TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Bioinspired DNN Architectures with Dendritic Structure

Author:
Lampros PANTZEKOS

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

September 13, 2023

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Bioinspired DNN Architectures with Dendritic Structure

by Lampros PANTZEKOS

Artificial Neural Networks (ANNs) implemented in Deep Learning architectures have been successfully used to solve a wide range of challenging machine learning tasks. However, in order to achieve top performance, they typically require a substantial amount of energy. In contrast, the brain operates at a very low energy level. Drawing inspiration from biological dendrites and the aforementioned limitations of ANNs, the Poirazi lab of IMBB-FORTH introduced a bio-inspired ANN architecture with a dendritic structure and receptive field. Regarding the learning rule, backpropagation is fully applied. Training parameters are updated using the Adam optimization algorithm instead of the classical gradient descent algorithm. Based on their initial high-level Keras implementation, a lower-level Numpy implementation was developed in this thesis to analyze and understand in depth this model and its training process. Following this, an FPGA-based architecture for the training process of this bio-inspired ANN was designed, implemented, and downloaded onto the Xilinx ZCU 102 board in this thesis. In this developed FPGA implementation, training has been accelerated and power/energy consumption has been greatly reduced as a result of leveraging the high parallelism and power efficiency of the FPGA. In particular, our proposed FPGA implementation executes an epoch of training (for the MNIST dataset) in only 2.3797 seconds rather than 37 seconds on the CPU (Keras) and 17 seconds on the GPU (Keras). Furthermore, it achieves 106.15 times greater energy efficiency than the CPU implementation (Keras) and 56.5 times greater energy efficiency than the GPU implementation (Keras).

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Bioinspired DNN Architectures with Dendritic Structure

by Lampros PANTZEKOS

Τα Artificial Neural Networks (ANNs), τα οποία είναι υλοποιημένα σε αρχιτεκτονικές Deep Learning, έχουν επιτυχώς επιλύσει ένα μεγάλο έυρος απαιτητικών προβλημάτων μηχανικής μάθησης. Ωστόσο, προκειμένου να επιτευχθεί μέγιστη απόδοση, απαιτούν υψηλά επίπεδα ενέργειας. Σε αντίθεση, ο εγκέφαλος λειτουργεί σε πολύ χαμηλά επίπεδα ενέργειας. Αντλώντας έμπνευση από τους δενδρίτες στην βιολογία και τους παραπάνω περιορισμούς των ANNs, το Poirazi lab του IMBB-ITE, παρουσίασε μία βιοεμπνευσμένη ANN αρχιτεκτονική, η οποία περιλαμβάνει δενδριτική δομή και receptive field. Αναφορικά με τον κανόνα εκμάθησης του μοντέλου, εφαρμόζεται πλήρως backpropagation. Οι παράμετροι εκμάθησης ενημερώνονται μέσω του Adam optimization αλγόριθμου, αντί της κλασικής gradient descent μεθόδου. Με βάση την αρχική τους, υψηλού επιπέδου υλοποίηση του μοντέλου σε Keras, υλοποίηθηκε αυτό σε αυτή τη διπλωματική σε χαμηλότερο επίπεδο σε Numpy, ώστε να αναλυθεί διεξοδικά το μοντέλο και η διεργασία του training. Έπειτα στα πλαίσια αυτής της διπλωματικής, σχεδιάστηκε και υλοποιήθηκε η αρχιτεκτονική της training διεργασίας αυτού του μοντέλου σε FPGA, καθώς και μεταφορτώθηκε αυτή η σχεδίαση στην πλακέτα ZCU 102 της Xilinx. Μέσω αυτής της υλοποίησης σε FPGA, το training επιταχύνθηκε και η κατανάλωση ενέργειας μειώθηκε σημαντικά, εκμεταλλευόμενοι την υψηλή παραλληλοποίηση και την ενεργειακή αποδοτικότητα που παρέχει η FPGA. Συγκεκριμένα, η υλοποίηση μας σε FPGA εκτελεί ένα epoch του training (για το MNIST dataset) σε μόνο 2.3797 δευτερόλεπτα αντί για τα 37 δευτερόλεπτα που απαιτεί η CPU (Keras) και τα 17 δευτερόλεπτα που απαιτεί η GPU (Keras). Επιπλέον, είναι 106.15 φορές πιο αποδοτική ενεργειακά συγκριτικά με την CPU και 56.5 φορές συγκριτικά με την GPU.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Apostolos Dollas, for his continued support and his valuable guidance during the procedure of this thesis. Moreover, I would like to express my gratitude for his knowledge and experience, which have been a source of inspiration and have contributed to my decision to work in the field of hardware.

Furthermore, I would like to thank the rest of my thesis committee, Prof. Michail Zervakis and Prof. Panayiota Poirazi, for evaluating my work.

In addition, I would like to express my gratitude to the Research Director Panayiota Poirazi and the Postdoctoral Researcher Spyros Chavlis from the Poirazi lab of the IMBB-FORTH for providing me with this opportunity to work on this thesis and expand my horizons beyond my current area of expertise. I would also like to specifically thank Spyros Chavlis for his time and for the enlightening meetings that enabled me to understand better the basic biological background for the bio-inspired features of my thesis model.

It is with great pleasure that I would like to express my gratitude to my friend and colleague, Nikoletta Palatianna, for her constant support and encouragement, as well as for her significant contribution to this thesis.

Furthermore, I would like to thank the ICS/FORTH CARV team for their guidance throughout this thesis, especially Dr. Gregory Tsagatakis for his valuable help. I would also like to thank Maria Argyriou, a graduate of TUC, for her valuable insight into the Vivado tools and the ZCU 102 platform. A special thanks to my colleague, Manolis Perakis, for his significant contribution regarding technical issues with the ZCU 102.

Last but not least, I would like to express my deepest gratitude to my family and friends for their support throughout my studies.

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

Biological neurons [1] typically consist of a soma (cell body), dendrites, and a single axon. There are billions of them in a human brain, and they are interconnected to each other. Their role is to receive and transmit signals from the brain. A biological neuron receives nervous impulses (input-signals) through its dendrites, processes this information in the soma, and then determines whether or not to send a neural impulse through its axon (which acts like a cable). Transmitting nervous impulses from one neuron to another is accomplished through synapses [2].

Artificial Neural Networks (ANNs) are layered organizations of interconnected artificial neurons (nodes), which mimic the function of biological neurons. They are mathematical models capable of learning and are used most commonly for classification problems. ANNs are typically inspired by the way neurons in the visual cortex function. The biological inspiration behind ANNs (and Deep Learning) is thus evident.

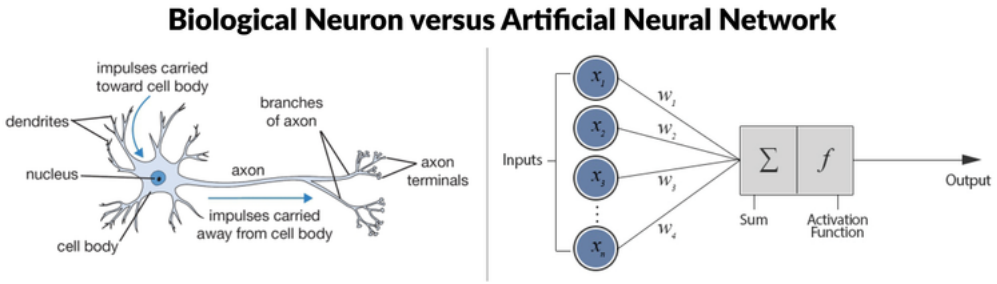

FIGURE 1.1: Biological Neuron versus Artificial Neural Network - https://www.datacamp.com/tutorial/deep-learning-python.

Typical ANNs have been widely and successfully used in multiple machine learning demanding tasks such as computer vision, speech recognition, autonomous driving, etc. However, their considerable energy requirements to achieve top performance raise serious concerns. In other words, millions of trainable parameters are demanded. Moreover, state-of-the-art Deep Learning architectures are accompanied by some substantial issues related to properties known as "transfer learning" [3] and "catastrophic forgetting" [4]. As far as the first one is concerned, they usually fail to generalize, which means transferring their expertise to new tasks without extensive retraining. The second property refers to their tendency to erase previously learned information upon learning new problems. Therefore, current Deep Learning architectures experience problems that would seem rudimentary to a human brain. Energy consumption in the brain is also extremely low. As a result, these facts strengthen our conviction that ANNs may be able to overcome the limitations mentioned above by increasing their bio-inspiration. This belief was expressed by S. Chavlis and P. Poirazi in their opinion article [5].

1.1 Motivation

Traditional ANNs assume that sensory input originating from the eyes is transferred sequentially. In addition, neurons are assumed to receive a linear summation of the incoming synaptic inputs, with dendritic influences being ignored. Besides serving as receptors, dendrites (in biological neurons) also play a complex role in integrating and propagating incoming signals, indicating that these assumptions are incorrect. In particular, passive dendrites attenuate signals traveling to the soma based on the morphological properties of dendritic trees (blue shaded area in the figure 1.2). A variety of voltage-gated channels ('active' mechanisms that are depicted in the red shaded area of the figure 1.2) are also present in many dendrites, supporting the generation of regenerative events (dendritic spikes). Dendritic spikes enable nonlinear signal processing in dendrites. Multiple nearby inputs integrated into the same dendrite, for example, produce a greater response than inputs integrated into separate dendrites. In this case, voltage-gated channels are activated, resulting in the generation of dendritic spikes. In comparison to the linear summation of the inputs, dendritic spikes have a much larger amplitude, thus amplifying the signal.

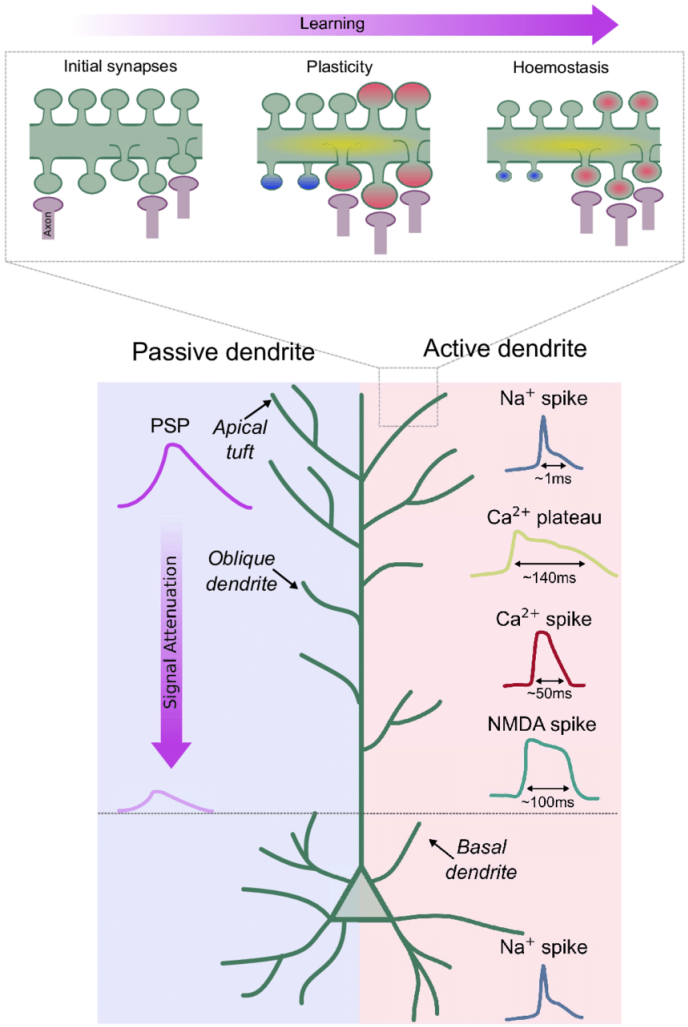

FIGURE 1.2: Schematic representation of dendritic features in biological neurons - https://www.researchgate.net/figure/Schematic-representation-of-dendritic-features-in-biological-neurons-Bottom-Schematic_fig1_352395202.

Considering the aforementioned limitations of ANNs and the effect of dendritic features on a biological neuron, it can be argued that ANNs may benefit from adopting dendritic structure and additional bio-inspired features. A comprehensive analysis of specific dendritic features is provided in the opinion article [5] mentioned above, along with their benefits and how they can be incorporated into ANNs. From this opinion article, there are several interesting points worth highlighting. Firstly, when incorporating dendrites into ANNs, it is possible to approximate dendritic spikes mathematically through activation functions. The attenuation of signals by passive

dendrites can also be described using activation functions. In essence, (artificial) neurons with dendritic structure function as two-stage or multi-stage ANNs, and dendrites serve as computing nodes. Secondly, the most substantial advancement of dendritic ANNs is that power consumption can be reduced significantly. In traditional ANNs, artificial neurons are typically represented as single nodes arranged in layers and interconnected in an all-to-all manner. In this way, millions of trainable parameters are required. On the other hand, dendritic structure provides sparsity to ANN, since each neuron is converted into a two-stage ANN (dendrites-soma). Each soma is connected to its dendrites only, and each dendrite is connected to its synapses only. Consequentially, dendritic ANNs require fewer trainable parameters than typical fully-connected ANNs, resulting in resource savings. There are some common methods used in ANNs, such as Dropout layer [6], which offers sparsity by removing certain connections randomly during training. Therefore, sparsity is generally desired in Deep Learning. In terms of performance accuracy, none of the current studies [7] [8] [9] employing dendritic ANNs has exceeded the state-of-the-art models. However, dendritic ANNs perform better than traditional ANNs when both have the same number of trainable parameters [7]. Furthermore, the dendritic location of inputs has been shown to determine the response of neurons. According to P.Poirazi et al. [10] [11], this fact enhances neurons' discrimination abilities. In addition to saving resources, sparsity also improves the network's ability to discriminate between similar input patterns [12].

An additional feature to increase bio-inspiration in ANNs is Receptive field (RF) [13], which offers structured connectivity. The RF is inspired by the human visual system, where each neuron captures a different piece of information from the field of view (the eyes' whole visible area). In Deep learning, the RF associates an output feature with an input region and it can be defined as the size of the region in the input that produces the feature. Essentially, each neuron is associated with a neighborhood of the inputs.

Backpropagation (BP) has proven to be an effective technique for training multi-layer neural networks. In this method, the parameters (weights and biases) are updated according to the network's error. It has not been proven that this method is bio-inspired and it's highly unlikely that the brain has a sub-area that calculates errors. Thus, this method cannot be regarded as bio-inspired. Even though backpropagation provides fast convergence and high performance, it consumes a considerable amount of power. This problem can

be addressed with more bio-inspired learning rules.

In ANNs and neuromorphic implementations, the aforementioned bio-inspired features are still largely unexplored. The most interesting aspect of a bio-inspired ANN is the potential power efficiency, which can be exploited by systems with limited resources (such as portable devices, mobile phones, microchips, etc.). By implementing a bio-inspired ANN architecture on a field-programmable gate array (FPGA), a more power efficient solution could be achieved. This is possible because this architecture requires significantly fewer trainable parameters than a traditional fully connected NN architecture. In hardware, parameters are associated with memory resources, which are commonly limited. Although CPU-based applications are easier to implement, they are not time or energy efficient due to their low parallelism and high power consumption. As far as Graphics Processing Units (GPUs) are concerned, they excel at parallel processing, delivering incredible acceleration when the same workload must be executed many times in rapid succession. However, GPUs tend to be highly power consuming. Compared to GPUs, FPGAs are considered to be more power efficient since only hardware functions are involved, whereas GPUs need a lot of power to facilitate software programmability and therefore they contain many gates. FPGAs are integrated circuits consisting of programmable logic blocks that can be configured to perform different logic functions. They offer a high level of parallelism too. In contrast to GPUs and Application Specific Integrated Circuits (ASICs), FPGA chips do not have hard-etched circuitry and can be reprogrammed as needed. This capability makes FPGAs an excellent alternative to ASICs, which require a long development time and a significant investment to design and fabricate. Due to their reconfigurability, FPGAs are ideal for applications where standards are constantly evolving. Moreover, FPGAs come in a variety of sizes, so designers can choose the one that is best suited to their application.

A further problem with state-of-the-art models is that they act as black boxes despite being highly efficient in many ML applications. In other words, among the set of actions taken, it is impossible to identify which one was responsible for the outcome. Therefore, increasing the bio-inspiration in the models is an attempt to resolve the credit assignment problem. The credit assignment problem, first discussed by Minsky in 1963, concerns determining which actions lead to a given result.

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 was proposed that incorporates dendritic-structure and receptive field. Regarding the learning rule, classic backpropagation is fully applied. The proposed model also included a second approach regarding learning, in which the 'Covariance rule' (plasticity rule) is applied to the first layer of the network, while backpropagation is used only at the remaining layers. The approach with the 'Covariance rule' is implemented in the thesis of Nikoletta Palatiana. 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 a first step, in order to understand and analyze 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 goal is to build an FPGA-based architecture for the aforementioned bio-inspired ANN in order 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 were able to achieve a significant improvement in 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 14.38x over CPUs (Keras) and 6.637x over GPUs (Keras).
- Our proposed FPGA implementation provides a throughput speedup of 15.548x over CPUs (Keras) and 7.143x over GPUs (Keras).
- Our proposed FPGA implementation achieves 106.15 times greater energy efficiency than the CPU (Keras) and 56.5 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 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. Additionally, we compare the results of our two FPGA-based architecture approaches.
- **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

In this chapter, we provide an introduction to Artificial Neural Networks, including an analysis of their training phase. The analysis is focused primarily on forward propagation, back propagation, and Adam optimization algorithm. In addition, a description of the activation functions is provided.

2.1 Artificial Intelligence, Machine Learning & Deep Learning

Artificial Intelligence (AI) can be defined as the ability of a machine to imitate intelligent human behavior. This intelligence is explicitly programmed with a set of if-else rules.

Machine Learning (ML) is a subset of AI that allows a system to automatically learn and improve from experience, without explicit programming. Here, the system is not rule-based, instead, it learns on its own by using multiple algorithms and techniques. In this way, the system is able to make predictions, recognizing patterns and features on the subject. In particular, the system is trained on related data-set of samples, which is provided as input. By comparing the expected outcome with its prediction (output), the system must then update the relevant sections of its structure that have contributed to output errors.

There are three main categories of ML algorithms. Supervised learning is defined by its use of labeled data-sets, which are designed to train or "supervise" algorithms into classifying data or predicting outcomes accurately. Using labeled inputs and outputs, the model can measure its accuracy and learn over time. Unsupervised learning uses ML algorithms to analyze and cluster unlabeled data-sets. These algorithms discover hidden patterns in

data without requiring human intervention, with the exception of validating the output variables. Reinforcement Learning refers to a method of rewarding desired behaviors and punishing undesired ones. In this scenario, an intelligent agent interacts with the environment and learns to act within it by trial and error.

Deep Learning (DL) is a field within ML and AI that concerns algorithms inspired by the biological structure and function of the human brain to aid machines with intelligence. This idea was prompted by the poor performance of ML in some specific cases, such as images, audio, and other unstructured data types that seem rudimentary to real brains.

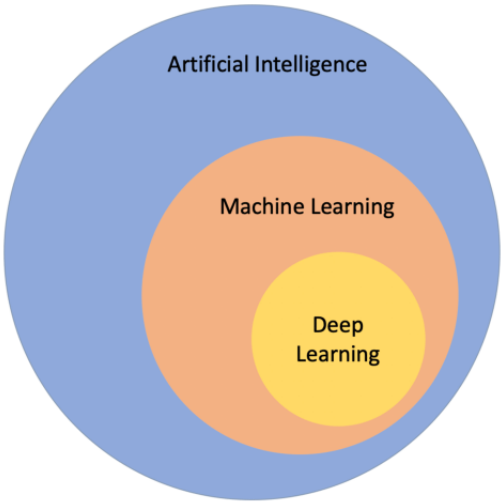

FIGURE 2.1: Artificial Intelligence, Machine Learning & Deep Learning - https://www.researchgate.net/figure/Artificial-intelligence-vs-Machine-leaning-vs-Deep-learning-Classification-and-regression_fig1_334429726.

2.2 Simple Neural Network (NN)

The basis of DL is Neural Network (NN), which is a process that mimics the way the human brain operates, with neurons that fire bits of information. NN is represented as a layered organization of interconnected artificial neurons (nodes). An artificial neuron is a mathematical function that attempts to solve a classification problem according to a specific architecture and incoming information.

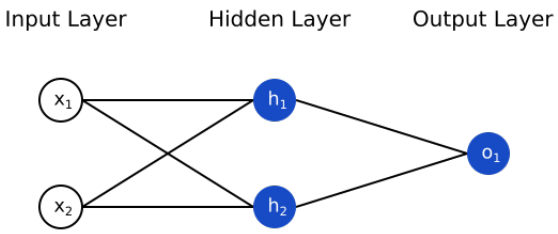

FIGURE 2.2: A Simple Neural Network - https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9.

NNs are used in a variety of applications today, including image processing (e.g. cancer detection, facial recognition), character recognition (e.g. fraud detection) and forecasting (e.g. stock market prediction, weather forecasting).

2.2.1 Classification Problem with linear boundary

For example, given a simple classification problem with a dataset of two classes on a 2D space, a NN is capable of effectively determining where to draw the boundary line to divide the data into two groups, one for each class.

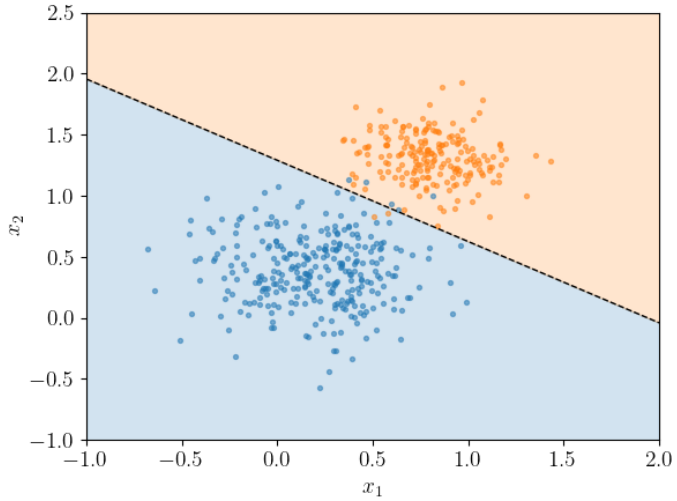

FIGURE 2.3: A Binary Classification problem - https://scipython.com/blog/plotting-the-decision-boundary-of-a-logistic-regression-model/.

This boundary line can be expressed as a linear equation:

$$w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0 \tag{2.1}$$

Vector notation can be used to abbreviate the linear equation described above as follows:

$$Wx + b = 0, \quad where\ W = (w_1, w_2) \in \mathbb{R},\ x = (x_1, x_2) \in \mathbb{R} \text{ and } b \in \mathbb{R}. \tag{2.2}$$

The input is referred to as x, the weights as W, and the bias as b.

Each input of coordinates (x_1, x_2) corresponds to a label, Y, which is what the NN tries to predict.

$$Y = label : 0 \text{ or } 1 \quad (0\text{->}blue\ \ points, 1\text{->}orange\ \ points) \tag{2.3}$$

The prediction is called $\hat{Y}$, and it is what the algorithm predicts the label will be.

$$\hat{Y} = \begin{cases} 1 & \text{if } Wx+b \geq 0 \\ 0 & \text{if } Wx+b<0 \end{cases} \tag{2.4}$$

The points above the line have $\hat{Y}$=1 and the points below the line have $\hat{Y}$=0. Moreover, the orange points have Y=1 and the blue points have Y=0. Therefore, the algorithm aims for $\hat{Y}$ to closely resemble Y. This means finding an optimal linear boundary line where the majority of the orange points lie above it and the majority of the blue points lie below it.

In the case of n columns of data, the space extends to n-dimensions. Here, the boundary is an n-1 dimensional hyperplane, which is a high dimensional equivalent of a line in 2D. This n-1 dimensional hyperplane can be expressed as an equation:

$$w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_n \cdot x_n + b = 0 \tag{2.5}$$

This equation can be abbreviated as:

$$Wx + b = 0, \quad where \quad W = (w_1, w_2, ..., w_n) \quad \text{and} \quad x = (x_1, x_2, ..., x_n). \tag{2.6}$$

Concerning the prediction, it remains the same as before.

2.2.2 Perceptron

Artificial neurons are known as perceptrons, which are the building blocks of NNs. Essentially, a perceptron is just a small graph that encodes the aforementioned equation (2.5, 2.6).

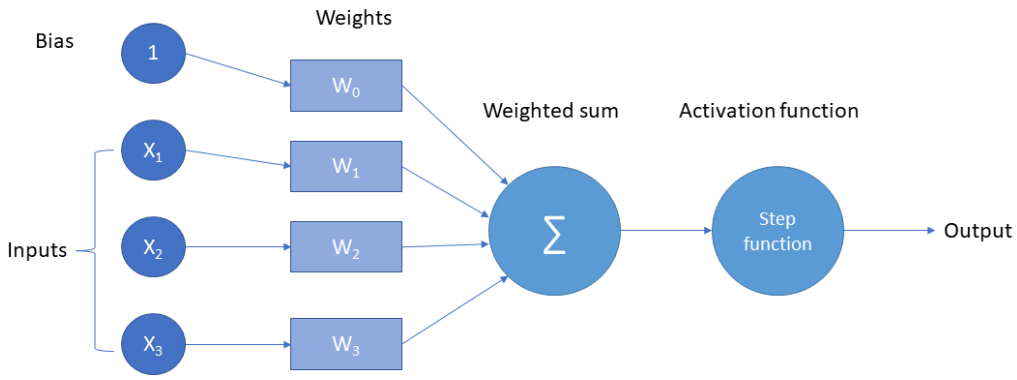

FIGURE 2.4: Perceptron - https://blog.camelot-group.com/2022/01/neural-networks-perceptron/.

In a perceptron, inputs are weighed separately, summed up, and the sum is then passed through an activation function to produce an output. Activation function decides whether a neuron can be activated or not. An in-depth analysis of the activation functions is provided in section 2.5. The prediction is obtained as a result of this process.

$$\hat{Y} = \varphi\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right) = \varphi\left(Wx + b\right), \tag{2.7}$$

where $\hat{Y}$ represents the prediction, φ represents the activation function, W represents a vector of weights, b represents the bias and x represents a vector of inputs. Prediction values are either 0 or 1, while inputs, weights, and bias are all real numbers.

Perceptron Algorithm

The perceptron algorithm attempts to correctly classify positive and negative points, by updating the weights of input signals. In the first step, the algorithm starts with random weights: $w_1,\ldots,w_n$ and b, providing an initial boundary (2.5, 2.6) that separates the data. Then, for each point with coordinates $(x_1,\ldots,x_n)$, label Y, and prediction $\hat{Y}$ (2.7):

- If the point is correctly classified, do nothing
- If the point is classified negative (Ŷ=0), but it has a positive label (Y=1):

$$\hat{w}_i = w_i + \alpha \cdot x_i, \quad \hat{b} = b + \alpha \tag{2.8}$$

- If the point is classified positive (Ŷ=1), but it has a negative label (Y=0):

$$\hat{w}_i = w_i - \alpha \cdot x_i, \quad \hat{b} = b - \alpha \tag{2.9}$$

where α is the learning rate.

In this way, the boundary moves closer to the misclassified point. This procedure is repeated until the error (of misclassification) has been minimized or until a defined number of epochs have passed. The learning rate α is a tiny number that is used to carefully classify a misclassified point, without misclassifying any other points.

2.2.3 Non-Linear Regions

In a more realistic model, the data cannot be separated by a linear function. When describing nonlinear regions, a curve should be used as a boundary. So, the perceptron algorithm needs to be redefined in a way that it can be generalized to other types of curves.

As a step in this direction, the Error function is introduced. This function simply describes the distance between the predicted outcomes and their target. The next step will be to find the direction in which the model must move in order to minimize this error (distance). By constantly taking steps to decrease the error, the problem can be solved. This method is called Gradient Descent, and it will be analyzed later. As this method involves tiny steps, the Error function cannot be discrete, it should be continuous. In this way, the model is able to detect even very small variations in error and determine in which direction it can most effectively decrease it.

In order to construct a continuous Error function, a penalty must be assigned to each point. When a point is misclassified, the penalty is approximately equal to the distance from the boundary, whereas when a point is correctly classified, the penalty is almost zero. The total error is obtained by adding the errors from all the points. So, making very small changes to the line

parameters causes very small changes in the error function, which results in a smaller total error.

Additionally, discrete predictions must be replaced by continuous predictions. Hence, the prediction will be regarded as a probability that depends on the distance from the line. Previously, the model consisted of a positive and a negative region, but now it encompasses the entire probability space. This probability space is obtained by combining the linear function (2.5, 2.6) with a nonlinear activation function (such as sigmoid).

2.2.4 Error Function (Cross-Entropy)

The Maximum Likelihood method picks the model that gives the existing labels the highest probability. Since the labels are considered independent events, the probability for a whole arrangement is the product of the probabilities of all the points based on the label they actually are. Thus, the goal is to maximize the probability.

In order to avoid product, the above concept is converted to summation by taking the negatives of the logarithms of these probabilities. The negative logarithm of the probability based on the point's actual label is a measure of error at that point. Correctly classified points will have small errors, whereas those classified incorrectly will have large errors. This concept is called Cross-Entropy. The total Cross-Entropy can be calculated by summarizing each point's Cross-Entropy, and it is expressed by the following formula:

$$Cross\_Entropy = -\sum_{i=1}^{m} \left(y_i \cdot \ln(p_i) + (1 - y_i) \cdot \ln(1 - p_i)\right) \tag{2.10}$$

For more than two classes, the formula of Multi-Class Cross-Entropy is as follows:

$$Multi - Class\ Cross\_Entropy = -\sum_{i=1}^{n}\sum_{j=1}^{m} \left(y_{ij} \cdot \ln(p_{ij})\right), \tag{2.11}$$

where m is the number of classes.

As Y_{ij} can be either 0 or 1, this formula ensures that only the negative logarithms of the probabilities of the events that have actually occurred are added.

By convention, the Error function is defined as the average of the error functions (Cross-Entropy) from all points rather than as a sum. Additionally, in NN, the probability of a point being on a given label is actually the prediction. As a result, Error function for binary classification problems is expressed as follows:

$$Error\_Function = -\frac{1}{m}\sum_{i=1}^{m}((1-y_i)\cdot\ln(1-\hat{y}_i)+y_i\cdot\ln(\hat{y}_i)) \tag{2.12}$$

Due to the prediction's equation $\hat{Y}$ (2.7), the formula can be expressed as follows:

$$E(W,b) = -\frac{1}{m}\sum_{i=1}^{m}\Big((1-y_i)\cdot\ln(1-\varphi(Wx^i+b))+y_i\cdot\ln(\varphi(Wx^i+b))\Big), \tag{2.13}$$

where Y_i is the label of the point x_i.

In a multi-class classification problem, the Error function is given by the multi-class Cross-Entropy.

$$Error\_Function = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n}\left(y_{ij}\cdot\ln(\hat{Y}_{ij})\right) \tag{2.14}$$

Therefore, the objective is to minimize Error function.

2.2.5 Gradient Descent

As mentioned previously, this algorithm aims to minimize the error by constantly taking tiny steps in the right direction. By considering the Error Function as E, the objective is to calculate the gradient of E at a point x=(x_1,...,x_n). The gradient of E is, in fact, the vector formed by the partial derivatives of E with respect to the weights and bias.

$$\nabla E = \left(\frac{\partial}{\partial w_1}E, ..., \frac{\partial}{\partial w_n}E, \frac{\partial}{\partial b}E\right) \tag{2.15}$$

The negative of the gradient actually indicates the direction in which the E can be decreased the most.

The Gradient Descent algorithm is similar to the perceptron one (2.2.2). It begins with a random weight set: w_1,...,w_n and b, which produces a boundary line (2.5, 2.6) and a probability function (2.7). For each point with coordinates

$(x_1,...,x_n)$, the error is calculated and then the weights and bias are updated as follows:

$$\hat{w_i} = w_i - \alpha \cdot \frac{\partial}{\partial w_i} E, \quad \hat{b} = b - \alpha \cdot \frac{\partial}{\partial b} E, \tag{2.16}$$

where $\frac{\partial}{\partial w_i} E$ *and* $\frac{\partial}{\partial b} E$ are the partial derivatives of E with respect to w_i and b correspondingly and α is the learning rate.

This procedure is repeated until the error is small or until a defined number of epochs (iterations) have passed. In contrast to the Perceptron algorithm (2.2.2), Gradient Descent accepts any value between 0 and 1 for $\hat{Y}$. Aside from moving closer to misclassified points, the boundary here also moves further away from correctly classified points.

2.3 Deep Neural Network (DNN)

Neural Networks combine linear models (perceptrons) into nonlinear models. Mathematically, each linear model provides a probability for each point. Each point's probabilities are weighed and added up, along with a bias. This result is then converted into a probability by applying an activation function (such as sigmoid). This approach can be used to solve nonlinear problems. A NN generally consists of the following layers. The input layer contains the data that is fed into the model. One or more hidden layers then perform the necessary computations using inputs from the input layer to produce results. In the output layer, the results from the previous hidden layer lead to a final prediction. Layers consist of nodes (perceptrons), and each node is connected to all the nodes in the next layer by a weight. The complexity of NN can be increased by adding more (hidden) layers and more nodes to each

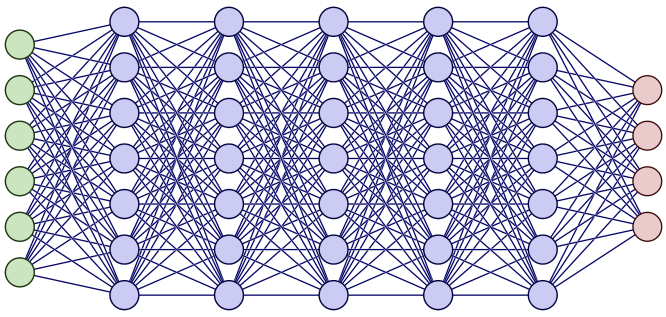

FIGURE 2.5: Example of a dense Deep Neural Network (DNN) - https://tikz.net/neural_networks/.

layer. Technically, an improvised NN with multiple hidden layers is a Deep Neural Network (DNN).

A NN model undergoes a training phase, during which it learns to accomplish a particular task and an inference (testing) phase, during which it performs the task it was trained for. The goal of training is to determine which parameters (Weights, Bias) should be applied among the nodes to accurately model the input data. This phase consists of two important stages: FeedForward and Backpropagation.

2.3.1 Feedforward (Full Forward)

The Feedforward process is how NNs produce output from input. This process is used to calculate the prediction of a perceptron (2.7). In more complicated NNs with more perceptrons (nodes) and layers, the output (prediction) is obtained by following the same principle.

Assuming that X_i=[X_1,X_2,...,X_n] represents the inputs (of the input layer), w_{ij}^1 represents the weight of node j (j=1,2,...,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:

$$Y_j^1 = \varphi(\sum_{i=1}^{n}(w_{ij}^1 \cdot X_i + b_j^1)) \tag{2.17}$$

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

$$Y_j^h = \varphi(\sum_{k=1}^{m}(w_{kj}^h \cdot Y_k^{h-1} + b_j^h)), \tag{2.18}$$

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_{kj}^h represents the weight associated between nodes k and j.

2.3.2 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 get a better model. The Backpropagation process is based on the Gradient Descent algorithm (2.2.5), 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 gradually decrease the error. The process is the same for multilayer perceptrons with the exception that the error function is more complex.

Gradients are computed using a technique known as 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:

$$\frac{\partial E}{\partial w_{k_j}^h} = \frac{\partial E}{\partial Z_j^h} \cdot \frac{\partial Z_j^h}{\partial w_{k_j}^h}, \tag{2.19}$$

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:

$$Z_j^h = \sum_{k=1}^{m} (w_{k_j}^h \cdot Y_k^{h-1} + b_j^h), \tag{2.20}$$

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 2.20, $\frac{\partial Z_j^h}{\partial w_{k_j}^h}$ is computed as follows:

$$\frac{\partial Z_j^h}{\partial w_{k_j}^h} = Y_k^{h-1} \tag{2.21}$$

Using equations 2.19 and 2.21, $\frac{\partial E}{\partial w_{k_j}^h}$ can be calculated as follows:

$$(2.19) \Rightarrow \frac{\partial E}{\partial w_{k_j}^h} = \frac{\partial E}{\partial Z_j^h} \cdot Y_k^{h-1} \tag{2.22}$$

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

$$\frac{\partial E}{\partial Z_j^h} = \frac{\partial E}{\partial Y_j^h} \cdot \frac{\partial Y_j^h}{\partial Z_j^h} = \frac{\partial E}{\partial Y_j^h} \cdot \hat{\varphi}(Z_j^h), \tag{2.23}$$

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 2.22 and 2.23, $\frac{\partial E}{\partial w_{kj}^h}$ can be expressed in the following way:

$$(2.22) \Rightarrow \frac{\partial E}{\partial w_{kj}^h} = \frac{\partial E}{\partial Y_j^h} \cdot \hat{\varphi}(Z_j^h) \cdot Y_k^{h-1} \tag{2.24}$$

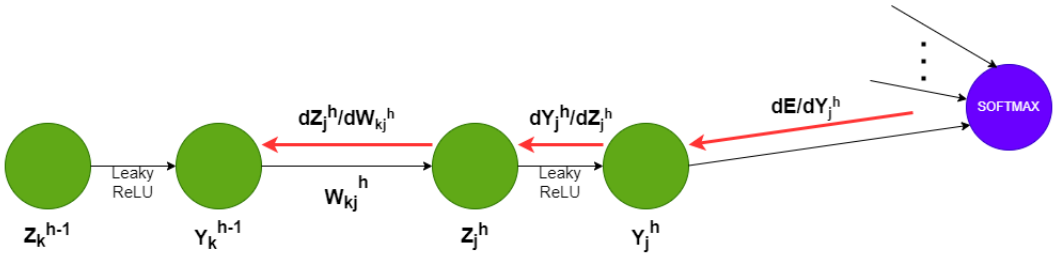

FIGURE 2.6: An illustration of Backpropagation in a NN.

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 $\frac{\partial E}{\partial b_j^h}$:

$$\frac{\partial E}{\partial b_j^h} = \frac{\partial E}{\partial Z_j^h} \cdot \frac{\partial Z_j^h}{\partial b_j^h} = \frac{\partial E}{\partial Z_j^h} \cdot 1 \tag{2.25}$$

According to the equations 2.23 and 2.25:

$$(2.25) \Rightarrow \frac{\partial E}{\partial b_j^h} = \frac{\partial E}{\partial Y_j^h} \cdot \hat{\varphi}(Z_j^h) \cdot 1 \tag{2.26}$$

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

$$\frac{\partial E}{\partial Y_j^{h-1}} = \frac{\partial E}{\partial Z_j^h} \cdot \frac{\partial Z_j^h}{\partial Y_j^{h-1}} = \frac{\partial E}{\partial Z_j^h} \cdot w_{kj}^h \tag{2.27}$$

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:

$$\frac{\partial E}{\partial Y_j^h} = -\frac{Y_i}{\hat{Y}_j^h} + \frac{1 - Y_i}{1 - \hat{Y}_j^h} \tag{2.28}$$

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

$$\frac{\partial E}{\partial Z_j^h} = \hat{Y}_j^h - Y_i \tag{2.29}$$

Update of Parameters

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

$$\hat{w_{k_j}^h} = w_{k_j}^h - \alpha \cdot \frac{\partial E}{\partial w_{k_j}^h}, \quad \hat{b_j^h} = b_j^h - \alpha \cdot \frac{\partial E}{\partial b_j^h}, \tag{2.30}$$

where α is the learning rate. As a general rule, learning rate should be a tiny number in order to make safer steps towards minimizing the error.

Underfitting and Overfitting

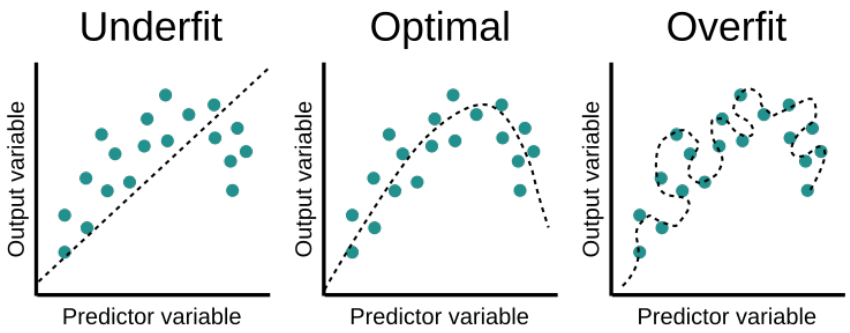

FIGURE 2.7: Underfitting and Overfitting - https://www.educative.io/edpresso/overfitting-and-underfitting.

Underfitting occurs when the model performs poorly on the training data. On the other hand, overfitting occurs when the model performs too well on the training data but fails to generalize to other data. This is a result of the model fitting too closely to the training data and learning from the noise.

Early Stopping

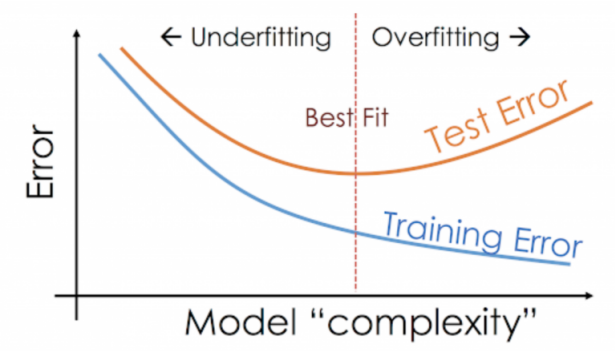

FIGURE 2.8: Early Stopping - https://www.analyticsvidhya.com/blog/2020/02/underfitting-overfitting-best-fitting-machine-learning/.

Early stopping is one of the most widely used methods for preventing overfitting. During the training phase, a small part of the training dataset is used for validation (testing during training). In this method, the training process continues until the validation error stops decreasing and starts increasing.

2.4 Optimization algorithms for updating network parameters

Although the classical Gradient Descent update rule (described in 2.30) is a powerful algorithm, it suffers from severe limitations in real-world scenarios with growing datasets. Since this method is applied to every data point in the dataset, it will become slower as the dataset grows, so the time to convergence will increase. Furthermore, as dataset grows, memory requirements increase. As a way to overcome Gradient Descent's limitations and speed up convergence in large datasets, Stochastic Gradient Descent (SGD) [14] was developed. SGD is a probabilistic approximation of Gradient Descent. Rather than calculating the gradient for the entire dataset, SGD calculates the gradient for a randomly selected subset of data.

2.4.1 Adam Optimization Algorithm

In SGD update method, a single learning rate is maintained for all parameter updates, and this learning rate remains constant during training. On

the other hand, the Adam optimization algorithm [15] computes individual adaptive learning rates for each parameter from estimates of first and second moments of the gradients. Adam combines the advantages of two popular methods: Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). AdaGrad [16] works well with sparse gradients. RMSProp (introduced by Geoffrey Hinton et al. [17]) is an optimization method closely related to Adam that works well in on-line and non-stationary settings. While RMSProp adapts parameter learning rates based on the average of first moment of the gradient (the mean), Adam uses the average of first and second moment of the gradient (the mean and the uncentered variance).

After backpropagation is completed, Adam optimization algorithm begins by getting the gradients of parameters (g_t). To accelerate the Gradient Descent algorithm, Adam calculates exponential moving averages of the gradient (m_t) and the squared gradient (v_t). The moving averages themselves are estimates of the first moment (the mean) and the second raw moment (the uncentered variance) of the gradient. By using averages, the algorithm converges towards minimum more rapidly. On a given training iteration t, the moving averages are calculated in equations 2.31 and 2.32, based on hyper parameters β_1, β_2 and gradient g_t. The β_1, $\beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages.

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \quad \text{(mean of the gradient)} \tag{2.31}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \quad \text{(uncentered variance of the gradient)} \tag{2.32}$$

Since these moving averages are initialized as vectors of 0's, they tend to be biased towards zeros, especially during the initial timesteps, and especially when the decay rates are small (i.e. β_1, $\beta_2 \approx 1$). This problem is easily counteracted by computing bias-corrected m_t (in equation 2.33) and v_t (in equation 2.34). Contrary to Adam optimization algorithm, RMSProp lacks a bias-correction term, which may lead to very large stepsizes and often divergence. This is another one difference between these two methods.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \text{(bias-corrected mean of the gradient)} \tag{2.33}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad \text{(bias-corrected uncentered variance of the gradient)} \tag{2.34}$$

The final step (2.35) concerns the update of parameters, based on the calculated bias-corrected moving averages with a step size α.

$$w_t = w_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.35}$$

For the tested machine learning problems, good default settings are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

2.5 Activation Functions

Nonlinear functions are the most commonly applied activation functions. A key purpose of an activation function is to introduce non-linearity, making the NN model capable of learning and performing more complex tasks.

2.5.1 Sigmoid / Logistic

As an input, the sigmoid activation function takes a real value, and as an output it returns a value between 0 and 1. So, it is commonly used to predict probabilities in models for binary classification problems.

$$\varphi(x) = \frac{1}{1 + e^{-x}}, \quad \text{range:(0,1)} \tag{2.36}$$

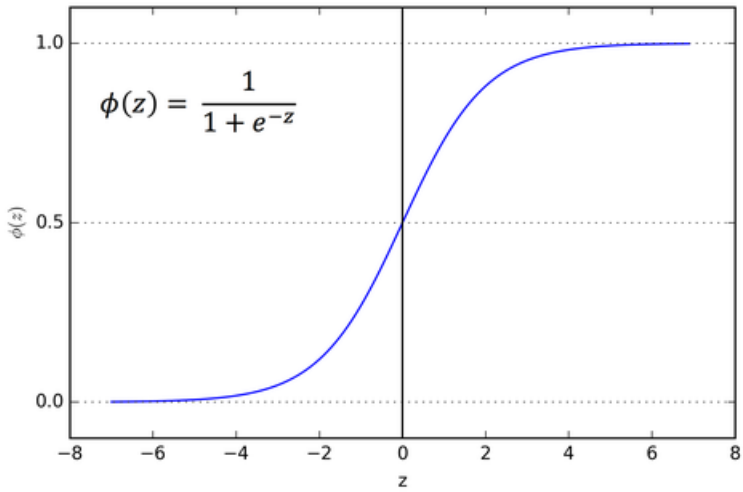

FIGURE 2.9: Sigmoid Activation Function - https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

2.5.2 Softmax

The softmax activation function [18] is a generalization of the logistic activation function and it is appropriate for multi-class classification problems, being commonly used in the output layer.

$$\varphi(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}, \quad \text{for } i = 1, ..., K \text{ and } x = (x_1, ..., x_K) \in \mathbb{R}^K. \tag{2.37}$$

2.5.3 Rectified Linear Unit (ReLU)

In DNN, ReLU is the most common activation function. It turns any negative input into zero, while it outputs any positive input as it is. ReLU's operation is similar to that of a biological neuron. It provides the NN with computational efficiency due to its simple calculation (without an exponential function) and enhanced sparsity because approximately half of the units are deactivated (having a non-positive input). Furthermore, it is less likely to suffer from vanishing gradient problem [19], since it only saturates in one direction, compared to the sigmoid activation function. This problem occurs when training an ANN using backpropagation based on gradient descent method, in cases where the gradient is too small, preventing the weight from updating its value.

$$\varphi(x) = \begin{cases} 0 & \text{for } x<0 \\ x & \text{for } x\geq 0 \end{cases}, \quad range : [0, \infty) \tag{2.38}$$

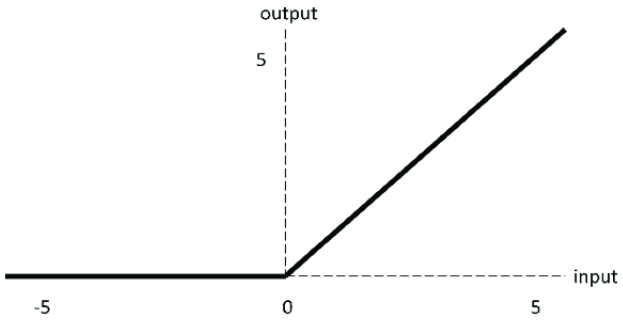

FIGURE 2.10: Rectified Linear Unit (ReLU) - https://www.researchgate.net/figure/ReLU-activation-function_fig7_333411007.

An important drawback of ReLU is when many neurons receive only negative inputs, resulting in output values of 0. Gradients will not flow in this case (will be 0), and therefore weights will not be updated. As a result, a large part of the model becomes inactive, causing the model to be less effective in training. This problem is known as “Dying ReLU problem“.

2.5.4 Leaky Rectified Linear Unit (Leaky ReLU)

Leaky ReLU, an improved version of ReLU that involves a small slope for negative values, can be used to solve "Dying ReLU problem". When the input is negative, this activation function returns a small negative number (0.01 times the input) instead of 0. In this way, it overcomes the problem of dead neurons.

$$\varphi(x) = \begin{cases} 0.01 \cdot x & \text{for } x<0 \\ x & \text{for } x \geq 0 \end{cases}, \quad range : [-\infty, \infty) \tag{2.39}$$

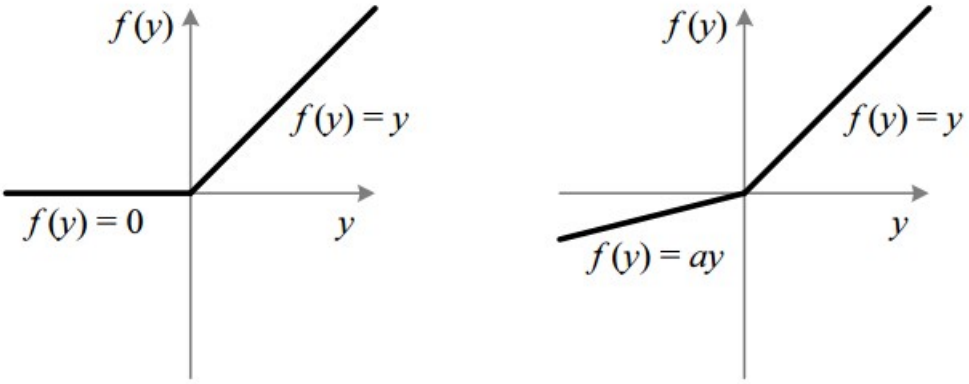

FIGURE 2.11: ReLU vs Leaky ReLU - https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

Chapter 3

Related Work

The purpose of this chapter is to discuss the related work of certain bio-inspired models and techniques, as well as our thesis methodology.

3.1 Brain-Inspired models

The growth of ANNs has allowed us to make tremendous advances in many fields. ANNs are generally derived from biology. However, they are biologically inaccurate and fail to replicate the actual mechanisms of neurons in our brain. Considering this fact along with the saturation of ANN research, it has become imperative to gain a deeper understanding of biological NNs.

3.1.1 Spiking Neural Networks

In theory, the transmission of signals from neurons to their target tissues is mediated by electrical impulses. When a neuron is electrically stimulated by others, it generates a voltage pulse at the soma which is called nerve impulse or action potential or spike. In NN, information is mainly transferred by spikes. Signals are propagated along neurons when spikes occur. As a result, spiking neural networks (SNN) [20] were developed to mimic biological NN more closely by using neurons that are more biologically realistic. In contrast to ANNs, which operate with continuous values, SNNs operate with discrete events (spikes) that occur at specific points in time. At any given time, each neuron has a value that corresponds to the electrical potential of biological neurons. A neuron's value can increase or decrease according to differential equations that represent various biological processes, including the membrane potential [21] of the neuron. When the value of a neuron exceeds a certain threshold, it spikes (fires), sending a signal to neighbouring neurons, which respond by changing their values. Then, that neuron's value

will immediately fall below its average, mimicking the refractory period of a biological neuron. According to theory, the refractory period determine the maximum frequency at which a single neuron can send action potentials. After some time, the neuron's value will gradually return to its average.

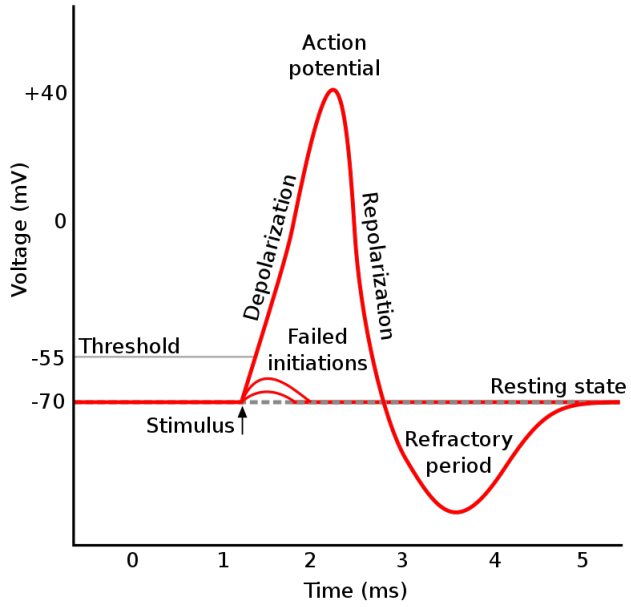

FIGURE 3.1: Plot of a typical action potential (nerve impulse) shows its various phases as the action potential passes a point on a cell membrane - https://en.wikipedia.org/wiki/Action_potential.

Several computational models have been developed using SNN, such as the Hodgkin and Huxley(HH) and the Izhikevich model. Leaky Integrate-and-Fire (LIF) is the most commonly used model due to its simplicity, and ability to model several hundreds of neurons.

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

Emmanouil Kousanakis, Apostolos Dollas et al. [22] presented a highly efficient FPGA-based architecture for a hybrid LIF model, which is implemented as a two-layer NN with dendrites and two learning rules (BCM and homeostatic plasticity). Their model analyses in detail the level of synapses and dendrites and can model generic neuron characteristics (mechanisms) from different areas of the cerebral cortex. Furthermore, they map sparse interconnections into a well defined memory structure which can be used to stream

data from an external memory. Using external memory to feed the FPGA enables different interconnection schemes to be initialized without requiring system synthesis.

3.1.3 Bridge between ANN and SNN with spiking neural unit

Despite their success in a few specific applications, SNNs lack a general universal approach to quickly designing and training architectures that can be applied to many other situations. Thus, it is unclear how to effectively scale them up to achieve high accuracy for common machine learning tasks and to materialize the benefits of the low-power neuromorphic hardware. The work of Stanisław Woźniak, Angeliki Pantazi et al. [23] bridges ANN and SNN architectures by proposing a spiking neural unit (SNU) that incorporates the biologically inspired dynamics of a spiking neuron of the LIF type in the form of a novel ANN unit. The SNU may operate in the discrete time domain as SNN, using a step function activation, or in the non-spiking ANN regime (soft variant of SNU), using continuous activations. In this way, SNU combines ANN acceleration approaches (that can be scaled to deep architectures) with energy-efficient neuromorphic SNN. In comparison to state-of-the-art ANNs, SNU-based networks perform competitively or better regarding training. Additionally, they propose an in-memory hardware implementation with in-the-loop training methodology using the SNU concept and backpropagation through time (BPTT) [24]. This implementation enables a highly efficient in-memory acceleration of the synaptic operations.

3.1.4 Speech recognition using bio-inspired Neural Networks

The work of Thomas Bohnstingl, Ayush Garg et al. [25] focuses on a biological-inspired automatic speech recognition system (ASR). Their architecture is based on a deep learning recurrent neural network (RNN) [26] approach called RNN transducer (RNN-T), which employs long short-term memory (LSTM) units [27]. They introduce novel neural connectivity concepts emulating the axo-somatic and the axo-axonic synapses. In their work, the LSTM units have been replaced with aforementioned SNU variants, which are simpler and can significantly reduce computational costs and latency by 50% and 40%, respectively. Moreover, this architecture provides competitive performance compared to its LSTM-based counterpart.

3.1.5 Novel online learning algorithmic framework for Deep Neural Networks

Typically, most of the bio-inspired models are trained with the error BPTT algorithm. Although BPTT-trained networks have shown significant success, this algorithm has severe limitations. In particular, it involves offline computation of the gradients due to the need to unroll the network through time. A biological NN, on the other hand, is capable of continuously adapting through online learning (in other words, processing and learning simultaneously from a continuous input stream). In order to approach conceptually the way the brain adapts to changing environmental conditions, online learning algorithms were developed for calculating the parameters' updates in real time as the input data arrives. The first online learning algorithms [28] had higher time complexity than BPTT, which explains why they remained rarely used in practice. The work of Thomas Bohnstingl, Stanisław Woźniak et al. [29] proposes a novel online learning algorithmic framework for deep RNNs and SNNs, called online spatio-temporal learning (OSTL). In general, there are two types of gradient flows in gradient-based training of RNNs. Gradients flowing between units within the same time step, and gradients flowing between units across different time steps. Their learning methodology proposes the biologically inspired separation of these gradients into two components: spatial and temporal. OSTL enables efficient online training of deep feed-forward SNN architectures with low complexity and comparable performance to BPTT. For shallow SNNs, OSTL is gradient equivalent to BPTT. Furthermore, OSTL has been further generalized to deep RNNs comprising spiking neurons or more complex units, such as LSTMs, demonstrating competitive accuracy in comparison with BPTT.

3.1.6 Novel biologically inspired optimizer for both ANN and SNN training

As compared to mammals' neural circuits, ANNs have a significantly simplified structure and dynamics, which largely explains their limitations. In backpropagation-based ANN training, several key mechanisms are not modeled, including synaptic integration and local regulation of weight strength. Inspired by these mechanisms, Giorgia Dellaferrera, Stanisław Woźniak et al. [30] proposed GRAPES (Group Responsibility for Adjusting the Propagation of Error Signals), a novel biologically inspired optimizer for both ANN and

SNN training. GRAPES quantifies the responsibility of each node in the network, as a function of the local weight distribution within a layer. When applied to gradient-based optimization algorithms, GRAPES provides a simple and efficient way to dynamically adjust the error signal at each node and to enhance the updates of the most relevant parameters. Their approach avoids additional memory penalties and is more biologically plausible than other optimizers because they do not store parameters from previous steps. They show that this biologically inspired mechanism improves the training of fully connected neural networks (FCNNs) by systematically accelerating convergence, increasing inference accuracy, and mitigating catastrophic forgetting. In addition, convolutional neural networks (CNNs) and SNNs perform better on temporal data as a result of their approach. These results validate the hypothesis that biologically inspired ANN and SNN models demonstrate superior performance in software simulations. Moreover, this mechanism is optimally suited for dedicated hardware implementations.

3.2 Thesis Approach

The bio-inspired models are typically implemented using SNNs. Alternatively, they can be built upon the SNUs, which allows us to leverage the biologically inspired neural dynamics of the spiking neuron in deep learning. The thesis model, however, is based on a simple ANN. We aim to increase the bio-inspiration of a typical ANN by adopting dendritic-structure and receptive field. Using a dendritic-structure, the artificial neuron is divided into its soma and its dendrites. Since dendritic-structure provides sparsity to ANNs, fewer parameters are required to train them, thereby reducing power consumption. Concerning the receptive field, it is inspired by the human visual system and offers structured connectivity, indicating that each neuron is associated with a neighborhood of inputs. One of the most interesting aspects of this bio-inspired model is its power efficiency and potential application to systems with limited resources (such as portable devices, mobile phones, etc.). As a further improvement, Adam optimization algorithm (2.4.1) is used for updating training parameters instead of classical gradient descent (2.30). Adam accelerates convergence in large datasets. Moreover, by computing individual adaptive learning rates for each parameter, this algorithm tends to be more bio-inspired, as dendrites in biological neurons support the generation of their own regenerative events (dendritic spikes). This thesis goal is to design and implement an FPGA-based architecture for this

bio-inspired ANN that speeds up training and reduces power/energy consumption further by exploiting the FPGAs' ability of high parallelism and power efficiency.

Chapter 4

System Modeling

In this chapter, thesis neuro-inspired ANN model will be explained extensively, providing a description of its architecture, its functionality and its connectivity structure. This model was introduced and implemented in Keras by the Poirazi lab of IMBB-FORTH. The focus of this implementation is the training process for this bio-inspired ANN model. The dataset used was MNIST. There were, however, some ambiguities in the high-level Keras' implementation due to its abstract nature. Therefore, a lower-level implementation was required in order to analyze and understand this model in greater detail. For this purpose, a Numpy implementation of this model was developed in this thesis. In this chapter, we will describe in detail the Numpy implementation as well as the basic procedures involved in training this model. Finally, profiling Numpy's implementation will allow us to examine, in terms of execution time and memory, the functions of the training process in order to identify those routines that consume disproportionately large amounts of time or memory.

4.1 Neuro-inspired ANN model

A neuro-inspired ANN model was introduced by the Postdoctoral Researcher S. Chavlis and the Research Director P. Poirazi, both from the Poirazi lab of IMBB-FORTH. Certain bio-inspired features are incorporated into the proposed architecture, such as the dendritic-structure and the receptive field (RF). By increasing the bio-inspiration in a typical ANN, we aim to achieve significant resource savings. Regarding the learning rule, it includes two different approaches. As far as the first strategy is concerned, classic backpropagation (2.3.2) is fully applied. In the second approach, a 'Covariance rule'

(plasticity rule) is applied to the first layer of the network, while backpropagation is used only at the remaining layers. Specifically, the presented thesis addresses the first strategy of learning rules, with the emphasis on how to integrate the bio-inspired features into a typical ANN so that the training process can be accelerated and the power consumption reduced. The approach with the 'Covariance rule' is implemented in the thesis of Nikoletta Palatiana, providing an in-depth report on it with interesting results. Furthermore, Adam optimization algorithm is selected as the method for updating the network's parameters (Weights and biases) rather than classical gradient descent. This update method computes individual adaptive learning rates for each parameter. So, it tends to be more bio-inspired since the information is encoded separately from its dendrites and not entirely from the neuron.

4.1.1 Bio-inspired Features

Dendritic-Structure

Dendrites are thin processes that extend from the cell body of neurons and are responsible for receiving signals from neighboring neurons. Additionally, dendrites possess passive properties that attenuate incoming signals and active mechanisms capable of generating dendritic spikes. Dendritic spikes enable nonlinear signal processing. Activation functions can be used to approximate these dendritic features mathematically. Thus, dendrites act as computing nodes. An artificial neuron in a typical ANN is represented as a single node, whose output (prediction 2.7) is computed as a weighted sum of all inputs (all-to-all manner) from the previous layer, followed by an activation function. In this way, millions of trainable parameters are required. As far as the presented model is concerned, it is equipped with dendritic-structure, whereby the artificial neuron is divided into the soma and its dendrites. In particular, each artificial neuron is converted to a 2-layer node structure (soma - dendrites) and produces its output only through its dendrites. Each soma is connected only to its dendrites, and each dendrite is only connected to its synapses. This feature provides sparsity to our model. As a result, fewer trainable parameters are required compared to typical fully-connected ANNs, allowing significant resource savings.

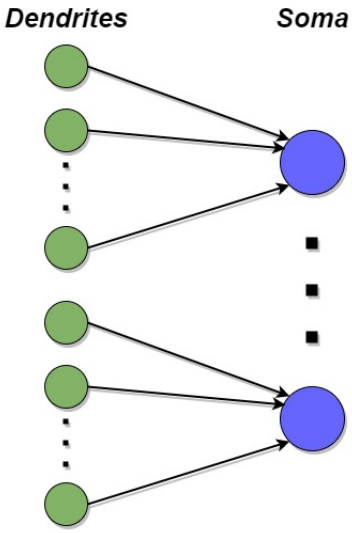

FIGURE 4.1: Dendritic-Structure Layer

Receptive Field (RF)

In the human visual system, each neuron captures a different piece of information from the field of view. A second bio-inspired feature, the so-called receptive field (RF), is derived from this fact. This feature offers structured connectivity. In particular, each soma corresponds to a neighborhood of the entire input and its dendrites receive inputs from smaller neighborhoods around the soma's one. Regarding the sampling of the input, samples are selected mostly from the center of the image, since it is most likely to contain some useful information there. In section 4.1.3, receptive field is elaborated and implemented.

4.1.2 Reference Model Architecture

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.

Among activation functions, ReLU is preferred over sigmoid (2.5.1) and others due to the computational and other factors discussed in section 2.5.3.

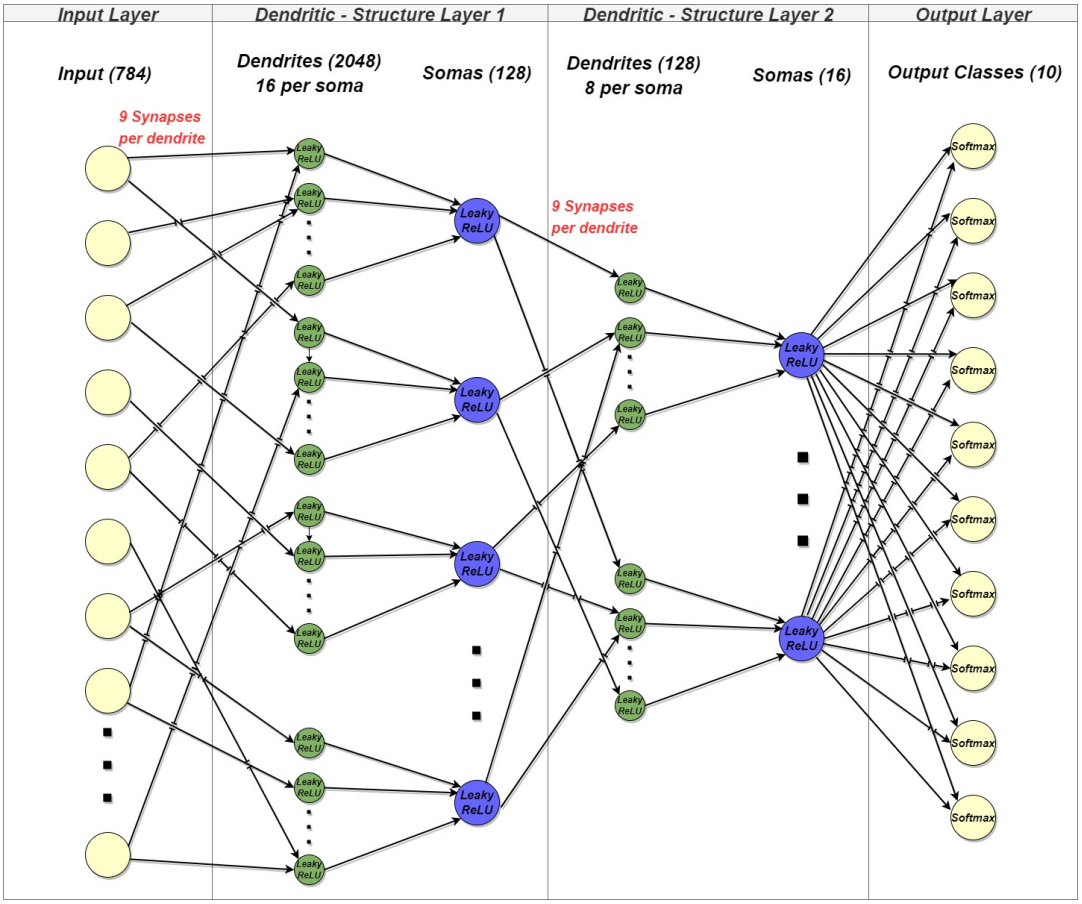

FIGURE 4.2: Model Architecture

Nevertheless, it suffers from "Dying ReLU problem" that can be solved by an improved version of it, called Leaky ReLU (2.5.4). 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 (2.5.2). The softmax layer is commonly 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 [31]. In MNIST, each input image has a size of 28x28 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 4.2.

4.1.3 Implementation of the Connectivity Structure

The aforementioned bio-inspired features provide the presented model with a sparse connectivity structure, which is implemented by utilizing masks on

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

the weights. In particular, each weight matrix is multiplied by a mask matrix. Masks are created as part of the initialization process and are assumed to remain stable throughout the entire process, while connectivity-structure changes in neuroscience.

The mask of the first layer represents the receptive field (4.1.1) and determines the connectivity between inputs and dendrites. It is constructed in three steps (nested loops). As a first step, each soma is assigned to a specific input-pixel using a semi-random allocation, in which samples from the center of the image are selected more frequently (70%) than from other areas. Each selected input-pixel serves as the center pixel of a neighborhood that refers to its corresponding soma. Then, the neighbors' values of each soma are obtained around its center pixel, seeking as many neighbors as the number of dendrites per soma (16 neighbors). Each soma's neighbor corresponds to one of its 16 dendrites and serves as the center pixel of a distinct neighborhood that refers to the synapses of that dendrite. In the following step, the neighbors' values of each dendrite are also obtained around its center pixel, in this case seeking 9 neighbors, as the number of synapses.

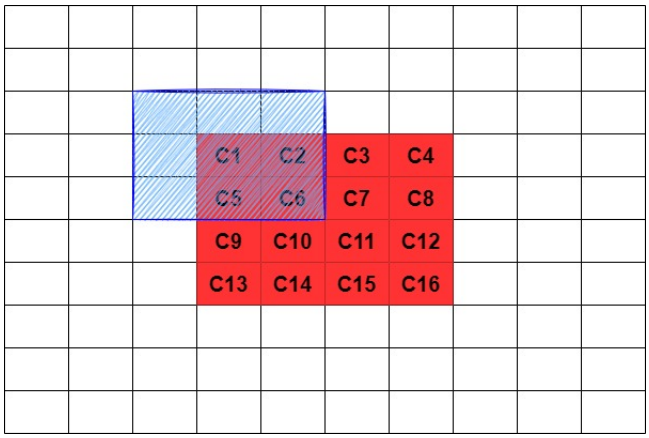

FIGURE 4.3: Receptive Field

This process is depicted in figure 4.3, showing an example taken from a section of the image (which normally has 28x28 pixels). The red pixels indicate the neighborhood of a soma. Each one of the 16 red pixels (C1,C2,...,C16) corresponds to a dendrite. For each red pixel, a blue neighborhood of 9 pixels is formed, representing the synapses of the corresponding dendrite. In the figure, the 9 blue pixels indicate synapses on dendrite C1. The same procedure is followed for all dendrites of each soma. By using this method, each dendrite is connected to 9 input-pixels.

In the second layer, the mask forms dendritic to somatic connections by connecting each soma with 16 consecutive dendrites. The following mask (of the third layer) indicates the transition from the first dendritic-structure layer to the second one, connecting each dendrite randomly with 9 somas of the previous layer. The mask for the fourth layer works similarly to the mask for the second layer, except that each soma is connected to 8 dendrites. Finally, a mask is not required at the fifth layer.

4.2 Software Implementations - Tools used (Keras - Numpy)

The software implementations include both the training and inference (or testing) procedures, along with the necessary initialization. The training procedure (figure 4.4) is divided into three main stages: Full-Forward propagation (Feed-Forward - 2.3.1), Backpropagation (2.3.2), and parameter updating (Adam Algorithm 2.4.1). The inference procedure consists only of Full-Forward propagation.

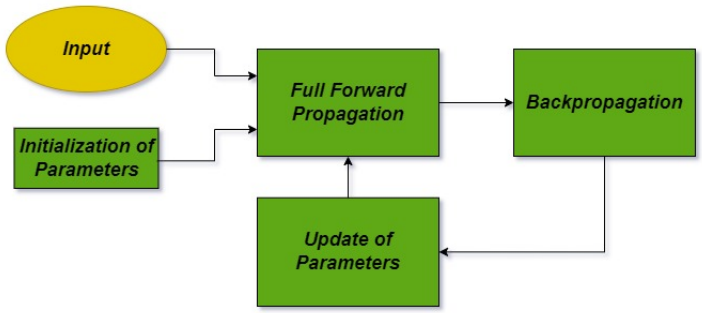

FIGURE 4.4: The basic steps of training procedure

The model was first implemented by S. Chavlis in Keras. Keras [32] 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 [33] is a Python library for handling large, multidimensional 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 experiment quickly 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 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 (2.5.4) 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.
- Early Stopping Callback (applied - patience=10 epochs): To stop training when a monitored metric (validation error) has stopped improving. It is explained in 2.3.2.
- Adam Optimizer (enabled): Adam optimization algorithm is explained in section 2.4.1.
- Error function (Multi-class Cross entropy 2.14): 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 [31] is used for training and testing. It includes:

- Training set of 60000 examples. According to validation split hyperparameter (20%), 48000 examples are used for training and 12000 for validation.
- Test set of 10000 examples.
- Images (Inputs) of 28x28 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 (see section 2.3.2). These specific callbacks are triggered when validation error stops improving for a certain number of epochs, thereby reducing

the learning rate or stopping the training earlier. Callbacks in Numpy, along with the validation process, have been implemented by Nikoletta Palatiana.

Data Type

In the Keras implementation developed by the Poirazi lab, the data type used is float. 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 its training process rather than to achieve optimizations. In the next chapter of FPGA design and implementation (in section 5.6.4), we will discuss in detail our exploration to determine the most appropriate data type to trade off precision of results with resource utilization.

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. 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 (4.1.3). 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 Definition of Numpy mathematical functions

The calculations in numpy implementation were performed using a few functions, which are explained below.

- **numpy.dot(A,B)** [34]: It is important to note that in our case, both A and B are 2D arrays, so this function performs matrix multiplication on the matrices A and B.

- **numpy.multiply(A,B)** [35]: It is used to compute the element-wise multiplication of the arrays A and B.
- **numpy.sum(A, axis=0, keepdims=True)** [36]: It returns the sum of array elements over a given axis. The axis=0 in our case refers to the the rows (first dimension). So, this function will sum all numbers along the rows. When keepdims is selected (True), the axis 0 (in this specific case), which would normally be 'lost', is kept as a dimension of size one.
- **numpy.ndarray.T** [37]: T attribute is used to return the transpose of a given array. By transposing a matrix, the row becomes the column and vice versa.

4.3.2 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 [38], 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) [39] 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). Afterwards, 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 (4.1), each weight matrix is filtered (with its corresponding mask) and stored as follows:

$$parameters["W" + str(layer\_id)] = np.multiply(parameters["W" + str(layer\_id)], Mask) \tag{4.1}$$

4.3.3 Full-Forward propagation

As described in Section 2.3.1, Full Forward Propagation is used to infer the outcome in NN models. Each layer's output is calculated sequentially to determine the network's outcome. A layer's output is derived from equations 2.17 and 2.18. 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:

$$Z\_curr = numpy.dot(Y\_prev, W\_curr) + b\_curr \tag{4.2}$$

$$Y\_curr = activation\_function(Z\_curr), \tag{4.3}$$

where numpy.dot (explained in 4.3.1) performs matrix multiplication on matrices W_curr and Y_prev. Y_prev corresponds to the input of the network in the first layer.

The Full Forward propagation algorithm (17) executes the equations 4.2 and 4.3 for each single layer. These equations provide the layer's output before and after the activation function (Z_curr and Y_curr respectively). With the exception of the first layer, which is fed by the network's input from dataset, each layer is fed by the output Y_prev ('activated' output) of its preceding layer. Before these equations can be performed in each layer of Full Forward, the corresponding activation function, weight matrix and bias vector must be loaded. The architecture list and the parameters list are used to retrieve these, respectively. Full Forward returns a list called forward_outputs containing each layer's outputs. Figure 4.5 illustrates this algorithm.

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]

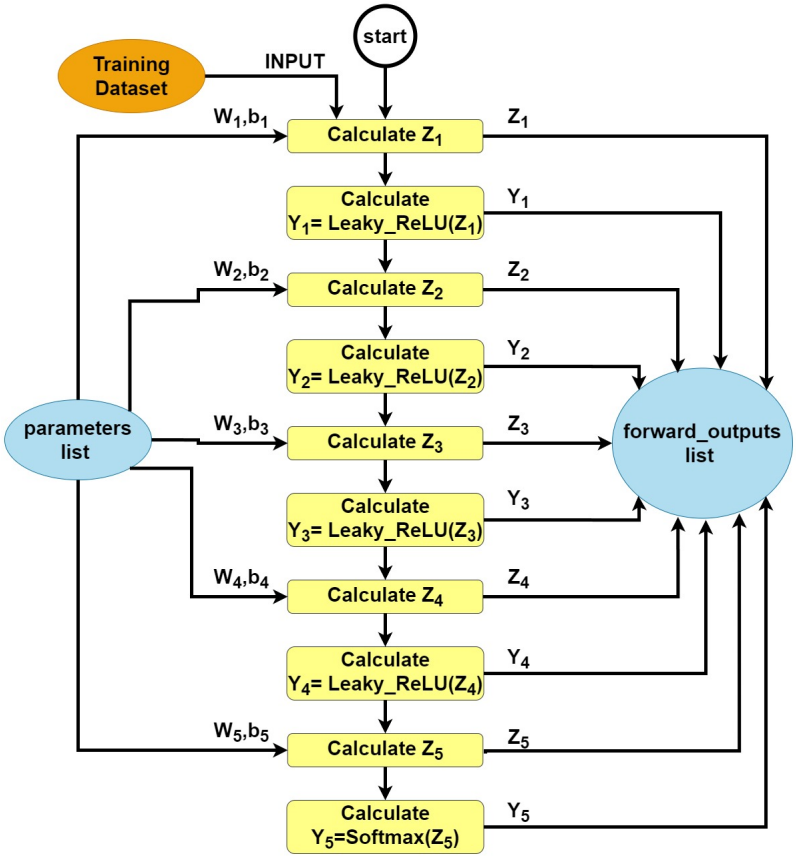

FIGURE 4.5: Full-Forward propagation in the model

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

4.3.4 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 2.3.2. 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 2.29, skipping the calculation of $\frac{\partial E}{\partial Y}$. Using Numpy, it can be calculated as shown in equation 4.7. 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 2.22, which is based on the chain rule method. The chain rule method is illustrated in Figure 2.6 for a better understanding of backpropagation. In Numpy, this equation is expressed as follows (4.4):

$$dW\_curr = numpy.dot(Y\_prev.T, dZ\_curr) / Batch\_Size\,, \tag{4.4}$$

where numpy.dot function and attribute T are described in section 4.3.1. 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 2.25. Numpy implements this equation as follows (4.5):

$$db\_curr = numpy.sum(dZ\_curr, axis = 0, keepdims = True) / Batch\_Size\,, \tag{4.5}$$

where numpy.sum function is explained in section 4.3.1.

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 2.27. This equation is implemented as follows in Numpy (4.6):

$$dY\_prev = numpy.dot(dZ\_curr, W\_curr.T)\,, \tag{4.6}$$

where numpy.dot function and attribute T are described in section 4.3.1.

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

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]

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 2.23. 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 (4.7):

$$dZ\_curr = \begin{cases} Y\_curr - Input\_Label & \text{for last (softmax) layer} \\ backward\_Leaky\_ReLU(dY\_curr, Z\_curr) & \text{for the other layers} \end{cases} \tag{4.7}$$

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 4.7). After each layer's calculations, the gradients for weights (dW_curr) and bias (db_curr) are stored in gradients list. To maintain the sparse connection structure, dW_curr is filtered with the corresponding mask (from the architecture list) before being stored. In Figure 4.6, we present an overview of the backpropagation process for each layer of our bio-inspired NN.

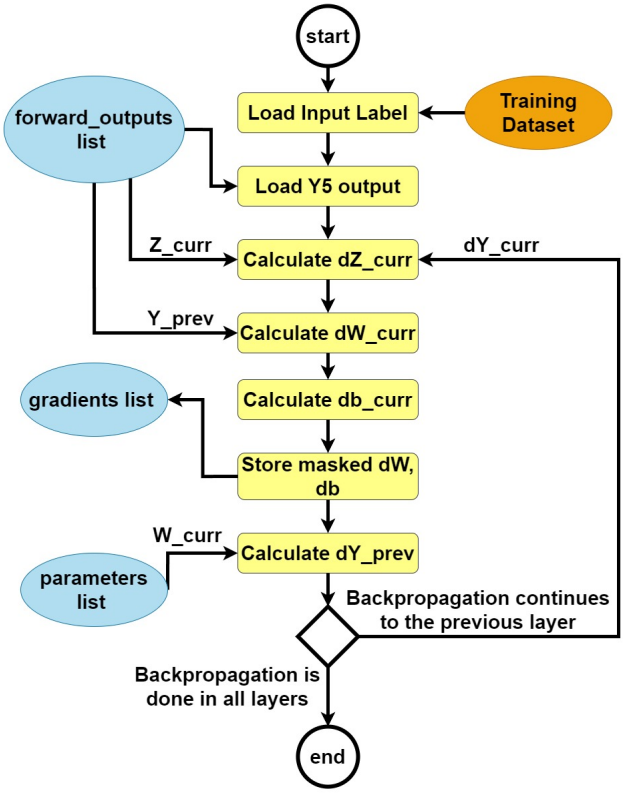

FIGURE 4.6: An overview of the backpropagation process of our bio-inspired NN.

4.3.5 Update method - Adam Algorithm

As discussed in section 2.4, 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 (2.4.1) is chosen as the method for updating network parameters (weights and biases) iterative based in training data. In this method, individual adaptive learning rates are computed for each parameter from estimates of first and second moments of the gradients. Rather than encoding information entirely from the neuron, it is encoded separately from its dendrites. As

Algorithm 2 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

a result, the Adam algorithm tends to be more bio-inspired than the methods noted above.

A detailed explanation of Adam Optimization algorithm's methodology is provided in section 2.4.1. Figure 4.7 illustrates the overview of Adam Optimization Algorithm process for a layer of our bio-inspired NN. 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 backpropagation algorithm produces this list. On the basis of equation 2.31, 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 based on equation 2.32. 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 based on equations 2.33 and 2.34. These equations 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 (as shown in equation 2.35). The dimensions of the necessary matrices are given in table 4.4.

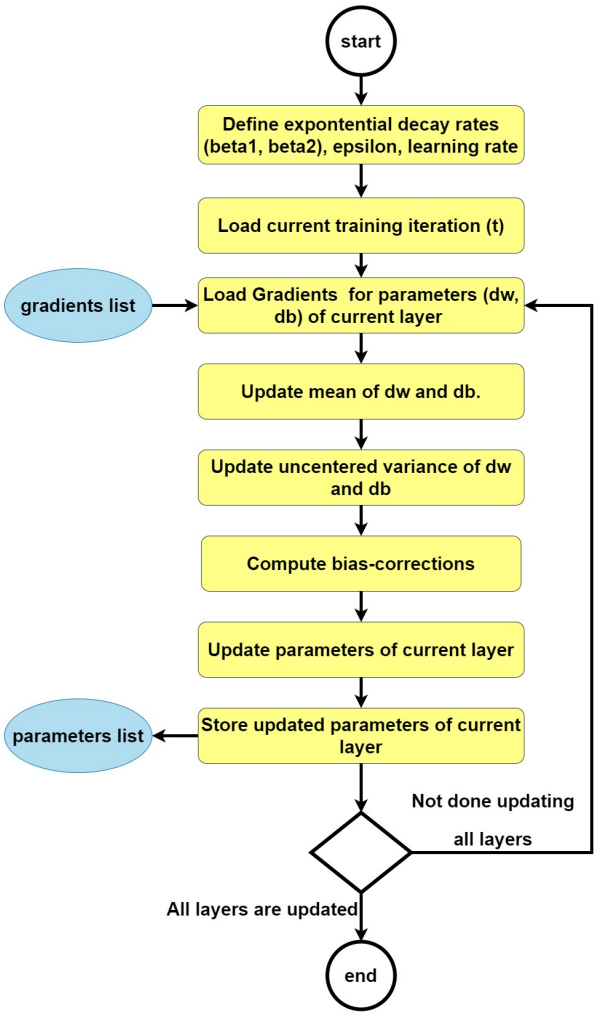

FIGURE 4.7: An overview of the Adam Optimization Algorithm process for each layer of our bio-inspired NN.

Algorithm 3 Adam Optimization Algorithm - Update Algorithm

procedure ADAM_UPDATE(*Input_Value*, *parameters*, *architecture*, *gradients*, *adam_values*, *learning_rate*, *t*)
    beta_1 ← 0.9, *beta_2* ← 0.999, *epsilon* ← $1e-8$
    for *layer_id* ← 1 to 5 **do**
        ▷ load dw and db of current layer
        dw ← *gradients*["*dW*" + *str*(*layer_id*)]
        db ← *gradients*["*db*" + *str*(*layer_id*)]

        ▷ update mean of the gradient of weights
        mean_dw_prev ← *adam_values*["*mean_dw*"+*str*(*layer_id*)]
        adam_values["*mean_dw*"+*str*(*layer_id*)] ←
                *beta_1***mean_dw_prev*+(1-*beta_1*)**dw*
        ▷ update mean of the gradient of biases
        mean_db_prev ← *adam_values*["*mean_db*"+*str*(*layer_id*)]
        adam_values["*mean_db*"+*str*(*layer_id*)] ←
                *beta_1***mean_db_prev*+(1-*beta_1*)**db*

        ▷ update uncentered variance of the gradient of weights
        uvar_dw_prev ← *adam_values*["*uvar_dw*"+*str*(*layer_id*)]
        uvar_dw_curr ← *beta_2***uvar_dw_prev*+$(1-beta\_2)$*$(dw**2)$
        adam_values["*uvar_dw*"+*str*(*layer_id*)] ←
                np.maximum(*uvar_dw_prev*, *uvar_dw_curr*)
        ▷ update uncentered variance of the gradient of biases
        uvar_db_prev ← *adam_values*["*uvar_db*"+*str*(*layer_id*)]
        uvar_db_curr ← *beta_2***uvar_db_prev*+$(1-beta\_2)$*$(db**2)$
        adam_values["*uvar_db*"+*str*(*layer_id*)] ←
                np.maximum(*uvar_db_prev*, *uvar_db_curr*)

        ▷ compute bias-corrections
        mean_dw_c ← *adam_values*["*mean_dw*"+*str*(*layer_id*)]/
                (1-*beta_1****t*)
        mean_db_c ← *adam_values*["*mean_db*"+*str*(*layer_id*)]/
                (1-*beta_1****t*)
        uvar_dw_c ← *adam_values*["*uvar_dw*"+*str*(*layer_id*)]/
                (1-*beta_2****t*)
        uvar_db_c ← *adam_values*["*uvar_db*"+*str*(*layer_id*)]/
                (1-*beta_2****t*)

        ▷ update weights and biases
        mask ← *architecture*["*mask*" + *str*(*layer_id*)]
        dw_f ← *np.multiply*(*learning_rate**(*mean_dw_c*/(*np.sqrt*(*uvar_dw_c*)+*epsilon*)),
                mask)
        db_f ← *learning_rate**(*mean_dw_c*/(*np.sqrt*(*uvar_dw_c*)+*epsilon*))
        parameters["*W*"+*str*(*layer_id*)] ←
                parameters["*W*"+*str*(*layer_id*)]-*dw_f*
        parameters["*b*"+*str*(*layer_id*)] ←
                parameters["*b*"+*str*(*layer_id*)]-*db_f*
    end for
    return *parameters* ▷ return updated parameters
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]

4.4 Profiling

The process of profiling involves analyzing the function calls, the execution duration of functions, the usage of particular instructions and the memory. These program parameters are measured while it is running. Profiling assists engineers in identifying the routines that consume a disproportionate amount of time or memory and optimizing them. As shown in figure 4.8, most of the training time (approximately 72%) is consumed by the Adam Optimization Algorithm.

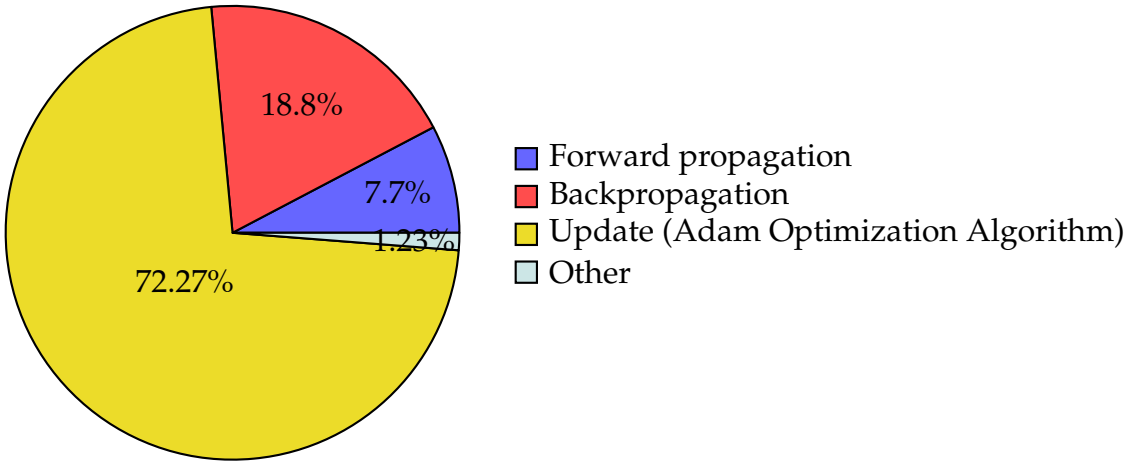

FIGURE 4.8: Analysis of how Numpy implementation consumes training time.

Figure 4.9 illustrates the analysis of the time spent calling each individual process of Adam. Numpy's Adam algorithm is presented in 37.

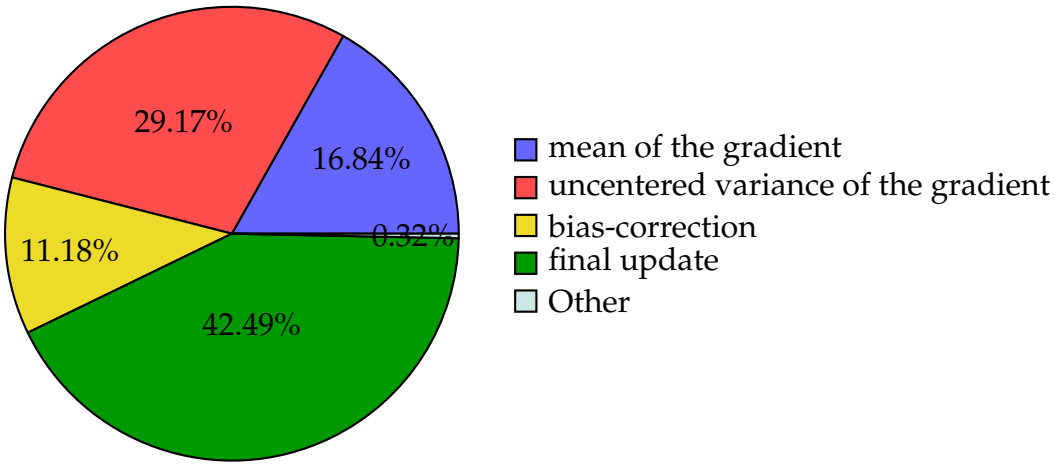

FIGURE 4.9: An analysis of the impact of the Adam Optimization Algorithm's individual (inner) functions.

It is estimated that 42.5% of Adam's time is spent on the final update of parameters. This can be explained by the fact that the final update step involves divisions, square roots and element-wise multiplications of matrices. The next most time-consuming step in the Adam process is the computation of the uncentered variance of the gradient, which takes 29% of the time. This occurs due to the exponentiation calculations of the dW and db matrices. It is therefore the complexity of Adam's calculations which makes it the most time-consuming part of the training process.

The analysis of the backpropagation can be found in figure 4.10. Algorithm 28 presents the Numpy implementation of backpropagation.

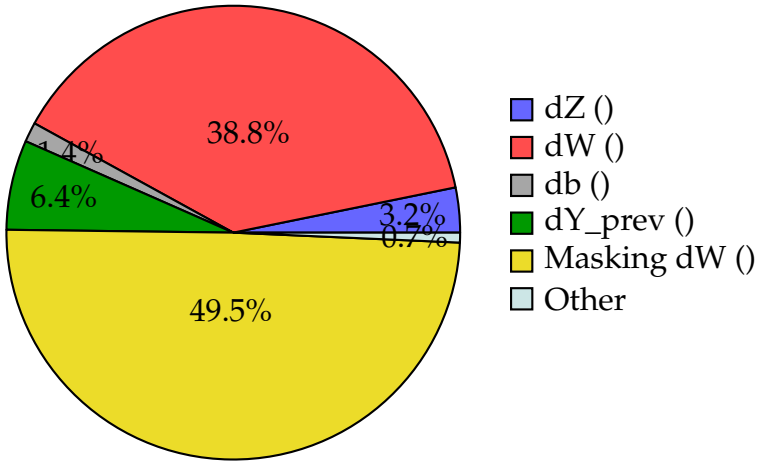

FIGURE 4.10: The time spent calculating each individual backpropagation term is analyzed.

Approximately half of the time during backpropagation is spent masking dW. In this step, the matrices (dW and Mask) are multiplied element-wise per layer. As the second most time-consuming part, dW calculation involves matrix multiplication per layer. Thus, most of the backpropagation time is consumed by the multiplication of matrices (element-wise or not).

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

The Amdahl's Law [40] 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:

$$S = \frac{1}{1 - P}, \tag{4.8}$$

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 forward propagation, backpropagation and Adam algorithm processes together is calculated as follows:

$$S = \frac{1}{1-(72.27\%+18.8\%+7.7\%)} = \frac{1}{0.0123} = 81.3x$$

Therefore, the optimization of only these three processes can result in a maximum theoretical speed-up of 81.3 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 Design and Implementation

In this chapter, the FPGA-based architecture for the training process of the bio-inspired ANN will be designed, implemented using Vivado tools and downloaded onto the Xilinx ZCU 102 evaluation board. It is described how each training process is designed and implemented and how parallelization is achieved at the level of computation. Furthermore, this chapter provides information about the Vivado tools used for implementing our design, the ZCU 102 platform, the AXI4 Interface Protocol, the PL-PS communication methods, and the memory configuration.

5.1 FPGA Design

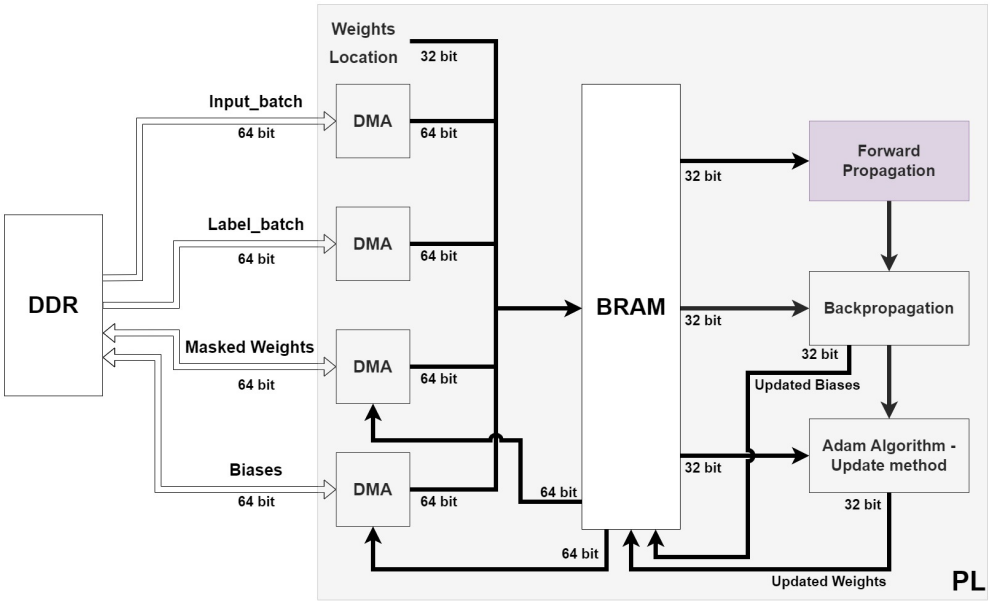

FIGURE 5.1: FPGA Design - Architecture. The Forward Propagation block was designed and implemented by Nikoletta Palatianna in her thesis.

The design of the FPGA-based architecture for the training process of the bio-inspired ANN is shown in figure 5.1. The Forward Propagation block is designed and implemented by Nikoletta Palatianna in her thesis. In this thesis, the backpropagation and Adam Optimization algorithm blocks are designed and implemented. The overall FPGA-based architecture of the bio-inspired ANN (fig. 5.1) is also part of this thesis.

We use the high-speed streaming protocol (explained in sections 5.3.1, 5.3.2) for inputs, labels, weights, and biases, which are data that must be continuously (in each training iteration) transferred between DDR and PL. However, we do not use these data in the order in which they are passed in the PL. As a result, they must be stored in BRAMs. According to figure 5.1, BRAM appears to be a single large block, but in reality, each type of data is stored in its own BRAM. In addition, these BRAMs are partitioned into smaller BRAMs so that they can provide more memory accesses per clock cycle and take advantage of parallelism. Figure 5.19 illustrates in detail the contents and partitions of BRAMs. As forward propagation, backpropagation, and Adam algorithm processes are executed sequentially, these (multiple) BRAMs can be shared between them without causing problems. In our approach, we exploit parallelism within each process block and specifically within each process block's layer. Section 5.4.1 provides more information about PL-PS communication and memory configuration. The implementation of this architecture using Vivado will be discussed in detail in section 5.4. Sections 5.5, 5.6 and 5.7 will analyze the implementation of Training IP, which includes forward propagation, backpropagation, and the Adam Algorithm (update). As the data type, single-precision floating point is used. In section 5.6.4, we will analyze this decision.

5.1.1 Sparse Connectivity and Weight Handling

As discussed in section 4.1.1, the dendritic structure and receptive field features provide sparsity to our model. In the aforementioned software implementations (4.3), sparse connectivity was achieved by masking the weight matrices of the network appropriately (explained in 4.1.3). As far as our FPGA design is concerned, the weight matrices that are passed in have already been masked. Section 4.4.1 of the memory profiling indicated that if we use only the non-zero masked values of the weight matrices, approximately 1.07 MB of memory will be needed for all the data of a training iteration (compared to initially 36.6 MB). BRAMs (5.3.2) are capable of storing up

to 4 MB, as shown in table 5.2, and therefore we are able to take advantage of them. Masks are generated for each layer and applied to weights at the level of software. As opposed to passing each masked weight matrix at its original dimensions (with zeros included as in the software), here (in FPGA) we pass only the non-zero masked weight values. In essence, each weight matrix of size [Input_nodes, Output_nodes] is converted into a masked weight matrix of size [Synapses, Output_nodes]. To ensure proper training calculations, a second matrix of size [Synapses, Output_nodes] is constructed for each layer to track the initial location of masked weight values. Here is an example of the proper use of masked weight matrices in the calculation of Z1 in the first layer of forward propagation:

for $j \leftarrow 0$ `to 2048` **do** ▷ *Output_nodes*

  for $k \leftarrow 0$ `to 9` **do** ▷ *Synapses*

    for $i \leftarrow 0$ `to 16` **do** ▷ *Batch_size*

      Z1[i][j] = Input[i][W_loc1[k][j]] * W_masked1[k][j] + b1[j];

      ▷ W_masked corresponds to the masked weight value matrix

Essentially, W_loc1 serves as a column index of the Input matrix for selecting the appropriate input value that corresponds to the masked weight value in each iteration. As a result, the calculations are performed appropriately. Here is the same example using the original dimensions of the weight matrix (as in the software implementation):

for $j \leftarrow 0$ `to 2048` **do** ▷ *Output_nodes*

  for $k \leftarrow 0$ `to 784` **do** ▷ *Input_nodes*

    for $i \leftarrow 0$ `to 16` **do** ▷ *Batch_size*

      Z1[i][j] = Input[i][k] * W1[k][j] + b1[j];

By using these two masked weight matrices (value and location) instead of the initial larger weight matrix, we reduce the memory footprint of training weights. The above example illustrates this clearly, where the for-loop of 784 iterations (Input_nodes in layer 1) has been replaced by a for-loop of 9 iterations (Synapses in layer 1). Table 5.1 provides the dimensions of these matrices, as well as the initial dimensions, for each layer of our bio-inspired NN. Bias matrices remain unchanged. It is important to note that the conversion

of weight matrices is feasible due to the stability of masks (and W_loc matrices as a result) throughout the training process in our bio-inspired ANN.

TABLE 5.1: Dimensions of the initial weight matrix, the masked weight matrix, and the masked weight location matrix for each layer of our bio-inspired NN.

Layer	Initial Weight Value Matrices [input_nodes, units]	Masked Weight Value Matrices [synapses, units]	Masked Weight Location Matrices [synapses, units]
1	[784, 2048]	[9, 2048]	[9, 2048]
2	[2048, 128]	[16, 128]	[16, 128]
3	[128, 128]	[9, 128]	[9, 128]
4	[128, 16]	[8, 16]	[8, 16]
5	[16, 10]	[16, 10]	[16, 10]

5.1.2 Backpropagation Block

As previously discussed, the forward propagation block was designed and implemented in the thesis of Nikoletta Palatianna. For the purpose of explaining our architecture, we present only the inputs and outputs of each layer of forward propagation block (fig. 5.2).

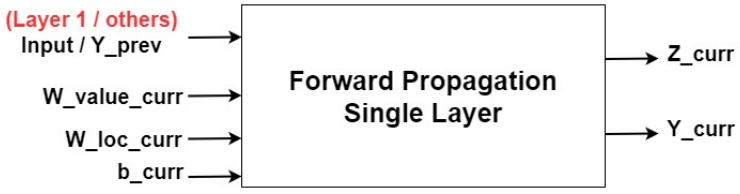

FIGURE 5.2: Single-layer forward propagation block inputs and outputs.

In section 4.3.4, we described how backpropagation was implemented in software (Numpy). The process proceeds backwards from the last (fifth) layer to the first layer. Figure 5.3 illustrates the high-level design of the backpropagation block. The outputs and inputs of a single layer backpropagation block are shown in figure 5.4. For each backpropagation calculation, we try to exploit parallelization along the batch. Moreover, we attempt to run all

calculations per layer at the highest level possible simultaneously. The limitation is that the calculation of dZ must be completed prior to the others, since the result of dZ is necessary to perform other calculations.

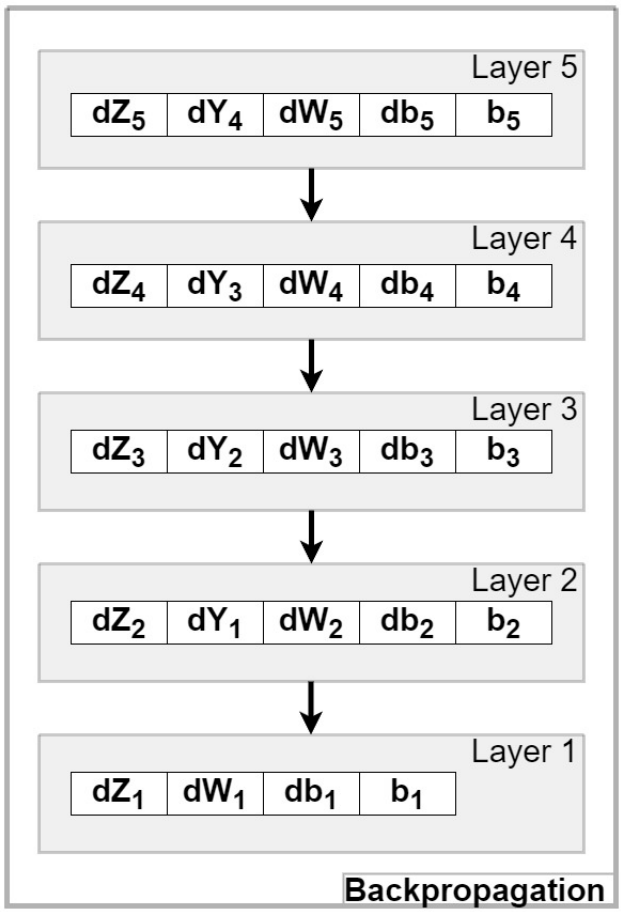

FIGURE 5.3: High-level Backpropagation Block Design.

The algorithm for the backpropagation process for a single layer in Vivado HLS is presented in 33. Layers 4 to 2 are covered by this algorithm. It should be noted that dZ is calculated differently in the fifth layer (which will be discussed later) and that the first layer does not include the calculation of dY_{prev}. This is the final version (second approach) of the HLS algorithm that has been optimized. The implementation of backpropagation using Vivado HLS will be discussed in detail in sections 5.6.1 (the first approach) and 5.7.1 (the second optimized approach). In these sections, we will discuss the problems encountered during implementation, how we resolved them, and why we opted for this second approach. Furthermore, these sections analyze the parallelization and scheduling of backpropagation computations. Detailed block designs for each backpropagation calculation are presented below.

Algorithm 4 Second Approach - Single layer Backpropagation in Vivado HLS

```
1: procedure OPTIMIZED_SINGLE_LAYER_BACKPROPAGATION_HLS
2:     for j ← 0 to Output_nodes do                                  ▷ Units
3:         for k ← 0 to Synapses do
4:             Wloc=W_curr_loc[k][j]
5:             Wcurr=W_curr_value[k][j]
6: # pragma HLS PIPELINE II=1
7:             for i ← 0 to Batch_size do
8:                 if Z_curr[i][j] ≤0 then                ▷ backward Leaky ReLU
9:                     dZ_curr[i][j]=dY_curr[i][j]∗0.1
10:                else
11:                    dZ_curr[i][j]=dY_curr[i][j]
12:                end if
13:
14:                ▷ (div_batch=1/Batch_Size)
15:                dZ_curr_batch[i][j] = dZ_cur[i][j] ∗ div_batch
16:                dW_curr_tmp[i] = Y_prev[i][Wloc]*dZ_curr_batch[i][j]
17:                dY_prev_tmp[i][k]+ = dZ_curr[i][j] ∗ Wcurr
18:            end for
19:
20:            tmpdw_curr = 0
21:            tmpdb_curr = 0
22:            for i ← 0 to Batch_size do
23:                dY_prev[i][Wloc]=dY_prev_tmp[i][k]
24:                dY_prev_tmp[i][k]=0
25:                tmpdw_curr+=dW_curr_tmp[i]
26:                tmpdb_curr+=dZ_curr_batch[i][j]
27:            end for
28:
29:            dW_curr[k][j]=tmpdw_curr
30:            db_curr[j]=tmpdb_curr
31:        end for
32:        b_curr[j] = b_curr[j]-db_curr[j]*learning_rate
33:    end for
34: end procedure
      =0
```

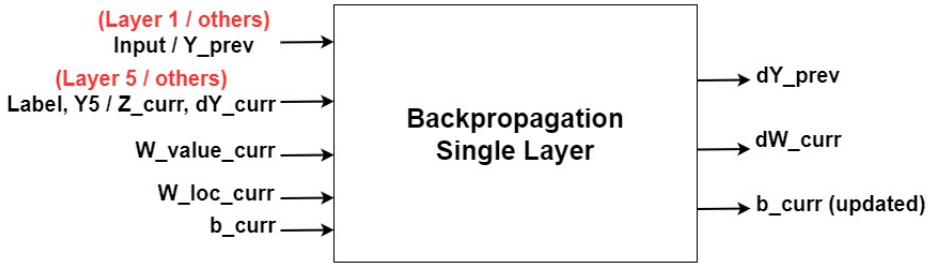

FIGURE 5.4: Single-layer backpropagation block inputs and outputs.

dZ Calculation

Softmax serves as the activation function in the 5th (last) layer, therefore, dZ_5 is calculated using its backward version. In particular, instead of lines 8 to 12 in algorithm 33, it is calculated as follows:

$$dZ_5[i][j] = Y_5[i][j] - Y\_Label[i][j], \tag{5.1}$$

where Y_5 is the output of forward propagation and Y_Label represents the input training labels from MNIST dataset.

The block diagram of this calculation is shown in figure 5.5.

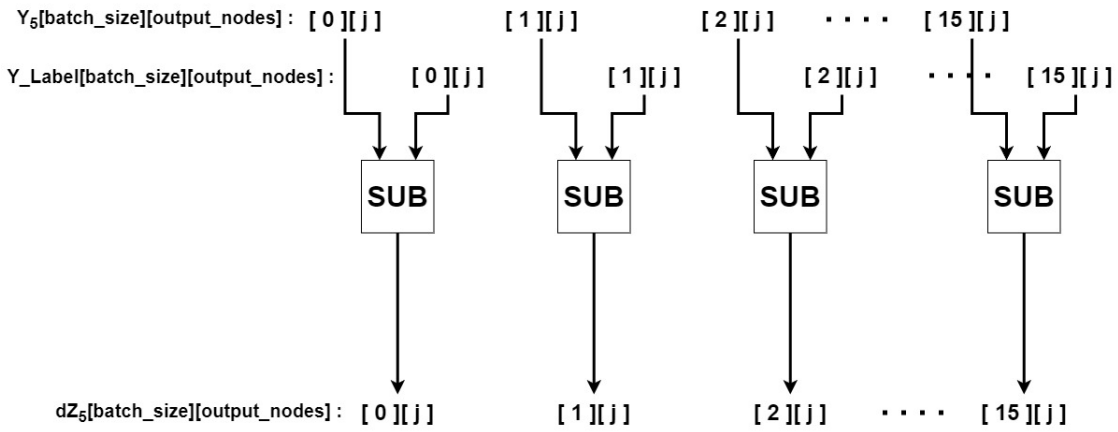

FIGURE 5.5: Block Design for dZ calculation of Backpropagation in layer 5 of our bio-inspired ANN. In the 5th (last) layer, softmax serves as the activation function. Therefore, we use its backward version to calculate dZ.

Layers 4 to 1 use leaky ReLU as their activation function, so dZ is calculated using its backward version in these cases. Figure 5.6 illustrates the block

diagram of this calculation for a single layer. The FCMP block concerns a Vivado HLS operator that compares single precision floating point values. Essentially, we design an if-else structure using the FCMP and MUX blocks.

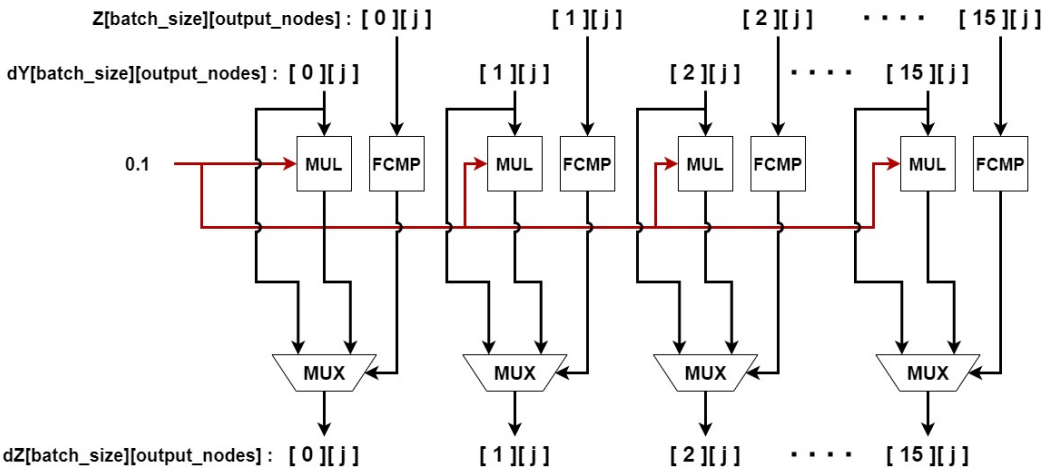

FIGURE 5.6: Block Design for dZ calculation of Backpropagation in layers 4 to 1 of our bio-inspired ANN. For these layers, the Leaky ReLU serves as the activation function. Therefore, we use its backward version to calculate dZ.

Since each term associated with this calculation has batch_size as a row index, 16 dZ calculations can be performed simultaneously (in both cases).

dY_prev Calculation

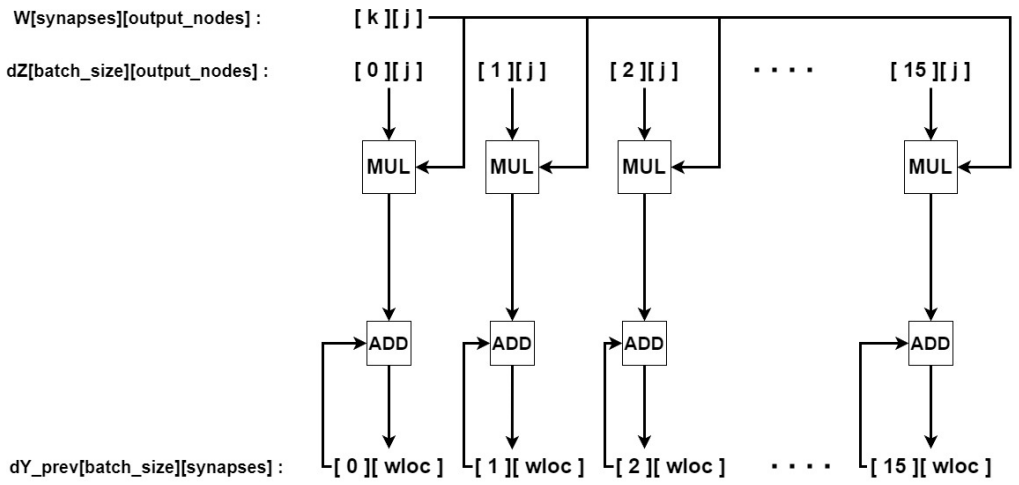

FIGURE 5.7: Block Design for dY_{prev} calculation of Backpropagation in layers 5 to 2 of our bio-inspired ANN. This calculation is not included in layer 1.

Figure 5.7 illustrates the block design for the dY_{prev} calculation. As with the dZ calculation, parallelization can be achieved along the batch, executing 16 dY_{prev} calculations simultaneously.

dW Calculation

Due to the fact that dW is not parallelized along the batch, we cannot execute multiple dW calculations in parallel. In order to compute a single value of dW, we must perform 16 (batch_size) iterations of calculations. Following is a figure (5.8) showing how we parallelize these calculations. In order to avoid dividing by 16, we multiply with 0.0625, which is equal to 1/16.

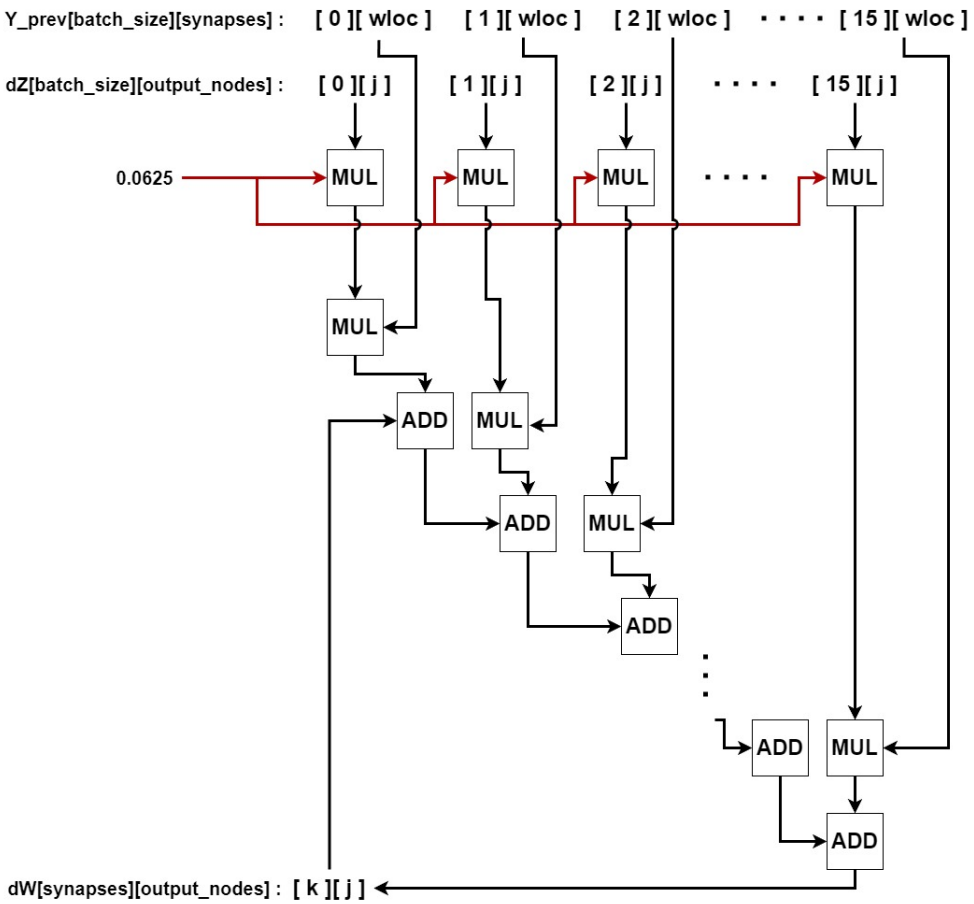

FIGURE 5.8: Block Design for dW calculation of Backpropagation. Rather than dividing by 16 (batch_size), we multiply with 0.0625, which is equal to 1/16.

db Calculation and Update of biases

As with the dW calculation, db is also not parallelized within a batch, and 16 iterations of calculations are required to calculate a single db value. Figure

5.9 illustrates the block diagram for computing *db*.

It is important to note that the biases are updated using the classical gradient descent method. In the implementation of our model, we were not able to obtain the desired results when biases were updated with Adam. We will discuss this in more detail later (in 5.5). This is the reason why the update of bias is included in the backpropagation process. Consequently, the Adam algorithm is used only to update the weights.

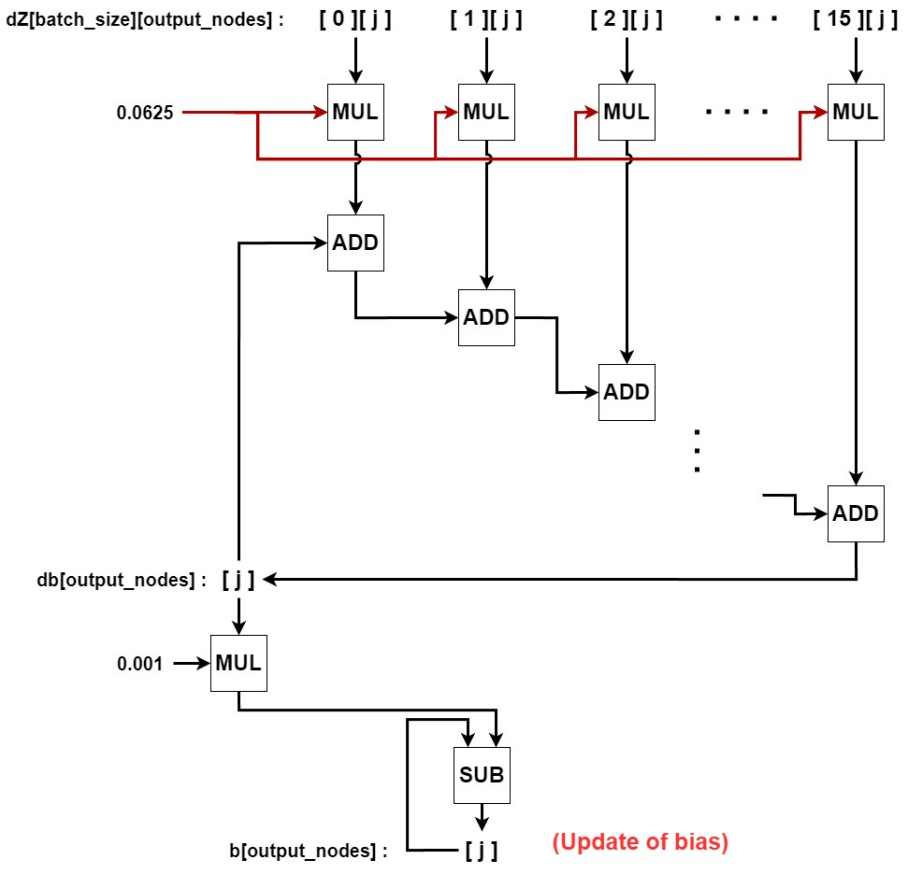

FIGURE 5.9: Block Design for calculating *db* of Backpropagation and updating bias. Rather than dividing by 16 (batch_size), we multiply with 0.0625, which is equal to 1/16.

5.1.3 Adam Algorithm Block

Section 4.3.5 analyzes Numpy's implementation of Adam Optimization Algorithm. A high-level diagram of the Adam Algorithm block for FPGA is illustrated in figure 5.10. Figure 5.11 shows the inputs and outputs of a single layer Adam algorithm block. Parallelization along the synapses

is exploited in this case. In Adam Algorithm computations, all the matrices have row indexes associated with Synapses and column indexes associated with Output_nodes. In most layers, the Output_nodes value is quite large, which is the reason for choosing Synapses for parallelization. Choosing Output_nodes would require a considerable amount of resources, which we could not afford. Furthermore, we attempt to run all calculations per layer simultaneously at the highest level possible.

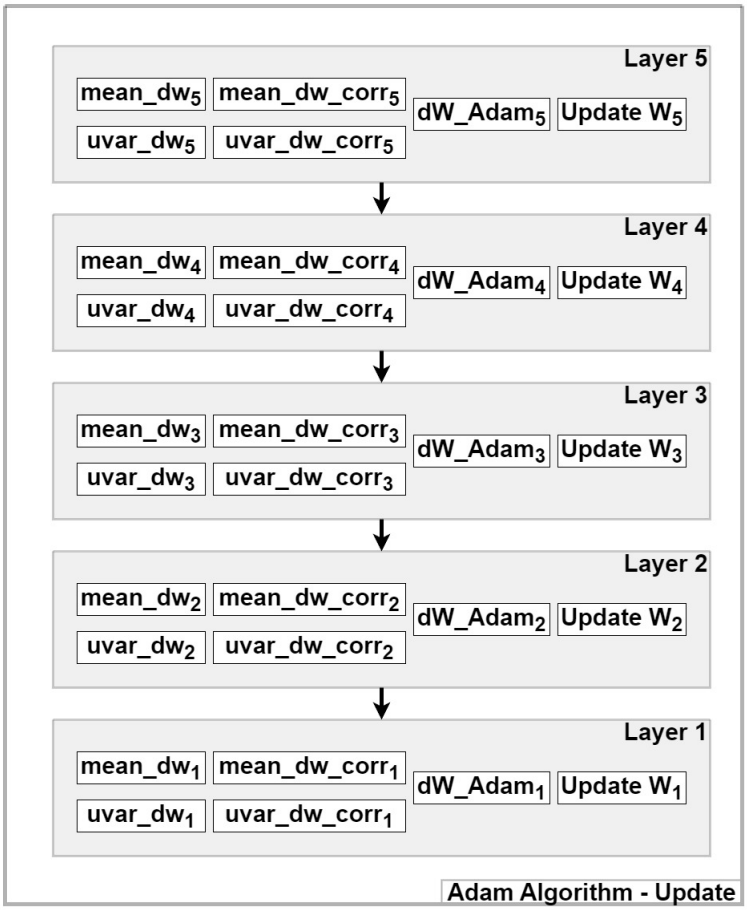

FIGURE 5.10: High-level Adam Algorithm Block Design.

In algorithm 16, the Adam algorithm for a single layer in Vivado HLS is presented. This is the final version (second approach) of the HLS algorithm. A detailed discussion of the Adam algorithm implementation using Vivado HLS will be provided in sections 5.6.2 (the first approach) and 5.7.2 (the second optimized approach). Throughout these sections, we analyze the parallelization and scheduling of Adam algorithm computations, as well as the challenges encountered during their implementation. Detailed block designs for each Adam Algorithm calculation are presented below.

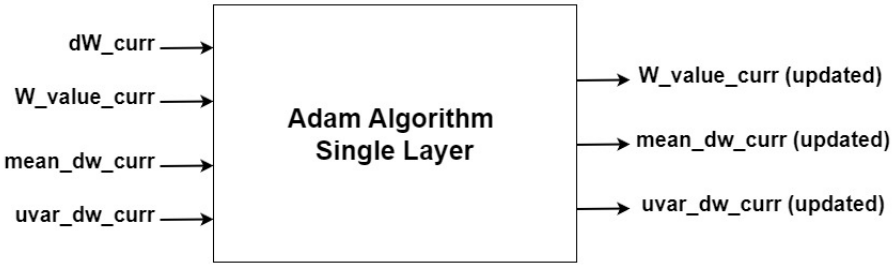

FIGURE 5.11: Single-layer Adam algorithm block inputs and outputs.

Algorithm 5 Second Approach - Single layer Adam algorithm Update method in Vivado HLS

1: **procedure** SINGLE_LAYER_ADAM_ALGORITH_UPDATE_HLS
2: **for** $j \leftarrow 0$ to $Output\_nodes$ **do** ▷ Units
3: # pragma HLS PIPELINE II=1
4: **for** $k \leftarrow 0$ to $Synapses$ **do**
5: $mean\_dw\_curr[k][j]$=0.9*$mean\_dw\_curr[k][j]$+0.1*$dW\_curr[k][j]$
6: $uvar\_dw\_tmp[k][j]$=0.999 $*$ $uvar\_dw\_curr[k][j]$+0.001*$dW\_curr[k][j]$*$dW\_curr[k]$
7: **if** ($uvar\_dw\_tmp[k][j] > uvar\_dw\_curr[k][j]$) **then**
8: $uvar\_dw\_curr[k][j]$=$uvar\_dw\_tmp[k][j]$
9: **end if**
10: ▷ t represents the current number of training iterations
11: $mean\_dw\_corr[k][j]$=$mean\_dw\_curr[k][j]$/(1-$powf(0.9,t)$)
12: $uvar\_dw\_corr[k][j]$=$uvar\_dw\_curr[k][j]$/(1-$powf(0.999,t)$)
13: $dW\_adam\_curr[k][j]$=$learning\_rate$*($mean\_dw\_corr[k][j]$/($hls::sqrtf(uvar\_dw\_corr[k][j])$+0.00000001)]
14: $W\_curr[k][j]$=$W\_curr[k][j]$-$dW\_adam\_curr[k][j]$
15: **end for**
16: **end for**
17: **end procedure**

mean_dw Calculation

Figure 5.12 illustrates the block design for the *mean_dw* computation. It is possible to carry out all steps (blocks) for the *mean_dw* calculation in parallel along the Synapses.

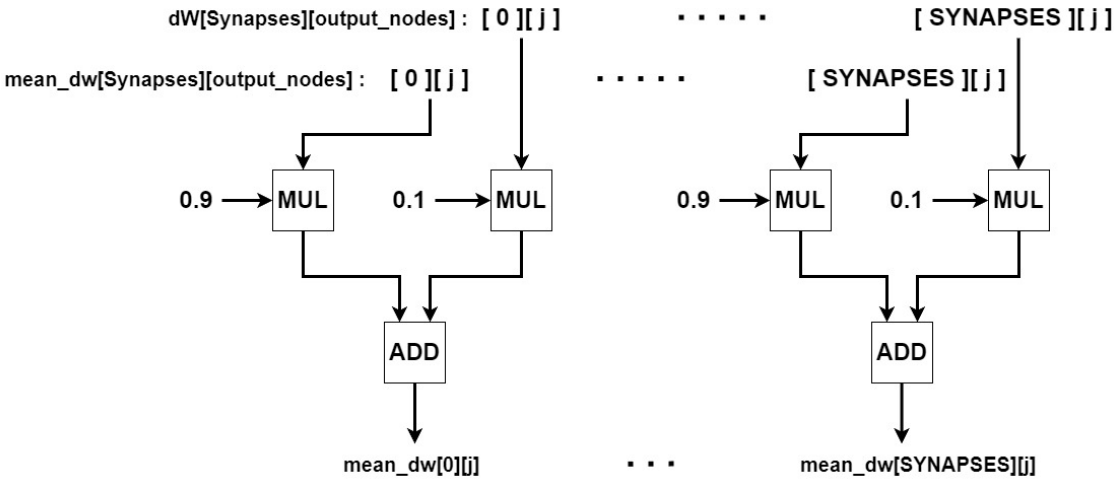

FIGURE 5.12: Block Design for *mean_dw* calculation of Adam Algorithm of our bio-inspired ANN.

uvar_dw Calculation

The figure 5.13 illustrates the block design for the computation of *uvar_dw*. As with *mean_dw*, all calculations (blocks) can be parallelized along the Synapses.

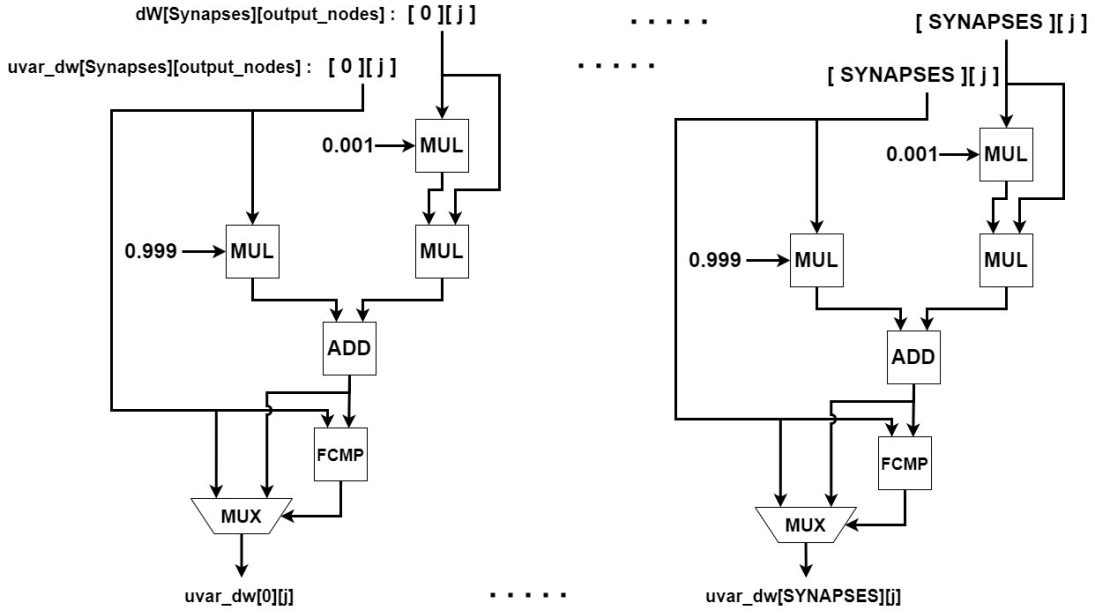

FIGURE 5.13: Block Design for *uvar_dw* calculation of Adam Algorithm of our bio-inspired ANN.

mean_dw_corr and uvar_dw_corr Calculation

A diagram of the block design for calculating *uvar_dw_corr* is shown in figure 5.14. The POW and SUB blocks are executed only once. For divisions, parallelism along synapses can be exploited. The Pow function used here is provided by the Vivado HLS math library for computing the power of a number and supports single-precision (float) computations. The term t represents the number of training iterations currently being conducted. The design of *mean_dw_corr* is similar to that of *uvar_dw_corr*, with the only difference being the use of the value 0.9 in Pow function instead of 0.999.

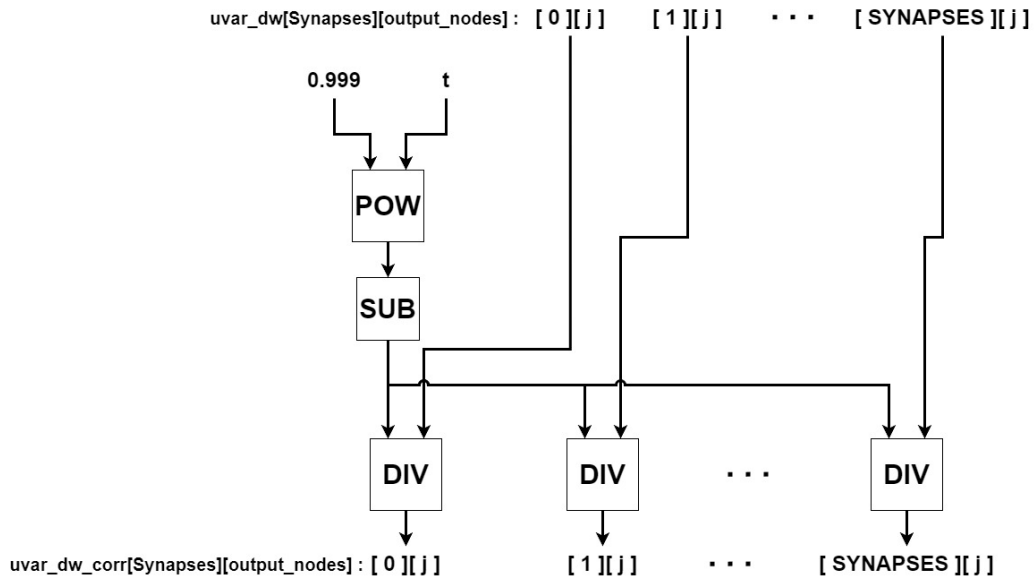

FIGURE 5.14: Block Design for *uvar_dw_corr* calculation of Adam Algorithm of our bio-inspired ANN.

dW_Adam Calculation and Update of Weights

Figure 5.15 illustrates the block design for computing *dW_Adam* and updating W. It is possible to parallelize all calculations (blocks) along the Synapses. The Sqrt function (SQRT block) provided by the Vivado HLS math library is used here to calculate single-precision floating point square roots.

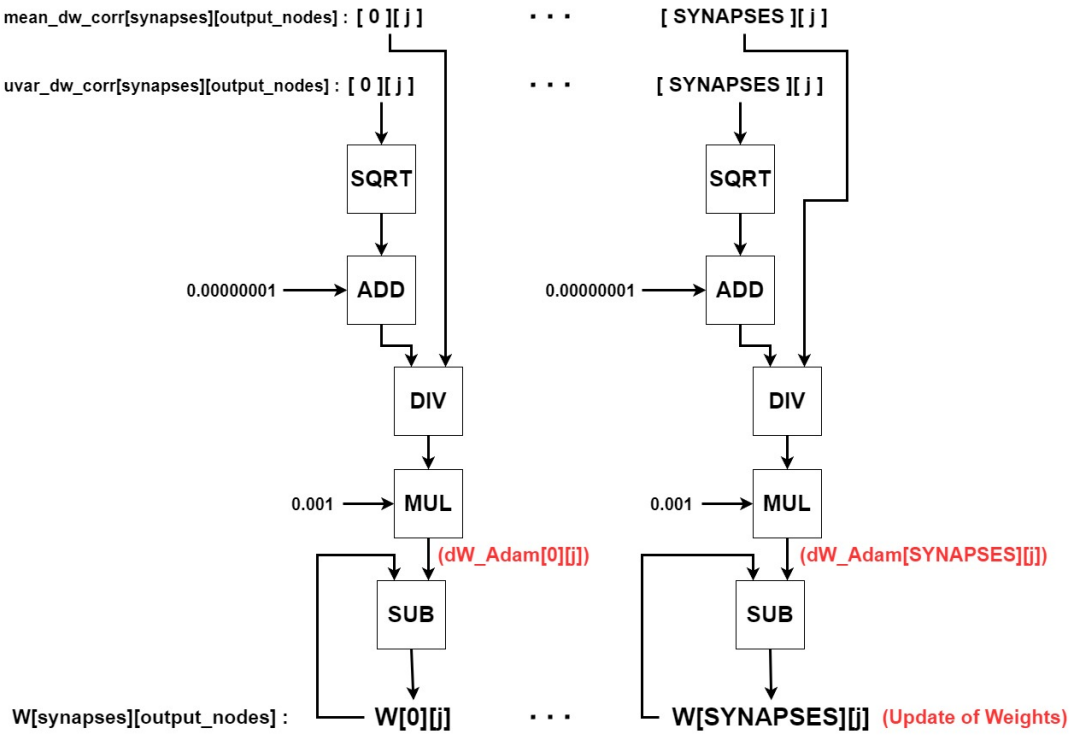

FIGURE 5.15: Block Design for calculating *dW_Adam* and updating the Weights of Adam Algorithm of our bio-inspired ANN.

5.2 Tools Used

In this thesis, the hardware implementation was developed using Vivado Design Suite HLx Editions tool package (2019.1.3 version) [41] provided by Xilinx. This package is fully supported and licensed by the Microprocessors Hardware Lab (MHL) at Technical University of Crete. There are three tools included in the package: Vivado IDE, Vivado HLS, and Vivado SDK. It is important to note that each of these tools complements the other.

5.2.1 Vivado High Level Synthesis (HLS)

In Vivado High-Level Synthesis (HLS) tool [42], an algorithm (function) written in C, C++, or System C is transformed into a low-level register transfer level (RTL) implementation in hardware description language (HDL) format. This format can be synthesized into a Xilinx field programmable gate array (FPGA). In this tool, algorithms are developed and verified at the C-level, allowing for a high level of abstraction while designing in hardware.

As a result, both the development and validation times are significantly reduced compared with traditional hardware description languages. A further advantage of Vivado HLS is that it provides hardware designers with the ability to control how the synthesis process is performed using optimization directives. The optimization directives are used to modify and control the behavior of the internal logic and I/O ports. They are optional, but highly recommended since they have a significant impact on the hardware implementation's performance.

As part of the High-Level Synthesis process, the scheduling phase determines which operations are performed during each clock cycle. Scheduling depends on optimization directives in addition to the clock cycle length and the duration of operations. Besides the scheduling phase, there are two more phases in the High-Level Synthesis process, binding and control logic extraction. In the binding phase, information about the target device is used to determine which hardware resource implements each scheduled operation.

Pipeline Directive

For example, pipeline directive enables concurrent execution of operations within a function or loop. In each execution step, not all operations must be completed before the next one begins. This directive can specify the number of clock cycles required from the pipelined function or loop to process new inputs. This number refers to the initiation interval (II). The goal is to improve throughput or reduce the initiation interval. The figure (5.16) illustrates an example of loop pipelining, which improves both the initiation interval (1 clock cycle instead of 3) and latency (4 clock cycles instead of 8), without requiring additional hardware resources.

Interface Directive

During the synthesis of the C code, top-level function arguments are converted into RTL I/O ports. Therefore, specifying the I/O protocol is essential in order to ensure the final design is compatible with other hardware blocks that support the same I/O protocol. This specification can be achieved by applying the interface directive, another important optimization directive.

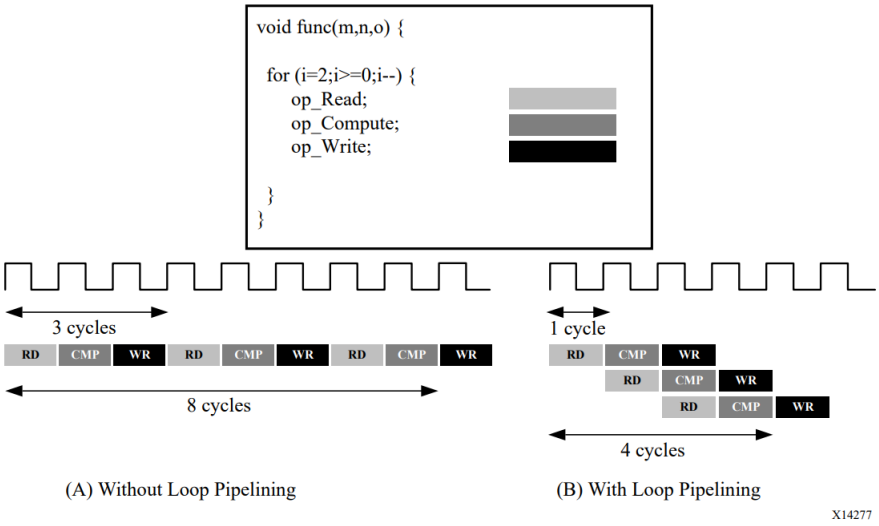

FIGURE 5.16: Loop Pipelining - https://docs.xilinx.com/v/u/2019.1-English/ug902-vivado-high-level-synthesis.

Array Partition Directive

Furthermore, arrays in the C code are synthesized into block RAM or UltraRAM in the final design. By default, an array is implemented as a single block RAM with a maximum of two data ports. As a result, read/write intensive algorithms can have limited throughput. By using the array partition optimization directive, arrays can be partitioned into multiple smaller ones or individual registers. This directive effectively increases the number of read/write ports and improves the throughput of the design. On the downside, it requires more memory instances.

Unroll Directive

Concerning loops, they are rolled by default, executing the logic of each iteration in sequence. Loops can be unrolled by applying the optimization directive unroll. This will create multiple loop instances to run them in parallel. However, it requires a lot of hardware to achieve maximum computational overlapping. It is also possible to pipeline loops, as mentioned previously. In nested loops, if the outer is pipelined, then the inner is unrolled.

Synthesis Report - Performance Metrics

High-level synthesis generates a synthesis report that includes performance metrics. These performance metrics are described below:

- **Area**: The amount of hardware resources needed to implement the design. Look-up tables (LUT), Flip-Flops (FF), registers, Block RAMs (BRAMs), and DSP elements are among the available hardware resources in the FPGA.
- **Latency**: The number of clock cycles required for the function or loop to compute all output values.
- **Initiation interval (II)**: The number of clock cycles required for the function or loop to process new input data.
- **Loop iteration latency**: The number of clock cycles required by a loop to complete one iteration.
- **Loop initiation interval**: The number of clock cycles before the next iteration of the loop starts to process data.
- **Loop latency**: The number of clock cycles to execute all iterations of the loop.

The implementation can be improved by experimenting with different optimization directives and reviewing the synthesis report each time. In addition to the report, an analysis perspective is available at the end of the synthesis process. Analysis perspective can assist hardware designers in detecting and resolving possible violations and timing and data dependencies. Moreover, it can be helpful in reducing latency and parallelizing the design. Hence, this analysis can help designers improve their designs further by providing a deeper understanding.

Following the analysis, the generated RTL implementation is compared to the original C code to verify it is functionally identical. Finally, Vivado HLS produces the IP block that can be imported into the Vivado IP catalog for use in the Vivado Design Suite.

As reported in [42], Vivado HLS desing flow includes:

1. Compile, execute (simulate), and debug the C algorithm.
2. Synthesize the C algorithm into an RTL implementation
3. Generate comprehensive reports and analyze the design.
4. Verify the RTL implementation.
5. Package the RTL implementation as an IP block.

5.2.2 Vivado IDE

Vivado Design Suite includes Vivado Integrated Design Environment (IDE) [43], which provides a graphical user interface (GUI) to design, validate, synthesize, implement, place and route any FPGA design. FPGA designs are written in HDL languages such as Verilog or VHDL, and IP blocks created using Vivado HLS (C/C++) can also be included. Vivado IDE is equipped with a powerful feature called Vivado IP (Intellectual Property) integrator [44], which enables designers to instantiate, connect and configure various IPs. This can be done interactively through its GUI interface or programmatically through the Tool Command Language (Tcl) programming interface. In this way, this tool facilitates the design of complex systems. IP cores, along with those generated from the Vivado HLS, can be accessed easily through the Vivado IP catalog.

The Vivado IP integrator utilizes the AMBA AXI4 interconnect protocol (described in 5.3.1) to connect various IPs together. According to this protocol, communication between IP cores is achieved via AXI master and AXI slave interfaces. When compared with traditional RTL-based IP connectivity, standard interfaces save time. The term interface refers to a grouping of signals with a common function. It would be more complex to connect IPs if each signal was connected separately on Vivado's GUI. Grouping these signals into an interface allows the connection to be represented graphically as a single one. The connection of IPs is also accelerated by the tool's connection automation feature. Furthermore, IP integrator provides Design Rule Checks (DRCs) to verify IP configuration and connectivity.

First, a block design is created, in which all the necessary IPs are added and connected appropriately according to the above protocol. Typically, each design contains the central processing unit (Zynq, MicroBlaze), custom IPs, and modules related to memory, communication, and interconnection. A clock signal and reset signal are present on every module, and a base address is automatically assigned to most modules. Base addresses can be viewed and edited using the address editor. The design is then validated to ensure that no errors or violations have occurred. Following that, the RTL design can be converted into a logic gate schematic through the synthesis process. Once the synthesis is successful, implementation (place and route) can begin. During implementation [45], the logical netlist is mapped into the physical array of the target Xilinx device. In particular, it involves logic optimization, placement of logic cells, and routing of connections between cells. After synthesis

and implementation, Vivado IDE generates many types of reports [46], including the most critical ones about timing, utilization and power. In a synthesized design, the tool's timing engine estimates the net delays based on connectivity and fanout, and in an implemented design, the net delays are based on the actual routing information. The utilization report breaks down the design utilization based on resource type. When the implementation is successful, the bitstream file can be generated for programming the FPGA.

5.2.3 Vivado SDK

The Xilinx Software Development Kit (SDK) [47] provides an IDE that helps developers create embedded software applications for Xilinx ARM processors, such as Zynq UltraScale+ MPSoC, Zynq-7000 SoCs, and the Microblaze microprocessor. It is based on the Eclipse open source standard. Among the features of the SDK are a C/C++ code editor, a compiler, build tools, flash memory management, automatic Makefile generation, as well as debugging and profiling capabilities. Vivado Design Suite versions 2019.2 and later integrate SDK, SDSoC, and SDAccel development environments into Vitis unified software platform.

Vivado IDE can launch SDK once the generated hardware is exported along with the generated bitstream file. SDK automatically imports the project hardware wrapper (from Vivado IDE), which includes system.hdf and other necessary files. The system.hdf file provides the address map for the processors of the target FPGA (for instance, ZCU102 board has 4 cortexa53 processors). By creating an application project, the project files are generated along with the Board Support Package (BSP), which includes the appropriate device drivers for all the peripherals in the hardware design and libraries to configure the FPGA. Users can control various settings of BSP and view information about it through the system.mss file. In project files, there is an important file called lscript.ld (Linker Script), which describes the memory layout of the target FPGA, and specifies where each section of the program should be placed in memory. This file allows users to define new memory regions, change the assignment of sections to memory regions, and modify stack and heap sizes.

The C/C++ programming procedure handles the PL part of the FPGA. It begins by writing data to BRAMs, initializing modules such as custom IPs and DMAs, and then executing them. To be able to program the FPGA, it

must be connected via the JTAG port to a PC. Furthermore, the UART port can be used to send and receive data as well as monitor and debug the FPGA.

5.3 FPGA Platform

The architecture of our thesis targets the Xilinx ZCU102 evaluation board [48]. The ZCU102 is a general purpose evaluation board for rapid-prototyping based on the Zynq® UltraScale+™ XCZU9EG-2FFVB1156E MPSoC (multi-processor system-on-chip), which combines a powerful processing system (PS) and user-programmable logic (PL) into the same device. High speed DDR4 SODIMM and component memory interfaces, FMC expansion ports, multi-gigabit per second serial transceivers, a variety of peripheral interfaces, and FPGA logic for user customized designs provides a flexible prototyping platform. There are three major processing units on the PS block of the board: a quad-core 64bit ARMv8-A Cortex-A53 (application processing unit - APU), a dual-core 32bit ARM v7-R Cortex-R5 (real-time processing unit - RPU) and an ARM Mali-400 MP2 graphics processing unit (GPU).

TABLE 5.2: 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

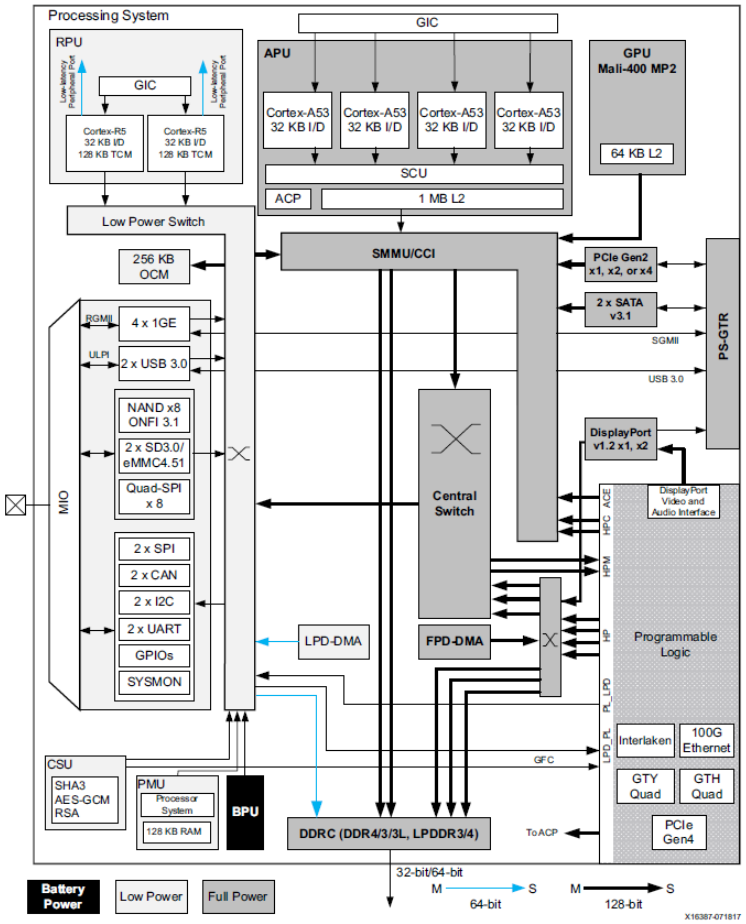

FIGURE 5.17: Zynq UltraScale+ MPSoC Top-Level Block Diagram - https://docs.xilinx.com/v/u/en-US/ug1182-zcu102-eval-bd.

5.3.1 AXI4 Interface Protocol

Advanced eXtensible Interface (AXI) [49] is part of ARM AMBA, a family of micro controller buses first introduced in 1996. In 2010, AXI4, the second major version of AXI, was released as part of AMBA 4.0. AXI is the primary interface standard used within the PL, for transferring data between IP blocks. As mentioned above, communication between IP cores is achieved through the use of AXI master and AXI slave interfaces. Multiple memory-mapped AXI masters and slaves can be connected together using AXI infrastructure IP blocks, such as the AXI Interconnect IP and the AXI SmartConnect IP. There are three types of AXI4 interfaces:

- **AXI4-Lite (slave interface)**: For simple, low-throughput memory-mapped communication that has a small logic footprint. For example, it is used

for the control signals (Start, IsDone, IsReady, return values) of an IP core, so the PS (Zynq/Microblaze) can control it.

- **AXI4-Stream**: For high-speed streaming data. This protocol is particularly useful for signal processing in video, communications and networking applications. AXI4-Stream removes the requirement for an address phase completely and allows unlimited data burst size. Therefore, AXI4-Stream interfaces and transfers are not considered to be memory-mapped.
- **AXI4 (Full or Master)**: For high-performance memory-mapped requirements, allowing high throughput bursts of up to 256 data transfer cycles with just a single address phase.

5.3.2 PL-PS Communication Methods

It is crucial to understand how PS-PL communication is accomplished. Data is first passed to the DDR of the processor. Then, in order to pass this data to the FPGA, the following three methods can be used:

- **Memory Mapped I/O**: This method performs input/output (I/O) between a processor (PS) and the peripheral devices (PL). The same address space is used to address both memory and I/O devices. Each component is mapped to an address value. The disadvantage of this method is that when we access the DDR randomly, we have to pay 30-50 clock cycles per request for the DDR initiation interval (initialization cost). Therefore, this method cannot handle multiple requests efficiently. However, sequential access seems to be more efficient.
- **AXI4-Stream**: This protocol supports point-to-point data streaming without requiring an address channel, providing a direct flow of data between DDR and PL. This technique eliminates the delay of requests and hides the DDR interval. In order to implement this technique, DMA IP (described in 5.3.3) can be used.
- **BRAM**: Using this method, data from the DDR is transferred to BRAM modules using a memory mapped or streaming (DMA) technique. BRAMs provide us with huge bandwidth. However, they are relatively small, only a few MBs in size. As a result, only small data sets can be used in this case.

5.3.3 AXI DMA

The AXI Direct Memory Access (AXI DMA) [50] core is a soft Xilinx IP core that provides high-bandwidth direct memory access between memory and AXI4-Stream target peripherals. Primary high-speed DMA data movement is through the AXI4 Read Master to AXI4 memory-mapped to stream (MM2S) Master, and AXI stream to memory-mapped (S2MM) Slave to AXI4 Write Master. The MM2S channel and S2MM channel operate independently. This IP core provides automatic burst mapping, the ability to queue multiple transfer requests, as well as byte-level data realignment allowing memory reads and writes starting at any byte offset location. Moreover, AXI DMA can be configured to work in any of the three modes: polling, interrupt, and Scatter-Gather (SG). Direct Register Mode (Scatter Gather Engine is disabled) provides a configuration for doing simple DMA transfers on MM2S and S2MM channels that requires less FPGA resource utilization. Source and destination address, as well as the access pattern, are accepted. The transaction starts by writing data to the DMA's registers. In interrupt mode, when the transaction is complete, it sends an interrupt to the processor. Scatter Gather operation allows a packet to be described by more than one descriptor. A typical use for this feature is to allow storing or fetching of headers from a memory location and payload data from another memory location. Software applications that take advantage of this can improve throughput. There is a maximum of 1024 bits per cycle that can be transferred using DMA on a memory bus.

5.4 FPGA Implementation

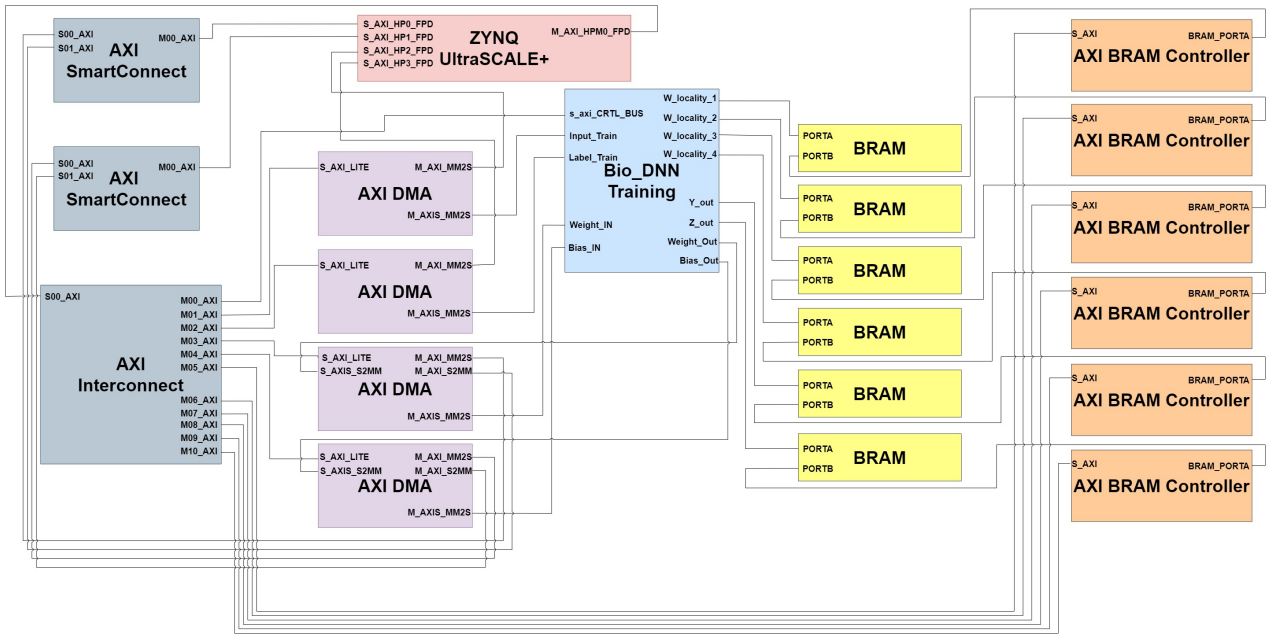

FIGURE 5.18: Bio-inspired ANN Block Design for FPGA.

The overall FPGA design of our bio-inspired ANN is shown in figure 5.18. Several modules are involved in the design, which will be explained below. First, the central processing unit (Zynq Ultrascale+) is responsible for handling all instructions, initializations, and data transfers. After that, the Bio_DNN Training IP (5.5) implemented in Vivado HLS manages the entire training process, including forward propagation, backpropagation, and parameter updating using the Adam algorithm. As far as DMAs (5.3.3) are concerned, they are used for high-speed data streaming between DDR and PL. Additionally, AXI Block RAM (BRAM) Controller IPs [51] and Block Memory Generator (BMG) IPs [52] are included in the design. The AXI BRAM Controller is designed as an AXI Endpoint slave IP for integration with the AXI interconnect and system master devices to communicate to local BRAM. Basically, this IP provides low-latency control of BRAM resources through an AXI4 (memory mapped) slave interface. As regards BMG IP, it is an advanced memory constructor that generates area and performance-optimized memories using embedded BRAM resources in Xilinx FPGAs. There are two independent read/write ports in this IP that have access to a shared memory space. As the last modules in this design, we have AXI Interconnect and two AXI SmartConnect IPs. These IPs are used to connect one or

more AXI memory-mapped master devices to one or more memory-mapped slave devices. Specifically, the AXI Interconnect is used to connect all the DMAs, BRAM controllers, and our Bio_DNN Training IP to one of the High-Performance (HP) Master AXI interfaces of the Zynq (PS) so that the Zynq can control them. The Zynq's HP slave AXI ports are utilized by the DMAs for data movement between the PS-DDR and PL. These connections are accomplished through AXI SmartConnect IPs.

5.4.1 PL-PS Communication and Memory Configuration

In each iteration of training, an input batch of images (16 images) and their labels must be passed from DDR to the Training IP (in PL of FPGA). It is also necessary to input and output the aforementioned masked weight value matrices (5.1.1) to the IP so that the training procedure can run with updated weight values at each training iteration. Similarly, the bias matrices of each layer must also be inputs and outputs of this IP. Thus, these data must be continuously transferred between DDR and PL. In these situations, AXI4-Stream (5.3.2) is the most suitable interface, as it eliminates the delay associated with requests and hides the DDR initialization cost. Connecting DDR and PL is accomplished using DMA IP.

It is important to note that the training calculations do not use these data in the order in which they are passed in the Training IP. In addition, these data must be used within the IP more than once. As a result, it is necessary to store them in the IP's internal BRAMs. By doing so, we are able to reuse these data multiple times and take full advantage of the huge bandwidth provided by BRAM (5.3.2). Using HLS, we are able to partition (5.2.1) a BRAM (array/matrix) into multiple smaller BRAMs, allowing us to have more than two memory accesses per clock cycle. The parallelization of the processes is enhanced as a result. In general, the BRAM is located in the PL part, which explains its flexibility and configurability. In the case of weight location matrices (5.1.1), they were stored in BRAMs (outside of the IP) since they remain stable throughout the entire process. BRAM Controllers are used to control the outside of IP BRAMs. All of the data described above is stored in separate BRAMs that are partitioned. The figure 5.19 illustrates the total content of the BRAM as well as the partitions for each data. In essence, the blocks represent different BRAMs.

FIGURE 5.19: Detailed information concerning the content of the BRAMs and the partitions of each data type. In essence, each block represents a BRAM.

5.4.2 Bandwidth

The memory bandwidth is expressed as follows:

$$Bandwidth = Data\_width \cdot Clock\_frequency\_PL \tag{5.2}$$

In order to improve theoretical bandwidth, it is necessary to increase the clock frequency of the PL and the data width as much as possible. Zynq's HP port limits the maximum memory bus size to 128-bit. Initially, we used a

32-bit datawidth. By upgrading to 64-bit, we observed significant improvements in both throughput and latency. The 128-bit approach (the maximum data width possible) did not result in any optimization. In spite of our efforts, we may not have implemented our architecture optimally to maximize bandwidth exploitation. We therefore decided to keep the 64-bit data width in our design.

It is theoretically possible to increase the bandwidth further by using multiple DMAs that stream data from different HP ports (of the Zynq). Due to this, we use four DMAs, each connected to a different HP slave port. There is one DMA for the training input (only input), one DMA for the training label (only input), one DMA for the network's weights (input and output), and one DMA for the network's biases (input and output). It is important to note that we did not measure the achieved bandwidth in our design or optimize our architecture to the limit. Our objective was to implement a functional and power-efficient architecture in order to speed up the training process of our bio-inspired ANN.

5.5 Overview of IP Block in Vivado HLS

The entire training procedure is implemented into one IP in Vivado HLS. Figure 5.20 shows the block diagram of Training IP. The training's forward process is implemented and analyzed extensively in the thesis of Nikoletta Palatiana. As part of this thesis, the backpropagation and the Adam Optimization algorithm are implemented.

It is important to note that the Adam algorithm is only applied to the weights of our network. The biases are updated using the classical gradient descent method (2.30). We were unable to obtain the desired results when biases were updated with Adam. This may be due to a miscalculation on our part. Our limited time constraints prevented us from resolving this issue. As a result, the network's biases are updated as part of the backpropagation process. Our design is simplified due to the use of classical gradient descent rather than Adam algorithm in bias update. There is a possibility that this simplification could negatively impact our design in terms of training error and accuracy. However, the classical update method saves hardware resources since Adam has a higher level of operational complexity. Our belief is that the impact of the bias update method in the final accuracy/error results is insignificant.

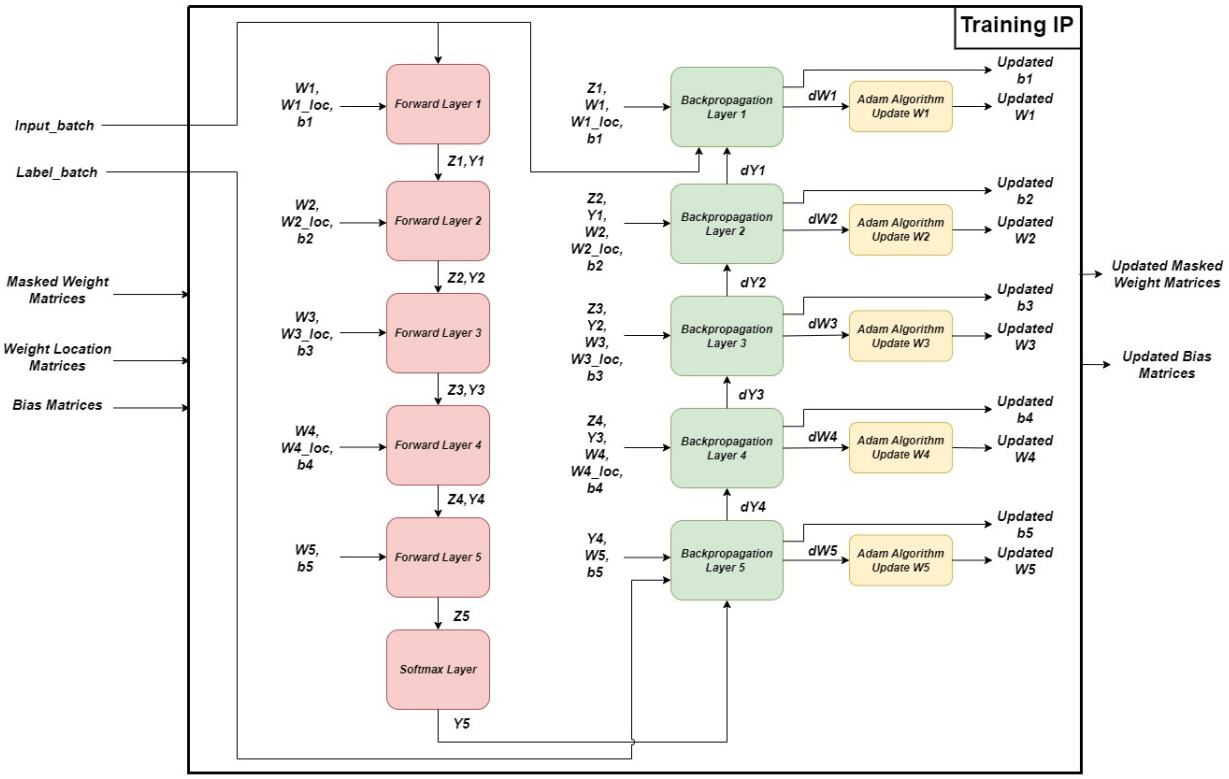

FIGURE 5.20: Block Diagram of Training IP in HLS.

It was previously noted (5.2.1) that Vivado HLS allows hardware designers to modify and control the default behavior of the synthesis process in terms of internal logic and I/O ports through the use of optimization directives. Consequently, we are able to develop specific high-performance hardware implementations. Additionally, by using optimization directives, it is possible to generate variations of the hardware implementation from the same C source code. This allows us to explore the design space, increasing our chances of finding an optimal solution. The HLS synthesis report (5.2.1) can be used to determine if the design meets our requirements.

The purpose of the two following sections is to analyze the scheduling of the backpropagation process, as well as the scheduling of the Adam Optimization algorithm. In HLS, the scheduling phase determines which operations will be performed during each clock cycle. A list of the basic Vivado HLS operators can be found in table 5.3. Besides the clock cycle length and the duration of operations, scheduling also depends on optimization directives.

TABLE 5.3: Basic Vivado HLS Operators

Operator	Description
FADD	Single-precision floating point addition
DADD	Double-precision floating point addition
FMUL	Single-precision floating point multiplication
DMUL	Double-precision floating point multiplication
FSUB	Single-precision floating point subtraction
DSUB	Double-precision floating point subtraction
FDIV	Single-precision floating point division
DDIV	Double -precision floating point division
FCMP	Single-precision floating point comparison
FSQRT	Single-precision floating point square root

5.6 Implementation of IP Block (HLS) - First Approach

5.6.1 Backpropagation in Vivado HLS

In this section, we analyze the scheduling of the backpropagation process of training IP. As a starting point, we will explain the case of layers 4 to 2, where Leaky ReLU was used as a backward activation function. Fifth (last) layer is distinguished by the use of softmax backward activation function. In the case of the first layer, dY_prev calculation is not required. There will be a discussion later on the differences in implementation of layers 1 and 5.

The backpropagation algorithm for a single layer of our network, which is written in C code in HLS, is presented here (25). This algorithm consists of three nested for-loops, which are illustrated separately below.

for $j \leftarrow 0$ `to` $Output\_nodes$ **do**

for $k \leftarrow 0$ `to` $Synapses$ **do**

pragma HLS PIPELINE II=1

for $i \leftarrow 0$ `to` $Batch\_size$ **do**

`<do calculations>`

Without any optimization, each layer would require Output_nodes * Synapses * Batch_size iterations. With the pipeline directive (5.2.1) applied to the second inner for-loop, we are able to perform concurrent operations within this loop. The 'II' specifies the desired initiation interval (5.2.1) for the pipeline.

As II=1, new input data can be processed in each clock cycle. By pipelining the second inner for-loop, the third inner for-loop will be automatically unrolled (5.2.1). Thus, 16 (Batch_size) loop instances are created for parallel execution. In essence, each layer requires Output_nodes * Synapses iterations that can be parallelized along the batch and started one at a time in each clock cycle. Furthermore, we apply array partitioning directives (5.2.1) to the majority of the matrices used in order to enhance the design's throughput. As mentioned above, arrays and matrices are typically synthesized into block RAMs with no more than two data ports in the final design. Due to partitioning, we are able to avoid **memory dependency** issues by increasing the number of ports for reading and writing per matrix (BRAM). In general, memory dependencies arise when the loop is pipelined with II=1 and the user performs more than two accesses per loop iteration without partitioning the array. When this occurs, the tool will not be able to schedule the operations in one cycle. Consequently, partitioning allows us to have more than two memory accesses per clock cycle and exploit the unrolling (parallelization) of the third inner loop. As a drawback, partitioning requires more memory instances.

Figure 5.21 shows the schedule of the total backpropagation calculations, except for the db calculation and the bias updating (explanation in 5.6.1). As can be seen, the iteration latency (5.2.1) is 73 clock cycles.

FIGURE 5.21: First Approach - Schedule of the main backpropagation calculations (without db calculation, bias update) in HLS. The number 73 represents the iteration latency (in cycles).

Below (in figure 5.22) is an explanation of how the second for-loop (of Synapses) is pipelined with II=1. In each iteration, the calculations are carried out according to the schedule mentioned above (5.21).

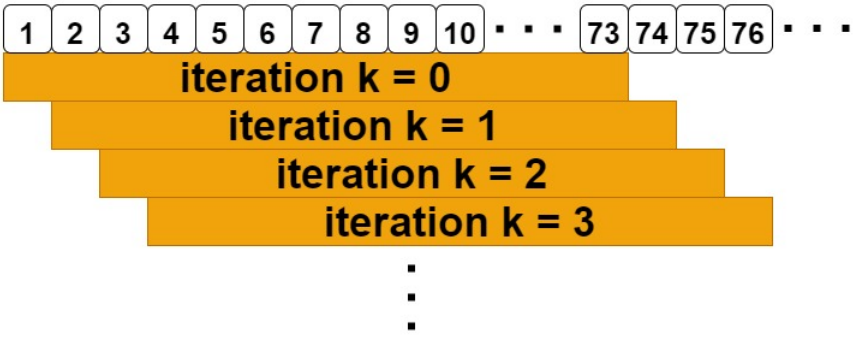

FIGURE 5.22: An illustration of how the second loop (of Synapses) in backpropagation is pipelined with II = 1. The iteration latency is 73 clock cycles.

A detailed explanation of each calculation is provided below.

Calculation of dY_prev

Initially, the calculation of dY_prev, with the software point of view, it was expressed as follows:

$$dY\_prev[i][Wloc] += dZ\_curr[i][j] * W\_curr\_value[k][j], \tag{5.3}$$

where $wloc$=$W\_curr\_loc[k][j]$ and j,k,i are the indexes of the above-mentioned three nested for-loops. Despite the fact that the dY_prev matrix (of each layer) was partitioned, we were unable to exploit the parallelization of 16 loop instances. This problem was caused by the presence of Wloc as an index for dY_prev, which resulted in a violation of II. Essentially, II violation occurs when the loop is pipelined with II=1 and memory (as described previously) or data dependencies are present. This particular situation involved a data dependency. **Data dependency** refers to a situation in which the current loop iteration uses the value generated by an earlier loop iteration and that value takes N cycles to be generated. This prevents the current iteration from starting on the next cycle. In this case, there is a data dependency between the load and store operations on the array dY_prev. This is due to the fact that the add operation of the float data requires four clock cycles. Figure 5.23 illustrates this II violation.

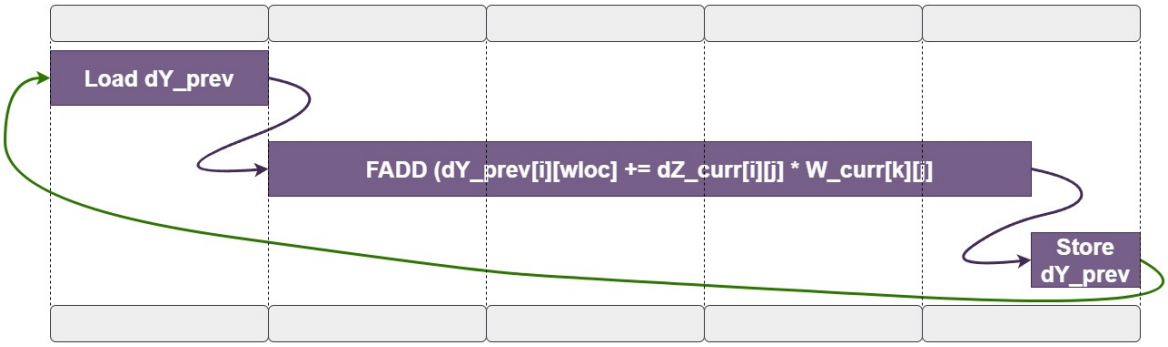

FIGURE 5.23: II Violation - Data dependency between the load and store operations on the array dY_prev

As a result of the II violation, the achieved initiation interval was increased to 5 instead of 1, which resulted in the design's total latency being significantly increased. To resolve this issue, we split the calculation of dY_prev into two individual (inner) loops of Batch_size (as shown in lines 17 and 21 of the algorithm 25). First, the calculations are performed and the results are stored in a temporary matrix (BRAM) independent of Wloc. The results of the temporary matrix are then appropriately stored in the regular dY_prev matrix through the use of Wloc. Figure 5.24 shows how HLS schedules dY_prev backpropagation operations (in green boxes). As shown in this figure, we exploit the unrolling of the third inner loop by running 16 (Batch_size) instances of each operation in parallel. To accomplish this, array partitioning was applied to the matrices dY_prev and dY_prev_tmp.

In layer 5, we can calculate dY_prev using the initial approach that we discussed (5.3), since this layer does not utilize mask, so there is no II violation. There is a peculiarity regarding the third layer. The third layer's mask allows the same input node to be assigned to more than one output node since its connections are random. The same is true (without the random nature of its connections) for layer 1, however, dY_prev does not need to be calculated in layer 1, as discussed earlier. As for the third layer, this means that Wloc_3 matrix does not take a unique value every time. Therefore, in this particular layer, line 23 (of algorithm 25) must be changed (5.4) to include the add operation that we attempted to avoid previously.

$$dY\_prev[i][Wloc] += dY\_prev\_tmp[i][k] \tag{5.4}$$

As a result, the II violation (5.23) we discussed above has been reinstated for this layer only. Layer 3 achieves II=5, meaning that new input data can

be processed every five clock cycles. Our efforts to resolve this issue were unsuccessful. Once this issue is resolved and the initiation interval is reduced in future work, the latency is expected to be significantly reduced.

Calculation of dZ_curr and dZ_batch

For layers 4 to 1, the dZ_curr and dZ_batch calculations are implemented in the same manner as shown in the HLS code (25). dZ_batch is calculated by dividing dZ_curr by Batch_size (16). In order to avoid the cost of division, we multiply by a value that has already been calculated (1/16). The purpose of using dZ_batch is to avoid performing the same multiplication more than once within each loop instance. Both dW_curr and db_curr require the same calculation. Figure 5.24 depicts the scheduling of dZ_curr (in orange boxes) and dZ_batch (in blue boxes) calculations. Similar to the dY_prev calculation, we can observe that 16 instances of each operation are executed in parallel along the batch, utilizing the unrolling of the third loop. As previously mentioned, softmax activation function is applied to the fifth layer. Hence, for the last layer only, the calculation of dZ_curr is performed as previously shown in equation 5.1 instead of lines 8 to 12 in the algorithm 25.

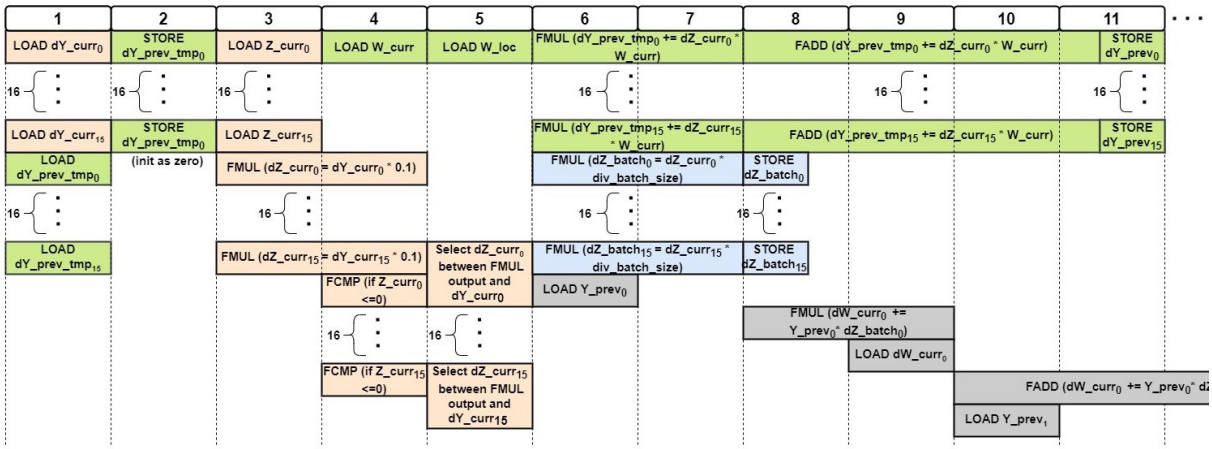

FIGURE 5.24: First Approach - Detailed schedule of dZ, dZ_batch and dY_prev backpropagation operations in HLS. The green boxes correspond to the dY_prev calculation, orange boxes to the dZ_curr calculation, blue boxes to the dZ_batch calculation, and gray boxes will be discussed in detail next.

Calculation of dW_curr

Since the unrolling of the third inner for-loop involves batches, we are not able to run 16 instances of each operation of dW in parallel, as with dZ and

dY. This occurs because we need to perform Batch_size iterations in order to calculate an individual value for dW. Due to this, we parallelize the operations in each step, including Load, FMUL, and FADD. The dW calculation is described in the HLS code (in line 16 of algorithm 25). In figure 5.25, the initial phase of the dW calculation scheduling is illustrated. Continuing the scheduling process, the operations shown in this figure are repeated in the same manner. The last part of dw scheduling is shown in figure 5.26. Due to the way this calculation is implemented, it requires a high number of iteration latency cycles to complete. Attempts were made to reduce latency by implementing it in a separate structure of nested loops. Despite our efforts, we were not able to improve the latency of the backpropagation process in total.

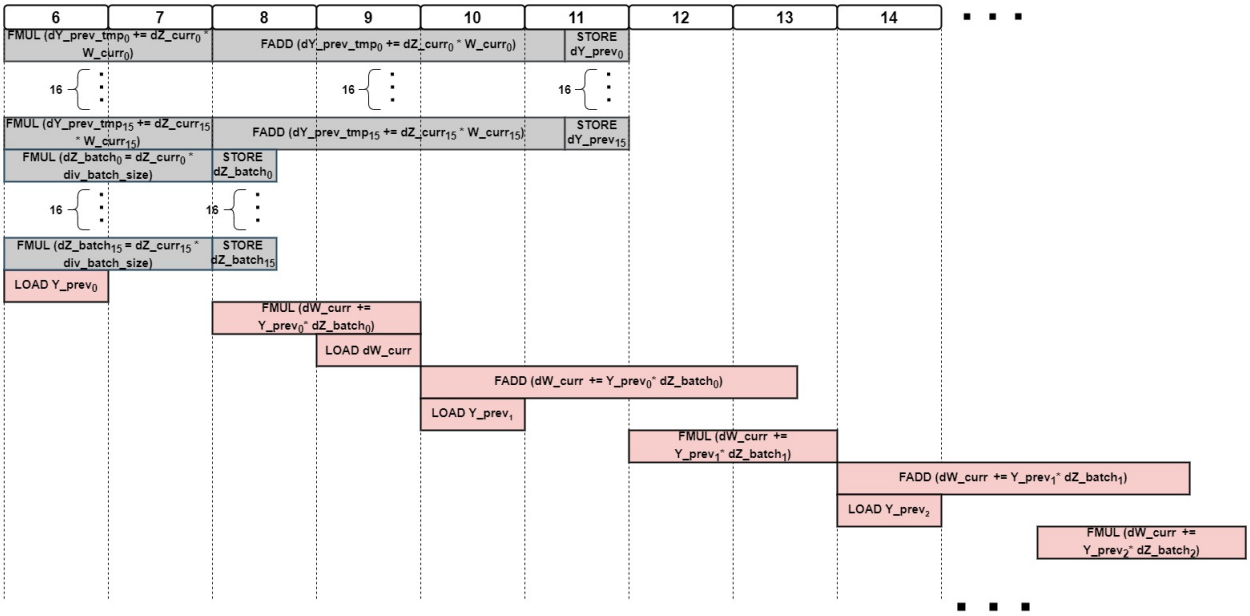

FIGURE 5.25: First Approach - Initial phase of scheduling dW backpropagation in HLS. The red boxes correspond to the dW calculation and gray boxes refer to previously discussed calculations.

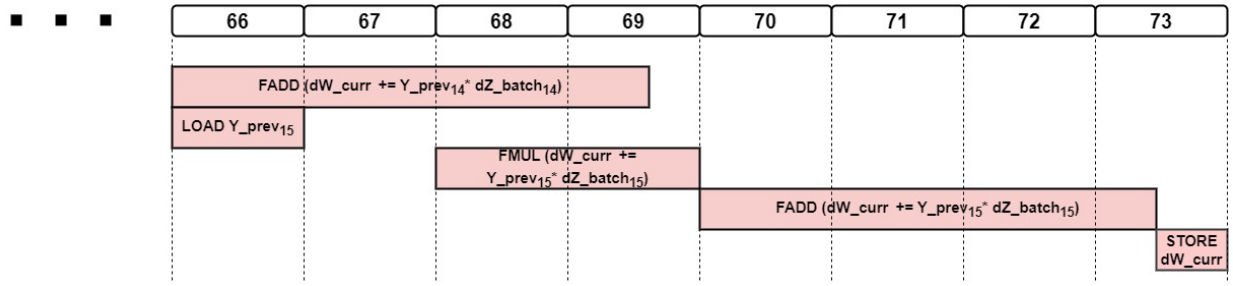

FIGURE 5.26: First Approach - Last phase of scheduling dW backpropagation in HLS.

Algorithm 6 First Approach - Single layer Backpropagation in Vivado HLS

```
1: procedure SINGLE_LAYER_BACKPROPAGATION_HLS
2:     for j ← 0 to Output_nodes do                                  ▷ Units
3:         for k ← 0 to Synapses do
4:             Wloc=W_curr_loc[k][j]
5:             Wcurr=W_curr_value[k][j]
6: # pragma HLS PIPELINE II=1
7:             for i ← 0 to Batch_size do
8:                 if Z_curr[i][j] ≤0 then                ▷ backward Leaky ReLU
9:                     dZ_curr[i][j]=dY_curr[i][j]∗0.1
10:                else
11:                    dZ_curr[i][j]=dY_curr[i][j]
12:                end if
13:
14:                ▷ (div_batch=1/Batch_Size)
15:                dZ_curr_batch[i][j] = dZ_cur[i][j] ∗ div_batch
16:                dW_curr[k][j]+ = Y_prev[i][Wloc]*dZ_curr_batch[i][j]
17:                dY_prev_tmp[i][k]+ = dZ_curr[i][j] ∗ Wcurr
18:            end for
19:
20:            for i ← 0 to Batch_size do
21:                dY_prev[i][Wloc]=dY_prev_tmp[i][k]
22:                dY_prev_tmp[i][k]=0
23:            end for
24:        end for
25:    end for
26: end procedure
    =0
```

Calculation of db_curr and update of b_curr

Despite our efforts, we were unable to efficiently integrate the db_curr calculation with the other backpropagation calculations within the three nested for-loops described above. This resulted in our decision to carry out this calculation separately. The db calculation is performed along with the update of b_curr. As discussed in the previous section (5.5), biases are updated using the classical method (not Adam). Since we do not require the dimension of synapses for these calculations, two nested for-loops are used, as shown in the HLS code (8).

Algorithm 7 First Approach - Single layer Bias update in Vivado HLS

```
1: procedure SINGLE_LAYER_BIAS_UPDATE_HLS
2:     for j ← 0 to Output_nodes do                          ▷ Units
3: # pragma HLS PIPELINE II=1
4:         for i ← 0 to Batch_size do
5:             db_curr[j] += dZ_curr_batch[i][j]
6:         end for
7:         b_curr[j] = b_curr[j] − db_curr[j] ∗ learning_rate
8:     end for
9: end procedure
```

The pipeline directive (5.2.1) is applied to the first for-loop to enable the concurrent execution of operations within this loop with II=1. So, the second inner for-loop is unrolled. However, we are not able to exploit parallelism along the batches. It is because we need Batch_size iterations in order to calculate an individual value (db), instead of calculating a different value (for example, db_1, db_2, etc.) for each Batch_size iteration (or loop instance). It is a similar case to that of dW. Below (5.27) is the schedule of db of backpropagation and bias update operations in HLS.

Due to the fact that the second inner for-loop is unrolled, the number of iterations required for the db and b calculation from each layer is equal to the number of Output_nodes (units). Theoretically, this is a small number, particularly when considering the pipelined nature of the first loop, with II=1, which allows new input data to be processed every clock cycle. Nevertheless, the iteration latency for each layer is 71 cycles, which is a high number given that it is implemented separately from the rest of the backpropagation process (dZ, dZ_batch, dW calculations).

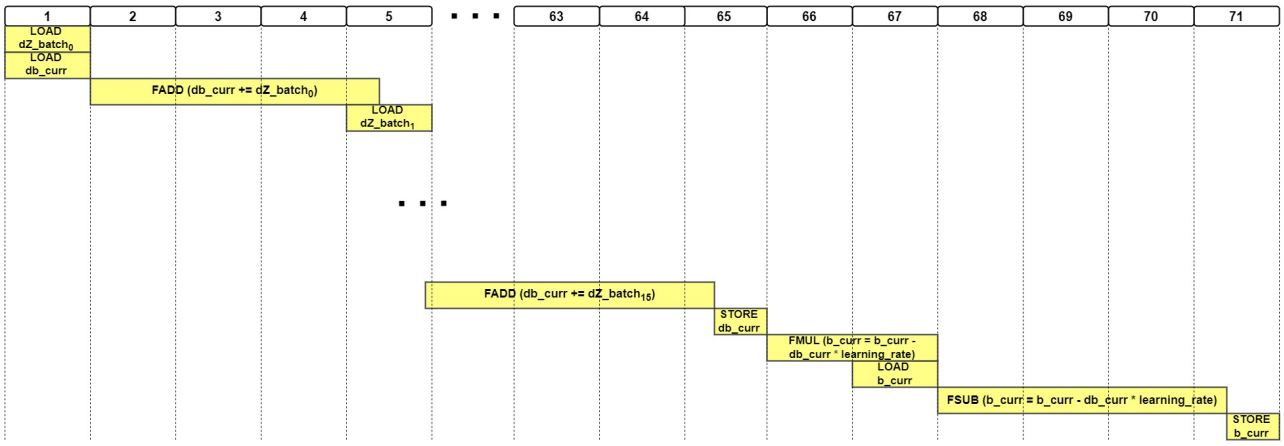

FIGURE 5.27: First Approach - A detailed schedule of db (of backpropagation) and bias update operations in HLS.

5.6.2 Update - Adam Optimization Algorithm in Vivado HLS

Initially, we decided to use the gradient descent update method (2.30) instead of the Adam algorithm for updating weights. The purpose of this was to check the currently implemented design more quickly. Our results indicated that the classical update method produced disappointing results in terms of training error and accuracy. The Adam algorithm, as described in section 4.3.5, is computationally complex, thus requiring considerable hardware resources. In spite of the complexity of the Adam, it has proven to be an effective and reliable method of updating parameters. Consequently, we had to ensure that the parallelization level of its HLS implementation does not exceed the resources of ZCU102.

In this first approach, we focused on creating a functional Adam implementation in HLS. Each Adam single layer consists of two nested for-loops for Output_nodes and Synapses, as shown in algorithm 16. The pipeline directive (5.2.1) is applied to the second inner for-loop with II=1. So, each iteration of this loop can begin within a clock cycle at a time. The total number of iterations is equal to Output_nodes * Synapses. This is a high number considering backpropagation with three nested for-loops (instead of two) requires the same number of iterations. In regards to the iteration latency for each layer, it is equal to 101 clock cycles. As a result, Adam has a high total latency for each layer.

In order to minimize the number of iterations, we need to apply pipeline directive to the first for-loop (instead of the second), resulting in the unrolling of the second inner for-loop. By doing so, the first problem that arises is

a memory dependency (5.6.1). As mentioned earlier, this issue can be resolved by applying the array partitioning directive. However, partitioning the dW_curr and W_curr matrices affects backpropagation implementation, by causing data dependency in the dW calculation (line 16 in algorithm 25). Figure 5.23 illustrates a similar case (dY_prev) of data dependency that was discussed in detail in section 5.6.1. In addition, a timing violation occurred (in backpropagation) which refers to a path of operations requiring more time than the available clock cycle. Thus, we kept the Adam implementation as presented in 16 (the pipeline applied to the first for-loop).

As part of Adam, square roots, divisions, and powers of numbers are calculated. Since the Adam algorithm calculations are complex, we have elected to use double precision operations (for this algorithm only). Double precision operations provide us with more accurate results. The resource usage would have been significantly increased with them if higher levels of parallelization had been achieved. Nevertheless, we are not experiencing this issue in this case since we have low parallelization and no loop unrolling.

Algorithm 8 First Approach - Single layer Adam algorithm Update method in Vivado HLS

1: **procedure** SINGLE_LAYER_ADAM_ALGORITH_UPDATE_HLS
2: **for** $j \leftarrow 0$ to $Output\_nodes$ **do** ▷ Units
3: **for** $k \leftarrow 0$ to $Synapses$ **do**
4: # pragma HLS PIPELINE II=1
5: $mean\_dw\_curr[k][j]$=0.9*$mean\_dw\_curr[k][j]$+0.1*$dW\_curr[k][j]$
6: $uvar\_dw\_tmp[k][j]$=0.999 $*$ $uvar\_dw\_curr[k][j]$+0.001*$dW\_curr[k][j]$*$dW\_curr[k][j]$
7: **if** ($uvar\_dw\_tmp[k][j] > uvar\_dw\_curr[k][j]$) **then**
8: $uvar\_dw\_curr[k][j]$=$uvar\_dw\_tmp[k][j]$
9: **end if**
10: ▷ t represents the current number of training iterations
11: $mean\_dw\_corr[k][j]$=$mean\_dw\_curr[k][j]$/(1-$pow(0.9, t)$)
12: $uvar\_dw\_corr[k][j]$=$uvar\_dw\_curr[k][j]$/(1-$pow(0.999, t)$)
13: $dW\_adam\_curr[k][j]$=$learning\_rate$*($mean\_dw\_corr[k][j]$/(hls :: $sqrt(uvar\_dw\_corr[k][j])$+0.00000001)]
14: $W\_curr[k][j]$=$W\_curr[k][j]$-$dW\_adam\_curr[k][j]$
15: **end for**
16: **end for**
17: **end procedure**

5.6.3 Forward propagation in Vivado HLS

The forward process is implemented in the thesis of Nikoletta Palatiana, as mentioned previously. Each forward's layer requires Synapses * Output_nodes pipelined iterations, which are started one at a time in each clock cycle (since II=1). A similar example of pipelining can be seen in figure 5.22. These iterations are parallelizable along the Batches due to loop unrolling. Iteration latency is 17 clock cycles. Therefore, the forward process is highly parallelized.

5.6.4 Design Space Exploration

In this section, we examine different approaches to designing based on the type of data. Comparisons are conducted in terms of the resources required, the execution time, and the precision of the results (error and accuracy in training and validation). Actually, we had to choose between floating point and fixed point calculations. Fixed point refers to the representation of numbers with a fixed number of digits after and before the decimal point. On the other hand, a floating-point representation allows the decimal point to be positioned relative to the significant digits of the number.

The initial decision was to select fixed-point over floating-point as the data type. It should be noted that the Adam algorithm was implemented using floating-point calculations as the only exception. This is due to the fact that Adam involves complex calculations, such as divisions and square roots, which must be performed with a sufficient level of precision. However, with this initial fixed point approach, the values in the design exceeded the limits defined by the fixed point declarations after numerous training iterations, resulting in incorrect training. As the range of numbers accepted in the fixed-point declaration increases, the precision of the calculations decreases. Despite our efforts, we were unable to properly declare fixed-points in order to ensure that the training would produce the desired results (in terms of error/accuracy). In addition, we observed an overflow in LUT usage, requiring more than was available (in ZCU 102).

As a second approach, fixed-point data types were primarily used for forward propagation, more specifically for weights, biases and inputs. For the rest of the data, we chose the floating-point data type. By doing so, we were able to achieve adequate training results (in terms of error/accuracy). Moreover, this prevents the overflow of LUTs. By choosing floating-point as the

data type for all the data, we were able to perform the training appropriately, achieving better training/validation results compared to the second approach. A further advantage of this float approach is that it uses approximately 37% fewer LUTs, 3% fewer DSPs, and 3% fewer FFs than the second fixed-point approach. One insignificant disadvantage to this approach is that the previous second fixed point approach executed 1,0003 times faster than this one. As a consequence, we decided to select the fully floating-point approach since it is more efficient in terms of training/validation results and requires fewer resources. As far as the double data type approach is concerned, the resources of the ZCU 102 were insufficient to support it.

5.7 Implementation of IP Block (HLS) - Second (Optimized) Approach

The second approach of implementing the training IP in HLS was aimed at optimizing the latency of the design. Upon analyzing the utilization of FPGA resources in our first approach, it became evident that there was room for improvement. In particular, we focused on optimizing the Adam algorithm and the db calculation along with bias update, as we identified these cases as an opportunity to optimize the design.

As discussed previously in section 5.6.2, we encountered several problems when applying array partitioning to W_curr and dW_curr matrices (BRAMs). The partitioning of these matrices would reduce the latency in Adam's implementation efficiently. Despite this, the problems that prevented us from achieving it occurred in the implementation of backpropagation rather than in Adam.

5.7.1 Optimized Backpropagation in Vivado HLS

Calculation of dW_curr

In the first backpropagation approach (5.6.1), dW was not partitioned. The current dW value was loaded once at the beginning of each iteration and stored at the end after the calculations were performed. In order to optimize the Adam algorithm, it was necessary to partition this matrix (BRAM) and resolve the aforementioned issues (5.6.2). The partitioning of dW results in multiple memory instances of it. This means that rather than loading, calculating, and storing the current iteration's dW value in each iteration, all

partitions of dW are loaded to calculate the corresponding value, and then all partitions are appropriately stored. As a consequence, there was a data dependency between the load and store operations on the matrix dW (as seen in a similar case in figure 5.23). In essence, the current iteration utilizes the dW value generated by an earlier iteration, which requires N cycles instead of one to generate that value. Thus, we were unable to exploit the II=1 of pipelining (5.2.1) and run the iteration instances in succession within a clock cycle.

With this optimized backpropagation implementation, we have resolved these problems. In order to achieve this, it is necessary to avoid dW loads. By using a temporary array and a temporary variable, we perform the required addition and multiplication operations of dW (lines 16 and 25 in algorithm 33). At the end of each iteration, the calculated value is stored in the dW matrix. We must, however, store all partitions of dW in each iteration due to the partitioning mentioned previously. Unfortunately, we were unable to avoid that issue, but it does not cause any dependencies. The updated HLS backpropagation algorithm is presented in 33. The detailed schedule of dW backpropagation operations is illustrated in Figures 5.28 (in initial cycles) and 5.29 (in last cycles).

Calculation of dY_prev, dZ_curr and dZ_batch

As in the first approach, the implementation of dY_prev (5.6.1) and dZ_curr (5.6.1) calculations is the same (as illustrated previously in figure 5.24. Regarding dZ_batch, in this approach it is calculated gradually as shown in figures 5.28 and 5.29 (blue boxes).

Calculation of db_curr and update of b_curr

In the first approach (5.6.1), the calculation of db_curr and the updating of b_curr were implemented in a separate structure of two nested for-loops for each layer. We were unable to integrate them into the structure of other backpropagation operations (dW_curr, dZ_curr, dZ_batch, dY_prev). As a result, our design was affected by additional latency.

According to the second approach, the db_curr calculation is combined with the other backpropagation calculations within the three nested for-loops (as shown in updated algorithm 33). In order to accomplish this, a temporary

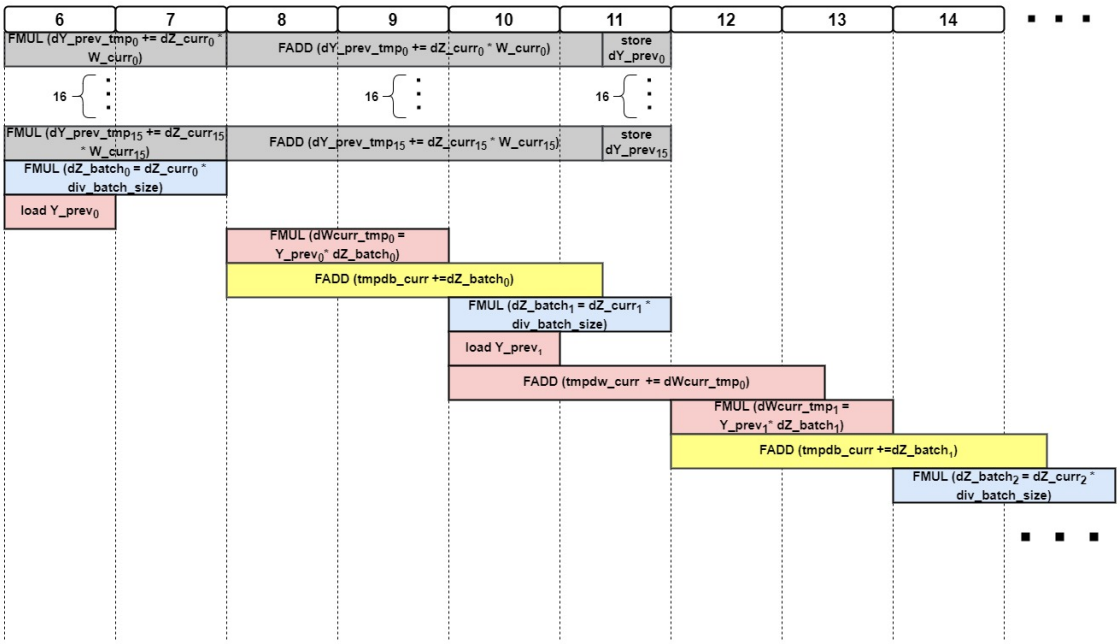

FIGURE 5.28: Second Approach - Detailed schedule of dW and db backpropagation operations in HLS. This figure illustrates how these operations are analyzed in initial clock cycles. The scheduling continues in the same manner as shown here. The red boxes correspond to the dW_curr calculation, yellow boxes to the db_curr calculation, blue boxes to the dZ_batch calculation and gray boxes refer to previously discussed calculations.

variable is used for the multiple addition operations (of db) during the iteration, similarly to how dW is implemented. Upon completion of these calculations, the result is stored appropriately in the db_curr matrix (BRAM) and the b_curr value is updated. The schedule of the above calculations is shown in figures 5.28 (for the initial cycles) and 5.29 (for the last cycles).

These changes increase the iteration latency of the total backpropagation implementation from 73 to 77. Nevertheless, this increase is insignificant since all backpropagation calculations are carried out within one structure of three nested for-loops (for each layer of our NN). This structure was described in section 5.6.1. In this way, Output_nodes pipelined iterations per layer of 71 iteration latency (from first approach) are eliminated, leading to a reduction in latency for our design. Consequently, the total optimized backpropagation schedule is formed as shown in figure 5.30.

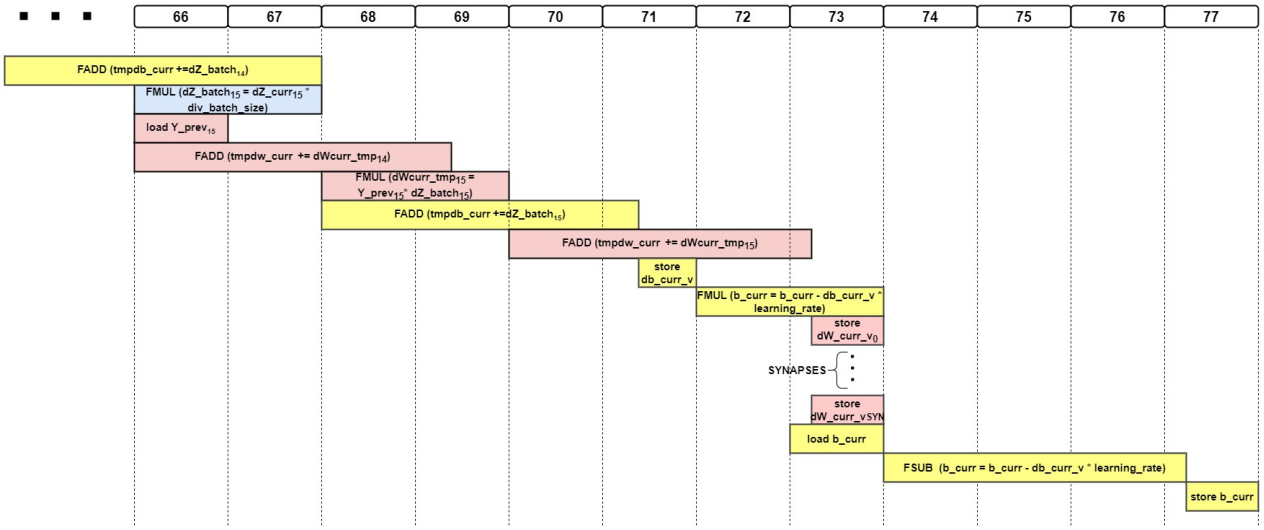

FIGURE 5.29: Second Approach - Detailed schedule of dW and db backpropagation operations in HLS. This figure illustrates how these operations are analyzed in last clock cycles. The red boxes correspond to the dW_curr calculation, yellow boxes to the db_curr calculation and blue boxes to the dZ_batch calculation.

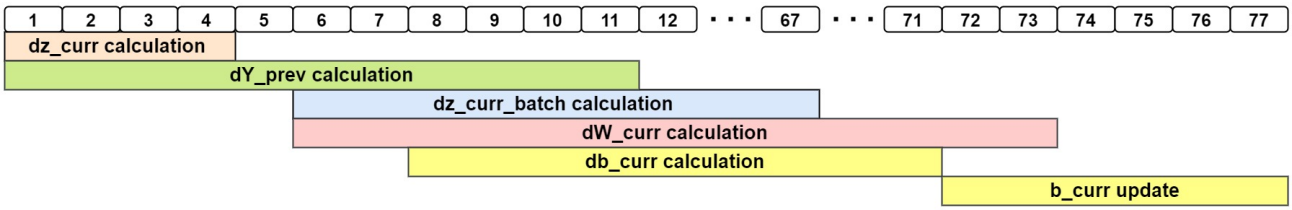

FIGURE 5.30: Second Approach - Schedule of the main backpropagation calculations in HLS. The number 77 represents the iteration latency (in cycles).

5.7.2 Optimized Update - Adam Optimization Algorithm in Vivado HLS

Since we have resolved the problems encountered when applying array partitioning to W_curr and dW_curr, we are able to significantly reduce the latency of Adam's implementation. This is achieved by applying the pipeline directive with II=1 to the first for-loop of Output_nodes rather than the second inner for-loop of Synapses (as in the first approach). For-loops are structured as follows:

for $j \leftarrow 0$ `to` *Output_nodes* **do**

```
# pragma HLS PIPELINE II=1
for k ← 0 to Synapses do
    <do calculations>
```

In this way, each iteration of the first for-loop is started one at a time in each clock cycle (since II=1). A similar case of pipelining was illustrated in figure 5.22 earlier. When the first for-loop is pipelined, the second (inner) for-loop is automatically unrolled, creating Synapses loop instances that can be executed in parallel. As a result, each layer requires Output_nodes pipelined iterations that are parallelizable along the Synapses. In order to exploit parallelization, we must partition W_curr and dW_curr matrices (BRAMs) on their first dimension, which corresponds to Synapses. By doing so, we are able to perform more than two memory accesses per clock cycle and execute all (Synapses) loop instances of each operation simultaneously along the Synapses. Based on the first approach, the total number of iterations needed for each layer was equal to Output_node * Synapses. In the second approach, only Output_node iterations are required (for each layer), resulting in a significant reduction in latency. Figure 5.31 illustrates the optimized Adam algorithm schedule for its main calculations. This is essentially how each pipelined iteration executes these calculations.The iteration latency is 53 clock cycles, a significant improvement over the first approach, which was 101 clock cycles. The figure shows only the parallelization among the main calculations, without illustrating the parallelization along the Synapses for each individual calculation.

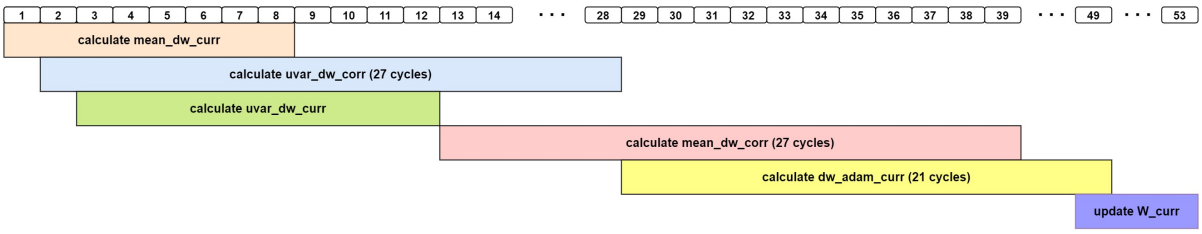

FIGURE 5.31: Second Approach - Schedule of the main Adam optimization algorithm (Update) calculations in HLS. The number 53 represents the iteration latency (in cycles).

Adam algorithm calculations were presented in the HLS code of the first approach (in 5.6.2). Besides the manner in which the pipeline directive is applied, another important difference between the first and second approaches

relates to the preference for float operations over double operations. The design is generally implemented using float data types. Due to the complexity of the calculations in the Adam algorithm, including computations of square roots, divisions, and powers of numbers, we chose double precision operations in the Adam's first approach in order to obtain more accurate results. However, since this approach increases parallelization, power consumption and resource usage have increased significantly. As a result, it is necessary to select float operations. In making Adam's calculations with float precision, we observe that there is no significant difference between the two approaches in terms of accuracy and error. Furthermore, we are able to save a substantial amount of resources and power in this way.

Chapter 6

Results

In this chapter, we present the results of our work. The first step will be to provide specifications of the CPU and GPU that will be used for the comparisons. As a next step, we present the resource utilization of our first and second architecture approaches implemented on the ZCU 102 FPGA. Following this, we analyze the performance metrics used for the comparisons. Lastly, we compare the results of our two FPGA-based architecture approaches, as well as the results of the preferred FPGA-based architecture with CPU and GPU implementations of our bio-inspired ANN (in Keras).

6.1 Specification of Compared Platforms

6.1.1 Intel i5-6500

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: Intel i5-6500 Specifications

Intel i5-6500	
Total Cores	4
Total Threads	4
Processor Base Frequency	3.20 GHz
Max Turbo Frequency	3.60 GHz
TDP	65 W
Max Memory Bandwidth	34.1 GB/s
Lithography	14 nm

Thermal Design Power (TDP) represents the average power, in watts, the processor dissipates when operating at Base Frequency with all cores active under an Intel-defined, high-complexity workload.

6.1.2 GPU

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. Below is a table 6.2 with GPU platform specifications.

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 Proposed Architectures

FPGAs are integrated circuits consisting of programmable logic blocks that can be configured to perform different logic functions. In comparison to GPUs, FPGAs are considered to be more energy/power efficient. In addition, they provide a high level of parallelism. It is important to note that FPGA chips do not have hard-etched circuitry and can be reprogrammed as required. As a result of their reconfigurability, FPGAs are ideal for applications in which standards are continually evolving. Moreover, FPGAs are available in a variety of sizes, so designers can choose the one that is most suitable for their application.

The purpose of this section is to present the final resource utilization of our two versions of FPGA-based architecture that have been ported to ZCU-102 FPGA (in table 6.3).

TABLE 6.3: Comparison of the first and second versions of the FPGA-based architecture (ZCU 102) - Resources Utilization

	Version 1	**Version 2**
Clock Frequency	125 MHz	125 MHz
BRAM Usage (%)	**59%**	62%
DSPs Usage (%)	22%	**19%**
FF Usage (%)	**14%**	21%
LUTs Usage (%)	**37%**	55%

6.2 Performance Metrics

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. According to this thesis, throughput is defined as the number of batches trained per second.

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

where a batch consists of 16 images.

6.2.3 Power Consumption

Power consumption is defined as the amount of energy consumed per unit time to perform a specific task. It is usually measured in Watts (W) or kilo-Watts (kW). The power consumption of a system is extremely important and should be kept as low as possible. The battery life of portable electronic devices such as cell phones and laptops is limited by power consumption. Low power consumption leads to higher energy efficiency and lower building costs. Using a simplified and smaller architecture for a design can increase energy efficiency.

6.2.4 Energy Consumption

Energy consumption (6.2) refers to the energy required for accomplishing a particular task in a specific amount of time. It is commonly measured using Joule (J) or kiloJoule (kJ). This metric value should also remain 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:

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

6.3 Performance Evaluation and Comparison

This section compares the performance of our FPGA-based implementation to CPU/GPU implementations in the performance metrics discussed above. We have implemented two versions of the training procedure for the FPGA-based bio-inspired ANN in ZCU 102. As the software implementation for the comparison between FPGA and CPU/GPU, we choose the high-level Keras implementation developed by S. Chavlis. This decision was made due to the fact that our software bio-inspired ANN implementation in Numpy underperforms when compared with the Keras implementation.For training, we utilize the MNIST database of handwritten digits, as described in section 4.2.1.

In section 4.2.1, we discussed the training settings associated with the software implementation. It is important to note that in the FPGA implementation, callbacks and shuffling of training data are not included. In addition, the validation procedure has not been implemented. As a part of the thesis of Nikoletta Palatiana, the validation procedure as well as callbacks will be implemented. Due to the absence of these features, we are not able to conduct the training procedure in an optimal manner. By incorporating these features, we will be able to prevent overfitting and reduce the learning rate when the validation error has stopped improving. This will enable us to reduce the validation error that has been achieved. As far as the remaining settings of software training are concerned, they are applied in a similar manner to software implementation. Note that we process a batch of 16 images

in each training iteration. Moreover, we perform 30 epochs of training on the entire MNIST dataset.

There is a problem encountered when running our design on the ZCU 102 evaluation board. This problem is related to the reading of the MNIST dataset files from the SD card of the board. There is an extremely slow reading of data from the SD card. Our design runs on the ZCU 102 board through a server and the MNIST files are too large. These may be the causes of this problem. As a result of this issue, we are unable to actually train the entire MNIST dataset on the ZCU 102 board, but are only able to measure the training time. For the purpose of calculating the epoch time, we processed the same batch of 16 images 3000 times, which is equal to the total number of batches used in MNIST training. Each training iteration involved passing data into memory and passing data out of memory in order to measure time appropriately. To determine the latency of our design, we measured the time it takes to perform training on a single batch. Training results in terms of accuracy/error are derived from the simulation level (Vivado HLS). As part of the simulation level, we also implemented the validation process (without callbacks) to provide us with the results of validation error/accuracy. To verify the functionality of our design, we ran the training procedure using a few hundreds of MNIST batches (rather than the entire MNIST dataset) on the ZCU 102 board and at the simulation level (Vivado HLS). Both cases produced the same results. Therefore, we believe that if the issue with reading the dataset were resolved, we would be able to perform training without any problems on ZCU 102 board. We could resolve this issue by executing our design directly on the board rather than through the server. Alternately, we may be able to resolve this issue by using a more efficient method of reading the dataset. Unfortunately, due to a limited time frame, we were unable to resolve this issue.

Table 6.4 summarizes the performance analysis of our bio-inspired ANN by comparing two versions of its FPGA-based architecture running on ZCU 102 and its Keras implementation running on both CPU and GPU.

TABLE 6.4: Performance Evaluation and Comparison - Two versions of 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 (Keras)	GPU (Keras)	FPGA V.1	FPGA V.2
Clock Frequency (MHz)	3600	1392	125	125
Throughput (Batches/s)	81.08	176.47	1029.16	**1260.66**
Throughput Speedup	1x	2.176x	12.693x	**15.548x**
Latency (ms)	13	6	1.08	**0.904**
Latency Speedup	1x	2.16x	12.04x	**14.38x**
Epoch execution time (s)	37	17	2.915	**2.3797**
Total On-Chip Power (Watt)	65	75	**7.407**	8.815
Power Efficiency	1x	0.866x	**8.775**	7.3738x
Energy Consumption (Joule) per Batch	0.845	0.45	0.008	**0.00796**
Energy Efficiency	1x	1.877x	105.62x	**106.15x**
Images/Joule	1.247	2.353	138.94	**143.01**

6.3.1 Comparison of two FPGA versions

Comparing the two approaches of our FPGA-based architecture, we observe (from table 6.4) that the second approach achieves lower latency (fig. 6.1) and higher throughput (fig. 6.2) than the first one, resulting in a faster design. This is also evident in the training execution time for an epoch (fig. 6.3), which takes only 2.3797 seconds as opposed to 2.915 seconds for the first approach. A downside of the second approach is that it consumes more ZCU 102 resources, with the exception of DSP resources, as shown in table 6.3. This leads to the second approach being less power-efficient than the first (fig. 6.4), requiring 8.815 Watts versus 7.407 Watts. However, by comparing the energy consumption per batch, we observe that the second approach is slightly more energy efficient (0.00796 Joule against 0.008 Joule as shown in fig. 6.5). In terms of training and validation error(fig. 6.8)/accuracy(fig. 6.7), these two approaches yield the same results. Therefore, we prefer the second approach over the first because it is more time and energy efficient.

6.3.2 Comparison of FPGA and CPU/GPU versions

A comparison is made here between our FPGA-based architecture and Keras implementations running on CPU and GPU. Table 6.4 presents a summary of their performance analysis. In the comparisons below, we refer to our (preferred) second FPGA approach as the proposed architecture and we use the CPU implementation in Keras as the reference implementation. There is a significant improvement in latency (fig. 6.1) and throughput (fig. 6.2) with our proposed architecture, with speedups of 14.38x and 15.548x respectively. The GPU implementation also performs better on both these metrics when compared to the CPU implementation, but with only a 2x speedup (approximately).

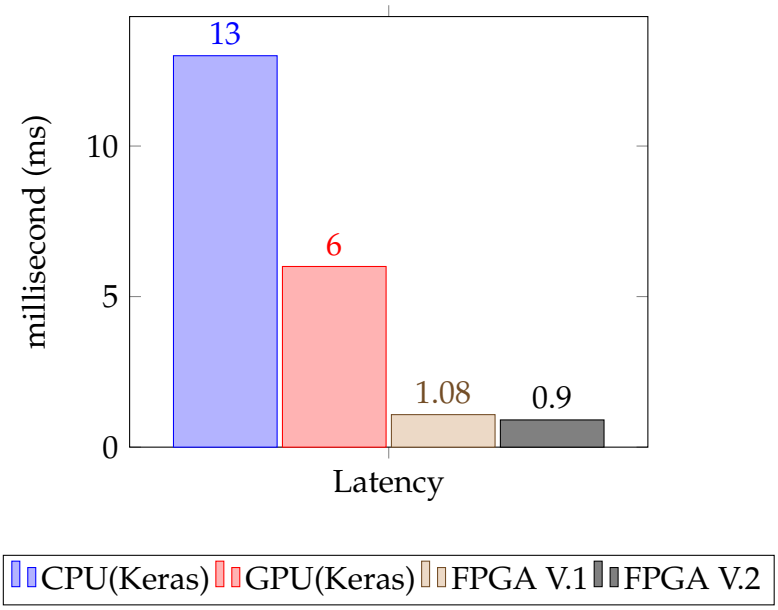

FIGURE 6.1: An analysis of the Latency of the compared platforms.

As for training time for an epoch (fig.6.3) for the entire MNIST dataset, our proposed design completes the task in 2.3797 seconds, compared to 37 seconds for the CPU and 17 seconds for the GPU.

The proposed architecture is also more efficient in terms of power consumption (fig. 6.4), requiring only 8.815 Watts as opposed to 65 Watts for the CPU implementation and 75 Watts for the GPU implementation. However, the most noteworthy aspect of our FPGA design is its energy efficiency (6.5), which is 106.15 times greater than that of CPUs and 56.5 times greater than that of GPUs.

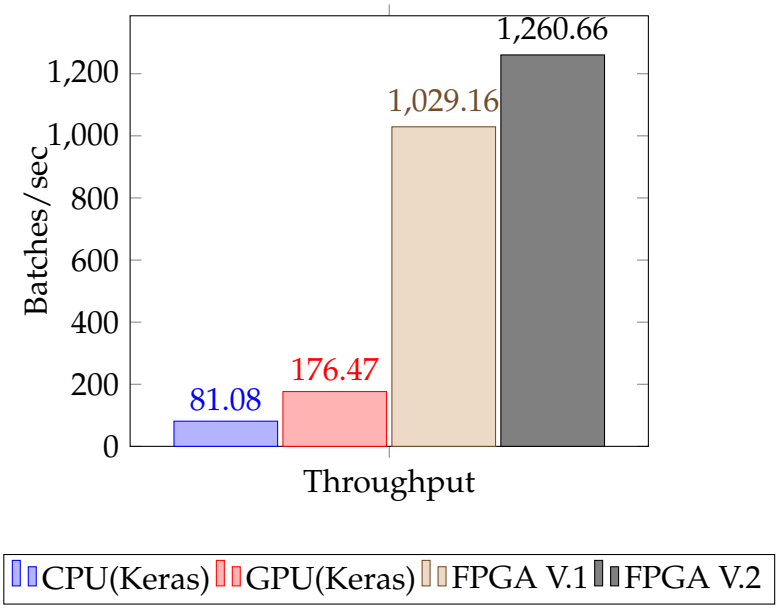

FIGURE 6.2: An analysis of the Throughput of the compared platforms.

As far as training and validation error/accuracy (fig. 6.8, 6.7) are concerned, our proposed architecture failed to match the performance of the CPU/GPU implementation. In spite of the closeness in training error/accuracy results between our FPGA implementation and Keras implementation, validation results are more accurate in order to establish the actual gap between them. According to these results, we achieve 95.5% validation accuracy compared to 97.5% for CPU/GPU implementations and 15.5% validation error compared to 9% and 10% for CPU and GPU, respectively. Nevertheless, given the high level of acceleration and energy efficiency provided by the FPGA implementation, the difference between these implementations in validation results is considered insignificant.

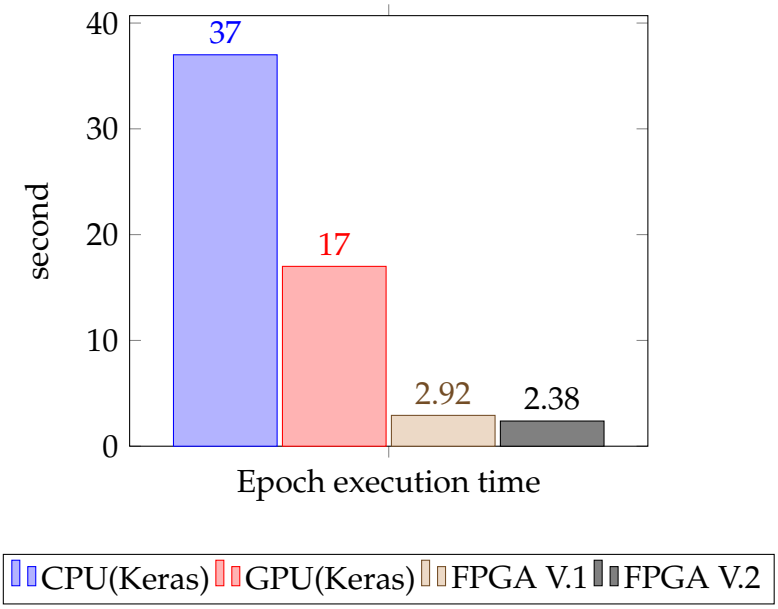

FIGURE 6.3: An analysis of the training execution time for an epoch of the compared platforms.

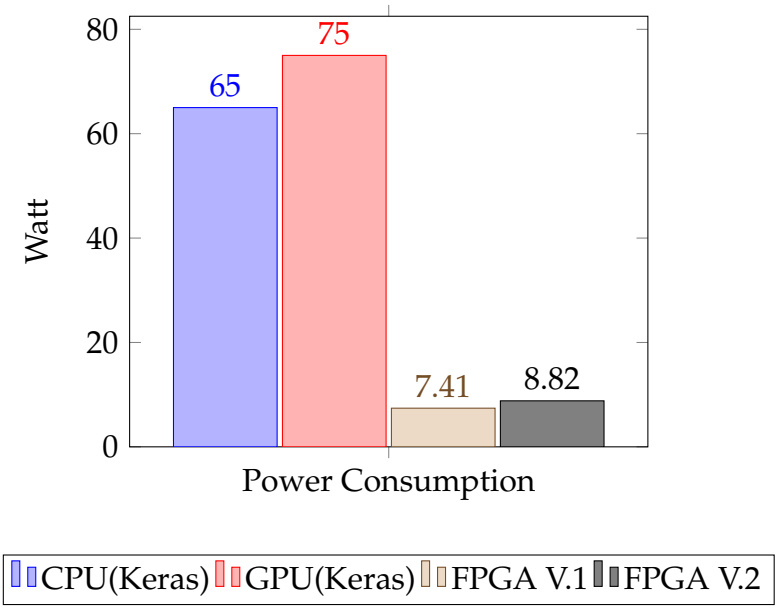

FIGURE 6.4: An analysis of the Power Consumption of the compared platforms.

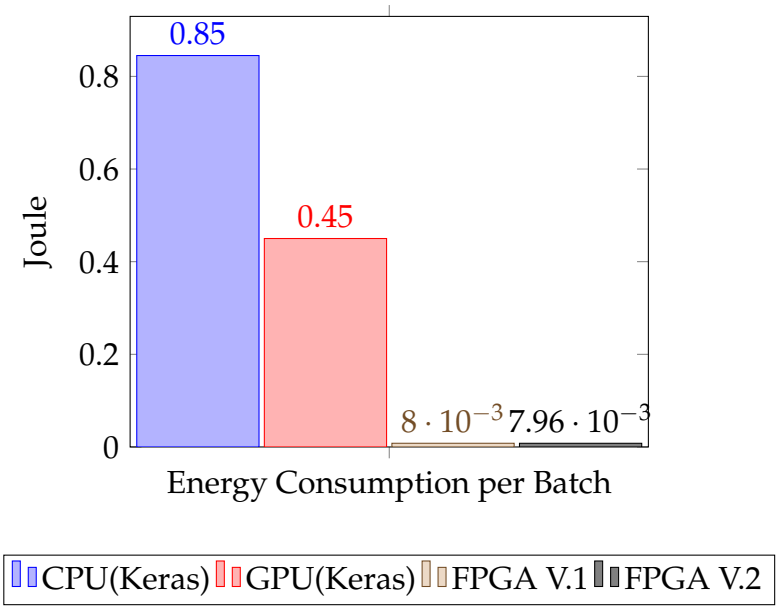

FIGURE 6.5: An analysis of the Energy Consumption per batch of the compared platforms.

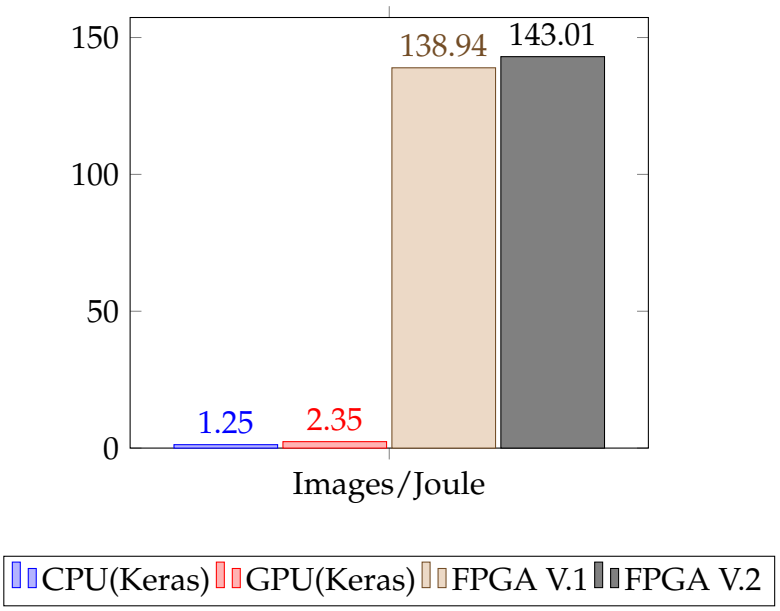

FIGURE 6.6: An analysis of the Images/Joule metric of the compared platforms.

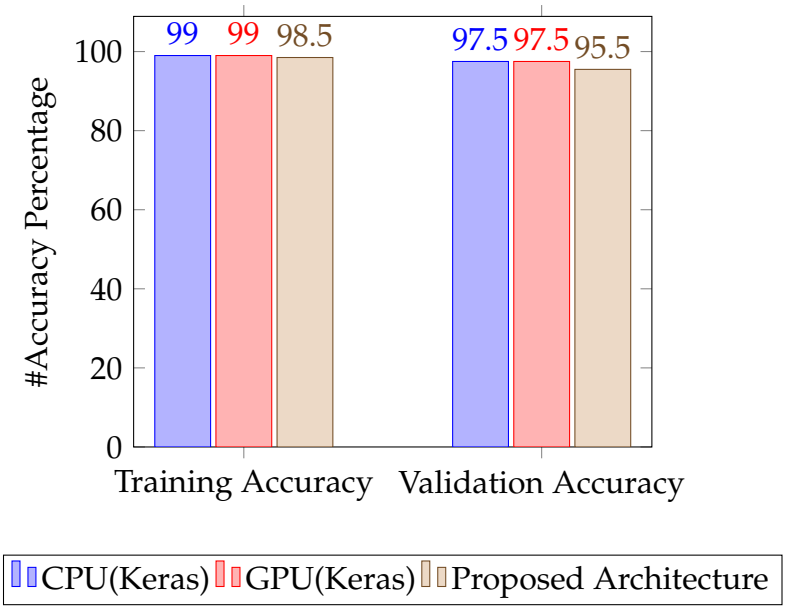

FIGURE 6.7: An analysis of the Accuracy in training and validation of the compared platforms.

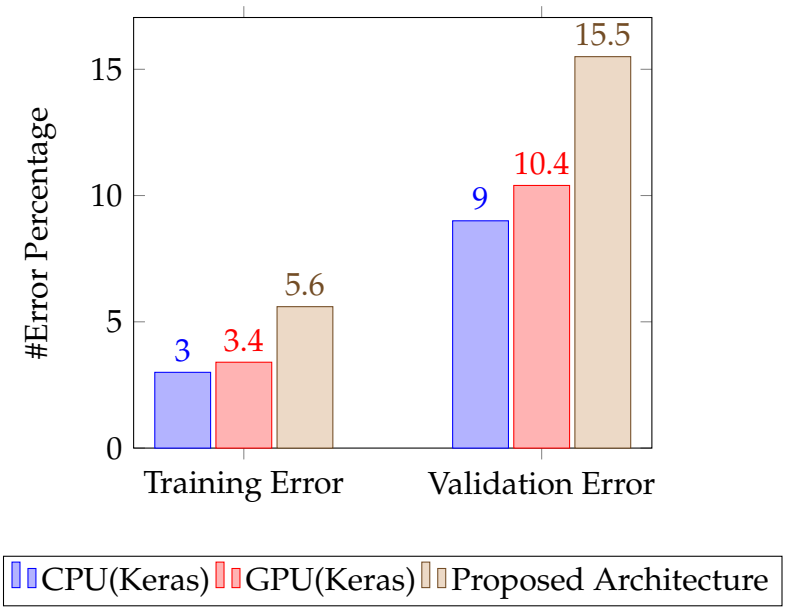

FIGURE 6.8: An analysis of the Error in training and validation of the compared platforms.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

ANNs have been successfully used to solve a wide range of challenging machine learning tasks. However, ANNs require a substantial amount of energy to achieve top performance. In addition, they suffer from problems that seem rudimentary to a human brain, including "transfer learning" and "catastrophic forgetting". In contrast, the brain consumes a very low amount of energy (< 20 watts), generalizes extremely well, and can learn continuously without erasing previously learned information upon learning new information. Although ANNs are generally derived from biology, they are biologically inaccurate. As a result of these facts, there is a growing interest in understanding biological NNs in greater depth and developing bio-inspired NNs such as SNNs.

Drawing inspiration from biological dendrites and the previously mentioned facts, the Postdoctoral Researcher S. Chavlis and the Research Director P. Poirazi, both from the Poirazi lab of the IMBB-FORTH, introduced a bio-inspired ANN architecture. This model enhances the bio-inspiration of a typical ANN by adopting dendritic-structure, receptive field, and the Adam optimization algorithm. This thesis aimed to build a FPGA-based implementation of this bio-inspired ANN training process to enhance its energy/power efficiency and speed up the training process. FPGAs are able to achieve this due to their high parallelism and power efficiency. Based on their initial high-level implementation in Keras, we first developed a lower-level implementation in Numpy. As a result of Numpy's implementation, we were able to analyze and understand this bio-inspired ANN model more deeply. The training procedure was then implemented using Vivado HLS within one IP. The training procedure includes forward propagation, backpropagation and

updating of the network's parameters (using Adam optimization algorithm). The forward propagation was implemented by Nikoletta Palatianna. Back-propagation and the Adam algorithm were implemented as part of this thesis. We exploited the tool's parallelization capabilities by using pipelining, loop unrolling, array partitioning and other directives. In the next step, we implemented our FPGA architecture in the GUI of the Vivado IDE. Specifically, we instantiated our custom Training IP along with other necessary modules and coordinated the communication between them as well as between DDR and PL in order to make the system work effectively.

We implemented two approaches of Training IP using Vivado HLS, which resulted in two FPGA-based implementations of our bio-inspired ANN. The second approach we propose optimizes latency and throughput. Due to this, it performs an epoch of training (for the entire MNIST dataset) faster than the first approach, requiring 2.3797 seconds as opposed to 2.915 seconds. A disadvantage of the second approach is that it requires more FPGA resources and more on-chip power (watt). However, it is slightly more energy efficient, consuming 0.00796 Joules per batch in comparison to 0.008 Joules per batch in the first approach. The second FPGA-based implementation approach is therefore preferred, since it is both faster and more energy efficient. Comparing our proposed FPGA design with the CPU implementation (in Keras) on the MNIST dataset reveals significant improvements in latency and throughput with speedups of 14.38x and 15.548x, respectively. As a result of these speedups, an epoch of training is completed significantly faster than before, taking only 2.3797 seconds (as mentioned earlier) as opposed to 37 seconds on a CPU. GPU implementation achieves only a 2x speedup in latency and throughput over CPU implementation, executing an epoch of training in 17 seconds. The most notable feature of our FPGA design is its high energy efficiency, which is 106.15 times greater than that of CPUs and 56.5 times greater than that of GPUs. In terms of accuracy/error results in training and validation, our proposed FPGA implementation did not achieve the same level of performance as the CPU/GPU implementation. The small gap in accuracy/error results between these implementations, however, is considered insignificant, given the high level of acceleration and energy efficiency provided by the FPGA implementation.

7.2 Future Work

In our FPGA design, we have not implemented the validation procedure and callbacks. The validation procedure has only been implemented at the simulation level of Vivado HLS. These features will be implemented in the thesis of Nikoletta Palatiana. It would be interesting to investigate the possibility of incorporating these features into the PL of FPGA. With these features incorporated, we will be able to conduct the training procedure in an optimal manner, resulting in a lower validation error.

7.2.1 Plasticity rules

In their bio-inspired ANN model (mentioned above), Postdoctoral Researcher S. Chavlis and Research Director P. Poirazi have proposed two different strategies regarding learning rules. This thesis presents the implementation of the first strategy, which uses classical backpropagation. According to the second strategy, the Covariance rule (plasticity rule) is applied to the first layer and backpropagation is applied to the remaining layers. The second strategy will be applied in the thesis of Nikoletta Palatiana. Covariance rule is an unsupervised rule, which does not consider the model's output (loss) while updating its parameters. In other words, the updating of the parameters (for the first layer) is a completely independent process based on the covariance of the inputs. By incorporating a covariance rule or another plasticity rule into our presented design, we are able to enhance the bio-inspiration of the model and may be able to further improve its time and energy efficiency.

7.2.2 Rewiring

The bio-inspired ANN model we developed is equipped with dendritic-structure and receptive field. As a result of these features, the presented model is characterized by a sparse connectivity structure, which is achieved by applying masks to the weights. According to the presented model, masks remain fixed throughout the entire process, while connectivity-structure typically changes in neuroscience. The development of a corresponding bio-inspired ANN in which the masks (connectivity-structure) are modified at regular intervals during training will be interesting. This feature, known as rewiring, further enhances bio-inspiration.

7.2.3 Better implementation of the FPGA architecture

As discussed in the section on FPGA implementation (5.4), we did not measure the achieved bandwidth in our design or optimize our architecture to its maximum performance. We tried to improve the theoretical bandwidth by increasing the data width from 32-bit to 64-bit and by using multiple DMAs that stream data from different HP ports of the Zynq(PS). Our goal was to implement a functional and power-efficient architecture that could speed up the training process of our bio-inspired ANN. This goal was achieved by optimizing training time (approximately 15,5 times faster than the CPU) and energy efficiency (approximately 106 times more energy efficient than the CPU). Nevertheless, our architecture is probably not as optimized as it could be in order to fully exploit theoretical bandwidth. Thus, our FPGA design may be able to be optimized.

7.2.4 Larger scale implementation

The bio-inspired ANN model we developed consists of five layers and is considered a small ANN. As a next step, this model should be implemented on a larger scale in order to compare it with current state-of-the-art models and draw interesting conclusions. This can be accomplished by adding more layers to our model. Alternatively, our small bio-inspired ANN can also be incorporated into a large CNN by replacing the last layer (dense layer) of CNN with it.

References

[5] Spyridon Chavlis and Panayiota Poirazi. "Drawing Inspiration from Biological Dendrites to Empower Artificial Neural Networks". In: *Current Opinion in Neurobiology* 70 (June 2021), pp. 1–10. URL: https://arxiv.org/abs/2106.07490v1.

[7] 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.

[8] Blake Camp, Jaya Krishna Mandivarapu, and Rolando Estrada. "Continual Learning with Deep Artificial Neurons". In: *arXiv preprint arXiv:2011.07035* 1 (Nov. 2020). URL: https://arxiv.org/abs/2011.07035.

[9] Ilenna Simone Jones and Konrad Paul Kording. "Can Single Neurons Solve MNIST? The Computational Power of Biological Dendritic Trees". In: *arXiv preprint arXiv:2009.01269* (2020). URL: https://arxiv.org/abs/2009.01269.

[10] 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.

[11] 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.

[12] 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.

[15] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference for Learning Representations (ICRL 2015)* 9 (Jan. 2017). URL: https://arxiv.org/abs/1412.6980v9.

[16] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *The Journal of Machine Learning Research* 12 (July 2011), pp. 2121–2159. URL: https://dl.acm.org/doi/10.5555/1953048.2021068.

[22] 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.

[23] Stanisław Woźniak et al. "Deep learning incorporating biologically inspired neural dynamics and in-memory computing". In: *Nature Machine Intelligence* 2.6 (June 2020), pp. 325–336. DOI: 10.1038/s42256-020-0187-0. URL: https://doi.org/10.1038/s42256-020-0187-0.

[24] P.J. Werbos. "Backpropagation through time: What it does and how to do it". In: *Proceedings of the IEEE* 78.10 (Oct. 1990), pp. 1550–1560. ISSN: 1558-2256. DOI: 10.1109/5.58337.

[25] Thomas Bohnstingl et al. "Speech Recognition Using Biologically-Inspired Neural Networks". In: *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (May 2022), pp. 6992–6996. ISSN: 2379-190X. DOI: 10.1109/ICASSP43922.2022.9747499.

[27] Alex Graves. "Sequence Transduction with Recurrent Neural Networks". In: *International Conference of Machine Learning (ICML) 2012 Workshop on Representation Learning* (Nov. 2012). URL: https://doi.org/10.48550/arXiv.1211.3711.

[28] Ronald J. Williams and David Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks". In: *Neural Computation* 1.2 (June 1989), pp. 270–280. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.2.270. URL: https://doi.org/10.1162/neco.1989.1.2.270.

[29] 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.

[30] Giorgia Dellaferrera et al. "Introducing principles of synaptic integration in the optimization of deep neural networks". In: *Nature Communications* 13.1 (Apr. 2022), pp. 1885–1898. DOI: 10.1038/s41467-022-29491-2. URL: https://doi.org/10.1038/s41467-022-29491-2.

External Links

[1] "Neuron - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Neuron.

[2] "Synapse - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Synapse.

[3] "Transfer Learning - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Transfer_learning.

[4] "Catastrophic Forgetting - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Catastrophic_interference.

[6] "Dropout Layer - Keras API reference." In: (). URL: https://keras.io/api/layers/regularization_layers/dropout/.

[13] "Receptive field - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Receptive_field.

[14] "Stochastic Gradient Descent (SGD) - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[17] "Introduction Lectures to RMSprop Gradient Descent Algorithm (Geoffrey Hinton)." In: (). URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[18] "Softmax Activation Function - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Softmax_function.

[19] "Vanishing gradient problem - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Vanishing_gradient_problem.

[20] "Spiking Neural Network (SNN) - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Biological_neuron_model.

[21] "The membrane potential - khanacademy.org." In: (). URL: https://www.khanacademy.org/science/biology/human-biology/neuron-nervous-system/a/the-membrane-potential.

[26] "Recurrent neural network (RNN) - Wikipedia." In: (). URL: https://en.wikipedia.org/wiki/Recurrent_neural_network.

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

[32] "About Keras - keras.io". In: (). URL: https://keras.io/about/.

[33] "Numpy - numpy.org". In: (). URL: https://numpy.org/.

[34] "numpy.dot - numpy.org." In: (). URL: https://numpy.org/doc/stable/reference/generated/numpy.dot.html.

[35] "numpy.multiply - numpy.org." In: (). URL: https://numpy.org/doc/stable/reference/generated/numpy.multiply.html.

[36] "numpy.sum - numpy.org." In: (). URL: https://numpy.org/doc/stable/reference/generated/numpy.sum.htmll.

[37] "numpy.ndarray.T - numpy.org." In: (). URL: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.T.html.

[38] "He normal initializer - tensorflow.org". In: (). URL: https://www.tensorflow.org/api_docs/python/tf/compat/v1/initializers/he_normal.

[39] "GlorotUniform initializer - tensorflow.org". In: (). URL: https://www.tensorflow.org/api_docs/python/tf/keras/initializers/GlorotUniform.

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

[41] "Vivado Design Suite HLx Editions - Xilinx". In: (). URL: https://www.xilinx.com/products/design-tools/vivado.html.

[42] "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.

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

[44] "Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator - UG994." In: (). URL: https://docs.xilinx.com/v/u/2019.1-English/ug994-vivado-ip-subsystems.

[45] "Vivado Design Suite User Guide: Implementation - UG904." In: (). URL: https://docs.xilinx.com/v/u/2019.1-English/ug904-vivado-implementation.

[46] "Vivado Design Suite User Guide: Design Analysis and Closure Techniques - UG906." In: (). URL: https://docs.xilinx.com/v/u/2019.1-English/ug906-vivado-design-analysis.

[47] "Xilinx Software Development Kit (SDK)." In: (). URL: https://www.xilinx.com/htmldocs/xilinx2019_1/SDK_Doc/sdk_getting_started/sdk_getting_started.html#sdk_getting_started.

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

[49] "Vivado Design Suite AXI Reference Guide - UG1037." In: (). URL: https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide.

[50] "AXI DMA v7.1 LogiCORE IP Product Guide (PG021)." In: (). URL: https://docs.xilinx.com/r/en-US/pg021_axi_dma/AXI-DMA-v7.1-LogiCORE-IP-Product-Guide.

[51] "AXI Block RAM (BRAM) Controller v4.1 - PG078." In: (). URL: https://docs.xilinx.com/v/u/en-US/pg078-axi-bram-ctrl.

[52] "Block Memory Generator v8.4 - PG058." In: (). URL: https://docs.xilinx.com/v/u/en-US/pg058-blk-mem-gen.