TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Reconfigurable Logic Based Acceleration of Convolutional Neural Network Training

Author:
Georgios FLENGAS

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Michalis ZERVAKIS
Asst. Prof. Grigorios TSAGKATAKIS(UoC)

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

March 15, 2024

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Reconfigurable Logic Based Acceleration of Convolutional Neural Network Training

by Georgios FLENGAS

In the rapidly evolving landscape of artificial intelligence and machine learning, the intricate nature of neural network architectures, combined with exponential data growth, has intensified the need for advanced computational training. Traditional CPUs and GPUs struggle to meet these demands, prompting exploration into the untapped potential of FPGA-based acceleration. This research introduces an innovative FPGA-tailored hardware architecture for training Convolutional Neural Networks (CNNs), prioritizing optimal accuracy, energy efficiency, and speedup over conventional CPU and GPU systems.

Building on prior research, we employ General Matrix Multiply (GEMM) and Image to Column(im2col) implementations, coupled with batch level parallelism. The workload distribution between the CPU and FPGA is intricately balanced, ensuring efficient collaboration, while multiple operations are synergistically combined to streamline computation time and reduce complexity. The integration of state-of-the-art machine learning algorithms with advanced FPGA design tools, including Vitis High-Level Synthesis (HLS), yields tailored IP blocks for each stage of the neural network training process.

Our Proposed Platform achieves a notable throughput of 374.32 images per second, surpassing the CPU rate of 258.7 images per second but falls behind GPU with a throughput of 1333.3 images per second, while operating at a significantly lower power consumption of 4.16 Watts (0.011 Joules per image). This positions the Proposed Platform as a leading candidate for energy-efficient neural network training, showcasing a 16.55$\times$ energy efficiency gain over CPUs and a 7.75$\times$ over GPUs.

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

Περίληψη

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Αρχιτεκτονική Επιτάχυνσης Εκμάθησης σε Συνελικτικά Νευρωνικά Δίκτυα

από τον Γεώργιο Φλέγγα

Στον ραγδαία εξελισσόμενο κόσμο της τεχνητής νοημοσύνης και της μηχανικής μάθησης, η πολύπλοκη φύση των αρχιτεκτονικών νευρωνικών δικτύων, σε συνδυασμό με την εκθετική αύξηση των δεδομένων, έχουν αυξήσει τις υπολογιστικές ανάγκες της εκπαίδευσης ενός δικτύου. Οι παραδοσιακές CPU και GPU δυσκολεύονται να ανταποκριθούν τις απαιτήσεις, γεγονός που προτρέπει στη διερεύνηση των αναξιοποίητων δυνατοτήτων της επιτάχυνσης με βάση τις FPGA. Η παρούσα έρευνα εισάγει μια καινοτόμο αρχιτεκτονική υλικού προσαρμοσμένη σε FPGA για την εκπαίδευση Συνελικτικών Νευρωνικών Δικτύων, δίνοντας προτεραιότητα στην βέλτιστη ακρίβεια, στην ενεργειακή απόδοση και στην επιτάχυνση έναντι των συμβατικών συστημάτων CPU και GPU.

Βασιζόμενοι σε προηγούμενες έρευνες, αξιοποιούμε υλοποιήσεις General Matrix Multiply και Image to Column, σε συνδυασμό με batch-level παραλληλισμό. Εξισορροπώντας την κατανομή του φορτίου εργασίας μεταξύ CPU και FPGA, εξασφαλίζουμε την αποτελεσματική συνεργασία τους, ενώ συνδυάζοντας πολλαπλές λειτουργίες επιτυγχάνουμε την εξοικονόμηση χρόνου εκτέλεσης και τη μείωση της πολυπλοκότητας. Η ενσωμάτωση αλγορίθμων μηχανικής μάθησης τελευταίας τεχνολογίας με προηγμένα εργαλεία σχεδίασης FPGA, συμπεριλαμβανομένου του Vitis High-Level Synthesis, παράγει προσαρμοσμένα IP blocks για κάθε στάδιο της διαδικασίας εκπαίδευσης του νευρωνικού δικτύου.

Η Προτεινόμενη Πλατφόρμα επιτυγχάνει σημαντικό ρυθμό επεξεργασίας 374,32 εικόνων ανά δευτερόλεπτο, υπερβαίνοντας το ρυθμό της CPU των 258,7 εικόνων ανά δευτερόλεπτο αλλά υστερεί σε σχέση με τη GPU που πετυχένει 1333,3 εικόνες ανά δευτερόλεπτο, ενώ λειτουργεί με σημαντικά χαμηλή κατανάλωση ισχύος των 4,16 Watts (0,011 Joules ανά εικόνα). Αυτό την αναδεικνύει ως μια πολύ ανταγωνιστική επιλογή για αποδοτική εκπαίδευση νευρωνικών δικτύων, επιτυγχάνοντας ωφέλεια απόδοσης ενέργειας 16,55× έναντι των CPUs και 7,75× έναντι των GPUs.

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Apostolos Dollas, for not only entrusting me with a captivating topic but for his exceptional guidance and support throughout the completion of this thesis. His expertise, constant encouragement, brilliant ideas, unwavering commitment, and steadfast belief in my abilities have been invaluable in shaping the trajectory of my research and academic development.

I would like to extend my gratitude to Asst. Prof. Grigorios Tsagkatakis for his valuable contributions to my work and expert guidance on the intricate domain of neural networks. Additionally, I am thankful to every member of the MHL-CARV group for their insightful conversations, exchange of ideas, and guidance throughout this endeavor.

I would like to extend my deepest appreciation to my mother for her unconditional support and the sacrifices she made, enabling me to study at the Technical University of Crete and dedicate this thesis to her. I am also grateful to my father and brother for always being there to listen and support me.

Finally, a special acknowledgment to Emmanouela, whose constant support and encouragement became a source of inspiration. During moments of adversity, her unwavering belief in my abilities pushed me to overcome obstacles and persevere in the face of challenges.

Lastly, to all the late-night coffees, the relentless debugging sessions, and the countless lines of code – thank you for keeping me company on this rollercoaster ride of research and discovery. As Winston Churchill once said, 'Success is not final, failure is not fatal: It is the courage to continue that counts.' Here's to the next adventure!

Flengas Georgios,
Chania 2024

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ML	**M**achine **L**earning
DL	**D**eep **L**earning
CNN	**C**onvolutional **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etwork
ANN	**A**rtificial **N**eural **N**etworks
GEMM	**Ge**neral **M**atrix **M**ultiply
im2col	**I**mage **to** **C**olumn
Conv	**Conv**olution
FC	**F**ully **C**onnected
ReLU	**Re**ctified **L**inear **U**nit
MAE	**M**ean **A**bsolute **E**rror
MSE	**M**ean **S**quared **E**rror
MNIST	**M**odified **N**ational **I**nstitute of **S**tandards and **T**echnology
CIFAR	**C**anadian **I**nstitute **F**or **A**dvanced **R**esearch
GPU	**G**raphic **P**rocessor **U**nit
CPU	**C**entral **P**rocessor **U**nit
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
IP	**I**ntellectual **P**roperty
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
BRAM	**B**lock **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
LUT	**L**ook **U**p **T**able
URAM	**U**ltra **R**andom **A**ccess **M**emory
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
AXI	**A**dvanced **eX**tensible **I**nterface
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata

MAC	**M**ultiply **A**nd **A**ccumulate
SoC	**S**ystem **o**n a **C**hip
TPU	**T**ensor **P**rocessing **U**nits
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
DDR4	**D**ata **R**ate type **4** memory
GDDR6	**G**raphics **D**ouble **D**ata **R**ate type **6** memory
I/O	**I**nput **O**utput
OPs	**O**perations **P**er **s**econd
TPs	transactions per **s**econd
FLOPs	Floating-point operations per **s**econd

Dedicated to my family and friends…

Chapter 1

Introduction

The field of artificial intelligence has seen tremendous growth in recent years, with convolutional neural networks (CNNs) emerging as a powerful tool for image and video recognition tasks. However, training these networks can be computationally intensive and time-consuming. In this thesis, we explore the use of reconfigurable logic to accelerate the training of CNNs. Reconfigurable logic offers a flexible and efficient alternative to traditional logic circuits, allowing us to optimize the hardware architecture for specific tasks. By leveraging this technology, we aim to reduce the time and energy required to train CNNs, making them more accessible and practical for real-world applications. In this thesis, we present our methodology and results for using reconfigurable logic to accelerate CNN training, along with potential applications and future directions for this technology.

1.1 Motivation

The driving problem behind this thesis stems from the pressing need to address the escalating demands of the contemporary AI and machine learning landscape. In an era marked by unprecedented advancements in these fields, the exponential growth of data and the intricacy of neural network architectures have intensified the computational requirements for training models. Traditional hardware platforms, including CPUs and GPUs, often fall short in meeting these escalating demands. This research aims at harnessing the untapped potential of FPGA-based acceleration to overcome these limitations. By seeking to optimize convolutional neural network training through hardware acceleration, this thesis also aims to address the goal of enhancing the efficiency and performance of deep learning (DL) systems. The pursuit of scalable, energy-efficient solutions is not merely an academic endeavor

but a crucial response to the burgeoning applications of AI in diverse domains, from healthcare to autonomous systems, where the need for rapid and resource-efficient model training is paramount. Thus, the motivation behind this thesis lies in the aspiration to contribute to the robust and sustainable development of AI technologies that can seamlessly integrate into the dynamic and evolving landscape of modern machine learning.

1.2 Scientific Contributions

This thesis contributes in many ways to the domain of hardware acceleration of Convolutional Neural Network training, encompassing both software and hardware aspects thereof. A novel C code tailored for CNN training with a focus on hardware acceleration was developed, optimizing memory utilization to enhance overall performance. Moreover, the research introduced Hardware Description Language (HDL) code designed to expedite forward and backward propagation, as well as the weights update process, thereby significantly contributing to the practical deployment of FPGA-based acceleration.

In addition to code optimization, a systematic workflow was established for generating extensive HLS test benches. These test benches facilitate the simulation and evaluation of the developed code within the Vitis HLS environment, further enhancing the practical implementation of FPGA-based acceleration. The FPGA architecture designed on Vitis HLS is characterized by its configurability, allowing seamless adjustment of parameters such as batch size, filter size, and the number of filters. This configurability results in a scalable architecture, enhancing versatility for various computational requirements.

In this thesis, we implemented a detailed post-synthesis analysis to evaluate the system's performance. By conducting extended testing and verification procedures, we aimed to ensure the efficiency and reliability of our reconfigurable logic-based approach. Through this analysis, we were able to validate the effectiveness of our design choices and optimizations, shedding light on the enhanced capabilities of our accelerated CNN training system.

The developed framework not only offers accelerated performance but also demonstrates superior energy efficiency compared to conventional compute methods executed on CPUs and GPUs. These findings collectively advance the state of the art in hardware acceleration of CNN training, providing both

valuable insights and tangible implementations. The research also provides useful perspectives on the performance and potential of FPGAs in addressing the computational demands and energy efficiency challenges inherent in traditional hardware platforms.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** In this chapter, we provide a detailed background on CNNs and their architecture.
- **Chapter 3 - Related Work:** This chapter we discuss traditional logic circuits and introduce reconfigurable logic as a potential alternative for accelerating CNN training. We review existing research on using reconfigurable logic for accelerating CNN training, analyze the strengths and weaknesses of different approaches and identify gaps in the literature that our research aims to address.
- **Chapter 4 - Robustness Analysis:** This chapter outlines the decision-making process for selecting networks to be accelerated using Reconfigurable Logic, compares the precision and performance of NumPy with a C implementation, and discusses the optimization of the neural network.
- **Chapter 5 - FPGA Implementation:** In this chapter, we introduce our Field-Programmable Gate Array (FPGA) target board and the Xilinx tools used, as well as describe how we implemented our approach on a FPGA.
- **Chapter 6 - Results:** This chapter presents our results from using reconfigurable logic to accelerate CNN training. We compare our approach to traditional methods and evaluate its effectiveness in terms of speed, energy consumption, and accuracy.
- **Chapter 7 - Conclusions and Future Work:** This chapter summarizes the main findings of the thesis and discusses potential applications and future directions for this technology.

Chapter 2

Theoretical Background

2.1 Machine Learning

The term *Machine Learning* was first introduced in 1959 by Arthur Samuel[1]. Machine learning is a branch of artificial intelligence focused on building applications that learn from data and improve their accuracy over time without being programmed to do so.

In data science, an algorithm is a sequence of statistical processing steps. In machine learning, algorithms are 'trained' to find patterns and features in massive amounts of data to make decisions and predictions based on new data. The better the algorithm, the more accurate the decisions and predictions will become as it processes increasingly data. The primary aim is to allow electronics to learn automatically without human intervention or assistance and adjust actions accordingly.

Machine learning algorithms are often categorized [2] as supervised or unsupervised, depending on the nature of the signal or feedback available to the learning system:

- **Supervised machine learning algorithms** can apply what has been learned in the past to new data using labeled examples to predict future events.
- **Unsupervised machine learning algorithms** ingests unlabeled data and uses algorithms to extract meaningful features needed to label, sort, and classify the data in real-time, without human intervention.
- **Semi-supervised learning algorithms** fall somewhere in between supervised and unsupervised learning, since they use both labeled and

unlabeled data for training – typically a small amount of labeled data and a large amount of unlabeled data.

- **Reinforcement machine learning algorithms** is a behavioral machine learning model that is similar to supervised learning, but the algorithm is not trained using sample data. This model learns as it goes by using trial and error. A sequence of successful outcomes will be reinforced to develop the best recommendation or policy for a given problem.

The way a machine learning application is being built can be broken down in 4 steps:

1. Select and properly prepare a training data set.
2. Choose an algorithm to run on the training data set
3. Training the algorithm to create the model.
4. Using and improving the model

An important subset of machine learning is deep learning (DL) [3], which has become the dominant approach for most of the ongoing work in the field. Deep learning algorithms define an artificial neural network that is designed to learn the way the human brain learns. Deep learning models require large amounts of data that pass through multiple layers of calculations, applying weights and biases in each successive layer to continually adjust and improve the outcomes. Deep learning models are typically unsupervised or semi-supervised.

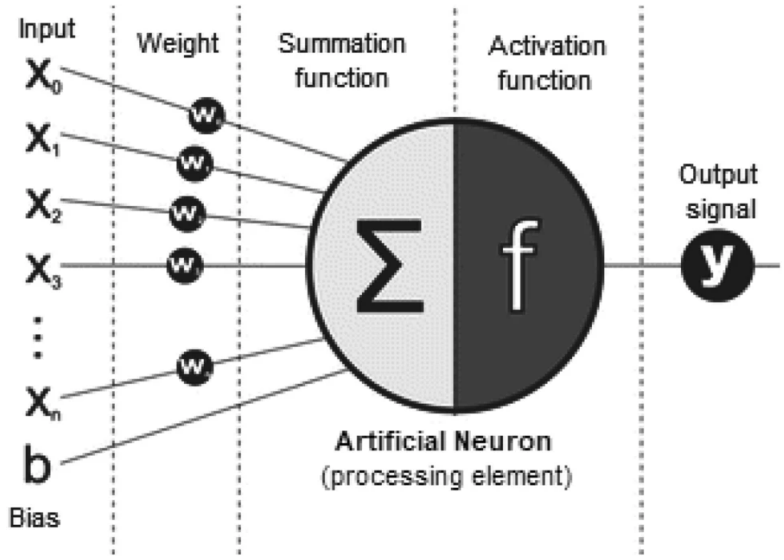

FIGURE 2.1: Schematical representation of an artificial neuron model.
Source: https://link.springer.com/article/10.1007/s42979-021-00815-1/figures/1

2.2 Artificial Neural Networks

An artificial neural network (ANN) [4] is a modeling technique inspired by the human nervous system that allows learning by example from representative data that describes a physical phenomenon or a decision process. Many of the recently achieved advancements are related to the artificial intelligence research area such as image and voice recognition, robotics, and using ANNs.

A unique feature of ANN is that they can establish empirical relationships between independent and dependent variables, and extract subtle information and complex knowledge from representative data sets. Like the structure of the human brain, the ANN models consist of neurons in a complex and nonlinear form. The neurons are connected by weighted links. All the processes in ANN models, such as data collection and analysis, network structure design, number of hidden layers, network simulation, and weights/bias trade-off, are computed through learning and training methods.

2.3 Convolutional Neural Networks

Convolutional networks [5], also known as convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that

has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be considered of as a 2D grid of pixels. They are widely used for image classification, object recognition and scene labeling tasks. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution.

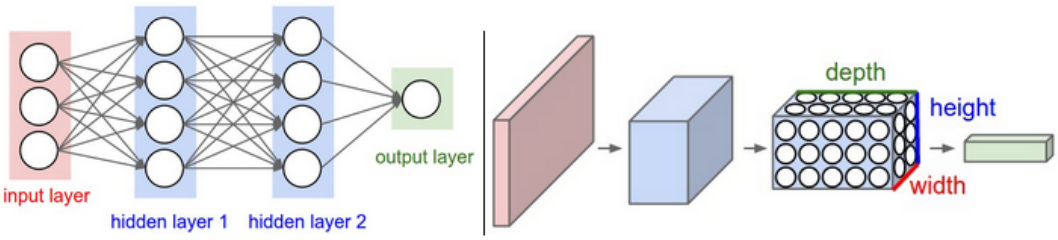

FIGURE 2.2: Left: 3-layer Neural Network. Right: A ConvNet neuron arranged in three dimensions (width, height, depth), as visualized in one of the layers.
Source: https://cs231n.github.io/convolutional-networks/

Convolution network architecture comprises a fixed set of layers designated for specialized functions. The most critical layers are as follows:

- Convolutional layer
- Pooling layer
- Full-connected layer

2.3.1 Convolutional Layer

The convolutional layer forms the core building block of a Convolutional Network, responsible for most of the computational heavy lifting. It applies a convolution operation to the input, passing the result to the next layer. The output produced by this layer is the result of the convolution between the input data and the filters(also called weights or kernels). Once the operation is completed, the biases are been added. The weights and biases are the trainable parameters of the network, which after the backpropagation part of the network is finished, are updated to improve the network performance.

Each convolutional layer is defined by the following hyperparameters:

- **Number of filters(F)**: The number of learnable filters.

- **Kernel size(K)**: It specifies the dimensions of the convolutional filters. Common kernel sizes are 3×3, 5×5, and 7×7.
- **Stride(S)**: Describes the way the filter slides over the input. Stride 1 results in filters moving one pixel at a time. Stride is 2 then the filters jump 2 pixels at a time as we slide them around, producing a smaller output.
- **Zero Padding(P)**: Convolution operation can be performed with or without zero-padding in three different ways. Valid: returns only those parts of the convolution that are computed without zero-padded edges. Same: Applies the amount of padding needed to result in the same width and height as the input. Full: Applies padding on the input's edges with a specified number of pixels per dimension

It is also important to know the size of the output. It is calculated based on the input size and the hyper-parameters of the convolution layer based on the following formula:

$$Out = \frac{InputSize - K + 2P}{S} + 1 \tag{2.1}$$

The number of the weights and biases of this type of layer can be calculated according to the following equations:

$$W_c = K^2 \times C \times F \tag{2.2}$$

K: The size of the filter,
C: The number of channels in the input volume,
F: The number of filters.

$$B_c = F \tag{2.3}$$

The way the convolution layer work is described by the following equation and is displayed in a quick example:

$$Out\,(i,j) = \sigma\left(\sum_{m=0}^{K-1}\sum_{n=0}^{K-1} I_{i\cdot S+m-P, j\cdot S+n-P} \times F_{m,n} + b\right) \tag{2.4}$$

σ: is the activation function,
$I_{i\cdot S+m-P, j\cdot S+n-P}$: The value of the input volume at position $(i \cdot S + m - P, j \cdot$

$S + n - P$),
$F_{m,n}$: The value of the filter at position (m,n),
b: The bias term associated with the filter.

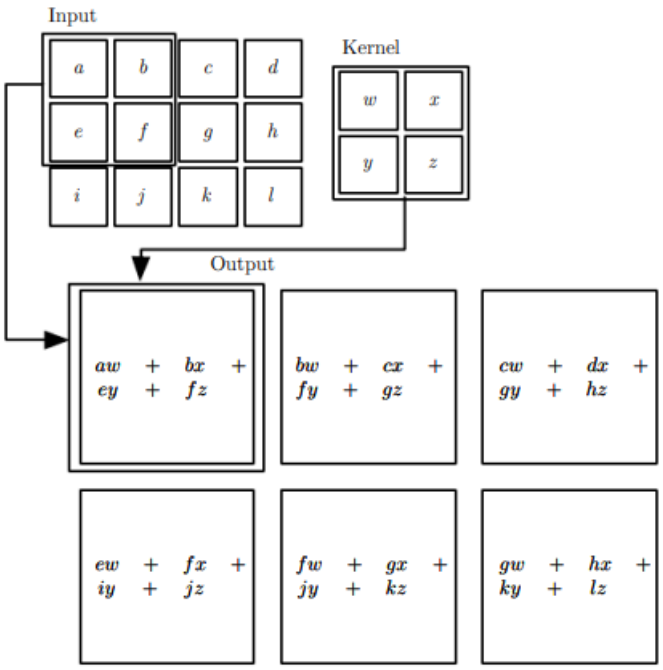

FIGURE 2.3: Naive Convolution Example. Source:[6]

Although the theory of the convolution layer is simple, the naive implementation of it can lead to a heavy computation tank, which can take quite some time to complete. As demonstrated in the following code it required 7 nested `for` loops to process a batch of data.

Algorithm 1 Naive Conv with nested for loops

```
for batch in 0..num_batches
  for filter in 0..num_filters
    for channel in 0..input_channels
      for out_h in 0..output_height
        for out_w in 0..output_width
          for k_h in 0..kernel_height
            for k_w in 0..kernel_width
              output[filter, out_h, out_w] +=
                kernel[filter, channel, k_h, k_w] *
                input[channel,out_h + k_h,out_w + k_w]
```

Generalized Matrix Multiplication (GEMM), is at the heart of deep learning. It's used in fully-connected layers, RNNs, etc., and can be used to implement

convolutions too. Note that the convolution operation essentially performs dot products between the filters and local regions of the input. If we lay out the filter into a 2-D matrix and the input patches in another, then multiplying these 2 matrices would compute the same dot product[7].

The local regions in the input image are stretched out into columns in an operation commonly called im2col. We rearrange the image into columns of a matrix so that each column corresponds to one patch where the conv filter is applied. Something to also consider when applying this technique, is the way we are arranging the data on one-dimensional arrays. Most modern DL libraries use a row-major storage order. This means that consecutive elements of the same row are stored next to each other. More generally with multiple dimensions, row-major means the first dimension changes the slowest as you scan the memory linearly.

The following example demonstrates the differences between naive convolution layer implementation and GEMM implementation [8]:

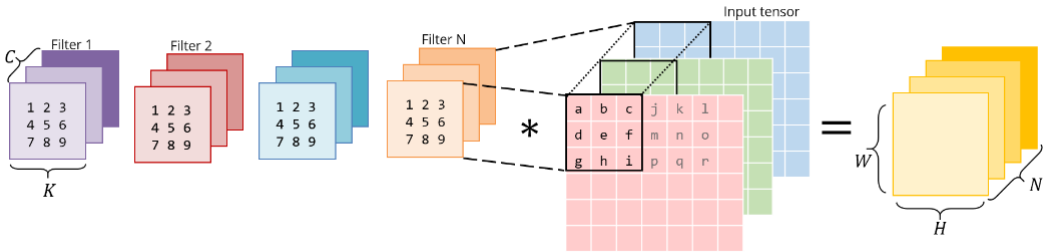

FIGURE 2.4: Naive Conv with filter size 3 of N filters
Source: https://sahnimanas.github.io/post/anatomy-of-a-high-performance-convolution/

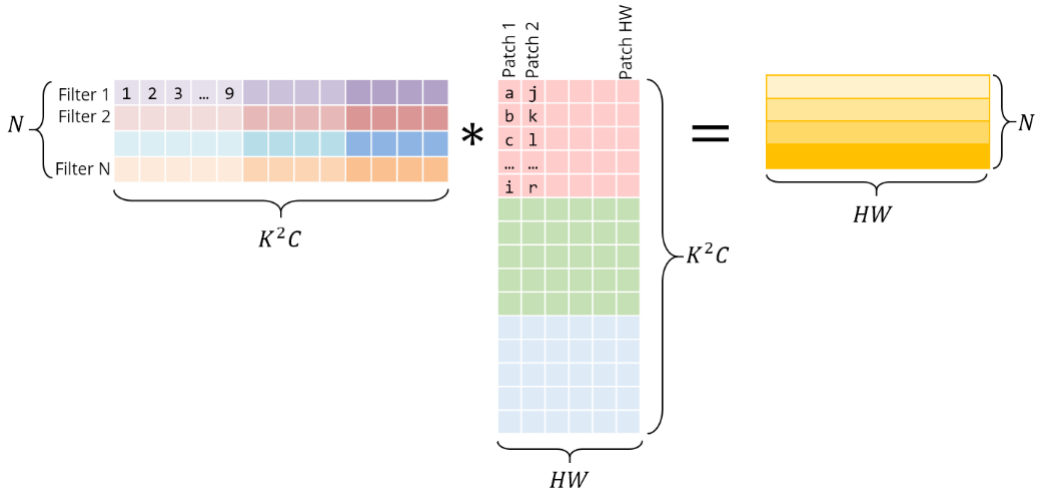

FIGURE 2.5: Right matrix is the result of im2col applied on input. Left matrix is the conv weights stored this way in memory. Source: https://sahnimanas.github.io/post/anatomy-of-a-high-performance-convolution/

This method is reported to be able to achieve up to 200× or more speedup compared to the naive implementation, at the expense of more memory consumption.

2.3.2 Weight Initialization

While building and training neural networks, it is crucial to initialize the weights appropriately to ensure a model with high accuracy. If the weights are not correctly initialized, it may give rise to the Vanishing Gradient problem or the Exploding Gradient problem. Hence, selecting an appropriate weight initialization strategy is critical when training DL models.

The vanishing gradient problem is an issue that sometimes arises when training machine learning algorithms through gradient descent. The critical point is that the calculated partial derivatives are used to compute the gradient as one goes deeper into the network. Since the gradients control how much the network learns during training, if the gradients are very small or zero, then little to no training can take place, leading to poor predictive performance. In the worst case, this may completely stop the neural network from further training further and improving. [9]

The exploding gradient problem occurs when large error gradients accumulate, resulting in substantial updates to neural network model weights during training. When the magnitudes of the gradients accumulate, an unstable network is likely to occur, which can cause poor prediction results or even a model that reports nothing useful whatsoever. [10]

Zero Initialization

As the name suggests, all the weights are assigned zero as the initial value is zero initialization. This turns out to be a mistake, because if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same. Thus, constant initializations are not preferred.

Random Initialization

This technique tries to address the problems of zero initialization since it prevents neurons from learning the same features of their inputs since our goal is to make each neuron learn different functions of its input and this technique gives much better accuracy than zero initialization. However, assigning values randomly to the weights, problems such as Overfitting, Vanishing Gradient Problem, Exploding Gradient Problem might occur. Random Initialization can be of two kinds: Random Normal (weights are initialized from values in a normal distribution), Random Uniform (weights are initialized from values in a uniform distribution).

Xavier/Glorot Initialization

Xavier Glorot and Yoshua Bengio examined the theoretical effects of weight initialization on the vanishing gradients problem[11]. They proposed an initialization scheme, which is based on the idea to initialize each weight from a Gaussian distribution with zero mean and variance based on the fan-in and fan-out of the weight. The variance is calculated based on the following equation :

$$Variance(W) = \frac{2}{n_{in} + n_{out}} \tag{2.5}$$

W: The initialization distribution for the neuron in question,
n_{in}: The number of input units,
n_{out}: The number of output units.

If sampling from a uniform distribution, this translates to sampling the interval [-r,r], where:

$$r = \sqrt{\frac{6}{n_{in} + n_{out}}} \tag{2.6}$$

also known as the normalized Xavier initialization:

$$W \sim U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right) \tag{2.7}$$

where U(a,b)denotes a uniform distribution between a and b.

He Initialization

Although it attempts to do the same, He initialization[12] is different from Xavier initialization. It is an initialization method for neural networks that takes into account the non-linearity of activation functions, such as ReLU activations.

A proper initialization method should avoid reducing or magnifying the magnitudes of input signals exponentially. Using a derivation they work out that the condition to stop this from happening is:

$$\frac{1}{2} n_l Var\left[w_l\right] = 1, \forall l \tag{2.8}$$

n_l: Represents the number of neurons (or units) in layer l,
$Var[w_l]$: Represents the variance of the weights in layer l.

This implies an initialization scheme of:

$$W \sim \mathcal{N}\left(0, \frac{2}{n_1}\right) \tag{2.9}$$

This means that the weights are initialized from a Gaussian distribution with zero mean and variance $\frac{2}{n_1}$

2.3.3 Activation Functions

An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations. The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output.

Some of the most important activation functions are presented next.

Sigmoid

The sigmoid function is highly adopted in artificial neural networks, however, it has fallen out of practice to use this activation function in real-world neural networks due to a problem known as the vanishing gradient. These neurons are known to provide the output that is smooth, real-valued, and therefore a bounded function of all the inputs. Following is the equation for the sigmoid neuron output:

$$\sigma(x) = \frac{1}{1+e^{x}} \tag{2.10}$$

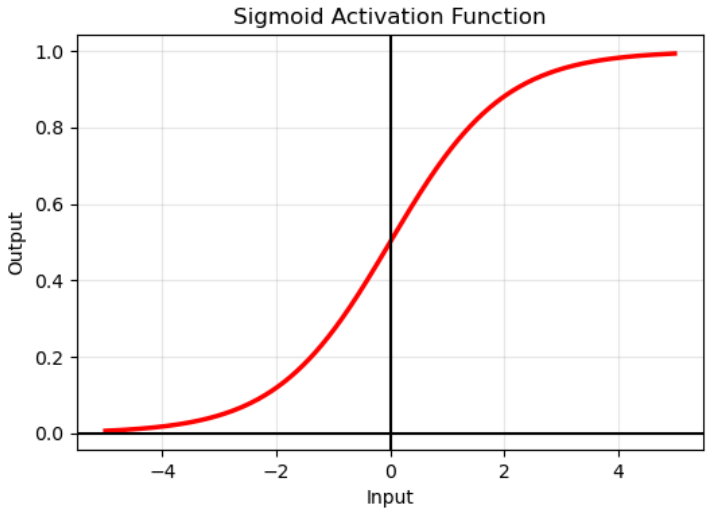

FIGURE 2.6: Sigmoid Function

Tanh

Tanh squashes a real-valued number to the range [-1, 1]. It's non-linear. But unlike Sigmoid, its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity. It is calculated based on the following:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{2.11}$$

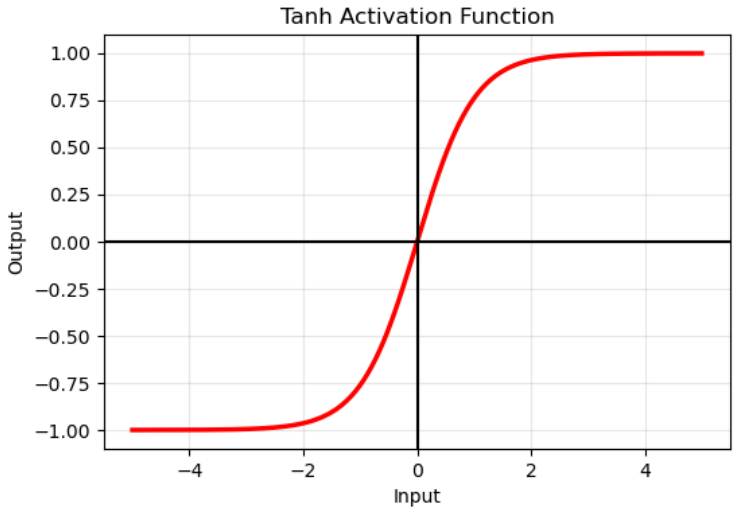

FIGURE 2.7: Tanh Function

Rectified Linear Unit (ReLU)

The Rectified Linear Unit [13] has become very popular in the last few years. It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form. Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero. Unfortunately, ReLU units can be fragile during training and can "die"

$$f(x) = \begin{cases} 0 & , x < 0 \\ x & , x > 0 \end{cases} \tag{2.12}$$

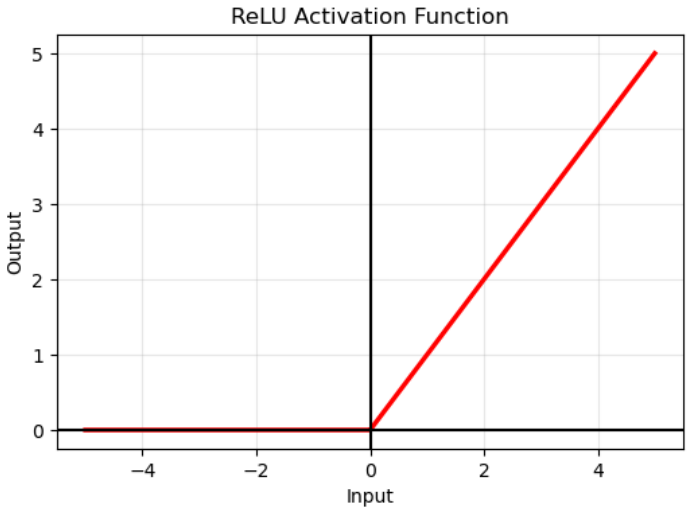

FIGURE 2.8: ReLU Function

Leaky ReLU

Leaky ReLU [12] is one attempt to fix the "dying ReLU" problem. Instead of being 0 when z<0, a leaky ReLU allows a small, non-zero, constant gradient a (normally, a=0.01). Some people report success with this activation function, but the results are not always consistent.

$$f(x) = \begin{cases} ax & , ax <= 0 \\ x & , x > 0 \end{cases} \tag{2.13}$$

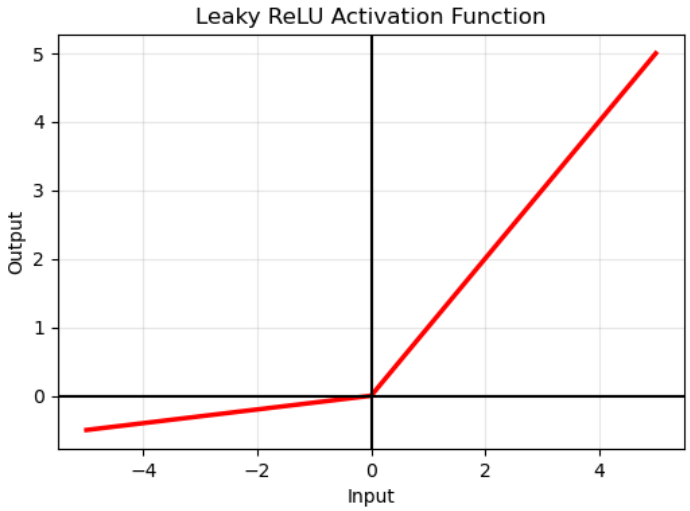

FIGURE 2.9: Leaky ReLU Function

Softmax

Commonly used as a final output activation function for multiclass classification. It normalizes the output to [0, 1], and makes its sum equal to 1. After this transformation, the i-th output's value designates the input's probability to be the class i out of the total K possible classes. After the softmax transformation is applied, the digit represented by the node with the highest probability will be the output of the CNN.

$$f_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_i}} \tag{2.14}$$

2.3.4 Pooling Layer

There can be multiple convolution layers and, between these convolution layers, there can be a pooling layer. The pooling layer is responsible for reducing the chances of over-fitting by reducing the spatial size of the input volume. The reduction of the spatial size implies reducing the number of parameters or the number of computations in the network. Just like the convolution layer, there is kernel size and stride. The size of the kernel is smaller than the feature map. For most cases, the size of the kernel will be 2×2 and the stride of 2. There are mainly two types of pooling layers.

The size of the output is calculated based on the input size, the pooling/kernel size of the maxpool layer and the stride, based on the following formula:

$$Out = \frac{InputSize - PoolSize}{Stride} + 1 \tag{2.15}$$

The first type is maxpooling layer. Maxpooling layer will take a stack of feature maps (convolution layer) as input. The node's value in the max pooling layer is calculated by just the maximum of the pixels in the window. semi-supervised.

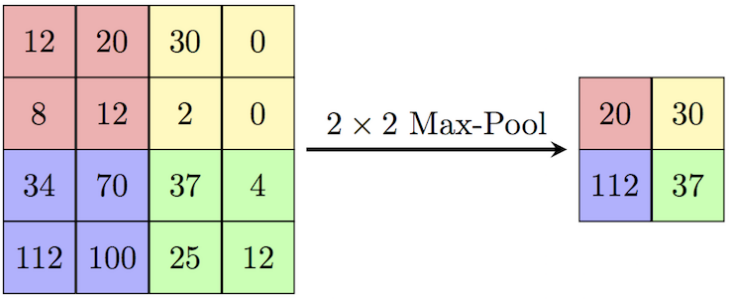

FIGURE 2.10: Example of maxpool with pool size 2. Source: https://computersciencewiki.org/index.php/File:MaxpoolSample2.png

The other type of pooling layer is the Average Pooling layer. Average pooling layer calculates the average of pixels contained in the window. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.

2.3.5 Fully Connected Layer

Fully Connected Layers are layers of nodes each of whom, much like neurons, is connected to all nodes of the previous layer. They are used for the supervised classification of the input and they have a huge number of parameters whose values are determined through the training of the network. In most popular machine learning models, the last few layers in the network are fully-connected ones.

The fully-connected layer receives in input a vector of nodes, activated in the previous convolutional layers. This vector passes through one or more dense layers, before being sent to the output layer. Before it reaches the output layer, an activation function is used for making a prediction.

The fully-connected layer's output, Out, is defined as:

$$Out\,(i) = B\,(i) + \sum_{j=1}^{N} I\,(j)\,W\,(i,j)\,,\ \text{for i=1,...,M} \tag{2.16}$$

B: The layer's biases,

W: The layer's weights,

I: The input with N input features.

The number of Parameters of a Fully Connected is depended on the type of layer that is connected to it:

- Number of Parameters of a Fully Connected Layer connected to a Conv Layer:

$$\begin{aligned} W_{cf} &= O^2 \times C \times F \\ B_{cf} &= F \end{aligned} \tag{2.17}$$

  O: The Size(width) of the previous layer's output,

  N: The number of kernels of the Conv Layer,

  F: The number of neurons of the FC layer.

- Number of Parameters of a Fully Connected Layer connected to a FC Layer:

$$\begin{aligned} W_{ff} &= F_{-1} \times F \\ B_{ff} &= F \end{aligned} \tag{2.18}$$

  F: The number of neurons of the FC layer,

  F_{-1}: The number of neurons of the previous FC layer.

While the convolutional and pooling layers generally use a ReLU function, the fully-connected layer can use two types of activation functions, based on the type of the classification problem:

- Sigmoid: A logistic function, used for binary classification problems.
- Softmax: A more generalized logistic activation function, it ensures that the values in the output layer sum up to 1. Commonly used for multi-class classification.

The activation function outputs a vector whose dimension is equal to the number of classes to be predicted. The output vector yields a probability from 1 to 0 for each class.

2.3.6 Loss Function

Once the forward part of the CNN is done executed, we have to check how accurate the results were. A loss function has to be implemented to compare the results with the label of each picture that went through the network. Generally speaking, the loss allows us to compare between some actual targets and predicted targets. It does so by imposing a "cost" (or, using a different term, a "loss") on each prediction if it deviates from the actual targets.

The goal when training a machine learning model is to minimize the loss. The reason is simple the lower the loss, the more the set of targets and the set of predictions resemble each other. This results in a better-performing machine learning model.

Mean Absolute Error

Mean absolute error (MAE)[14] is a loss function used for regression. The loss is the mean overseen data of the absolute differences between true and predicted values. It is calculated based on the following function:

$$L\left(y,\widehat{y}\right)=\frac{1}{N}\sum_{i=0}^{N}|y-\widehat{y}_i| \tag{2.19}$$

where $\widehat{y}_i$ is the predicted value.

A disadvantage of MAE is that the gradient magnitude is not dependent on the error size, only on the sign of y - ŷ. As a result, the gradient magnitude will be large even when the error is small, which in turn can lead to convergence problems.

Mean Squared Error

Mean squared error (MSE)[15] is the most commonly used loss function for regression. The loss is the mean overseen data of the squared differences between true and predicted values. Calculated as:

$$L\left(y,\widehat{y}\right)=\frac{1}{N}\sum_{i=1}^{N}\left(y-\widehat{y}_i\right)^2 \tag{2.20}$$

MSE is sensitive towards outliers and given several examples with the same input feature values, the optimal prediction will be their mean target value. This should be compared with Mean Absolute Error, where the optimal prediction is the median. MSE is good to use if you believe that your target data, conditioned on the input, is normally distributed around a mean value, and when it's important to penalize outliers extra much.

Cross-Entropy

Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverge from the actual label. So predicting a probability of .012 when the actual observation label is 1 would be bad and result in a high loss value. A perfect model would have a log loss of 0.

In binary classification, where the number of classes M equals 2, cross-entropy can be calculated as:

$$L\left(y, \widehat{y}\right) = \frac{1}{N}\sum_{i=0}^{N} y_i \cdot \log \widehat{y_i} + (1 - y_i) \cdot \log\left(1 - \widehat{y_i}\right) \tag{2.21}$$

If M>2M>2 (i.e. multiclass classification), we calculate a separate loss for each class label per observation and sum the result.

$$L\left(y, \widehat{y}\right) = -\sum_{i=1}^{N} y_i \cdot \log \widehat{y_i} \tag{2.22}$$

When working with batches, we need to make predictions over multiple examples, so the average of the loss over all examples needs to be calculated.

2.3.7 Gradient Descent

Gradient descent[16] is an optimization algorithm used to minimize some functions by iteratively moving in the direction of the steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model.

Conversely, stepping in the direction of the gradient will lead to a local maximum of that function; the procedure is then known as gradient ascent. The size of these steps is called the learning rate[17]. With a high learning rate,

we can cover more ground with each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom.

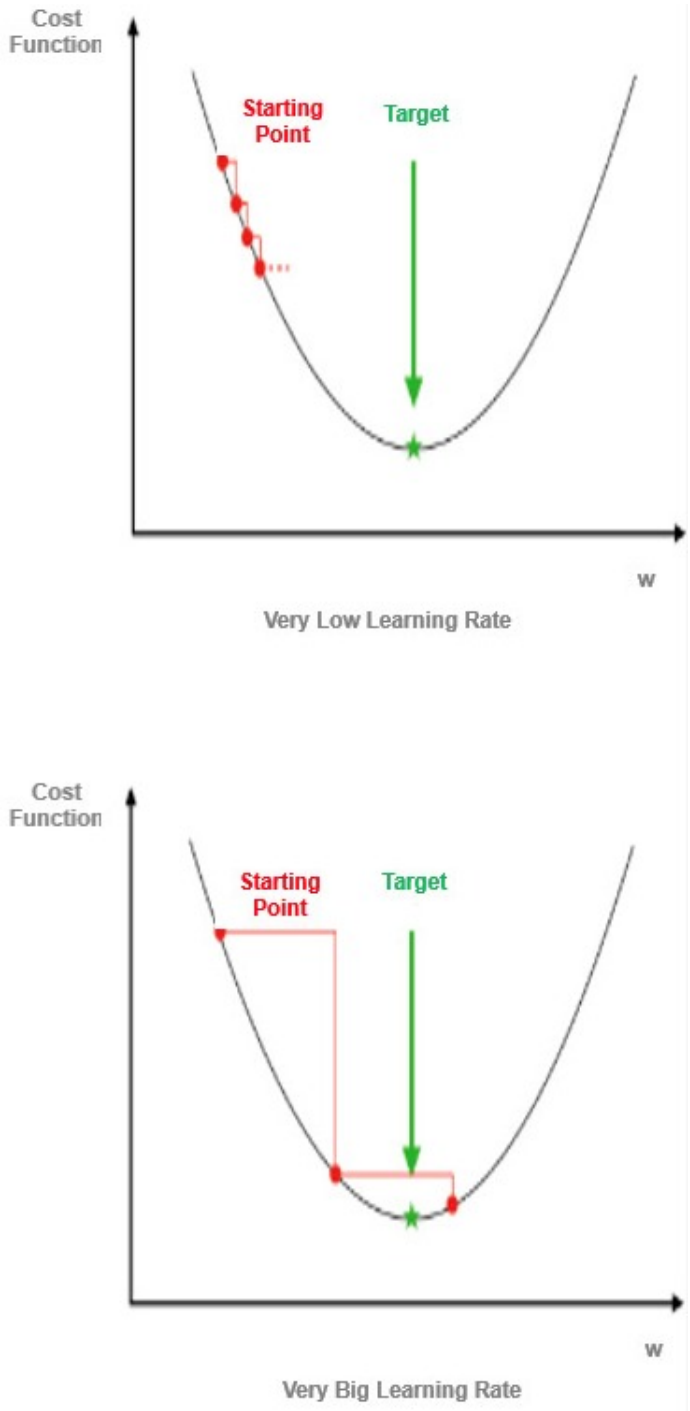

FIGURE 2.11: Effect of high and low learning rate.

In large-scale applications, the training data can have on the order of millions of examples. Hence, it seems wasteful to compute the full loss function over the entire training set to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over batches of the training data. This batch is then used to perform a parameter update. This process is known as Mini-batch gradient descent[18].

2.3.8 Back Propagation

The goals of back-propagation[19] are straightforward: adjust each weight in the network in proportion to how much it contributes to the overall error. If we iteratively reduce each weight's error, eventually we'll have a series of weights that produce good predictions.

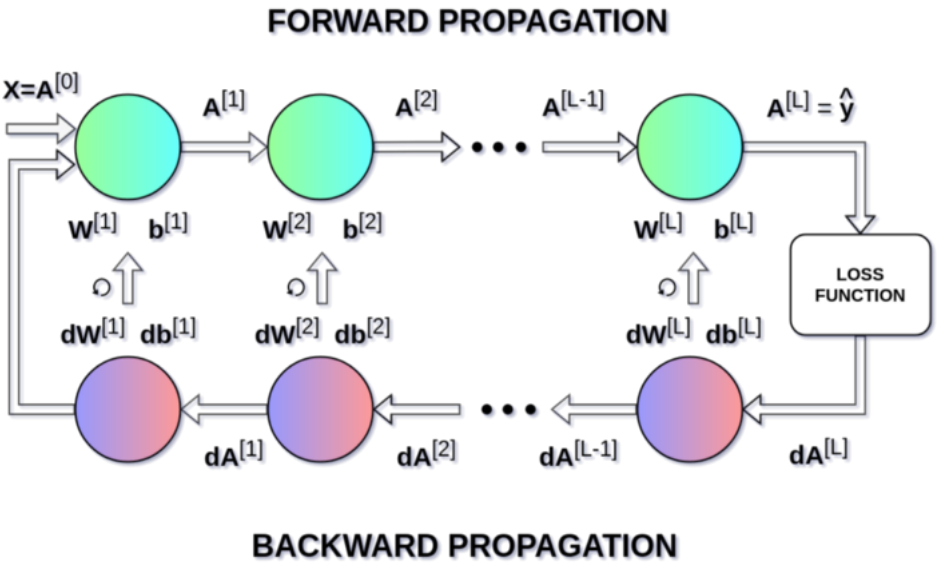

FIGURE 2.12: Backpropagation Illustrated.
Source: https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba

When we use a feed-forward neural network to accept an input x and produce an output ŷ, information flows forward through the network. The inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces ŷ. This is called forward propagation. During training, forward propagation can continue onward until it produces

a scalar cost $J(\theta)$. The back-propagation algorithm, often simply called back-prop, allows the information from the cost to then flow backward through the network, to compute the gradient[6].

Computing the analytical expression of the gradients is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure. The parameters of the neural network are adjusted according to the following formulae:

$$\begin{aligned} W^{[l]} &= W^{[l]} - \alpha \times dW^{[l]} \\ b^{[l]} &= b^{[l]} - \alpha \times db^{[l]} \end{aligned} \tag{2.23}$$

a represents the learning rate, which allows us to control the value of the performed adjustment. A low learning rate can result in a very slow learning network and a high one can result in not being able to hit the minimum. The parameters dW and db are calculated using the chain rule, partial derivatives of the loss function with respect to W, and b. The size of dW and db is the same as that of W and b respectively. These variables are calculated following the formulas:

$$\begin{aligned} dW^{[l]} &= \frac{1}{dW^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \\ db^{[l]} &= \frac{1}{db^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)} \\ dA^{[l-1]} &= \frac{dL}{dA^{[l-1]}} = W^{[l]T} dZ^{[l]} \\ dZ^{[l]} &= dA^{[l]} * g'\left(Z^{[l]}\right) \end{aligned} \tag{2.24}$$

where Z is the output of a layer, A is the activation output of the corresponding layer, m is the number of examples from the training set and g' is the derivative of the non-linear activation function.

2.4 Datasets

Data forms the main source of learning in Machine learning. The data that is being referenced here can be in any format, can be received at any frequency, and can be of any size. When it comes to handling large datasets in the Machine learning context,there are some new techniques that have evolved and

are being experimented with. There are also more big data aspects, including parallel processing, distributed storage, and execution. When we think of data, dimensions come to mind. To start with, we have rows and columns when it comes to structured and unstructured data. A collection of rows or instances is called a dataset.In the context of Machine learning, there are different types of datasets that are meant to be used for different purposes. An algorithm is run on different datasets at different stages to measure the accuracy of the model. There are three types of dataset: training, testing, and evaluation datasets. Any given comprehensive dataset is split into three categories of datasets and is usually in the following proportions: 60% training, 30% testing, and 10% evaluation.

2.4.1 MNIST

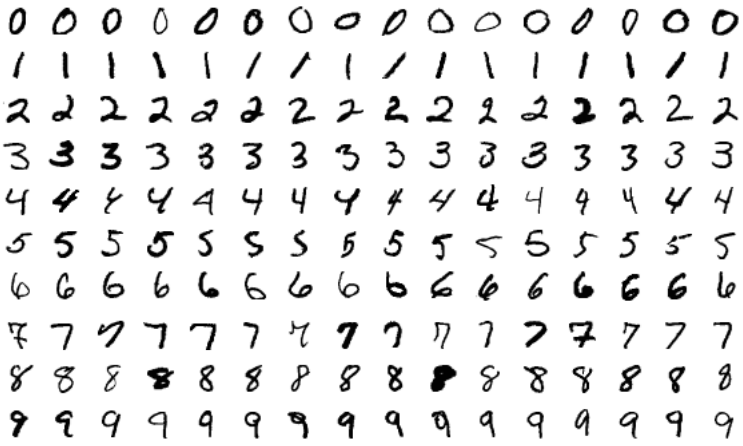

FIGURE 2.13: Sample images from MNIST dataset.
Source: https://www.researchgate.net/figure/Visualization-of-MNIST-Digits-The-digits-have-been-size-normalized-and-centered-in-a_fig1_348377922

The MNIST database (Modified National Institute of Standards and Technology database) [20] [21] is a large collection of handwritten digits. It has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger NIST Special Database 3 and Special Database 1 which contain monochrome images of handwritten digits. The digits have been size-normalized and centered in a fixed-size image. The original black and white images from NIST were size normalized to fit in a 20×20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result

of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28×28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28×28 field.

2.4.2 CIFAR-10

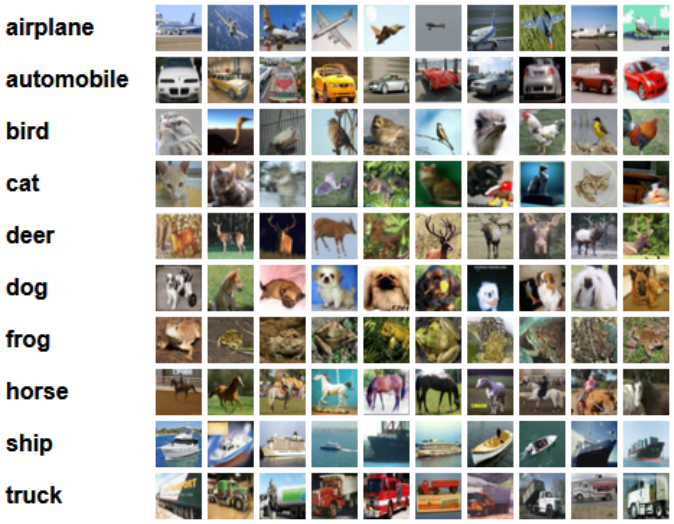

FIGURE 2.14: Sample images from CIFAR dataset. Source: [22].

The CIFAR-10 dataset (Canadian Institute for Advanced Research) [22] [23] is a subset of the Tiny Images dataset and consists of 60000 32×32 color images. The images are labelled with one of 10 mutually exclusive classes: airplane, automobile (but not truck or pickup truck), bird, cat, deer, dog, frog, horse, ship, and truck (but not pickup truck). There are 6000 images per class with 5000 training and 1000 testing images per class.

2.4.3 ImageNet

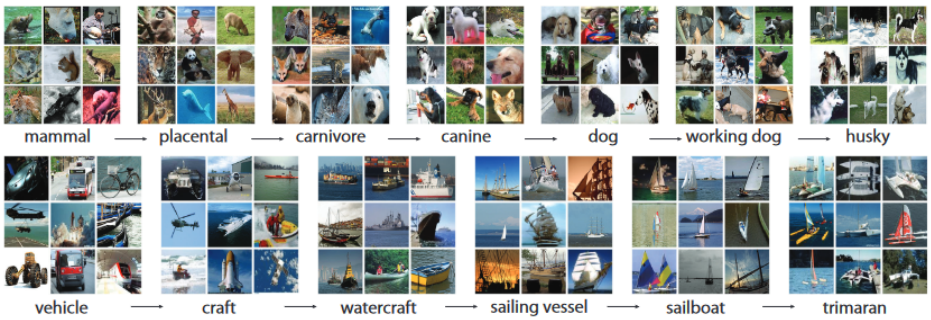

FIGURE 2.15: Sample images from ImageNet dataset. Source: [24].

The ImageNet dataset [24] [25] contains 14,197,122 annotated images according to the WordNet hierarchy. Since 2010 the dataset is used in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), a benchmark in image classification and object detection. The publicly released dataset contains a set of manually annotated training images. A set of test images is also released, with the manual annotations withheld. ILSVRC annotations fall into one of two categories: (1) image-level annotation of a binary label for the presence or absence of an object class in the image, e.g., "there are cars in this image" but "there are no tigers," and (2) object-level annotation of a tight bounding box and class label around an object instance in the image, e.g., "there is a screwdriver centered at position (20,25) with width of 50 pixels and height of 30 pixels".

Chapter 3

Related Work

3.1 Deep Learning Software Frameworks

FIGURE 3.1: Famous Deep Learning Frameworks.
Source:https://marutitech.com/top-8-deep-learning-frameworks/

A deep learning framework is a software package used by researchers and data scientists to design and train deep learning models. The idea of these frameworks is to allow people to train their models without digging into the algorithms underlying deep learning, neural networks, and machine learning. These frameworks offer building blocks for designing, training, and validating models through a high-level programming interface.

3.1.1 TensorFlow:

TensorFlow [26][27] is one of the most popular deep learning frameworks. Developed by the Google Brain team, TensorFlow supports languages such as Python, C++, and R to create deep learning models along with wrapper libraries. While the core tool allows the user to build and deploy models on

browsers, TensorFlow Lite can be used to deploy models on mobile or embedded devices. Also, if the user wish to train, build, and deploy ML/DL models in large production environments, TensorFlow Extended serves the purpose. It demands extensive coding, and it operates with a static computation graph. So, the user will first need to define the graph and then run the calculations. In case of any changes in the model architecture, the user will have to re-train the model.

3.1.2 PyTorch:

PyTorch [28][29] is an open-source Deep Learning framework developed by Facebook, based on the Torch library. It runs on Python but also has a C++ frontend. It excels in training, building, deploying small projects and prototypes. In contrast to Tensorflow, it operates with a dynamically updated graph, meaning that the user can make the necessary changes to the model architecture during the training process itself.

3.1.3 Keras:

Keras [30] library was developed with quick experimentation as its main selling point. Written in Python, it can run on top of TensorFlow, Theano, Microsoft Cognitive Toolkit, and PlaidML. Its speed, comes from its built-in support for data parallelism, and hence, it can process massive volumes of data while accelerating the training time for models. It is best suited for learning and prototyping, promoting fast experimentation with deep neural networks, while writing readable and precise code.

3.1.4 Caffe:

Caffe [31][32] is a deep learning framework made with expression, speed, and modularity in mind. It is developed by Berkeley AI Research and by community contributors, supporting interfaces like C, C++, Python, MATLAB, and Command-Line. The most significant benefit of using Caffe's C++ library is accessing the deep net repository 'Caffe Model Zoo'[33]. It contains networks that are pre-trained and can be used immediately. Whether it is modeling CNNs or solving image processing issues, this has got to be the go-to library.

3.2 Hardware Acceleration

In the past decade, Convolutional Neural Networks have demonstrated state-of-the-art performance in various Artificial Intelligence tasks. Four general classes of hardware are used for accelerating neural networks: high-power CPUs, low-power embedded microprocessors, FPGAs, and GPUs. Conventionally, CNNs have been executed on CPUs and GPUs; however, their low throughput and/or energy efficiency presents a bottleneck in their use.

3.2.1 Graphics Processing Units

GPUs are the preferred research method for training and running CNNs because they hide memory access penalties by compensating for image throughput. Training models is a hardware-intensive operation, and a good GPU will ensure that neural network operations operate smoothly. Gigabytes of training images are loaded onto the video RAM (VRAM) of the GPU, and the operations are distributed among hundreds of small cores optimized for general matrix multiplication. The training and inference operations of a CNN are batched over tens to hundreds of images at a time to minimize the data transfer overhead between the RAM and GPU core on the GPU card. However, GPUs can draw hundreds of watts of power and are inefficient for applications that require low latency. State-of-the-art graphics cards such as the Nvidia Tesla V100[34] are capable of 14 teraFLOPS with single-precision floating point,offer 1134 GB/s memory bandwidth, but they can draw up to 250 watts at peak consumption .

3.2.2 Central Processing Units

The CPU is the core component that defines a computing device. Most of these CPUs are general-purpose, flexibly programmable, and built for good performance on a wide range of computational workloads. Many different types of processors suitable for embedded systems exist, with various tradeoffs regarding speed and power requirements. Single Instruction Multiple Data (SIMD) units and Multiply and Accumulate (MAC) instructions are common among processors for media processing, and they can utilize multiple cores. However, CPUs compute results sequentially and are thus not ideally suited for the highly parallel problem presented by convolutional neural networks.

3.2.3 Application-Specific Integrated Circuits

ASICs are computer chips that combine several different circuits all on one chip – it's a "system-on-a-chip" (SoC) design – allowing it to be custom programmed to combine several related functions that together carry out a specific overall task. They are the ideal solution when it comes to maximum performance and maximum energy efficiency. However, ASICs are not suited for irregular computation, and they further require much of the algorithm to be frozen at design time. For this reason, ASICs are typically only built to accelerate a certain aspect of CNNs. Until CNN research and infrastructure stabilizes, ASICs designed for CNN may require too much design effort and capital investment to become viable for widespread adoption. Tensor Processing Unit (TPU) [35] is a custom ASIC, built by Google, focused on machine learning acceleration. Compared to a server-class Intel Haswell CPU and an Nvidia K80 GPU, it achieves up to 200x and 70x speedup respectively.

3.2.4 Field-Programmable Gate Arrays

FPGAs are integrated circuits, which are sets of circuits on a chip. Those circuits, or arrays, are groups of programmable logic gates, memory, or other elements. They can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from ASICs, which are custom manufactured for specific design tasks. FPGA designs are best suited for calculations that can be heavily parallelized by building custom processing engines using programmable logic blocks. Algorithms that require data-dependent branching and decisions are less suited for this type of parallelization and result in poor utilization of the computational power. Products like Xilinx® Alveo™ U50 [36] Data Center accelerator cards can provide up to 316 GB/s bandwidth using 75W, offering 872 thousand Look-Up tables, 1,7 million registers, 5,952 DSP slices, and 28 MB of on-chip SRAM.

3.3 The FPGA Perspective

In recent years, Convolutional neural networks have been one of the primary subject of research by large companies and research teams around the world. The top priority was to increase model accuracy, training speed and energy efficiency. Software libraries have been introduced, ASIC circuits and

GPU frameworks have been developed to enhance and optimize neural network performance. Due to their several attractive features, FPGAs present as promising platforms for hardware acceleration of CNNs [37], while providing higher energy efficiency than both GPUs and CPUs and higher performance than CPUs[38]. FPGAs are primarily used for CNN inference acceleration, proposing varous designs to map pretrained neural networks on FPGAs resulting in high throughput and low latency [39][40]. Due to limited resources, it is more complex to efficiently implement CNN training on FPGAs. The main restrains for the innefficiency of FPGAs for CNN training are:

- It includes forward and backward propagation and the weight updates, resulting in 3× computation operation count compared to CNN inference.
- Data generated during both forward and backward propagation need to be available during the weight update operations, something that can be difficult for on-board memory management and data reusing in dynamic random access memory (DRAM).
- Different patterns of memory access between the 3 opperations, which can lead to low memory access efficiency.

Recently, the development of FPGA-based architectures for the training of CNNs have been on the rise. F-CNN[41] was one of the first frameworks implemented for accelerating the training of CNNs in FPGAs, introducing a hybrid CPU/FPGA design. The CPU is the controller, FPGA is the computation accelerator and DRAM on the FPGA card is used to store each computation module's input and output data. Caffe Barista[42] works similarly, providing a memory-aware model for executing an FPGA-based general matrix multiply (GEMM) kernel, with all of the inputs being pre-processed by the CPU into a tiled layout that is sequentially stored in memory.

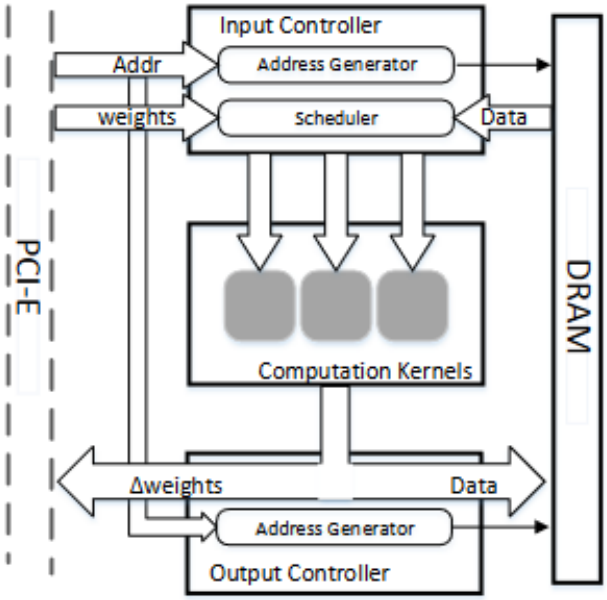

FIGURE 3.2: The basic architecture of modules in F-CNN. Source: [41].

FPDeep[43] presented a way to map a specific CNN training logic to a multi-FPGA cluster or cloud, executing training in a fine-grained inter and intra-layer pipeline manner minimizing the time that features need to remain available while waiting for back-propagation. One of the main subjects of focus is finding the optimal floating-point format. Works like [44] investigate training in 32 bits, 24 bits, 16 bits, and mixed precisions to find the optimal floating-point format for low power and smaller-sized edge device, while [45] introduces the complete process of mapping the network structure to the FPGA based on the Yolov3-tiny algorithm and optimizes the accelerator architecture for Zedboard to make it under limited resources to achieve the best performance. DarkFPGA [46] accelerates the entire DNN training on a single FPGA using a low-precision 8-bit integer and presenting strategies such as batch-focused data sequence CHWB and tiling strategies. Other papers like [47] are using 16-bit-fixed-point precision for complete CNN training, or present hybrid fixed-point floating-point architectures like [48] which demonstrate an architecture using a training method with fixed-point quantized weights is also presented, or [49] where all GEMM instructions using 8-bit integer numbers to the PL and the ARM processor to compute the rest of the network in floating-point.

Exploiting any kind of parallelism is essential for FPGA architectures. DarkFPGA[46] is using batch-level parallelism, while Zhiqiang Liu et al [50] try to exploit inter-output parallelism, intra-output parallelism, row-level parallelism, and operator-level parallelism.

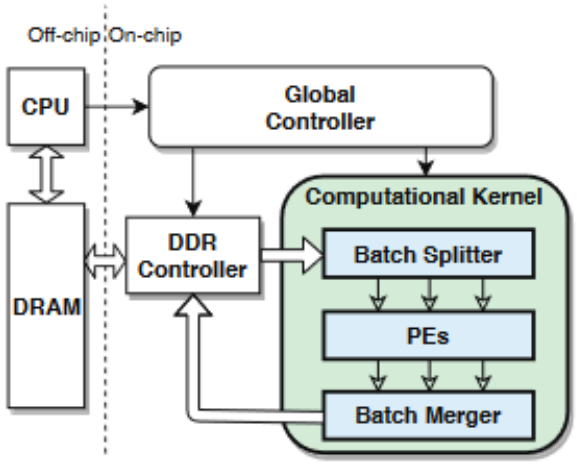

FIGURE 3.3: DarkFPGA System overview. Source:[46].

Caffeinated FPGA[51] introduced the XCLProgram layer as a method for giving the user greater control over how the FPGA is programmed and simple-to-use test benches. Facilitated by XCLProgram layers, pipeline layers package multiple kernels into a single binary, with kernel-kernel communication occurring through local memory structures on the FPGA. FeCaffe [52],is another hierarchical software and hardware design methodology based on the Caffe framework that enables FPGA to support deep learning development features, such as training and inference with Caffe. FeCaffe is capable of supporting almost full features during CNN network training and inference respectively with high degree of design flexibility, expansibility and reusability for deep learning development. A really important step for accelerating and making CNN training feasible on FPGAs is the use of image-to-column transformation and GEMM, instead of the naive convolution implementation as demonstrated by [53] [54]. Some other great ideas include the integration of convolution-pooling block to achieve low area and low energy consumption[55], accelerating the algorithm to minimize the number of arithmetic operations and memory accesses[56], and developing a data reshaping approach with intra-tile continuous memory allocation and weight reuse [57]. Last but not least, TrainWare[58] states that the weight update operation from

the training process is the primary factor of high energy consumption due to its substantial memory accesses.

The results of some of those ideas are presented in the following table:

	Throughput	Energy Efficiency
F-CNN	62.06 GFLOPS	N/A
FPDeep	1157 GOPS(Per FPGA)	37.09 GOPS/J
DarkFPGA	1417 GOPS	104.96 GOPS/W
FeCaffe	24 GFLOPS	N/A
TrainWare	16.2 GMAC/s	617 GOPS/W
EF-Train	46.99 GFLOPS	6.09 GFLOPS/W
ZynqNet	6.3 FPS	0.53 images/J

TABLE 3.1: Performance of FPGA implemented designs

Different measuring types are used because they offer different perspectives on the performance characteristics of FPGA-implemented CNN designs. The choice of metric depends on various factors such as the specific operations involved in the CNN model, the hardware architecture of the FPGA, and the performance requirements of the application. For instance, if the CNN model heavily relies on floating-point arithmetic, GFLOPs might be the most relevant metric. Conversely, if fixed-point operations are significant, GOPs might be more appropriate.

3.4 Thesis Approach

This thesis proposes a novel hardware architecture for training Convolutional Neural Networks on FPGA devices, with the aim of achieving optimal training accuracy while improving energy efficiency and throughput speedup compared to traditional CPU and GPU systems. The research is motivated by the pressing need to address the escalating demands of the contemporary AI and machine learning landscape, where the exponential growth of data and the intricacy of neural network architectures have intensified the computational requirements for training models. Traditional hardware platforms, including CPUs and GPUs, often fall short in meeting these escalating demands, making FPGA-based acceleration a promising alternative.

To achieve the research goals, we leverage prior work findings and utilize GEMM and im2col implementations, as well as batch level parallelism, while balancing the workload between the CPU controller and the FPGA. We also combine various operations to save computation time and reduce complexity. The proposed methodology builds upon existing knowledge in both deep learning and hardware acceleration domains, integrating state-of-the-art machine learning algorithms with advanced FPGA design tools such as Vitis High-Level Synthesis (HLS). This approach aims to develop a comprehensive solution capable of outperforming traditional CPU- or GPU-based systems in terms of energy efficiency and speed.

Chapter 4

Modeling and Robustness analysis

4.1 Python Implementation

The initial network for CNN training with Mini-Batch Gradient Descent is developed on pure python using the NumPy library and trained with the MNIST data set. It consists of a Convolution layer with 12 3×3 filters initialized with He initialization using Relu as activation function, a Maxpool layer with pool size 2, and a fully connected layer using Softmax as activation function. To measure the accuracy of the network, the Categorical Cross-Entropy Loss function is used.

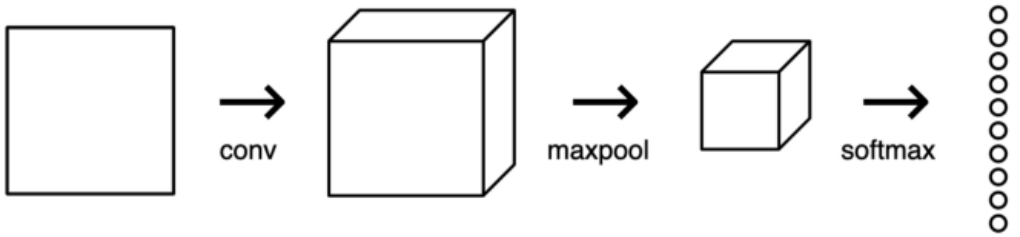

FIGURE 4.1: 3 layer CNN

4.1.1 Convolution layer

The first layer of the network is a Conv layer with 12 3×3 filters, using the naive Convolution Layer implementation . The input of this layer is four-dimensional tensor with shape [batch, channel, height, width]=[batch, 1, 28, 28] and the output is [batch,12,26,26]. Once the Convolution is completed,

the output goes through the ReLu activation function and the result is forwarded to the next layer.

Using a 12 3×3 filters means that we will have a total of 108 weights, which alongside with biases, are used for classification during the forward phase and are updated during the backpropagation phase. These weights are initialized using the He initialization[12]. This initialization technique was used to counter the vanishing/exploding weights problem, which resulted in the network's performance slowly declining to 0. Another positive effect was that since normal distribution was replaced by the He initialization, the network results were improved after the 2nd epoch for around 50 % accuracy to 75%.

4.1.2 Maxpool layer

The convolutional layers aren't supposed to reduce the size of the image significantly. Instead, they make sure that each pixel reflects its neighbors. This makes it possible to perform downscaling, through pooling, without losing important information.

A widespread method to do so is max pooling, in other words using the maximum value from a cluster of neurons at a previous layer. Indeed, maxpooling layers have a size and a width. Unlike convolution layers, they are applied to the 2-dimensional depth slices of the image, so the resulting image is of the same depth, just of a smaller width and height by dividing them by the pool size. The presented network uses pool size 2. As a result, the output of the Maxpool Layer is [batch,28,28,12].

4.1.3 Fully Connected Layer

The fully-connected layer is a combination of a flattening layer and a dense layer using Softmax as the activation function. The input is flattened into a feature vector and passed through a network of neurons to predict the output probabilities

The rows are concatenated to form a long feature vector. The feature vector is then passed through the dense layer and is multiplied by the layer's weights, summed with its biases, and passed through 10 nodes Softmax, each representing each digit since we are working with MNIST. This layer will have $28 \times 28 \times 12 = 9408$ weights connected to each node, so a total of 94080 weights, which are initialized with the He initialization as well.

After the softmax transformation is applied, the digit represented by the node with the highest probability will be the output of the CNN and the Loss Function is being calculated.

4.1.4 Results

Using the network described a Keras model was implemented so comparisons could be made.The training parameters that were used for the results presented are: batch size= 128, learning rate=0.01, 10000 training samples, and 1000 test samples for 20 epochs. The run of that model gave the following results for a 20 epochs test:

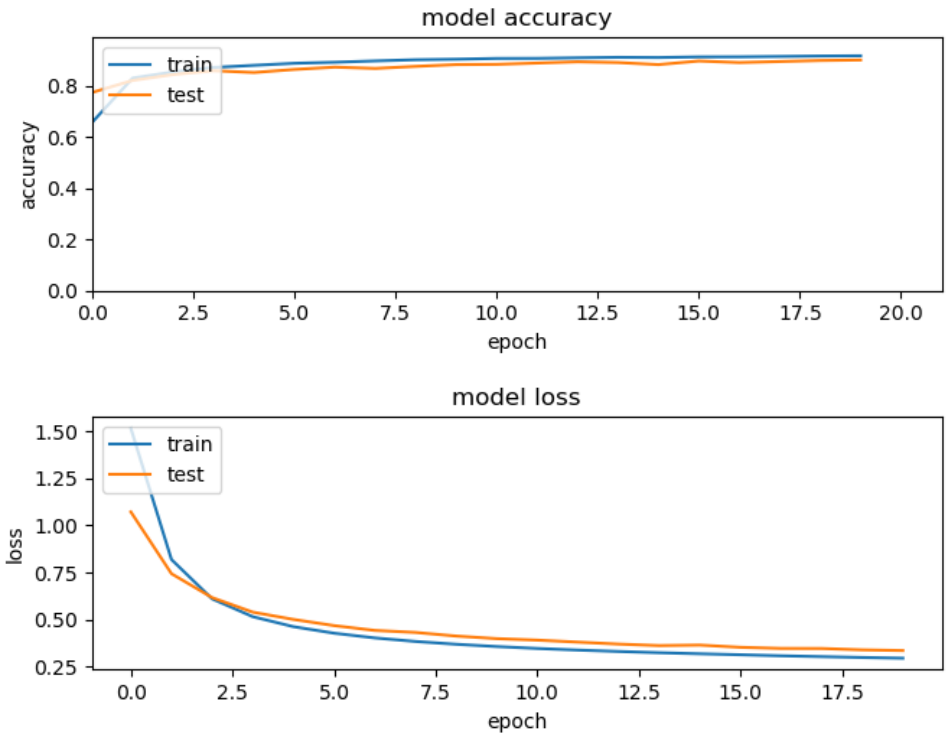

FIGURE 4.2: Keras results for 3 layer CNN

Running the same network on the NumPy based implementation, the results are the following:

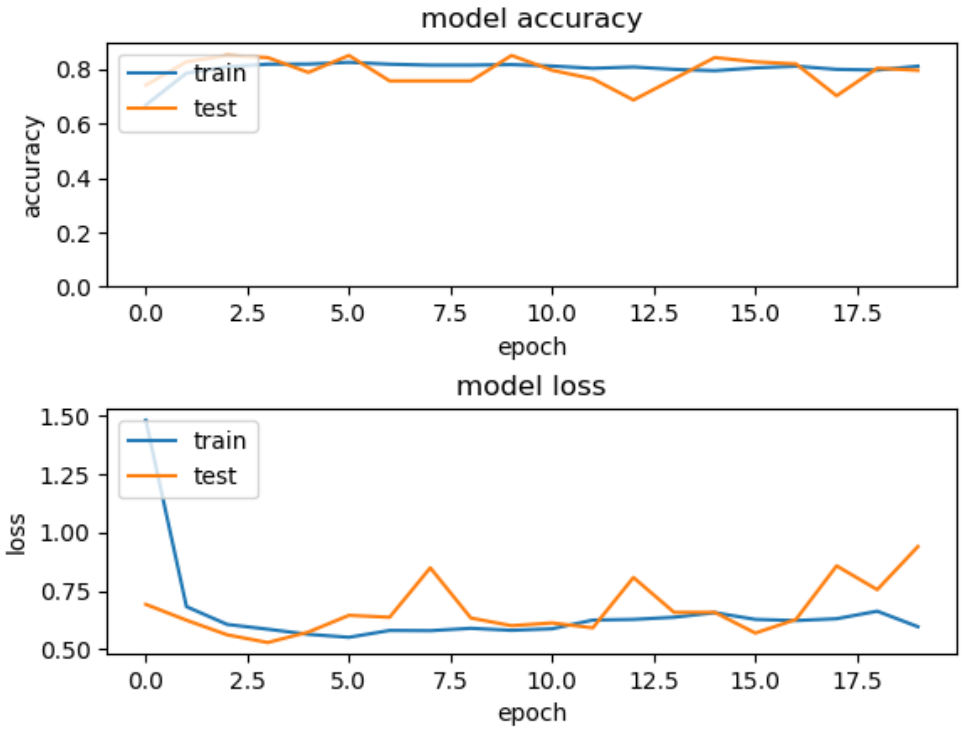

FIGURE 4.3: NumPy results for 3 layer CNN

Furthermore, some smaller examples were executed on the NumPy to compare the precision point results. By default, NumPy is using 64-bit float numbers and it can support down to 16 bits. The network that was used for these experiments was identical to the one that was described previously, but with the exception that the batch size was 100 and the number of training samples 2000 and test samples 500. The following table represents a review of the results:

Precision Used	Time	1st Epoch Acc	1st Epoch Loss	5th Epoch Acc	5th Epoch Loss	10th Epoch Acc	10th Epoch Loss
fw32bw32	61.89 min	42%	4.89	78%	0.70	82%	0.58
fw16bw32	62.27 min	40%	4.46	77%	0.73	84%	0.54
fw16bw16	57.02 min	45%	4.00	72%	0.67	83%	0.51

TABLE 4.1: Precision Testing Accuracy and Loss

4.2 C Implementation

The main goal of this thesis is to develop a hardware acceleration architecture for a CNN. Although python is a good language to develop software thanks

to its various libraries, especially on Machine Learning, it is not suitable for hardware development. Some of the reasons are execution speed, power, and memory consumption, and not being suitable for low-level programming. C/C++ is usually the language of choice when it comes to embedded systems.

For this reason, we created our own C library for running Convolutional Neural Network training that consist of Convolution, Maxpool, Fully-Connected, and Softmax layers. Initially, a small network like the one used on Python implementation will be used, but a slightly larger network the LeNet-4 will be demonstrated so we can further verify the correct operation of our framework. In the following diagram the flow of the process of one batch of data is displayed:

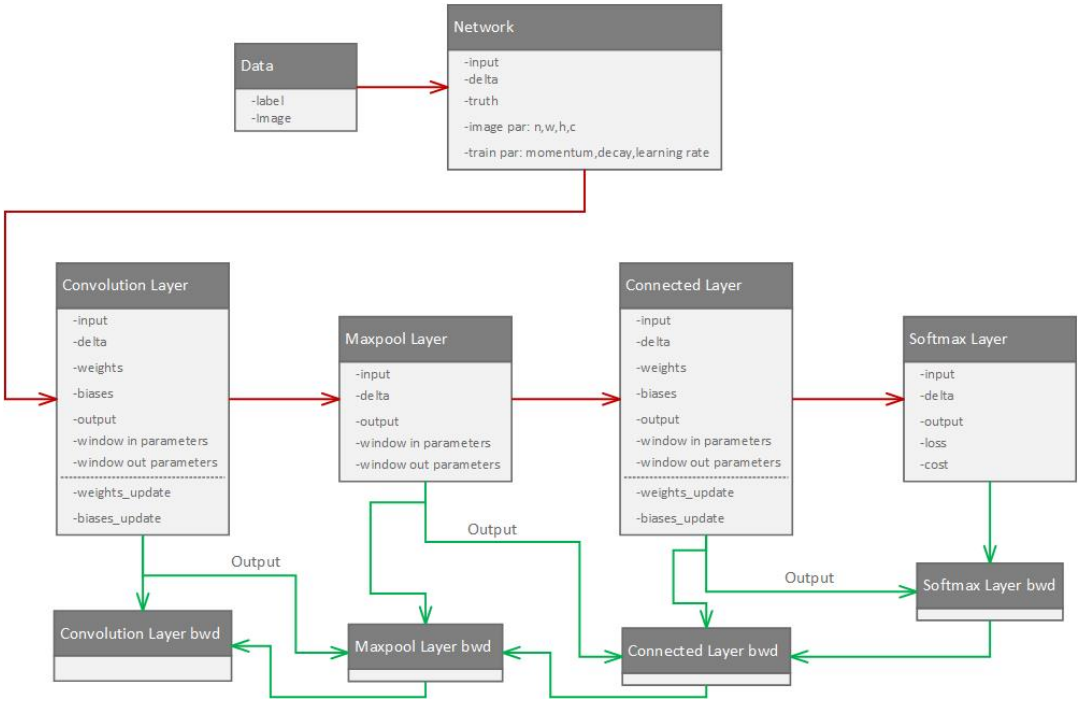

FIGURE 4.4: Data flow between the layers of C based simple CNN

4.2.1 Convolution layer

The initial design of the NumPy code was using the naive implementation for a convolution layer. A convolution is applied to small regions of an image, sampling the values of pixels in this region, and converting it into a single pixel. It is applied to each region of pixels in the image, to produce a new image. The idea is that pixels in the new image incorporate information about

the surrounding pixels, thus reflecting how well a feature is represented in that area.

Unfortunately, this implementation is extremely slow. Matrix multiplication, or matmul, or Generalized Matrix Multiplication (GEMM), is at the heart of deep learning. It's used in fully-connected layers, RNNs, etc., and can be used to implement convolutions too achieving up to 200× speedup. Conv is, after all, a dot-product of the filter with input patches. If we lay out the filter into a 2-D matrix and the input patches in another, then the multiplying these 2 matrices would compute the same dot product. This laying out of the image patches into a matrix is called im2col, for image to column. We rearrange the image into columns of a matrix, so that each column corresponds to one patch where the conv filter is applied.

4.2.2 Results of 3 layer CNN

The first test to validate that the C-based architecture we implemented, was working properly was a batch size comparison test. We used learning rate=0.01, momentum 0.9, decay 0.00005, epoch 20, and iterations 2000. We want to check the accuracy throughout the training process and the speed of the whole training execution.

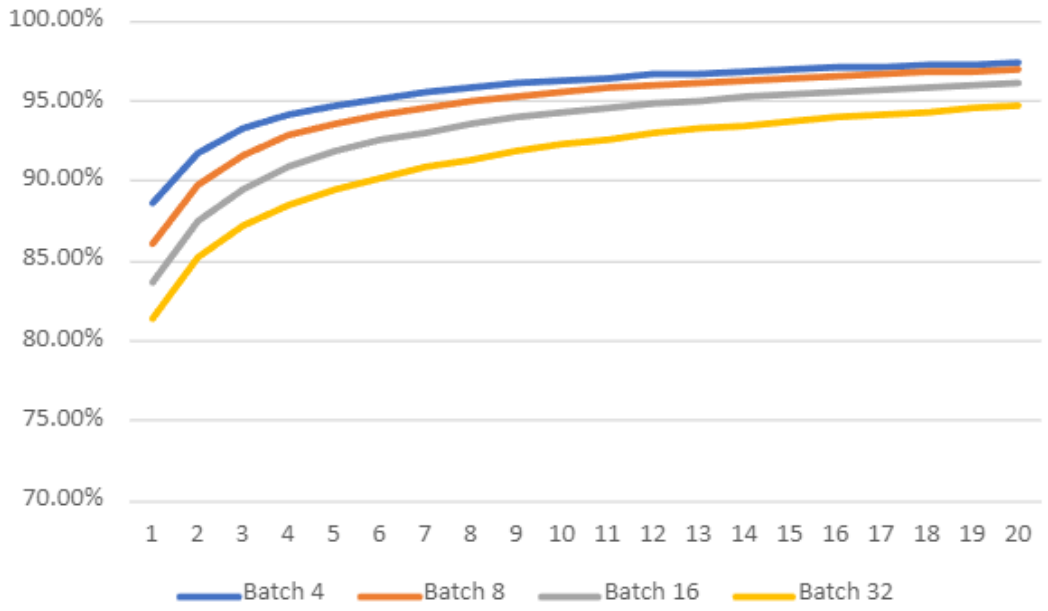

FIGURE 4.5: Batch size accuracy comparison on C-based 3 layer CNN

	Batch 4	Batch 8	Batch 16	Batch 32
Time(s)	682	642	623	615

TABLE 4.2: Batch size execution time comparison on C-based 3 layer CNN

We notice, that even though a higher batch size leads to a better execution time, it's at the expense of better accuracy. For the initial tests, it was decided that getting a higher accuracy was better and we could optimize the speed/accuracy trade-off later, so batch size 4 was used. Due to earlier Python testing was done with batch size 128, we present the equivalent results using batch 4 on the following figures:

FIGURE 4.6: NumPy based 4 batch simple CNN

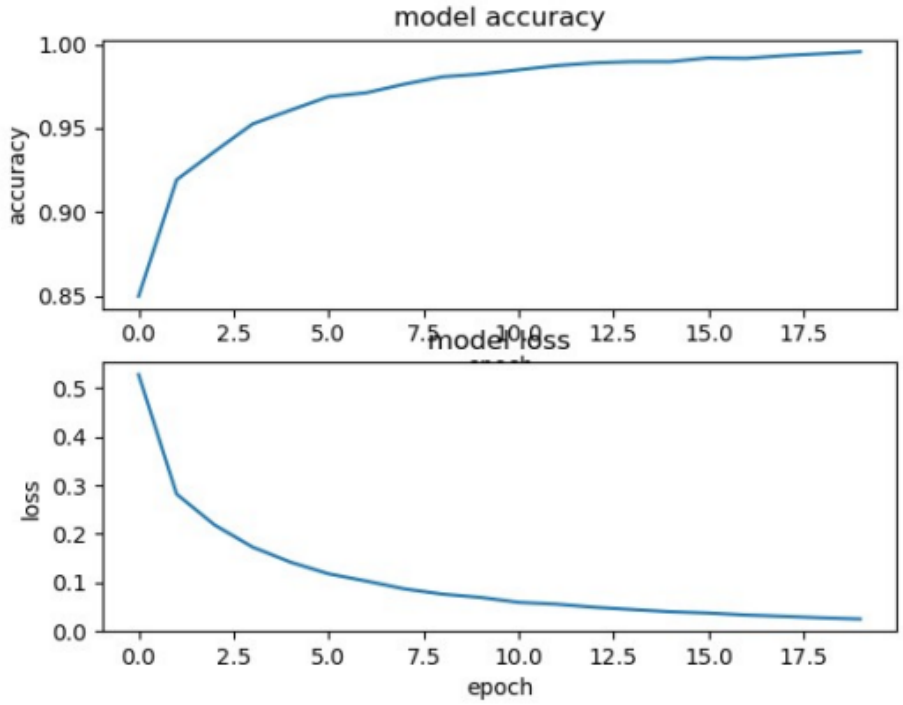

FIGURE 4.7: Keras based batch 4 simple CNN

We notice that the NumPy implementation is not good enough, but it has proven through extended testing that it can achieve close to Keras model results. The major downside of this model as stated earlier though is the execution time, which was 23202 seconds. In comparison the C model executes the same network in 682 seconds.

The results produced by the C code are the following:

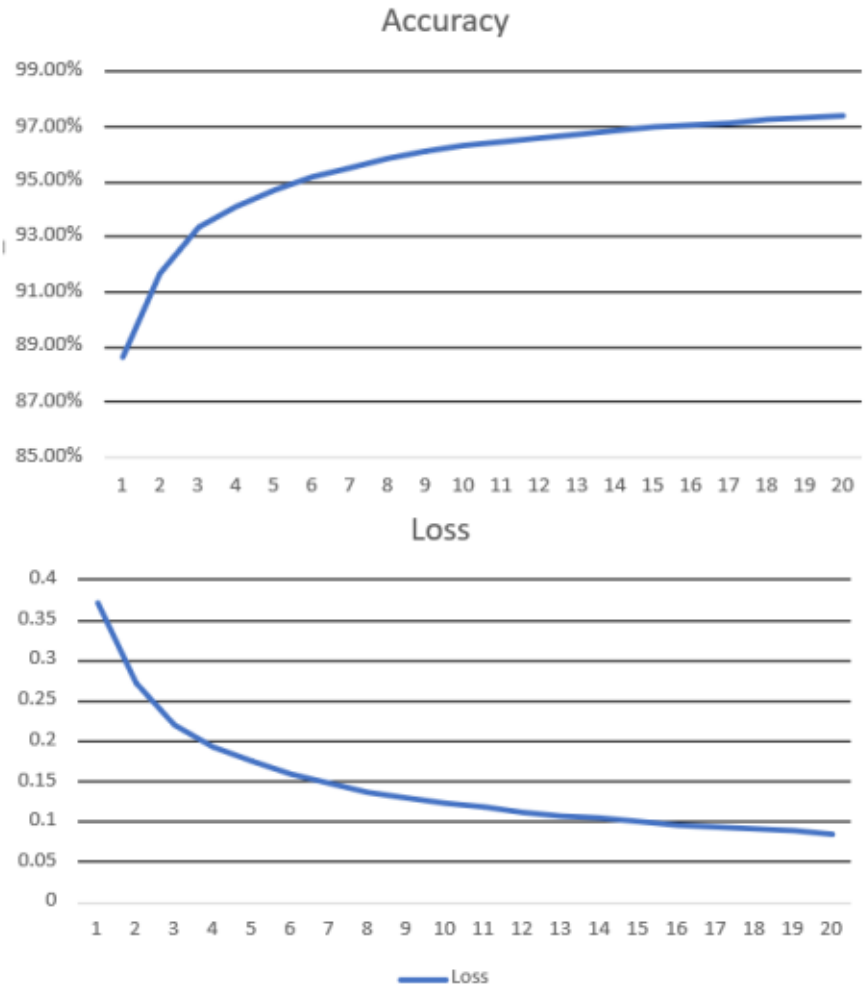

FIGURE 4.8: C based 4 batch simple CNN

It is quite visible that the C model has similar behavior to the one made in Keras. The C code is executed on a single thread and has no parallelization optimizations. This results in Keras having way better execution times. The simple CNN with batch 4 was executed on Keras on 72 seconds.

4.2.3 LeNet-4

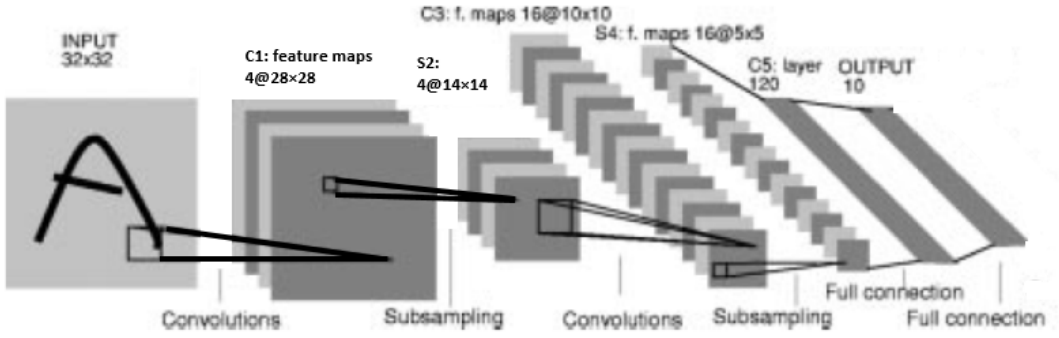

FIGURE 4.9: Presentation of LeNet-4.
Source: https://sh-tsang.medium.com/paper-brief-review-of-lenet-1-lenet-4-lenet-5-boosted-lenet-4-image-classification-1f5f809dbf17

To further test the proper function of the C coded layers that can be used to create various networks, a LeNet-4 [59] architecture has been implemented, although it has some small variations compared to the standard model. The networkcomprises of 6 layers:

1. Convolution Layer with 4 filters sized 5×5, padding=0 and stride=1
2. MaxPooling Layer with size 2×2, padding=0 and stride=2
3. Convolution Layer with 16 filters sized 5×5, padding=0 and stride=1
4. MaxPooling Layer with size 2×2, padding=0 and stride=2
5. Fully Connected Layer with output size of 120
6. Fully Connected Layer with output size of 10
7. SoftMax function for results prediction

This network will have a total of 51,050 trainable parameters and 292,466 connections overall.

In the following figures, the results of running the described network are presented while comparing the C implementation with Keras using no momentum, batch size 4, learning rate=0.01, decay 0.00005, epoch 20, and iterations 2000 :

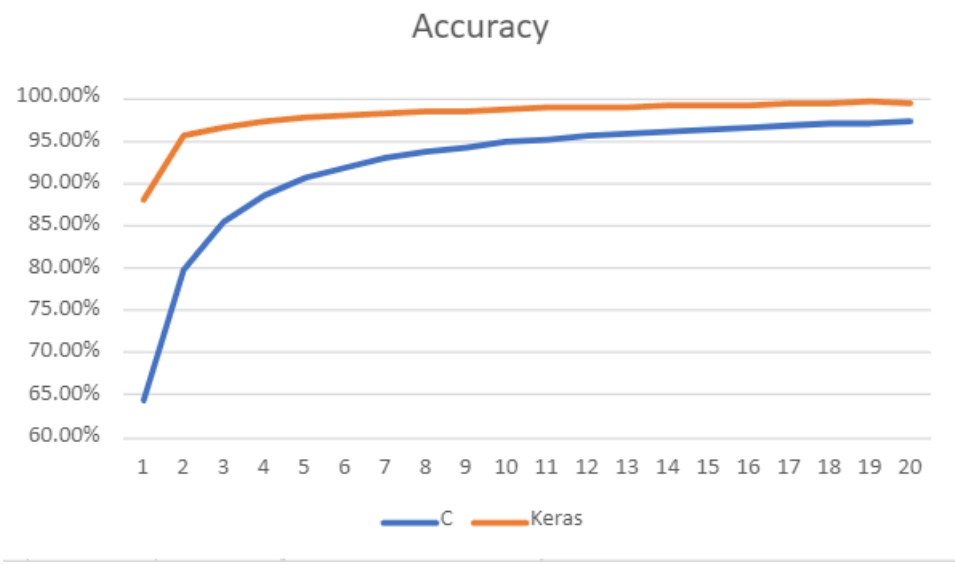

FIGURE 4.10: LeNet-4 accuracy in C and Keras

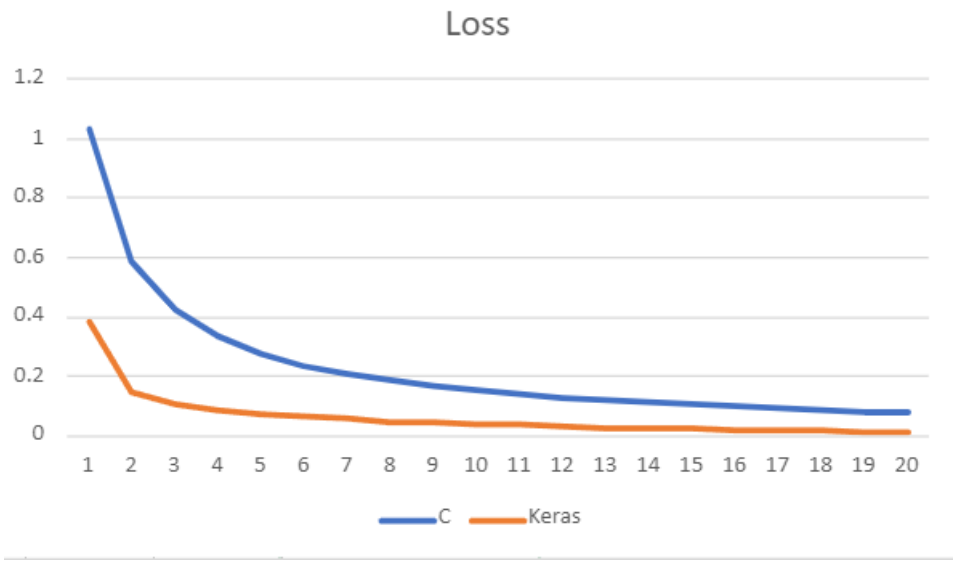

FIGURE 4.11: LeNet-4 loss in C and Keras

The C implementation is following the same behavior as Keras, but it doesn't reach the same endpoint accuracy/loss. Furthermore, the execution time of the C code is 1353 seconds, while Keras runs for 151 sec.

The final test was to run the whole MNIST dataset, which includes 60k pictures, and use momentum=0.9 but only for 10 epochs.

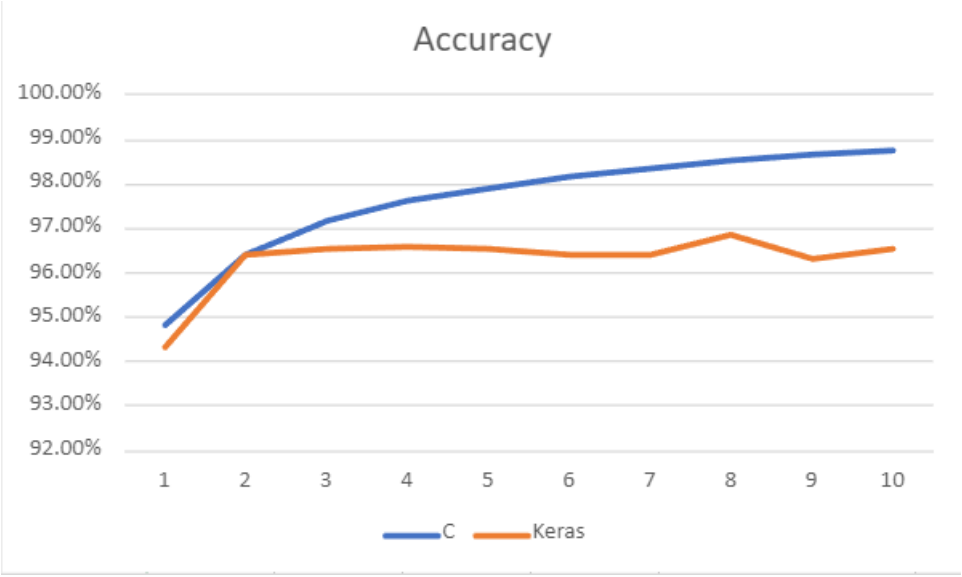

FIGURE 4.12: LeNet-4 with momentum=0.9 accuracy in C and Keras

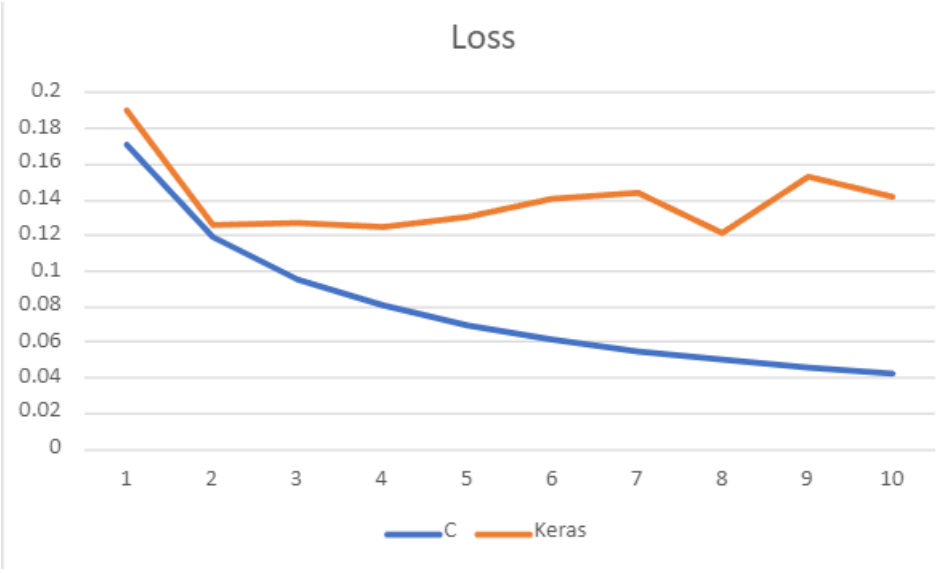

FIGURE 4.13: LeNet-4 with momentum=0.9 loss in C and Keras

While initial observations may suggest peculiar results, a deeper examination reveals that zooming out and considering the overall accuracy trends can provide valuable insights. These discrepancies, which may seem significant at first glance, can often be attributed to the inherent randomness present in the input data. This randomness has the potential to introduce

fluctuations in performance from epoch to epoch, underscoring the dynamic nature of neural network training processes.

4.2.4 Number of filter experiment

The network that is trained consists of a Convolution layer with 3×3 filters initialized with He initialization using Relu as activation function, a Maxpool layer with pool size 2, and a fully connected layer using SoftMax as activation function. To measure the accuracy of the network, the Categorical Cross-Entropy Loss function is used. The following table presents the results of the tests run in CPU for 20 epochs using 200 random images of the MNIST dataset per epoch to determine how the different number of filters used on the convolution layer affect the performance of the network:

N. Filters	Time(sec)	Accuracy
1	4.753	88.27%
2	7.075	88.23%
3	9.168	88.03%
4	11.389	88.57%
5	13.500	88.49%
6	15.881	90.56%
7	18.303	90.74%
8	20.297	90.81%
9	22.873	90.59%
10	24.758	90.65%
11	26.788	90.69%
12	29.344	91.04%

TABLE 4.3: Influence of number of filters on 3 Layer CNN

In our investigation of the CNN architecture, we explored the impact of varying the number of filters on both execution time and accuracy. Starting with an initial configuration of 12 filters and 40 epochs, we achieved a commendable 91.2% accuracy in just 18.23 seconds. Intriguingly, when we reduced the number of filters to 1, the execution time dropped significantly to 9 seconds, accompanied by a minimal decrease in accuracy to 91.2%. This reduction in the number of convolutional weights from 108 to 9 highlighted the efficiency gained through this simplification. Further experimentation revealed that increasing the epoch count to 66 with a single filter led to a slightly improved

accuracy of 92.05%, albeit with a longer execution time of 29.5 seconds. However, the introduction of 6 filters proved to be a pivotal adjustment, achieving an impressive 93.29% accuracy in 29.09 seconds after just 19 epochs. Notably, the 6-filter network surpassed the accuracy of the original 12-filter configuration after only 11 epochs, taking 16.8 seconds. This outcome not only underscores the efficacy of a reduced filter set but also emphasizes the importance of thoughtful parameter tuning in optimizing CNN architectures for both accuracy and efficiency.

Comparing MNIST and CIFAR –10

In the process of selecting the optimal neural network for implementation on an FPGA, a conclusive evaluation was conducted. This assessment involved a comprehensive examination of the network's performance across various filter quantities, utilizing two distinct datasets—MNIST, which is the primary focus of our research, and CIFAR-10. The experiment was executed over 10 epochs, with each epoch comprising 10,000 randomly selected images from both datasets. The obtained results are summarized as follows:

N. Filters	MNIST Time	MNIST Acc	CIFAR-10 Time	CIFAR-10 Acc
1	2 min	94.45%	5 min 48 sec	35.18%
6	6 min 51 sec	99.47%	15 min 36 sec	55.08%
12	12 min 15 sec	99.53%	26 min 14 sec	58.92%

TABLE 4.4: Comparing MNIST and CIFAR-10 on 3 Layer CNN

The 6-filter network demonstrates performance parity with the 12-filter network on the MNIST dataset, achieving close performance on the CIFAR-10 dataset, all the while significantly reducing the number of parameters. Its inherent advantages lie in the improved potential for parallelization compared to the 1-filter network. Simultaneously, it addresses memory constraints more efficiently, achieving performance levels close to the 12-filter network.

Although the 6-filter network appears to be the best choice that ensures a balanced trade-off between computational efficiency and memory utilization in the context of hardware implementation, we decided to work with the 12-filter network. This decision was a result of the fact that our network has a very simple topology, and if we wanted to guarantee that larger and more

complex networks can actually be trained with FPGA, we had to make sure that a larger number of parameters would not be a limiting factor.

Chapter 5

FPGA Design

We developed a versatile library comprising various IP modules that can be flexibly combined to construct diverse CNN models. These modules underwent rigorous post-synthesis testing to ensure their proper functionality. By verifying each module's performance, we established a robust foundation for assembling customized CNN architectures tailored to specific requirements within the FPGA ecosystem.

5.1 Tools Used

By leveraging the capabilities of Vitis HLS, Vivado, and Vitis, this thesis aims to achieve significant performance improvements in the training of convolutional neural networks. The seamless integration of these tools provides a robust framework for accelerating CNN training algorithms on reconfigurable logic platforms, such as FPGAs, thereby addressing the computational demands of modern deep learning applications

5.1.1 Vitis High Level Synthesis (HLS)

Vitis High-Level Synthesis (HLS)[60] is a tool provided by Xilinx, which enables designers to develop high-performance digital hardware designs using C, C++, or OpenCL. Vitis HLS allows for a higher abstraction level in the design flow, abstracting away the low-level details of hardware implementation. It offers automated optimizations and a straightforward transition from software to hardware implementation, making it an ideal tool for accelerating components of the CNN training algorithms.

5.1.2 Vivado IDE

Vivado[61] is a comprehensive design environment for FPGA-based system development. It provides a range of tools for hardware description language (HDL) design, synthesis, simulation, implementation, and verification. Vivado offers advanced optimizations and supports high-level synthesis integration, allowing efficient utilization of FPGA resources and faster development cycles. It serves as a crucial tool for transforming the CNN training components developed in Vitis HLS into hardware designs.

5.1.3 Vitis

Vitis[62] is a unified software platform that enables accelerated application development on heterogeneous platforms, including FPGAs, CPUs, and GPUs. It integrates with the Vivado design environment and provides a higher-level programming model for FPGA-based acceleration. Vitis simplifies the design and deployment process, allowing developers to leverage the full potential of reconfigurable logic.

5.2 FPGA Platform

5.2.1 Xilinx Zynq UltraScale+ MPSoC

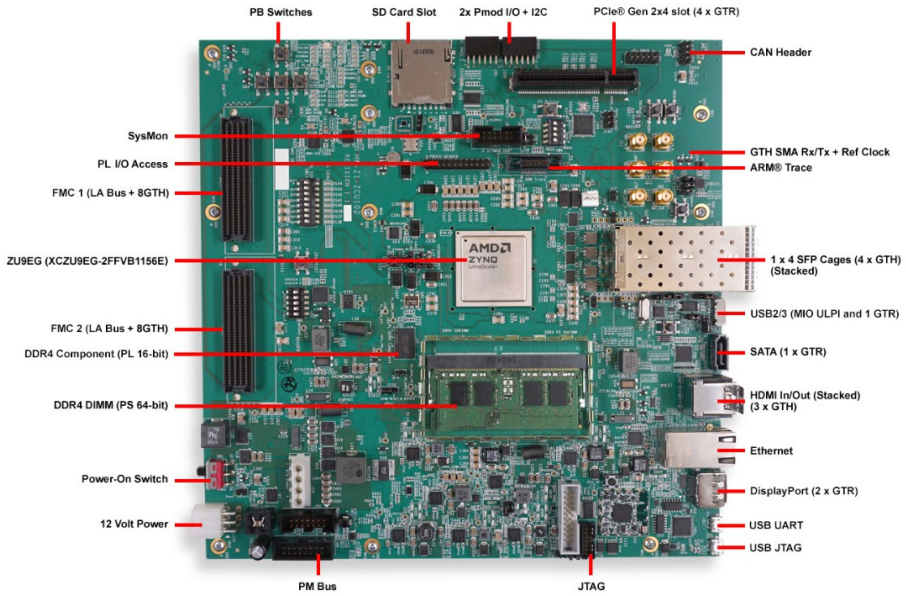

FIGURE 5.1: ZCU102 board preview.
Source: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

The Xilinx Zynq UltraScale+ MPSoC is a family of highly integrated System-on-Chip (SoC) devices that combine powerful processing systems with user-programmable logic, supporting a wide range of applications. It integrates 64-bit quad-core or dual-core Arm Cortex-A53 and dual-core Arm Cortex-R5 based processing systems with Xilinx programmable logic UltraScale architecture, offering high-speed connectivity, advanced security, and built on TSMC's 16FinFET+ process technology. The family is available in different variants, including dual-core, quad-core, and video codec versions, to cater to diverse application requirements [63] [64].

5.2.2 ZCU102 Evaluation Board

The ZCU102 Evaluation Board is a development kit that enables designers to kickstart designs for various applications such as automotive, industrial, video, and communications. It features a Zynq UltraScale+ MPSoC

with a quad-core Arm Cortex-A53, dual-core Cortex-R5F real-time processors, and a Mali-400 MP2 graphics processing unit based on 16nm FinFET+ programmable logic fabric. The board supports all major peripherals and interfaces, making it suitable for a wide range of applications. It provides features such as DDR4 SODIMM, PCIe Root Port, USB3, DisplayPort, SATA, Ethernet, and FPGA Mezzanine Card (FMC) interfaces for I/O expansion, among others [65] [66].

5.3 HLS Design

This section of the thesis focuses on the development of various IP blocks using Vitis HLS, tailored for each stage of the neural network training process. The primary objective is to address the mathematical operations essential to convolutional neural network training, specifically General Matrix Multiply computations, as well as the crucial im2col and col2im processes. By leveraging the capabilities of Vitis HLS, these IPs are designed and optimized to enhance the performance and efficiency of CNN training. They enable accelerated computations and facilitate the extraction of meaningful features from input data. The subsequent sections delve into the detailed design and implementation of these IPs, underscoring their significance in the reconfigurable logic-based acceleration of CNN training

5.3.1 Convolution Layer Forward Propagation

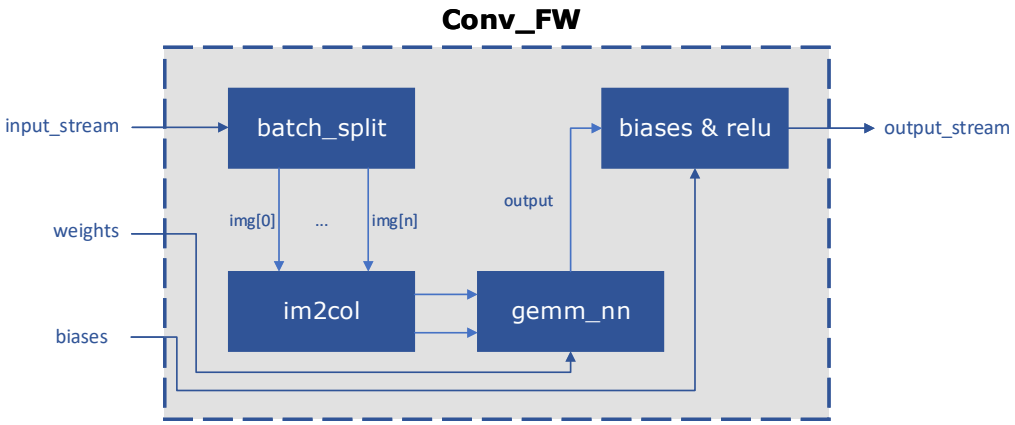

FIGURE 5.2: Schematic of the convolutional layer forward

As noted on the thesis approach (paragraph 3.4), we aim to gain significant advantage through the acceleration of the GEMM functions. Accelerating GEMM operations holds paramount importance in our implementation, as it is a fundamental mathematical operation extensively utilized in CNN training. We will also leverage parallelism as much possible to achieve higher performance on this layer. It is important to note that our implementation takes advantage of flattening 2D arrays into 1D arrays in order to utilize continuous memory access and optimize performance.

GEMM N x N

The GEMM algorithm systematically computes the product of two matrices, namely matrix A (with dimensions M×K) and matrix B (with dimensions K×N), to produce matrix C (with dimensions M×N. Each element of C is obtained by multiplying the corresponding elements of A and B and then summing them across the K dimension. This process is iterated for every element of C, resulting in complete matrix multiplication. In the context of the convolution layer's forward operation and the fully connected layer's backward operation, the GEMM operation is employed to perform matrix multiplications between two regular matrices. To establish a baseline in Vitis HLS, we use the C code from the research presented in paragraph 3.2, as the foundation, with the goal of achieving the highest possible acceleration while minimizing the utilization of system resources.

The highest possible acceleration for gemm_nn is achieved using the following optimizations:

1. Using the pipeline directive on the 2 inner loops
2. Using array partitioning for the data buffers used for the inputs of the gemm function.
3. Swapping the order of inner nested for loops.

In GEMM, three nested loops are typically used: two loops for iterating over the rows and columns of the resulting matrix, and an inner loop for performing the actual multiplication and accumulation of values. The order in which these loops are arranged can affect the memory access patterns and cache utilization, which can impact performance.

The principle of spatial locality suggests that accessing nearby memory locations is faster than accessing distant memory locations. Therefore, keeping

the innermost loop as the one that accesses the matrix elements can potentially benefit from better cache utilization. This is because the elements accessed in a consecutive manner within the inner loop are more likely to reside in the cache, resulting in faster access times.

Image to Column(im2col)

The im2col function is a fundamental technique used in CNNs to efficiently transform image data for the convolution operation. Given an input image, im2col converts it into a matrix where each column corresponds to a receptive field (also known as a kernel or filter) in the original image. This transformation allows for the use of matrix multiplication, which is computationally more efficient than directly applying convolutional filters on the input.

During the im2col process, the image is divided into overlapping patches, and each patch is then flattened into a column in the output matrix. The size of the patches and the degree of overlap are determined by the convolutional filter's dimensions and stride.

By utilizing im2col, CNNs can take advantage of highly optimized matrix multiplication libraries and parallel processing capabilities, leading to significant speed-ups during forward and backward passes in training and inference. This function, like GEMM, consists of three nested loops. After extensive testing, the highest acceleration is achieved by using the pipeline directive, on the innermost loop and the loop flattening directive on the two outer loops.

Conv FW

In the forward propagation process, the results obtained for the `gemm_nn` and the `im2col` provide a significant boost. Additionally, an AXI Stream interface is employed for the input and output variables, and the pipeline directive is utilized to achieve batch-level processing.

Usually, the forward propagation of a convolution layer would appear as follows:

Algorithm 2 Convolution Forward Algorithm

```
function convolution_forward(input, weights, biases, output):
    for i = 0 to batch - 1:
        b = workspace
        c = output + i * N * M
        im = input + i * inputSize

        if ksize == 1:
            b = im
        else:
            im2col(im, b)

        gemm_nn(weights, b, c)

    addBiases(output, biases)
    applyRelu(output)
```

Following substantial refinements of the design after experimentation, we have achieved a significant acceleration by combining the following three processes

1. addition of the three biases
2. application of the activation function
3. writing into the output stream

With the use of the pipeline directive, the convolution forward algorithm transforms into a faster and more efficient process.

It is also important to note that we took advantage of the ternary operator because it performs more efficiently in the Vitis HLS environment, offering additional speed compared to classic if statements.

Algorithm 3 HLS Convolution Forward Algorithm

```
function HLS_conv_FW(input, weights, biases, output):
    for i = 0 to batch - 1:
    #PRAGMA PIPELINE
        stream_image(image)
        if ksize == 1:
            b = image
        else:
            im2col(image, b)

        gemm_nn(weights, b, outp)
    j=0
    k=0
    for i = 0 to batchedOutput - 1:
    #PRAGMA pipeline
        tmp = outp[i] + biases[i]
        output.write((tmp > 0) ? tmp : 0.1*tmp)
        (j == windowSize) ? (j = 0, k++) : j++
        (k == NFilters) ? (k = 0) : k
```

5.3.2 Maxpool Layer Forward Propagation

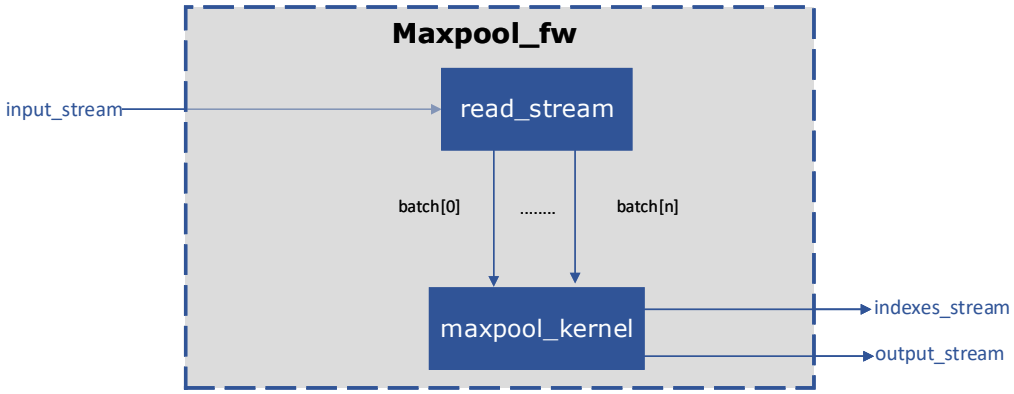

FIGURE 5.3: Schematic of the maxpool layer forward

Although this function is simple and does not involve complex calculations, we can leverage the six nested for loops. Implementing an AXI Stream interface on the input, output, and indexes variables, the input stream is read into a local array using a pipelined for loop. We capitalized on the sequential calculation of output and indexes to achieve acceleration through the implementation of the AXI Stream. Additionally, we utilized the pipeline directive

on the width loop and the loop flattening directive on the batch, channel, and height loops.

5.3.3 Fully Connected Layer Forward propagation

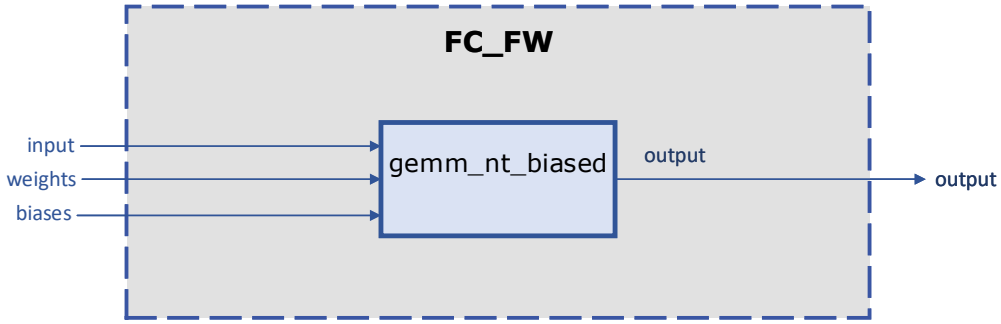

FIGURE 5.4: Schematic of the FC layer forward

GEMM N x T

The GEMM algorithm is also utilized for matrix multiplications between a regular matrix A (with dimensions M×K) and its transpose, matrix B^T (with dimensions N×K). When matrix A and its transposed counterpart B^T are multiplied using GEMM, the resulting matrix C has dimensions M×N. Each element of C is obtained by multiplying the corresponding elements of A and B^T and then summing them across the common dimension K. This process is repeated for every element of C, resulting in a comprehensive matrix multiplication. This operation finds significance in various computational tasks, particularly in the context of the convolution layer's backward operation and the fully connected layer's forward operation.

Working on the baseline design, the highest possible acceleration is achieved using the following optimizations:

1. Swapping the order of inner nested for loops.
2. Using the pipeline directive on the second loop.
3. Using the loop unrolling directive on the innermost loop.

FC FW

Following the same methodology as convolutional layer, we are going to leverage the results we achieved at the `gemm_nt` function. Furthermore, we found some significant speed-up by combining the bias addition operation into the `gemm_nt` function, essentially creating a `gemm_nt_bias` function and eliminated an unnecessary bottleneck.

5.3.4 Softmax Forward propagation

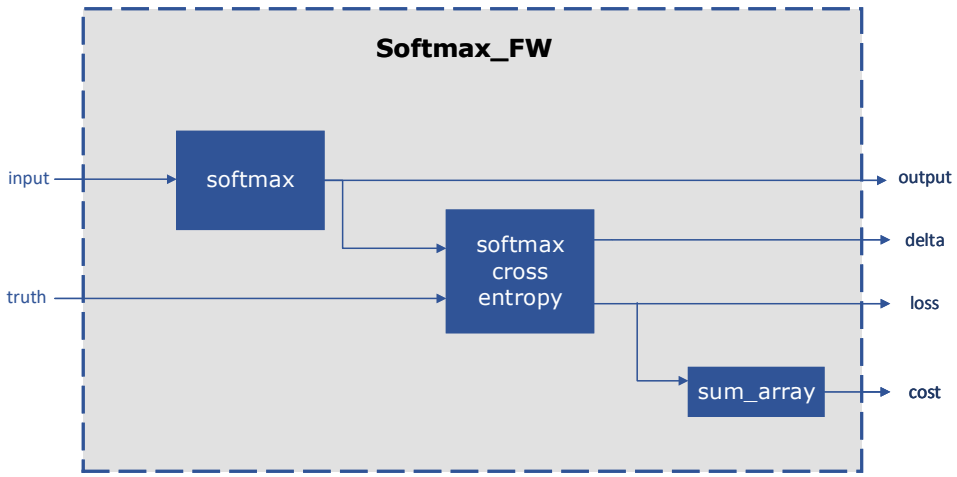

FIGURE 5.5: Schematic of the softmax layer forward

The softmax layer, much like the maxpool layer, is characterized by its limited computational complexity and minimal data manipulation. Consequently, it boasts computational efficiency and swift execution. However, there exists some potential to further enhance its speed through the exploration of techniques that can confer a competitive advantage to the overall network performance.

The forward propagation of the softmax layer consists of 2 parts:

1. calculation of the softmax output for each image of the batch
2. application of the loss function

For the first part, we utilized the pipeline directive to achieve batch-level parallelism, while applying the loop unrolling directive to the function that calculates the output for each image. For the second part, we employed the pipeline directive.

5.3.5 Softmax Backward propagation

FIGURE 5.6: Schematic of the softmax layer backward

The backward propagation of the softmax layer is a simple for loop that is getting a small acceleration thanks to the pipeline directive.

5.3.6 Fully Connected Layer Backward propagation

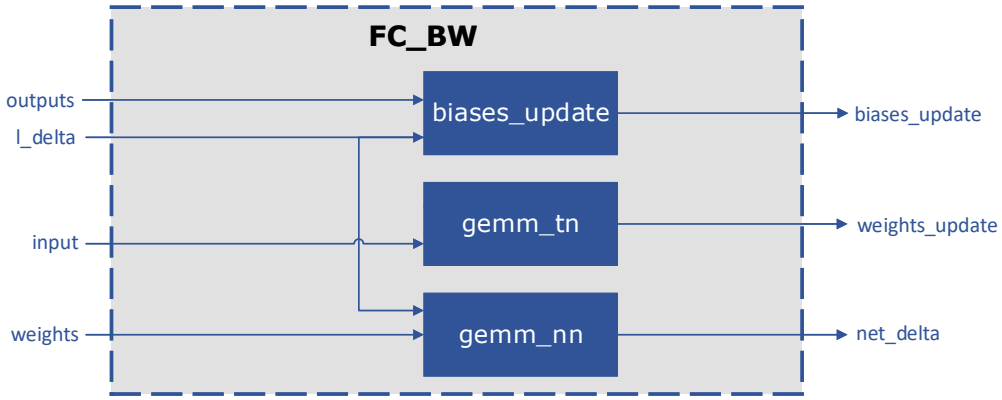

FIGURE 5.7: Schematic of the FC layer backward

Working the same way as the forward part of the FC layer, we gonna use the `gemm_nn` and `gemm_tn` function, alongside with `biases_updates` the optimal execution of this function.

GEMM T x N

The GEMM Transposed x Normal algorithm is also effectively employed for matrix multiplications between a transposed matrix A^T (with dimensions K×M) and a regular matrix B (with dimensions K×N). When the transposed

matrix A^T and matrix B are multiplied using GEMM, the resulting matrix C has dimensions M×N. Each element of C is obtained by multiplying the corresponding elements of A^T and B and then summing them across the common dimension K. This process is repeated for every element of C, leading to a comprehensive matrix multiplication. Such an operation holds particular relevance in various computational tasks, including both convolution layer's and and the fully connected layer's backward operations in machine learning and neural networks.

In the case of this functions we were able to use the exact same optimizations as `gemm_nn` :

1. Array partitioning on the input arrays A and B
2. Swap the order of two inner nested for loops
3. Use the pipeline directive at the second of the three nested loops

FC BW

After employing the accelerated versions of the `gemm_tn` and `gemm_nn` functions, and gaining additional acceleration in the biases update process through the use of the pipeline and loop flattening directives, we arrive at the final version of the fully connected backward propagation function.

5.3.7 Maxpool Backward propagation

FIGURE 5.8: Schematic of the maxpool layer backward

This function is quite simple, with only one for loop being executed. We are applying the pipeline directive to this function to achieve a small boost in performance, but there was no better optimization we could attain.

5.3.8 Convolution Layer Backward

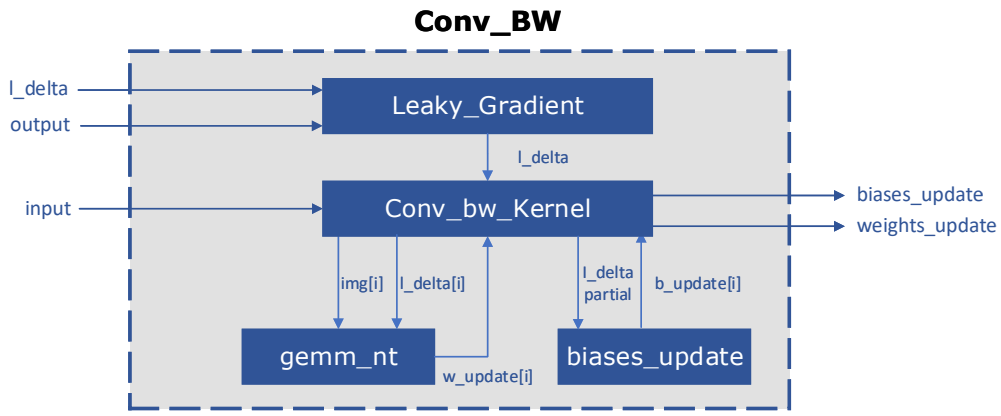

FIGURE 5.9: Schematic of the convolutional layer backward

For the backward process, we leverage the work done for the `gemm_nn`, `gemm_nt` and `im2col` functions. Additionally, we pipeline the for loops used for the calculation of the activation function gradient and achieve batch-level parallelism on weights and biases calculation. In the `biases_update` calculation, we apply the loop flattening directive on the outer loop as well as the pipeline directive in the inner loop.

5.3.9 Weights and Biases Update

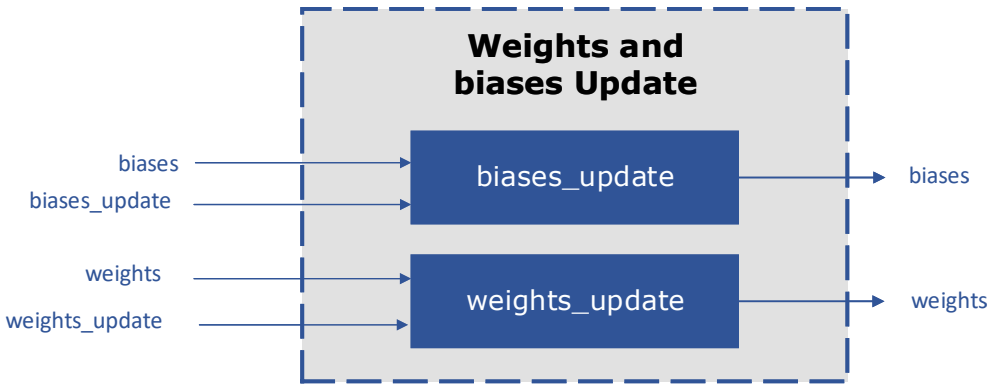

FIGURE 5.10: Schematic of weights and biases updates

We generate a unique IP for each layer's weights and biases update operation, as this process can be parallelly executed with other processes of the CNN as long as the backward pass of the associated layer is completed. The weights and biases update process is straightforward, involving three AXPY operations:

1. An AXPY operation for the application of biases update.
2. An AXPY operation for applying decay on the weights update variable.
3. Another AXPY operation for the application of weights update.

The term "AXPY" stands for "a times x plus y," and it refers to a common operation in linear algebra and numerical computing. In mathematical notation, the AXPY operation is often expressed as $y = \alpha \cdot x + y$

Unfortunately, this operation can be slow due to increased I/O, and despite pipelining the simple for loop, the achieved speed-up is not sufficient.

To address this, we took advantage of the fact that the two AXPY operations associated with weight updates have the same size for loop applications accessing the exact same elements. These two operations were combined into one using the following algorithm:

Algorithm 4 AXPY operation

```
function AXPY(N, ALPHA, *X, INCX, *Y, INCY):
    for i = 0 to N - 1:
        Y[i*INCY] += ALPHA * X[i*INCX]
```

To apply decay on weights updates we use the following function call:

```
AXPY(NWeights, -decay*batch, weights, 1, weights_update, 1)
```

To update the weights we call the following function:

```
AXPY(NWeights, L_r/batch, weights_update, 1, weights, 1)
```

The mathematical operations demonstrate how the two functions were combined into one:

$$weights[i] = weights[i] + \frac{L\_r}{batch} * weight\_updates[i] \tag{5.1}$$

$$weight\_updates[i] = weight\_updates[i] - decay * batch * weights[i] \tag{5.2}$$

Combining **5.1** and **5.2** :

$$\begin{aligned} weights[i] &= weights[i] + \frac{L\_r}{batch} * (weight\_updates[i] - decay * batch * weights[i]) \\ weights[i] &= weights[i] - L\_r * decay * weights[i] + \frac{L\_r}{batch} * weight\_updates[i] \end{aligned} \tag{5.3}$$

$$weights[i] = (1 - L\_r * decay) * weights[i] + \frac{L\_r}{batch} * weight\_updates[i] \tag{5.4}$$

Using the equation **5.4** we achieved a significant acceleration on an operation that is essentially limited by I/O.

5.3.10 Combined Top Module

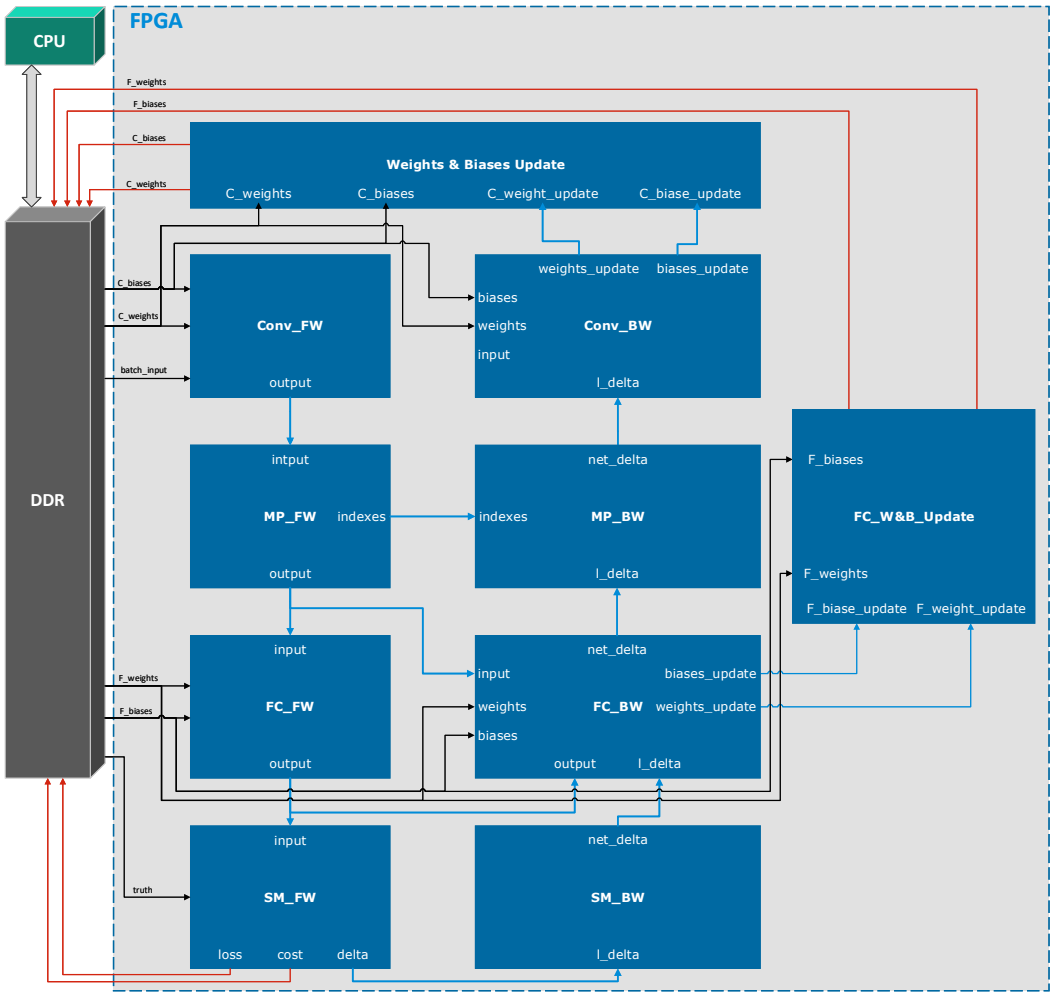

FIGURE 5.11: 3 Layer CNN combining the IPs in a Top Module

Figure 5.11 illustrates the coherent assembly of IPs within a Top Module, showcasing the connectivity and communication with the DDR. This design showcases the collaborative synergy of the individual components in constructing a 3-layer CNN described on chapter 4. This architecture represent a simple example of how someone can utilise our IP modules to train a CNN in FPGAs. We will be using this example in order to evaluate our work and present the results in the next chapter.

This network can be easily configured to be trained with different datasets, and the user can simultaneously use different batch sizes, numbers and sizes of filters, padding sizes, and stride sizes on convolution and max-pool layers, as well as decay and learning rates. It's important to note that the update of the fully-connected layer runs in parallel with the execution of the max-pool backward and convolution backward, resulting in a minor speedup compared to running it at the end of the backward propagation of the whole network alongside the update of the convolution layer.

As previously mentioned, this serves as an example network to demonstrate the functionality and synergy between our IP modules. Users are encouraged to leverage these modules to construct networks tailored to their preferences, ensuring they do not exceed the resource constraints of the target FPGA. Additionally, users have the flexibility to integrate elements of our work with their own, thereby enhancing performance in specific scenarios.

Chapter 6

Results

In Chapter 6, our evaluation of the hardware acceleration architecture for Convolutional Neural Network training focuses on performance metrics and outcomes. We analyze power consumption, latency, throughput, and resource utilization to demonstrate efficiency gains achieved through design optimizations. Although we don't directly compare accuracy and loss between platforms due to the FPGA design's C code similarity with the CPU design showcased on chapter 4, extensive testing through simulations confirms that accuracy and loss are almost the same on both platforms.

6.1 Specification of Compared Platforms

In order to evaluate the proposed FPGA architecture, comparisons were made with CPU and GPU platforms. The CPU used was the Intel® Core™ i7-11800H [67], and the GPU was the NVIDIA GeForce RTX 3060 Mobile [68].

6.1.1 Intel® Core™ i7-11800H

The Intel® Core™ i7-11800H is a high-end octa-core processor designed for gaming laptops and mobile workstations, making it suitable for training convolutional neural networks due to its high performance and multi-core architecture. This capability enables it to handle the intensive computational tasks involved in training CNN models. Its specifications are presented in the following table :

Cores / Threads	8/16
Max Turbo Frequency	4.60 GHz
TDP	45 W
Max Memory Bandwidth	51.2 GB/s
Lithography	10 nm

TABLE 6.1: Intel® Core™ i7-11800H specifications

6.1.2 NVIDIA GeForce RTX 3060 Mobile

The NVIDIA GeForce RTX 3060 Mobile is a graphics card designed for laptops and is suitable for deep learning, including training convolutional neural networks, although it may not offer the same level of performance as higher-end GPUs. The RTX 3060 Mobile features a significant number of CUDA cores and GDDR6 memory, capable of handling GPU-intensive tasks, including deep learning training with frameworks like TensorFlow and PyTorch. Its specifications are presented in the following table :

CUDA Cores	3840
Tensor Cores	120
GPU Memory	6 GB GDDR6
Clock Frequency	1425 MHz
Memory Bandwidth	336 GB/s
Power Consumption	115W

TABLE 6.2: NVIDIA GeForce RTX 3060 Mobile specifications

6.1.3 Proposed Platform

This work's proposed platform focuses on implementing a library of IPs for each part of the convolutional neural network. In order to properly evaluate our design, we combined everything into one single IP to compare the results with the other two platforms.

The combined IP focuses on implementing the 3-layer CNN training model previously described in chapter 4 of this thesis. The network contains a convolution, a max pool, and a fully-connected layer, with 108 weights and 12 biases at the convolution layer, while the fully connected layer has 94,080 weights and 10 biases. This network configuration performs close to the LeNet 4 performance in accuracy and loss, has a large enough number of parameters to make it complex enough, so we will secure we do not face I/O

bottlenecks, especially when dealing with layers containing a high number of trainable parameters.

The resource utilization for implementing the aforementioned configuration of the proposed platform is depicted in the following table:

Clock Frequency (MHz)	300
BRAM Usage	41%
DSP Usage	5%
FF Usage	6%
LUT Usage	16%

TABLE 6.3: Proposed platform resource allocation

6.2 Performance Metrics

6.2.1 Power Consumption

In the context of evaluating the energy efficiency of hardware platforms, power consumption refers to the amount of electrical power utilized by a given hardware device or platform during its operation. It is a pivotal metric because it directly impacts the operational costs and overall sustainability of the hardware. Power consumption is typically measured in watts (W) and encompasses both dynamic and static power components.

Efficiently managing power consumption is crucial for designing energy-efficient hardware platforms, as it directly affects factors such as battery life (in portable devices), electricity bills, and the environmental impact of the hardware's operation. In the broader scope of energy efficiency evaluation, assessing power consumption helps researchers and engineers make informed decisions about hardware selection, considering the trade-offs between performance and energy conservation.

6.2.2 Energy Consumption

Energy consumption is the integral of power consumption over time and provides insights into the overall efficiency of each platform. It is measured in watt-hours (Wh) or joules (J). The total energy consumption is calculated by multiplying the average power consumption by the application runtime:

$$Energy = Power * time \tag{6.1}$$

This metric is crucial for assessing the long-term sustainability and operational costs of each hardware platform.

Two additional important metrics that fall into the Energy Consumption category are the Energy Consumption per Image and Image per Jule, which are calculated by the following equations:

$$Energy\ Consumption\ per\ Image = \frac{Total\ Energy\ Consumed}{Number\ of\ Images\ Processed} \tag{6.2}$$

Total Energy Consumed is the overall energy consumed during the execution of the task (measured in joules),

Number of Images Processed is the total number of images processed during the task.

$$Images\ per\ Joule = \frac{Number\ of\ Images\ Processed}{Total\ Energy\ Consumed} \tag{6.3}$$

These metrics provide insights into the efficiency of your image processing task in terms of energy consumption and throughput.

6.2.3 Latency

Latency measures the time delay between initiating a computational task and receiving the result. In the case of FPGAs, latency is influenced by the hardware design, clock frequency, and the complexity of the implemented algorithm. CPUs and GPUs, on the other hand, exhibit latency influenced by the number of processing cores, cache hierarchy, and memory bandwidth. Latency is calculated as the time elapsed between task initiation and completion, providing insights into the responsiveness of each platform.

Latency speedup, in the context of Amdahl's Law [69], represents the improvement in the overall system latency when a portion of the task is parallelized. The latency speedup ($S_{latency}$) can be calculated by the following:

$$S_{latency} = \frac{L_1}{L_2} \tag{6.4}$$

L_1 is the latency of architecture 1,

L_2 is the latency of architecture 2

6.2.4 Throughput

Throughput represents the number of operations completed per unit of time and is crucial for applications requiring high computational throughput. It is measured in operations per second (OPS) or transactions per second (TPS). For FPGAs, throughput is influenced by parallelism in the hardware design, while CPUs and GPUs rely on the number of cores and parallel execution capabilities. Throughput is calculated as the reciprocal of latency, providing a quantitative measure of computational efficiency.

6.3 Proposed Platform Evaluation

6.3.1 HLS Implementation Evaluation

In order to evaluate the effectiveness of our design choices for the Vitis HLS implementation, we established a baseline version for each IP that runs the naive implementation of each algorithm without any optimizations aiming to increase performance. Afterwards, we present and compare the resources consumed by each IP and its accelerated counterpart, as well as its latency.

Convolution Forward

On the following table we can see the results of the synthesis at Vitis HLS, comparing a non optimised baseline version of the Conv FW IP and the fully optimised version:

	Baseline	Accelerated
Latency(cycles)	4124448	107167
Latency(absolute)	41.244 ms	1.072 ms
BRAM	14	215
DSP	17	81
FF	1496	23821
LUT	1902	16829
URAM	0	0

TABLE 6.4: Conv Forward Resources utilization and Time tables

The baseline function takes 4124449 cycles, while the optimized one 107167 cycles, resulting in a 38.5 times faster execution. This improvement is the

result of combining the work done to improve the `gemm_nn` and the `im2col` functions, deploying an AXI Stream interface for input and output variables, achieving batch-level parallelism, and combining the bias addition, application of the activation function, and writing to the output stream.

Maxpool forward

	Baseline	Accelerated
Latency(cycles)	793065	75283
Latency(absolute)	7.931 ms	0.753 ms
BRAM	0	207
DSP	0	0
FF	319	1102
LUT	699	2088
URAM	0	0

TABLE 6.5: Maxpool Forward Resources utilization and Time tables

We achieve a 10.5 times faster execution than the baseline design. Pipelining has been instrumental in the results of this layer implementation, as well as implementing an AXI Stream interface for the input and output variables.

Fully Connected Forward

	Baseline	Accelerated
Latency(cycles)	3876435	188260
Latency(absolute)	38.764 ms	1.883 ms
BRAM	0	0
DSP	5	10
FF	751	4553
LUT	900	2801
URAM	0	0

TABLE 6.6: FC Forward Resources utilization and Time tables

The result is 20.5 times faster execution, which is achieved mainly by the results gained at the acceleration of the gemm_nt function. This is something that validates our assumption that the biggest gains on our implementation will come from that particular function.

Softmax Forward

	Baseline	Accelerated
Latency(cycles)	1978	178
Latency(absolute)	19.78 µs	1.78 µs
BRAM	0	0
DSP	22	73
FF	1463	8469
LUT	2509	10051
URAM	0	0

TABLE 6.7: Softmax Forward Resources utilization and Time tables

This layer is computationally simple, and at the same time, the number of parameters is extremely low. Despite the operations being sequential and having data dependencies between them, we achieved the highest possible acceleration, with an 11 times faster execution than the baseline, by employing batch-level parallelism and loop unrolling.

Softmax Backward

	Baseline	Accelerated
Latency(cycles)	281	47
Latency(absolute)	2.81 µs	0.47 µs
BRAM	0	0
DSP	2	2
FF	343	412
LUT	309	309
URAM	0	0

TABLE 6.8: Softmax Backward Resources utilization and Time tables

This function does not offer any opportunity for further acceleration, as the number of operations is limited. The 6 times acceleration represents the maximum achievable improvement for this function, achieved through pipelining the operations.

Fully Connected Backward

	Baseline	Accelerated
Latency(cycles)	7131353	283031
Latency(absolute)	71.314 ms	2.830 ms
BRAM	0	0
DSP	5	32
FF	983	14491
LUT	1415	10874
URAM	0	0

TABLE 6.9: FC Backward Resources utilization and Time tables

Deploying the performance gains found in the gemm_nn and gemm_tn functions was crucial for enhancing the performance of the fully connected backward layer. Additionally, optimizations in the biases_update calculation process contributed to a 25 times faster execution time than the baseline model.

Maxpool Backward

	Baseline	Accelerated
Latency(cycles)	301057	37640
Latency(absolute)	3.011 ms	0.376 ms
BRAM	0	0
DSP	2	2
FF	364	450
LUT	902	905
URAM	0	0

TABLE 6.10: MP Backward Resources utilization and Time tables

Unfortunately, due to an I/O bottleneck, this function cannot achieve further performance improvement. The computation intensity being quite low, we achieve the highest possible acceleration through pipelining, resulting in an 8 times faster execution.

Convolution Backward

	Baseline	Accelerated
Latency(cycles)	3625997	377197
Latency(absolute)	36.260 ms	3.772 ms
BRAM	14	14
DSP	5	10
FF	997	3517
LUT	1894	3836
URAM	0	0

TABLE 6.11: Conv Backward Resources utilization and Time tables

The backward process of the convolution layer optimizes computational efficiency by leveraging existing functions such as gemm_nn, gemm_nt, and im2col, while also implementing pipeline and loop flattening directives to enhance parallelism and accelerate calculations, particularly in activation function gradient and biases update computations. The final result is 9.6 times faster.

Weights and Biases Update

	Baseline	Accelerated
Latency(cycles)	2272	125
Latency(absolute)	22.7 µs	1.24 µs
BRAM	0	0
DSP	5	30
FF	581	1982
LUT	672	1738
URAM	0	0

TABLE 6.12: Convolution Weights and Biases Update Resources utilization and Time tables

	Baseline	Accelerated
Latency(cycles)	1881694	94096
Latency(absolute)	18.817 ms	0.941 ms
BRAM	0	0
DSP	5	30
FF	621	2002
LUT	698	1751
URAM	0	0

TABLE 6.13: FC Weights and Biases Update Resources utilization and Time tables

To accelerate the weights and biases update process in CNN layers, unique IPs are generated for parallel execution post-backward pass. Despite employing pipelining and loop optimization, the operation's I/O constraints persist. However, by recognizing the identical size and element access in the two weight update AXPY operations, significant acceleration is achieved, effectively mitigating the I/O limitations. The final product is 20 times faster than the baseline.

6.3.2 CPU and FPGA timings comparisons

A comparison of high importance to be made is between our HLS implementation and the CPU implementation between each IP and it's corresponding function:

Function	FPGA(ms)	i7-11800H(ms)
conv-fw	1.072	2.44
maxpool-fw	0.753	2.231
fc-fw	1.883	2.087
softmax-fw	1.78 µs	9 µs
softmax-bw	0.47 µs	1 µs
fc-bw	2.830	3.968
maxpool-bw	0.376	0.177
conv-bw	3.772	3.177
fc-update	0.941	0.498
conv-update	3.4 µs	1.24 µs
Batch Run	10.686	15.46

TABLE 6.14: FPGA vs CPU

In summary, the FPGA consistently outperforms the CPU across various individual operations and exhibits superior performance in the overall batch run. Notably, the FPGA demonstrates substantial advantages in operations such as softmax forward and backward passes, convolution forward pass, maxpool forward pass, and fully-connected backward pass, showcasing a significant speedup in comparison to the CPU.

While opportunities for further acceleration exist in the convolution backward pass process, it is important to note that the processes involving fully-connected forward pass, maxpool backward pass, and weights updates are constrained by input/output (IO) limitations and, therefore, cannot be subjected to additional acceleration efforts.

6.3.3 Power Consumption Analysis

The average power consumption of our proposed platform can be calculated by Vivado. It generates a report that breaks down the consumption of each PL element. The following figure presents the results:

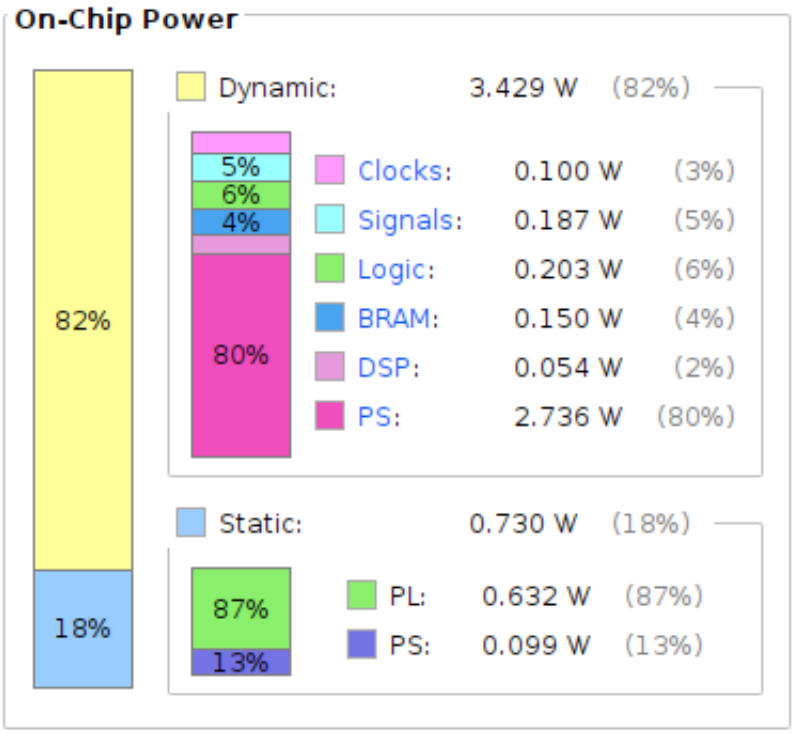

FIGURE 6.1: Power Consumption Estimation by Vivado

As expected, the PL part consumes the most power since it's responsible for the acceleration of the neural network architecture, while the PS only provides the weights and biases initialization and is responsible for feeding the network with batches of data. The Total On-Chip power is measured at 4.16 W.

6.3.4 Resources Allocation

Ensuring scalability was a pivotal aspect of this endeavor, mindful of the resource limitations posed by the ZCU102. To address this, we extended our efforts by implementing two additional architectures in addition to the one previously discussed. The comparison of resource allocation spans across these three architectures:

1. MNIST dataset using 6 filters on Convolutional Layer, resulting into a network with 54 convolution weights and 6 biases and 47040 FC weights and 10 biases

2. MNIST dataset using 12 filters on Convolutional Layer, resulting into a network with 108 convolution weights and 12 biases and 94080 FC weights and 10 biases
3. CIFAR-10 using 12 filters, resulting into a network with 324 convolution weights and 12 biases and 122880 FC weights and 10 biases

The result of this comparisson are presented on the following table:

	MNIST 6	MNIST 12	CIFAR-10 12
BRAM Usage	24%	41%	65%
DSP Usage	5%	5%	10%
FF Usage	6%	6%	9%
LUT Usage	16%	16%	32%

TABLE 6.15: Resource allocation comparison between architectures

The resource analysis of the MNIST architectures underscores a notable increase in BRAM usage when transitioning from 6 to 12 filters, while DSP, FF, and LUT utilization remain relatively stable. Specifically, for the MNIST dataset with 6 filters, BRAM usage is recorded at 24%, and this figure rises to 41% with 12 filters. Despite this rise in BRAM utilization, DSP, FF, and LUT maintain consistent percentages between the two configurations. In contrast, the architecture tailored for the more intricate CIFAR-10 dataset, employing 12 filters, demonstrates heightened resource demands. However, the current resource utilization patterns reveal substantial available resources in DSP, FF, and LUT. This surplus suggests ample capacity for the implementation of even more complex neural network architectures, allowing for the full harnessing of the FPGA's potential.

6.4 Final Performance

The latency speedup, as well as the power and energy efficiencies for every platform are calculated compared to the CPU. Everything is calculated for training one batch of images through the same neural network, on the three different platforms. The network that we compare the 3 platform is the simple 3 layer CNN described on chapter 5.3.10 and we use 12 filters with size 3 on convolution layer and batch size 4, training it with the MNIST dataset.

Metric	CPU	GPU	Proposed Platform
Clock Frequency (MHz)	4600	1425	300
Throughput (Images/sec)	258.7	1333.3	374.32
Latency (ms)	15.46	3	10.686
Latency Speedup	1x	5.1x	1.45x
Total On-Chip Power (Watt)	47	115	4.16
Power Efficiency	1x	0.408x	11.29x
Energy Cons./Image (Joule)	0.182	0.086	0.011
Images/Joule	5.5	11.6	89.98
Energy Efficiency	1x	2.1x	16.55x

TABLE 6.16: Platforms Performance Results Comparison

Some of the results are also presented in the following figure:

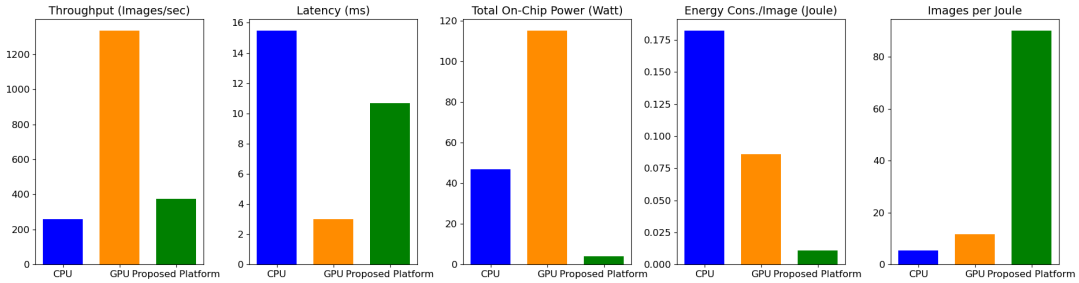

FIGURE 6.2: Results presented in bar charts

The performance metrics paint a compelling picture of the three platforms' capabilities in neural network training. In terms of throughput, the GPU emerges as the leader, showcasing superior efficiency by processing 1333.3 images per second. The Proposed Platform follows with a throughput of 374.32 images per second, outperforming the CPU's rate of 258.7 images per second. Notably, this highlights the GPU's prowess in handling concurrent image processing tasks. However, when it comes to latency, the GPU demonstrates its strength by achieving the lowest response time of 3 milliseconds, underscoring its ability to rapidly process tasks. The Proposed Platform, with a latency of 10.686 milliseconds, and the CPU, with 15.46 milliseconds, follow, indicating their respective processing speeds.

The latency speedup further emphasizes the efficiency gains of the GPU, showcasing a notable 5.1x improvement over the CPU. The Proposed Platform also demonstrates a small 1.45x speedup. Moving to power consumption, the GPU requires the most at 115 watts, while the CPU consumes 47 watts. In stark contrast, the Proposed Platform operates at a remarkably lower power consumption of 4.16 watts. The power efficiency metric solidifies the superiority of the Proposed Platform, boasting an 11.29$\times$ improvement over the CPU and surpassing the GPU which achieves 0.408$\times$. With an energy consumption of only 0.011 Joules per image, the Proposed Platform excels, reaffirming its energy-saving capabilities. Impressively, it achieves 89.98 images per Joule, surpassing both the GPU and CPU. Overall, the Proposed Platform emerges as a promising contender for energy-efficient neural network training, outshining both the CPU and GPU in terms of energy efficiency, achieving a 16.55$\times$ energy efficiency over CPU and 7.75$\times$ over GPU.

The following figure vividly captures the comprehensive comparison between the three platforms, presenting a holistic view of their performance through an insightful scatter plot:

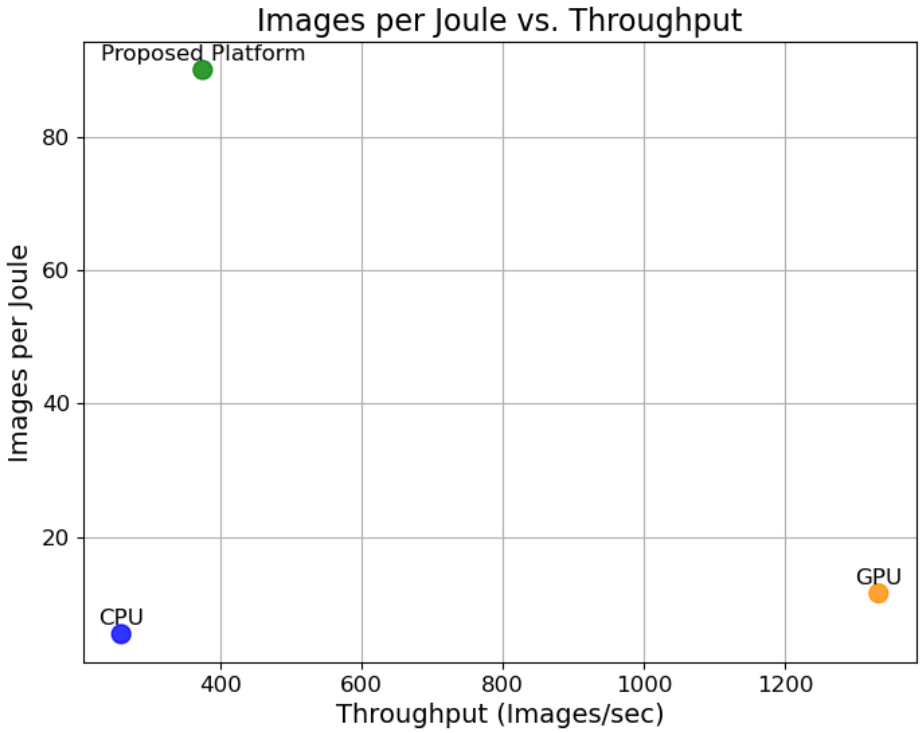

FIGURE 6.3: Relationship between images per Joule and throughput, highlighting the trade-off between efficiency and processing speed.

In summary, the proposed platform stands out as the optimal choice for those prioritizing energy efficiency in neural network training, offering impressive performance metrics with significantly lower power consumption and higher efficiency gains compared to both the GPU and CPU. On the other hand, the GPU emerges as the preferred option for achieving the highest throughput and minimizing latency, making it the go-to choice for tasks demanding rapid and concurrent image processing. In contrast, the CPU lags behind in every scenario, demonstrating inferior performance metrics and making it the least favorable option across the evaluated criteria. Ultimately, the proposed platform and GPU each cater to distinct priorities, with the former excelling in energy efficiency and the latter leading in throughput and latency performance.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This work proposes a novel hardware architecture for training Convolutional Neural Networks on FPGA devices, with the aim of achieving optimal training accuracy while improving energy efficiency and throughput speedup compared to traditional CPU and GPU systems. The research is motivated by the pressing need to address the escalating demands of the contemporary AI and machine learning landscape, where the exponential growth of data and the intricacy of neural network architectures have intensified the computational requirements for training models.

The proposed methodology leverages prior work findings and utilizes GEMM and im2col implementations, as well as batch level parallelism, while balancing the workload between the CPU controller and the FPGA. The FPGA architecture designed on Vitis HLS offers configurability, scalability, accelerated performance, and superior energy efficiency compared to conventional methods executed on CPUs and GPUs. The thesis contributes significantly to hardware acceleration for CNN training by developing a novel C code optimized for hardware acceleration, introducing HLS code for forward and backward propagation, and establishing a systematic workflow for generating extensive HLS test benches.

The FPGA-based architecture developed in this thesis showcases improved throughput over CPU and energy efficiency over both CPU (**x15.16**) and GPU (**x5.1**), making it a promising alternative to traditional hardware platforms for CNN training.

7.2 Future Work

This thesis not only contributes to the current research landscape but also lays the groundwork for future exploration, unveiling numerous opportunities to advance and broaden the scope of FPGA-based acceleration in convolutional neural network training.

Firstly, exploring the potential of combining layers for resource sharing presents an intriguing prospect. Investigating strategies to consolidate layers, such as merging convolutional layers with maxpool layers, can optimize hardware resource utilization. This approach may lead to more efficient and streamlined architectures, enhancing both speed and efficiency in CNN training.

Another avenue for future exploration involves transitioning from floating-point precision to fix-point precision operations. Employing fix-point precision can significantly reduce computational complexity and resource requirements, potentially resulting in accelerated training without compromising model accuracy.

Expanding the research to encompass larger networks, such as the LeNet-4 architecture, is paramount. Scaling up the network size poses unique challenges, including increased computational demands and potential resource bottlenecks. Addressing these challenges will contribute to the development of FPGA-based acceleration solutions that can handle more complex and larger-scale neural network architectures.

Furthermore, delving into advanced weight optimization techniques is crucial for further improving the efficiency of FPGA-based CNN training. Investigating novel algorithms and strategies for optimizing network weights can lead to models with improved convergence rates, accuracy, and reduced hardware resource utilization.

Finally, the implementation and acceleration of additional layers tailored for various network architectures represent a fertile area for future exploration. Adapting FPGA-based acceleration to accommodate diverse layer types beyond the conventional convolutional and pooling layers can extend the applicability of the proposed framework to a broader range of neural network architectures.

References

[1] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3.3 (July 1959), pp. 210–229. DOI: 10.1147/rd.33.0210.

[3] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. "A Fast Learning Algorithm for Deep Belief Nets". In: *Neural Computation* 18.7 (July 2006), pp. 1527–1554. DOI: 10.1162/neco.2006.18.7.1527.

[5] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[7] Kazushige Goto and Robert A. van de Geijn. "Anatomy of high-performance matrix multiplication". In: *ACM Transactions on Mathematical Software* 34.3 (May 2008), pp. 1–25. DOI: 10.1145/1356052.1356053.

[9] Y. Bengio, P. Frasconi, and P. Simard. "The problem of learning long-term dependencies in recurrent networks". In: *IEEE International Conference on Neural Networks*. IEEE. DOI: 10.1109/icnn.1993.298725.

[10] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "Understanding the exploding gradient problem". In: (Nov. 2012). arXiv: 1211.5063 [cs.LG].

[11] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[12] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: (Feb. 2015). arXiv: 1502.01852 [cs.CV].

[13] Xavier Glorot, Antoine Bordes, and Y. Bengio. "Deep Sparse Rectifier Neural Networks". In: vol. 15. Jan. 2010.

[14] CJ Willmott and K Matsuura. "Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance". In: *Climate Research* 30 (2005), pp. 79–82. DOI: 10.3354/cr030079.

[15] Erich L Lehmann and George Casella. *Theory of point estimation*. Springer Science & Business Media, 2006.

[16] Boris Polyak. *Introduction to Optimization*. May 1987.

[17] Bernard Delyon. "Stochastic approximation with decreasing gain : Convergence and asymptotic theory". In: (July 2000).

[18] J. Bilmes et al. "Using PHiPAC to speed error back-propagation learning". In: *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 5. 1997, 4153–4156 vol.5. DOI: 10.1109/ICASSP.1997.604861.

[19] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0.

[21] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[23] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[24] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2009. DOI: 10.1109/cvpr.2009.5206848.

[27] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: (Mar. 2016). arXiv: 1603.04467 [cs.DC].

[29] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: (Dec. 2019). arXiv: 1912.01703 [cs.LG].

[30] François Chollet et al. *Keras*. https://github.com/fchollet/keras. 2015.

[32] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: (June 2014). arXiv: 1408.5093 [cs.CV].

[35] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, June 2017. DOI: 10.1145/3079856.3080246.

[37] Kalin Ovtcharov et al. *Accelerating Deep Convolutional Neural Networks Using Specialized Hardware*. Feb. 2015. URL: https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/.

[38] Sparsh Mittal and Jeffrey S. Vetter. "A Survey of Methods For Analyzing and Improving GPU Energy Efficiency". In: (Apr. 2014). arXiv: 1404.4629 [cs.AR].

[39] Kamel Abdelouahab, Maxime Pelcat, and François Berry. "Accelerating the CNN Inference on FPGAs". In: *Deep Learning in Computer Vision*. CRC Press, Mar. 2020, pp. 1–40. DOI: 10.1201/9781351003827-1.

[40] Fotakis Tzanis. "Analysis and design methodology of convolutional neural networks mapping on reconfigurable logic". en. In: (2020). DOI: 10.26233/HEALLINK.TUC.86843.

[41] Wenlai Zhao et al. "F-CNN: An FPGA-based framework for training Convolutional Neural Networks". In: *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, July 2016. DOI: 10.1109/asap.2016.7760779.

[42] Diederik Adriaan Vink et al. "Caffe Barista: Brewing Caffe with FPGAs in the Training Loop". In: (June 2020). arXiv: 2006.13829 [cs.DC].

[43] Tong Geng et al. "FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters". In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Apr. 2018. DOI: 10.1109/fccm.2018.00021.

[44] Muhammad Junaid et al. "Optimal Architecture of Floating-Point Arithmetic for Neural Network Training Processors". In: *Sensors* 22.3 (Feb. 2022), p. 1230. DOI: 10.3390/s22031230.

[45] Hongbo Zhang et al. "Yolov3-tiny Object Detection SoC Based on FPGA Platform". In: *2021 6th International Conference on Integrated Circuits and Microsystems (ICICM)*. IEEE, Oct. 2021. DOI: 10.1109/icicm54364.2021.9660358.

[46] Cheng Luo et al. "Towards Efficient Deep Neural Network Training by FPGA-Based Batch-Level Parallelism". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Apr. 2019. DOI: 10.1109/fccm.2019.00016.

[47] Shreyas Kolala Venkataramanaiah et al. “Automatic Compiler Based FPGA Accelerator for CNN Training”. In: (Aug. 2019). arXiv: 1908.06724 [cs.LG].

[48] Kim Bjerge, Jonathan Horsted Schougaard, and Daniel Ejnar Larsen. “A scalable and efficient convolutional neural network accelerator using HLS for a System on Chip design”. In: (Apr. 2020). arXiv: 2004.13075 [cs.CV].

[49] Sean Fox et al. “Training Deep Neural Networks in Low-Precision with High Accuracy Using FPGAs”. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, Dec. 2019. DOI: 10.1109/icfpt47387.2019.00009.

[50] Zhiqiang Liu et al. “An FPGA-based processor for training convolutional neural networks”. In: *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, Dec. 2017. DOI: 10.1109/fpt.2017.8280142.

[51] Roberto DiCecco et al. “Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks”. In: (Sept. 2016). arXiv: 1609.09671 [cs.CV].

[52] Ke He et al. “FeCaffe: FPGA-enabled Caffe with OpenCL for Deep Learning Training and Inference on Intel Stratix 10”. In: *FPGA 2020 The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Nov. 18, 2019). DOI: 10.1145/3373087.3375389. arXiv: 1911.08905v1 [cs.DC].

[53] Feixue Tang et al. “Optimization of Convolution Neural Network Algorithm Based on FPGA”. In: *Embedded Systems Technology*. Springer Singapore, 2018, pp. 131–140. ISBN: 9789811310263. DOI: 10.1007/978-981-13-1026-3_10.

[54] Chunhua Xiao et al. “SDST-Accelerating GEMM-based Convolution through Smart Data Stream Transformation”. In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, May 2022. DOI: 10.1109/ccgrid54584.2022.00049.

[55] JiUn Hong et al. “Design of Power-Efficient Training Accelerator for Convolution Neural Networks”. In: *Electronics* 10.7 (Mar. 2021), p. 787. DOI: 10.3390/electronics10070787.

[56] David Gschwend. “ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network”. In: (May 2020). arXiv: 2005.06892 [cs.CV].

[57] Yue Tang et al. “EF-Train: Enable Efficient On-device CNN Training on FPGA through Data Reshaping for Online Adaptation or Personalization”. In: *ACM Transactions on Design Automation of Electronic Systems* 27.5 (Sept. 2022), pp. 1–36. DOI: 10.1145/3505633.

[58] Seungkyu Choi et al. “TrainWare: A Memory Optimized Weight Update Architecture for On-Device Convolutional Neural Network Training”. In: *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, July 2018. DOI: 10.1145/3218603.3218625.

[59] Yann Lecun et al. “Comparison of learning algorithms for handwritten digit recognition”. In: Jan. 1995.

[69] Gene M Amdahl. “Validity of the single pro cessor approach to achieving large scale computing capabilities”. In: 1967. URL: https://inst.eecs.berkeley.edu//~n252/paper/Amdahl.pdf.

External Links

[2] IBM Cloud Education. *What is Machine Learning?* July 2020. URL: https://www.ibm.com/cloud/learn/machine-learning.

[4] *Explained: Neural networks*. URL: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.

[8] *Anatomy of a High-Speed Convolution*. URL: https://sahnimanas.github.io/post/anatomy-of-a-high-performance-convolution/.

[20] *THE MNIST DATABASE of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/.

[22] *The CIFAR-10 dataset*. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[25] *ImageNet*. URL: https://image-net.org/.

[26] *TensorFlow - Website*. URL: https://www.tensorflow.org/.

[28] *PyTorch - Website*. URL: https://pytorch.org/.

[31] *Caffe - Website*. URL: https://caffe.berkeleyvision.org/.

[33] *Caffe Model Zoo*. URL: https://github.com/BVLC/caffe/wiki/Model-Zoo.

[34] *NVIDIA Tesla V100*. URL: https://www.nvidia.com/en-gb/data-center/tesla-v100/.

[36] *Alveo U50 Data Center Accelerator Card*. URL: https://www.xilinx.com/products/boards-and-kits/alveo/u50.html#overview.

[60] *Vitis High-Level Synthesis User Guide*. URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction.

[61] *Vivado Design Suite*. URL: https://docs.xilinx.com/r/en-US/ug910-vivado-getting-started/Vivado-Design-Suite-Overview.

[62] *Vitis Unified Software Platform*. URL: https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/Getting-Started-with-Vitis.

[63] *UltraScale MPSoC Architecture*. URL: https://www.xilinx.com/products/technology/ultrascale-mpsoc.html.

[64] *Zynq™ UltraScale+™ MPSoC*. URL: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[65] *Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit*. URL: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[66] *ZCU102 Evaluation Board User Guide*. URL: https://www.xilinx.com/support/documents/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.

[67] *Intel® Core™ i7-11800H specifications*. URL: https://www.intel.com/content/www/us/en/products/sku/213803/intel-core-i711800h-processor-24m-cache-up-to-4-60-ghz/specifications.html.

[68] *NVIDIA GeForce RTX 3060 Mobile*. URL: https://www.nvidia.com/en-eu/geforce/graphics-cards/30-series/rtx-3060-3060ti/.