples. To achieve this, a sufficient training set D of pairs of x and y is required, where x is the sample and y is the categorical label, a real number, or a combination of both. On the other hand, unsupervised learning is a branch of machine learning that involves algorithms extracting patterns from unlabeled data. Unlike supervised learning, it does not require explicit guidance to learn patterns and relationships among the data.

2.4.1 Simple Neural Network

Neural networks are a type of machine learning used in deep learning algorithms, and they have been inspired by the function of biological neurons. The Perceptron is the simplest type of neural network [12] [13], introduced by Frank Rosenblatt in 1957. The Perceptron is a linear binary classifier that consists of only one layer of input and output nodes. Each input has an associated weight. The perceptron operates as the function $f(x)$ that maps inputs to a single binary value. The input layer merely transmits the inputs and does not perform any computation, while the output node performs the dot product between the weights and the inputs. Once the product is computed, the step function turns the output into either binary 0 or 1.

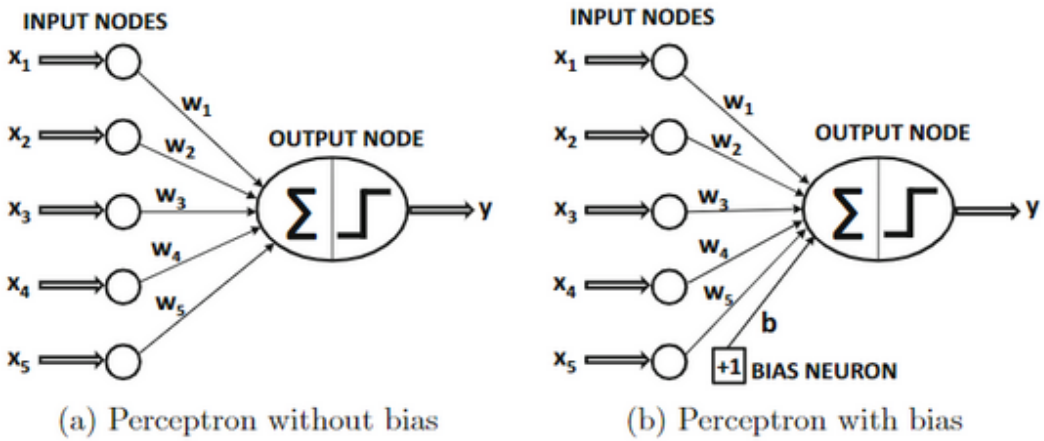

FIGURE 2.1: The basic architecture of the perceptron. Source: [14]

2.4.2 Artificial Neural networks

Structure

The perceptron is a type of neural network with a single computational layer. It can only distinguish binary values, which makes it useful for binary classification. However, it has a limitation in that it can only recognize objects that are linearly separable. In real-world scenarios, there are objects that are non-linearly separable, meaning they cannot be described with a linear function. To address this limitation, the Multilayer Perceptron (MLP) was introduced. The MLP is composed of many perceptrons or neurons, making it a type of Artificial Neural Network (ANN). A typical ANN is shown in the figure below 2.2.

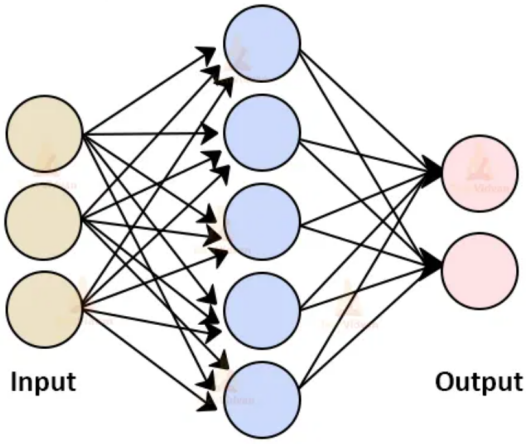

FIGURE 2.2: Architecture of an ANN. Source: [15]

The figure illustrates an ANN that comprises an input layer, a hidden layer, and an output layer. Each neuron in one layer forms connections only to neurons in the subsequent layer, and all connections have weights assigned to them. Unlike the perceptron, the ANN does not employ a step function, which is suitable mostly for linear computations. The use of non-linear activation functions plays a critical role in the network's ability to represent complex functions. Some of these activation functions will be introduced later in the document.

Data Handling

Neural networks are a type of supervised learning technique, which means that in order for them to function properly, they require a large data set. This data set should be divided into three smaller groups. The first is the training set, which is the actual set that the model learns from. The second is the validation set, which is used for fine-tuning the parameters of the model, such as the hidden units. The third and final set is the test set, which is used solely to assess the performance of the trained model.

Functionality

The training phase of an ANN involves feeding N pairs of training samples into the network to minimize the cost function.

Feedforward

When a neural network is fed data from the training set, it undergoes forward propagation. The data is received by the input layer and is then propagated to the hidden layers until it reaches the output layer. The output layer produces a vector of probabilities, which is also known as scores. The objective is to obtain the highest score that corresponds to the category label of the data. To evaluate the result, we need to compute a function that calculates the error or distance between the output scores and the desired ones. Typically, the network begins with a larger error that is gradually minimized during the training process until the model can be considered accurate[16].

Assume that X_i=[X_1,X_2,...,X_n] represents the inputs (of the input layer), $w_{i_j^1}$ represents the weight of node j ($j = 1, 2, \ldots, q$) associated with an input i, and b_j^1 represents the bias. A nonlinear activation function φ converts values into probabilities. The output, Y_j^1, of a node j in the first hidden layer ($h = 1$) is given by equation:

$$Y_j^1 = \varphi(\sum_{i=1}^{n}(w_{i_j^1} \cdot X_i + b_j^1)) \tag{2.1}$$

In the subsequent hidden layers and the output layer ($h = 2, 3, \ldots$), the output of a node j is expressed by equation:

$$Y_j^h = \varphi(\sum_{k=1}^{m}(w_{k_j^h} \cdot Y_k^{h-1} + b_j^h)), \tag{2.2}$$

where k corresponds to a node of the previous layer ($h-1$), Y_k^{h-1} is the output of a node k, m indicates the number of nodes in the previous layer and $W_{k_j^h}$ represents the weight associated between nodes k and j.

There are two common functions for calculating the errors used in different cases. Means Square Error is generally preferred for regression problems, while Cross-Entropy is for classification problems. These functions are called loss function or cost function. In our thesis, we work with a multi-class classification problem, so we will focus on the Cross-Entropy function, which is given by the following equation.

$$Error\_Function = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n}\left(y_{i_j} \cdot \ln(\hat{Y}_{i_j})\right) \tag{2.3}$$

The main objective of the backward phase in deep learning is to update the weights and biases accurately. This is done by calculating a gradient vector that indicates, for each weight, how much the error would increase or decrease if the weight is increased by a tiny amount. The weight vector is then adjusted in the opposite direction to the gradient vector[deep learning review]. To calculate the gradient of the loss function with respect to the different weights, the chain rule of differential calculus is used. This process is called the backward phase because the gradients are calculated in the backward direction, starting from the last layer.

The final step is the backpropagation of the error and the update of the weights.

2.4.3 Backpropagation

The feedforward process is followed by calculating the error. Afterward, the error signal is propagated backwards to all weights and biases, and based on that, these parameters are updated to improve the model. The Backpropagation process is based on the Gradient Descent algorithm, in which the gradient of the error function (E) is calculated and a step is taken (by adjusting the network's weights and biases) towards the negative direction of the gradient to decrease the error gradually. The process is the same for multi-layer perceptrons except that the error function is more complex.

Gradients are computed using a technique known as the chain rule. For a single weight $W_{k_j^h}$ associated with nodes k (of the previous layer $h-1$) and j (of the current layer h), the gradient is as follows:

$$\frac{\partial E}{\partial w_{k_j^h}} = \frac{\partial E}{\partial Z_j^h} \cdot \frac{\partial Z_j^h}{\partial w_{k_j^h}}, \tag{2.4}$$

where Z_j^h refers to the output of a node j (of the current layer h) before applying an activation function. Z_j^h is given by equation:

$$Z_j^h = \sum_{k=1}^{m} (w_{k_j^h} \cdot Y_k^{h-1} + b_j^h), \tag{2.5}$$

where m indicates the number of nodes in the previous layer and Y_k^{h-1} is the output of a node k (of the previous layer $h-1$) after applying an activation function.

According to the equation 2.5, $\frac{\partial Z_j^h}{\partial w_{k_j^h}}$ is computed as follows:

$$\frac{\partial Z_j^h}{\partial w_{k_j^h}} = Y_k^{h-1} \tag{2.6}$$

Using equations 2.4 and 2.6, $\frac{\partial E}{\partial w_{k_j^h}}$ can be calculated as follows:

$$(2.4) \Rightarrow \frac{\partial E}{\partial w_{k_j^h}} = \frac{\partial E}{\partial Z_j^h} \cdot Y_k^{h-1} \tag{2.7}$$

Using chain rule method, $\frac{\partial E}{\partial Z_j^h}$ can be expressed as follows:

$$\frac{\partial E}{\partial Z_j^h} = \frac{\partial E}{\partial Y_j^h} \cdot \frac{\partial Y_j^h}{\partial Z_j^h} = \frac{\partial E}{\partial Y_j^h} \cdot \hat{\varphi}(Z_j^h), \tag{2.8}$$

where $\hat{\varphi}$ is the backward activation function and Y_j^h is the output of a node j after applying an activation function.

Based on equations 2.7 and 2.8, $\frac{\partial E}{\partial w_{k_j^h}}$ can be expressed in the following way:

$$(2.7) \Rightarrow \frac{\partial E}{\partial w_{k_j^h}} = \frac{\partial E}{\partial Y_j^h} \cdot \hat{\varphi}(Z_j^h) \cdot Y_k^{h-1} \tag{2.9}$$

The gradient is computed similarly for a single bias b_j^h, except that $\frac{\partial Z_j^h}{\partial b_j^h} = 1$. This results in the following calculation of $\frac{\partial E}{\partial b_j^h}$:

$$\frac{\partial E}{\partial b_j^h} = \frac{\partial E}{\partial Z_j^h} \cdot \frac{\partial Z_j^h}{\partial b_j^h} = \frac{\partial E}{\partial Z_j^h} \cdot 1 \tag{2.10}$$

According to the equations 2.8 and 2.10:

$$(2.10) \Rightarrow \frac{\partial E}{\partial b_j^h} = \frac{\partial E}{\partial Y_j^h} \cdot \hat{\varphi}(Z_j^h) \cdot 1 \tag{2.11}$$

To continue backwards, the Backpropagation process to the previous layer, $\frac{\partial E}{\partial Y_j^{h-1}}$ is calculated as follows:

$$\frac{\partial E}{\partial Y_j^{h-1}} = \frac{\partial E}{\partial Z_j^h} \cdot \frac{\partial Z_j^h}{\partial Y_j^{h-1}} = \frac{\partial E}{\partial Z_j^h} \cdot w_{k_j^h} \tag{2.12}$$

Backpropagation starts by taking the derivative of the Error function, so the derivative will vary depending on which error function and activation function are used. By using sigmoid as the activation function and Binary Cross Entropy as the Error function, $\frac{\partial E}{\partial Y_j^h}$ can be calculated as follows:

$$\frac{\partial E}{\partial Y_j^h} = -\frac{Y_i}{\hat{Y}_j^h} + \frac{1 - Y_i}{1 - \hat{Y}_j^h} \tag{2.13}$$

When using softmax activation function and Multi-Class Cross Entropy, $\frac{\partial E}{\partial Z_j^h}$ is computed as follows:

$$\frac{\partial E}{\partial Z_j^h} = \hat{Y}_j^h - Y_i \tag{2.14}$$

Update of Parameters

As a result of backpropagation, weights and biases are updated as follows:

$$\hat{w_{k_j^h}} = w_{k_j^h} - \alpha \cdot \frac{\partial E}{\partial w_{k_j^h}}, \quad \hat{b_j^h} = b_j^h - \alpha \cdot \frac{\partial E}{\partial b_j^h}, \tag{2.15}$$

where α is the learning rate. As a general rule, the learning rate should be a tiny number to incorporate safer steps toward error minimization.

2.4.4 Performance

Every machine learning algorithm needs to be assessed to determine its performance. In classification tasks, the accuracy is a useful metric for this purpose. Accuracy refers to the proportion of examples for which the model produces the correct output. On the other hand, the error rate, which is the opposite of accuracy, is also an important metric. Error rate or loss counts the number of examples for which the model predicts the incorrect output to the total number of examples. To evaluate an algorithm's performance, it is

common to use unseen data, which is a subset of the input data that has no overlap with the subset used for training.

2.4.5 Activation functions

As we have discussed earlier, when we move from one layer to another in a neural network, every node passes the weighted sum of the inputs through a function. In the case of the Perceptron, this function is represented by a step function. However, to achieve learning in general situations, such as non-linear regions, more complex functions are used.

Sigmoid

The sigmoid function is so-called because of its S-shaped curve. This function is typically defined between 0 and 1, which makes it especially useful for models that output probabilities. The figure below illustrates this concept.

$$\phi(x) = \frac{1}{1 + \epsilon^{-z}} \tag{2.16}$$

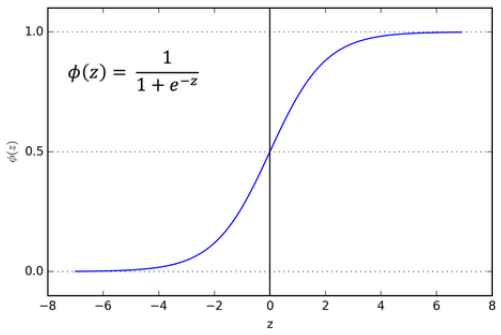

FIGURE 2.3: The sigmoid function. Source: [17]

ReLU

One of the most widely used activation functions in deep learning is called the rectified linear unit (ReLU). This function can be defined as the half-wave rectifier, which is represented as $f(z) = max(z, 0)$. While the sigmoid function used to be more popular in the past, ReLU generally performs better and faster in deep learning networks[[Deep learning,Yann LeCun1,2, Yoshua Bengio3 & Geoffrey Hinton4,5]].

$$R(z) = \begin{cases} z, & \text{for } z \geq 0 \\ 0, & \text{for } z \leq 0 \end{cases} \tag{2.17}$$

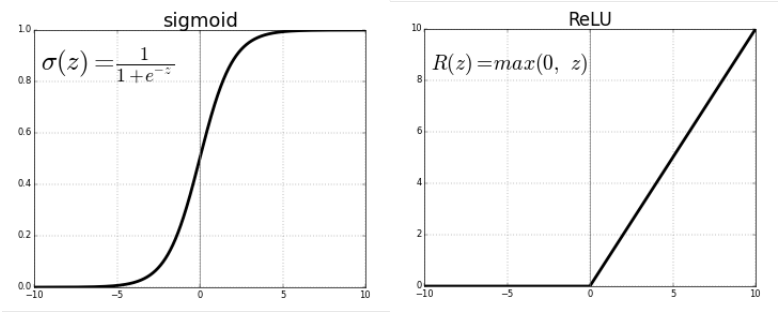

FIGURE 2.4: ReLU vs sigmoid function. Source: [17]

The ReLU function is a mathematical function that behaves linearly for positive inputs while turning negative inputs to zero. This linear behavior is beneficial when using a gradient-based method to optimize a machine learning model. ReLU is commonly used in neural networks, however, it has a limitation in that it zeros out all negative values, which can decrease the model's ability to learn correctly. This is because zeroing out a node forces it to not participate in the calculation of the final probabilities, which can negatively impact the overall performance of the model.

LeakyReLU

The Leaky ReLU addresses the weakness of the ReLU by introducing a small factor (usually 0.01) to multiply negative values, simulating a leakage effect.

$$R(z) = \begin{cases} z, & \text{for } z \geq 0 \\ \alpha z, & \text{for } z \leq 0, \text{ with } \alpha \leq 1 \end{cases} \tag{2.18}$$

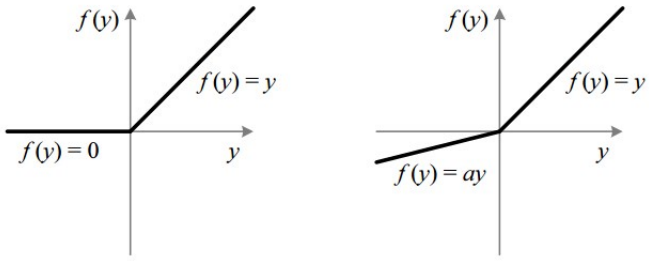

FIGURE 2.5: Leaky RELU vs Relu function. Source : [17]

Softmax

The softmax function is typically used for the final layer of a neural network, producing the output layer. This function is used specifically for multiclass classification problems and is a generalization of the sigmoid function, which is used in logistic regression. It should be noted that the softmax function can only be used in cases where the classes are mutually exclusive.

In mathematics, Softmax is defined as,

$$S(y)_i = \frac{\epsilon^{y_i}}{\sum_{j=1}^{n}(\epsilon^{y_j})} \tag{2.19}$$

where n is the number of the possible classes, and y is the input vector to the softmax function, which consists of n elements.

2.4.6 Overfitting

Overfitting occurs when a model fits precisely to the training data set, but it cannot perform accurately on new and unseen data. To avoid this issue, a test set is used to verify the model's generalization with respect to different and novel data. If a network is overfitted, it cannot make correct predictions or classify data accurately. A low error rate for the training dataset, followed by a high error rate for the test dataset, is an indicator of overfitting[18].

2.4.7 Underfitting

Underfitting is a phenomenon that occurs when a neural network is not trained for a sufficient duration or when the quality and quantity of the dataset are insufficient to create the correct weights and biases. It is the opposite of overfitting.

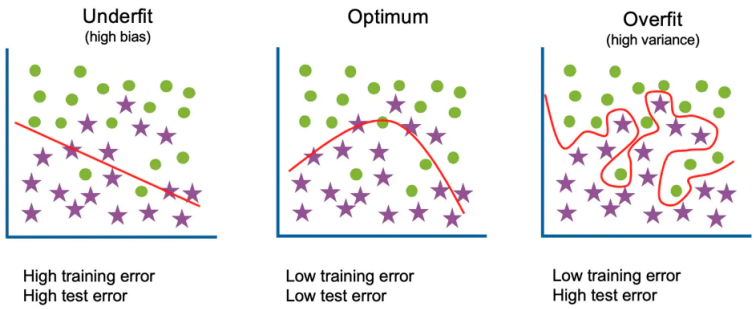

FIGURE 2.6: Overfitting. Source: [19]

2.4.8 EarlyStopping

As mentioned earlier, a neural network undergoes training until the desired error rate is achieved. However, this intense process may result in overfitting of the model, which means that noise can be incorporated into the training process. To prevent overfitting, a technique called early stopping is used, which stops the training of the model before it overfits. One way to avoid overfitting is to reduce the number of repetitions during the training process, i.e., the epochs. However, this can lead to underfitting. By using early stopping, a large number of training epochs is still possible, since the training will stop if the model's performance becomes steady. However, the risk of underfitting is still present, and the designer of the model should be aware of this potential issue.

Chapter 3

Related Work

Throughout this chapter, we will present some bioinspired models and techniques, as well as some FPGA implementations.

3.1 Towards New Generation, Biologically Plausible Deep Neural Network Learning

The model described in the work of Anirudh Apparaju & Ognjen Arandjelovic[20] describes a biologically plausible artificial neural network that has a fully connected network structure. The model includes visible neurons, hidden neurons, and output neurons. The hidden layers use a Hebbian synaptic plasticity rule as an unsupervised training rule, while stochastic gradient descent is used for training the fully connected perception. The network uses only forward-propagating signals, meaning that there is no backpropagation of signals past the final layer. The text also describes how synapses are modulated by activity of hidden units and how dynamic processes are described by differential equations. To speed up the learning process, minibatches and proxy ranking are used. The unsupervised algorithm of the model is designed to find useful representations of raw data without the need for particular task-specific knowledge. The high-level representations of convolutional neural networks and early visual processing areas of animal brains are similar to these representations. The paper proposes a biologically plausible alternative training methodology for artificial neural networks. Comparing the proposed methodology to conventional learning on two widely used public datasets, CIFAR and MNIST, it was found that biologically plausible learning is more robust to data scarcity and noise.

3.2 Architecture for a hybrid LIF SNN with dendrites and plasticity rules

Emmanouil Kousanakis, Apostolos Dollas, et al. [21] introduced an FPGA-based architecture that utilizes a hybrid LIF model with two learning rules (BCM and homeostatic plasticity) to analyze synapses and dendrites in detail. The model can represent generic neuron characteristics from different areas of the cerebral cortex. Additionally, the team mapped sparse interconnections into a well-defined memory structure, which can be used to stream data from an external storage device. By using external memory to feed the FPGA, the interconnection schemes can be initialized without requiring system synthesis, allowing for greater flexibility in the initialization process.

3.3 Online Spatio-Temporal Learning in Deep Neural Networks

SNNs are neural networks that are based on insights from neuroscience, which consist of interconnected neurons through synapses. These neurons maintain a temporal trace of past neuronal events, and learn signals transporting information spatially from the environment or other brain regions to individual neurons. Although several neuron models have been developed, the training has often been done with spike-timing-dependent plasticity (STDP) Hebbian rules.

On the other hand, machine learning applications have primarily focused on incorporating the layered, highly interconnected topology of biological neural networks into artificial neural networks (ANNs), including recurrent neural networks (RNNs). However, the ubiquitous error backpropagation through time (BPTT) algorithm has severe limitations in scenarios involving online learning. In the work of Bohnstingl, Wozniak, et al, [22], an online learning methodology is proposed, called Online Spatio-Temporal Learning (OSTL), for deep recurrent networks of spiking neurons. The OSTL separates gradients into spatial and temporal components, allowing for simple integration into deep learning frameworks with comparable performance to BPTT. They derive OSTL for deep feed-forward SNNs, shallow feed-forward SNNs, deep RNNs, and generalize it to generic RNNs. This framework enables efficient online training for temporal data and opens up possibilities for trainable recurrent networks in low-power IoT and edge AI devices.

3.4 Single-Pass Covariance Matrix Calculation on a Hybrid FPGA/CPU Platform

In a study conducted by the Systems group at ETH Zurich under the support of Nevis Laboratories at Columbia University [23], a new method has been proposed for computing the covariance matrix using a hybrid FPGA/CPU model. In particle physics, the covariance matrix is widely used to reduce dimensionality and filter data. However, due to the curse of dimensionality, which is a phenomenon that occurs exponentially as dimensionality increases, covariance matrices are computationally complex and memory-intensive. To overcome this, estimators such as the maximum likelihood estimator can be used to approximate the covariance matrix to sufficient accuracy, but they can be vulnerable in the presence of non-normally distributed random variables. A new design has been presented that uses a unique decomposition of the covariance matrix, which requires only one pass of data to calculate the covariance matrix. This design has been implemented on a hybrid FPGA/CPU system and provides a speed-up of up to five orders of magnitude over previous FPGA implementations.

3.5 Thesis Approach

The aim of this project is to enhance the bio-inspired features of an ANN by incorporating dendritic structure, receptive field and synaptic plasticity. An FPGA architecture will be designed to train this network. Dendritic structures are used to divide an artificial neuron into its soma and dendrites. This provides sparsity to ANNs, which means that fewer parameters are required for training them, thereby reducing power consumption. The receptive field is inspired by the human visual system and offers structured connectivity, indicating that each neuron is associated with a neighborhood of inputs. Synaptic plasticity occurs when synapses in a network modify the strength of their connections without explicit instruction from previous or subsequent layers. These bio-inspired features have been shown to be useful in a wide range of tasks, such as continuous learning and noise reduction.

Instead of classical gradient descent, the Adam optimization algorithm is used for updating training parameters. This algorithm computes individual adaptive learning rates for each parameter, which is characterized as more bio-inspired, in accordance with dendrites in biological neurons that

are responsible for the generation of their own regenerative events (dendritic spikes).

This thesis aims to design and implement an FPGA-based architecture for this bio-inspired ANN that will accelerate training and reduce power/energy consumption further. The high parallelism and power efficiency of FPGAs will be utilized for this purpose. This bio-inspired model has the potential to be applied to systems with limited resources, such as portable devices and mobile phones.

Chapter 4

System Modeling

Throughout this chapter, we will provide a detailed description of the design and implementation of this module in order to better understand and analyze it. For the purpose of studying how the covariance learning rule acts on dendritic-structured artificial neural networks, the Poirazi lab at IMBB-FORTH introduced and developed this model in Keras. However, due to the considerable amount of time needed to train this network, an FPGA was considered a much more efficient platform. To implement it on an FPGA, the first step was to rewrite the model in Python using only the Numpy library, then convert it into a low-level language such as C and implement some functions in a manner compatible with the hardware.

The profiling of Python's and C's implementations will also enable us to the identification of the processes that consume a significant amount of memory and time. We can attempt to optimize them at the hardware level.

A key feature of this thesis is that it focuses on the training process for this bio-inspired ANN model.

4.1 Neuro-inspired ANN model with Covariance Rule

As mentioned earlier, the model was developed by the Poirazi lab at IMBB-FORTH, specifically by postdoctoral researcher S. Chavlis and research director P. Poirazi.

This study integrates two bioinspired features into a common neural network: the dendritic structures of the layers and the receptive field layer method. This chapter will explain both of them.

Based on this model two different approaches were developed regarding the learning rule.

The first strategy is the classic backpropagation rule of an ANN, which is presented in Lampros Pantzekos' thesis, Bioinspired DNN Architectures with Dendritic Structure, which provides a comprehensive analysis of the strategy and its implementation. The second approach, which is the concept of this thesis, is the combination of the Covariance Rule and the Backpropagation Rule. In this case, covariance is applied to the first layer of the network and backpropagation is applied to the remaining layers.

Furthermore, the Adam optimization algorithm, rather than classic gradient descent, is selected as the method for updating the network's parameters (Weights and biases). This update method computes individual adaptive learning rates for each parameter. As a result, it tends to be more bio-inspired since the information is encoded separately from the dendrites rather than entirely from the neurons themselves.

An important goal of this work is to increase the bio-inspiration in order to achieve significant resource savings.

It is important to note that some sections of the Lampros Pantzekos' thesis have been incorporated into our model to improve reading and understanding. Those are the 4.3.1 , 4.3.2, 4.3.3, 4.3.4

4.1.1 Bio-inspired Features

Dendritic-Structure

A dendrite is a thin extension of a neuron's cell body that receives signals from its neighbors. Dendrites are equipped with passive properties that attenuate incoming signals and active mechanisms capable of generating dendritic spikes, which enable nonlinear signal processing. Using activation functions, we can mathematically represent these dendritic features, essentially by treating dendrites as computational nodes.

In a typical artificial neural network (ANN), an artificial neuron is represented as a single node, with its output prediction computed through an all-to-all manner of summing weighted inputs from the preceding layer, followed by an activation function. This conventional structure requires millions of trainable parameters.

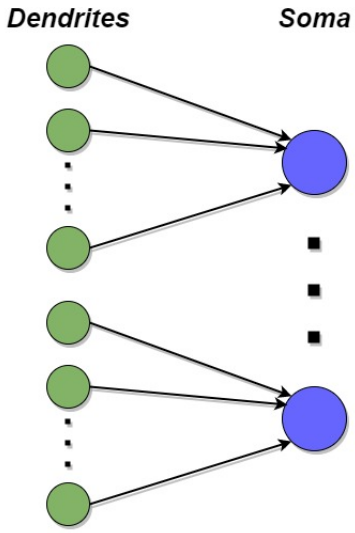

FIGURE 4.1: Dendritic-Structure Layer

In contrast, our model features a dendritic structure, where each artificial neuron is divided into a soma and its associated dendrites. This transforms each artificial neuron into a two-layer node structure, with the soma connected solely to its dendrites and each dendrite exclusively linked to its synapses. This unique design promotes sparsity in our model, resulting in a reduced need for trainable parameters compared to traditional fully connected ANNs, leading to significant resource savings.

Receptive Field (RF)

A receptive field is a region of the observed visual space whose luminance and structural patterns affect a neuron's activity.In the primary visual cortex, receptive fields correspond to visual angles and are organized by their position in the retina. In fact, different neurons tend to have different receptive fields and thus select and process specific directional information. The way we model this is by considering that each soma is related only to a specific neighborhood across the entire input, and each dendrite receives input from smaller neighborhoods surrounding each soma. When it comes to training, sampling of the input data is implemented in a way that areas expected to contain useful information are primarily selected. For example, in Convolutional Neural Networks (CNNs) random cropping of the input image is biased to central areas since they have greater chances to contain useful information.

Thus, the receptive field, or field of view, is a basic concept in deep CNNs for a certain layer in the network. Unlike in fully connected networks, where the value of each unit depends on the entire input to the network, a unit in neural networks can only depend on a region of the input. This region in the input is the receptive field for that unit. [24]

Implementation of the Connectivity Structure

In our model, this feature is reflected in how the inputs connect to the first layer of dendrites. To establish a receptive field, we employ binary masks, which are matrices sized to equal the product of the number of inputs with the number of dendrites. Each dendrite comprises 9 synapses, where each synapse forms a connection with a pixel from the image. The binary mask differentiates these connections by assigning 1s to indicate the presence and 0s to indicate the absence of a connection. For instance, in the MNIST dataset with 28*28 pixel images and 2048 dendrites in the 1st layer, the shape of the mask would be [784, 2048]. If we opt for 9 synapses per dendrite, the mask would have a total of 9 connections set to 1 for each column.

The model can support three methods for making a receptive field. These are the serial, semi-random, and random methods. As part of the serial method, the first pixel is chosen as the center of the neighborhood, which moves on to the next pixel each time. As part of the random method, the centers are selected randomly. A third method involves randomly selecting pixels that, 70% of the time, are located in the center of the image. For each soma, we found its 16 nearest neighbors, which are the dendrites. Then, for each of these dendrites, we create a neighborhood of nine neighbors, which represent synapses.

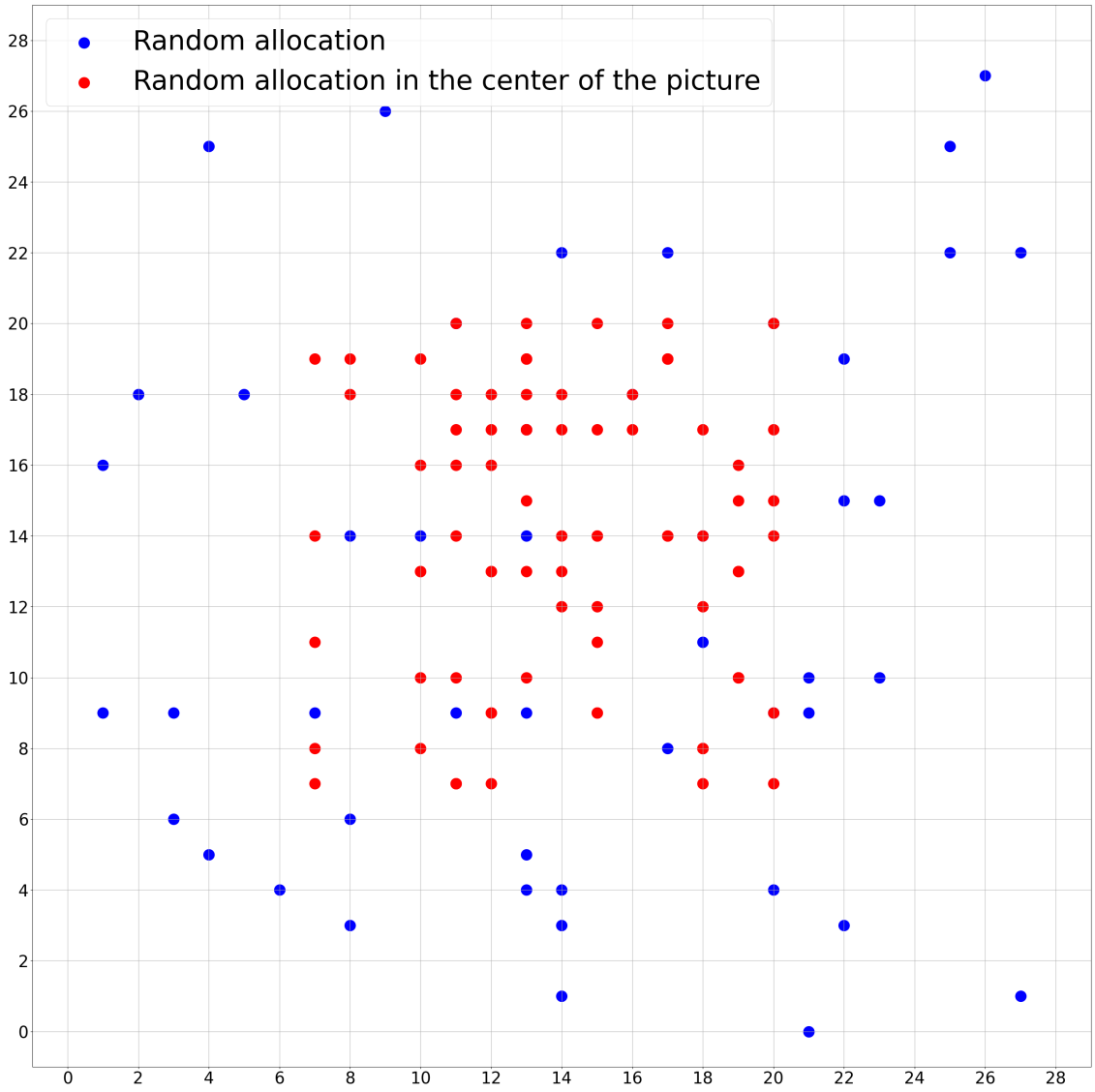

FIGURE 4.2: Semi-Random allocation of centroids

The described process is illustrated in Figure 4.2, where each soma is represented as a dot. Notably, 70% of the somas, displayed as red dots, are located in the [7, 21] range on both the x and y axes, corresponding to the center of the image. Meanwhile, the remaining 30% , blue dots, are selected randomly from locations across the entire image.

All these dots collectively form the second layer of the neural network. For each dot in this layer, a neighborhood of 16 points is created, representing the dendrites that consist of the first layer in our model. This relationship is illustrated in the figure below (Figure 4.3), where the blue dot signifies a selected soma from the prior image, and the yellow dots indicate the associated dendrites.

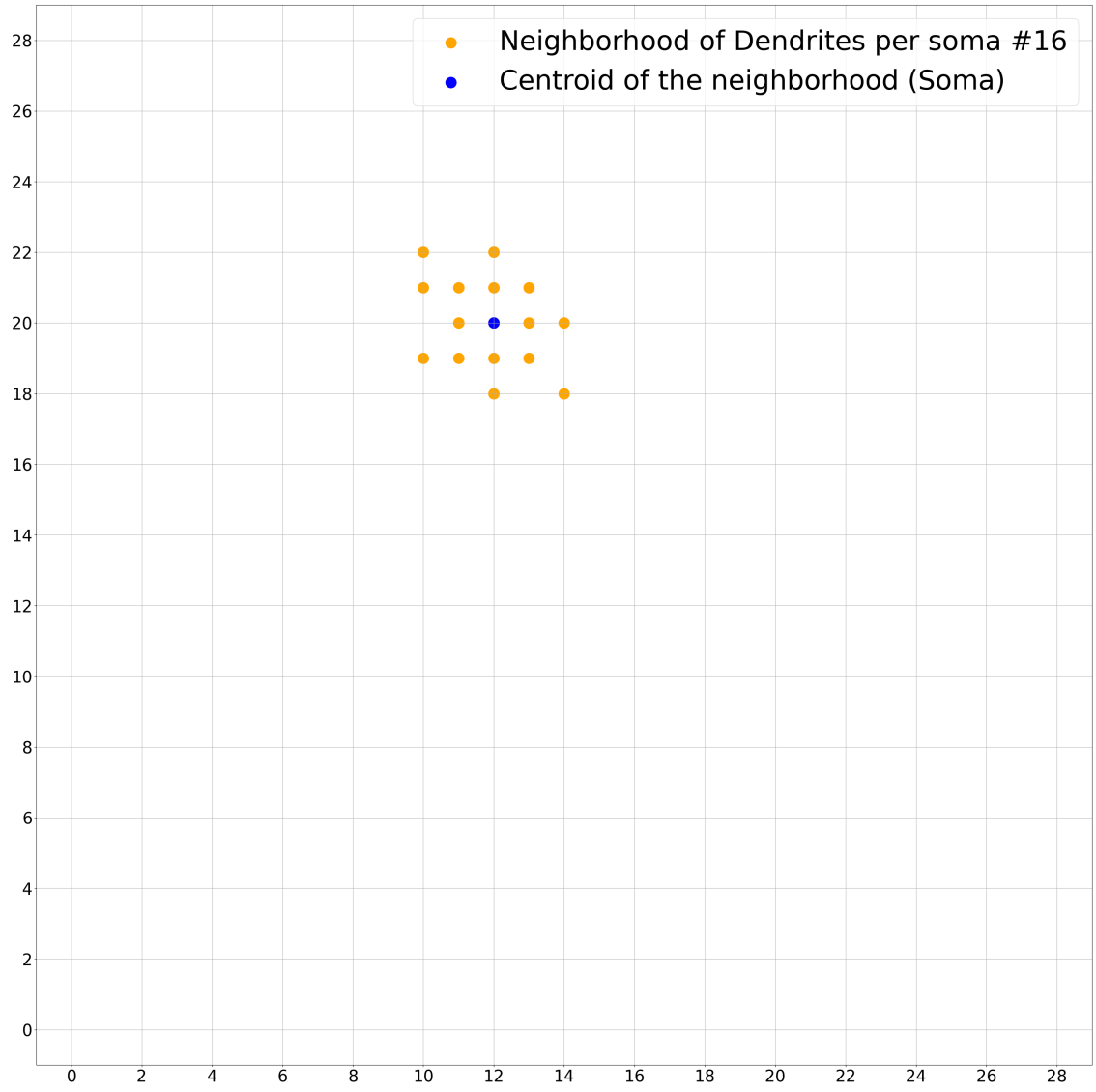

FIGURE 4.3: Random allocation of Soma with its dendrites

In the same way, each dendrite has a neighborhood of nine synapses, which are the connections between the pixels of the image and the dendrite. The figure 4.4 illustrates this.

FIGURE 4.4: Dendrite and its synapsis

The mask of the third layer indicates the connections from the first dendritic-structure layer to the second one, randomly linking each dendrite with 9 somas from the prior layer. Similarly, the mask for the fourth layer functions similarly to that for the second layer, except that each soma is connected to eight dendrites. Notably, a mask is unnecessary for the fifth layer.

These masks are set during the initialization process and remain constant thereafter while the connectivity-structure changes in the nervous system.

Hebbian Learning

In the book "The Organisation of Behaviour", Donald O. Hebb proposed a mechanism to update weights between neurons in a neural network. As a result of this weight-updating method, neurons were able to learn. The process is known as Hebbian Learning. The general principle is that any two cells that are repeatedly active at the same time will tend to become 'associated' so that activity in one facilitates activity in the other. Therefore, cells that fire together are wired together. This relationship is described by the following equation.

$$\Delta w_{ij} = \alpha \cdot x_i \cdot y_j, \tag{4.1}$$

where Δw_{ij} represents the change in the synaptic weight between neuron i and neuron j, α is the learning rate, x_i is the activity or output of the presynaptic neuron i, y_j is the activity or output of the postsynaptic neuron j.

Hebb's proposition is rooted in the observed biological phenomenon of synaptic plasticity. This refers to the ability of the connections (synapses) between neurons to change in strength over time in response to activity patterns. According to Hebbian learning, synaptic connections are strengthened based on correlated activity between pre- and postsynaptic neurons. This correlation-based learning is thought to play a fundamental role in various forms of learning and memory in the brain. Artificial neural networks can benefit from this learning rule in several ways. Firstly, it's biologically plausible. Moreover, Hebbian learning is unsupervised, meaning it can learn from unlabeled data without requiring explicit guidance, which simplifies the learning process and enables the network to recognize underlying patterns and structures within the data. In addition, Hebbian learning allows continuous integration of new knowledge over time, making it ideal for scenarios where the network must continuously learn from incoming data streams without forgetting previously acquired knowledge. It is important to note, however, that this rule has several key drawbacks, which call for further investigation and development. It is possible that the strengthening of connections between neurons that fire together can lead to indiscriminate learning. As a result, it may not always capture the precise features or patterns required for a particular task. For this reason, a more powerful approach was proposed by S. Chavlis which is mentioned as the Covariance Rule and it is explained in the following sections.

Covariance Rule

Traditionally, neural networks are able to learn through backpropagation, as discussed in Chapter 2. Essentially, this method is part of supervised learning, and it involves comparing the outcome with the expected outcome based on the input and then making corrections accordingly. In contrast, we incorporate non-supervised learning at the first level of the network in this study. This is referred to as the Covariance Rule.

In order to understand the Covariance Rule we have to understand 3 crucial concepts. Those are the variance of a variable, the covariance of multiple variables, and the covariance matrix.

4.1.2 Variance

Variance is a fundamental statistical measure that quantifies the degree of dispersion or spread of a set of data in a given statistical area. It provides insight into how individual values in a dataset deviate from the mean (average) of the dataset. Variance measures how far a set of values spreads out from the mean.

The symbol for variance is $\{\sigma^2\}$.

A lower variance indicates that the values are closer together, while a higher variance indicates that the values are spread out more. Variance can be used to measure the accuracy of predictions or the efficiency of a system.

The formula for calculating the variance of a set of n data points $\{x_1, x_2, \cdots, x_n\}$ is as follows:

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2, \tag{4.2}$$

where x_i is each data point and $\overline{x}$ is the mean of the data points

4.1.3 Covariance

Covariance is a statistic value that measures how two random variables are related. It is calculated as the covariance of random variables X and Y, providing insights into their linear relationship. It determines whether two variables move in the same direction or in the opposite direction.

The formula for calculating the covariance between two variables is,

$$\sigma(x,y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y}), \tag{4.3}$$

where n is again the number of observations, x_i and y_i is the data point, and $\overline{x}$, $\overline{y}$ are the means of the x and y.

Covariance can be difficult to interpret due to the wide range of possible results. Its values can range from negative infinity to positive infinity. The comparison of covariances between datasets with varying scales may result in misleading results. Therefore, it cannot be used to measure the strength of a relationship. Rather, it is used to determine the direction of a relationship. Consequently, we can have 2 possible outcomes for covariances either positive or negative covariance. Positive covariance occurs when 2 variables tend to move in the same direction. Higher values of one variable correspond to higher values of the other. For example, increased study hours often coincide with higher grades. In contrast, negative covariance indicates that the variables move in opposite directions. Higher values of one variable correspond to lower values of the other. For example, increased physical activity often coincides with decreased body weight.

4.1.4 Covariance Matrix

The covariance matrix is a symmetric matrix that summarizes the covariances between multiple variables. In a dataset with k variables, the covariance matrix provides a comprehensive overview of how each variable covaries with every other variable. It is essentially a square matrix where the diagonal elements represent the variances of individual variables. On the other hand, the non-diagonal elements represent covariances between variables.

The covariance matrix is symmetric, reflecting that the covariance between variables $x_i y_i$ and $x_j y_j$ is the same as the covariance between $y_j x_j$ and $y_i x_i$.

The covariance matrix is given by the $C_{i,j} = \sigma(x_i, y_i)$, where $C \in (\mathbb{R}^{d \times d})$ and d is the number of the variables.

where n is the number of observations, x_i refers to the variable of x and y_i to the variable y, and $\overline{x}\,\overline{y}$ are the mean values of the observations respectively.

Combining the two formulas above 4.2 and 4.3 , we can now calculate the Covariance matrix.

$$C = \begin{bmatrix} \sigma(x,x) & \sigma(x,y) \\ \sigma(y,x) & \sigma(y,y) \end{bmatrix} \tag{4.4}$$

Implementation of the Covariance Rule

The application of this rule in a layer of a model is happening by skipping the back-propagation method to update the weights and using the covariance matrix. In our case, we use a 3D covariance, which practically is stuck of covariance matrices as many as the number of the dendrites of the layer. The variables of the matrix are the weights of the synapses of each dendrite that the matrix refers to and the observations are the weights multiplied with the inputs throughout the batch of each epoch. Then, for each synapse, we find the summation of the covariance matrix values. In the end, the updates of the weights are the sum of those summations multiplied by the sigmoid of the feed-forward.

4.1.5 Reference Model Architecture

Combining all the above we have concluded to the presented neural network

The model consists of 2 dendritic-structure layers. However, a dendritic-structure layer is equivalent to 2 layers of a typical ANN, because the neuron is divided into the soma and its dendrites, as previously mentioned.

- 1st dendritic-structure layer: 128 somas with 16 dendrites per soma (2048 dendrites in total) and 9 synapses per dendrite.
- 2st dendritic-structure layer: 16 somas with 8 dendrites per soma (128 dendrites in total) and 9 synapses per dendrite.

After each layer, the weights are passed through an activation function. Those are described in section 2.17 and you can apply them to a model. In our case, ReLU is preferred for all of our layers.

Nevertheless, it suffers from the "Dying ReLU problem," which can be solved by an improved version of it, called Leaky ReLU. As a result, Leaky ReLU will serve as the activation function in these layers. In addition, there is an output layer, at which the scores are converted into probabilities through the application of the softmax function (2.19). The softmax layer is commonly

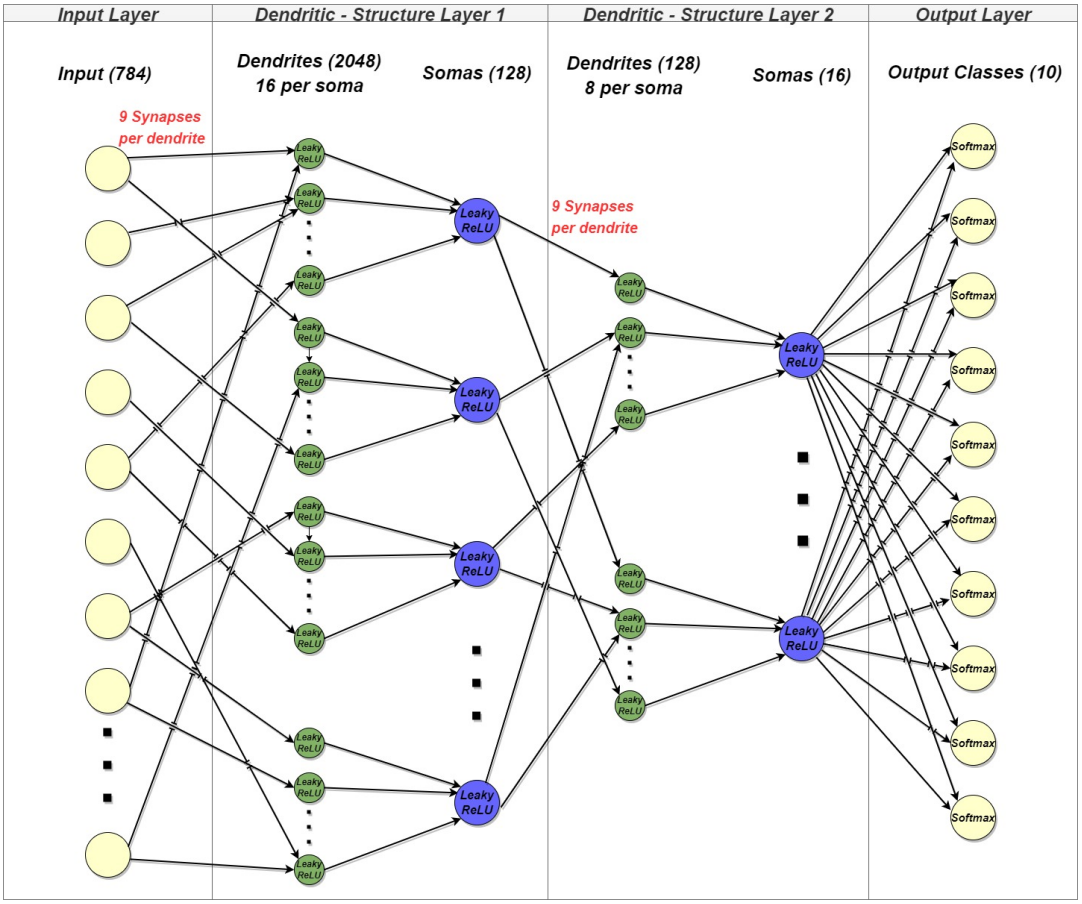

FIGURE 4.5: Model Architecture. Source: [25]

used as the final layer of an ANN model since it is appropriate for multi-class classification tasks. Therefore, the model contains 5 layers. As input, this model uses images from the MNIST dataset[26]. In MNIST, each input image has a size of 28×28 pixels, which is converted to a one-dimensional array of 784 pixels. Detailed information about each layer can be found in table 4.1, which includes the number of input and output nodes within parentheses. Moreover, the thesis model architecture is illustrated in figure 4.5.

TABLE 4.1: Detailed description of each layer in the model.

	Input nodes	Output nodes(Units)	Activation Function
1	Inputs (784)	Dendrites (2048)	Leaky ReLU
2	Dendrites (2048)	Somas (128)	Leaky ReLU
3	Somas (128)	Dendrites (128)	Leaky ReLU
4	Dendrites (128)	Somas (16)	Leaky ReLU
5	Somas (16)	Output Classes (10)	Softmax

4.2 Software Implementations - Tools used (Keras - Numpy)

The software implementations include the necessary initialization, training, and testing procedures. The training procedure differs between the first layer and consecutive ones. For the first layer, the training steps are Forward propagation and the Covariance Rule, while there is no need for Backpropagation and Update since the covariance is an unsupervised learning rule that updates the weights. For the rest of the layers, the steps are the common steps of a DNN (figure 4.6), which are the Full-Forward propagation (Feed-Forward), Backpropagation, and parameter updating (Adam Algorithm). The Full-Forward propagation and Covariance rule will be presented in detail in this thesis while the Backpropagation and the Update will be just mentioned with a basic explanation as is more detailed in Bioinspired DNN Architectures with Dendritic Structure[25] thesis.

The testing procedure consists only of Full-Forward propagation.

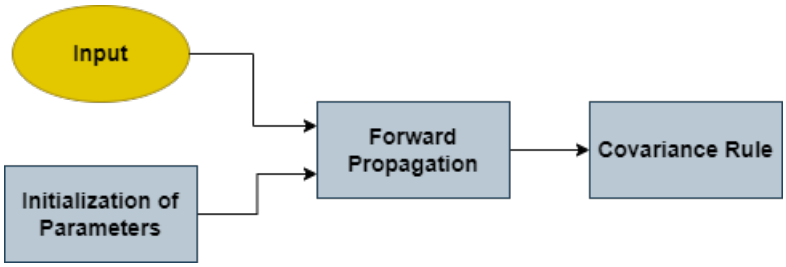

FIGURE 4.6: The basic steps of training procedure for the Hebbian Layer

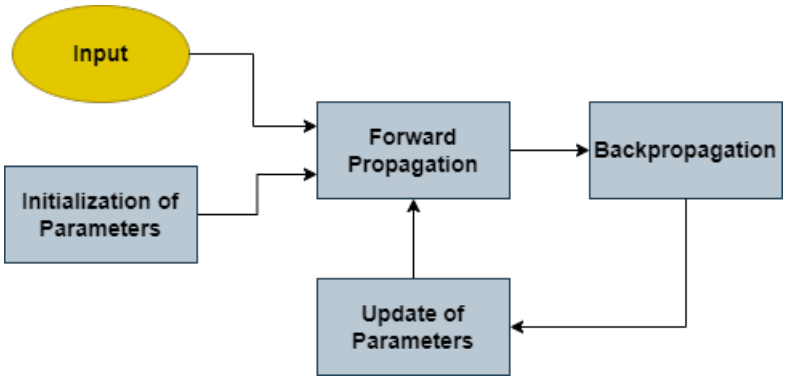

FIGURE 4.7: The basic steps of training procedure for the rest of the layers

The model was first implemented by S. Chavlis in Keras. Keras is a high-level neural network Application Programming Interface (API) written in Python. It runs on top of TensorFlow, an open-source machine learning platform that offers multiple abstraction levels for building and training models. Keras supports fast experimentation with DNNs by providing numerous implementations of commonly used NN building blocks. Based on Keras implementation, Numpy was used to implement the model at a lower level. Numpy is a Python library for handling large, multi-dimensional arrays and matrices rapidly and easily. In addition, it provides many computing tools (such as mathematical functions, random number generators, linear algebra routines, and more) that can be applied to these matrices and arrays.

Initially, a validity check was conducted on the given Keras code, since it was in the testing stage. The simplicity of Keras allowed us to conduct fast experiments with the bio-inspired model and gain a basic understanding of its features. There were, however, some ambiguities in the Keras' implementation due to its abstract nature. Therefore, a lower-level implementation was necessary. The Numpy implementation is able to provide deeper insight into the model's features and function.

4.2.1 Hyperparameter and Training Configuration

A hyperparameter is a parameter that controls the learning process in machine learning. Our bio-inspired ANN model is trained with the following settings that were used in Poirazi lab's Keras implementation as the most appropriate ones:

- Batch Size (16): The number of training samples that will be processed in one training iteration.
- Number of Epochs (30): The number of times a neural network is trained on a whole dataset.
- Number of Classes (10): Number of output nodes in the last layer
- Learning Rate (0.001): It controls how fast the neural network can learn. Using a small number as the learning rate allows us to minimize network error by taking safer steps.
- Validation Split (0.2): It is the percentage of total training data that will be used for validation.

- Leaky ReLU alpha (0.1): Leaky ReLU (2.18) allows a small slope for negative values (f(x) = alpha * x if x < 0, f(x) = x if x >= 0).
- Shuffle (True): Whether to shuffle the training data before each epoch.
- ReduceLROnPlateau Callback (applied - patience = 5 epochs): To reduce learning rate when a metric (validation error) has stopped improving.
- ReduceEta Callback (applied - patience = 1 epoch): To reduce the learning rate of the covariance rule when validation error has stopped improving.
- Early Stopping Callback (applied - patience = 10 epochs): To stop training when a monitored metric (validation error) has stopped improving.
- Adam Optimizer (enabled): Adam optimization algorithm is an iterative optimization algorithm used to minimize the loss function during the training of neural networks
- Error function (Multi-class Cross entropy 2.3): It measures the difference between two probability distributions for a given random variable/set of events.
- Accuracy: It measures the number of correct predictions made by our model in relation to the total number of predictions made.

Data-set

The MNIST database of handwritten digits is used for training and testing. It includes:

- Training set of 60000 examples. According to the validation split hyperparameter (20%), 48000 examples are used for training and 12000 for validation.
- Test set of 10000 examples.
- Images (Inputs) of 28×28 pixels. Each input image of (28,28) shape is converted to a one-dimensional input array of 784 input pixels.

Training is conducted in batches of 16 (Batch_size) images. Rather than passing one image of 784 pixels as an input per training iteration, we pass 16 images of 784 pixels. In addition, at the end of every epoch, a validation process is carried out using a separate set of images, in order to evaluate the progress

of the training. After validation is complete, we use some callbacks, such as ReduceLROnPlateau and Early Stopping, to prevent underfitting and overfitting. These specific callbacks are triggered when the validation error stops improving for a certain number of epochs, thereby reducing the learning rate or stopping the training earlier.

Data Type

The Keras implementation developed by the Poirazi lab uses float as the data type. Our Numpy implementation maintains this choice of data type since the goal of implementing the bio-inspired model in Numpy is to gain a deeper understanding of its features and training process rather than to achieve optimizations.

4.3 Numpy Implementation

Our approach is to use the high-level Keras implementation of Poirazi lab as a reference for developing a lower-level Numpy implementation in order to gain a more comprehensive understanding of its features and its training process rather than to optimize it. Also, we will use this as a base to implement the C program for the hardware design, which is explained in the next chapter. In the first part of the code, the MNIST dataset is downloaded and converted from ubyte files to numpy arrays. This conversion facilitates the processing of data. A float32 data type is used for these numpy arrays, and their values are normalized to [0,1]. As far as the data labels (targets) are concerned, they are converted from target vectors to categorical targets (binary class matrices). Following that, the masks are constructed as described previously (4.1.1). Model architecture (4.1) is defined as a list that contains information about each layer. In particular, this list (called architecture) specifies each layer's output nodes (units), activation function, mask, and initialization method for its parameters.

4.3.1 Generation of parameters (Initialization phase)

In the initialization process, weight matrices and bias vectors are created for each layer, depending on its input and output shapes. In the first four layers, the weights are generated using the 'He normal' initialization method, where samples are drawn from a truncated normal distribution centered at zero with standard deviation given by $stddev = \sqrt{\frac{2}{fan\_in}}$. For the final

layer, the 'Glorot uniform' initializer (or Xavier uniform initializer) is applied, which generates weights based on samples drawn from a uniform distribution within [-limit, limit], where $limit = \sqrt{\frac{6}{(fan\_in+fan\_out)}}$. Bias vectors are initialized with zeros at each layer. Selecting an initializer closely relates to selecting an activation function. "Glorot" (or "Xavier") initialization is appropriate for NN layers with sigmoid activations. This initializer is also suitable for the final layer of our model since softmax is a generalization of sigmoid. The "he" initialization is a modified version of "Glorot", which was specifically designed for layers using ReLU activation functions. So, it is used in the first four layers, where leaky ReLU is used (an improved version of ReLU). Afterward, each weight matrix is filtered (by element-wise multiplication) with its corresponding mask. The weights of non-desired connections will, therefore, be zero. During the initialization process, all masked weight matrices and bias vectors are stored in a list called parameters. Using numpy (4.5), each weight matrix is filtered (with its corresponding mask) and stored as follows:

$$parameters["W" + str(layer\_id)] = np.multiply(parameters["W" + str(layer\_id)], Mask) \tag{4.5}$$

4.3.2 Full-Forward propagation

As described in Section 2.4.2, Full Forward Propagation is used to predict the outcome in NN models. Each layer's output is calculated sequentially to determine the network's outcome. Specifically, it is calculated by multiplying its input matrix (Y_prev) by its weight matrix (W_curr) and adding its bias vector (b_curr). This result (Z_curr) is then passed through an activation function. The dimensions of these matrices are given in table 4.2. The above calculation is carried out as follows in Numpy:

$$Z\_curr = numpy.dot(Y\_prev, W\_curr) + b\_curr \tag{4.6}$$

$$Y\_curr = activation\_function(Z\_curr), \tag{4.7}$$

where numpy.dot performs matrix multiplication on matrices W_curr and Y_prev. Y_prev corresponds to the input of the network in the first layer.

Those equeations are executed for every layer of the model, and together, they consist of the Forward Propagation algorithm. The output of each layer

becomes the input of the next one. The only exception is the first layer, which takes its inputs from the batch of the dataset.

TABLE 4.2: Detailed description of each matrix dimensions in Full Forward propagation. In the first layer, Y_prev refers to the input of the network.

	Forward Matrices	Dimensions
1	Y_prev (Layer's Input)	[Batch_Size, Input_nodes]
2	W_curr	[Input_nodes, Output_nodes]
3	b_curr	[1, Output_nodes]
4	Z_curr	[Batch_Size, Output_nodes]
5	Y_curr	[Batch_Size, Output_nodes]

Algorithm 1 Full Forward propagation

procedure FULL_FORWARD($Input\_Value, init\_parameters, architecture$)
  $Y\_curr \leftarrow Input\_Value$
  for $layer\_id \leftarrow 1$ to 5 **do** ▷ for each single layer
    $Y\_prev \leftarrow Y\_curr$
    $activation \leftarrow architecture["activation" + str(layer\_id)]$
    $W\_curr \leftarrow parameters["W" + str(layer\_id)]$
    $b\_curr \leftarrow parameters["b" + str(layer\_id)]$
    $Z\_curr \leftarrow np.dot(Y\_prev, W\_curr) + b\_curr$
    if $activation == "leaky\_relu"$ **then**
      $Y\_curr \leftarrow leaky\_relu(Z\_curr)$
    else if $activation == "softmax"$ **then**
      $Y\_curr \leftarrow softmax(Z\_curr)$
    end if
    $forward\_outputs["Z" + str(layer\_id)] \leftarrow Z\_curr$
    $forward\_outputs["Y" + str(layer\_id)] \leftarrow Y\_curr$
  end for
  return $forward\_outputs$
end procedure

4.3.3 Backpropagation

The goal of backpropagation is to minimize the network's error by adjusting its weights and biases. Gradients in the error function relating to these

parameters determine the level of adjustment. The backpropagation process is described in detail in section 2.4.3. The first step of backpropagation is to calculate the derivative of the error function. In our model, softmax is used as the activation function of the last layer and multi-class Cross-Entropy is used as the error function. In this case, $\frac{\partial E}{\partial Z}$ can be directly calculated based on equation 2.14, skipping the calculation of $\frac{\partial E}{\partial Y}$. Using Numpy, it can be calculated as shown in equation 4.11. Using the calculated $\frac{\partial E}{\partial Z}$, the last layer's $\frac{\partial E}{\partial W}$ and $\frac{\partial E}{\partial b}$, as well as the $\frac{\partial E}{\partial Y\_prev}$, are calculated. The gradient for W, $\frac{\partial E}{\partial W}$, is calculated using the equation 2.7, which is based on the chain rule method. In Numpy, this equation is expressed as follows (4.8):

$$dW\_curr = numpy.dot(Y\_prev.T, dZ\_curr) / Batch\_Size\,, \tag{4.8}$$

where Y_prev refers to the output of the previous layer, or, in other words, the input of the current layer.

Similarly, the gradient for bias, $\frac{\partial E}{\partial b}$, is computed as shown in equation 2.10. Numpy implements this equation as follows (4.9):

$$db\_curr = numpy.sum(dZ\_curr, axis = 0, keepdims = True) / Batch\_Size\,, \tag{4.9}$$

During backpropagation, the error signal is propagated (backwards) from each layer to the previous one. This can be accomplished by calculating $\frac{\partial E}{\partial Y\_prev}$ according to equation 2.12. This equation is implemented as follows in Numpy (4.10):

$$dY\_prev = numpy.dot(dZ\_curr, W\_curr.T)\,, \tag{4.10}$$

The calculated $\frac{\partial E}{\partial Y\_prev}$ will be used to calculate the $\frac{\partial E}{\partial Z}$ of the previous layer. As for the remaining layers, $\frac{\partial E}{\partial W}$, $\frac{\partial E}{\partial b}$ and $\frac{\partial E}{\partial Y\_prev}$ are calculated using the same methods as in the last layer. However, $\frac{\partial E}{\partial Z}$ is determined differently. In particular, the chain rule technique is used to calculate it based on equation 2.8. Since Leaky ReLU serves as the activation function for the rest of the layers, its backward version is used here. In Numpy, $\frac{\partial E}{\partial Z}$ is calculated in the following manner (4.11):

$$dZ\_curr = \begin{cases} Y\_curr - Input\_Label & \text{for last (softmax) layer} \\ backward\_Leaky\_ReLU(dY\_curr, Z\_curr) & \text{for the other layers} \end{cases} \tag{4.11}$$

TABLE 4.3: Detailed description of each matrix dimensions in Backpropagation.

	Backpropagation Matrices	Dimensions
1	dZ_curr	[Batch_Size, Output_nodes]
2	dY_curr	[Batch_Size, Output_nodes]
3	dW_curr	[Input_nodes, Output_nodes]
4	db_curr	[1, Output_nodes]
5	dY_prev	[Batch_Size, Input_nodes]

The dimensions of the necessary matrices are given in table 4.3. Each layer of the Full Backpropagation algorithm (28) begins by loading the weight matrix of the current layer (W_curr) from the parameters list. Furthermore, the 'activated' output of the previous layer (Y_prev), as well as the 'non-activated' output of the current layer (Z_curr), are loaded from the forward_outputs list. In the next step of the algorithm, dZ_curr ($\frac{\partial E}{\partial Z}$), dW_curr ($\frac{\partial E}{\partial W}$), db_curr ($\frac{\partial E}{\partial b}$) and dY_prev ($\frac{\partial E}{\partial Y\_prev}$) are calculated for the last (softmax) layer. The terms mentioned above are then calculated for each remaining layer. Separating the last layer from the others is due to the different calculation method for dZ_curr (shown in equation 4.11). After each layer's calculations, the gradients for weights (dW_curr) and bias (db_curr) are stored in the gradients list. To maintain the sparse connection structure, dW_curr is filtered with the corresponding mask (from the architecture list) before being stored. , we present an overview of the backpropagation process for each layer of our bio-inspired NN.

Algorithm 2 Single layer Backpropagation

procedure SINGLE_LAYER_BACKPROPAGATION($dY\_curr, W\_curr, Z\_curr, Y\_prev$)
  if $Z\_curr \leq 0$ **then** ▷ backward Leaky ReLU
    $dZ\_curr = dY\_curr * 0.1$
  else
    $dZ\_curr = Z\_curr$
  end if
  $dW\_curr = np.dot(Y\_prev.T, dZ\_curr) / Batch\_size$
  $db\_curr = np.sum(dZ\_curr, axis=0, keepdims=True) / Batch\_size$
  $dY\_prev = np.dot(dZ\_curr, W\_curr.T)$
  return $dY\_prev, dW\_curr, db\_curr$
end procedure

Algorithm 3 Full Backpropagation

procedure FULL_BACKPROPAGATION($Input\_Value, Input\_Label, forward\_outputs, init\_parameters, architecture$)
  for $layer\_id \leftarrow 5$ to 1 **do** ▷ starting from the last layer
    $Z\_curr \leftarrow forward\_outputs["Z" + str(layer\_id)]$
    $Y\_prev \leftarrow forward\_outputs["Y" + str(layer\_id - 1)]$
    $W\_curr \leftarrow parameters["W" + str(layer\_id)]$
    if $layer\_id == 5$ **then** ▷ for the last layer
      $Y\_5 \leftarrow forward\_outputs["Y" + str(layer\_id)]$
      $dZ\_curr \leftarrow Y\_curr - Input\_Label$
      $dW\_curr \leftarrow np.dot(Y\_prev.T, dZ\_curr)/Batch\_size$
      $db\_curr \leftarrow np.sum(dZ\_curr, axis{=}0, keepdims{=}True)/Batch\_size$
      $dY\_prev \leftarrow np.dot(dZ\_curr, W\_curr.T)$
    else ▷ for the layers 4 to 1
      $dY\_curr \leftarrow dY\_prev$
      if $Z\_curr \leq 0$ **then** ▷ backward Leaky ReLU
        $dZ\_curr \leftarrow dY\_curr * 0.1$
      else
        $dZ\_curr \leftarrow Z\_curr$
      end if
      $dW\_curr \leftarrow np.dot(Y\_prev.T, dZ\_curr)/Batch\_size$
      $db\_curr \leftarrow np.sum(dZ\_curr, axis{=}0, keepdims{=}True)/Batch\_size$
      $dY\_prev \leftarrow np.dot(dZ\_curr, W\_curr.T)$
    end if
    $mask \leftarrow architecture["mask" + str(layer\_id)]$ ▷ masking dw
    $dW\_curr \leftarrow np.multiply(dW\_curr, mask)$
    $gradients["dW" + str(layer\_id)] \leftarrow dW\_curr$
    $gradients["db" + str(layer\_id)] \leftarrow db\_curr$
  end for
  return *gradients*
end procedure

4.3.4 Update method - Adam Algorithm

As discussed the classical Gradient Descent is inefficient when dealing with large datasets since it is applied to every single data point. As far as the Stochastic Gradient Descent is concerned, instead of using the entire dataset

for each iteration, only a random small batch is selected to calculate the gradient and update the parameters. In this way, it accelerates convergence in large datasets. However, this update method maintains a single and constant learning rate for all parameter updates throughout training. In biological neurons, dendrites support the generation of their own regenerative events (dendritic spikes). Considering this fact, the Adam optimization algorithm is chosen as the method for updating network parameters (weights and biases) iterative based on training data. In this method, individual adaptive learning rates are computed for each parameter from estimates of the first and second moments of the gradients. Rather than encoding information entirely from the neuron, it is encoded separately from its dendrites. As a result, the Adam algorithm tends to be more bio-inspired than the methods noted above.

Initially, the Numpy implementation of Adam (37) specifies the exponential decay rates (β_1, β_2) and the ϵ (to prevent division by zero) based on tested machine learning problems. For each layer of the network, the gradients for weights (dW) and biases (db) are retrieved from the 'gradients' list. The back-propagation algorithm produces this list. The algorithm updates the first moment estimates (the mean) of the gradient of the weights (mean_dw) and biases (mean_db). The second raw moment estimates (the uncentered variance) of the gradient of the weights (uvar_dw) and biases (uvar_db) are then updated. In order to keep track of the previous training iteration's values for mean_dw, mean_db, uvar_dw, and uvar_db, a list called 'adam_values' is used. Due to the fact that these moment estimates are initialized as vectors of 0's, they are biased towards zeros. To address this issue, the algorithm computes bias-corrected moment estimates. These are calculated based on the current training iteration, which is represented by the input variable t. Finally, the parameters are updated using these bias-corrected moment estimates. The dimensions of the necessary matrices are given in table 4.4.

Algorithm 4 Adam Optimization Algorithm - Update Algorithm

```
1: procedure ADAM_UPDATE(Input_Value, parameters, architecture, gradients,
   adam_values, learning_rate, t)
2:     beta_1 ← 0.9, beta_2 ← 0.999, epsilon ← 1e − 8
3:     for layer_id ← 1 to 5 do
4:         ▷ load dw and db of current layer
5:         dw ← gradients["dW" + str(layer_id)]
6:         db ← gradients["db" + str(layer_id)]
7:
8:         ▷ update mean of the gradient of weights
9:         mean_dw_prev ← adam_values["mean_dw"+str(layer_id)]
10:        adam_values["mean_dw"+str(layer_id)] ←
                                    beta_1*mean_dw_prev+(1-beta_1)*dw
11:        ▷ update mean of the gradient of biases
12:        mean_db_prev ← adam_values["mean_db"+str(layer_id)]
13:        adam_values["mean_db"+str(layer_id)] ←
                                    beta_1*mean_db_prev+(1-beta_1)*db
14:
15:        ▷ update uncentered variance of the gradient of weights
16:        uvar_dw_prev ← adam_values["uvar_dw"+str(layer_id)]
17:        uvar_dw_curr ← beta_2*uvar_dw_prev+(1 − beta_2)*(dw ∗ ∗2)
18:        adam_values["uvar_dw"+str(layer_id)] ←
                                    np.maximum(uvar_dw_prev, uvar_dw_curr)
19:        ▷ update uncentered variance of the gradient of biases
20:        uvar_db_prev ← adam_values["uvar_db"+str(layer_id)]
21:        uvar_db_curr ← beta_2*uvar_db_prev+(1 − beta_2)*(db ∗ ∗2)
22:        adam_values["uvar_db"+str(layer_id)] ←
                                    np.maximum(uvar_db_prev, uvar_db_curr)
23:
24:        ▷ compute bias-corrections
25:        mean_dw_c ← adam_values["mean_dw"+str(layer_id)]/
                                    (1-beta_1**t)
26:        mean_db_c ← adam_values["mean_db"+str(layer_id)]/
                                    (1-beta_1**t)
27:        uvar_dw_c ← adam_values["uvar_dw"+str(layer_id)]/
                                    (1-beta_2**t)
28:        uvar_db_c ← adam_values["uvar_db"+str(layer_id)]/
                                    (1-beta_2**t)
29:
30:        ▷ update weights and biases
31:        mask ← architecture["mask" + str(layer_id)]
32:        dw_f ← np.multiply(learning_rate*(mean_dw_c/(np.sqrt(uvar_dw_c)+epsilon)),
                   mask)
33:        db_f ← learning_rate*(mean_dw_c/(np.sqrt(uvar_dw_c)+epsilon))
34:        parameters["W"+str(layer_id)] ←
                                    parameters["W"+str(layer_id)]-dw_f
35:        parameters["b"+str(layer_id)] ←
                                    parameters["b"+str(layer_id)]-db_f
36:    end for
37:    return parameters                          ▷ return updated parameters
38: end procedure
```

TABLE 4.4: Detailed description of each matrix dimensions in Adam Optimization Algorithm.

	Adam Algorithm Matrices	Dimensions
1	dW (from backpropagation)	[Input_nodes, Output_nodes]
2	db (from backpropagation)	[1, Output_nodes]
3	mean_dw (mean_dw_c)	[Input_nodes, Output_nodes]
4	mean_db (mean_db_c)	[1, Output_nodes]
5	uvar_dw (uvar_dw_c)	[Input_nodes, Output_nodes]
6	uvar_db (uvar_db_c)	[1, Output_nodes]
7	dw_f	[Input_nodes, Output_nodes]
8	db_f	[1, Output_nodes]

4.3.5 Hebbian layer

As we mentioned earlier our model's first layer is a Hebbian Layer. This means that the Covariance Rule is applied in combination with the Feed Forward Propagation. The way it works is that we create 3D tensors for the weights, the inputs, and the masks, and we perform the necessary calculations on them without the need of looping on different arrays. Practically, this means that we have a batch, i.e., a stuck of images = 16 matrices with dimensions equal to units = 2048 and pixels = 784. Then we transpose those tensors into having the 3D dimension equal to [z,x,y] = [2048, 784, 16]. So we have stuck on 2048 independent matrices/datasets with the variables being the pixels and the observations being the batches. However, for the dimension related to pixels, we select only the 9 synapses that are active, and we reduce the dimensions of the tensor to [z,x,y] = [2048, 9, 16]. Then, for each Covariance matrix, we sum column-wise, i.e., for each variable, to obtain the delta_updates.

Then we average the outcomes of the Feed Forward algorithm from the first layer to get a mean value for each unit and we pass it through a sigmoid function. Thereafter we multiply the result with the deltas_updates and the eta factor. The end product of this process is the update of the weights.

This means that, although we have written much less code, we have abused the memory. However, in the chapter 5 we will see how we managed to minimize those tensors and alternate the logic in order to save memory and reduce computations.

Algorithm 5 Hebbian Layer

procedure HEBBIAN($Input\_Value, Mask, Weights, batch\_size, units, synapses, eta_hl$)
  $steepness \leftarrow 100$
  $threshold \leftarrow 0.01$
  $forget \leftarrow 1$

▷ Repeating the tensors into making them 3D and transpose them to the [units, pixel, batch_size]

  $Weights\_tiled \leftarrow tileAndTranspose(Weights, [units, pixel, batch_size])$
  $Inputs\_tiled \leftarrow tileAndTranspose(Input\_Value, [units, pixel, batch_size])$
  $Mask\_tiled \leftarrow tileAndTranspose(Mask, [units, pixel, batch_size])$

  $Weights\_masked \leftarrow np.multiply(Weights\_tiled, Mask\_tiled)$
  $Cov\_in \leftarrow np.multiply(Weights\_masked, Inputs\_tiled)$

▷ Reduce the size of the dimension of units into synapses_number because this locations only are non zero
  $Cov\_in \leftarrow gatherNonZero(Cov\_in, axis = 1)$

▷ Cov_out is a tensor of dimension [synapsis, synapsis, units]
  $Cov\_out \leftarrow CovarianceRule3D(Cov\_in)$

▷ Changes dimensions into [synapsis, units]
  $delta\_weights \leftarrow np.sum(Cov\_out, axis = 0)$

▷ Scatter deltas into a matrix of dimension[pixels, units]
  $delta\_weights \leftarrow scatter(delta\_weights)$
  $activities \leftarrow np.mean(Z\_curr, axis = 0, keepdims = True)$
  $binary\_like\_mask \leftarrow sigmoid(activities, steepness, threshold)$

  $updates \leftarrow np.multiply(delta\_weights, binary\_like\_mask)$
  $Weights \leftarrow Weights * forget + eta_hl * updates$
  return $Weights, updates$
end procedure

4.3.6 Callbacks

Callbacks were also implemented in numpy. The most important callbacks are the following: ReducePRL which is used to reduce the learning rate when the cost is not improving and the ReduceEta which is used to reduce the eta for the Hebbian layer when the cost has stopped being improved. In order to implement them we had to initialize some parameters at the beginning of the training process and to check the cost value of the model at the end of each epoch.

Also, the early stop callback, which stops the training process if the cost value has not been improved after a predefined number of epochs, is implemented. In particular, we have set this number to ten. In addition, the time callback, which shows how much time is needed for each epoch, was applied.

Finally, the checker callback was deployed, which stores the weights that give the minimum value of accuracy and checks recurrently if better accuracy is achieved to update the weights respectively.

4.3.7 Validation

The validation process in an Artificial Neural Network (ANN) is a critical component in gauging the model's effectiveness and generalization capability. The dataset is typically partitioned into three subsets: training, validation, and test sets. The training set is used to train the model, while the validation set is employed to assess performance during training and fine-tune hyperparameters. The test set remains untouched until the final evaluation to ensure an unbiased assessment.

As the model undergoes training, periodic evaluations occur on the validation set. Performance metrics like accuracy and loss are monitored, serving as indicators of how well the model generalizes to new, unseen data. If degradation in performance is observed on the validation set, adjustments to the model or training process may be necessary to counter overfitting.

Crucially, the validation set plays a pivotal role in hyperparameter tuning. Different configurations are tested, and the validation set's performance aids in identifying optimal hyperparameters, such as learning rates or regularization strengths, ensuring the model's robustness.

Once training and tuning are complete, the final model evaluation transpires on the test set. This set, unseen during training, provides an unbiased measure of the model's real-world performance. Throughout this process, the validation set acts as a guide, helping strike a balance between model complexity and generalization, ultimately yielding a well-performing neural network.

4.4 Profiling

The process of profiling involves analyzing the function calls, the execution duration of functions, the usage of particular instructions and the memory. These parameters are measured in real-time, i.e., while the program is running. Profiling assists engineers in identifying the routines that consume a disproportionately higher amount of time or memory and optimizing them. As shown in figure 4.8, most of the training time (approximately 96%) is consumed by the Hebbian layer while 4% is consumed by the Forward, Backward propagation, update of the weights using the Adam algorithm, and callbacks.

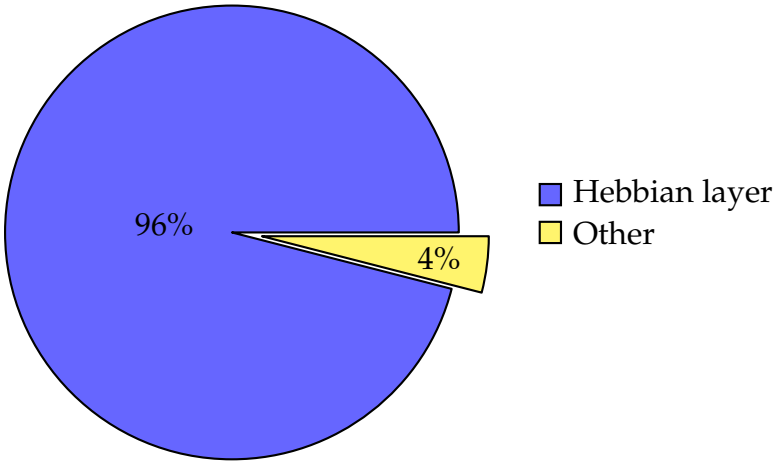

FIGURE 4.8: Analysis of how Numpy implementation consumes training time.

Considering that the Hebbian layer consumes almost the whole time of the epoch, profiling and analyzing this function is essential to keep us informed when high-time consumption is happening. Attention should be paid to the latter, in the sense that we try to optimize it in Hardware. Figure 4.9 illustrates the analysis of the time spent when we call each individual process of Hebbian.

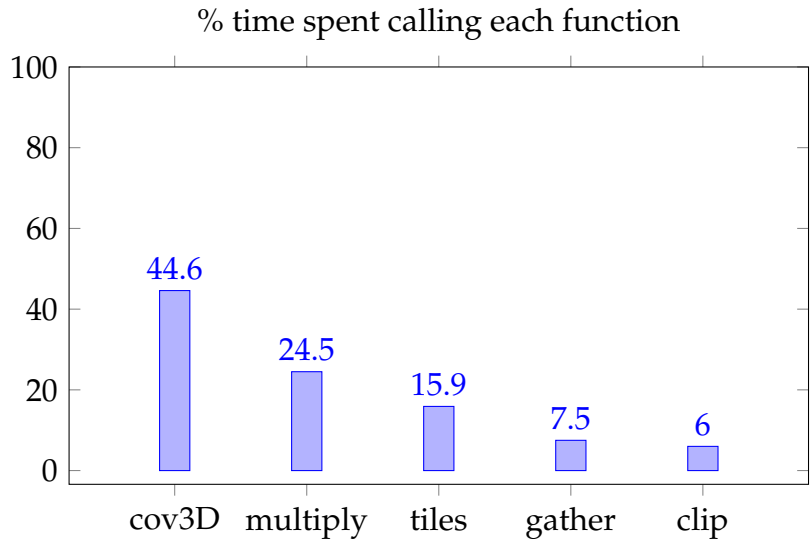

FIGURE 4.9: An analysis of the impact of the Hebbian's layer (inner) functions.

As we can see, 44%, i.e., almost half the time of the Hebbian layer, is consumed by the computations of the stuck with covariance matrices (cov3D). The other half-time goes to functions like np.multiply, which are element-wise multiplications of matrices. Those are the masks and the computation of the final weights. In addition, a significant amount of time is consumed in np.tiles, which makes a 3D tensor based on copies of a 2D matrix. As we have already mentioned, this process is memory-consuming, but here we can see that it is also very time-consuming. Whereas the rest of the functions that are used in the Hebbian layer, such as the gather and clip functions and others, affect less the execution time. In addition all of these functions are a part of a greater function which can be parallelized as it is explained in the next chapter5.

If we analyze further the Hebbian layer and go deep into the cov3D function, we will see that 80% of the function time goes to the np.cov function. So if we make the calculation

$$np.covTime * 3DCovarianceTime * HebbianTime * TrainingTime, \quad (4.12)$$

, where np.covTime is the time percentage consumed by np.cov inside the 3DCov function. The 3DCovarianceTime is the percentage of 3DCov inside the Hebbian layer. The HebbianTime is the time consumed by the Hebbian

layer during the total training time. Using the equation below:

$$0.8 * 44.6 * 0.96 * 1 = 0.34$$

we can see that only the np.cov function consumes 34% of the whole training process of our network.

4.4.1 Memory Profiling

Considering all the matrices required for one training iteration in our bio-inspired ANN implemented in Numpy, we calculated that approximately 36.6 MB of memory will be required. Approximately 98% of the memory required is determined by the size of the weight matrices. We mean by this that the sizes of the dw, mean_dw, and uvar_dw matrices are equal to the sizes of their respective weight matrices. Our ideal scenario will be to limit our data size (for each training iteration) to less than 4 MB, so that we can store all the data in BRAMs of FPGA. BRAMs are located in the PL (programmable logic) part of the FPGA and provide huge bandwidth. We will discuss in depth the BRAMs and our FPGA design in the following chapter. The bio-inspired ANN we developed is sparse due to its dendritic structure. It is important to emphasize that masks remain stable throughout the training process. As a result, once the weight matrices have been masked, they contain a large number of zero values that remain zero throughout the training process. However, we use them in their original dimensions in Numpy, which consumes a considerable amount of memory. By considering only the non-zero (masked) values of the weight matrices (after masking), which are actually used for the calculations, the weight matrices are drastically reduced in size. By using this method, approximately 1.07 MB of memory will be required, which is less than 4 MB. The next chapter will provide a detailed explanation of how this method can be applied to weight matrices.

4.5 Discussion

Amdahl's Law[27] is a formula used to determine the maximum theoretical speedup that can be obtained by improving a particular part of a system. This formula is expressed as follows:

$$S = \frac{1}{1 - P}, \tag{4.13}$$

where S represents the maximum theoretical speed-up and P is the fraction that represents the benefits from the improvement of the system resources.

As a result of this formula and the analysis of the main processes in figure 4.8, the maximum theoretical speed-up of the Hebbian algorithm process is calculated as follows:

$S = \frac{1}{1-(96\%)} = 25\times$. However, because we optimize other parts of our networks, such as forward propagation, backward propagation, and update, we can achieve a far greater speedup, which approximately is 3% of our network. For a deeper analysis one can address to the "Bioinspired DNN Architectures with Dendritic Structure thesis" [25]. $S = \frac{1}{1-(96\%+3\%)} = 100\times$.

Therefore, the optimization can result in a maximum theoretical speed-up of 100 times for our model's training process. In reality, this speed-up is unrealistic since it ignores the overhead associated with communication and other real-world factors. Nonetheless, this large theoretical speed-up serves to demonstrate that our model's training process can be parallelized to a great extent. By utilizing the high parallelism capabilities of FPGAs, an implementation of this model based on FPGA technology can significantly accelerate training.

Chapter 5

FPGA Implementation

The following chapter introduces the FPGA-based architecture of the training process of the bio-inspired ANN. The implementation was performed using the Vivado tools and subsequently executed to the Xilinx ZCU 102 evaluation board. Detailed explanations are provided regarding the design and execution of the training algorithm, as well as the methods employed to achieve parallelization at the computational level. Additionally, this chapter offers insights into the Vivado tools employed for design implementation, the ZCU 102 platform, the AXI4 Interface Protocol, communication methods between PL-PS, and memory configuration.

5.1 FPGA Platforms

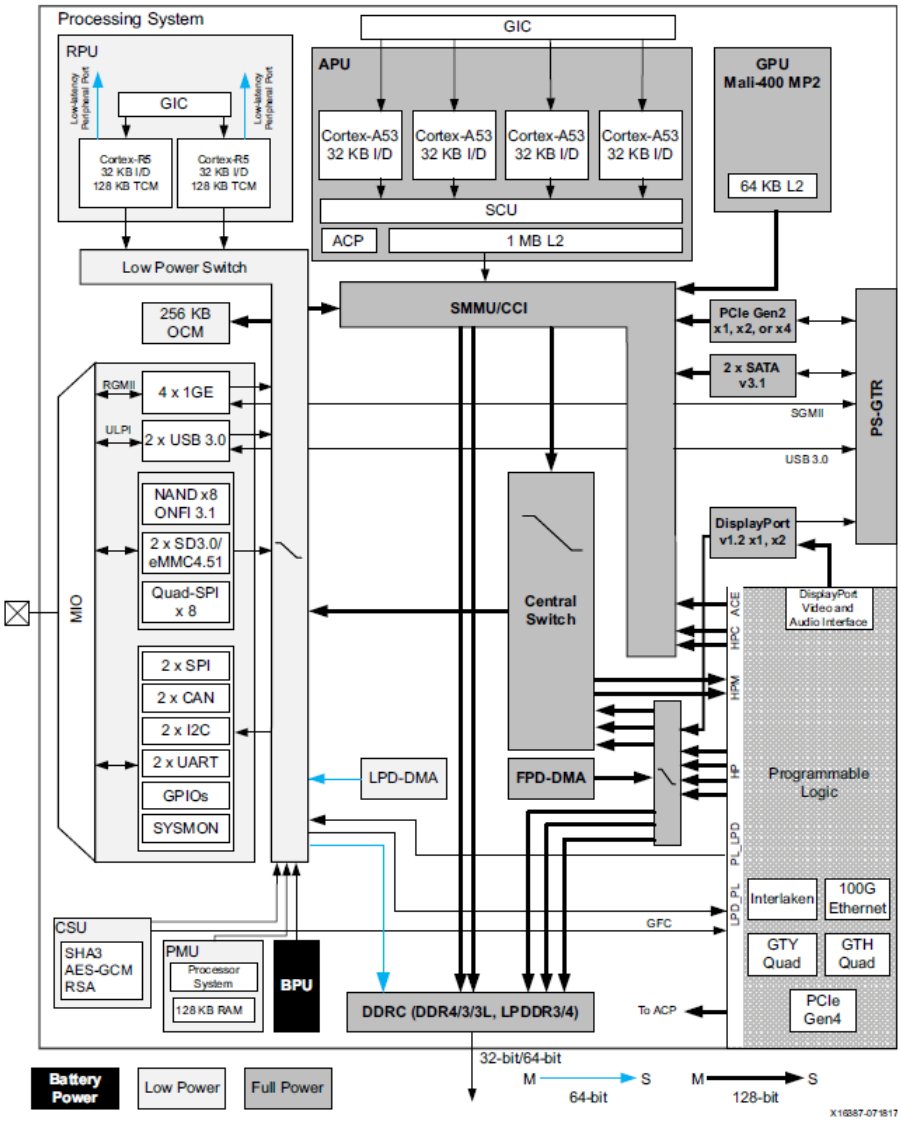

FIGURE 5.1: Zynq UltraScale+ MPSoC Top-Level Block DiagramZCU102 Evaluation Board

This thesis is focused on the utilization of the Xilinx ZCU102 evaluation board [28], widely known for its robust architecture tailored for rapid prototyping. At its core lies the Zynq® UltraScale+™ XCZU9EG-2FFVB1156E MPSoC, seamlessly integrating a potent processing system (PS) with user-programmable logic (PL) within a single device. This convergence empowers developers with a versatile platform featuring high-speed DDR4 SODIMM and component memory interfaces, FMC expansion ports, multi-gigabit per second serial transceivers, a rich array of peripheral interfaces, and FPGA

TABLE 5.1: ZCU102 Specifications.

Feature	Resource Count
Logic Cells	599550
Flip-Flops	548160
DSP Slices	2520
LUTs	274080
BRAMs	912
Block RAM	4MB
PS DDR	4GB
PL DDR	512MB

logic for customizable designs, facilitating agile prototyping endeavors. Noteworthy, elements of the PS include a quad-core 64-bit ARMv8-A Cortex-A53 (application processing unit - APU), a dual-core 32-bit ARM v7-R Cortex-R5 (real-time processing unit - RPU), and an ARM Mali-400 MP2 graphics processing unit (GPU), each contributing to the board's formidable processing capabilities.

5.2 Tools Used

The hardware implementation of this thesis was developed using the Xilinx design suite[29], which is compatible with the zcu102. The Design Suite was supported and fully licensed by the Technical University of Crete Microprocessors & Hardware Lab (MHL). This suite offers three key tools. The first one is the HLS tool, facilitating user to start the design with no prior experience in low-level hardware design. Implementation and improvement of the design is achieved by writing code in C/C++. The Vivado HLS converts the C/C++ into VHDL/Verilog script. Afterward, this script act as an input in the second tool of the Xilinx package, which is the Vivado IDE. This app is used for the block design of the components of the FPGAs and their connectivity. The third tool that we used is the Vivado SDK. It is the part where the configuration of the FPGA takes place and where the code for the functionality and the control of the system is written.

5.3 FPGA Design

Initially, this thesis was designed by combining the IP block of the Hebbian layer with the architecture of the previously mentioned thesis[25] Figure 5.2. This resulted in the design illustrated in Figure 5.3. However, when this was implemented, the design failed to pass through the Vivado SDK implementation due to failing endpoints. Upon investigating this error, we discovered that our design faced a failing issue with the communication between PS and PL. This meant that despite the placing of our design was successful the resources were not enough for the routing of the design.

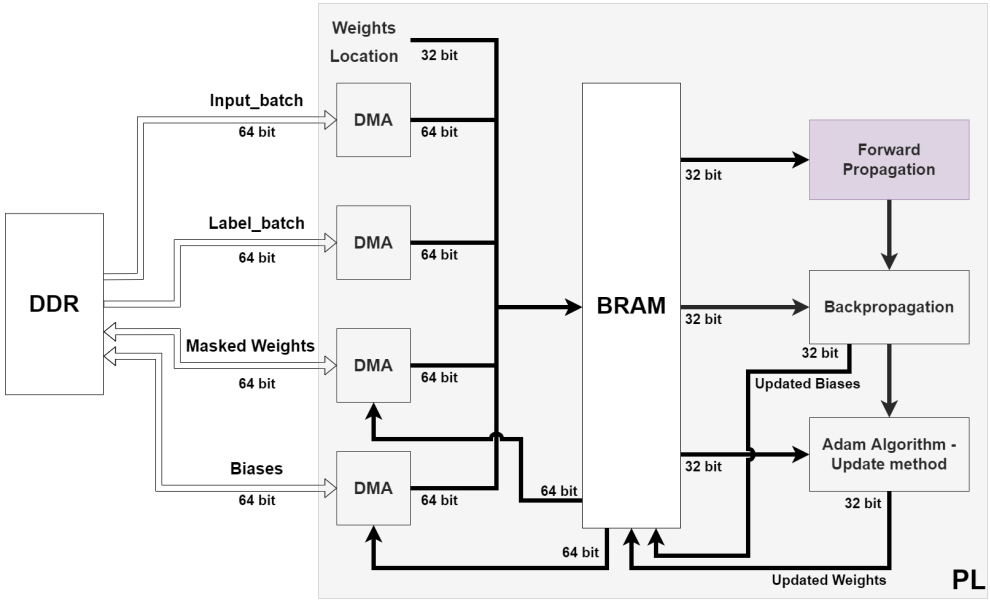

FIGURE 5.2: Inital FPGA Design - Architecture.

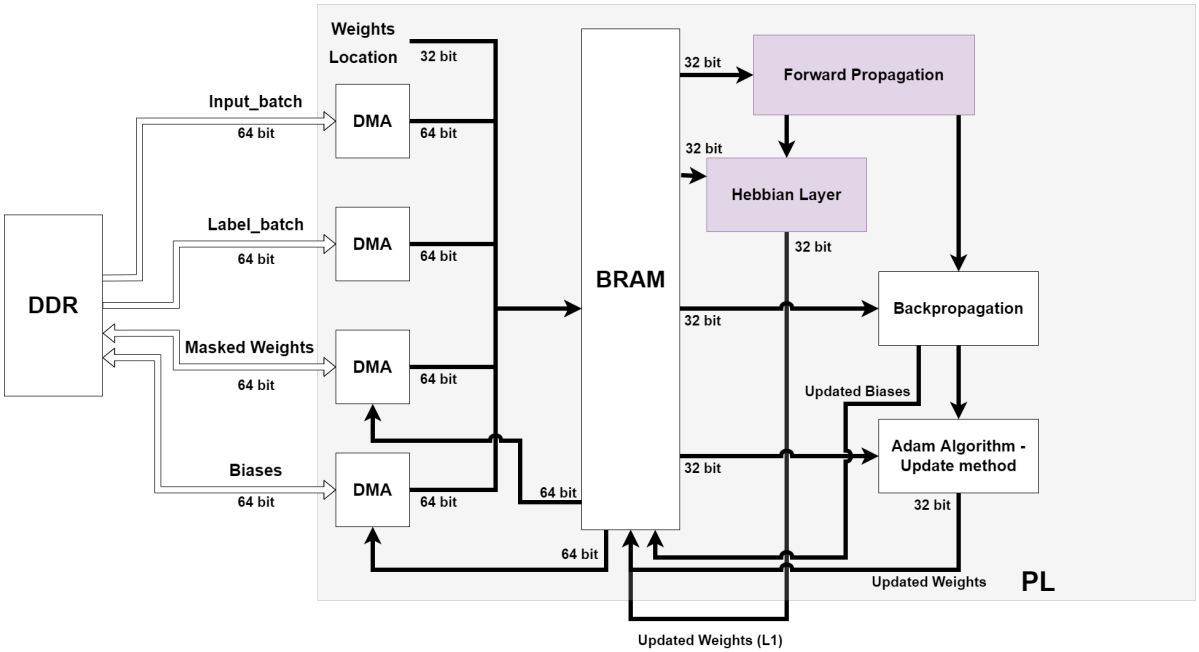

FIGURE 5.3: Inital FPGA Design - Architecture Combined with Hebbian layer.

To address this issue, the first step was to apply Vivados' strategies to prioritize routing rather than timing. Since this was not sufficient, the next step was to reduce the IP block's optimizations to free up some resources. Even so, our design continued to fail, so we decided to start designing based on Hebbian's requirements from the beginning.

As a result, we present our next and final approach to the design of the FPGA in figure 5.4 from a block design perspective. In the following figure the Forward Propagation block refers to the Forward Propagation algorithm 17, the Hebbian Layer to the Hebbian Algorithm using Covarince matrices 23, the Backpropagation block to the Backpropagatin algorithm 28 and the Adam-Update block to the update algoirthm 37.

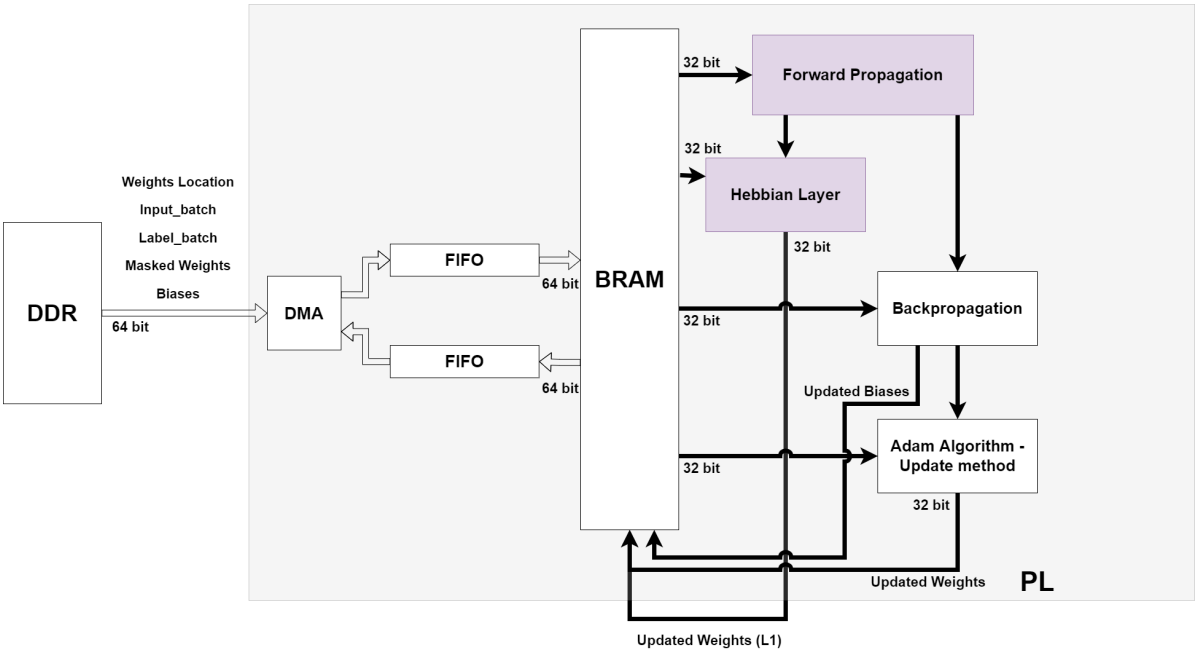

FIGURE 5.4: Final FPGA Design - Architecture combined with Hebbian layer.

5.3.1 Vivado High-Level Synthesis (HLS)

Vivado High-Level Synthesis (HLS) is a tool from Xilinx that allows designers to write C, C++, and SystemC code and convert it into RTL (register-transfer level) code that can be implemented on Xilinx FPGAs. It is a tool that allows designing, optimizing, and verifying designs at a high level of abstraction, which results in work at a higher level of productivity. Vivado HLS provides several features to help designers write high-level code that can be translated into efficient RTL.

One key feature of the HLS is that it supports the simulation of the source code and it can estimate the clock speeds and hardware resources that are needed for the design. Also, it can provide a visual analysis of each simulation and notify the designer of timing or data dependencies.

Another feature is the automatic hardware/software partitioning, which provides memory management and support for various design methodologies such as pipelining and dataflow. Additionally, Vivado HLS includes a wide range of optimization techniques to automatically improve the performance of designs, including resource sharing, loop unrolling, and pipelining. One of the key benefits of using Vivado HLS is that it allows working at a higher level of abstraction, which can remarkably reduce the time and effort required to design, verify, and implement FPGA-based systems.

Last but not least, Xilinx HLS can export the source code as an IP block that can be used further as a part of a larger and more abstract block design in the Vivado IDE.

Directives and Pragmas

Pragmas in High-Level Synthesis (HLS) provide additional information to the HLS tool that can influence the design implementation. The designer uses pragmas to specify how the tool should optimize and map the design into the target architecture. Pragmas also control the schedule of the operations, pipeline the loops, and handle the design's memory. They can help improve the FPGA's performance and resource utilization. It is important to understand the behavior and impact of the pragmas on the design.

A list of the most significant directives is provided below:

- HLS Interface: Is used for specifying the IP block's interface (inputs-outputs) and is used only for top-level functions.
- HLS Unroll: Is used to unroll a loop. Instead of executing consecutively the code included in a loop, multiple copies of the loop body are created in the RTL design. In this way, loops are forced to be executed together at the same time.

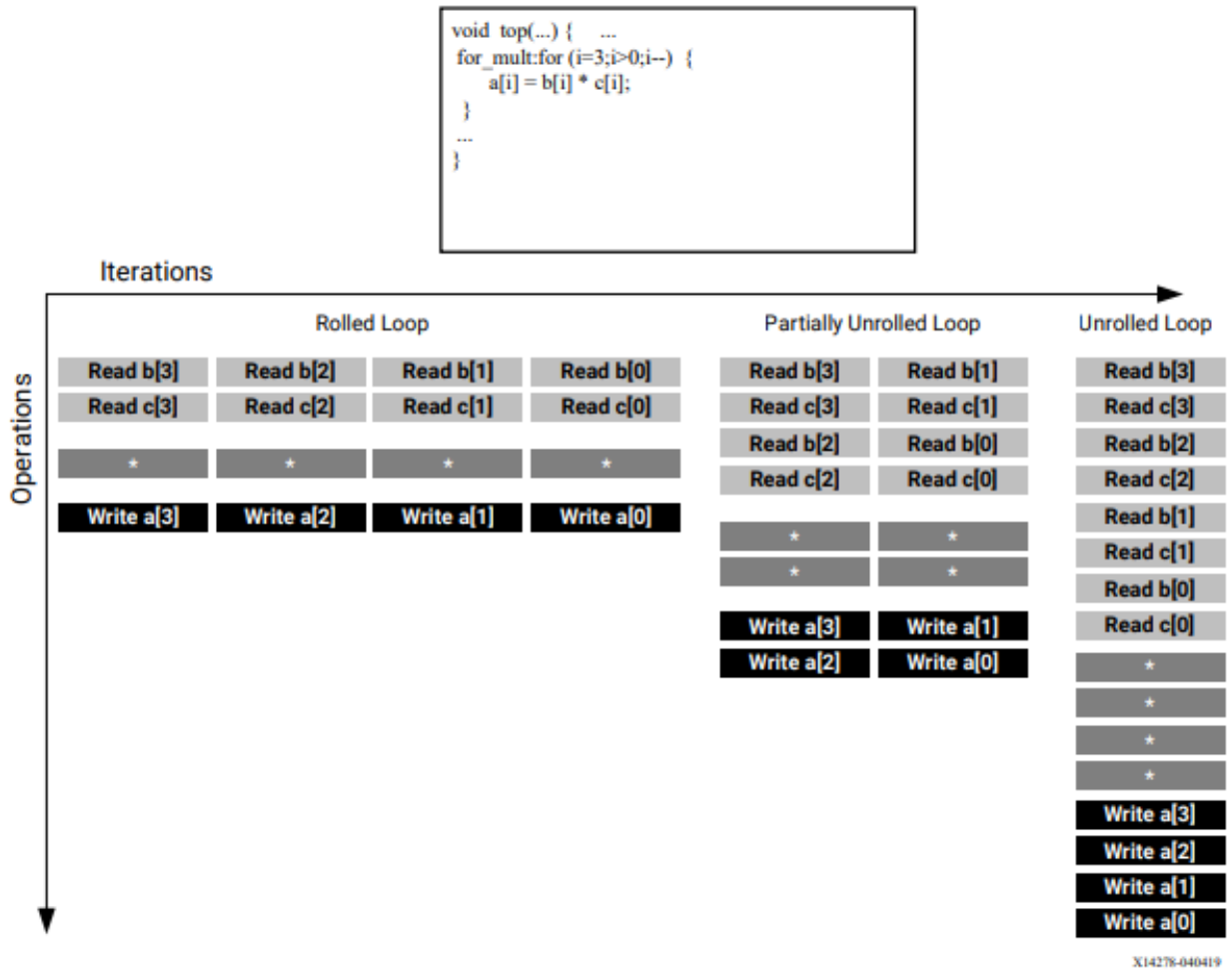

FIGURE 5.5: Unroll Pragma - Unroll for-loops to create multiple instances of the loop body. Source: [30]

- HLS Pipeline: Allows concurrent execution of operations to reduce the initiation interval (II) for a loop. A pipelined function or loop can process new inputs every <N> clock cycles, where <N> is the II of the loop or function. The maximum II one can achieve is 1 which means the function or loop processes a new input every clock cycle. You can specify the initiation interval through the use of the II option for the pragma. In nested loops, the outside loop is pipelined and all the inside loops are unrolled.
- HLS Array_Partition: Splits an array into smaller arrays or individual elements as it is depicted in the following graph.

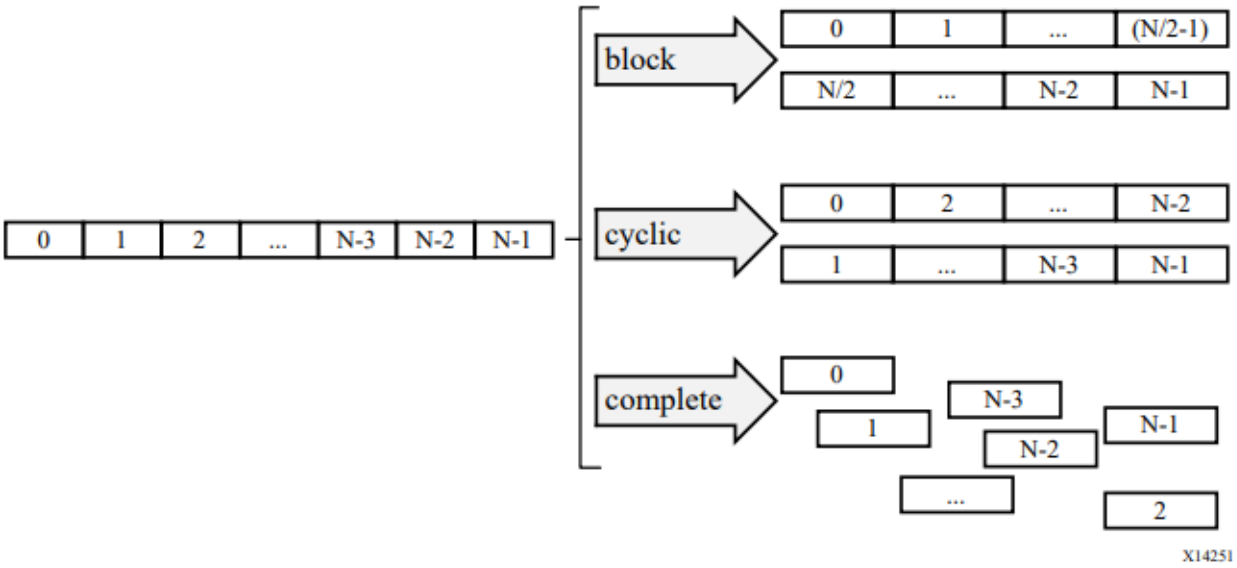

FIGURE 5.6: Array_Partitioning Pragma - Partitions large arrays into multiple smaller arrays to improve access to data and remove block RAM bottlenecks. Source: [30]

As a result, RTL is deployed with multiple small memories or multiple registers contrary to one large memory. This increases effectively the number of read and write ports on the storage. The design may be able to achieve higher throughput. However, it is necessary to have a greater number of memory instances or registers.

- HLS Inline: Function is detached from the hierarchy. After inlining, the function is dissolved into the calling function, and it no longer appears as a separate hierarchy level in the RTL. As a result, the resources can be assigned more efficiently to different functions. Additionally, when a function loses its hierarchy, all directives from the top-level function are applied to the inner function.
- HLS Dataflow: Enhances the concurrency of RTL implementations by allowing functions and loops to overlap in their operation, increasing the overall throughput of the design. DATAFLOW optimization effectiveness is based upon the flow of the data from one task to another within the design. DATAFLOW optimization cannot be performed using the HLS tool due to the following coding styles.

  Limitations of Control-Driven Task-Level Parallelism for additional details are the following:.

- Single-producer-consumer violations
- Feedback between tasks
- Conditional execution of tasks
- Loops with multiple exit conditions

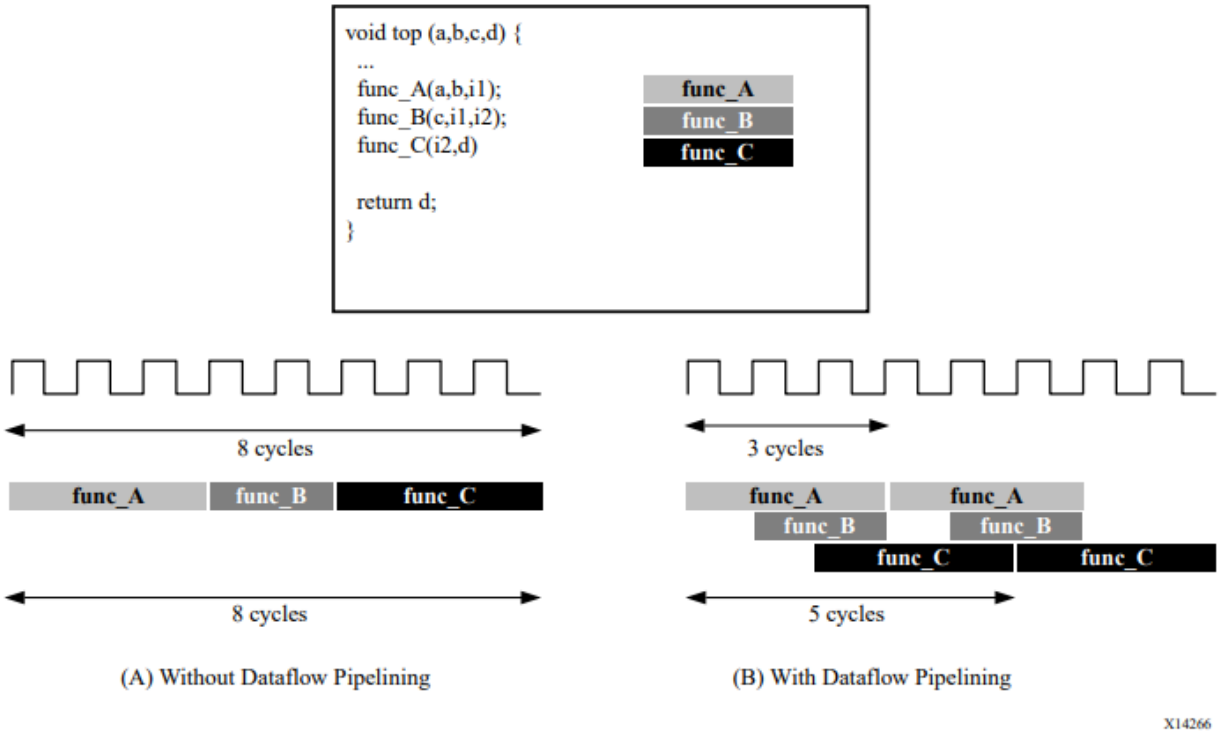

FIGURE 5.7: Dataflow Pragma - Enables task level pipelining, allowing functions and loops to execute concurrently. Used to optimize throughouput and/or latency. Source: [30]

HLS Implementation

As we discussed in the previous chapter, our algorithm was converted from Python high level libraries into a simpler and more understandable code. The algorithms must, however, be transformed into lower-level code in C language to be eligible for implementation and development on an FPGA. Optimizations should be considered during algorithm design. First, we noticed that our matrices, which were at this point 3-dimensional, contained multiple zeros. Thus, we realized that we had a collection of data that did not provide any information to our network. As a solution, we decided to keep two matrices with less data instead of one large matrix with sparse data.

FIGURE 5.8: Illustrates the final matrices we use for weights. From 1 big matrix to 2 smaller ones

Therefore, instead of a matrix with the shape of [pixels, units] which in our case is equal to [784,2048] and requires 6.42 MB of memory, we use a matrix with the location of the synapses and one with the weights of the synapses that correspond to those locations. Those matrices have the shape of [pixels, synapses] where the synapses are at least one order of magnitude smaller than the units. So, in this case, we have 2 matrices of shape = [2048,9], which both require 0.04 MB of memory.

It is important to point out that a short type could be used to store the data in the location matrix so that each slot only requires two bytes rather than four. However, for greater network compatibility, we decided to use the 4-byte option and store the location as an integer. Consequently, we only required 0.02 MB instead of 6.42 MB for the weights of the first layer.

Altering the covariance algorithm's logic was the next key concept that we decided to consider. In the previous chapter 4.3.5, we discussed the need for

two 3D matrices to parallelize computations. For example, to feed the covariance algorithm, element-wise multiplications of weights and inputs are facilitated by 3D matrices whose shape is [batch_size, pixels, units]. To produce a 3D matrix, as illustrated above, multiple copies of a 2D matrix are required. The following figure provides a better understanding of the concept, where we can see that the Input tensor is multiplied by the Weights tensor to create the 3D input for the covariance rule.

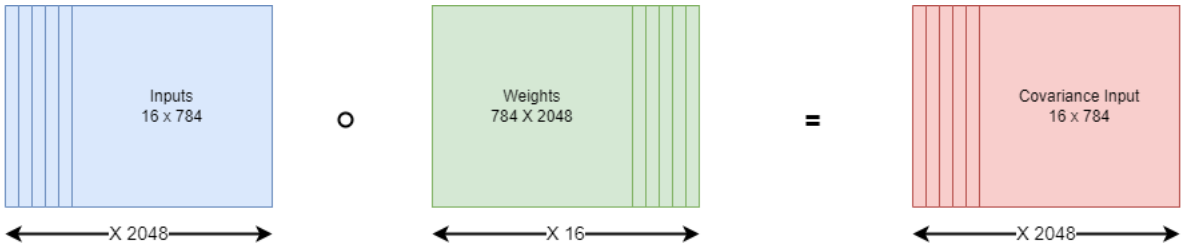

FIGURE 5.9: 3D Covariance Input

To avoid all this, we decided to keep the initial 2D matrices and change the algorithm to satisfy those dimensions. So, we decided that some parts should be implemented more sequentially. We then optimized and parallelized them in our FPGA based on Vivado tools.

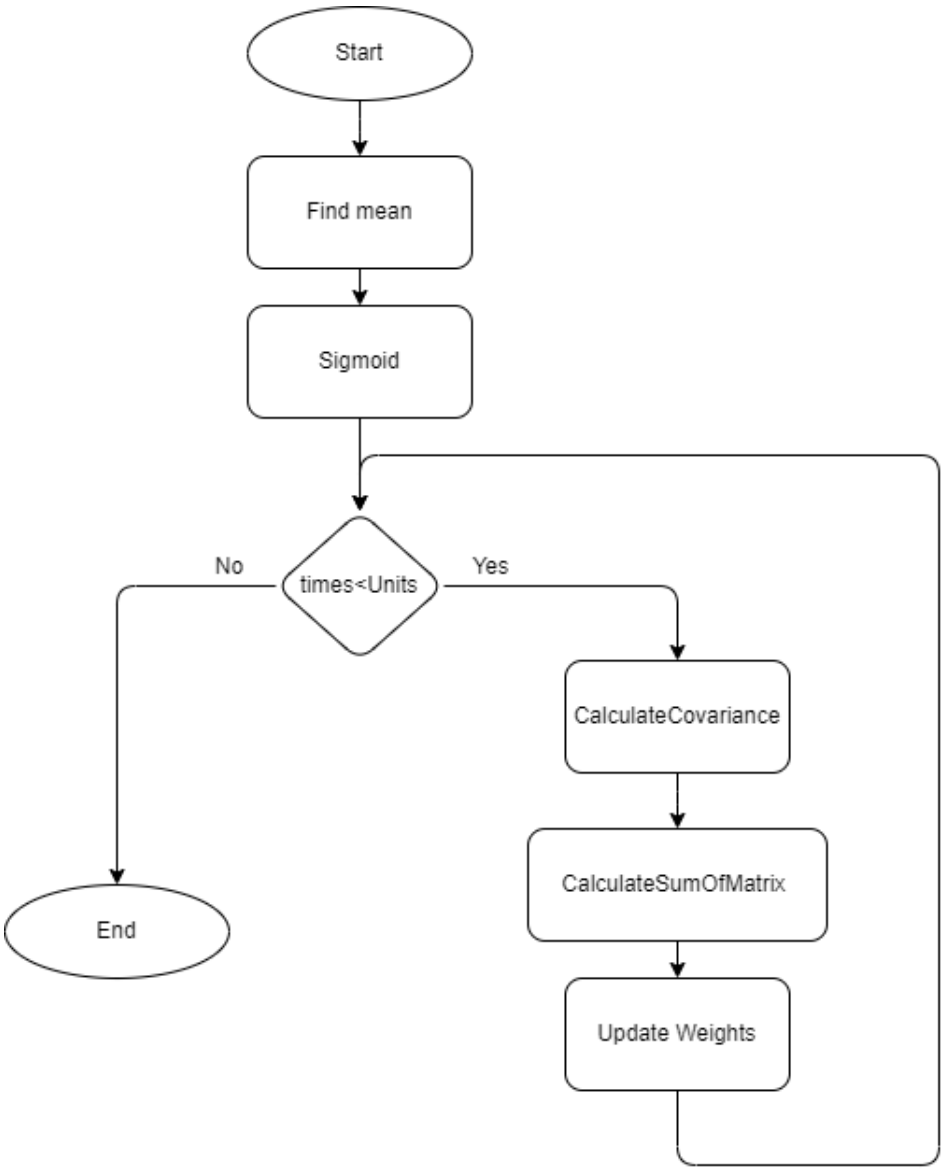

FIGURE 5.10: Hebbian - Covariance Algorithm

Algorithm 6 Hebbian Layer Using Covariance Rule

```
1: procedure HEBBIAN
2:     a1 ← 1/16
3:     a2 ← 1/(16 − 1)
4:     ▷ Finds mean of batches for each unit
5:     for j ← 0 to 2048 do
6:         for i ← 0 to 16 do
7:             sumPerUnit[j] ← sumPerUnit[j] + Z1[i][j]
8:         end for
9:         mask[j] ← sumPerUnit[j] ∗ a1
10:    end for

11:    ▷ Pass the mask through a sigmoid function
12:    for i ← 0 to 2048 do
13:        mask[j] ← 1.0/(1 + hls :: expf(−100 ∗ (mask[u] − 0.01)))
14:    end for
15:    ▷ 3D Covariance Rule applied
16:    for u ← 0 to 2048 do
17:        ▷ Starts to compute 2D covariance for each unit
18:        for s ← 0 to 9 do
19:            ▷ Find the correct pixel from the image
20:            pixel ← wlo1[s][u]
21:            weight = W1_value[s][u]

       ▷ Creating the input of the covariance the rows are the batches and
   the columns are the synapses
22:            for b ← 0 to 16 do
23:                inputCov[b][s] ← Xinput[b][pixel] ∗ wvalue;
24:            end for
25:        end for
       ▷ Covariance of 2D matrix - gives the relationship of every synapse
   through the batch
26:        for j ← 0 to 9 do
27:            for i ← 0 to 16 do
28:                sum[j] ← sum[j] + inputCov[i][j]
29:            end for
30:            avg[j] ← sum[j] ∗ a1
31:        end for
```

Algorithm 7 Part 2

```
32:         for j ← 0 to 9 do
33:             for i ← 0 to 16 do
34:                 mx[i][j] ← inputCov[i][j] − avg[j]
35:             end for
36:         end for
37:         ▷ Covariance Output
38:         for i ← 0 to 9 do
39:             for j ← 0 to 9 do
40:                 for k ← 0 to  do
41:                     tempmul[k] ← mx[k][i] ∗ mx[k][j]
42:                 end for
43:                 for k ← 0 to  do
44:                     mout[i][j] ← (mout[i][j] + tempmul[k])
45:                 end for
46:                 mout[i][j] ← a2 ∗ mout[i][j]
47:             end for
48:         end for
49:         for s1 ← 0 to 9 do
50:             for s2 ← 0 to 9 do
51:                 dW1[u][s] ← dW1[u][s] + mout[s][s2]
52:             end for
53:             dW1[u][s] ← dW1[u][s] ∗ mask[u]
54:             W1_updated[u][s] ← W1_value[u][s] + (etahl ∗ dW1[u][s])
55:         end for
56:     end for
57:     return
58: end procedure
```

Having implemented the C algorithm and tested it against the numpy results on the same dataset, we began optimizing the actual code.

Initially, we isolated the identical computations occurring in every loop. Therefore, to minimize computation time, we excluded them from the loop and performed the calculations only once at the beginning of the program.

Then, as an optimization strategy, we began implementing the appropriate Vivado directives in our code. Each time a change was made we analyzed both synthesis and analysis tabs to understand how the design was affected.

So the first thing was to organize the code into functions and create hierarchies inside our FPGA. This helped to easily apply directives targeted to specific parts of our design. For instance, one of the most crucial directives is Dataflow. Despite the fact that it does not impact the latency of the design, it did improve the throughput. We can calculate the Clock Cycles by using the following equation.
$CCtotal = latency + ((batchesPerEpoch - 1) * interval)$

In particular, in our design, we have a latency of around 567066 Clock cycles but we have an interval of 252618 so the total clock cycles for one epoch is $CCtotal = 567066 + ((3000 - 1) * 252618)$ instead of CCtotal = (3000 * 567066) which was at the beginning where interval and latency are equal. This is illustrated in the figure 5.12

This could not be possible without incorporating the separate functions in combination with the Dataflow pragma. However, Dataflow Pragma utilization follows a set of predefined rules, as listed earlier, thus we were forced to increase the memory usage of some data that were handled by more than one functions. For example, the weights of the first layer and its location matrix are used both by the Forward propagation and by the Covariance Rule, so in order to be able to use Dataflow, we had to store 2 different matrices with the same data that will be read by different functions. In this way, memory cohesion is preserved. In figure 5.11 we show how the data moves inside those functions.

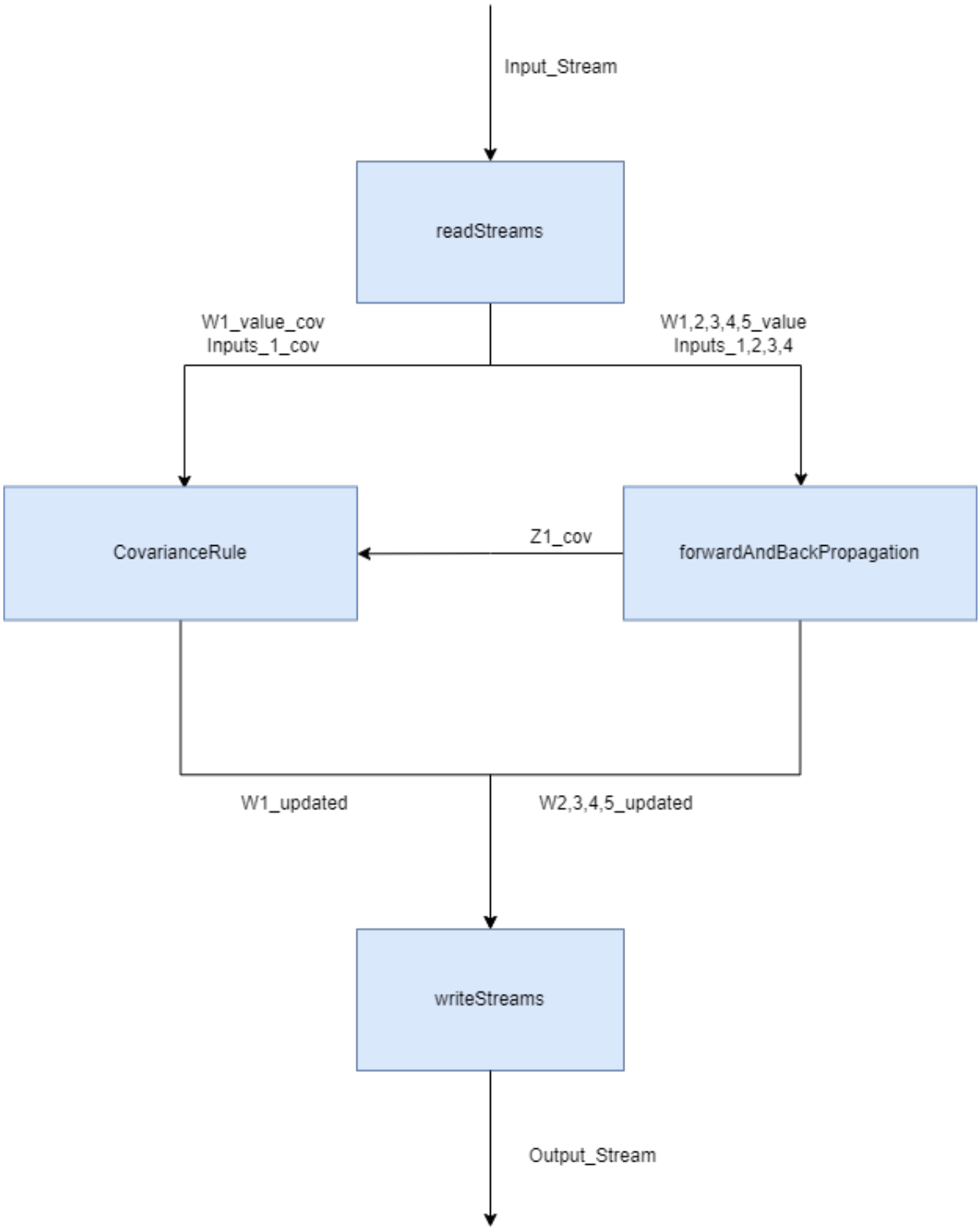

FIGURE 5.11: Functions and DataFlow

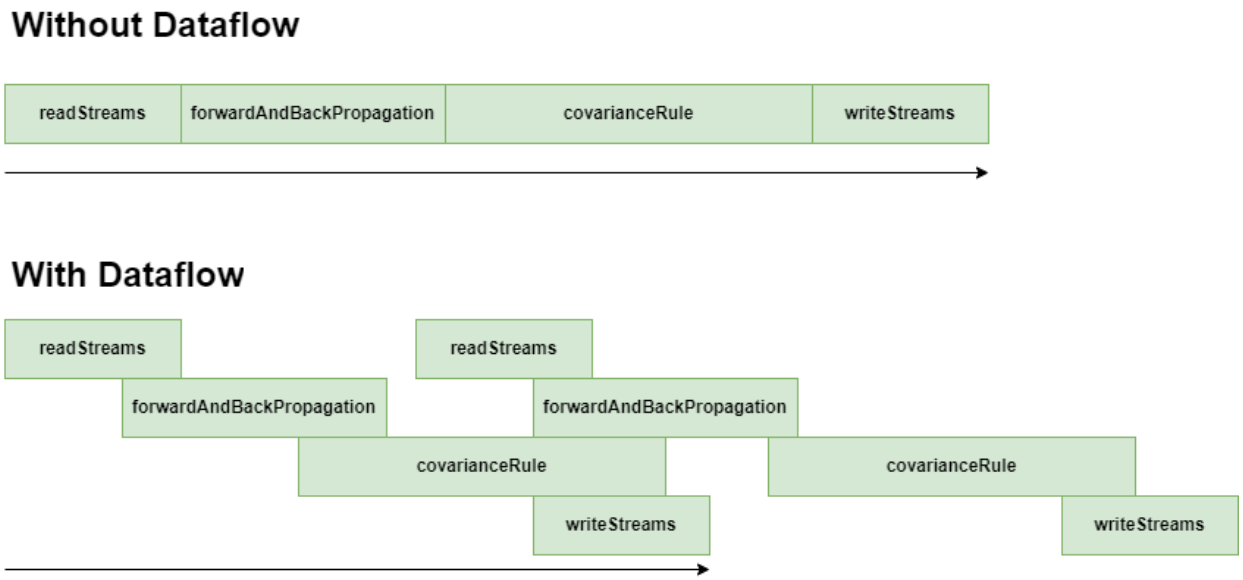

FIGURE 5.12: DataFlow impact on our design

The use of Pipeline Pragma was another optimization utilized. Incorporating this directive into the for-loop gives results as if concurrent operations of the loop are allowed. In this section, we demonstrate only two cases of using it among multiple times its utilization in our project. In addition, it plays an important role in the covariance algorithm and the feed-forward algorithm. In the absence of this pragma, every load, store and computation would be executed in a similar manner to a CPU, in other words, sequentially. Further, we should note that when the pipeline pragma is applied within a nested loop inside the inner loop, a loop unrolling is performed. As a consequence, we should also utilize the memory partitioning pragma to access more memory data and optimize performance.

As discussed earlier, our model has sparse connectivity, so when calculating the output of the layer, we should consider only the related inputs (neurons). In order to accomplish this, we have two different matrices, one for storing the weight values and one for determining what input these values correspond to. Below, a flowchart shows how each output is calculated.

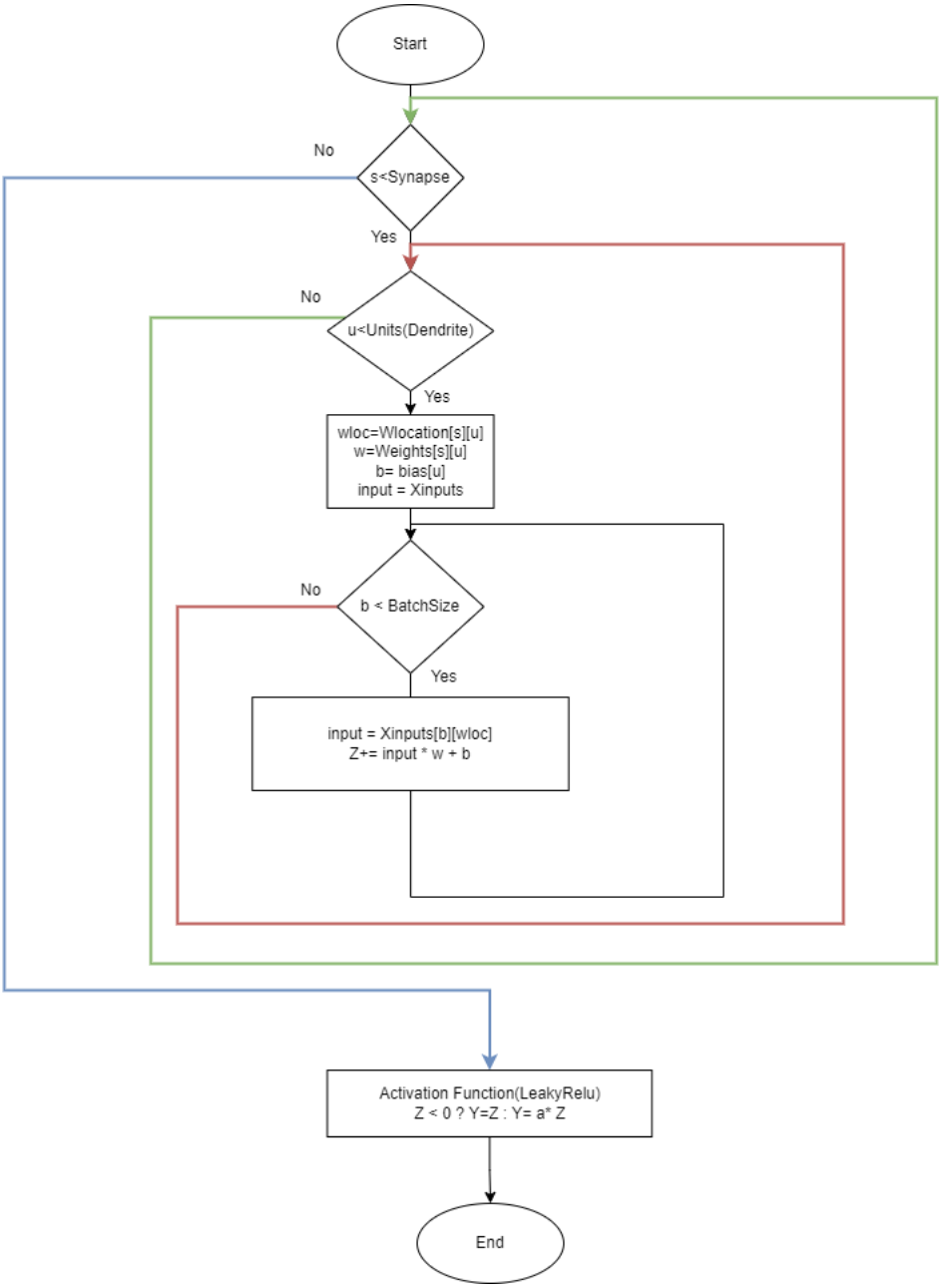

FIGURE 5.13: Flowchart of forward algorithm

This order of calculation allows us to accomplish two things. The first is that for the same unit the Wlock and Wmask remain the same, which allows us to apply them to the whole batch. The second is that it facilitates the optimization of the process, as we can parallelize it without spending excessive resources. We applied the pipeline inside each synapse so that we can see that each synapse has a pipeline of units, as illustrated in Figure 5.14, and each unit has parallel computations, as illustrated in Figure 5.15.

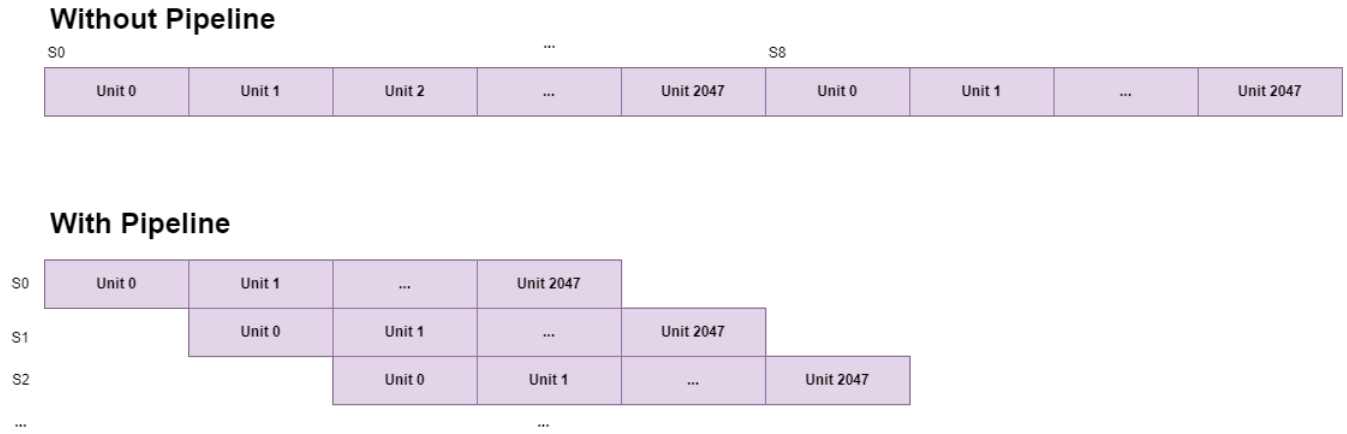

FIGURE 5.14: Pipeline of each Synapse

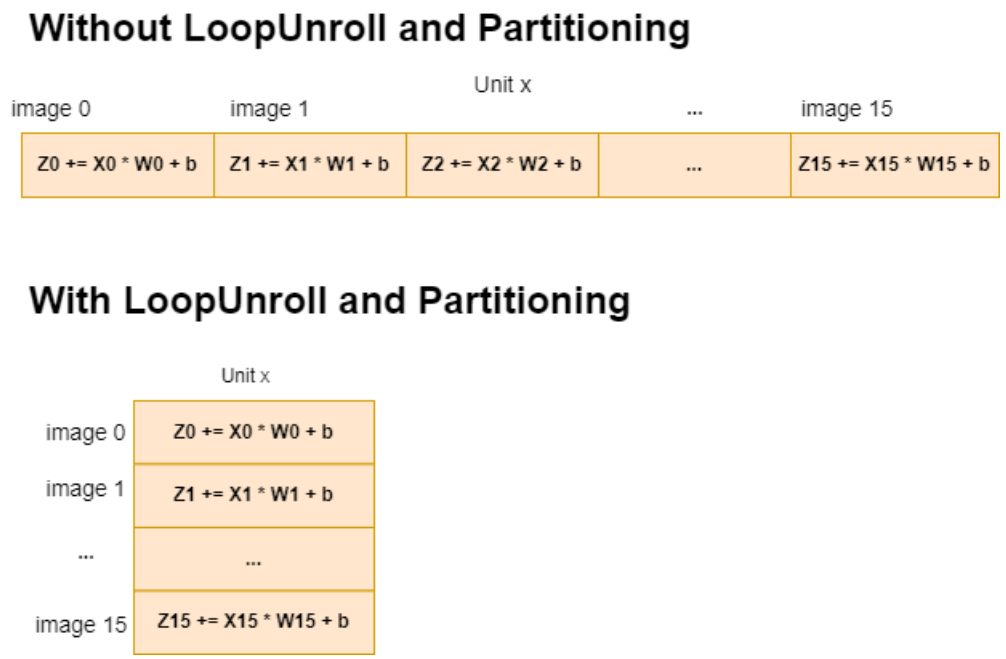

FIGURE 5.15: Loop Unroll impact on our design

To summarize, by examining the Synthesis5.17 and Analysis5.16 tool in the Vivado HLS, we can observe that our design achieves a latency of 563343 clock cycles with a 10 nanosecond clock. Furthermore, we have an interval of 252318, which was our main objective to reduce. In our FPGA, this is determined by the Covariance Rule, which is the slowest part. As well, we can see that the resources are kept below 60%. This is considered as a condition for the next step, which if met, the design could be successful. We can see that

we have negative slack which is automatically corrected when we export the IP.

	Negative Slack	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
bioDNN_training	0,80	674	197	104185	101788	563343	252318	dataflow
forwardAndBackprop	0,80	298	102	69693	67393	245908	245908	none
cov	0,59	46	95	33338	28194	252317	252317	none
initParam	-	0	0	490	4575	29637	29637	none
writeStreams	-	0	0	456	1052	35478	35478	none

FIGURE 5.16: Analysis Tab of our HLS Design

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.554	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
563343	563343	252318	252318	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	128	-
FIFO	0	-	15	132	-
Instance	344	197	104013	101254	0
Memory	330	-	128	13	0
Multiplexer	-	-	-	261	-
Register	-	-	29	-	-
Total	674	197	104185	101788	0
Available	1824	2520	548160	274080	0
Utilization (%)	36	7	19	37	0

FIGURE 5.17: Synthesis Tab of our HLS Design

5.3.2 Vivado IDE and Design Architecture

Since we have already discussed our IP Block on the PL, we can now analyze our Vivado architecture and its interface. As it is shown in figure 5.18 our IP communicates with the PS using a DMA and 2 FIFOs.

DMA

The AXI DMA is a Xilinx IP core enabling high-bandwidth memory access for AXI4-Stream peripherals. It facilitates data movement between memory and peripherals, with MM2S and S2MM channels operating independently. It offers burst mapping, queueing, and byte-level realignment. Configurable in polling, interrupt, or Scatter-Gather modes, it starts transactions by writing to its registers. DMA transfers up to 1024 bits per cycle on a memory bus.

FIFO

FIFOs are used in FPGAs to store and retrieve data in an order. They are designed to consume as little memory as possible. FIFOs are useful since the IP(PL) and PS have different clock domains. FIFOs also act as temporary storage buffers, allowing data to be temporarily stored when the rate of data production exceeds the rate of consumption or vice versa. This helps in decoupling data sources and sinks, improving overall system performance. An IP, for example, can produce results faster than the processor can consume them. By using FIFO, the IP will not be stalled by the PS.

Master-Slave Protocol

A Master-Slave protocol refers to a communication scheme in which one device initiates and controls communication with another device, known as the slave. The master device transmits commands and data, while the slave device responds to these commands and performs the requested actions. It is commonly used in memory interfaces, peripheral communication, and bus architectures.

In Xilinx devices, the Master-Slave protocol is utilized extensively within the context of various interfaces, including AXI (Advanced eXtensible Interface), memory interfaces, and peripheral interfaces. For example, in an AXI-based

system, the AXI master initiates read or write transactions to access memory-mapped registers or memory locations, while the AXI slave responds to these transactions by performing the requested data transfers or operations.

AXI4 Interface Protocol

AXI4[31], an evolution from AXI3, boosts interconnect performance and efficiency, especially in multi-master setups. It introduces features like extended burst lengths (up to 256 beats), Quality of Service signaling, and support for multiple region interfaces.

As a simplified version of AXI4, AXI4-Lite is designed for components with simpler interfaces. Transactions are limited to one burst length, with data accesses matching the bus width and no exclusive accesses. It is ideal for simple, low-throughput memory-mapped communication, such as control signals of a memory-mapped module.

AXI4-Stream facilitates one-way data transfers from master to slave, minimizing signal routing. It accommodates single or multiple data streams over shared wires, and various data widths within the same interconnect. It's particularly well-suited for FPGA implementations and high-speed streaming data.

Architecture - Design explanation

Having discussed all of the above, we can now describe our design in more detail. Two axis Stream ports are provided, one for sending data to the PL and one for receiving it. These ports are connected to the DMA, which is responsible for moving data between the DDR and PL. Zynq is not aware that the DMA is connecting with two FIFOS. A FIFO is used for incoming data and another for outgoing data. All interrupts are also connected in order to be able to use either the polling method or the interrupt method.

Additionally in our system, we implemented data packing between the processor system (PS) and the programmable logic (PL) in order to maximize data transfer efficiency and system performance. By combining multiple data items into a single transfer, we can fully utilize the available bandwidth between the PS and PL, reducing overhead and latency associated with individual transfers. This approach also allows for better resource utilization within

the FPGA, enabling parallel processing of multiple data items and improving overall system throughput. Additionally, data packing facilitates synchronization between the PS and PL by ensuring that related data items are transferred together, enhancing system reliability and predictability. Therefore, we have used a data packing of 64 bits, which means that we have combined two variables. Even though Zynq's HP port can support a maximum memory bus size of 128 bits, we observed no significant improvement, so we decided to maintain the 64-bit data width.

This increased the bandwidth of our design, and there are several other ways to accomplish this. Despite this, we chose not to dive further into this matter, instead choosing to optimize the firmware on the processor and IP rather than create a complex communication design that would be difficult to scale and maintain.

Design Steps

Once the Block design was completed, we had to validate it by passing it through Syntheses, which transforms the RTL design into an array of gates. Here are the first results of logic utilization and timing analysis, which are more reliable after the implementation of our design on the FPGA resources. Placement and routing are the steps involved in the implementation of our design. Typically the first step is the optimization of the netlist created by synthesis, then it places the netlist on the FPGA cells and then builds the interconnection between those cells. Following the implementation step, the timing analysis and utilization report should be reviewed. It is not guaranteed that the system will behave as expected if those steps do not succeed. As part of this step, we can also check whether there is space for increasing the clock in order to optimize the design time-wise. This can be calculated by using the 5.1 equation[32].

$$f_{max} = \frac{1}{T - WNS}, \tag{5.1}$$

where WNS is the worst negative slack and T is the period of the current clock.

For example, in our case, we started with a clock at 100MHz and observed that the design's WNS was 1.613ns. So, we applied this equation and increased our clock to 119MHz, which dropped the training time. However,

more details will be discussed in the next chapter.

Bitstream generation was the last step. After all these steps were completed successfully, we programmed our system's processor using the Vivado SDK.

5.3.3 Vivado SDK

With the Vivado SDK, one can control and program the FPGA processor. Each component in the FPGA is memory mapped, and using the Vivado SDK, we can write software applications that interact with those hardware components. This interaction may involve reading from or writing to registers, controlling peripherals, and managing data flow between the processor and the FPGA fabric.

We used an SD card to pass the dataset to the DDR. For the Weights, we created header files that should be initialized every time with the random algorithm. Our program first initializes the memory, then initializes the DMA and the IP, and then starts sending and receiving data. The following table presents in detail what those data are and in what order they are sent. Every time an epoch is finished our program prints the average accuracy and error of our training. Our program works by polling the IP in order to check if it is finished and to send new data to it. As a future work, interrupts may be introduced in order to run some validation tests while the next epoch is running in the PL.

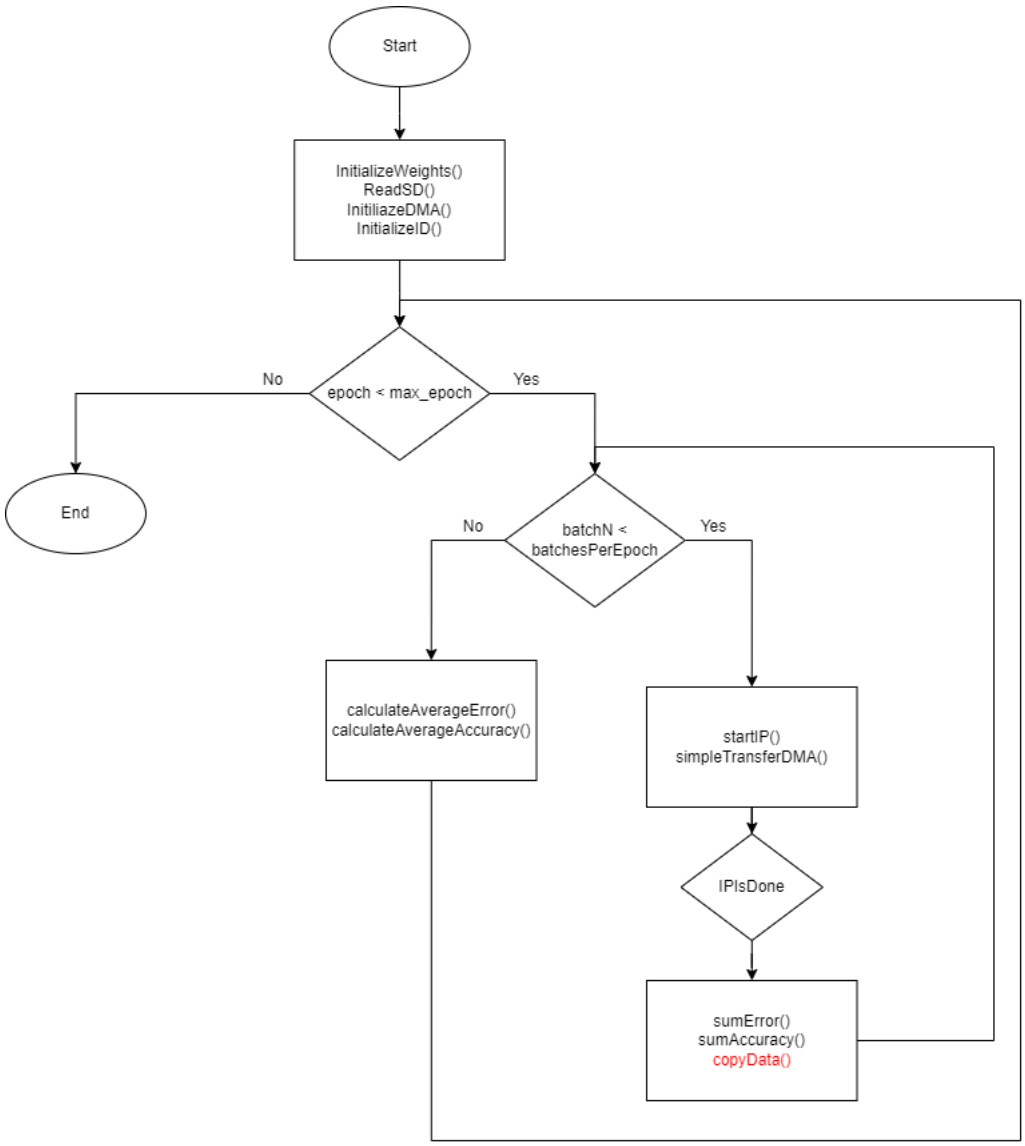

FIGURE 5.19: FlowChart of the microcontroller-With red color is the part that we removed later and is explained below

After analyzing the initial run of our program, we noticed that a significant portion of each epoch's time was being consumed by data copying. We realized the urgent need for optimization. It was estimated that approximately one-third of the total epoch time was devoted to this task, which was an unacceptable loss of time. Theoretically, we could make our design 1.5 times faster based on Amdahl's Law.

Our solution involved a drastic redesign: instead of employing two separate matrices that required copying between them, we adopted a single array with dimensions matching the maximum size between these matrices. As a

result of this new approach, data is overwritten in place rather than duplicated.

After designing our matrix layout, we separated the data into alternating sections and stable sections based on the layout of our matrix. As a result, in order to transmit or receive data over IP, we only need to handle the necessary bytes that are passed to the DMA. In this way, we significantly reduced unnecessary operations. By distinguishing between alternating and stable data as well as common and differently interpreted data zones, we've devised a systematic strategy for efficient data flow within our program.

After incorporating all these methods and techniques, we managed to reach the 1.5x speed-up theoretical limit and our time per epoch had decreased from 28 seconds to 19 seconds.

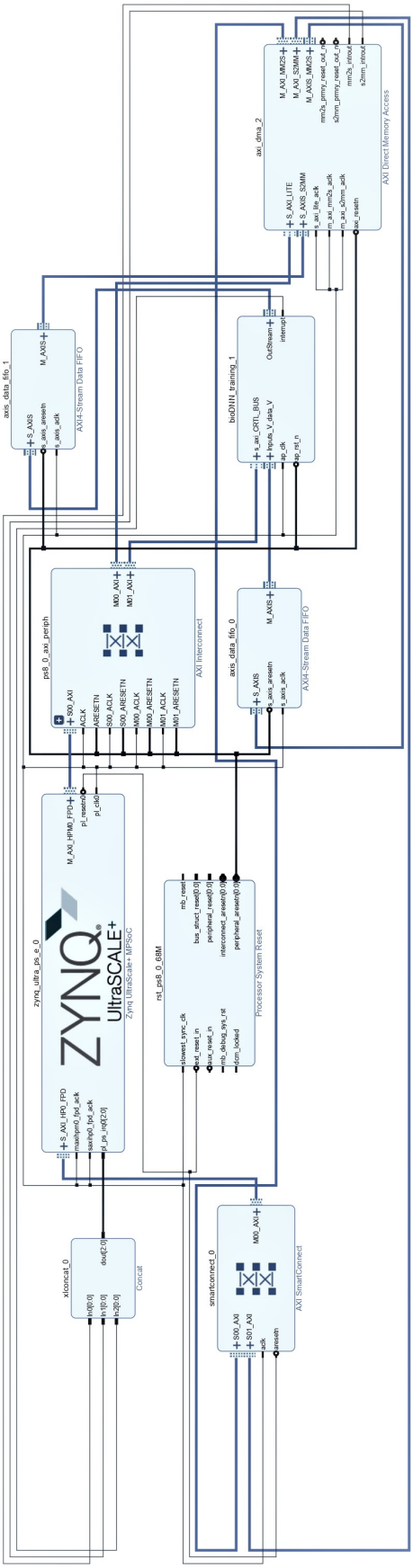

FIGURE 5.18: Bio-inspired ANN Block Design for FPGA

Chapter 6

Results

In this chapter, we will discuss the findings of our study. First, we will provide details about the specifications of the CPU and GPU that we used in our comparative analysis. Next, we will analyze the resource utilization required for our architecture implementation on the FPGA with plate number 1. Then, we will thoroughly examine the performance metrics used for comparisons. Finally, we'll compare the results of our bio-inspired ANN's FPGA-based architecture with those of CPU and GPU implementations, which we developed using Keras.

6.1 Specification of Compared Platforms

6.1.1 AMD Ryzen™ 7 3700X

Despite the fact that CPU-based applications are easier to implement, their low parallelism and high power consumption make them inefficient in terms of both time and energy. The following table 6.1 presents CPU platform specifications.

TABLE 6.1: AMD Ryzen™ 7 3700X Specifications

AMD Ryzen™ 7 3700X	
Total Cores	8
Total Threads	16
Processor Base Frequency	3.60 GHz
Max Turbo Frequency	4.40 GHz
TDP	65 W
Max Memory Bandwidth	47.68 GiB/s
Lithography	7nm

6.1.2 NVIDIA GeForce GTX 1050 Ti

GPUs are capable of parallel processing, delivering incredible acceleration when the same workload must be executed many times in rapid succession. Their disadvantage is that they tend to consume a large amount of energy/power. In table 6.2, below, the specifications of NVIDIA GeForce GTX 1050 Ti are described.

TABLE 6.2: GPU Specifications

	NVIDIA GeForce GTX 1050 Ti
CUDA Cores	768
GPU Memory	4 GB GDDR5
Boost Clock	1392 MHz
Memory Interface	128-bit
Memory Bandwidth	112 GB/s
Power Consumption	75 W

6.1.3 NVIDIA GeForce RTX 3060 12 GB

NVIDIA RTX series demonstrates a significant difference against the GTX series concerning the core they employ. The CUDA cores, as shown in table 6.2, are the processing units in the NVIDIA GTX series, which are in charge of executing parallel tasks. Whereas, Tensor cores, found in newer NVIDIA GPUs like those in the RTX series, table 6.3, are ideal for deep learning tasks, particularly for operations involving tensors (multi-dimensional arrays). Having more CUDA cores and dedicated Tensor cores typically results in quicker training and inference times. GPUs equipped with specialized Tensor cores, such as those found in NVIDIA's RTX series, are capable of accelerating specific deep learning tasks, particularly those related to training and inference. This acceleration is achieved through techniques such as mixed-precision training and tensor core-based matrix multiplication.

Below is a table with RTX series GPU platform specifications.

TABLE 6.3: GPU Specifications

	NVIDIA GeForce RTX 3060
Tensor Cores	768
GPU Memory	12 GB GDDR6
Boost Clock	1320 MHz
Memory Interface	192-bit
Memory Bandwidth	360 GB/s
Power Consumption	170 W

6.1.4 Zynq UltraScale+ MPSoC ZCU102

The purpose of this section is to present the final resource utilization of our FPGA-based architecture that have been ported to ZCU-102 FPGA (in table 6.4).

TABLE 6.4: FPGA-based architecture (ZCU 102) - Resources Utilization

Clock Frequency	**150 MHz**
BRAM Usage (%)	**64%**
DSPs Usage (%)	**7%**
FF Usage (%)	**16%**
LUTs Usage (%)	**49%**

6.2 Throughput and Latency Speedup

6.2.1 Latency

Latency refers to the time required to accomplish a single task. As defined in this thesis, latency is the time it takes for a specific platform to perform training on a batch of images (16 images).

6.2.2 Throughput

In general, throughput is a measure of how many units of information a system can process in a given amount of time. In other words, it refers to the

maximum rate of processing. In our case, throughput is measured as the number of batches trained per second.

$$Throughput = \frac{Batches}{Time(sec)}, \tag{6.1}$$

where a batch consists of 16 images.

6.3 Energy Consumption

Energy consumption (6.2) refers to the amount of energy required to complete a particular task in a given amount of time. The amount of energy is typically measured in Joules (J) or KiloJoules (kJ). It is also important to maintain this parameter at the lowest level possible.

$$E = P \cdot T, \tag{6.2}$$

where E represents the energy in Joules, P indicates the required power for the device to function and T is the time needed to execute the task.

The Images/Joule metric can be calculated as follows:

$$\frac{Images}{Joule} = \max\left(\frac{Throughput}{Power}, \frac{1}{Power \cdot Latency}\right) \tag{6.3}$$

6.4 Power Consumption

Power consumption refers to the amount of electrical energy used by a device over time, typically measured in Watts (W) or kiloWatts (kW). It's a critical factor to consider because it affects not only the performance and longevity of devices but also their environmental impact and operating costs.

For example, in portable electronics like smartphones and laptops, minimizing power consumption is essential for extending battery life. Devices with lower power consumption can run longer on a single charge, making them more convenient and efficient for users. Additionally, reducing power usage in these devices helps decrease energy waste and increase energy efficiency.

6.5 Performance Analysis for all Platforms

This section compares our FPGA-based architecture against Keras implementations using CPUs and GPUs. A summary of the performance analysis can be found in 6.5. In the comparisons below, we use the CPU implementation in Keras as the reference implementation. With our architecture, we see significant improvements in latency (fig. 6.1) and throughput (fig. 6.2).

TABLE 6.5: Performance Evaluation and Comparison - The FPGA-based architecture (ZCU 102 board) compared to Keras-Tensorflow running on both CPU and GPU. Numpy results are not included since the goal of Numpy implementation was to gain a better understanding of the bio-inspired ANN model rather than to optimize it.

	CPU	GPU GTX	GPU RTX	FPGA
Clock Frequency (MHz)	3600	1392	1320	150
Throughput (Batches/s)	6.25	42.85	66.67	222.88
Throughput Speedup	1x	6.82x	10.67	**35.66x**
Latency (ms)	160	23	15	4.49
Latency Speedup	1x	6.96x	10.67	35.66x
Epoch execution time (s)	490	70	45	13.46
Total On-Chip Power (Watt)	65	75	170	6.69
Power Efficiency	1x	0.87x	0.38x	**9.71x**
Energy Consumption (Joule) per Batch	10.04	1.73	2.55	0.03
Energy Efficiency	1x	6.01x	4.08x	**346x**
Images/Joule	0.096	0.571	0.39	33.30

6.5.1 Comparison of FPGA and CPU/GPU versions

In this section, we provide several diagrams to illustrate our design characteristics concerning the latency, throughput, time execution and some energy metrics.

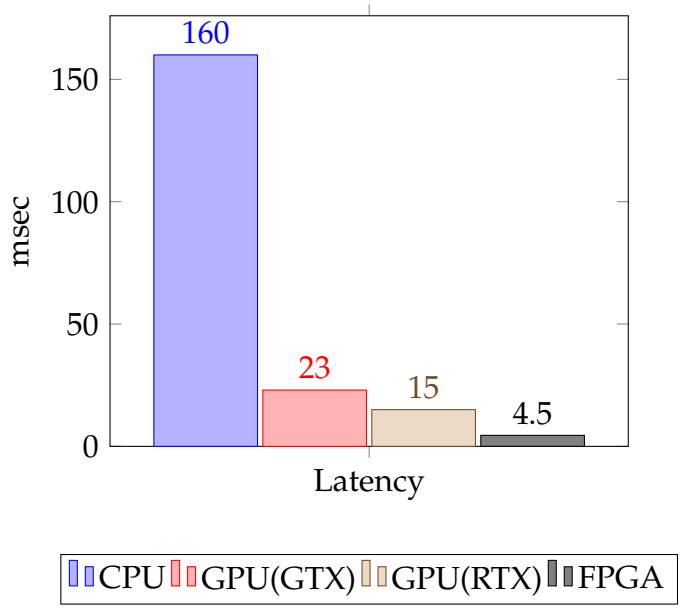

FIGURE 6.1: An analysis of the Latency of the compared platforms.

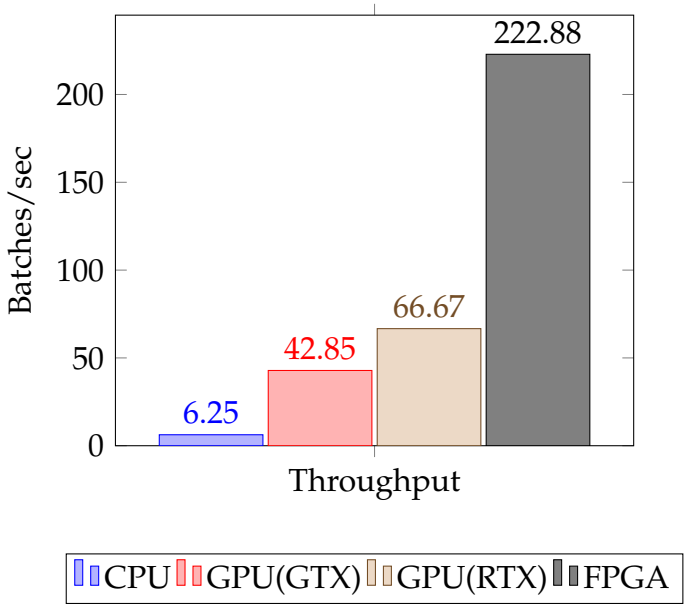

FIGURE 6.2: An analysis of the Throughput of the compared platforms.

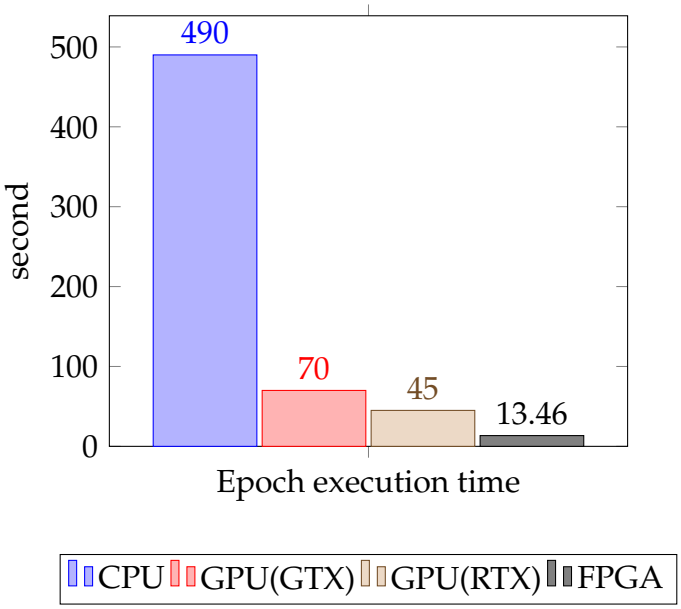

FIGURE 6.3: An analysis of the training execution time for an epoch of the compared platforms.

Our design achieves epoch training in 13.46 seconds, compared to 490 seconds for the CPU and 40 seconds for the RTX series GPU for the entire MNIST dataset (fig.6.3).

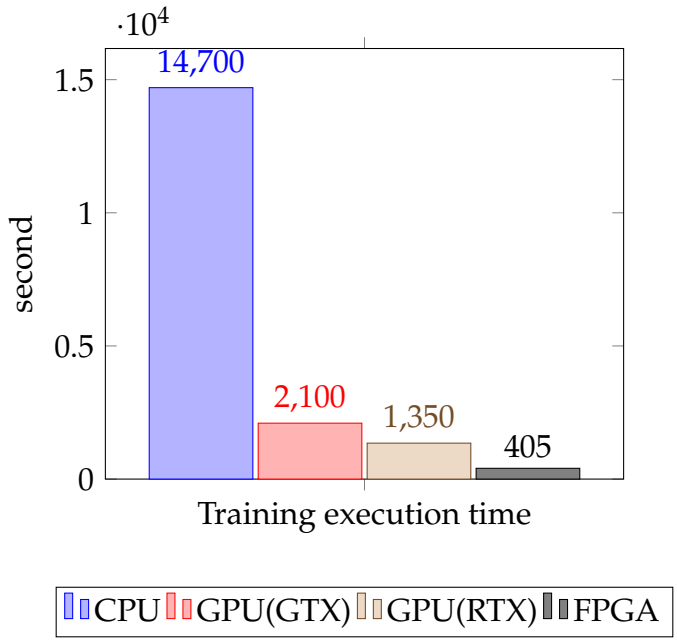

FIGURE 6.4: An analysis of the training execution time for the total training of the compared platforms.

The impact of the FPGA is prominent in the timing section when we consider the length of time required for the entire process of training the network. For

example, in a CPU, it would take 4 hours to train this network of 30 epochs, while in an FPGA, it only takes 6 minutes. In GPU, it would take 22 minutes

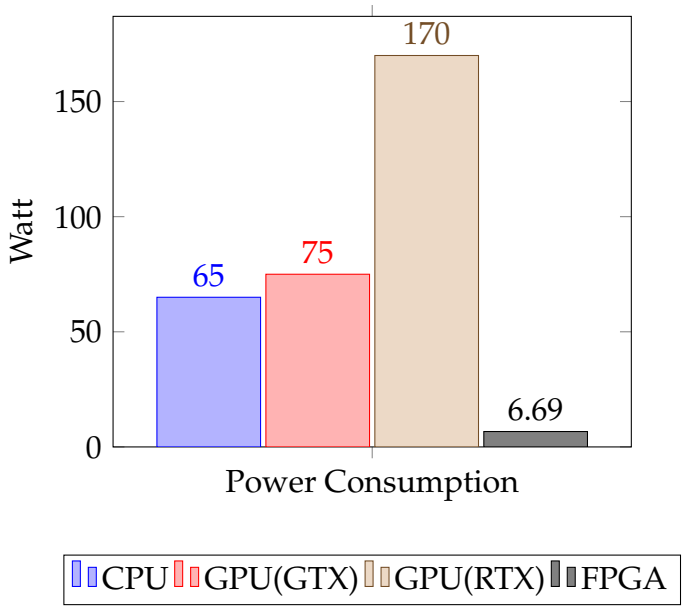

FIGURE 6.5: An analysis of the Power Consumption of the compared platforms.

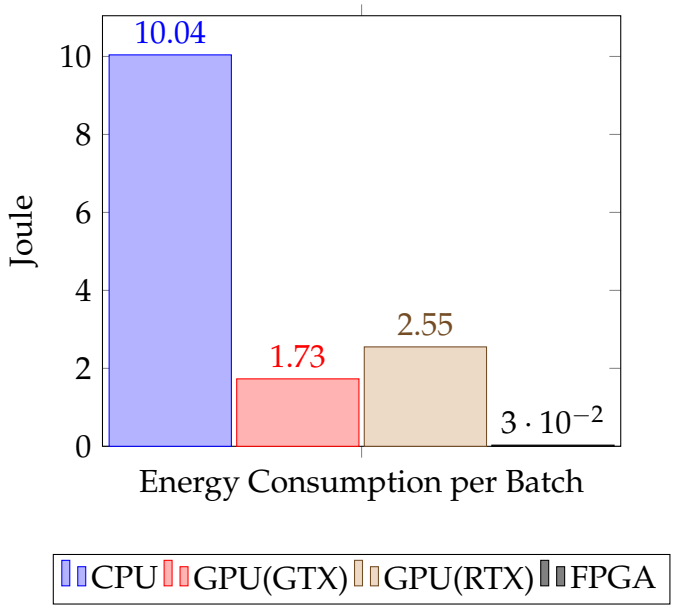

FIGURE 6.6: An analysis of the Energy Consumption per batch of the compared platforms.

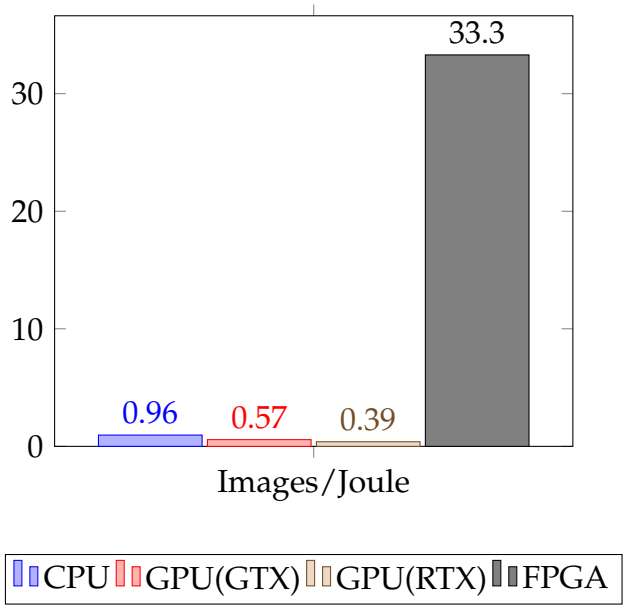

FIGURE 6.7: An analysis of the Images/Joule metric of the compared platforms.

Furthermore, the proposed architecture consumes less power than the CPU implementation or GPU implementation (fig. 6.5), requiring only 6.69 Watts rather than 65 Watts and 75 Watts, respectively. A notable aspect of our FPGA design is its energy efficiency (6.6), which is 346 times greater than CPU efficiency and 85 times higher than GPU efficiency.

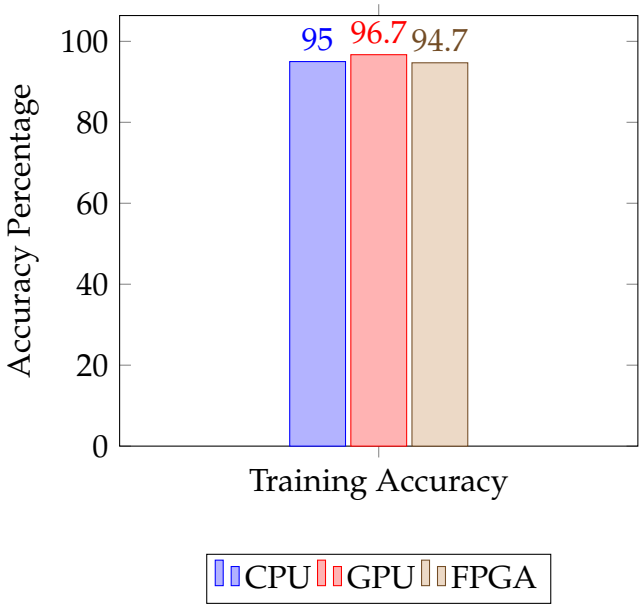

FIGURE 6.8: An analysis of the Accuracy in training and validation of the compared platforms.

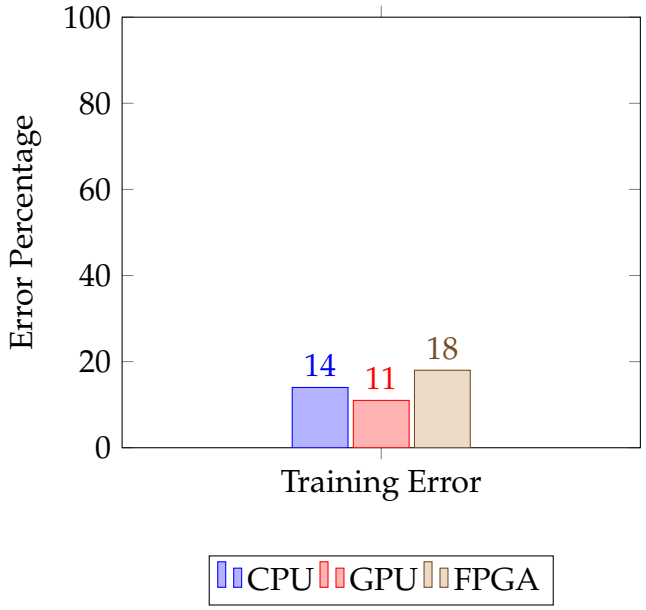

FIGURE 6.9: An analysis of the Error in training and validation of the compared platforms.

As far as training and validation error/accuracy (fig. 6.9, 6.8) are concerned, our FPGA-based architecture yields similar outcomes when compared to implementations on both CPUs and GPUs. The training of the network is considered to be successful, albeit the task was not particularly complex.

The results of this study indicate that even though FPGAs require significant learning efforts, they are still able to deliver better performance when it comes to latency and energy efficiency when compared to conventional CPUs.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Artificial neural networks (ANNs) are a technology inspired by the brain and have found applications in various fields, including computer vision and natural language processing. ANNs are great at recognizing images, understanding speech, and detecting anomalies due to their ability to learn complex patterns and connections from vast amounts of data. However, experts believe that ANNs still have potential for further advancement, and research is focused on improving their performance, efficiency, and interpretability. This involves exploring new network structures like convolutional and recurrent networks and more bio-inspired approaches.

Researchers are also using hardware accelerators to increase the efficiency of neural network computations. Field-Programmable Gate Arrays (FPGAs) have emerged as a promising platform for accelerating artificial neural network inference. FPGAs can be customized to optimize the hardware architecture for specific neural network models, bypassing the traditional use of CPUs or GPUs to achieve significant improvements in speed and energy efficiency. FPGAs are highly compatible with resource-limited settings, making them a perfect fit for edge computing and IoT implementations.

The aim of the thesis was to build an FPGA-based implementation of a bio-inspired ANN training process to enhance its energy/power efficiency and speed up the training process. The ANN model was originally developed using Keras by Postdoctoral Researcher S. Chavlis and Research Director P. Poirazi. The model was rewritten using plain Python and the Numpy library to allow for a more detailed examination and greater comprehension of the ANN's biologically-inspired architecture.

The training process was redesigned using Vivado HLS, which involved a different approach to the Hebbian rule, forward propagation, backpropagation, and updating of the network's parameters (using the Adam optimization algorithm). Pipelining, dataflow, loop unrolling, and array partitioning were applied to boost efficiency and maximize performance. The FPGA architecture was implemented within the Vivado IDE's graphical interface, and a microcontroller was programmed to handle communication with the IP.

Comparing the FPGA design with the CPU implementation (in Keras) on the MNIST dataset revealed significant improvements in latency and throughput with speedups of 35.66x. As a result, an epoch of training is completed in only 13.46 seconds, as opposed to 490 seconds on a CPU. GPU implementation achieved a 10.67x speedup in latency and throughput over CPU implementation, executing an epoch of training in 40 seconds. The FPGA design's most notable feature is its high energy efficiency, which is 346 times greater than that of CPUs and 85 times greater than that of GPUs. In terms of accuracy/error results in training and validation, the FPGA implementation reaches the same level of performance as the CPU/GPU implementation.

7.2 Future Work

7.2.1 Rewiring

In the model, masks remain constant while connectivity structure typically changes in neuroscience. It would be interesting to develop a corresponding bio-inspired ANN, in which the masks (connectivity structure) are modified at regular intervals during training. This feature is called rewiring and seems to be a logical next step in our design, as communication between the PS-PL, the Vivado block design, and the IP would not need to be modified at all.

7.2.2 Interrupts instead of polling method

The transition from polling to interrupt-driven programming can greatly enhance the performance of embedded systems programs. Unlike the polling method, the interrupt method only interrupts the processor when necessary, which reduces overhead and improves efficiency. By incorporating interrupts, system resources can be utilized more efficiently, allowing the processor to handle additional tasks while waiting for external events. For instance,

using this approach, we can allow the microcontroller to perform other computations such as the processing of the validation set, until it is time to acquire the IP results.

7.2.3 Abstracted design

Incorporating more abstract design strategies that promote versatility and adaptability can be highly beneficial in embedded systems programming. By shifting to a higher level of abstraction in our design approach, we can easily adapt programs to different hardware platforms and requirements. This abstract layer would allow us to modify our network without having to design the FPGA from scratch. Additionally, our design achieves a significant speedup, which can enable neuroscientists to experiment with alternative structures and hyper-parameters of the network more efficiently.

7.2.4 Larger scale implementation

Our model is based on nature and has five layers, making it a small DNN. To compare it with current state-of-the-art models, we need to implement it on a larger scale, which requires adding more layers. Alternatively, we can replace the last dense layer of a large CNN with our small bio-inspired ANN.

References

[1] Spyridon Chavlis and Panayiota Poirazi. "Drawing Inspiration from Biological Dendrites to Empower Artificial Neural Networks". In: *Nature* 452.7186 (June 2021), pp. 436–441. URL: https://doi.org/10.1038/nature06725.

[2] Attila Losonczy, Judit K. Makara, and Jeffrey C. Magee. "Compartmentalized dendritic plasticity and input feature storage in neurons". In: 70 (June 2008), pp. 1–10. URL: https://arxiv.org/abs/2106.07490v1.

[3] Tiago Branco and Michael Häusser. "The single dendritic branch as a fundamental functional unit in the nervous system". In: *Current Opinion in Neurobiology* 20.4 (2010), pp. 494–502. URL: https://www.sciencedirect.com/science/article/pii/S0959438810001170.

[4] Xundong Wu et al. "Improved Expressivity Through Dendritic Neural Networks". In: *32nd Conference on Neural Information Processing Systems* (2018), pp. 8068–8079. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/e32c51ad39723ee92b285b362c916ca7-Paper.pdf.

[5] Spyridon Chavlis, Panagiotis C. Petrantonakis, and Panayiota Poirazi. "Dendrites of dentate gyrus granule cells contribute to pattern separation by controlling sparsity". In: *Hippocampus* 27.1 (Jan. 2017), pp. 89–110. DOI: https://doi.org/10.1002/hipo.22675.

[6] Panayiota Poirazi, Terrence Brannon, and Bartlett W. Mel. "Pyramidal Neuron as Two-Layer Neural Network". In: *Neuron* 37.6 (Jan. 2003), pp. 989–999. ISSN: 0896-6273. DOI: https://doi.org/10.1016/S0896-6273(03)00149-1. URL: https://www.sciencedirect.com/science/article/pii/S0896627303001491.

[7] Panayiota Poirazi and Bartlett W. Mel. "Impact of Active Dendrites and Structural Plasticity on the Memory Capacity of Neural Tissue". In: *Neuron* 29.3 (2001), pp. 779–796. ISSN: 0896-6273. DOI: https://doi.org/10.1016/S0896-6273(01)00252-5. URL: https://www.sciencedirect.com/science/article/pii/S0896627301002525.

[8] Jordan Guerguiev, Timothy P Lillicrap, and Blake A Richards. "Towards deep learning with segregated dendrites". In: *eLife* 6 (Jan. 2017). Ed. by Peter Latham, e22901. ISSN: 2050-084X. DOI: 10.7554/eLife.22901. URL: https://doi.org/10.7554/eLife.22901.

[9] Aritra Bhaduri et al. "Spiking neural classifier with lumped dendritic nonlinearity and binary synapses: A current mode VLSI implementation and analysis". en. In: *Neural Comput.* 30.3 (Mar. 2018), pp. 723–760.

[10] Panayiota Poirazi and Athanasia Papoutsi. "Illuminating dendritic function with computational models". In: *Nature Reviews Neuroscience* 21.6 (June 2020), pp. 303–321. URL: https://doi.org/10.1038/s41583-020-0301-7.

[11] F. Valverde and M. V. Facal-Valverde. "Neocortical layers I and II of the hedgehog (Erinaceus europaeus): I. Intrinsic organization". In: *Anatomy and Embryology* 173.3 (Feb. 1986), pp. 413–430. ISSN: 1432-0568. DOI: 10.1007/bf00318926. URL: http://dx.doi.org/10.1007/BF00318926.

[12] Allen Newell. "A Step toward the Understanding of Information Processes: Perceptron. An Introduction to Computational Geometry." In: *Science* 165.3895 (Aug. 1969), pp. 780–782. ISSN: 1095-9203. DOI: 10.1126/science.165.3895.780. URL: http://dx.doi.org/10.1126/science.165.3895.780.

[14] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. London: Springer, 2019.

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (May 2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: http://dx.doi.org/10.1038/nature14539.

[20] Anirudh Apparaju and Ognjen Arandjelovic. "Towards New Generation, Biologically Plausible Deep Neural Network Learning". In: *Sci* 4 (Dec. 2022), p. 46. DOI: 10.3390/sci4040046.

[21] Emmanouil Kousanakis et al. "An Architecture for the Acceleration of a Hybrid Leaky Integrate and Fire SNN on the Convey HC-2ex FPGA-Based Processor". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2017), pp. 56–63. DOI: 10.1109/FCCM.2017.51.

[22] Thomas Bohnstingl et al. "Online Spatio-Temporal Learning in Deep Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* (2022), pp. 1–15. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2022.3153985.

[23] Arnold, Lukas On and Owaida, Muhsen. “Single-Pass Covariance Matrix Calculation on a Hybrid FPGA/CPU Platform”. In: *EPJ Web Conf.* 245 (2020), p. 09006. DOI: 10.1051/epjconf/202024509006. URL: https://doi.org/10.1051/epjconf/202024509006.

[24] Wenjie Luo et al. “Understanding the Effective Receptive Field in Deep Convolutional Neural Networks”. In: *29th Conference on Neural Information Processing Systems* 72.12 (Dec. 2001), pp. 4477–4479. URL: http://link.aip.org/link/?RSI/72/4477/1.

[25] "Lampros Pantzekos". *"Bioinspired DNN Architectures with Dendritic Structure"*. 2023.

External Links

[15] "Architecture of an ANN". In: (). URL: https://www.analyticsvidhya.com/blog/2021/07/understanding-the-basics-of-artificial-neural-network-ann/.

[17] "The sigmoid function". In: (). URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[18] "Overfitting vs Underfitting". In: (). URL: https://docs.xili[https://www.ibm.com/cloud/learn/overfitting.

[19] "overfitting". In: (). URL: https://www.ibm.com/cloud/learn/overfitting.

[26] "MNIST database of handwritten digits." In: (). URL: http://yann.lecun.com/exdb/mnist/.

[27] "Amdahl's law - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Amdahl%27s_law.

[28] "ZCU102 Evaluation Board." In: (). URL: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[29] "Vivado Design Suite User Guide: Using the Vivado IDE - UG893." In: (). URL: https://docs.xilinx.com/v/u/2019.1-English/ug893-vivado-ide.

[30] "Vivado Design Suite User Guide: High-Level Synthesis - UG902." In: (). URL: https://docs.xilinx.com/v/u/2019.1-English/ug902-vivado-high-level-synthesis.

[31] "AXI4 Interface Protocol". In: (). URL: https://www.xilinx.com/products/intellectual-property/axi.html#details.

[32] "UltraFast Design Methodology Guide for the Vivado Design Suite". In: (). URL: https://docs.xilinx.com/v/u/2019.1-English/ug949-vivado-design-methodology.